Introduction to Program Design, Abstraction and Problem Solving

Chapter 6 arrays and vectors

Textbook: C++ for Everyone by Cay Horstmann

CS 215

What we have done

- Functions.
- Variable scope.
- Call-by-value and call-by-reference.

Example: minimum and maximum numbers

Suppose you have a sequence of numbers: 32.0 55.1 98.2 125.0 76.8.
How would you find the largest and smallest?

- Go through the sequence one by one.
- But how to do that in C++?
- `double n1, n2, n3, n4, n5;` ?
- But how could you loop over those?
- And what if there are a hundred?

Arrays

The **array** is a data structure to store a list of values. Arrays are:

- **Homogeneous**: All the elements have the same type.
- **Fixed size**: an array is allocated with a certain size, and holds exactly that many elements: no more, no less.
- **Ordered**: The elements come in a sequence: first, second, third, . . .
- **Random access**: You can access any element by its index number.

So instead of having a bunch of `double` variables, we have one array variable that holds several `double`s.

n1 =	32.0
n2 =	55.1
n3 =	98.2
n4 =	125.0
n5 =	76.8

$$\texttt{values} = \left\{ \begin{array}{c} 32.0 \\ \hline 55.1 \\ \hline 98.2 \\ \hline 125.0 \\ \hline 76.8 \end{array} \right.$$

Defining an array

- To define an array in C++, use square brackets:

  type name[*size*];

  ```
  double values[5];
  ```

  You can make your code clearer by defining a constant:

  ```
  const int SIZE = 10;
  double values[SIZE];
  ```

  - The size *must* be a constant, not a variable.

- There are several ways to initialize an array:

  ```
  int squares[5] = { 0, 1, 4, 9, 16 };
  int squares[] = { 0, 1, 4, 9, 16 }; // The same.
  int squares[5] = { 0, 1, 4 }; // 0, 1, 4, 0, 0
  ```


Accessing an array

$$\texttt{squares} = \left\{ \begin{array}{c} 0 \\ \hline 1 \\ \hline \cancel{4}5 \\ \hline 9 \\ \hline 16 \end{array} \right.$$

- Accessing an element of an array also uses square brackets:

  ```
  cout << squares[2]; // 4
  ```

  The number inside the brackets is called the **index** into the array.

- You can modify the values in an array:

  ```
  squares[2] = 5;
  ```

- Remember, all the elements must have the same type!

  ```
  squares[3] = "Hello"; // Error!
  ```

Indices and counting

Arrays start counting from zero, not one!

$$\texttt{squares} = \begin{cases} \texttt{squares[0]} = 0 \\ \texttt{squares[1]} = 1 \\ \texttt{squares[2]} = 4 \\ \texttt{squares[3]} = 9 \\ \texttt{squares[4]} = 16 \end{cases}$$

- That means that the index must be between 0 and *one less than* the size of the array.

```
int squares[5];
squares[4] = 16; // Good
squares[5] = 25; // Error
```


Bounds errors

The legal indices of an array are between 0 and `size-1`. Trying to access any other index is called a **bounds error**. What happens if you run code with a bounds error?

- "Undefined behavior"—anything could happen!
- That makes bounds errors hard to debug.
  - Maybe the debugger will catch it.
  - Maybe your program will crash.
  - Maybe it will overwrite the next variable.
  - Maybe it will overwrite your *code*.

Array Limitation

Arrays are great, but they do have some limitations:

- The array capacity must be known in advance.
- We need to keep track of the current size by hand.
  - And have to remember to pass it to functions.
- Arrays work strangely with functions: we'll see more about this next time.

Vectors

C++ provides a data structure called the **vector** that solves many of the problems we mentioned in the previous slide.

- Vectors are not fixed in size or capacity.
  - You can keep adding things forever.
  - . . . until you run out of memory, anyway.
- They keep track of their own size.
  - No extra variables or constants needed!
  - And no extra function parameters.
- They can be passed by value or reference, and returned.

Defining vectors

The syntax for vectors is very different from arrays:

```
vector<type> name;
```

Example: `vector<double> scores;`

- You must `#include <vector>` first!
- Can specify an initial size in *parentheses*:

  ```
  vector<int> squares(5);
  ```

  - The size is zero if not specified.
  - Get the current size with `squares.size()`
- Access elements like an array: `cout << squares[2];`
  - Indices count from zero, like an array.
  - Valid indices are between 0 and `size-1`.
  - Still no protection from bounds errors!

