Functions and arrays

Arrays and pointers

Michael Burrell

March 21, 2024

Readings

Chapter 2 — 2.3.2, 2.4.2

Chapter 3 — 3.5

Chapter 6 — 6.2 (especially 6.2.3)

Goals for this set of slides

- Be able to use arrays with functions
- Use array operations with pointers
- Understand `const`

Call by value and C

- To start with, we will consider functions calls like C does
- C is a much simpler language than C++
  - The basics of C++ are the same as C
  - Later, we will see extra features and complications that C++ allows
- In C, *all* arguments to functions are call-by-value
  - Call-by-reference is not supported

Call by value and call by reference

Call by value — a copy of the argument is made and push onto the call stack, to be read as a parameter by the function

Call by reference — no copy of the argument is made. Instead, a pointer/reference to the value is made and passed

Consequences of call-by-value


```
1  void foo(int x) {
2      x *= 2;
3  }
4  int main() {
5      int x = 3;
6      foo(x);
7      std::cout << x << std::endl;
8  }
```

A consequence of call-by-value is that `main`'s variable `x` is not changed, which aids in clarity (maintaining scope).

Call-by-reference

- Sometimes we actually do want call-by-reference
- Sometimes it is more efficient (avoiding making a copy)
- Sometimes we want another function to change the value of a variable

Call-by-reference

- Sometimes we actually do want call-by-reference
- Sometimes it is more efficient (avoiding making a copy)
- Sometimes we want another function to change the value of a variable
- We can use pointers to achieve this

Example of call-by-reference


```
1  void divide(int dividend, int divisor, int*
       quotient, int* remainder) {
2      *quotient = dividend / divisor;
3      *remainder = dividend % divisor;
4  }
5  int main() {
6      int q, r;
7      divide(305, 17, &q, &r);
8      cout << q << "␣" << r << endl;
9      return 0;
10 }
```

Call-by-reference

- Passing a pointer into a function allows that function to change a variable in another stack frame
- This can hurt readability (violates scoping), but adds a lot of power and flexibility to the language

Call-by-reference wrap-up

- Call-by-value is the usual way arguments are passed to functions
- Call-by-reference can be used when we want to modify a value inside the function
- Call-by-reference can also be used for efficiency reasons, avoiding making a copy of the argument

Pointer arithmetic

- Remember that arrays decay into pointers very commonly in C and C++
- Even when using `[ ]` in C++, it is using pointers
- The following two statements are identical:

```
1 x[3] = 4;
2 *(x+3) = 4;
```

Pointer arithmetic


```
*(x+3) = 4;
```

- The above line of code is using *pointer arithmetic*
- If `x` is a pointer to the first element in an array, then `x+1` is a pointer to the second element in the array, and so on
- Note that, when using pointer arithmetic, we don't have to consider the size of an element
  - If `sizeof (int)` is 4, then the compiler knows to multiply by 4 for any arithmetic we're doing

Pointer arithmetic

Pointer arithmetic can be done from the middle or end of an array, as well.

```
1 int x[] = { 3, 4, 5, 6, 7 };
2 int* y = x + 2; // y is pointing to the 5
3 int* z = y - 1; // z is pointer to the 4
4 cout << z[3] << endl; // prints out 7
```

Conclusion pointer arithmetic

- By default, the [ ] operator in C++ works through pointer arithmetic
- Given a pointer to the first element in an array, [ ] adds on a value to get a pointer to a different element
- [ ] implicitly does a * operator to get the actual value

Arrays are different

- C++ is primarily a call-by-value language
- For small, primitive pieces of data (like `int` or `double`), this is not a problem
  - Passing one of these to a function simply means that the function gets a *copy* of the argument
- But for bigger and more complicated data structures, this can cause problems
  - Making a copy of an array could be a computationally intensive task

Arrays are different

- C++ is primarily a call-by-value language
- For small, primitive pieces of data (like `int` or `double`), this is not a problem
  - Passing one of these to a function simply means that the function gets a *copy* of the argument
- But for bigger and more complicated data structures, this can cause problems
  - Making a copy of an array could be a computationally intensive task
- Arrays *cannot* be passed as arguments to functions

