Lecture 3: Review of C

Professor:
Dr. Yanmin Gong

Graduate Teaching Assistants:
Francisco E. Fernandes Jr.
feferna@okstate.edu

Khuong Vinh Nguyen
Khuong.V.Nguyen@okstate.edu

School of Electrical and Computer Engineering
Oklahoma State University
Spring 2019

The General Form of a Simple Program

- Simple **C** programs have the form:

```
directives

int main(void)
{
      statements
}
```

- **C** uses **{** and **}** in much the same way that some other languages use words like **begin** and **end**.
- Even the simplest **C** programs rely on three key language features:
  - **Directives**
  - **Functions**
  - **Statements**

Directives

- Before a **C** program is compiled, it is first edited by a **preprocessor**.
- Commands intended for the **preprocessor** are called **directives**.
- Example:

```
#include <stdio.h>
```

- `<stdio.h>` is a ***header*** containing information about **C**'s standard **I/O** library.
- Directives always begin with a # character.
- By default, directives are one line long; there's no semicolon or other special marker at the end.

Directives

- Our standard directive will be the following:

  `#include “stm32l476xx.h”`

- The **`stm32l476xx.h`** file contains:
  - Data structures and the address mapping for all peripherals.
  - Peripheral's registers declarations and bits definition.
  - Macros to access peripheral’s registers hardware.
- Example:

```
/********************  Bit definition for RCC_AHB2ENR register  ***************/
    #define  RCC_AHB2ENR_GPIOAEN                  ((uint32_t)0x00000001U)
    #define  RCC_AHB2ENR_GPIOBEN                  ((uint32_t)0x00000002U)
    #define  RCC_AHB2ENR_GPIOCEN                  ((uint32_t)0x00000004U)
    #define  RCC_AHB2ENR_GPIODEN                  ((uint32_t)0x00000008U)
    #define  RCC_AHB2ENR_GPIOEEN                  ((uint32_t)0x00000010U)
    #define  RCC_AHB2ENR_GPIOFEN                  ((uint32_t)0x00000020U)
    #define  RCC_AHB2ENR_GPIOGEN                  ((uint32_t)0x00000040U)
    #define  RCC_AHB2ENR_GPIOHEN                  ((uint32_t)0x00000080U)
    #define  RCC_AHB2ENR_OTGFSEN                  ((uint32_t)0x00001000U)
    #define  RCC_AHB2ENR_ADCEN                    ((uint32_t)0x00002000U)
    #define  RCC_AHB2ENR_RNGEN                    ((uint32_t)0x00040000U)
```

Functions

- A **function** is a series of statements that have been grouped together and given a name.
- **Library functions** are provided as part of the **C** implementation.
- A function that computes a value uses a `return` statement to specify what value it "returns":

```
return x + 1;
```

The `main` Function

- The **`main`** function is mandatory.
- **`main`** is special: it gets called automatically when the program is executed.
- **`main`** returns a status code; the value 0 indicates normal program termination.
- If there's no **`return`** statement at the end of the **`main`** function, many compilers will produce a warning message.
  - The compiler used in our labs does not requires a **`return`** statement at the end of the **`main`** function.

Statements

- A **statement** is a command to be executed when the program runs.
- Asking a function to perform its assigned task is known as **calling** the function.
- For example, to display a string we call the **`printf`** function:

```
printf("To C, or not to C: that is the question.\n");
```

Printing Strings

- When the **`printf`** function displays a **string literal -** characters enclosed in double quotation marks - it doesn't show the quotation marks.
- **`printf`** doesn't automatically advance to the next output line when it finishes printing.
- To make **`printf`** advance one line, include `\n` (the ***new-line character***) in the string to be printed.
- However, when programming at the Bare Metal Layer, we do not have access to the **`printf`** function.
- When developing for embedded systems, debugging is done by manually verifying the values of the processor registers.

Comments

- A **comment** begins with **`/*`** and end with **`*/`**.

```
/* This is a comment */
```

- A single line comment can be written using **`//`**. For example:

```
// This is a single line comment
```

- Comments may appear almost anywhere in a program, either on separate lines or on the same lines as other program text.
- Comments may extend over more than one line.

```
/* Name: pun.c
   Purpose: Prints a bad pun.
   Author: K. N. King */
```

Overall Program Structure

```
#include "stm32l476xx.h"

int main(void){
    RCC->AHB2ENR |= 0x02;  // Enable clock of Port B

    GPIOB->MODER &= ~(3<<4); // Clear mode bits
    GPIOB->MODER |=   1<<4;  // Set mode to output

    GPIOB->OTYPER &= ~(1<<2); // Select push-pull output

    GPIOB->ODR   |= 1 << 2;  // Output 1 to turn on red LED

    while(1) {
    }
}
```

This **directive** will always be used in all our programs!

Always use **`int main(void)`**!

Comments

The red lines are the **statements** of this program.

Remember: when modifying registers, always use **bitwise operations**!

All of your programming logic goes inside this dead loop.

Indentation is important to keep your code organized. We normally use a single Tab or four Spaces.

Variables and Assignment

- Most programs need a way to store data temporarily during program execution.
  - Our brain does this as well to remember key events in our life!
  - Or, perhaps, our phone number or home address.
- These storage locations are called **variables**.
- **Variables** must be **declared** before they are used.

Variable Types

- Every variable must have a **`type`**.
- **`C`** has a wide variety of types, including **`int`** and **`float`**.
- A variable of type **`int`** (short for *integer*) can store a whole number such as **`0`**, **`1`**, **`392`**, or **`-2553`**.
  - The largest **`int`** value is typically **`2,147,483,647`** but can be as small as **`32,767`**.
- A variable of type **`float`** (short for *floating-point*) can store much larger numbers than an **`int`** variable.
- Also, a **`float`** variable can store numbers with digits after the decimal point, like **`379.125`**.
- Drawbacks of **`float`** variables:
  - Slower arithmetic
  - Approximate nature of **`float`** values

Variable Types

- Variables can be declared one at a time:

```
int height;
float profit;
```

- Alternatively, several can be declared at the same time:

```
int height, length, width, volume;
float profit, loss;
```

Variable Types

- When dealing with the **`32-bit registers`** in our **`ARM Cortex-M4`** processor, we are going to use a special variable type called **`fixed width integer types`**.
- These are the most common ones we are going to be using:
  - **`uint8_t`**
  - **`uint16_t`**
  - **`uint32_t`**
  - **`uint64_t`**
- These represent **`unsigned integer type`** with *`width of exactly`* **8**, **16**, **32** and **64 bits**, respectively.

