Vectors and Objects

Victor Eijkhout, Susan Lindsey

Fall 2024

last formatted: October 10, 2024

1. 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.

2. Create in the constructor

```
class with_vector {
private:
  vector<float> x;
public:
  with_vector( int n ) {
    x = vector<float>(n); };
};
```

Problem: vector gets created twice.

3. 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 n )
    : name(name),
      values(vector<double>(n)) {
  };
};
```

Even shorter:

```
  named_field( string name,int n )
    : name(name),values(n) {
  };
```

Multi-dimensional arrays

4. 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.

5. Matrix class

```
1 // array/matrixclass.cpp
2 class matrix {
3 private:
4   vector<double> the_matrix;
5   int m,n;
6 public:
7   matrix(int m,int n)
8     : m(m),n(n),the_matrix(m*n) {};
```

(Can you combine the *get/set* methods, using **??**?)

Exercise 1

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

Exercise 2

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:

```
1 // array/matrix.cpp
2 A.set(3.);
3 cout << "Sum of elements: "
4      << A.totalsum() << '\n';
```

Output:

```
Sum of elements: 30
```

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

6. Matrix class; better design

Linearized indexing:

Class:

```
1 // array/matrixclass.cpp
2 class matrix {
3 private:
4   vector<double> the_matrix;
5   int m,n;
6 public:
7   matrix(int m,int n)
8     : m(m),n(n),the_matrix(m*n)
      {};
```

Methods:

```
1 void set(int i,int j,double v) {
2   the_matrix.at( i*n +j ) = v;
3 };
4 double get(int i,int j) {
5   return the_matrix.at( i*n +j );
6 };
```

Exercise 3

In the matrix class of the previous slide, why are `m,n` stored explicitly, unlike in the matrix class of section **??**?

Exercise 4

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

Implement matrix-matrix multiplication.

7. 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 5

- Write a class *pascal* so that *pascal*(*n*) is the object containing *n* rows of the above coefficients. Write a method *getvalue*(*i*,*j*) that returns the (i, j) coefficient.
- Write a method *print* that prints the above display.

8. Pascal implementation

The object needs to have an array internally. The easiest solution is to make an array of size $n \times n$.

- Optimize your code to use precisely enough space for the coefficients.
- Write a class *storage* that provides *get\set* methods that only read from the data structure. The *pascal* class can then inherit from it, and do the coefficient calculation. Do you use public or private inheritance?

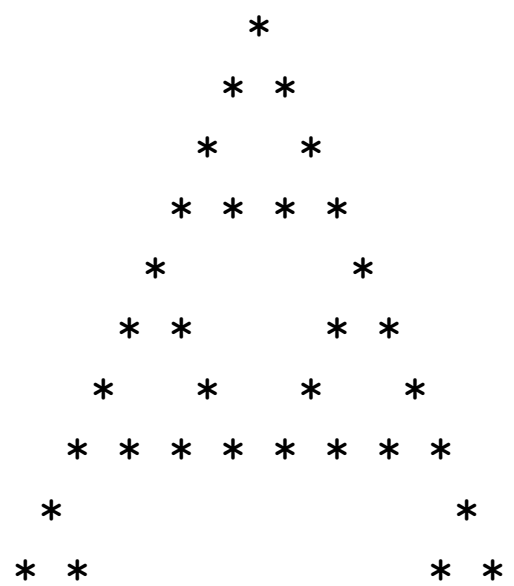
Exercise 6

Extend the *storage* class:

- If a coefficient outside the initial triangle is asked, the triangle should dynamically be extended to the row of that coefficient.
- This requires the *storage* class to extend the space for the coefficients.
- It also requires the *pascal* class to track how many rows have been filled in, and possibly compute some missing coefficients.

Exercise 7

- 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.



- Accept any number of integers; for each, print out the triangle module that number. On zero: stop.

Inherit from containers

9. What is the problem?

You want a `std::vector` but with some added functionality.

```
// proposed construct call:
namedvec<float> x("xvec",5);
// proposed usage:
x.size();
x.name();
x[4];
```

10. Has-a std container

You could write

```
class namedvec {
private:
    std::string name;
    std::vector<float> contents;
public:
    namedvec( std::string n,int s );
    // ...
};
```

The problem now is that for every vector method, `at`,`size`,`push_back`, you have to re-implement that for your `namedvec`.

11. Inherit from vector

Named vector inherits from standard vector:

```
1 // object/container0.cpp
2 #include <vector>
3 #include <string>
4 class namedvector
5   : public std::vector<int> {
6 private:
7   std::string _name;
8 public:
9   namedvector
10    ( std::string n,int s )
11    : _name(n)
12    , std::vector<int>(s) {};
13  auto name() {
14    return _name; };
15 };
```

```
1 // object/container0.cpp
2 namedvector fivevec("five",5);
3 cout << fivevec.name()
4      << ": "
5      << fivevec.size()
6      << '\n';
7 cout << "at zero: "
8      << fivevec.at(0)
9      << '\n';
```

Exercise 8

Extend the code for *namedvector* to make the class templated.

