Introduction To SSH and C++

CPSC 457 Week 1

Jesse Francis

January 21, 2020

Outline

Welcome

Useful Information

Web Pages:

- https://teaching.jessef.xyz/w20/CPSC457/
- http://pages.cpsc.ucalgary.ca/~hudsonj/CPSC457W20/

Contact:

- Email: jesse.francis1@ucalgary.ca
- After tutorials

Assignments

All programming questions on assignments must run on the CPSC Linux servers or workstations.

Unfortunately, we are in a Windows lab, so we will need to use SSH to work on them during tutorials.

I will be marking your assignments so if you have any questions about them, feel free to ask me.

How To Use These Slides

- Throughout these slides, you will come across symbols similar to the one in the header of this slide.
- Any time there is a pdf or source file corresponding to the slide, one of these will be in the header.
- On this slide there is no file attached, but assuming there is and you are using a pdf reader that supports embedded files (`Adobe Acrobat`, `Okular`, `Firefox`), you will be able to save or open the file.

Up and Running

Using SSH

Follow these steps to SSH from the Windows lab computers into the CPSC Linux servers:

1. Open a Command Prompt
2. Type `ssh <cpsc username>@linux.cpsc.ucalgary.ca`
   - i.e., `ssh jesse.francis1@linux.cpsc.ucalgary.ca`
3. Press enter to execute the command.
   3.1 You may be told that the authenticity of the host can't be established and asked if you still want to connect. If that is the case, type `yes` and press enter.
4. Enter your password when prompted.
5. You should now have a command prompt logged into the Linux server. Commands executed here will execute on Linux.

Hello World in C++

Save the following code as `hello.cpp` somewhere in your network drive, `U://` in my case. This will make it available to the Linux server.

```cpp
#include <iostream>

int main() {
  std::cout << "Hello world!\n";
}
```

Compiling Code

To compile the code we just wrote, on the Linux server `cd` to the directory where you saved the code and run:

```
g++ hello.cpp -O2 -Wall -std=c++17 -o hello
```

Compiling Code

To compile the code we just wrote, on the Linux server `cd` to the directory where you saved the code and run:

```
g++ hello.cpp -O2 -Wall -std=c++17 -o hello
```

- `g++` is the command to run a `C++` compiler

Compiling Code

To compile the code we just wrote, on the Linux server `cd` to the directory where you saved the code and run:

```
g++ hello.cpp -O2 -Wall -std=c++17 -o hello
```

- `g++` is the command to run a `C++` compiler
- `hello.cpp` is the `C++` file we want to compile

Compiling Code

To compile the code we just wrote, on the Linux server `cd` to the directory where you saved the code and run:

```
g++ hello.cpp -O2 -Wall -std=c++17 -o hello
```

- `g++` is the command to run a `C++` compiler
- `hello.cpp` is the `C++` file we want to compile
- `-O2` is an option that tells `g++` to *Optimize* the assembly code resulting from compilation.
  There are additional optimization options.
  See the gnu online docs for a complete list.

Compiling Code

To compile the code we just wrote, on the Linux server `cd` to the directory where you saved the code and run:

```
g++ hello.cpp -O2 -Wall -std=c++17 -o hello
```

- `g++` is the command to run a `C++` compiler
- `hello.cpp` is the `C++` file we want to compile
- `-O2` is an option that tells `g++` to *Optimize* the assembly code resulting from compilation.
- `-Wall` tells `g++` to warn us about every potential error it can.

Compiling Code

To compile the code we just wrote, on the Linux server `cd` to the directory where you saved the code and run:

```
g++ hello.cpp -O2 -Wall -std=c++17 -o hello
```

- `g++` is the command to run a `C++` compiler
- `hello.cpp` is the `C++` file we want to compile
- `-O2` is an option that tells `g++` to *Optimize* the assembly code resulting from compilation.
- `-Wall` tells `g++` to warn us about every potential error it can.
- `-std=c++17` tells `g++` to use the `C++17` standard.
  Without this option, `g++` uses the `C++14` standard by default.
  See the gnu online docs for standards support in `g++`.
  Currently, `g++ 9.2.1` is installed on the Linux servers.

