Introductory Robot Programming

ENPM809Y

Lecture 02 – Structure of a C++ Program

Zeid Kootbally
zeidk@umd.edu

Fall 2020

Overview I

Overview II

Overview III

Elements of Programming

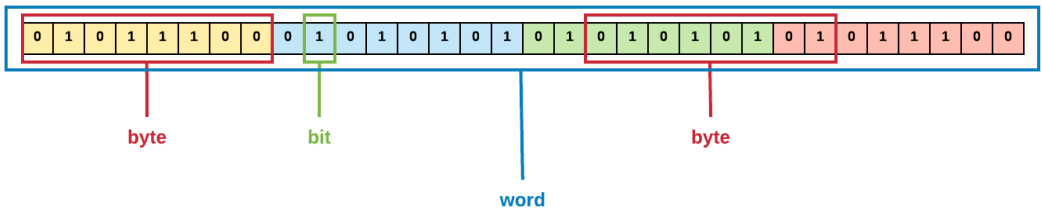
- Variables.
  - Store information to be referenced and manipulated in a computer program.
- Input/Output.
  - Input/Output elements allow the program to interact with external entities.
- Comments and documentation.
- Flow Control.
  - Iterations.
  - Conditionals.
- Subroutines/Functions.

Computer Memory

- Basic architecture of a typical computer system:
  - Disk used to store your program.
  - CPU (Central Processing Unit) that processes data (read and write).
  - RAM (Random-access Memory) is a contiguous block of storage used by the computer to store.
  - Bus that allows the movement of data between the CPU and the memory.
- Source code written with a text editor (or IDE) is stored on disk.
- During execution, code and variables are loaded into memory (RAM) and then executed by the CPU.

Computer Memory | Bits, Bytes, and Words

- The most fundamental unit of computer memory is the bit.
- A collection of 8 bits is called a byte.
- A collection of 4 bytes, or 32 bits, is called a word.



- Each individual data value in a data set is usually stored using one or more bytes of memory.

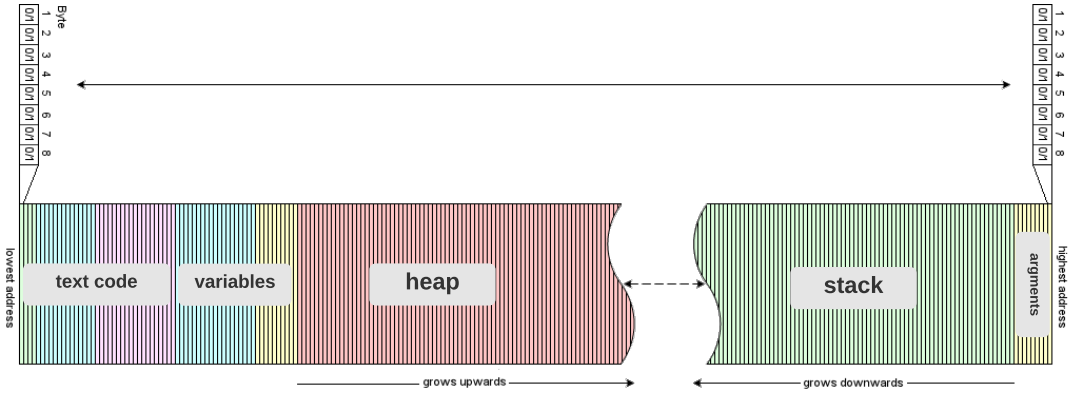
Computer Memory | Bits, Bytes, and Words

- 1 byte (8 bits) can have a value ranging from 0 (0000 0000) to 255 (1111 1111).
- This is $2^8 = 256$ different combinations of bits forming a single byte.
- 8-bit examples:
  - 0100 1101 = $2^6 + 2^3 + 2^2 + 2^0 = 64 + 8 + 4 + 1 = 77$
  - 0110 1001 = $2^6 + 2^5 + 2^3 + 2^0 = 64 + 32 + 8 + 1 = 105$
- The same reasoning can be applied for 16 bits, 32 bits, 64 bits, etc.

Computer Memory | Storing Data in Memory

- Your computer memory can be seen as a series of cubbyholes (or **memory location**).
- Each cubbyhole is one of many, many such holes all lined up.
- Each memory location is numbered sequentially.
  - These numbers are **memory addresses**.

