Introduction to Programming & The Internet of Things

Module 3: Introduction to Programming

1 Modules Background

There is a pervasive need to study the performance of built structures and their abilities to provide occupants with comfort in an efficient manner. The Internet of Things (IoT) provides methods of obtaining and studying data about our built environments. As such, these modules introduce civil, architectural, and environmental engineers to tropics of electrical engineering. Students will gather the abilities needed to program and deploy necessary sensors in the Internet of Things (IoT), as well as gain the ability to gather and analyze data collected.

Contents

2 Introduction

Before we start to code, let's think a little bit about why code was developed! In our process to build robot minions, we needed to develop a way to communicate with inanimate objects. The most popular way to do this is by controlling the intensity of electric potential (density of electric charge). Luckily, there exists platforms which heavily simplify this process and distill it into a form of language.

In order to have a conversation with this new language, we need to be speaking with something that understands. Normally your instructions will be sent into the machine, compiled (or "translated" into numbers), and then executed. While your computer can do this quite well natively and without Internet, the process can be quite different depending on the type of computer you have. Because of this, we'll be using an online compiler so you don't have to think about this process and can focus on learning the language. You can find it here: [1]. Normally compiling this on your computer would require the use of an "interpreter", but we won't have to worry about that as it's all being done online for us.

Note: the Arduino development environment primarily uses a C/C++ language format, so that will be the one discussed.

3 Initial Formalities

Each coding language has their particularities, and in C++ we must do a bit of an initial dance before we can begin our conversation. This includes telling the compiler (language translation engine inside the computer) what kind of words you're about to use.

This can be done by providing the computer with a list of references it should use (think of this as giving the computer a chemistry textbook before you ask it about electron orbitals). This is done at the beginning of the program and normally looks something like this:

```
#include <iostream>
```

In this case, we've given the compiler a book about reading input and pushing output to the screen. This is kind of like telling the computer how we'd like it to communicate with us. If only it was this easy with humans.

Let's say we want to use a common set of definitions. Maybe the human equivalent would be telling someone you have a thick Boston accent so when you say "cawfee" they know that you're talking about the caffeine rich drink. To tell the compiler this is our intention, we need to include the line

```
using namespace std;
```

at the top of our program. Our header won't change as we're not doing anything too complicated to start off, so feel free to throw this at the top of your code:

```
#include <iostream>
#include <string>
using namespace std;
```

Another note before we continue. To have the computer print off things so you can follow its logic and read useful results, use this structure:

```
cout << thing << '\n';
```

Where "thing" is whatever you'd like to print out. You can use this to check if your logic was correct or see where the program was failing!

Note: In the Arduino IDE, this looks a little bit different. Instead, we need to initialize the "Serial Monitor" and use it to print information off. That looks something like this:

```
Serial.begin(9600);
Serial.println("This will be printed to the Serial Monitor");
```

The serial monitor only needs to be initialized one time, so you'll only need to write "Serial.begin(9600);" once in the setup function. After that, the println function can be called as many times as you'd like. Here's a guide from Adafruit which even gives some information about adding input functionality via the Serial Monitor: [2].

4 Value Storage

4.1 Variables

One of the most fundamental parts of spoken language is a basic definition of the meaning of words. If I speak of a raccoon for example, there's a basic understanding that it's a fuzzy little trash monster with particular black marks around its face. This context allows you to have more fluid conversations, without defining what a raccoon is every time you need to describe to your neighbors why your chobani yogurt from yesterday is now on their lawn. Our machines call newly introduced words "variables", and a collection of variables is called an "array".