Variable Types

- When **`main`** contains **declarations**, these must precede **statements**:

```
int main(void)
{
  declarations
  statements
}
```

Defining Names for Constants

- If you are writing a program that use the same constant number throughout your code, you can use a feature called **`macro definition`**.
- For example:

```
#define RED_LED_PIN 2
```

- The above line will make the name **`RED_LED_PIN`** equal to the numeral **`2`**.
- When a program is compiled, the **`preprocessor`** replaces each **`macro`** by the value that it represents.
- The value of a macro can be an expression:

```
#define RECIPROCAL_OF_PI (1.0f / 3.14159f)
```

- If it contains operators, the expression should be enclosed in parentheses.
- Using only upper-case letters in macro names is a common convention.

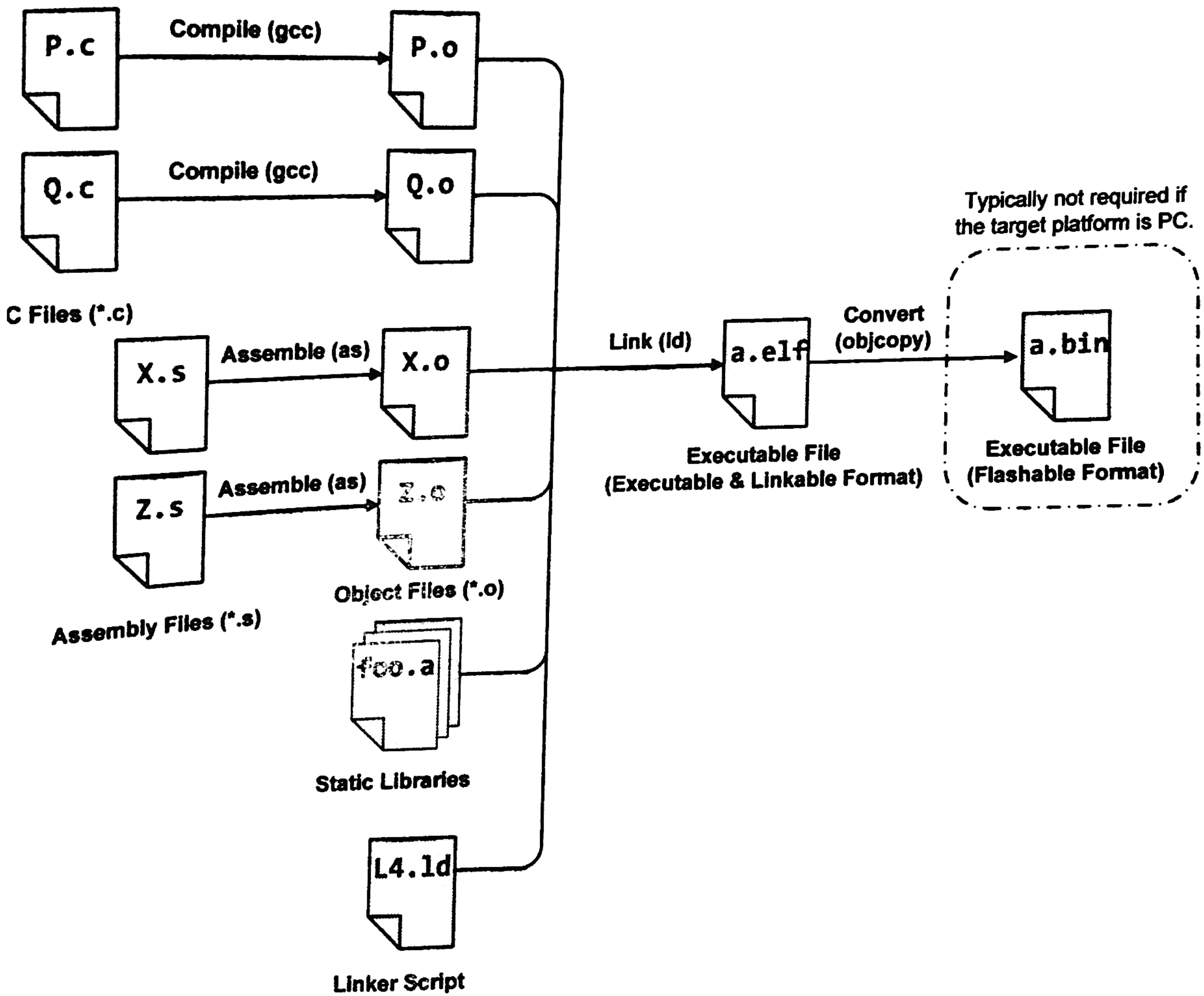
GNU Compiler

- The **GNU Compiler Collection** (**GCC**) consists of a suite of free, open-source, and widely used programming and debugging tools for many types of processors, such as x86/x64, ARM, MIPS, and AVR. The following lists a few important tools.
  - The **GNU C compiler** (**gcc**) translates a **C** source file to an assembly file or to an object file (machine code).
  - The **assembler** (**as**) converts an assembly program to an object file.
  - The **linker** (**ld**) links object files and pre-compiled libraries into an executable file in a format such as **ELF** (**Executable and Linkable Format**).

GNU Compiler

- To program microprocessors, flash programmers often require us to convert the **`ELF`** format to a specific binary format that can be directly written to **`flash`** or **`ROM`**. We can use **`objcopy`** to achieve the conversion.
- The **`debugger`** (**`gdb`**) allow us to debug a program step by step.