Computer Memory | Storing Data in Memory



Variables

- Store information to be referenced and manipulated in a computer program.
- Provide a way of labeling data with a descriptive name, so our programs can be understood more clearly by the reader and ourselves.
- "Containers that hold information":
  - Label and store data in memory.
  - Use this data throughout your program by referencing its label.

```
int age = 20;
std::cout << age << std::endl;
```

Variables

- Imagine if we programmed using specific memory locations, e.g., "move the number 20 to the memory address 4359".
  - This is not efficient and error prone: If the address 4359 is already occupied, you will not be able to "move 20 to 4359".
- The best way to achieve this is to bind the number 20 with a name (e.g., `age`): "move 20 to `age`".
- The compiler now knows that 20 is bound to `age` and it figures out a free location to store 20 through `age`.
- Any time you want to access this 20 either to read or modify it, you will just need to use `age`, and not the address where 20 is stored.

Variables

- A variable reserves one or more **memory locations** in which you can store values.
- When defining a variable, you must tell the compiler about the type of this variable.
  - The type of the variable is used by the compiler to know how much memory is needed to store the variable and what kind of value you want to store in the variable.
- You MUST declare a variable before you can use it.

```
int main(){
    num = 21;//compiler error, variable hasn't been declared
    return 0;
}
```

Variables | Primitive Datatypes

- All variables use data-type during declaration to restrict the type of data to be stored.
- Every data type requires different amount of memory.
- Data types in C++ are mainly divided into two types:
  - Primitive data types:
    - Built-in C++ data types which can be used directly by the user to declare variables.
  - Abstract or user defined data types:
    - Data types defined by the user (e.g., C++ classes and structures).

Variables | Primitive Datatypes

- Fundamental types implemented directly by the C++ language.
  - Void type: Type with an empty set of values.
  - Boolean type: Used to represent `true` and `false`. Zero is `false` and a non-zero value is `true`.
  - Integer types: Types used to represent whole numbers (non-decimal values).
  - Character types: Types used to represent a character.
  - Floating point types: Types used to represent real numbers (with decimal point).
- Note: A separate PDF file on data types will be uploaded on canvas.

Variables | Declaring a Variable

- C++ is a strongly-typed language, and requires every variable to be declared with its type before its first use.
- This informs the compiler the size to reserve in memory for the variable and how to interpret its value.
- The syntax to declare a new variable in C++ is straightforward, we simply write the **type** followed by the variable **name** (i.e., its identifier).
  - `int var_i;//--variable of type int with the identifier var_i`
  - `double var_d;//--variable of type double with the identifier var_d`
  - These two variables can now be used in the scope[1] they were declared.

[1] A scope is defined by `{}`.

Variables | Declaring a Variable

- If declaring more than one variable of the same type, they can all be declared in a single statement.

```
int main(){
    int var_i1, var_i2, var_i3;//-- 3 variables of the same type

    return 0;
}
```

- This has exactly the same meaning as:

```
int main(){
    int var_i1;
    int var_i2;
    int var_i3;

    return 0;
}
```

Variables | Variable Identifiers

- Variable identifiers:
  - Can only contain letters, numbers, and underscores.
  - MUST begin with a letter or an underscore.
  - MUST be different from C++ keywords.
  - Are case sensitive:
    - `int Var;` and `int var;` are two different variables.
    - `int BREAK;` is a valid variable declaration because `BREAK` ≠ `break` (`break` is a C++ keyword).
- See Google C++ Style Guide for variable naming.

Variables | Variable Identifiers

- You cannot declare variables with the same identifier in the same scope.

```
int main(){
    double x_point;//--ok
    double x_point;//--compiler error, variable already defined
    return 0;
}
```

Variables | Variable Identifiers

- The following variable declarations are valid.

```
int main(){
    double x_point;  //--ok
    double X_point;  //--ok
    double _x_point; //--ok
    return 0;
}
```

Variables | Variable Identifiers | Best Practice

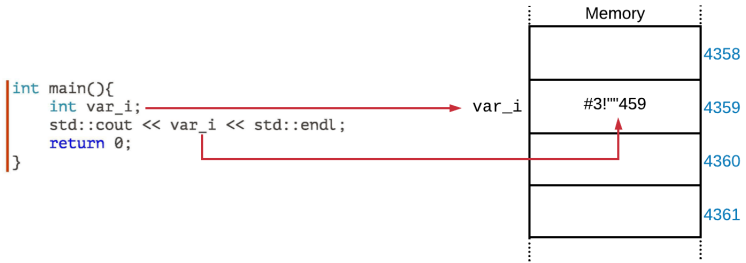
1. Be consistent with the naming conventions:
   - camelCase: `xPoint, myVariableExample`
   - snake_case: `x_point, my_variable_example`
2. Use meaningful names:
   - `double x_point;` instead of `double xp;`
   - Make it easy for yourself and for people who are collaborating with you or will take over after you.
3. Never use variables before initializing them.
4. Declare variables close to where you need them:
   - If you are using 20 variables in your program, do not declare all of them at the beginning of your code.
   - Instead, declare them right before you use them.