```
1 // object/container.cpp
2 namedvector<float> fivetemp("five",5);
3 cout << fivetemp.name()
4      << ": "
5      << fivetemp.size() << '\n';
6 cout << "at zero: "
7      << fivetemp.at(0) << '\n';
```

Exercise 9

Extend the code from 21 and 8 to make a namespaced class *geo::vector* that has the functionality of *namedvector*.

```
1 // object/container.cpp
2 using namespace geo;
3 geo::vector<float> float4("four",4);
4 cout << float4.name() << '\n';
5 float4[1] = 3.14;
6 cout << float4.at(1) << '\n';
7 geo::vector<std::string> string3("three",3);
8 string3.at(2) = "abc";
9 cout << string3[2] << '\n';
```

Other array stuff

12. Array class

Arrays:

```
#include <array>
std::array<int,5> fiveints;
```

- Size known at compile time.
- Vector methods that do not affect storage
- Zero overhead.

13. Random walk exercise

```
1 // rand/walk_lib_vec.cpp
2 class Mosquito {
3 private:
4   vector<float> pos;
5 public:
6   Mosquito( int d )
7     : pos( vector<float>(d,0.f) ) { };
```

```
1 // rand/walk_lib_vec.cpp
2 void step() {
3   int d = pos.size();
4   auto incr = random_step(d);
5   for (int id=0; id<d; ++id)
6     pos.at(id) += incr.at(id);
7 };
```

Finish the implementation. Do you get improvement from using the array class?

14. Using subarrays

Form *subarray* as part of *array* that starts at the second element:

```
double *array = new double[N];
double *subarray = array+1;
subarray[1] = 5.; // same as: array[2] = 5.;
```

Using 'subarrays' would be useful, for instance in a quicksort algorithm:

```
// Warning: this is pseudo-code
void qs( data ) {
  if (data.size()>1) {
    // pivoting stuff omitted
    qs( data.lefthalf() ); qs( data.lefthalf() );
  }
}
```

15. Span

Create a `span` from a `vector`, starting at its second element:

```
#include <span>
vector<double> v;
std::span<double> v_span( v.data()+1,v.size()-1 );
```

16. Alter sub-vector

Alter a subset of a vector through a span:

Code:

```
1 // span/subspan.cpp
2 vector v{1,2,3};
3 span data( v.data(),v.size() );
4 span tail = data.last(2);
5 for ( auto& e : tail )
6   e = 0;
7 cout << format
8   ("{},{},{}\n",v[0],v[1],v[2]);
```

Output:

```
1,0,0
```

17. mdspan

Create 2D `mdspan` from vector:

```
1 // mdspan/index2std.cpp
2 // matrix in row major
3 vector<float> A(M*N);
4 std::mdspan
5   Amd{ A.data(),std::extents{M,N} };
```

18. Four-d mdspan matrix

Construct a multi-dimensional span from a vector:

```
1 // mdspan/midpoint.cpp
2 vector<float> ar10203040(10*20*30*40);
3 auto brick10203040 =
4   md::mdspan< float,
5               md::extents<size_t,10,20,30,40> >
6                   ( ar10203040.data() );
7 auto midpoint = brick10203040[5,10,15,20];
```

19. Rowsum calculation

Given `mdspan mat`, find its sizes, extract each row, and the sum of its elements:

```
1 // mdspan/index2std.cpp
2 int M = mat.extent(0); int N = mat.extent(1);
3 vector<float> rowsums(N);
4 for ( int row=0; auto& rs : rowsums ) {
5   auto the_row =
6     rng::iota_view(0,M)
7     | rng::views::transform
8       ( [mat,row] (int col) -> float {
9         return mat[row,col]; } );
10  rs = rng::accumulate( the_row, 0.f );
11  row++;
12 }
```

Note that `the_row` is a view, not a data structure.

Array creation

C-style arrays still exist,

```
1 // array/staticinit.cpp
2 {
3   int numbers[] = {5,4,3,2,1};
4   cout << numbers[3] << '\n';
5 }
6 {
7   int numbers[5]{5,4,3,2,1};
8   numbers[3] = 21;
9   cout << numbers[3] << '\n';
10 }
```

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