C++ basics: Lecture 5

Code the Universe Foundation Presented by: Mikdore

7th June 2020

1 Quick recap

1.1 The language

The C++ language is a modern, open, fast, object oriented, compiled and *standardized* language. It was developed in 1980's by Bjarne Stroustrup, originally called "C with classes", however it has evolved beyond that. C++ today is maintained by an ISO committee, which releases a new *standard* about every 3 years. There are currently 4 standards, with C++20's release nearing, 5. Those releases are:
C++98 C++11 C++14 C++17 (C++20) In our course we will be using C++17. The standard is your holy book. It specifies everything there is to know about the language. It specifies what the result of each expression is, what keywords do, how to define a class, and what the standard library contains, and how what it contains works. *What is not in the standard cannot be assumed to work, therefor all your code should follow the standard.*
Any time you want to lookup any functionality or part of the standard, your go-to place should be `cppreference.com` If you feel the need to look directly at what the standard says (a technical specification, hard to read 1700 pages), you should know, that the standard isn't technically public, it has to be bought. But, the final draft is, and the difference to the release versions are minor, and the technical specification parts are the same.

1.2 Why we do and do not use C++

C++ is a great language. It is very fast and established, but that comes at a cost. The speed part comes at the cost of two things, firstly, code portability. You can, and should write as much code as you can to be cross platform, however the bigger a project becomes, the harder that gets. And when it comes to stuff like GUI's, Audio, Networking, etc. there is no cross platform solution. You can use libraries which support multiple platforms, and that is usually enough, but those libraries have to be in part written for each platform they support differently. The second disadvantage is the how much code you have to write, and how hard it is to get right. C++ is a low level language. We have the power to manipulate the hardware in any way

1.3 The making of a C++ program

C++'s code is not run by itself, to make our program, we need a compiler. The compiler creates so-called object files. An object file includes assembly code and function, class and variable declarations, but an object file cannot be run by itself. For that there's the linker. A linker takes object files, and links them, i.e. connects all their functionality together. C and C++ object files also get linked with the standard C and C++ libraries respectively.

If you don't use any tools, you will have to invoke the compiler yourself. The two most popular compilers which you should know the command syntax of, are Clang and GCC (If you use Visual Studio, it'll do the work for you, same as CMake). The basic syntax is identical. To compile your main.cpp and create an executable, write:

```
g++ main.cpp -o main.exe
clang++ main.cpp -o main.exe
```

1.4 Code files

C++ code goes into a text file, whose endings vary, but by far the most used one is *.cpp*
The file's structure looks like this:

```
#include <...>
...
int main()
{
    //code here
}
```

The *int main()* function is important. The only code which will be executed is the one we put into the main() function. That is because the standard library needs to set up some stuff stuff before our program can execute. After it has done that, it'll call main().
Try it yourself: make a program without a main and compile it. You will get a linking error.

All code inside functions is separated into lines, and code blocks. Lines always end with a semicolon; and a code block is just lines surrounded by brackets.

2 Variables, data types and literals

Literals are symbols in your program which denote a certain data type, and a certain value at once.

```
5; //int literal
```

```
'a'; //char literal, value 65
5.0; //double literal
4.2f; //float literal
"Hello there"; //!!! this literal is of type const char[12], but it is
    implicitly convertible to string
```

A variable is a name we give to a place in memory, which holds a value of a specific type. To define a variable, your code will look like this:

```
type name;
```

Where type is any type, be it int, double, std::string, std::vector<int> or *any other default or user defined type*. However defining a primitive, i.e. int, double, float, etc. merely tells the compiler such a variable exists, the variables value is *undefined*. *Undefined behaviour* in C++ is bad. It means the Standard lets the *implementation* i.e. the C++ compiler and the standard library which is being linked with do what they want, which may be setting an int to zero, or doing nothing. In the latter case the int will usually hold some garbage value in RAM. Therefor, you should always initialize a variable (this does not necessarily apply for objects, as those may have a default constructor, which may do all the work needed to meaningfully use it). To define a variable, you can use one of four options:

```
int x = { 0 };
int x { 0 }; //there is some consensus that this is the best one
int x = 0;
int x(0);
```

Each of them having their own advantages and disadvantages, use whatever you want, but you should aspire for consistency.

