1. Explanation of the Topic

Introduction to C++ Basic Syntax

- **Preprocessor Directives:**
    - Lines starting with #, such as #include <iostream>, tell the compiler to include additional libraries (in this case, the iostream library for input/output operations).
- **The main() Function:**
    - Every C++ program starts execution in the main() function.
    - Its signature is usually defined as int main(), which means it returns an integer value (usually 0) to the operating system upon completion.
- **Statements and Semicolons:**
    - Each instruction (or statement) ends with a semicolon (;).
- **Blocks and Braces:**
    - Code that belongs together is grouped inside curly braces { }.
- **Variables and Data Types:**
    - Variables (like int a;) store data and must be declared with a data type.
    - In our example, int represents an integer type.

Commenting in C++

- **Single-Line Comments:**
    - Begin with // and continue until the end of the line.
    - They are used to explain code or to temporarily disable code.
- **Multi-Line Comments:**
    - Begin with /* and end with */.
    - Useful for longer explanations or commenting out blocks of code.

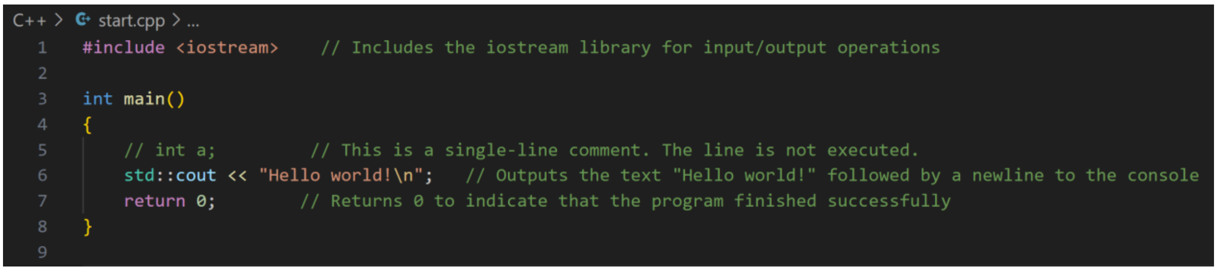
Using the g++ Compiler and Compilation Flags

- **g++ Compiler:**
    - g++ is a widely used compiler for C++ programs provided by the GNU Compiler Collection.
- **Common Compilation Flags:**
    - **-o**
        - **Usage:** g++ source.cpp -o output_executable
        - **Purpose:** Specifies the name of the output file (executable).
    - **-W**
        - **Usage:** Often used to enable additional warnings.

    - **Purpose:** Alerts you about potential issues in your code.
    - **Note:** -Wall is a more comprehensive flag (see below).
  - **-pedantic**
    - **Usage:** g++ source.cpp -pedantic
    - **Purpose:** Enforces strict adherence to the C++ standard.
    - **Tip:** Helps ensure portability and standard-compliant code.
  - **-Wall**
    - **Usage:** g++ source.cpp -Wall
    - **Purpose:** Activates most warning messages to help catch common mistakes.
  - **-E**
    - **Usage:** g++ source.cpp -E
    - **Purpose:** Runs only the preprocessor, outputting the expanded source code.
  - **-S**
    - **Usage:** g++ source.cpp -S
    - **Purpose:** Compiles the code to assembly language instead of an executable.
  - **-O Optimization Flags:**
    - **Usage:** -O0, -O1, -O2, -O3
    - **Purpose:** Controls the level of code optimization:
      - O0: No optimization (default, easier debugging).
      - O1, O2, O3: Increasing levels of optimization for better performance.
  - **-static**
    - **Usage:** g++ source.cpp -static
    - **Purpose:** Instructs the compiler to link libraries statically, meaning all libraries are included in the executable.