Compiling Code

To compile the code we just wrote, on the Linux server `cd` to the directory where you saved the code and run:

```
g++ hello.cpp -O2 -Wall -std=c++17 -o hello
```

- `g++` is the command to run a `C++` compiler
- `hello.cpp` is the `C++` file we want to compile
- `-O2` is an option that tells `g++` to *Optimize* the assembly code resulting from compilation.
- `-Wall` tells `g++` to warn us about every potential error it can.
- `-std=c++17` tells `g++` to use the `C++17` standard.
- `-o hello` tells `g++` to *output* an executable named `hello` in the current directory

The code can then be run with

```
./hello
```

Recap

You should now know how to

- SSH to Linux servers,
- write a simple `C++` program and save it in a directory that is shared with the Linux server, and
- compile and run the code you wrote in Windows on the Linux server via SSH.

If you need detailed information after this tutorial, a guide is available here

C++

Fundamental Types

`C++` has most of the fundamental/primitive types that you should be used to.

i.e., `bool`, `char`, `short`, `int`, `long`, `float`, `double`

Arrays

In modern `C++` there are two main types of arrays.

1. The `C` style:
   ```
   int example[5]{1, 2, 3, 4, 5};
   ```
2. The `C++` style:
   ```
   std::array<int, 5> example{1, 2, 3, 4, 5};
   ```

if Statements

The general syntax of an **if** statement is as follows:

```
if (condition) {
  statement
} else if (condition) {
  statement
} else (condition) {
  statement
}
```

The **else if** and **else** parts are optional.

Ternary/Conditional Operator

`C++` has a conditional operator. The general syntax is as follows:

```
condition ? statement_1 : statement_2
```

Ternary/Conditional Operator

C++ has a conditional operator. The general syntax is as follows:

`condition ? statement_1 : statement_2`

If `condition` is `true`, `statement_1` is executed.

If `condition` evaluates as `false`, `statement_2` is executed.

```cpp
#include <iostream>

int main() {
  int i = 6;
  std::cout << "i " << (i < 5 ? "<" : ">=") << " 5\n";
}
```

Looping in `C++`

There are two main looping methods defined in the `C++` standard.

- **`for` loops**
- **`while` loops**

Additionally, we also have recursion.

for Loops

There are two types of **for** loops:

1. The traditional **for** loop:

```
for(init statement, condition, iteration expression) {
  statement
}
```

2. The range-based **for** loop:

```
for(range declaration : range expression) {
  statement
}
```

Example of the Range-Based `for` Loop

```cpp
#include <array>
#include <iostream>

int main() {
  std::array<int, 5> arr{1, 2, 3, 4, 5};
  // we can drop the template arguments if using C++17:
  // std::array arr{1, 2, 3, 4, 5};

  for (int num : arr) {
    std::cout << num << " ";
  }
  std::cout << "\n";
}
```

This will print:

```
1 2 3 4 5
```

Example of the Traditional `for` Loop

```cpp
#include <iostream>

int main() {
  long acc{0};
  for (int i{20000000}; i > 0; i--) {
    acc += i;
  }
  std::cout << acc << std::endl;
}
```

while Loops

The general syntax for a while loop is

```
while (condition) {
  statement
}
```

Example of the `while` Loop

```cpp
#include <iostream>

int main() {
  long acc{0};
  int i{20000000};
  while (i > 0) {
    acc += i;
    i--;
  }
  std::cout << acc << std::endl;
}
```

Recursion

The final method we will look at when doing iteration is recursion.
Unlike `Java`, `C++` compilers implement tail call optimization.

This means that we can recurse infinitely without blowing the stack.

Note: You must compile with optimization higher than `-O1` in order to get tail call optimization.
i.e., `-O2` or `-O3`

Example of Recursion

Example:

```cpp
#include <iostream>

long sumIntegersUpTo(int i, long acc) {
  if (i == 0)
    return acc;
  return sumIntegersUpTo(i - 1, acc + i);
}
long sumIntegersUpTo(int i) { return sumIntegersUpTo(i, 0); }

int main() {
  long f = sumIntegersUpTo(20000000);

  std::cout << f << std::endl;
}
```

Structured Bindings

Given an array `{1, 2}`, you may want to assign `1` to a variable `x` and `2` to a variable `y`.