Arrays decay into pointers

- Arrays decay into pointers anyway, though
- Any time the name of an array is used in an expression, it has just decayed into a pointer
- Pointers are small, primitive things
  - They can be copied easily and efficiently
  - A pointer to the first element of an array has almost everything need anyway

Small example


```
1  int foo(int* x)
2  {
3      return x[0] + x[1];
4  }
5  int main()
6  {
7      int z[] = { 3, 4, 5 };
8      cout << foo(z) << endl;
9      return 0;
10 }
```

Arrays and functions

- Arrays cannot be passed to arrays
- But pointers can!
- Since an array decays into a pointer to its first element, we can still access arrays in functions

Size of an array

Remember this trick?

```
1 for (size_t i = 0; i < sizeof x / sizeof x[0]; i++) {
2     // ...
3 }
```

This only works if `x` is an array. It doesn't work if `x` is a pointer!

Losing information

- When we pass an array to a function, we *only* get a pointer to its first element
- We have lost all information about how big that array is
- There is no way to recover that information
- If a function needs to know how big an array is, that information has to be passed in explicitly

Passing in the size of the array


```
1  double sum(double* vals, size_t n)
2  {
3      double s = 0;
4      for (size_t i = 0; i < n; i++) {
5          s += vals[i];
6      }
7      return s;
8  }
9  int main()
10 {
11     double x[] = { 7e2, 3.1e-5, -6 };
12     cout << sum(x, sizeof x / sizeof x[0]) << endl;
13     return 0;
14 }
```

Common pattern

- Whenever you have a function which is accepting a pointer to the first element of an array as a parameter, you should always always *always* have another parameter right next to it (of type `size_t`) so the function knows how many elements are in the array
- There is no other way of determining the size of the array

Example

Class exercise

Let's write a program which has an array of prices and produces a receipt, with a total cost.
Then let's modify it so that the cheapest item will be changed to become free. Pointers will make this straightforward for us.

Modifying arrays


```
1  double evil_sum(double* vals, size_t n)
2  {
3      double s = 0;
4      for (size_t i = 0; i < n; i++) {
5          s += vals[0];
6      }
7      vals[0] = 0; // im in ur array messing up ur
           values
8      return s;
9  }
```

Sometimes we want a guarantee that our array won't be messed with!

Arrays and functions and mutability

- When passing arrays, we have to pass a pointer to the array
- Using a pointer, we can modify values
- Sometimes we want a guarantee that our values won't be messed with!
  - This can make our program easier to understand

const

- The keyword `const` in C++ can help us with this
- It indicates that data cannot be changed

Examples of const


```
1 int x = 4;
2 x++;    // OK
3 const int y = 4;
4 y++;    // ERROR! Compiler will disallow this!
```

It is of especial use with pointers


```
1 int x = 5;
2 const int* y = &x; // OK
3 cout << *y << endl; // OK
4 *y = 10;   // ERROR! Compiler will disallow this!
```

A tale of two consts

- When using `const` in conjunction with pointers, there are two places to put the `const`
- Before the `*` means that the values pointed at cannot be changed
- After the `*` means that the pointer itself cannot be changed

const and pointers


```
1  int x[] = { 5, 6 };
2  const int* y = x;
3  int* const z = x;
4  const int* const w = x;
5  (*y)++; // ERROR
6  y++;    // OK
7  (*z)++; // OK
8  z++;    // ERROR
9  (*w)++; // ERROR
10 w++;    // ERROR
```

A tale of two consts

- When in doubt, just remember that the `const` *before* the $*$ is what you want 99% of the time

const and functions

- In this course, from this point forward, we will be *const-correct*
- Any time a pointer is suitable as being declared with `const` (before the `*`), we will declare as so

Final example sum


```
1  double sum(const double* vals, size_t n)
2  {
3      double s = 0;
4      for (size_t i = 0; i < n; i++) {
5          s += vals[i];
6      }
7      return s;
8  }
```

Example

Class exercise

Let's write a program which defines a function which finds the largest number in an array.

In these slides, we studied

- C++'s pass-by-value nature
- Arrays cannot be passed to functions (but pointers can!)
- Working with arrays and functions
- `const`