MTH 4300, Lecture 4

Functions;
Arrays;
Memory

E Fink

February 10, 2025

1. Functions

Just like in Python, we use functions to break down our programs into smaller units, which are easier to write, reason about, and debug.

In C++, you cannot (really) define a function within a function. Hence, your functions should be defined outside of `main()`, probably prior to it.

To define a function in C++, you don't use a keyword. Instead, you write code with the following syntax:

return_type fn_name (*params*) {
    body
}

where "*params*" should be a list of the form

type1 param1, type2 param2, type3 param3

(of course, using as many or as few parameters as your function needs). The top line is called the *signature line*.

This function can be called at any lower line in the program.

Functions

So, for example, a function might look like:

```
// Returns the distance between a and b in the alphabet,
// if a and b are letters with the same case.
int letter_dist(char a, char b) {
      int difference = std::abs(b - a);
      return difference;
}
```

Notice that the return value is an `int`, which matches the return type which begins the signature line.

Aside from the peculiarities of the signature line, the basic structure of simple C++ functions should match with what you'd expect from Python. And the same goes for calling them: e.g.

```
cout << letter_dist('h', 'e') << endl;
```

should produce 3, since `h` and `e` are three letters apart in the alphabet. (You DON'T put in the parameter and return types when you call the function, just when you write it.)

L4x1_func.cpp

Functions

It is not uncommon to want to *use* a function at a point in your program prior to where you *define* it. (One case where this may be necessary: when function A calls upon function B, and function B calls upon function A.)

In this case, you can write a *forward declaration* of your function: this is line, outside of any function body, which looks like

return_type fn_name (type1, type2);

For example, a forward declaration of the function from before would be

```
int letter_dist(char, char);
```

After such a forward declaration, and line of code referencing the function will be legal, as long as a function definition like the one above is written somewhere in the program, below the declaration.

L4x2_forward.cpp

Functions

You might recall that some functions don't return a value. These functions should be supplied with the return type `void`.

L4x3_void.cpp

Functions are meant to *help* you write code. They work best when you can clearly delineate how to call the function and what the function returns. Comments describing how the function operates are often less helpful.

L4x4_prime.cpp

2. Arrays

There are a couple of analogues to the Python list that we will discuss. The first one is the C++ *array*. Here are the most striking differences.

- Arrays are *homogeneous*: every entry must be of the same data type, which is specified when the array is declared (see below).
- Arrays have to have a fixed length: they have to have a size specified at the time of compilation.
- You cannot "print an entire array" or "copy an entire array": if you need to accomplish these things, you do them one element at a time.
- **Appending, slicing, `max`, `sum`, `len`, etc. do not exist for plain arrays.** That's just the way it is, unfortunately.

The above may make you think that these arrays are a pain to work with. You would be correct. *Vectors*, however, will be more powerful.

Arrays

The declaration of an array takes the form
type var_name [*length*] ;
where *length* has to be a positive, LITERAL integer (or a constant integer). Alternatively, you can use an initializer list in curly braces

type var_name [] = { *val1, val2, val3, . . .* };

but note that this type of list assignment is ONLY allowed at the time of declaration.

Once you create an array, you use (zero-based) indexing like in Python to retrieve or alter entries of the array.

L4x5_array.cpp

The next example shows a bunch of things that don't work in C++!

L4x6_nono.cpp

Arrays

You can have multi-dimensional arrays in C++, as well. The declaration for such an array would look like
type var_name [*num_rows*] [*num_cols*] ;
which would declare an array with *num_rows* entries, each of which is an array with *num_cols* entries.

L4x7_twod.cpp

You cannot write a function that returns an array. You can have an array parameter to a function, although there are important things to discuss about that, which we'll return to later.

3. Memory

An object is a FIXED parcel of memory with associated a FIXED data type. Let's explore this more. Consider a program whose `main()` function is

```
int main(){
      int x = 5, y = 7;
      y = x;
}
```

The variables `x` and `y` will be associated with a particular 4 bytes of memory (since `int`s are stored in 4 bytes $= 32$ bits, usually), each of which are located at particular addresses. During program execution, `x` and `y` will always refer to those exact memory locations.

So, when we write `y = x;`, the value contained in the memory space for `x` is copied into the memory space for `y`, writing over what was present before. This behavior is referred to as *value semantics*: when an assignment occurs, the assigned-to variable refers to the same object, but that object's value is changed.

The alternative to value semantics is *reference semantics*: during an assignment, the assigned-to variable receives the *address* of the object on the right side. (Python uses reference semantics, whether you noticed it or not!)

Memory

We can explore this with the *address* operator. If `x` is a variable, then `&x` evaluates to the memory location of that variable. You can print these addresses, although it will be shown in hexadecimal – this is base 16, which has 16 digits (0-9, A = ten, B = eleven, C = twelve, D = thirteen, E = fourteen, F = fifteen), and place values which are powers of 16 – shown with a prefix of `0x`.

L4x8_address.cpp

An important point about arrays: they are stored in consecutive memory addresses. In the last examples, note that the addresses of `x[0]`, `x[1]`, `x[2]`, `x[3]`, all differ by 4 – since `ints` are stored in 4 bits.

Also, printing out the name of an array usually leads to the address of its first element printing out – unless you happen to have an array of `chars`, in which case you should avoid printing out the name, unless your array ends with the `'\0'` character.