GNU Compiler



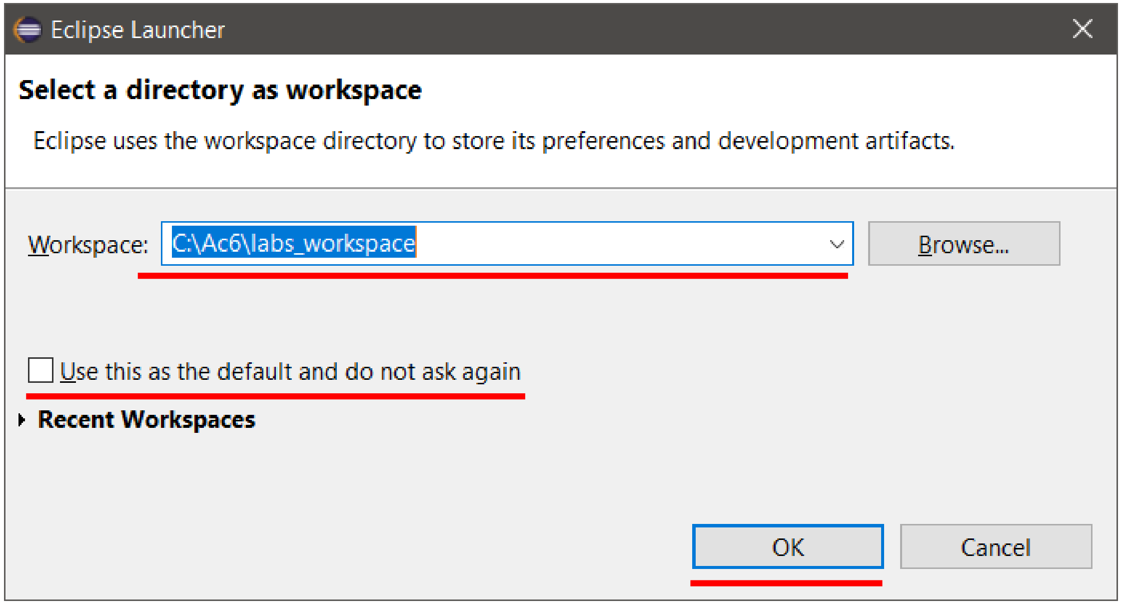
GNU Compiler

- Commands to build the project given in the previous slide:

```
gcc -c -g -o P.o P.c
gcc -c -g -o Q.o Q.c
as  -g -o X.o X.s
as  -g -o X.o X.s
as  -g -o Z.o Z.s
ld TL4.ld -lfoo -o a.elf P.o Q.o X.o Q.o
objcopy -O binary a.elf a.bin
```

Creating a New Project on System Workbench

- The first time you open the System Workbench IDE, you will have to select a folder where all your projects will be located.



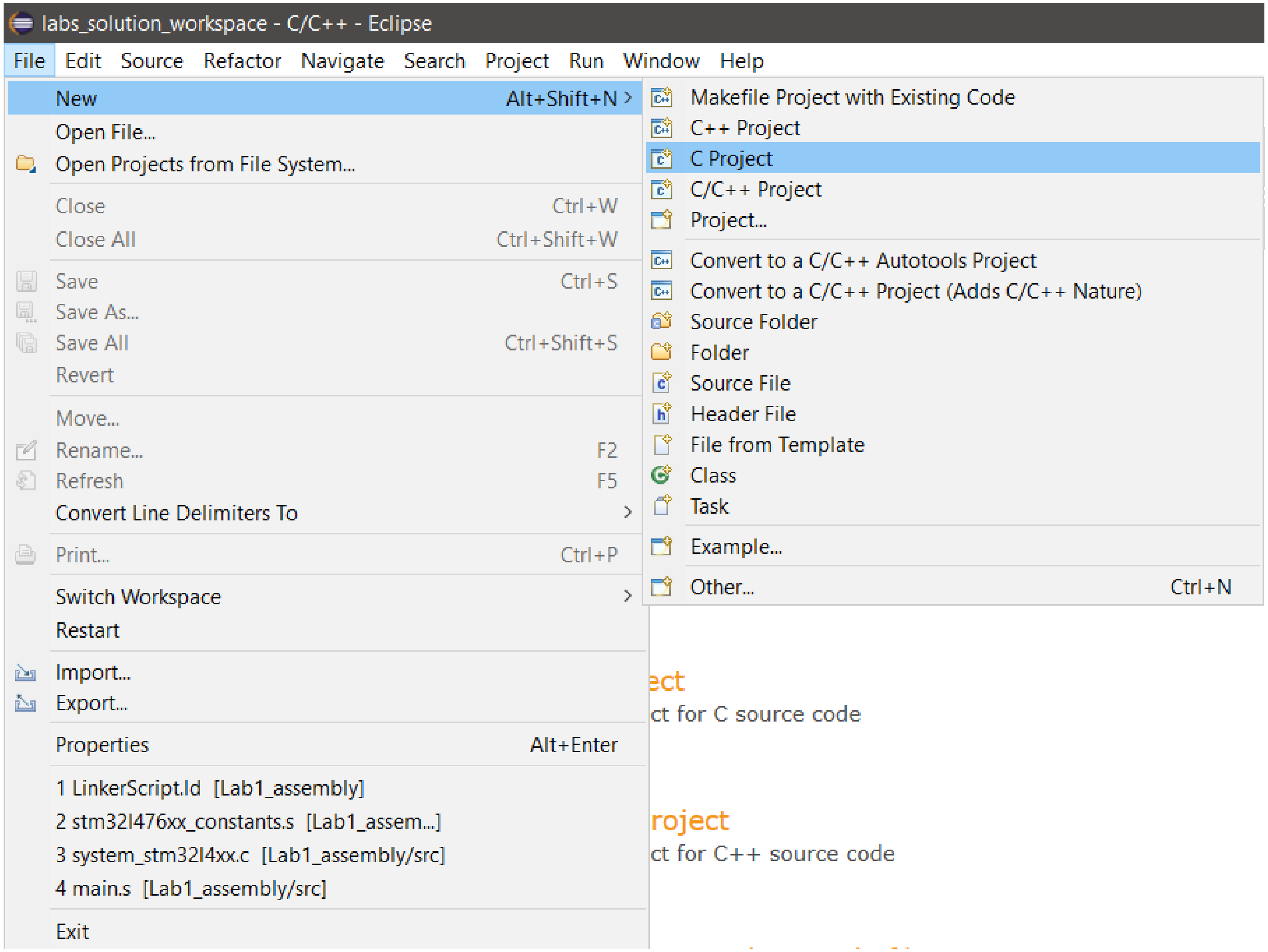
Important: Your workspace folder CANNOT contain any spaces in its name! Otherwise, you will face compilation errors.