Variables | Initializing a Variable

- When the variables in the previous examples are **declared**, they have an indeterminate value[2] until they are assigned a value for the first time.
- Variables with indeterminate values are called uninitialized variables.
- Uninitialized variables are dangerous:
  - When you declare a variable, it is assigned a memory location by the compiler, the default value of that variable is whatever (garbage) value happens to already be in that memory location!
  - If you forget to assign a value to that variable and try to use its value, you will use garbage data in your program.

[2]Not been given a known value.

Variables | Initializing a Variable



- `var_i` is declared, the compiler assigns the variable to a free memory location (e.g., 4359).
- Other applications may have used the same memory location earlier and left garbage data behind (e.g., #3!"459).
- `std::cout` outputs the value of `var_i`, garbage data is displayed.

Variables | Initializing a Variable

- You should always give a known value to a variable at the point of definition (when it's declared), this is called the initialization of the variable.
- Once you are more comfortable with C++ , there may be certain cases where you omit the initialization for optimization purposes. But this should always be done selectively and intentionally.

Variables | Initializing a Variable

- In C++ , there are three ways to initialize variables.
- They are all equivalent and are reminiscent of the evolution of the language over the years:
  1. C-like initialization (because it is inherited from the C language):
     ```
     int var_i = 1;
     ```
  2. Constructor initialization (introduced by the C++ language):
     ```
     int var_i (1);
     ```
  3. List initialization (introduced in C++11):
     ```
     int var_i {1};
     ```

Variables | Initializing a Variable | Narrowing Conversion

- A narrowing conversion changes a value to a data type that might not be able to hold some of the possible values. For example, a fractional value is rounded when it is converted to an integral type, and a numeric type being converted to Boolean is reduced to either true or false .
- List initialization does not allow narrowing.

```
#include <iostream>

int main (){
  int var_i1 = 5.5;
  int var_i2 (3.5);
  int var_i3 {2.5};

  std::cout << var_i1 << std::endl; //-- narrowing conversion
  std::cout << var_i2 << std::endl; //-- narrowing conversion
  std::cout << var_i3 << std::endl; //-- error
}
```

Variables | Initializing a Variable

- All three ways of initializing variables are valid and equivalent in C++

```cpp
#include <iostream>

int main (){
  int var_i1 = 5;     //--ok: initial value: 5
  int var_i2 (3);     //--ok: initial value: 3
  int var_i3 {2};     //--ok: initial value: 2
  int result {};      //--ok: zero initialization

  var_i1 = var_i1 + var_i2;
  result = var_i1 - var_i3;
  std::cout << result << std::endl;

  return 0;
}
```

Variables | Initializing a Variable

- Know the difference between declaration, initialization and assignment.

```
int main(){
    int var_i1;      //--declaration
    var_i1 = 3;      //--assignment

    int var_i2 = 2; //--C-style initialization
    var_i2 = 4;      //--assignment

    int var_i3{1};  //--List initialization

    return 0;
}
```

Variables | Initializing a Variable

- Without looking at the previous slide, identify declaration, initialization and assignment.

```
1  int main(){
2      int var_i3{1};
3      var_i3 = 4;
4
5      int var_i2 = 2;
6
7      int var_i1;
8      var_i1 = 3;
9
10     int var_i3 = 2;
11 }
```

Variables | Initializing a Variable | Zero Initialization

- Slide 27 mentions zero initialization.
- This occurs when you initialize your variables with no value (e.g., `int result{}`).
- If the type of your variable is of scalar type (integral and float for variables), the variable initial value is the integral constant zero.

```
int main(){
    int var_i {};
    double var_d {};

    std::cout << var_i << sdt::endl;//--0
    std::cout << var_d << sdt::endl;//--0
    return 0;
}
```

Variables | Size of Variables

- The `sizeof` operator determines the size (in bytes) of a type or variable.
  - `sizeof(type)`: Returns size in bytes of the object representation of type.
  - `sizeof expression`: Returns size in bytes of the object representation of the type that would be returned by expression, if evaluated.

```
float f_var;

std::cout << sizeof(int) << std::endl;    //--4 bytes on my machine
std::cout << sizeof(double) << std::endl; //--8 bytes on my machine
std::cout << sizeof(f_var) << std::endl;  //--4 bytes on my machine
std::cout << sizeof f_var << std::endl;   //--4 bytes on my machine
```

Variables | Global Variables

- A global variable is a variable with global scope, meaning that it is visible (hence accessible) throughout the program, unless shadowed.
- It is usually best to put all global declarations near the beginning of the program, before the first function.
- Global variables can induce unexpected results due to mistakes made by the programmer. Since a global variable can be accessed anywhere in the code, it can be modified anywhere.

