MTH 4300, Lecture 3

Global Scope;
`char`, String Literals, and `std::string`;
`for` Loops;
Functions

E Fink

February 4, 2025

1. Global Scope

Most of the code we write will lie in the bodies of functions, `main()` or others. However, we can have lines which declare and initialize variables outside of any function body. These variables are called *global*. Such variables are accessible within all functions whose bodies appear after their declarations.

Global variables make it easy for many different parts of your program to share a single variable, but this is a recipe for confusion. **The use of global variables should generally be avoided**, at least for variables that are subject to change.

L3x1_global.cpp

2. `char`, String Literals, and `std::string`

`char` is an integer type, representing a SINGLE character. `char` literals are typed with SINGLE quotes. Escape characters, like `\n` for newline, `\"` for the double quote character, etc. can also be put in single quotes for a `char` literal.

Each character corresponds to an integer between 0 and 127, with the correspondence given by ASCII (American Standard Code for Info Interchange):

`'A'` = 65	`'B'` = 66	`'a'` = 97	`'b'` = 98
`'1'` = 49	`'2'` = 50	etc.	

Because of this, you can do arithmetic with characters. **L3x2_char.cpp**

Observe that we are seeing *type conversions* take place. In the first instance: when we add a `char` and an `int`, the `char` value gets converted to an `int`; then the `int` values get added.

In the second instance, we do the same thing, except that when we assign the result to a `char` variable, the `int` value gets converted to a `char`. This is riskier: what if an `int` value is greater than 127?

`char`, String Literals, and `std::string`

There are two main kinds of *strings* in C++: C-strings and `std::string`.

C-strings are basically raw arrays of characters. `std::string` is NOT a fundamental type, but a *class*: a data type constructed around C-strings, which implements convenient functions that you would want to perform with strings. You `#include <string>` to use the `std::string` class.

String literals are typed with DOUBLE quotes, and can accommodate escape characters. The primary reason I bring up C-strings at all is that these string literals are C-strings, which means that you can't do a lot with the literal values – for example, you can't concatenate string literals with each other.

But you can assign string literals to `std::string` variables, and then do the things that you want with those variables.

char, String Literals, and std::string

Some notes about std::string variables:

- If x is a std::string, the length of this string is x.size(). Beware: this function will be UNSIGNED, which means that you can get yourself into trouble if you, say, set x equal to an empty string and then refer to x.size() - 1.
- You can concatenate std::strings like in Python, with +. (You can also concatenate std::strings with chars or C-strings, but you cannot concatenate C-strings with each other – there has to be a std::string at the front of the expression.)
- Unlike in Python, strings ARE mutable, and you can alter individual characters by indexing.
- If x is a std::string, then x.substr(i, len) will return a new substring of x, starting from index i and with len characters (unless the end of x is encountered first).

L3x3_stdstr.cpp

char, String Literals, and std::string

If s is a string variable, the line

```
std::cin >> s;
```

will cause the program to pause and wait for the user to type in some input and press Enter. Then, it will **ignore leading whitespace**, and then all the subsequent entered characters **until the first whitespace character** will be written to s. ("Whitespace" includes spaces, tabs, and newlines.)

If you want to use cin to write an entire line of input, spaces and all, into a string variable s, you can use

```
std::getline(std::cin, s);
```

L3x4_input.cpp

3. for Loops

There is a `for` loop in C++, although it lacks the magic of Python `for` loops. These `for` loops are glorified `while` loops. Syntax:

`for(` *initializer* ; *test* ; *increment* `){`
    body
}

When this loop is encountered:

- first, the initializer is executed;
- then the test is evaluated: if it evaluates to `true`, the body runs, otherwise the loop terminates;
- after the body is executed, the increment code runs, and then the last bullet happens again.

L3x5_for.cpp

Notes: `++i` is a shortcut for `i += 1`; also, see the warning about using comparisons to unsigned ints, like those returned by the `.size()` function.

for Loops

Note that the code on the last page is just a shortcut for

```
initializer;
while( test ){
    body
    increment;
}
```

for loops are appropriate for definite loops – for example, when you are processing data about a sequence. The advantage of them is that they collect initializer, test and increment on the same line of code.

L3x6_bottle.cpp

4. 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.)

L3x7_func.cpp