It is recommended to create a folder in your C:\ unit.

- If you don't want to always the folder every time you open the IDE, you can check the box **`Use this as the default and do not ask again`**.
- Click on the OK button to open the IDE.

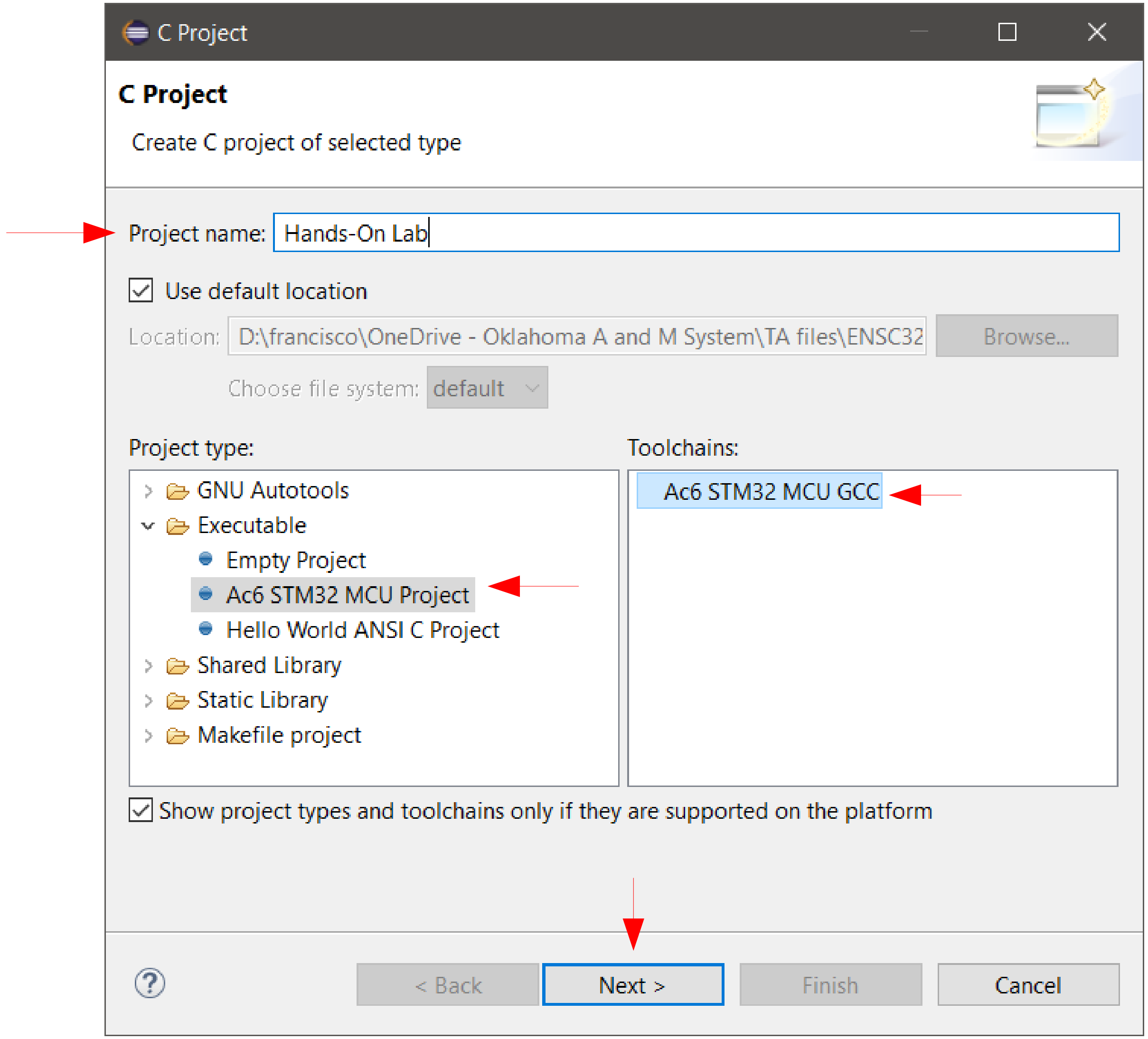
Creating a New Project on System Workbench

- Once the IDE has opened, you need to select **`File` → `New` → `C Project`**.



