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?

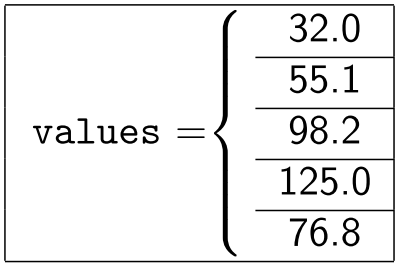
Arrays

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

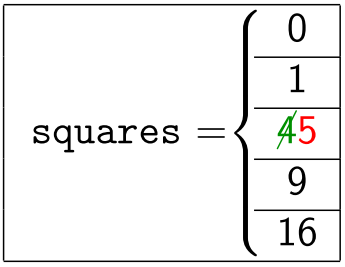
n1 =	32.0
n2 =	55.1
n3 =	98.2
n4 =	125.0
n5 =	76.8



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



- 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.

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];
```

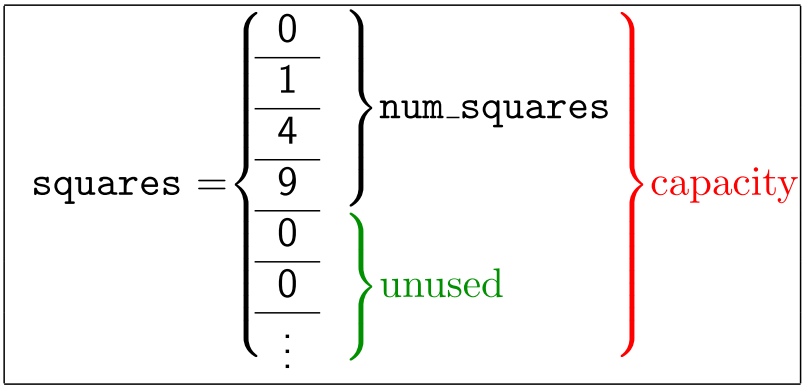
- Use a **companion variable** to track how many actual values are in the array (the **current** or **actual** size).

```
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:

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:

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`).

If a function is declared to return "void", then which statement below is true?

A. The function cannot return until reaching the end of the function body.

B. The function needs a return statement that always returns the integer value zero.

 C. The function can have a return statement with no return value.

D. The function cannot be invoked unless it is in an assignment statement.

What is the error in the following function definition?

```
int find_min(int x, y)
{
      int min = 0;
      if (x < y) { min = x; }
      else { min = y; }
      return min;
}
```

A. The function returns the maximum instead of the minimum of the two arguments.

B. The function does not return a value.

C. The function returns 0 if the first and second arguments are equal.

 D. The function does not specify a type for the second parameter variable.

What are the values of x and y after executing the given code snippet?

```
void mystery(int& a, int& b) {
   a = b;
   b = a;
}
int main()  {
   int x = 10;
   int y = 11;
   mystery(x, y);
   return 0;
}
```

A. x = 10 and y = 11
B. x = 11 and y = 10
C. x = 10 and y = 10

D. x = 11 and y = 11
E. x = 0 and y = 0

4

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 9.
  - `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;
}
```

Coding with your instructor

Please download the source file from the following link:

https://www.cs.uky.edu/~yipike/CS215/DemoArrays.cpp

We will practice how to use arrays to organize data items.