Growing vectors

You can add an element to the end of a vector with `push_back`.

```
scores.push_back(87.5);
```

- This increases the size of the vector by 1.
- Vectors don't support array-style initialization:

  ```
  vector<int> squares = { 0, 1, 4 } // Error
  ```
- Instead, use repeated calls to `push_back`:

  ```
  vector<int> squares; // size 0
  squares.push_back(0);
  squares.push_back(1);
  squares.push_back(4); // now size 3
  ```
- Another way to initialize a vector:

  ```
  vector<int> squares(3); // start at size 3
  squares[0] = 0; squares[1] = 1; squares[2] = 4;
  ```
- This won't work:

  ```
  vector<int> squares;
  squares[0] = 0; // Error: 0 is out of bounds!
  ```

Shrinking vectors

You can also remove the last element of a vector with `pop_back`.

```
scores.pop_back();
```

- This decreases the size of the vector by 1.
- The popped data is lost forever.
- You must first check that the vector has at least one element. Otherwise it is the same as a bounds error.

```
if (scores.size() > 0)
    scores.pop_back();
```


Vector algorithms: filling

One way to fill a vector is similar to filling an array: pre-allocate the vector with the amount of space you need, then use a loop and indexing.

```
vector<double> roots(5); // 5 copies of 0.0
  for (int i=0; i<5; i++)
      roots[i] = sqrt(double(i));
```

Another way is to use `push_back` in a loop.

```
vector<double> roots; // empty
for (int i=0; i<5; i++)
    roots.push_back(sqrt(double(i)));
```

Copying a vector is very easy:

```
lucky_numbers = squares;
```

Vector algorithms: maximum and minimum

Back to our first example: how to find the maximum value in a vector?
Use a loop again, with a variable to keep track of the largest value so far.

`largest` = 95

`values` =

42
25
78
95
46

Vector algorithms: maximum and minimum

```
vector<double> values;
......
double largest = values[0];
for (int i = 1; i < values.size(); i++) {
    if (values[i] > largest)
        largest = values[i];
}
```

- Element zero is initially the maximum.
- For each element *after that*:
  - If it's larger than the maximum, it becomes the new maximum.
- After we repeat for all elements, `largest` holds the maximum.
- Minimum is the same: just use < instead of >.

Vector algorithms: removing

What if you want to remove an element from a vector?

values.size() = 4

values =

42
25
7895
9546
46

```
int pos = 2;
for (int i = pos + 1;
      i < values.size(); i++)
{
  values[i - 1] = values[i];
}
values.pop_back();
```

- Say we want to delete the element with index of 2.
- We don't want a gap in our vector.
- So we have to move all the other elements down.
- And finally remove the now-duplicate last element.

Vector algorithms: inserting

Inserting into the middle of a vector needs a similar approach. Now we must move *up* all the elements after the new one.

values.size() = 5

values =

42
25
9518
4695
046

```
int pos = 2, newval = 18;
values.push_back(0);
for (int i = values.size() - 1;
      i > pos; i--)
{
    values[i] = values[i - 1];
}
values[pos] = newval;
```

- Say we want to insert the new element at index of 2.
- Push a new element onto the end.
- Move all the elements following the insertion spot up by one.
  - Have to start at the end and work backwards. Why?
- Finally, put the new element in its place.

Other vector/array algorithms

Chapter 6 of the textbook has many more algorithms for vectors and arrays. You should become familiar with them all:

- Linear search.
- Sum and average.
- Printing elements with separators.
- Inserting and removing when order doesn't matter.
- Reading input into an array.

Next time

Next time we'll also discuss:

- Arrays and functions.
- Partially filled arrays.
- Multidimensional arrays and vectors.

Vectors versus arrays

`int array_var[SIZE];` `vector<int> vec_var;`

+ Vectors track their own size.
+ Vectors can shrink and expand.
+ Vectors can be more easily copied.
- Vectors are a little less efficient.
- Vectors are harder to initialize (before C++11).

- Arrays are lower-level than vectors.
  - In fact, vectors are usually implemented using arrays.
- Interfacing with older code may require arrays.

Array algorithms: copying

What if you want to store values from one array into another?

```
int powers[5] = { 0, 1, 2, 4, 9 };
int lucky_numbers[5];
lucky_numbers = powers; // Doesn't work!
```

- Array elements can be assigned, but arrays cannot.
- We'll need to copy elements one at a time with a loop.

```
for (int i = 0; i < 5; i++)
{
    lucky_numbers[i] = squares[i];
}
```

- Make sure the destination array is big enough!