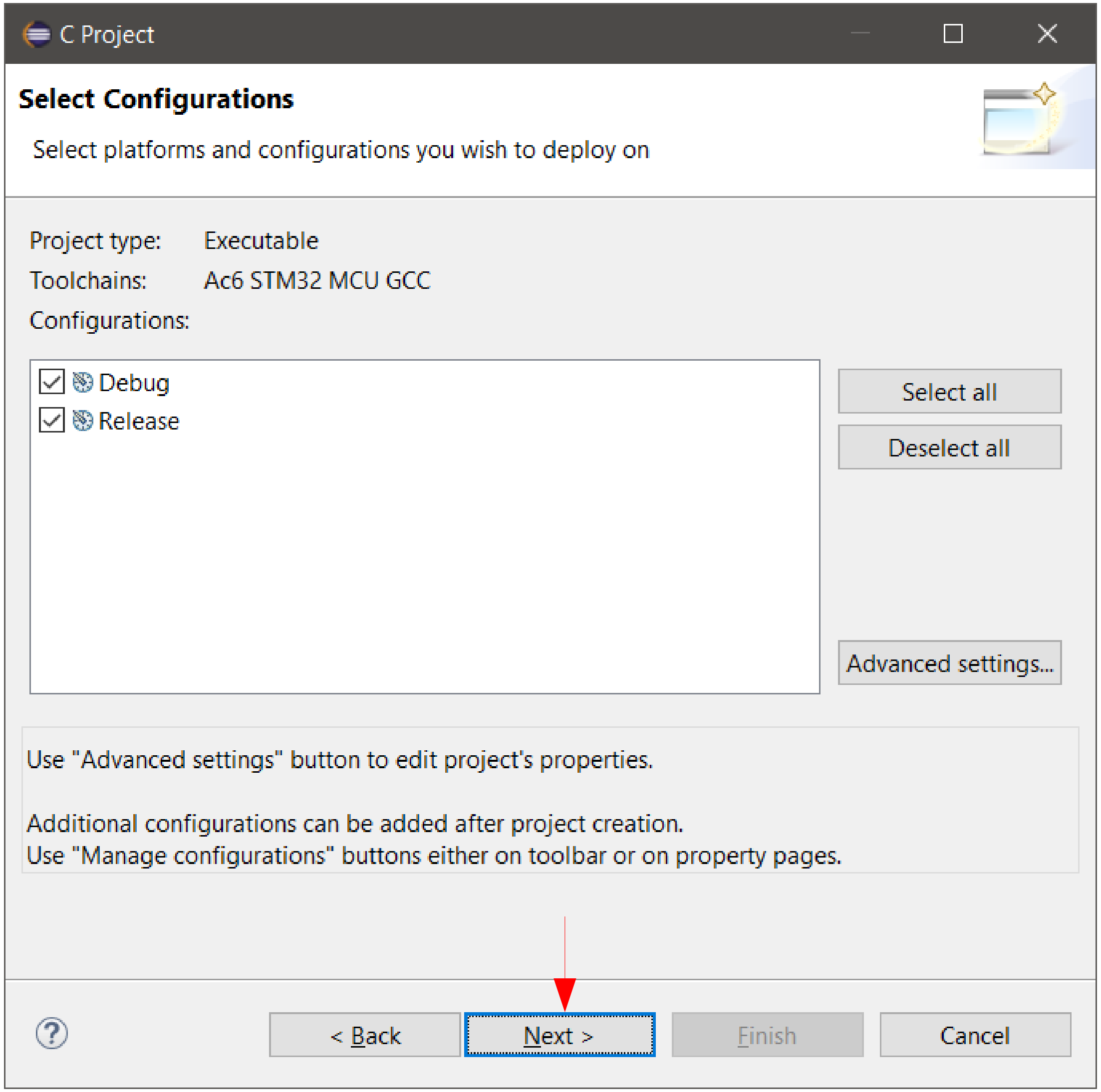
Creating a New Project on System Workbench

- On the new window, give a name for your project, select **Ac6 STM32 MCU Project → Ac6 STM32 MCU GCC**, and click on **Next**.



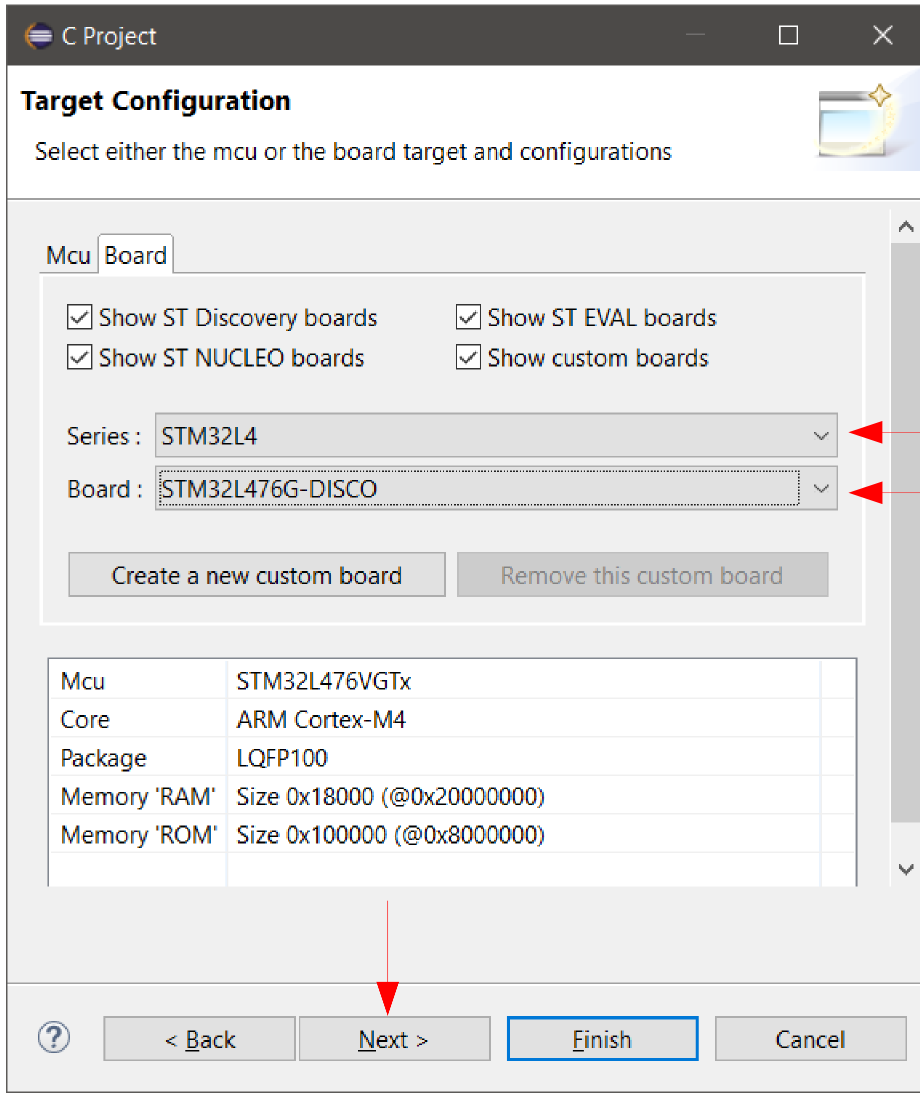
Creating a New Project on System Workbench

- On the window called **`Select Configurations`**, do not change anything, and just click on the **`Next`** button.



