02-Variables and Input Output

In the previous section, we looked at a simple "Hello, world!" program to understand how C++ programs are structured. In this section, we will expand on this example by learning how console input/output (I/O) works, and how to store and serialize data in C++.

> **Note: This is going to be the most exhaustive introductory chapter, so bear with me :)**
> If you feel like you know most of this already and want a refresher on this topic, head down to the **TL;DR** section.

Input/Output (I/O)

In order for programs to do anything helpful for end users, they need to be able to be fed data, and give data back to their sender. All software, from calculator firmware to device drivers to video game scripts, works this way.

C++ offers much functionality for reading data from the system, and sending data back to the system. In our sample program, we *print* a string `"Hello, world!\n"` to the console via:

```
        std::cout << "Hello, world!\n";
```

`std::cout` is an object which handles sending output data to ***`stdout`***, the *standard output* stream handle. Many types of data can be written to `stdout` through `std::cout`. `"Hello, world!\n"` is a string (i.e., a series of *characters*) which are printed literally to `stdout`. There are various other types of data that we can define and write to the console.

Just like there are ways to write data to the system, there are equivocally ways to read data *from* the system. `std::cin` is an object which allows us to read data from ***`stdin`***, the *standard input* stream handle. **Before** we go into how to read data from the system, however, **we must understand how to store data in the first place**. Data can be stored in a program using *variables*.

For advanced readers:

> You can read more about `stdout`, `stdin`, and `stderr` at https://en.wikipedia.org/wiki/Standard_streams. In the *UTRGV C++ Primer*, we use the abstractions over these stream handlers provided by the C++ Standard Template Library (STL), `std::cout`, `std::cin`, and `std::cerr` respectively.

Variables

A *variable*, put simply, is a **unit of data stored** somewhere in a program. You may think of a variable in programming similarly to a variable in mathematics; a variable is something that *can change* (usually). Variables in software, much like variables in math, are **placeholders for information**. The difference is that in software, a variable can store any type of information, rather than only numbers.

Variables have three parts: a **type**, a **declaration**, and a **definition**:

- Type - The *type* of a variable refers to what manner (*lit.* what type) of data is stored in a variable. Also called **data types**.
- Declaration - The *declaration* of a variable tells the compiler that a variable with a certain *symbol* (i.e., a name) exists.
- Definition - The *definition* of a variable is the value that it is assigned to. It may also be referred to as the *value* or the *initialization* of a variable.

For example, if we want to define an *integer* variable `x` with the value of `5`, we may do:

```
int x = 5; // <-- a variable definition is a statement, so it ends with a
;
```

> Semantically, we could also say that "`x` *is a variable of type* `int`*, which is initialized with the value* `5`"

Variables (except for `const` ones and compile-time ones) can be reassigned later. So we can write the same semantic as:

```
int x; // declaration (with *default* init)
x = 5; // definition/reassignment
```

Before we learn how to utilize variables to write our programs, it's important to learn what a *type* really is.

Data Types

There are various **data types** native to C++, as well as ones specified by the C++ standard template library (STL). Data types fall into two primary groups:

- **Built-in** types (a.k.a. **fundamental** types)
- **User-defined** types

Built-in Types

Built-in data types (also known as fundamental types) are a set of basic types in C++ which store simple numerical data. The following fundamental types are:

- **Integral types (integers)** - Numerical types which store integer values (1, 2, 3, etc.)
- **Floating-point types (decimals)** - Numerical types which store decimal values (3.14, 2.19, etc.)
- **Character types** - Numerical types which store and represent characters (parts of a string of text).
- **Boolean type** - `bool` represents `true` or `false`.
- `void` - an incomplete type, often means *no return type*. We will cover this in `05-Functions`.

Below is a list of basic C++ built-in types and what data they may store. We will use these data types recurrently without delving too much into them. Keep in mind that some terms here are glossed over, as they are either not relevant, or covered in later sections.

Typename/keyword	Type of data type	Range
`int` / `signed int`	An integral data type.	On most 32 bit and 64 bit systems, an `int` is guaranteed to be at least 32 bits, meaning that it is in the range of `[-2^31, 2^31-1]`, or `[-2147483648, 2147483647]`.
`unsigned int`	An *unsigned* integral data type.	Similarly to a `signed int`, is guaranteed to be at least 32 bits on most commercial systems, but does *not* have a negative range, allowing the programmer to store values `[0, 2^32-1]`, or `[0, 4294967295]`
`float`	A floating-point data type.	On 32 bit and 64 bit systems, a `float` is guaranteed to be 32 bits. Unlike with integral data types, this does not affect the range of this variable, but rather its decimal precision. A float has roughly 7 digits (after `0.`) of reliable precision. Also referred to as a *single-precision floating-point type*.
`bool`	The boolean data type.	The `bool` is a built-in data type which can be set to `true` or `false`. `bool` is often the result of comparisons, which we will cover in `03-Conditional Statements`.