2. Code Analysis

Below is the provided C++ code snippet with explanations:



```
C++ > start.cpp > ...
#include <iostream>    // Includes the iostream library for input/output operations

int main()
{
    // int a;         // This is a single-line comment. The line is not executed.
    std::cout << "Hello world!\n";   // Outputs the text "Hello world!" followed by a newline to the console
    return 0;        // Returns 0 to indicate that the program finished successfully
}

```

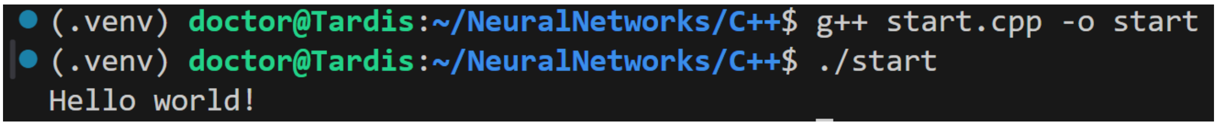
Key Points:

- **Header Inclusion (#include <iostream>):**
  - Allows the program to use std::cout for printing text.
- **Main Function (int main()):**
  - Marks the starting point of program execution.
- **Commented Code (// int a;):**
  - Demonstrates how to comment out code that you do not want to execute.
- **Output Statement (std::cout << "Hello world!\n";):**
  - Uses the << operator to send the string to the standard output (console).
- **Return Statement (return 0;):**
  - Signals successful program termination.

Compilation:

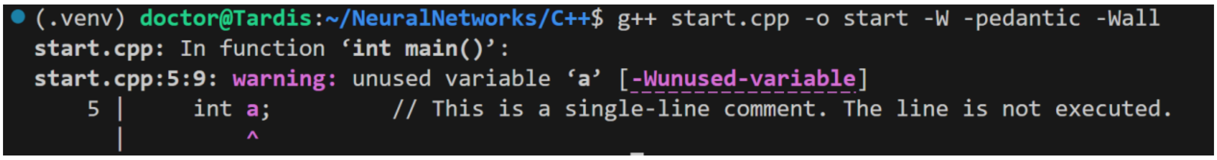
Remember to access the folder, in which your .cpp file is stored through the console (using *cd <foldername>* command in linux). All commands in this section are executed in that folder.

- **Using command** `g++ start.cpp -o start`
  - Creates compiled file *start*, which you can execute using *./start* command



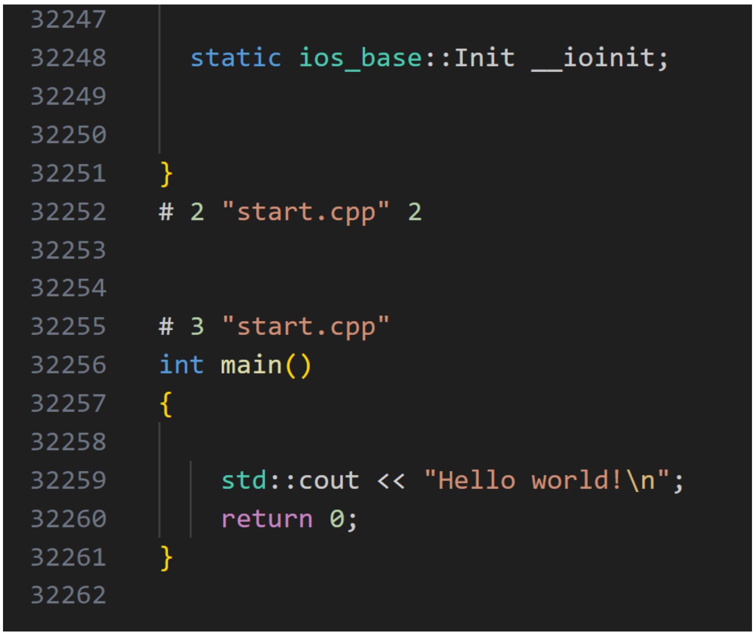
```
(.venv) doctor@Tardis:~/NeuralNetworks/C++$ g++ start.cpp -o start
(.venv) doctor@Tardis:~/NeuralNetworks/C++$ ./start
Hello world!
```

