Intro to C++

Rachel Wood

2023-03-30

This portfolio will provide a brief introduction to the basics of C++. # Background C++ is a compiled programming language, which runs much more quickly than the more popular interpreted languages (e.g. Python and R). Lines of code in these languages are more compact than C++, however the simplicity gained comes at the cost of speed. The necessity for precision in C++ allows it to run much faster

It is widely portable, running on almost any machine and using the full power of the processor.

Storing and Compiling Files

We can create/edit C++ files in the linux terminal (in this example a file called `hello.cpp`) by using the following command

```
nano hello.cpp
```

While not explicitly necessary, it is convention to use the `.cpp` file extension for C++ files.

Then in our file we write the following function which prints the string `Hello World`:

```
#include <iostream>

int main()
{
    std::cout << "Hello from C++" << std::endl;

    return 0;
}
```

The next sections will explain the various commands use, for now we focus on how to create and run C++ files.

After saving this, we run the following in the terminal:

```
g++ hello.cpp -o hello
```

This tells the C++ compiler (called `g++`) to compile the contents of the file `hello.cpp` and store the executable program in a file called `hello`. Finally we can run this program by typing:

```
./hello
```

```
Hello from C++
```

Building Blocks of C++

This section focuses on the basic building blocks of every C++ program.

Syntax

One of the reasons C++ is harder to learn than python or R is that it requires specific syntax. Below is a list of miscellaneous things that are important to remember:

- All instructions to be executed need to end in a semicolon
- Comments are indicated by a double slash `//` for a single line. Multiline comments begin with `/*` and end with `*/`
- Any library used in the code must be included in the header as: `#include <library name>`
- A function or command from an external library must be used with a reference to the library: `<library>::<command>`
- We can remove the need for the previous point by including `using namespace <library>` in the header

Variables

Any variable used in C++ has to be declared along with its type. For example, if we are declaring an integer variable called `a` with initial value 4, we would include the following line in our program:

```
int a = 4;
```

Some other common variable types are:

- `float`: A floating point number
- `double`: A floating point number with double the precision of the `float` type.
- `bool`: A logical variable taking values true or false
- `std::string`: A string

We often also want to print a string or the value of a variable. For this we use the `std::cout` and `std::endl` commands. The first line below shows how these are used for a string, the second for a variable to print and the third for if we want to print multiple things in the same line:

```
std::cout << "a is" << std::endl;
std::cout << a << std::endl;
std::count << "a is" << a << std::endl;
```

Conditions

We can include conditional statements in C++ as with many other programming language, using `if`, `else if` and `else`:

```
if (statement 1){
    //code to be executed if statement 1 is true
} else if (statement 2){
    //code to be executed if statement 2 is true but statement 1 is not
} else {
    //code to be executed if statement 1 and statement 2 are not true
}
```

Loops

Again, C++ is very similar to other languages, a minimal example of a loop is given below:

```
for (int i=1; i <=10, i++){
    std::cout << i << std::endl;
}
```

There are always three terms separated by a semicolon in the brackets before the `for`:

- The first term is the state of a counter variable at the beginning of the loop. In this case we are initialising the variable `i` to have a starting value of `1`.
- The second term gives the condition under which program should keep looping through the code. In this case, we say we want the code inside the loop to be evaluated as long as `i = 10`.
- The final term specifies how we change the counter variable at each iteration of the loop. In this example, we increase the counter by one: `i = i+1` which can be shortened to `i =+1` or `i++`.

Functions

The general syntax for C++ functions is:

```
<type of return_value> function_name(<type of argument1> argument1, ...){
    //function code

    return return_value;
}
```

with the dots representing other arguments if they are needed.

Any C++ program must include a function called `main`, this contains the code which runs when the program starts. It will often call other functions, for example the following file contains the `sum()` function, which is then called in `main()`:

```
#include <iostream>

int sum(int a, int b){
    int c = a + b;
    return c;
}

int main(){
    std::cout << "1 + 2" << sum(1,2) << std:endl;

    return 0;
}
```

An important note is that a function must be defined before it is called.

Variable Type Conversion and Scopes

The previous section briefly introduces how to declare variables with their type. This section will provide an overview of the typing and scoping rules in C++, which are much stricter than the rules in say, Python or R.

