**Beginning C++ Programming by Frank J. Mitropoulos**My Course Notes

Troubleshooting

* CMake/NMake errors:
* Ensure that your antivirus isn’t blocking the build process by quarantining files.
* Restart computer.
* “Error: could not load cache”:
* Tools > CMake > Reload CMake Project.

Introduction

* There are multiple versions of C++: C++98, C++03, **C++11**, **C++14**, and **C++17**. The digits represent the year that version was released. The former two versions are referred to as classic C++, while the latter three are referred to as modern C++.
* C++98 was the first official standard. C++11 added many new features to the language. The other versions mainly corrected issues with the language or simplified pre-existing features.
* The process to create run an application is as follows:
* The developer inputs C++ code into header (.h) and source (.cpp) files in an IDE.
* The pre-processor looks for pre-processor directives such as *#include* and processes them.
* The compiler converts the high-level C++ code into low-level machine/binary/object code (.obj).
* The linker combines our code with other libraries and outputs an executable (.exe).
* The course will use CodeLite since it’s free, but the instructor stated that CLion by JetBrains is his IDE of choice. I have opted to use CLion as I have access to it and think highly of JetBrains.

Installation and Setup

* You can execute C++ code through an IDE, the CLI (command line interface), or via a website such as ‘repl.it’.

Curriculum Overview

* No notes taken.

Getting Started

* Code completion aids the developer by predicting what they will input and suggesting it to save time.
* Pre-processor directives don’t end in a semi-colon.
* *cout* is tied to the console and is used to output data.
* *<<* is the insertion operator that outputs the following data.
* Text between quotation marks represents a string literal: e.g. “Hello world!”.
* Statements end in a semicolon.
* *‘return 0;*’ in *main()* to indicate that there weren’t any problems.
* *cin* is also tied to the console and is used to input data from the user.
* *>>* is the extraction operator that stores the input.
* **Variables** store data.
* To declare a variable state its type and give it a name, e.g. ‘*int favourite\_number;*’.
* *#include <iostream>* includes the input/output library where *cout* and *cin* are defined.
* *endl* prints a new line and flushes the buffer.
* To build means to compile it and link it. This results in object files. The build process saves time by only building the files that it has to.
* The clean process removes the object files, but then you must build your program again.
* **Compiler errors** occur when the code doesn’t follow programming rules. It does this by identifying syntax and semantic errors.
* **Syntax error** refers to when the structure of the code is incorrect, e.g. ‘*cout << “Errors << endl*’ in this case the trailing quotation character is missing.
* **Semantic error** refers to when the structure is correct, but the code is undefined, e.g. ‘*int a = b + c’* if *a* and *b* are *int*s and *c* is a person then it may not make sense to add them.
* Making one error will lead the compiler to detect many errors. So fixing one error will resolve many compiler errors.
* **Compiler warnings** occur when code can be compiled, but has potential issues, e.g. printing an uninitialized variable ‘*int data; cout << data;*’.
* Both warnings and errors should be avoided whenever possible.
* **Linker errors** occur when libraries or object files are missing.
* **Runtime errors** occur when the program is running, e.g. dividing by zero, file not found, out of memory, etc. These can crash the program. To crash means that the program ended abruptly. Exception handling is used to deal with runtime errors.
* **Logic errors** occur when the code is technically correct, but the logic behind it is incorrect thus allowing the program to do something it shouldn’t do.

Structure of a C++ Program

* **Keywords** are reserved terms that hold special meaning in programming languages. Their meaning can’t be redefined in any way, e.g. *return, int,* etc. C++ has around 90.
* **Identifiers** are names given by the programmer, e.g. *main, include, my\_variable, cout*, etc.
* **Operators**accept one or more operands and perform an action with them, e.g. *+, -, <<, %, /, ^, &&, ::,* etc.
* **Pre-processor directives**tell the pre-processor program what to do, e.g. *#include <iostream>* tells the pre-processor to place the contents of that source file in its place. It also replaces the comments with a space.
* **Comments** allow the programmer to describe meaning or explain themselves next to the code. *//* is used to place a single-line comment, everything on that line is ignored by the compiler. */\* … \*/* is used to place a multi-line comment, everything in between is ignored.
* Ideally code is self-documenting and easy to read. Avoid unnecessary comments as it makes the code harder to read.
* All C++ programs must have one *main* function. It’s where execution begins in the program. Returning 0 indicates that the program terminated successfully.
* There are two acceptable function signatures for *main*:

|  |  |
| --- | --- |
| int main() {  return 0;  }  Program.exe | int main(int argc, char \*argv[]) {  return 0;  }  Program.exe argument1, argument2 |

The first signature is for when the program doesn’t accept any arguments, the second signature is for when the program does.

* *argc* counts the number of arguments provided, *argv* (argument vector) contains the value of each argument. These can be provided from the command-line. A vector refers to data in a one dimensional array.
* *main* is a function that is called by the operating system. A **function** is a name that refers to a block of code.
* **Namespaces** allow grouping code to avoid naming conflicts.
* Example: A library imported from a company might have their own definition of *cout*, however with the use of namespaces you’ll be able to specify exactly which one you want to use: *std::cout* or *company::cout*.
* *std* is the name of the standard namespace which contains code from the STL.
* If you don’t foresee any naming conflicts, then you can use a **usingdirective**, e.g. ‘*using namespace std;*’ now you won’t have to type ‘*std::*’ before using identifiers declared in the *std* namespace.
* This isn’t recommended in large programs due to the potential increase in naming conflicts. Instead use a **using declaration** in which you specify the identifier directly, e.g. *using std::cout*.
* Using directives and declarations can be defined at global scope or block scope.

Variables and Constants

Arrays and Vectors

Statements and Operators

Controlling Program Flow

Characters and Strings

Functions

Pointers and References

OOP - Classes and Objects

Operator Overloading

Inheritance

Polymorphism

Smart Pointers

Exception Handling

I/O and Streams

The Standard Template Library (STL)

Bonus Material and Source Code