Creating a New Project on System Workbench

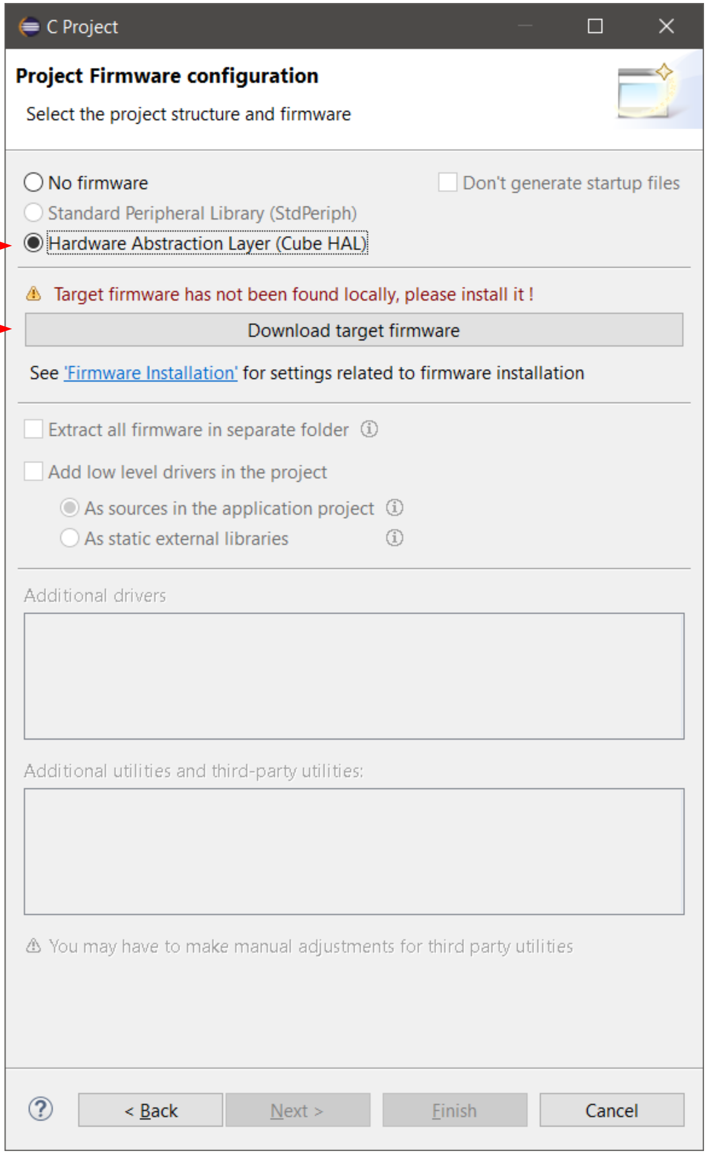
- On the window called **`Target Configuration`**, make sure everything is identical to the picture below, and click on **`Next`**:



DON'T click on Finish at this point!

Creating a New Project on System Workbench

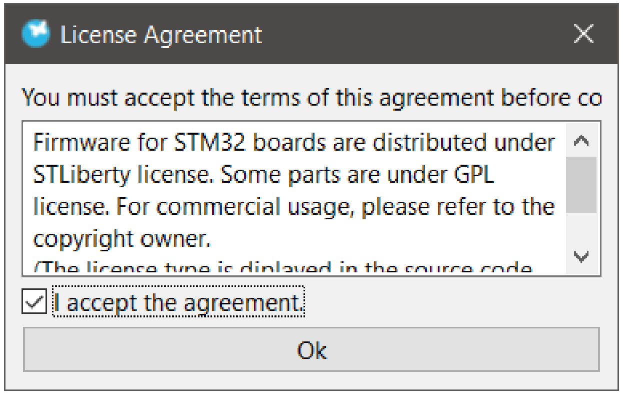
- On **Project Firmware Configuration**, select **Hardware Abstraction Layer (Cube HAL)**, and click on **Download target firmware**.



Note: you only need to download the target firmware once. After this first download, there will be no need to download again.

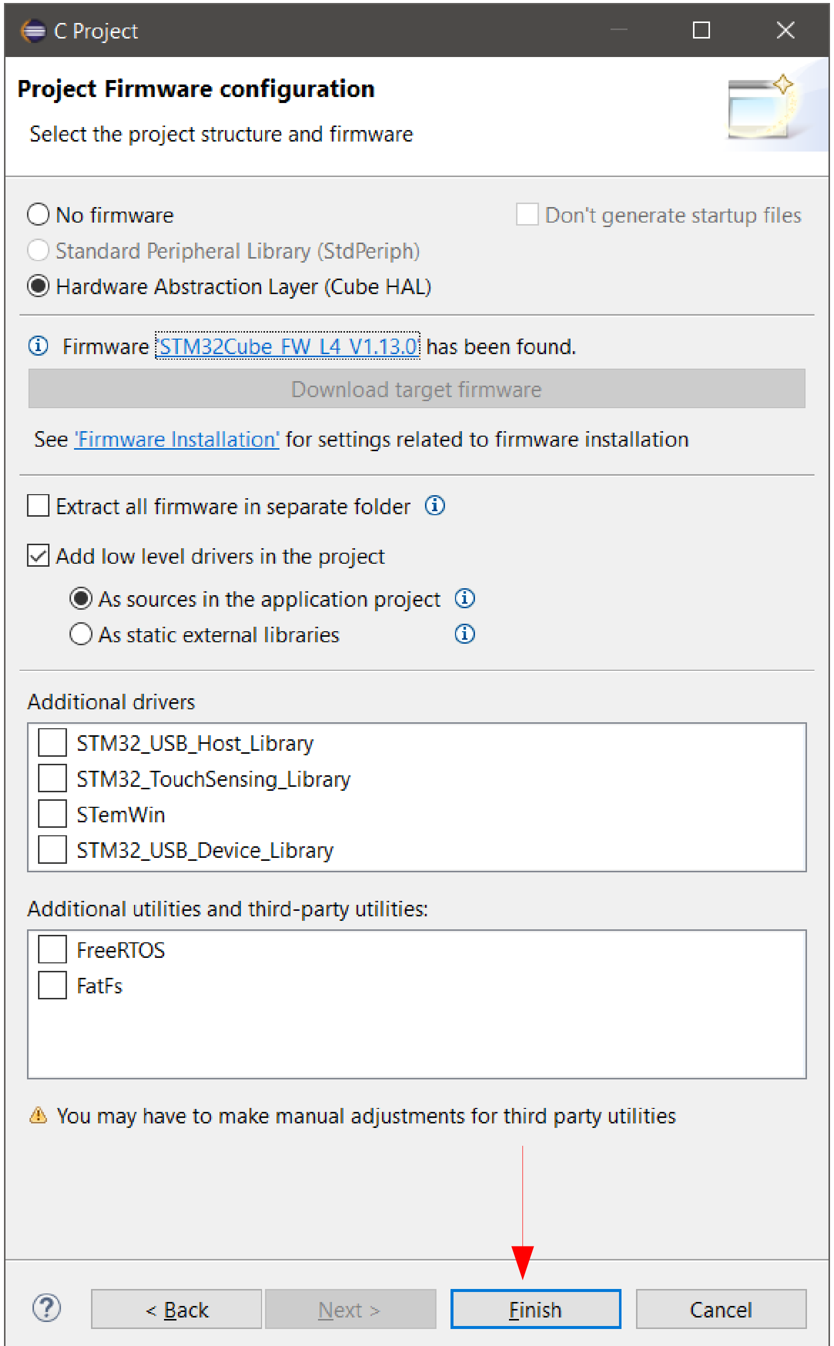
Creating a New Project on System Workbench