2.1 Arrays

You can define an array of any type, to do that, the syntax is as follows:

```
type name[size];
```

Size has to be a constant integral expression, that is either a literal or a constexpr variable (more on that in later lectures). An array's values can be set two ways:

```
int a[2] = {1,2};
int b[2];
b[0] = 1;
b[1] = 2;
```

Using operator [index], we can access the value at that index. If the index is bigger than the size of the array, we get UB (undefined behaviour). Under some

very special circumstances, index can be negative. We are going to revisit arrays in a future lesson, and cover them in-depth.

2.2 Type alias's, casts, and typedefs

A typedef and alias have the same basic functionality, which is to define a name for a type, for the purposes of encapsulations, or just ease.

```
//typedef syntax: typedef <type> <name>
//Same as in C
typedef int integer;
typedef double rational;
//C-ism:
typedef struct {int a; int b;} Pair;

//alias syntax: using <name> = <type>
using integer = int;
using rational = double;
//You should never use this, as it is utterly unnecessary, and confusing
using Pair = struct {int a; int b;};
/*
In C++, you'd just declare a struct
*/
struct Pair
{
    int a;
    int b;
}; //The semicolon is required here
```

To cast a type to another, you use a cast expression. The most used and most basic one is static_cast, and it is the only one we're mention as of now.

```
double d = 1;
int i = static_cast<int>(d);
```

The above code casts the variable d of type double *explicitly* to an int. (Side note: don't use (int) as a cast, it is bad).

3 Scopes and namespaces

Everyone knows the meaning of the word scope as it applies to a gun. A soldier for example will have a scope on his rifle to see the enemy clearly over a long range. But as he looks through the scope, he cannot see things near him. The same principle of restricting vision, or rather *access* applies to scopes in computer programming. There is always one scope, the global scope. The global scope is everything in a file not inside a function, class/struct or namespace. Every code block, functon definition, class/struct definition, loop body, in

short, every brackets start a new scope. The important thing about scopes is, everything outside a scope X can't see the contents of X. X can see everything in its parent scope (excluding other scopes). Every variable which is inside the same scope as we operate in is called a *local*. That brings us to the second, even more important fact, that local variables get destroyed (We will examine the meaning of this statement in another lecture) when the program leaves the current scope. Namespaces are code blocks declared as follow:

```
namespace our_namespace
{

}
```

Namespaces contain function/class/struct/variable declarations and definitions. To access a *member* of a namespace, you have to prefix that member with ¡namespacename¿::, for example, the vector class is part of the std (standard) namespace, we access it by using:

```
std::vector<int> v;
```

Namespaces are important, because they allow us not to break ODR (more on that next lecture). For example, consider this project structure: main.cpp:

```
#include <iostream>
#include "library.h"
int max(int a, int b)
{
  return a > b ? a : b;
}
int main()
{
    int a, b;
    //do stuff using our library
    //print result
    std::cout << "The greatest number is: " << max(a,b) << "\n";
}
```

library.h:

```
...
int max(int a, int b)
{
  return a > b ? a : b;
}
...
```

Trying to compile this (for example with "clang++ library.cpp main.cpp -o main", will yield something akin to this:

```
main.cpp:(.text+0x0): multiple definition of `max(int, int)'
/tmp/library-17cb1f.o:library.cpp:(.text+0x0): first defined here
clang: error: linker command failed with exit code 1
```

And we didn't even know of the existence of the other max! The solution is for the library to wrap everything inside a namespace, for example

```
namespace library
{
    ...
}
```