Typename/keyword	Type of data type	Range
`char`	A character data type.	A `char` is an 8-bit (1-byte) integral data type which is used to represent a single ASCII character. *Arrays* of `char`s may be used to represent strings, which we will cover later in this section. Their range is not very relevant, but being 8 bits, they can store values in the range of `[-128, 127]`, or `[0, 255]` if unsigned.

For advanced readers:

> Any fundamental types not mentioned here may or may not be mentioned in a later section. If you want to learn more about fundamental types, take a gander at https://en.wikipedia.org/wiki/Data_type or https://en.cppreference.com/w/cpp/language/types. Both are good.

Similarly to our `int x = 5` example above, the aforementioned fundamental types may be initialized via:

```
int signed_n = -127; // notice that `signed` isnt written explicitly here
signed int also_signed_n = -127; // valid but unnecessary
unsigned int unsigned_n = 65535;

float pi = 3.14159f; // float literals are postfixed with an `f`
double e = 2.718281828459045; /* double-precision type, usually 64 bits
                                                              with 15

digits of reliable precision. */
bool condition = true;
char c = 'A';
```

You may notice that all of the values we initialize the above variables to are values written in the program source code itself. We call variable values written in the program source directly ***literals***.

Arithmetic Operators

You can perform arithmetic such as addition, subtraction, division, multiplication, etc. using ***mathematical operators***. An **operator** in C++ is a symbol that allows you to perform a built-in operation on either 1 or 2 numerical values:

```
int x = 5;
int y = 3;

int z = x + y; // z = 8
```

```
z = x - y;      // z = 2
z = x / y;      // z = 1 (numbers are rounded down for integer division)
z = x * y;      // z = 15
z = x % y;      // remainder! z = remainder of x / y, which is 2.
```

You may also use ***compound operators***, for example:

```
int x = 3;
int z = 2;

z = x + z; // 5
z = 2;
z += x;    // equivalent to z = z + x;. Also 5.
```

and ***unary operators*** to increment and decrement variables:

```
int z = 1;

++z; // z = 2
--z; // z = 1
--z; // z = 0
```

In our "hello world" example, you may have noticed the use of `<<` to conjoin `std::cout` and our string in question:

```
std::cout << "Hello, world!\n";
```

`<<` is *also* an operator, but it's usage depends on the object you're using it on.

For advanced readers:

> The `<<` operator is called the *left bit-shift* operator, and is a *bit-wise operator*. Internally, it shifts the bits of a variable in a number, so `0b0011 << 2`, for example, returns `0b1100`.

User-defined Types

User-defined types in C++ are a broader set of data types that can be defined by the user. This can be done through creating structs, classes, enums, typedefs, aliases, etc., all of which we will cover in **`08-Structs and Classes`**.

For advanced readers:

> Structs are records of variables that are wrapped in a single context. This means that you can have multiple values, say, a `Student` structure with values `name`, `grade`, and `age`:

```
enum Grade {
        FRESHMAN,
        SOPHOMORE,
        JUNIOR,
        SENIOR
};

struct Student {
        std::string name;
        Grade grade;
        int age;
};

int main() {
        Student you = { "Your name", FRESHMAN, 19 };
        std::cout << "Student with name " << you.name
                  << " is " << you.age << " years old.\n";
}
```

The C++ standard library, as mentioned before, provides many such types, which we will cover iteratively over the course of this series. In this section, we will cover what is arguably the most fundamental user-defined data type: *strings*.

Strings

In software, a **string** is a sequence of characters which is used to store human-readable text. Without a doubt, you read character strings every day: from game tool-tips, to social media messages, websites, command consoles, text editors, to this very PDF (or HTML) document. Strings are quintessential to the usefulness of computers to end users, such as yourself.