Using the C++17 standard, we can do this using a structured binding declaration as follows:

```
  int arr[] = {1, 2};
  auto [x, y] = arr;
```

Notice the use of **`auto`**. When using a structured binding declaration, we must use **`auto`** rather than **`int`** and the compiler should determine the correct type.

Structured Bindings (Cont'd)

Using `auto` allows us to do nice things, such as:

```cpp
std::tuple<int, char> tup{4, 'c'};
auto [x, y] = tup;
```

In this case, `4` is assigned to an `int` variable `x` and `'c'` is assigned to a `char` variable `y`.

`C` Strings Vs `C++ Strings`

Typically, `C` style strings are faster than `C++` strings. However, once you begin operating on the strings, `C++` strings have the opportunity to become more efficient.

`C` Strings Vs `C++ Strings` (Cont'd)

C style strings are stored as a `char` array terminated by a `'\0'` character:

```
char c_string[6] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

`C` Strings Vs `C++ Strings` (Cont'd)

C style strings are stored as a `char` array terminated by a `'\0'` character:

```
char c_string[6] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

C++ style strings are stored in an instance of the `std::string` class. This class has many functions and instance variables which take more memory.

```
std::string cpp_string{"hello"};
```

`C` Strings Vs `C++ Strings` (Cont'd)

Since `C` strings are just an array of characters, if we want to find out how long a `C` style string is, we have to traverse the array until we find the `'\0'` character. This operation takes $O(n)$ time (where n is the length of the string).

`C` Strings Vs `C++ Strings` (Cont'd)

Since `C` strings are just an array of characters, if we want to find out how long a `C` style string is, we have to traverse the array until we find the `'\0'` character. This operation takes $O(n)$ time (where n is the length of the string).

Since `C++` strings have instance variables and functions, we can just call `std::string::size()` or `std::string::length()` to get the length in $O(1)$ time.

i.e., calling `cpp_string.length()` returns `5`

Command Line Arguments

In our `hello.cpp` program, we were not able to take input. One way we can take input is via the command line.

Command Line Arguments

To do so, we modify our program as follows:

```cpp
#include <iostream>

int main(int argc, char const *argv[]) {
  if (argc > 1) {
    std::cout << "Hello " << argv[1] << "!\n";
  } else {
    std::cout << "Provide at least one argument.\n";
  }
}
```

Save it as `helloV2.cpp` and compile it with

```
g++ helloV2.cpp -O2 -Wall -o helloV2,
```

then run it with `./helloV2 Jesse` to output:

```
Hello Jesse!
```

Command Line Arguments (Cont'd)

***argv**	The array storing C-style strings representing the command line arguments.
argc	An `int` indicating the amount of arguments given on the command line.

Notice that when we get the first command line argument, we use `argv[1]`. This is because `argv[0]` contains the command used to call the program. In the previous example it would be `./helloV2`.

Iterating Over Command Line Arguments

We can use a `for` loop and `argc` to iterate over command line arguments:

```cpp
#include <iostream>

int main(int argc, char const *argv[]) {
  for (int i = 1; i < argc; i++) {
    std::cout << "Hello " << argv[i] << "!\n";
  }
}
```

Converting Strings To Integers

Suppose we want to get an integer from the command line. We can do so using

- the `C` library function, `atoi()`,
- the `C++` Standard Library function, `std::atoi()`, or
- the `C++` Standard Library function `std::stoi()`.

Converting Strings To Integers (Cont'd)

Since we are using C++, and `std::atoi` doesn't support exceptions, we will use `std::stoi`.

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

int main(int argc, char const *argv[]) {
  if (argc != 2) {
    return 0;
  }
  int count = std::stoi(argv[1]);
  std::cout << "Counting up...";
  for (int i = 1; i <= count; i++) {
    std::cout << " " << i;
  }
  std::cout << std::endl;
}
```

Converting Strings To Integers (Cont'd)

If we run `./argToInt 3`, we get

```
Counting up... 1 2 3
```

If we run `./argToInt five`, we get:

```
terminate called after throwing an instance of
'std::invalid_argument'
```

We can handle this exception.

Handling Exceptions

By updating our assignment of count as follows:

```
  int count{0};
  try {
    count = std::stoi(argv[1]);
  } catch (std::invalid_argument &e) {
    std::cout << "You must enter an int on the command line.\n";
    return 0;
  }
```

the program no longer crashes when we use `five` as an argument.