Then in our main we don't have to worry about such stuff. This is a trivial example, but consider two almost identical functions. One typo could call a wholly different function than intended on accident! You can now see why *using namespace std;* at the global scope is a bad idea. The cost of using namespace std; is an unclear program, which may be broken. The cost of not using namespace std; is you have to write "std::" each time you use a standard function, which is basically zero. There are also anonymous namespaces, which have no name, and whose members can be accessed without the ¡name¿:: prefix, but they are only visible inside the file they are defined in. **Tip:** If you have long namespaces, like for example "boost::multiprecision::cppint", you can use aliases to declare

```
namespace bigint = boost::multiprecision::cpp_int;
```

And then just use bigint::

4 Control structures

Control structures allow us to redirect the flow of our program. They are if/else statements and loops

4.1 If and else (and switch)

An if statement has the following syntax:

```
if(expression) /* do stuff in here*/ ;
//One line:
if(is_invalid(input)) std::exit(1);
if(is_invalid(input))
    std::exit(1); //Equivalent to the first, but it is ugly
if(is_invalid(input))
std::exit(1);     //Equivalent to the first and second, but very
    confusing and ugly
//Multi-line, we need to use a block:
if(is_invalid(input))
```

```
{
    std::cout << "invalid input, terminating \n";
    std::exit(1);
}
```

Where expression is *convertible* to a boolean. Arithmetic data types can be converted to bool implicitly, if they are >0 they represent *true*, if they are 0 or less, they represent *false*. By appending an else after an if statement, we can branch our execution flow:

```
//Else can be in the one-line form too
if(input == 0)
{
    std::cout << "You entered zero \n";
} else
{
    std::cout << "You entered " << input << "\n";
}
```

By putting another if after our else, we can branch our code indefinitely:

```
if(input == 0)
{
    std::cout << "You entered zero \n";
} else if(input == 1)
{
    std::cout << "You entered one \n";
} else if(input == 2)
{
    std::cout << "You entered two \n";
} else
{
    std::cout << "You entered " << input << "\n";
}
```

Boy, that's a lot of typing. Why can't there be a way to do the same thing faster? Well, there is, it is called a *switch* statement. A switch has the form of:

```
switch(expression)
{
    case X: /* do stuff*/
            break;
    case Y: /* do stuff*/
            break;
    case Z: /* do stuff*/
            break;
    default: break;
}
```

"Stuff" can be one or multiple lines, or a block. The *break* is very important. The way switch works, is that it *jumps* to the case which matches the expression, and the blindly executes everything after that. The break stops the execution inside the switch after it's done with the case specific code. We can however use that fact to our advantage, to execute the same code for multiple cases:

```
switch(expression)
{
    case X:
    case Y:
    case Z:
           /*Do the same stuff for all*/
           break;
}
```

There is one special label, default. The switch will jump to default if there is no match. Defaults can be omitted. Knowing all this, we can put it all together to simplify our input if/else:

```
switch(input)
{
    case 0: std::cout << "You entered zero \n";
           break;
    case 1: {
               std::cout << "You entered one \n";
           }
           break;
    case 2: {
               std::cout << "You entered two \n";
               break;
           }
    default: std::cout << "You entered " << input << "\n";
}
```

4.2 Loops

In C++ there are 2, technically 3 kinds of loops. *While*, *do-while* and *for*. The syntax of while is:

```
while(condition)
{
    //do stuff
}
```

While loops are most commonly used if you don't know how many *iterations* you need to execute. One example could be, that "While the input is wrong, ask the user to re-enter their input".
Do-while loops have the following syntax:

```
do
{
    //Do stuff
} while (condition); //the semicolon has to be here
```

The difference to a while loop is, that a do while will always execute the code inside it once. It first starts checking the condition going into the 2nd iteration. A do-while is semantically the same as:

```
//code
while(condition}
{
    //same code
}
```

The primary use of do while therefor is not having to repeat the same code twice. Which for one makes the overall code cleaner, and second, if the code is big (thousands of lines), it will reduce the compiled code's binary file size, which can actually improve performance. The most used loop is a for loop. It is primarily used to execute a known number of iterations, and/or when you need a counter of how many iterations you are in. The syntax is as follows:

```
for(<initial expression>; <condition>; <post iteration operation>)
{
    //Do stuff
}
```