To define a variable to the machine in C/C++, we first have define the word type (think verb, noun, or adjective). In C++, this definition looks like this:

```
int raccoons = 10;
```

Where the first part (int) tells our compiler what kind of variable we're working with, the second part tells it what the name is, and the third part tells it what initial value we'd like it to have. We can leave it without a value as well, but we need to define an initial value eventually in the main loop before we can work with it. That would look something like this:

```
int raccoons;

int main() {
raccoons = 10;
}
```

Another common type of value is a decimal number, and because that's comprehended in a different way by the computer it has a different type from 'int'. Instead, decimal numbers are stored as 'float' or 'double' types. Declaration of a decimal number looks like this:

```
float thing = 5.23;
```

Notice that every line ends with a semicolon (;)? This is our way of telling the computer that we've finished a sentence. It's kind of like adding a period to the end of a sentence! Note: some languages like python don't use this and instead count the number of spaces to tell when expressions are finished. Check this out if you want to learn more [3].

4.2 Arrays

4.2.1 Declaration

Arrays are just collections of one type of thing, so we need to tell the compiler two things: what the thing is, and that we're about to give it a collection of that thing. Here's an example:

```
int raccoon_locations [5] = { 16, 2, 77, 40, 33 };
```

Perhaps this array is describing the locations of each raccoon. We can't tell from this declaration. What we can tell is what type of information is being stored (int) and how many elements are contained within the array (5). In fact, because we've listed every value within the array, the compiler can count "5" elements by itself so the five is unnecessary. This would be an equivalent statement:

```
int raccoon_locations [] = { 16, 2, 77, 40, 33 };
```

Just like a variable, we can leave the array empty without defining the numbers that are stored. That would look like this:

```
int raccoon_locations [5];
```

Where 5 can be replaced with any integer describing the size of the array.

4.2.2 Accessing Array Values

Fun fact about arrays in most programming languages: they start with index 0! So if we'd like to access the first term of the array, we'd call:

```
raccoon_locations[0];
```

Here's a more in depth summary of arrays (even two dimensional) [4]. Of course it would defeat the purpose of arrays if they weren't easy to work with. To access all of the values in an orderly way, I present for loops!

5 Flow Control

5.1 If Statement

As a kind of building block for decision making, we'd like our computer to be able to make simple decisions. For example, turn on the lights when the brightness of the room is extremely low or water the plant when it's dry. To do this, we need to be constantly looking at the light values or moisture content of the plant. Then, we need to define for the computer what "dark" and "dry" are in terms of the variable of interest. Thus if statements were invented.

In English, we might say that something is dry **if** it has less than 10% water content (just made that percentage up for the sake of an example). If you have a device that measures water content, it would be nice **if** it would add water when the content is less than 10%. Yes, the bolded **if**s are a bit dramatic, but hopefully it drives the point home that we're going to replicate this logic!

Here's how you can use if statements:

```
// Plant Watering Machine
#include <iostream>
#include <string>
using namespace std;

int water_content=25;

int main ()
{
  if ( water_content < 10)
  {
    cout << "Plants need watering";
  }
  else{
    cout << "Plants are okay for now";
  }
  return 0;
}
```

→ Plants are okay for now

But what **if** our water value is less than 10 (alright alright, I'll cut it out with the bold font)? Then we would see this instead:

→ Plants need watering

What's going on inside of the "if" statement? Under the hood, our variable *water_content* is being compared against the value on the other side of the **comparison operator** (in this case "<"). If our value is indeed less than the variable we're comparing against, our system evaluates to *TRUE*. Otherwise, our system evaluates this to *FALSE*. This true/false term is called a **boolean** or **"bool"**, and it gets passed into the if statement. If true, it executes the first block of code. If false (else), it executes the second block.

There are a number of comparison operators, here's a reference if you're interested [5].