Variables | Global Variables

- No global variable, one local variable.

```
#include<iostream>

int main(){
    double x_position {16.5}; //--local variable
    std::cout << x_position << std::endl;

    return 0;
}
```

- `x_position` is local to the `main` function.

Variables | Global Variables

- One global variable, no local variable.

```
1 #include<iostream>
2 double x_position {16.5}; //--global variable
3 int main(){
4     std::cout << x_position << std::endl;//--16.5
5     return 0;
6 }
```

- `x_position` is a global variable.
- It can be accessed anywhere in the program (e.g., line 5).

Variables | Global Variables

- One global variable, one local variable.

```
1  #include<iostream>
2
3  double x_position {16.5}; //--global variable
4
5  int main(){
6      double x_position {3.5}; //--local variable
7      std::cout << x_position << std::endl;
8
9      return 0;
10 }
```

Variables | Global Variables

- In the previous example we have two variables with the same identifier: `x_position`
- The compiler will not throw an error because although having the same identifier, these variables are not declared in the same scope:
  - `x_position` (line 3) is a global variable and has global scope.
  - `x_position` (line 6) has local scope to the `main` function.
  - `x_position` (line 7) is the local variable because it shadows any global variable with the same identifier (see §1 in slide 32).

Variables | Global Variables | Best Practice

- If you plan to use global variables, try to differentiate them with local variables (e.g., prefix global variables with `g_`).

```
#include<iostream>
double g_x_position {16.5}; //--global variable

int main(){
    double x_position {20}; //--local variable
    std::cout << g_x_position << std::endl; //--16.5
    std::cout << x_position << std::endl; //--20
    return 0;
}
```

- Avoid using global variables as it can induce errors in your program: Since they are accessible anywhere in the program, their value can be changed anywhere in the program.

Constants

- A constant is an expression with a fixed value.
- Attempting to change the value of a constant will result in a compiler error.
- Constants must be initialized when they are created.

Constants

- C++ has two types of constants:
  - Literal constant: A value typed directly into your program wherever it is needed.
    - The following statement assigns the integer variable `width` the value `10`. The `10` in the statement is a literal constant. You cannot assign a value to `10`, and its value cannot be changed.

      ```
      double width = 10;
      ```
  - Symbolic constant: A constant represented by a name, just like a variable. For instance:

    ```
    const double kPi = 3.1415926;
    const char kTab = '\t';
    ```

    - Make sure you follow the Google Style Guide to properly name constants.

Constants | Types

- Constants can be of any of the basic data types and can be divided into:
  - Integer literals.
  - Floating-Point literals.
  - Characters.
  - Strings.
  - Boolean values.

Constants | Types | Best Practice

- Trying to build (compile time) the code below will generate an error since a constant cannot be modified.

```
#include <iostream>

int main(){
    const int kSide {10}; //--ok
    kSide = 20; //--error, read only
    return 0;
}
```

Constants | Variable Vs. Constant

- Be efficient in your programs.
- If you know your data will have its value modified in the program, then use a variable. Otherwise, use a constant.
- This may seem obvious but programmers use variables most of time without thinking if constants may be more appropriate in certain cases.

Macros

- A macro is a fragment of code which has been given a name.
- Macro is defined with the `#define` directive.
- When the preprocessor expands a macro name, the macros expansion replaces the macro invocation.
- There are two kinds of macros. They differ mostly in what they look like when they are used.
  - Object-like macros resemble data objects when used.
  - Function-like macros resemble function calls.
- More about macros can be found here.

