# Vectors and Objects

Victor Eijkhout, Susan Lindsey

Fall 2024

`last formatted: October 3, 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

```cpp
// array/matrix.cpp
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);
  };
```

(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:**

```cpp
// array/matrix.cpp
A.set(3.);
cout << "Sum of elements: "
     << 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

Better idea:

```
// array/matrixclass.cpp
class matrix {
private:
  vector<double> the_matrix;
  int m,n;
public:
  matrix(int m,int n)
    : m(m),n(n),the_matrix(m*n) {};
  void set(int i,int j,double v) {
    the_matrix.at( i*n +j ) = v;
  };
  double get(int i,int j) {
    return the_matrix.at( i*n +j );
  };
```

# 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 `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.

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

# 8. Exercise continued

- The object needs to have an array internally. The easiest solution is to make an array of size $n \times n$.
- Your program should accept:
    1. an integer for the size
    2. any number of integers for the modulo; if this is zero, stop, otherwise print stars as described above.

# Optional exercise 6

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
  ```

# 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:

```cpp
// object/container0.cpp
#include <vector>
#include <string>
class namedvector
  : public std::vector<int> {
private:
  std::string _name;
public:
  namedvector
    ( std::string n,int s )
    : _name(n)
    , std::vector<int>(s) {};
  auto name() {
    return _name; };
};
```

```cpp
// object/container0.cpp
namedvector fivevec("five",5);
cout << fivevec.name()
     << ": "
     << fivevec.size()
     << '\n';
cout << "at zero: "
     << fivevec.at(0)
     << '\n';
```

# Exercise 7

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

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

# Exercise 8

Extend the code from 21 and 7 to make a namespaced class `geo::vector` that has the functionality of `namedvector`.

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

```
// rand/walk_lib_vec.cpp
class Mosquito {
private:
  vector<float> pos;
public:
  Mosquito( int d )
    : pos( vector<float>(d,0.f) ) { };
```

```
// rand/walk_lib_vec.cpp
void step() {
  int d = pos.size();
  auto incr = random_step(d);
  for (int id=0; id<d; ++id)
    pos.at(id) += incr.at(id);
};
```

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:

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

Output:

```
1,0,0
```

# 17. mdspan

Create 2D `mdspan` from vector:

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

# 18. Four-d mdspan matrix

Construct a multi-dimensional span from a vector:

```
// mdspan/midpoint.cpp
vector<float> ar10203040(10*20*30*40);
auto brick10203040 =
  md::mdspan< float,
              md::extents<size_t,10,20,30,40> >
                  ( ar10203040.data() );
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:

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

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

# Array creation

C-style arrays still exist,

```
// array/staticinit.cpp
{
  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)