Note: "==" is the comparison operator which evaluates if two values are equal. This returns a boolean value. "=" is an **assignment operator**, which assigns new values to a variable. Here's an example:

```
// Plant Watering Machine
#include <iostream>
#include <string>
using namespace std;
```

```
int water_content=25;
int dry_threshold = 10;
bool water_dry = 0;

int main ()
{
  for (int i = 0; i <6; i++){
    cout << "Water Content:\t\t " << water_content << "\n";
    cout << "Water Moist Bool:\t " << water_dry << "\n\n";

    water_content -= 5;
    water_dry = water_content == dry_threshold;
  }
  return 0;
}
```

```
Water Content:25
Water Moist Bool:0

Water Content:20
Water Moist Bool:0

Water Content:15
Water Moist Bool:0

Water Content:10
Water Moist Bool:1

Water Content:5
Water Moist Bool:0

Water Content:0
Water Moist Bool:0
```

Notice two things:

1. The only time Water Moist Bool evaluated to "1" was when water content was the same as our dry_threshold.
2. How odd the line "water_dry = water_content == dry_threshold;" looks. What's going on? First, water_content is getting evaluated against dry_theshold. The computer is looking to see if the values are the same! This is evident through the use of the "==" comparison operator. Then, this value is being assigned to the water_dry boolean variable, which stores 1s for TRUE and 0s for FALSE.

5.2 For Loop

What if I'd like to tell my computer to keep going until a condition is met? For loops are the answer, and here's how they're used:

```
// Raccoon Age Sum
#include <iostream>
#include <string>
using namespace std;

int raccoon_ages[] = {2, 4, 9, 5, 1};
int result=0;

int main ()
{
  for ( int n=0 ; n<5 ; n++ )
  {
    cout << "Current Pass: " << n << '\n';
    result += raccoon_ages[n];
  }
  cout << "Total Age: " << result;
  return 0;
}
```

A couple of notes before I get into for loops, notice how the first line begins with "//". This tells the compiler that whatever is written after is a **comment** for the person who wrote it and it won't be run in the code.

Likewise, the

```
int main () {
...
return 0;
}
```

seems a bit strange, doesn't it? That's because this is a method! The "int" isn't describing a variable this time, but telling us what type of value this "method" is going to return. This is my favorite way of thinking why methods exist:

Let's say I'm teaching a little kid how to make a muffin. This requires a lot of steps! Cracking the eggs, mixing the batter, setting the oven temperature, etc. What if next week I'd like my small person to make a pie? I'd like to use that same "setting oven" knowledge again, and maybe mixing the batter. Wouldn't it be nice if I could just ask the kid to remember what steps they took? Methods allow us to do that. For now, we only need the **main** method. If you want to learn more, check out this resource: [6].

Back to for loops. Kind of a weird structure. Within the parenthesis there are three different segments blocked off by semicolons. The first segment gives us an opportunity to define a variable and give it an initial value. This is usually used to set a "looping" value (in our case the value is named "n") and is only called on the first pass. After the initial value is called, the program will take one pass through the for loop. On every pass after this, the loop will do whatever is in the third segment, and then check the criteria against what's in the second segment. If it evaluates to true, it will take another pass through the loop.

Back to our example! Here's the output of the code when we run it:

```
Current Pass: 0
Current Pass: 1
```

```
Current Pass: 2
Current Pass: 3
Current Pass: 4
Total Age: 21
```

What's happening? On the first pass, our "n" term is 0 and the loop is run. On our second pass, the n term is increased by 1, checked against our criteria, validated, and looped again. Next, the n term increases to 2, checked against the criteria, validated, etc. This goes until our last loop. On the final loop, our n term becomes 5, is checked against our criteria, shown **not** to be less than 5 as it equals 5, and the for loop is exited. To show that the final value is indeed five, let's rewrite our code to allow our "n" term to be printed outside the **scope** of our for loop.