Macros | Object-like Macros

- An object-like macro is a simple identifier which will be replaced by a code fragment.
- They are most commonly used to give symbolic names to numeric constants.
- Syntax : `#define <identifier> <value>`

```
#include<iostream>

#define LENGTH 10
#define WIDTH  5
#define OPEN_CURLY {

int main() OPEN_CURLY
   int area;

   area = LENGTH * WIDTH;
   std::cout << area;
   return 0;
}
```

Macros | Function-like Macros

- You can also define macros whose use looks like a function call.
- To define a function-like macro, you use the same `#define` directive but you put a pair of parentheses immediately after the macro name.

```
#include<iostream>

#define LENGTH 10
#define WIDTH  5
#define NEWLINE '\n'
#define LOG(x)  std::cout << x;

int main() {
    int area;

    area = LENGTH * WIDTH;
    LOG(area)
    std::cout << NEWLINE;
    return 0;
}
```

Macros | Undefining and Redefining Macros

- If a macro ceases to be useful, it may be undefined with the `#undef` directive.
- `#undef` takes a single argument, the name of the macro to undefine.
- You use the bare macro name, even if the macro is function-like.
- `#undef` has no effect if the name is not a macro.

```
#include<iostream>

#define LENGTH 10
#define LOG(x)  std::cout << x;

int main() {
    LOG(LENGTH)
    #undef LENGTH
    #undef LOG
}
```

Macros | Undefining and Redefining Macros

- Once a macro has been undefined, that identifier may be redefined as a macro by a subsequent `#define` directive.

```
1  #include <iostream>
2
3  #define LENGTH 10
4
5  int main(){
6      std::cout << LENGTH << std::endl; //-- 10
7      {//--creating a new scope
8          #undef LENGTH
9          #define LENGTH 200
10         std::cout << LENGTH << std::endl; //-- 200
11     }
12     std::cout << LENGTH << std::endl; //-- 200
13     return 0;
14 }
```

Macros | Undefining and Redefining Macros

- If you want to redefine a macro inside your code you can use `#undef` to cancel its definition first, then redefine it with `#define`.
- Note that you cannot have local and global macros the same way you can have local and global variables.
- We created a new scope inside the `main` function (lines 7–11) in which we redefined the macro `LENGTH`.
- When we display the value of `LENGTH` outside the scope (line 12), the redefined macro is displayed, meaning that the same macro `LENGTH`, originally defined at line 3 is used in the whole program. We just changed its value.
- Note: Using and redefining macros should be avoided at any cost. However you will see this practice in C++ legacy code and you should be aware of it.

Macros | Macros Vs. Constants

- Usually, it is recommended to use compile time constant variables over macros.
  - **Scope-based mechanism**: The scope of macros is limited to the file in which they are defined. Macros which are created in one source file are NOT available in a different source file. Constants can be scoped and obey all scoping rules.
  - **Ease of debugging**: If you receive an error during compilation, it will be confusing because the error message will not refer the macro name but the value and it will appear a sudden value, and one would waste lot of time tracking it down in code.
- Question: Do macros take additional memory?

Input/Output (I/O)

- Input/Output elements allow the program to interact with external entities.
  - Print something out to the terminal screen.
  - Capture what the user inputs on the keyboard.
- C++ uses streams to perform input and output operations with a screen, a keyboard, or a file.
  - e.g., interacting with the keyboard and the screen:

```
int number;
std::cout << "Enter an integer: ";
std::cin >> number;
```

Input/Output (I/O)

- The header `<iostream>` defines the standard input/output stream objects.
- Including `<iostream>` automatically includes also `<ios>`, `<streambuf>`, `<istream>`, `<ostream>`, and `<iosfwd>`.
- `cout` and `cin` are objects representing streams.

Input/Output (I/O) | cout

- `cout` and <<
  - The `cout` object in C++ is an object of class `ostream`.
  - `cout` is said to be "connected to" the standard output device, which usually is the display screen.
  - `cout` and << insert the data into the output stream.
  - << is called the insertion operator.

Input/Output (I/O) | cout

- cout and << insert the value of data into the output stream:

```
int data;
data = 1;
std::cout << data;
```

- Can be chained:

```
int data;
data = 1;
std::cout << "data is " << data;
```

- Does not automatically add line breaks:

```
int data;
data=1;
std::cout << "data is " << data << std::endl;
std::cout << "data is " << data << "\n";
```

Input/Output (I/O) | cin

- `cin` and `>>`
  - The `cin` object in C++ is an object of class `istream`.
  - `cin` is said to be attached to the standard input device, which usually is the keyboard.
  - `cin` and `>>` extract data from the input stream based on data's type.
  - `>>` is called the extraction operator.

Input/Output (I/O) | cin

- `cin` and >> extract data from the input stream based on data type:

```
int data2;
std::cin >> data1;
```

- Can be chained:

```
int data1;
int data2;
std::cin >> data1 >> data2;
```

- Can fail if the entered value(s) cannot be interpreted.