- A **`License Agreement`** will pop-up, check **`I accept the agreement`**, and click on **`OK`**.



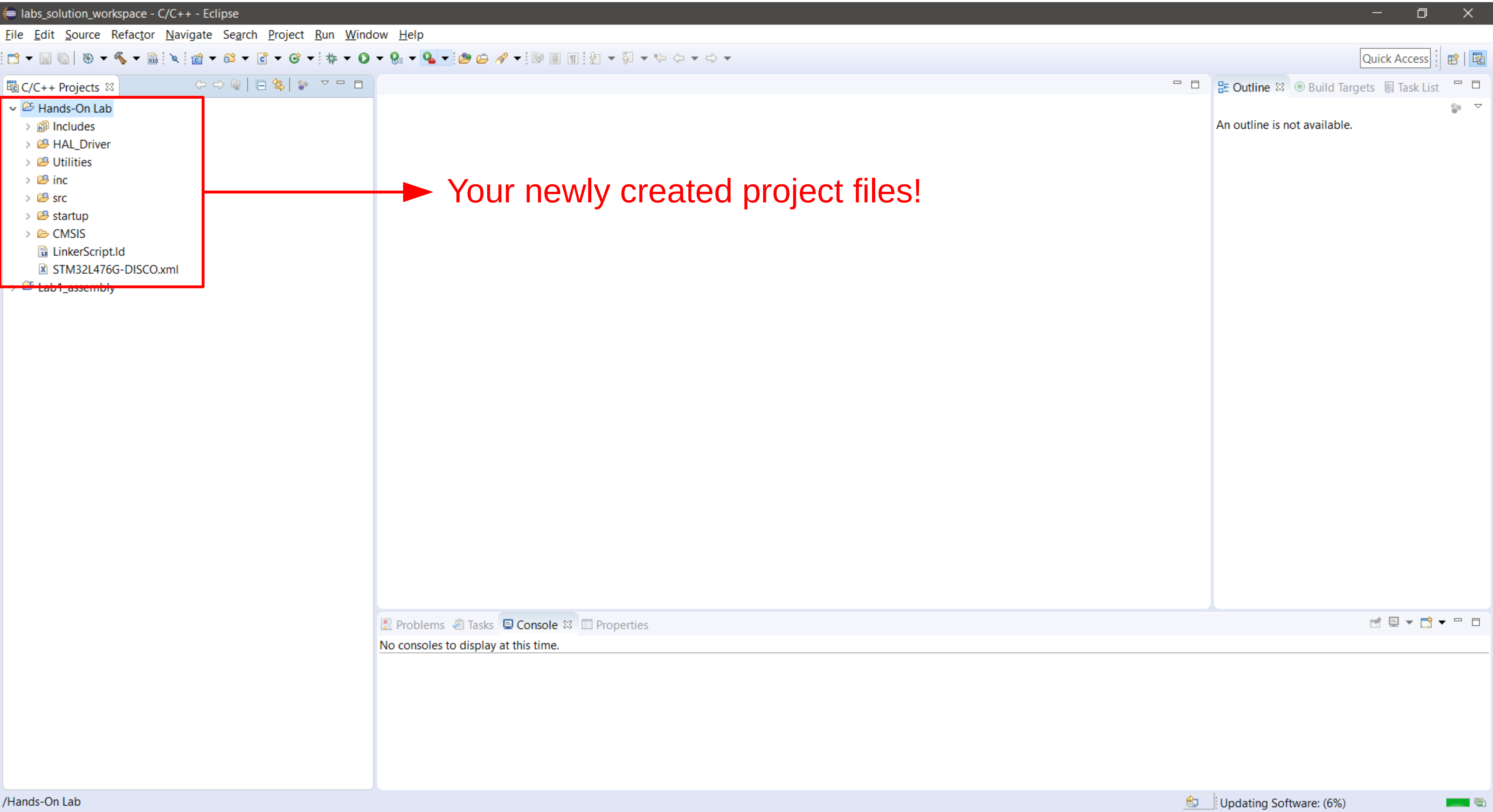
Creating a New Project on System Workbench

- Once the download is completed, you can click on **`Finish`**.
- Do not change the other configurations!