Partially filled arrays

- The **capacity** is the maximum number of elements.
- Define the array with the capacity as its size:

```
const int CAPACITY = 100;
double scores[CAPACITY];
```

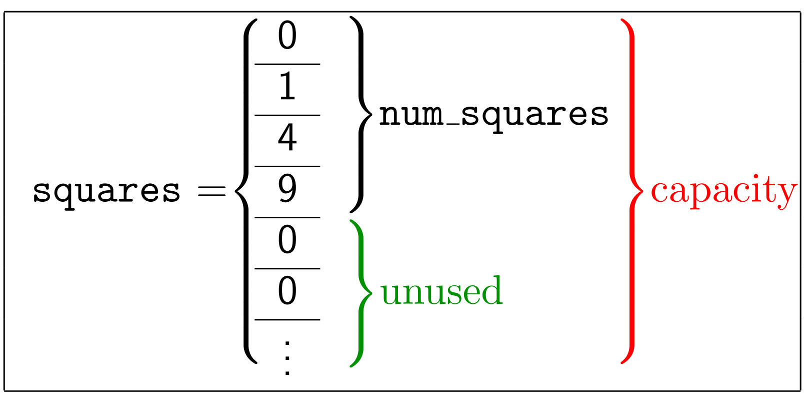
- Use a **companion variable** to track how many actual values are in the array (the **current** or **actual** size).

```
int num_scores = 0; // scores is empty
num_scores = 4; // scores now holds four elements
```

- You can name the companion variable whatever you want.
  - Try to make the meaning clear.
  - Common conventions: `num_scores`, `score_count`, `nscores`

Partially filled arrays

```
int squares[100] = { 0, 1, 4, 9 } ;
int num_squares = 4;
```



- The extra elements of the array are still there.
- It is the programmer's responsibility to track the actual size.

Partially-filled arrays: inserting

To add something to the end of a partially-filled array:

1. Make sure `curr_size < capacity` !
2. Put the new item into `array[curr_size]`
3. Increment `curr_size`.

To add to the middle, use the same algorithm as for vectors:

1. Increment `curr_size`.
2. Shift elements down.
   - Starting at the end and going backwards to the insertion spot: `array[i] = array[i-1]`
3. Put the new element in its place.

Partially-filled arrays: removing

To remove something from the end of a partially-filled array:

1. Make sure the current size is not zero!
2. Decrement `curr_size`.

Why didn't we have to change the array?

To remove from the middle, same as for vectors:

1. Shift elements up.
   - Starting at the item being removed and going up to the end: `array[i] = array[i+1]`
2. Remove the last element (decrement `curr_size`).

Arrays and functions

- Arrays can be passed as parameters to functions, but they act a little strange.
- The function doesn't know the size of the array.
  - Pass it as a second parameter.
  - If you need the capacity as well, add a third parameter.
  - Array parameters are written without a size:
    `double maximum(double values[], int size)`
- Arrays are *always passed by reference*.
  - But don't write an address of operator!
  - We'll learn the reason in chapter 7.
  - `void multiply(double values[], int size, double factor)`

Arrays and functions

- Arrays cannot be used as return values!
  - If you need to "return" an array, make it a parameter.
  - That works because arrays are passed by reference.
  - If the function resizes the array, you must return the new size.
    ```
      int remove_element(double values[], int size, int pos)
    num_scores = remove_element(scores, num_scores, 0);
    ```
  - Alternatively, pass the size by reference.
    ```
    void remove2(double values[], int &size, int pos)
    remove2(scores, num_scores, 0);
    ```
- Some algorithms need both the size and the capacity.
  ```
  void insert(double values[], int &size, int capacity,
              int pos, double newval)
  ```

Arrays and functions: example

Let's write a function to remove an element from an array.We saw the function signature in the previous slide:

```
int remove_element(double values[], int size, int pos)
{
    if (size > 0 && pos < size) {
        for (int i = pos + 1; i < size; i++)
            values[i - 1] = values[i];
        size--;
    }
    return size;
}
```


Vectors and functions

- Vectors can be passed as parameters to functions.
- Vectors can be used with call-by-value or call-by-reference.
- Vectors can also be used as return values.

Vectors and functions: call-by-value example

Let's write a function to compute the sum of numbers from a vector.

```
double sum(vector<double> values)
{
    double total = 0;
    for (int i = 0; i < value.size(); i++)
        total = total + values[i];
    return total;
}
```

Some programmers use a constant reference for efficiency as we introduced in Chapter 5:

```
double sum(const vector<double>& values)
```


Vectors and functions: call-by-reference example

Let's write a function to multiply all values of a vector with a given factor.

```
void multiply(vector<double>& values, double factor)
{
    for (int i = 0; i < value.size(); i++)
        values[i] = values[i] * factor;
}
```

Vectors and functions: return value example

Let's write a function to return a vector of squares from 0^2 to $(n-1)^2$.

```
vector<int> squares(int n)
{
    vector<int> result;
    for (int i = 0; i < n; i++)
        result.push_back(i*i);
    return result;
}
```


Two-dimensional arrays

Sometimes you need to store tabular data in your program.

