MTH 4300, Lecture 2

Types, Objects, Values, Variables; Numeric Types; `if-else`, `while` and Scope; `char`, String Literals, and `std::string`

E Fink

February 2, 2025

1. Types, Objects, Values, Variables

A data type is a collection of possible values, together with operations you can perform on or with those values. In C++, there are a collection *fundamental data types*, two of which we've seen (`int` and `double`), as well as more complex types that are built up from the fundamental ones.

An object is a FIXED parcel of memory associated with a FIXED data type. A value is simply an arrangement of bits in that parcel of memory, which would represent a particular value. (For example, an `int` value of 27 would be represented in memory as something like `00011011`.)

A variable in C++ is an object that can be given a name (the variable's *identifier*). Identifier rules are the same as in Python, except that the reserved words in C++ are a little different.

C++ variables have FIXED types and FIXED addresses! Both of these are different than in Python.

Types, Objects, Values, Variables

Since we're talking about types and variables: why is it necessary to declare our variables before we use them, and why must variables have fixed types?

- Declarations make the compiler aware of all variable names being used – they also guard against typos in variable names.
- They allow compile-time type checking: finding code with operations on values having inappropriate types. These give you *compile-time* errors, which are a GOOD thing – these errors are much easier to fix than errors which are caught later on.
- Declarations usually *define* variables, which means to set aside memory space for them – but the compiler needs to know just *how much* space, and the data type lets the compiler know how much space to reserve.

2. Numeric Types

The fundamental data types are all basically numbers. There are two general categories of these data types: those that are integers, and those that are floating points.

The integer types include, among others,

- `int` (the standard integer);
- `long` and `long long` (integers that offer more storage);
- `unsigned int` (like regular `int`, but with only values ≥ 0);
- `char` (representing single characters – each one corresponds to a small integer via ASCII);
- `bool` (values are `true` and `false`, with `false` being represented by 0 and `true` being represented by any non-zero value).

The standard floating point type is the `double`, for "double-precision floating point". There are also `float` (a less-precise floating point type, usually not advantageous for us), and `long double` (a more-precise floating point type, usually not needed for us).

Numeric Types

Here are some warnings about numeric types for those familiar with Python:

- The basic integer type, `int`, is (usually) stored in 4 bytes = 32 bits of memory. That means that an `int` variable usually can hold values between -2147483648 and 2147483647 (e.g., between -2^{31} and $2^{31} - 1$). If a reassignment on an `int` takes it out of this range, unpredictable behavior can occur. (In fact, this is UNDEFINED BEHAVIOR in C++.)
- If you need bigger integer values, `long long` can accommodate integers up to ± 9 quintillion.

L2x1_bignum.cpp

Numeric Types

More warnings:

- An `int` divided by an `int` gives an `int`! The fractional part gets cut off; the quotient is "rounded towards zero."
- The % operator works differently for negative `int`s than in Python.
- Exponentiation is achieved with the `pow()` function, for which you need to `#include <cmath>`. That library also includes functions like `sqrt()`, `sin()`, `exp()`, `log()`, etc., all to be prefixed with `std::`.
- An uninitialized numeric variable will contain a random "garbage" value. (More UNDEFINED behavior.)

L2x2_math.cpp

The example also shows the use of `static_cast`, which is a safe way to convert between values of related types.

3. `if-else`, `while` and Scope

`if`, `else` and `while` work a lot like they do in Python. Some things to note:

- The `bool` literals are `true` and `false` – note that the first letters are lowercase.
- Instead of `or`, there is `||`; instead of `and`, there is `&&`; instead of `not`, there is `!`.
- Instead of `elif`, we write `else if`.
- The control statement next to `if`, `else if` or `while` should be enclosed in parentheses, and the body that the statement controls should be contained in a pair of curly braces.
- Be aware that any integer value that is non-zero will be implicitly converted to `true`.

L2x3_if.cpp

Write a program which asks the user to enter donation amounts, until the funding goal of 10000 is met.

L2x4_donation.cpp

if-else, while and Scope

One more thing worth noting about C++: chained comparisons don't work. E.g., if you wish to say $7 \leq x < 11$, write

```
(7 <= x) && (x < 11)
```

rather than

```
7 <= x < 11
```

(The parentheses in that expression are unnecessary, but do serve to reinforce the order of evaluation.)

Why is this necessary? Logical expressions involving comparisons are evaluated like any other expressions, unlike in Python, which performs a type of magic when it sees chained inequalities. For example, let's evaluate `7 <= x < 11` in C++ when `x` is, say, 4.

Comparison operators are all at the same level of precedence, and evaluate from left to right. So, to evaluate `7 <= 4 < 11`, we start by evaluating the <= operator, which produces

```
false < 11
```

Then, since `false` is treated as `0`, this expression would evaluate to `true`, even though 4 is NOT between 7 and 11.

`if-else`, `while` and Scope

A collection of statements enclosed in curly braces is referred to as a *block statement*. Recall the concept of *scope*: variables defined in functions are generally only accessible within the body of that function (below the line where they are declared).

While this is true for C++ functions too, it is ALSO true for any variables defined in a block statement: such variables will cease to exist at the end of the block.

L2x5_scope.cpp

`if-else`, `while` and 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.

L2x6_global.cpp

`if-else`, `while` and Scope

If you have a block within an outer scope, you are allowed to declare variables in this block which share names with variables declared in the outer scope. This is called *hiding*: references to the shared variable name will refer to the inner-scoped one, while the outer-scoped one is "hidden." Again, good sense says to avoid abusing this feature.

(Note that you cannot declare two variables with the same name in the same block, unless one of those variables is in a block *within* the block. Ech.)

L2x7_hiding.cpp

4. `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.
L2x8_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 types 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 `+`.
- Unlike in Python, strings ARE mutable, and you can alter individual characters by indexing.
- Suppose that `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, all the 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);`

L2x9_stdstr.cpp