- **Using command** `g++ start.cpp -o start -W -pedantic -Wall` **with uncommented line** `5        int a;`
  - Creates compiled file *start* and shows any warnings in the console



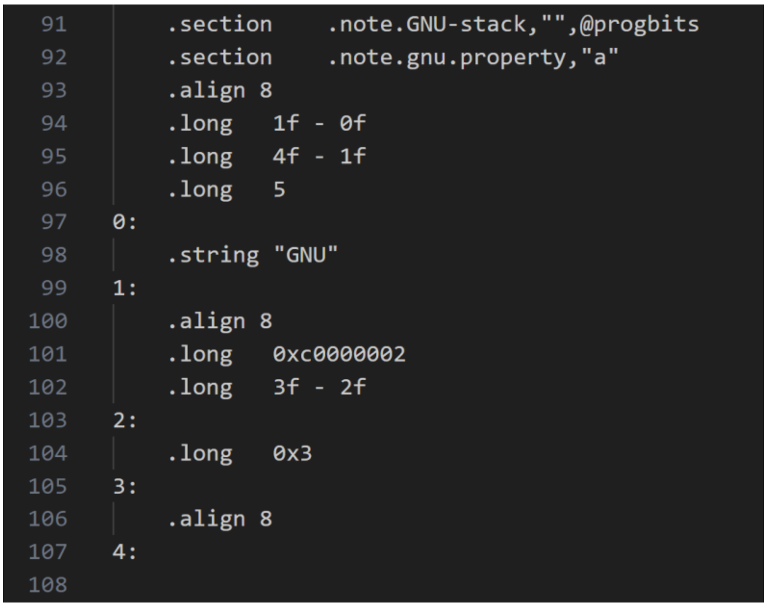
```
(.venv) doctor@Tardis:~/NeuralNetworks/C++$ g++ start.cpp -o start -W -pedantic -Wall
start.cpp: In function ‘int main()’:
start.cpp:5:9: warning: unused variable ‘a’ [-Wunused-variable]
    5 |     int a;          // This is a single-line comment. The line is not executed.
      |         ^
```

- **Using command** `g++ start.cpp -o start1 -E`
  - Creates only preprocessed file *start1*. Notice how many additional lines are created during the preprocessing for such a simple program. That's why writing efficient code in C++ is so important. Also comments from original code are not included in that file.



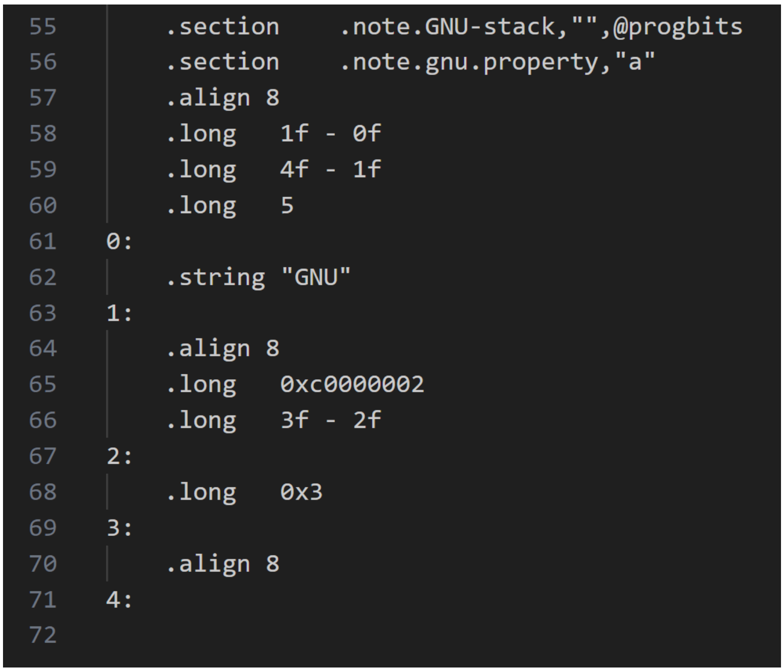
```

    static ios_base::Init __ioinit;


}
# 2 "start.cpp" 2


# 3 "start.cpp"
int main()
{

    std::cout << "Hello world!\n";
    return 0;
}

```

- **Using command** `g++ start.cpp -o start1.asm -S`
  - Creates an assembly file *start1.asm*. The C++ source code is translated into assembly instructions specific to the target architecture. This step allows inspecting how the high-level code is transformed before machine code generation. By default, the optimization level is set to *-O0*, meaning no optimizations are applied, resulting in straightforward and less efficient assembly output.