Types

In C++, once a variable has been defined with a type, it cannot take a value of a different type. It also cannot be declared twice. For example, the following is not valid:

```cpp
#include <string>

int main(){
    int a = 42;

    a = "dog";

    std::string a = "dog";

    return 0;
}
```

and produces the following error:

```
g++ broken_vars.cpp -o broken_vars
```

```
broken_vars.cpp: In function ‘int main()’:
broken_vars.cpp:6:9: error: invalid conversion from ‘const char*’ to ‘int’ [-fpermissive]
     6 |     a = "dog";
       |         ^~~~~
       |         |
       |         const char*
broken_vars.cpp:8:17: error: conflicting declaration ‘std::string a’
     8 |     std::string a = "dog";
       |                 ^
broken_vars.cpp:4:9: note: previous declaration as ‘int a’
     4 |     int a = 42;
       |         ^
```

For some combinations of variable types however, we can convert the data itself to be held in a variable of a different type. There are two different ways this can go:

- In the simple case, we can convert data to a type which supports a wider range of values than the original (e.g integer to float or float to double)
- It is also possible to convert data in the other direction, but this can lead to data loss.

Below is an example of how we can do this:

```cpp
#include <iostream>
int main(){
    int a = 50;
    float b = 3.56;
    double c = 4e50;

    // these are simple conversions with no risk of data loss:

    float a1 = a;
    double b1 = b;

    // these are conversions with a risk of losing data

    int b2 = b;
    float c2 = c;

    return 0;
}
```

Scopes

In C++ a variable is defined for the area within the area of the program within the set of brackets inside which the variable is defined. We call this the variable's scope. Outside this scope the variable is undefined and cannot be referenced or accessed.

For example, a variable `b` defined inside a loop as follows cannot be accessed outside the loop:

```cpp
for (int i; i < n ; i++) {
    int b = 25;
}
```

Further, the loop counter `i` is also not accessible outside the loop as it is declared in the initialiser of the `for` loop.

It is however possible to declare a variable globally in a file by including the declaration outside of any function. This is due to the file being a scope as well, however this is generally not recommended.

These scoping rules mean that we can define variables of the same name in different scopes as these are considered different variables.

This even applies for nested scopes, for example:

```cpp
#include <iostream>

int main()
    {
    int a = 10;

    for (int i=0; i<2; ++i)
    {
        std::string a = "hello";
        std::cout << "a = " << a << std::endl;
    }

    std::cout << "a = " << a << std::endl;

    return 0;
}
```

gives us:

```
g++ scopes.cpp -o scopes
./scopes
```

```
a = hello
a = hello
a = 10
```

The `auto` Keyword

To save time while coding, we can use the `auto` keyword when declaring variables instead of the type. This is only useful for variables whose type is obvious from the initial value assigned to it. As before, the variable keeps the same type for it's scope and can't be redefined.

For example if we declare

```
auto a = sqrt(2);
auto b = sum(1,2);
auto c = 3.54;
```

`a` will be declared as a `double`, `b` will be `int` and `c` will be of type `float`.

Vectors

We can include vectors in our C++ file using the `<vector>` header file and declaring a vector `v` using the format `std::vector<type> v;` where the type can be any type for atomic variables. This means any vector can only contain one fixed type of variable. Some useful commands related to vectors include:

`.push_back()`: appends items to the end of the vector `.size()`: returns the size of a vector `[]`: as in R and Python, square brackets can be used to access elements of the vector `for (auto x : v)`: this allows you to loop over elements in the vector `v` `std::vector< std::vector<int> >`: creates a matrix through nested vectors. We can similarly use `.push_back` to add columns or rows.

The following piece of code shows how we could create a matrix and set its value and print it:

```
#include <iostream>
#include <vector>

int main()
{
    std::vector< std::vector<int> > m;

    // create a 3x3 matrix
    for (int i=1; i<=3; ++i)
    {
        //create space for a row
        std::vector<int> row;

        for (int j=1; j<=3; ++j)
        {
            row.push_back( i * j );
        }

        //now save the row in the matrix
        m.push_back(row);
    }

    //loop over elements of m (columns)
    for (auto x : m)
    {
        //loop other elements of the columns
        for (auto y : x){
              std::cout << y << " ";
          }
          std::cout << std::endl;
    }
    return 0;
}
```

We can see the result when we run this:

```
g++ vectors.cpp -o vectors
./vectors
```

```
1 2 3
2 4 6
3 6 9
```