	hw1	hw2	**hw3**	hw4	
Charlie Brown	90	80	**85**	95	row 0
Olaf Snowman	**95**	**100**	**98**	**100**	row 1
Harry Potter	60	70	**60**	90	row 2
	col 0	col 1	col 2	col 3	

- Organized as **rows** and **columns**
- To locate an entry we need a row number and a column number.
  - All of the entries are the same type.
  - We call this a matrix or a **two-dimensional array**.
  - In C++, we use an array with two subscripts:
    ```
    const int STUDENTS = 3, HOMEWORKS = 4;
    double scores[STUDENTS][HOMEWORKS];
    ```
  - This is like an array of 3 arrays, each with 4 doubles.
  - By convention we put the row number first: array of rows.

Defining 2D arrays

To define an array of zeros:

```
const int STUDENTS = 3, HOMEWORKS = 4;
double scores[STUDENTS][HOMEWORKS];
```

To create the array pre-initialized:

```
double scores[3][4] = {
    { 90.0, 80.0, 85.0, 95.0 },
    { 95.0, 100.0, 98.0, 100.0 },
    { 60.0, 70.0, 60.0, 90.0 }
}; // Don't forget the semicolon!
```

- The first dimension is the number of rows.
- The second is the number of things in each row.
- Both dimensions must be constants!

Using 2D arrays

	hw1	hw2	**hw3**	hw4	
Charlie Brown	90	80	**85**	95	row 0
Olaf Snowman	**95**	**100**	**98**	**100**	row 1
Harry Potter	60	70	**60**	90	row 2
	col 0	col 1	col 2	col 3	

- To access a particular element, use a double subscript:
  ```
  cout << scores[1][2]; // 98
  ```
- Remember, the row comes first, then the column.
- Can also change the value:
  ```
  scores[0][3]++;// 96
  ```

Nested loops

If we want to loop over a 2D array, we need two loops: one for the rows, and one for the columns.

- The loops will be **nested**, one inside the other.

```
for (int i = 0; i < STUDENTS; i++)
{
    // Process the ith row.
    for (int j = 0; j < HOMEWORKS; j++)
    {
        // Process the jth column of the ith row.
        cout << "\t" << scores[i][j];
    }
    // Start a new line after every row.
    cout << endl;
}
```

- The outer loop iterates over the rows.
- The inner loop iterates over the columns of that row.

2D arrays and functions

It is possible to pass 2D arrays to functions. They behave similarly to 1D array parameters, with one big exception.

- You **must** specify the number of columns as a constant.

```
void print_scores(double scores[][HOMEWORKS]);
```

- Why is this necessary?
  - Even though the array looks 2-dimensional, everything in the computer is a linear (one-dimensional) sequence of numbers.
    90, 80, 85, 96, 95, 100, 98, **100**, 60, 70, 60, 90
  - If you ask for `scores[1][3]`, the compiler must figure out where in the linear sequence it is.
    - Skip 1 row, then skip 3 numbers.
    - How big is a row? That's what we have to tell the function.
    - Skip 1*HOMEWORKS + 3 = 7 numbers.

2D arrays in memory

scores =	scores[0] =	scores[0][0] = 90
		scores[0][1] = 80
		scores[0][2] = 85
		scores[0][3] = 96
	scores[1] =	scores[1][0] = 95
		scores[1][1] = 100
		scores[1][2] = 98
		scores[1][3] = 100
	scores[2] =	scores[2][0] = 60
		scores[2][1] = 70
		scores[2][2] = 60
		scores[2][3] = 90

2D vectors

It is also possible to define and use two-dimensional vectors.

- New rows can be added easily.
- And each row can be a different length.
- Really: a vector of vectors.

```
vector<vector<double> > scores
```

  - Important: You should have a space in > >
  - Otherwise older C++ versions think you're using the extraction operator.
- Each row of scores is a vector of doubles.

Initializing 2D vectors

2D vectors are a bit of a pain to initialize. We need a nested loop, and a temporary vector for the "current" row.

```
vector<vector<double> > scores; // 0 x 0
for (int i=0; i < STUDENTS; i++)
{
    vector<double> tmpvec;
    for (int j=0; i < HOMEWORKS; j++)
    {
        tmpvec.push_back(100.0);
    }
    scores.push_back(tmpvec);
}
```


Using 2D vectors

Looping over a 2D vector is similar to a 2D array. The biggest difference is that we have to check the size of each row.

```
for (int i = 0; i < scores.size(); i++)
{
    // Process the ith row.
    for (int j = 0; j < scores[i].size(); j++)
    {
        // Process the jth column of the ith row.
        cout << "\t" << scores[i][j];
    }
    // Start a new line after every row.
    cout << endl;
}
```