Input/Output (I/O) | cin | Example

```
std::cout << "Please enter two numbers separated by a space: ";

int num1;
int num2;
std::cin >> num1 >> num2;
std::cout << "You entered: " << num1 << ", " << num2 << std::endl;
```

Input/Output (I/O) | Quiz

- What is the name of the following operators?
  - ::
  - <<
  - >>
- Spot the error(s) in the following code:

```
#include<iostream>

void main(){
    int x{3.15};
    int y = 4;
    std::cout >> x >> ", " >> y >> endl
}
```

Namespaces

- Complex C++ code requires:
  1. Developers' code.
  2. C++ Standard Library (`std`) + code.
  3. Third party libraries + code.
- With libraries and codes coming from different sources, we need a way to make sure we are using the correct functions, variables, etc, from the correct library.

Namespaces

- Scenario:
  - My project needs to include two third party libraries, from companies X and Y.
  - Library from Company X contains a function named `print()`.
  - Library from Company Y contains a function named `print()`.
  - If I import libraries from companies X and Y and I call `print()` in my code, the C++ compiler will not know which one to use: naming conflict.
  - Note: This applies with other C++ components (variables, classes, struct, etc).

Namespaces

- Namespaces are used to organize code into logical groups and to prevent naming conflicts.
- `std` is the namespace for **C++ Standard Library**: *A collection of classes and functions, which are written in the core language and part of the C++ ISO Standard itself.*
  - `cin`, `cout`, and other objects[3] are defined in it.
- `std::cout` means that we are using the `cout` object from the C++ Standard Library (`::` is called the scope resolution).

[3] Will be discussed in a future lecture.

Namespaces

- Writing C++ code without using the C++ Standard Library is almost impossible.
- Many programmers find it tedious to type `std::` all the time.
- C++ provides different mechanisms for handling objects of a namespace.
  1. Explicit use of namespaces.
  2. Through the `using namespace` directive.
  3. Through qualified `using namespace` variant.

Namespaces | Explicit Use

```
#include<iostream>

int main(){
    int number;
    std::cout << "Enter an integer: ";
    std::cin >> number;
    std::cout << "You entered: " << number << std::endl;
    return 0;
}
```

Namespaces | Directive

```cpp
#include<iostream>
using namespace std; //--access everything in the std namespace

int main(){
    int number;
    cout << "Enter an integer: ";
    cin >> number;
    cout << "You entered: " << number << endl;
    return 0;
}
```

Namespaces | Qualified Variant

```cpp
#include<iostream>

//--use only what you need from the Standard Library
using std::cout;
using std::cin;
using std::endl;

int main(){
    int number;
    cout << "Enter an integer: ";
    cin >> number;
    cout << "You entered: " << number << endl;
    return 0;
}
```

Namespaces | Best Practice

- Avoid polluting your C++ code with the `using namespace std;` directive.
  - Particularly if you are using this directive in .h files.
  - It is OK to use this directive in .cpp files.
- Best practice is to explicitly use the namespace (e.g., `std::cout`) or use the qualified variants option (e.g., `using std::cout`).
- In future classes, we will:
  - Use `using namespace std;` only in main.cpp.
  - Use `using std::cout, using std::cin, etc` when working in other .cpp and .h files besides main.cpp.

Exercices | Exercise 1

- Write a single output statement using `cout` that displays "Hello ENPM809Y".

Exercices | Exercise 2

- Write the code that will output the number of students in a class.
- Use `cout` and the insertion operator <<
- `number_of_students` has already been declared and initialized for you.
- The output should read This class has 15 students.

```
int main(){
    int number_of_students{15};
    /*
    Write your code here
    */
    return 0;
}
```

Exercices | Exercise 3

- Using `cout` and the insertion operator <<, ask the user to enter their date of birth.
  - The terminal output should display Please enter your date of birth in the format: d m y:
- Using `cin` and the extraction operator >>, store the user's inputs in the following variables:
  - The variable `d` (integer) represents the day.
  - The variable `m` (integer) represents the month.
  - The variable `y` (integer) represents the year.
- Using `cout` and the insertion operator <<, display Your date of birth is 12/23/1950, assuming the user's inputs were 23 12 1950.

Next Class | 09/10

- Program Flow Control.
- We will also use Doxygen in next lecture.
- Zoom lecture.
- No quiz and no assignment.
- Stay safe!

References