The initial expression will get executed once, the condition works the same as a while loop's, and the post iteration operation is an expression which will get executed each time the loop finishes an iteration. To make it clearer here's an classic example involving arrays:

```
int arr[5] = {1,2,3,4,5};
for(int i = 0; i < 5; i++) //Note that array indices are zero based.
{
    std::cout << "Position: " << i << " value: " << arr[i] << "\n";
}
/*
    This will print:
        Position: 0 value: 1
        Position: 1 value: 2
        Position: 2 value: 3
        Position: 3 value: 4
```

```
        Position: 4 value: 5
*/
```

Each of the three expressions in the (brackets) of a for loop can be empty. Every type of loop can theoretically be implemented in terms of another loop.

4.3 Jump statements

In C++ you can use these keywords to *jump*: break; continue; return; goto;

4.3.1 Break

You can only use break inside a switch or a loop. the statement *break;* will jump out of the current scope, destroying all locals.

4.3.2 Continue

Continue can only be used withing loops, the statement *continue* will skip the rest of the code in the loop, and begin the next iteration.

4.3.3 Return

A return statement will end the execution of a function, destroying all locals, and will return a value of return-type, unless the function is void.

4.3.4 Goto

The syntax of goto is as follows:

```
int i = 0;
label:
if(i >= 4) goto out;
std::cout << i << "\n";
i++;
goto label;
out:
```

That code will print: 0
1
2
3
Goto is safe in a way, because if you jump out of a scope, all locals will be destroyed. But it leads to spaghetti code, and has only one good use (which can more or less be replaced with a lambda), and that is jumping out of nested loops:

```
for(...)
{
    for(...)
    {
        for(...)
        {
            if(x == y) goto out;
        }
    }
}
out:
```

That syntax is cleaner and faster than checking for some condition on each level. It also avoids x having to accessible to inside the scope of the first loop.

5 Functions

Functions are handy way to reuse and encapsulate code. The basic syntax of a function is:

```
type function_name(type argument1, ...)
{

}
```

Type can be a return type, which means anytime the function is called at some point a "return object;" statement will be executed, where object is of the same type as the function. One exception is type void, where the function is not required to have a return statement, if it has one, in the form of "return;" The execution will resume at the point the function was called. Functions can call other functions, including itself. The computer however will have to store the address in RAM where to return to upon a function's end. Therefor too many function calls will yield a stack overflow, and the program will crash.

6 References

Variables are stored in memory. A reference to a variable *refers* to the same place in memory as the variable. A reference is syntactically the same as a normal variable, but it has to be assigned to a already existing variable when defined. Modifying a reference will modify the original variable too.

```
//General case:
type&  reference = variable;
//Specific case:
int i = 5;
int& reference = i;
```

```
//Modifying 'reference' is modifying 'i' as well
reference = 1;
std::cout << i; //Will print 1
```

A common use case of references is modifying a variable:

```
#include <iostream>
void set5(int& i)
{
    i = 5;
}
int main()
{
    int i = 1;
    set5(i);
    std::cout << i; //Will print 5
}
```

Another common use of references is to avoid copying objects. A const reference is a reference to an object, but it is "read only", meaning you can't modify the object. Consider these two function:

```
int sum_copy(std::string str)
{
  int sum = 0;
  for(char c : str)
  {
    sum += static_cast<int>(c); //(int)c;
  }
  return sum;
}
int sum_ref(const std::string& str)
{
  int sum = 0;
  for(char c : str)
  {
    sum += static_cast<int>(c); //(int)c;
  }
  return sum;
}
```

Both of them sum up the integer values of the characters of a string, but the copy version copies the argument by default, which is an unneeded overhead. This can mean that the reference version can be multiple times faster for some inputs! Moreover, since the reference version is a const reference, the string object cannot be modified, and some types cannot be copied! **Takeaway:** prefer const references to pass-by-value whenever possible