In C++, strings have various different implementations, but the most important and ubiquitous one that you'll be working with is `std::string`. A string variable may be defined via:

```
std::string s = "This is a string.";
```

For advanced readers:

> *Strings in C and C++ are implemented internally as arrays of `char` values, but we will not cover arrays until `06-Arrays and References`.*

Strings are essential for writing human-readable text to a computer console. Everything you read on a computer system, *even basic numerical values*, is rendered

from a string of text. Basic data that is written to the console is *serialized* into strings internally. For example:

```
int x = 5;
std::cout << "Value of x: " << x << '\n';
```

Output:

```
Value of x: 5
```

In this example, we write a value `x` (which is initialized to `5`) to *`stdout`*. `5` visually prints to the console, which is done by serializing the numerical data `5` being written into the output string. It is usually encoded as the `char` value of `'5'`.

> Don't worry if this is a bit confusing. We will delve into data serialization in a future section, and you'll make sense of chars and numerical types early on :)

Now that we've introduced the concept of variables and types, we can finally learn how to utilize mutable data to read information from our system!

Reading from the System (Back to I/O!)

Like we mentioned earlier, you can read data from the system using ***`std::cin`***, which gives us access to the *standard input stream* ***`stdin`***.

Conversely to `std::cout`, we can read from `std::cin` using the `>>` operator:

```
#include <iostream>

int main() {
        int age;
        std::cout << "Enter your age: ";
        std::cin >> age; // Read integer from console

        std::cout << "You are " << age << " years old.\n";
}
```

Depending on your input, the output would look something like:

```
Enter your age: 22
You are 22 years old.
```

Similarly, you may read floats, chars, and strings from the input console. If we want to read a string from the console, we *could* do:

```
#include <iostream>

int main() {
        std::string name;
        std::cout << "What is your name? ";
        std::cin >> name;

        std::cout << "Hello, " << name << "!\n";
}
```

If you run this program, entering your name probably works as intended, if you are entering only your first name. However, try entering your full name and observing the output:

```
What is your name? Emmett Tomai
Hello, Emmett
```

The output probably isn't what you intended! This is because when writing data from `stdin` into a string, only the first word is read. More specifically, an input string is read until a white-space character is read. If you want to read an entire line from the console, you can use `std::getline()`:

```
#include <iostream>
#include <string>

int main() {
        std::string full_name;
        std::cout << "What is your full name? ";
        std::getline(std::cin, full_name);

        std::cout << "Hello, " << full_name << "!\n";
}
```

As per the name, `std::getline()` reads text from the specified stream (in this case, `std::cin`) until the end of the line. More specifically, it reads the input until a *newline character* `'\n'` is read. A newline character is written to the input stream when you hit ENTER on your keyboard.

Our output would look like:

```
What is your full name? Emmett Tomai
Hello, Emmett Tomai!
```

There are various ways to read data from the console. In future sections, we will include many example snippets that will utilize console I/O. In the next section, we will go over how to evaluate input data to affect the behavior of our programs.

TL;DR

Variables

Variables are placeholders for data. They are comprised of a *type*, a *symbol*, and a *value*.

Variables are initialized via:

```
int x = 5;
double pi = 3.14159;
std::string s = "I am a string!";
```

Values that are written in the program source and evaluated at compile time are called ***literals***. `5`, `3.14159`, and `"I am a string!"` written above are examples of literals.

There are various built-in data types and infinitely many more user-defined types (since you can define them yourself, hence the name!).

Operators

Operators allow you to perform operations on one or more variables. You can perform arithmetic, incrementation/decrementation, and bit-wise operations using operators on numerical types:

```
int x = 5;
int y = 3;

int z = x + y;
z += x;
z = 5 % 3;

unsigned char u8 = 0b00110011 << 2; // u8 = 0b11001100
```

Input/Output

I/O operations allow us to read data from the system and send data back to the system. We can read from and write to the command console using `stdin` and `stdout` respectively. We access them using `std::cin` and `std::cout`:

```cpp
#include <iostream> // access to std::cin, std::cout
#include <string> // access to std::getline(), std::string

int main() {
        std::string name;
        int age;
        double gpa;

        std::cout << "Enter your name: ";
        std::getline(std::cin, name);

        std::cout << "Enter your age: ";
        std::cin >> age;

        std::cout << "Enter your GPA (weighted 4.0): ";
        std::cin >> gpa;

        std::cout << name << " (" << age << ") has a GPA of " << gpa <<
".\n";
}
```

```
Enter your name: Emmett Tomai
Enter your age: 22
Enter your GPA: 3.4
```

> Notice: I made this data up. Emmett Tomai is not 22 not did he have a 3.4 GPA at graduation. Or maybe he did, who knows.

What's Next?

In the next section, we will learn how to evaluate user input using *conditional statements*.

Courtesy of *DeCentralize the Web* (2024)