The **scope** [7] is the area in which a variable (such as our "n" term) can be called. Because unnecessary variables take up the resources of the computer, there are times when the computer will erase them when it can. Looping variables are often found in this category of extra variables after their use, so we need to define our "n" term outside of the for loop if we want to call it later.

```
// Raccoon Age Sum
#include <iostream>
#include <string>
using namespace std;

int raccoon_ages[] = {2, 4, 9, 5, 1};
int n,result=0;

int main ()
{
  for ( n=0 ; n<5 ; n++ )
  {
    cout << "Current Pass: " << n << '\n';
    result += raccoon_ages[n];
  }
  cout << "Final n term: " << n << "\n";
  cout << "Total Age: " << result;
  return 0;
}
```

The result of this code shows us what the final value of our "n" term will be!

```
Current Pass: 0
Current Pass: 1
Current Pass: 2
Current Pass: 3
Current Pass: 4
Final n term: 5
Total Age: 21
```

Voila. Now you know for loops!

5.3 While Loop

Another loop you might like to use is the while loop, which runs what's inside indefinitely until a condition is met. An example will explain better than 10,000 words:

```
// While Loop Example
#include <iostream>
#include <string>
using namespace std;

int n=0;

int main ()
{
  while ( n<10 )
  {
    cout << "Current value: " << n << '\n';
    n += 2;
  }
  cout << "Final n term: " << n << "\n";
  return 0;
}
```

The output of this code looks like this:

```
Current value: 0
Current value: 2
Current value: 4
Current value: 6
Current value: 8
Final n term: 10
```

Awesome. As it's name implies, this code will only run **while** our condition (in this case $n < 10$) is met. To tell us a little bit more about the order of operations, let's start the code with n=1 and see what happens.

```
// While Loop Example
#include <iostream>
#include <string>
using namespace std;

int n=1;

int main ()
{
  while ( n<10 )
  {
    cout << "Current value: " << n << '\n';
    n += 2;
  }
  cout << "Final n term: " << n << "\n";
  return 0;
}
```

Output:

```
Current value: 1
Current value: 3
Current value: 5
Current value: 7
Current value: 9
Final n term: 11
```

Isn't the program supposed to stop at 10? Not exactly. Every time our value is about to enter the loop, it's evaluated against our condition (as defined in the parentheses next to the "while"). If the condition is met, it runs whatever is inside the loop. If not, it passes to the next chunk of code outside the loop. In our case, when n=9 the statement n<10 still evaluates as true so it runs through the loop. It's only when n=11 that this statement no longer evaluates as true, but by this point the value of our n term has already been modified beyond 10!

This resource [8] covers the information listed above in greater depth, as well as introducing the do-while loop, the go-to function, and jump statements. I know, it's so exciting!

6 Exercises

Q1 - Find the average value of the array:

```
float values[] = {4.2, 5.7, 2.2, 1.7, 9.1};
```

Q2 - Find the max term of the array:

```
float values[] = {4.5, 11.7, 2.25, 23.4, 9.1};
```

Q3 - Approximate the Value of PI. Hint: A circle's area is: $\pi * D$ and area of a square is $D * D$. If one were to randomly throw darts at the two shapes on a wall and compare the difference, maybe π might pop out...

If you'd like to delve deeper into the language, I recommend MIT OCW's course [9]. It'll give a good overview of a lot of the fundamental components of coding and go deeper into some of the topics covered in this module.

References

[1] Online cpp compiler and development environment. http://cpp.sh.

[2] Serial monitor guide. https://learn.adafruit.com/adafruit-arduino-lesson-5-the-serial-monitor?view=all.

[3] C++ variables. http://www.cplusplus.com/doc/tutorial/variables/.

[4] C++ arrays. http://www.cplusplus.com/doc/tutorial/arrays/.

[5] Comparison operators. http://en.cppreference.com/w/cpp/language/operator_comparison.

[6] Functions. http://www.cplusplus.com/doc/tutorial/functions/.

[7] Computer science scope. https://en.wikipedia.org/wiki/Scope_(computer_science).

[8] Control loop. http://www.cplusplus.com/doc/tutorial/control/.

[9] Kovacs G. Marrero J. Dunietz, J. Introduction to c, January 2011.