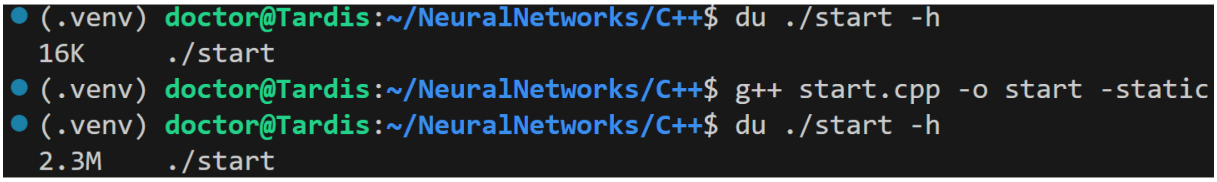
```
	.section	.note.GNU-stack,"",@progbits
	.section	.note.gnu.property,"a"
	.align 8
	.long	1f - 0f
	.long	4f - 1f
	.long	5
0:
	.string "GNU"
1:
	.align 8
	.long	0xc0000002
	.long	3f - 2f
2:
	.long	0x3
3:
	.align 8
4:

```

- **Using command** `g++ start.cpp -o start2.asm -S -O2`
  - Generates an optimized assembly file *start2.asm*. The C++ source code is translated into assembly instructions with optimization level *-O2*, which enables various performance improvements such as eliminating redundant computations, inlining functions, and optimizing loops. As a result, the generated assembly code is more efficient and often shorter than with *-O0*.



```
	.section	.note.GNU-stack,"",@progbits
	.section	.note.gnu.property,"a"
	.align 8
	.long	1f - 0f
	.long	4f - 1f
	.long	5
0:
	.string "GNU"
1:
	.align 8
	.long	0xc0000002
	.long	3f - 2f
2:
	.long	0x3
3:
	.align 8
4:

```

- **Using command** `g++ start.cpp -o start -static`
  - Generates a statically linked executable *start*. Unlike dynamically linked executables, which rely on shared libraries at runtime, a statically linked binary includes all necessary library code within itself. This results in a significantly larger file size but makes the program independent of external shared libraries. To see file size use command du ./start -h



```
● (.venv) doctor@Tardis:~/NeuralNetworks/C++$ du ./start -h
  16K     ./start
● (.venv) doctor@Tardis:~/NeuralNetworks/C++$ g++ start.cpp -o start -static
● (.venv) doctor@Tardis:~/NeuralNetworks/C++$ du ./start -h
  2.3M    ./start
```

3. Practical Exercises

Exercise 1: Modify the "Hello world!" Program

Task:
Modify the provided code so that it prints two lines:

- First line: "Hello world!"
- Second line: "Welcome to C++ programming."

Hint:
Use two std::cout statements or include the newline character \n in one statement.

Exercise 2: Use Multi-Line Comments

Task:
Create a program that prints your name and age in two lines. Use a multi-line comment at the beginning of the code to describe the program.

Hint:
Place the multi-line comment between /* and */.

Exercise 3: Experiment with Compilation Flags

Task:
Add an integer variable with value of your choice to code from Exercise 2. Compile the program with different flags. Try using -*Wall* and -*pedantic* to see the warnings and ensure standard compliance.

Hint:
Name files with your name – this will come in handy while uploading to moodle.

Expected Outcome:
You should see a warning about the unused variable when you compile with the -Wall flag.