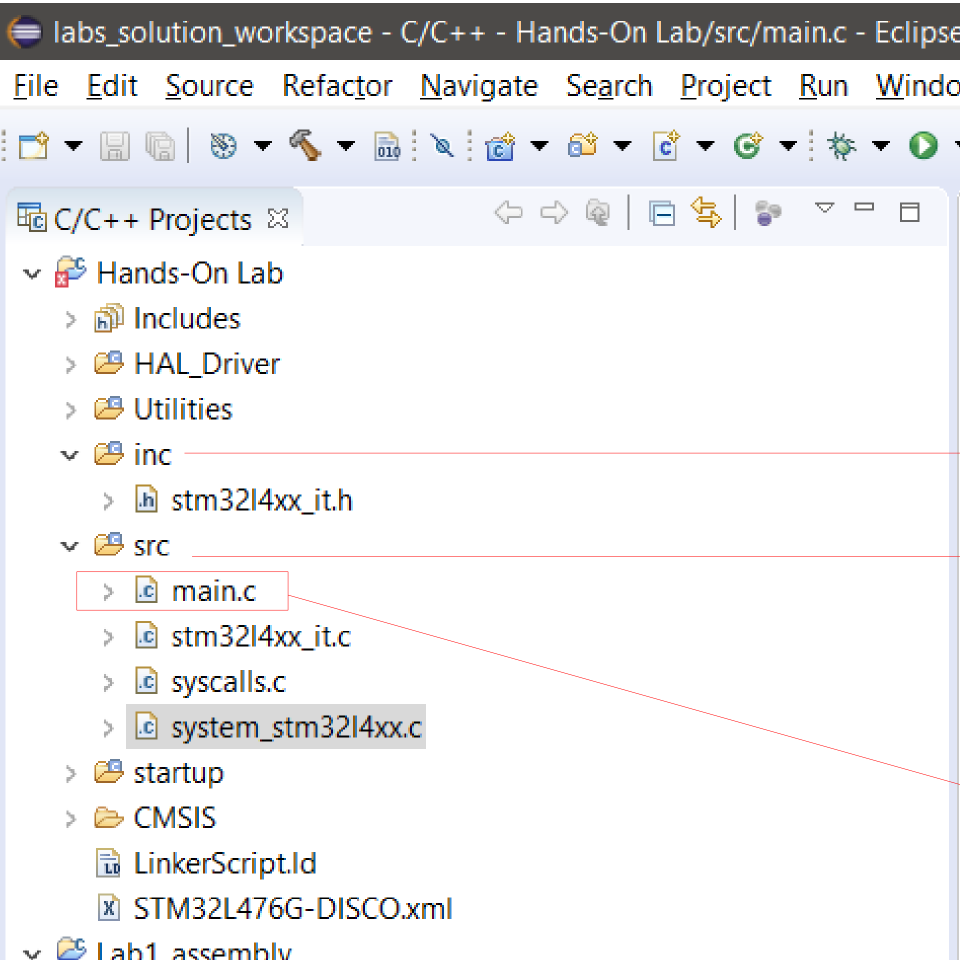


Creating a New Project on System Workbench

- Now, your project is created and you will have access to all code files on the panel on the left in the IDE.



Creating a New Project on System Workbench



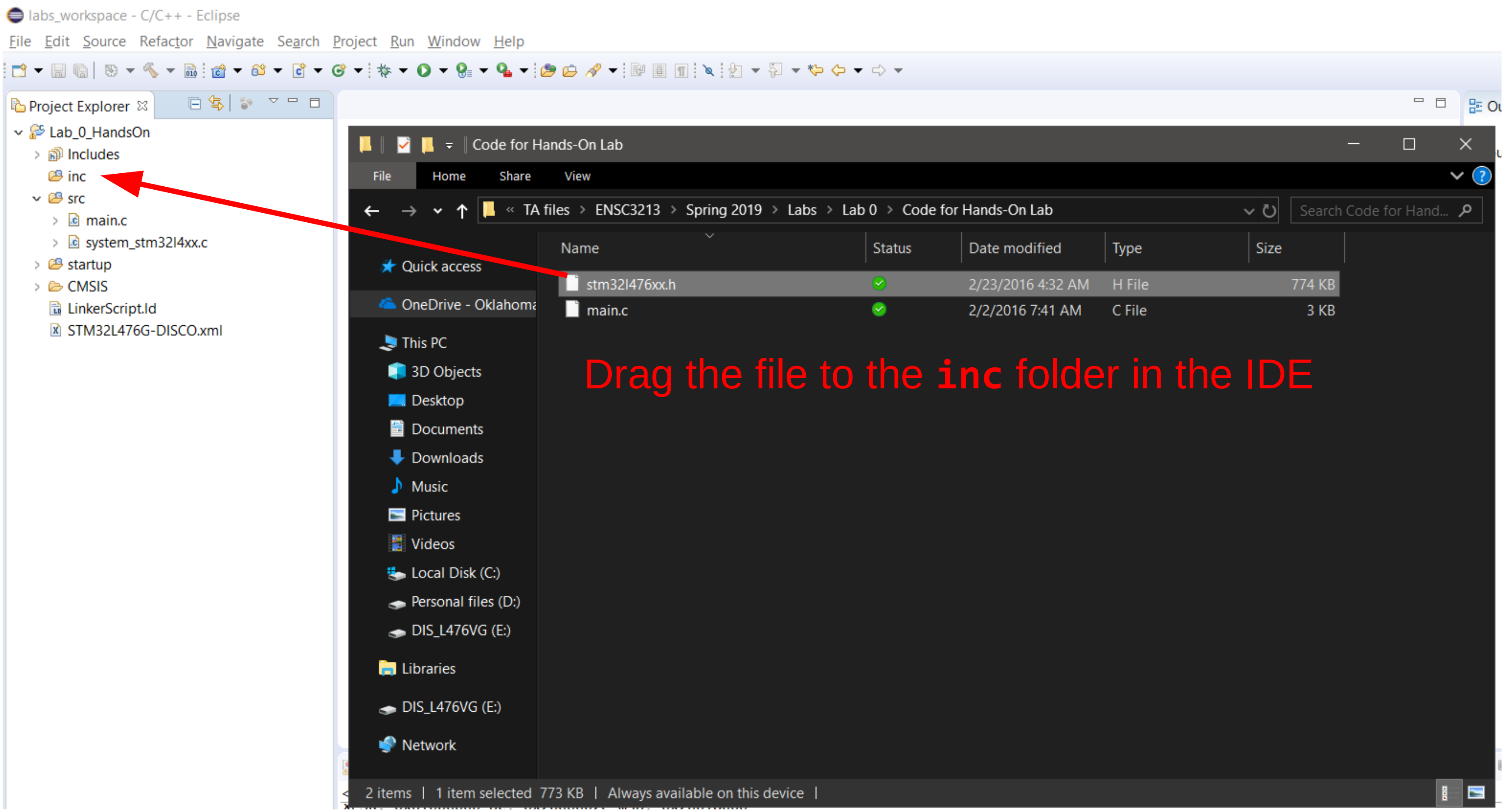
The **`inc`** folder will contain all our **`.h`** files.

The **`src`** folder will contain all our **`.c`** files and **`.s`** files.

Our **`main.c`** will be created inside the **`src`** folder.

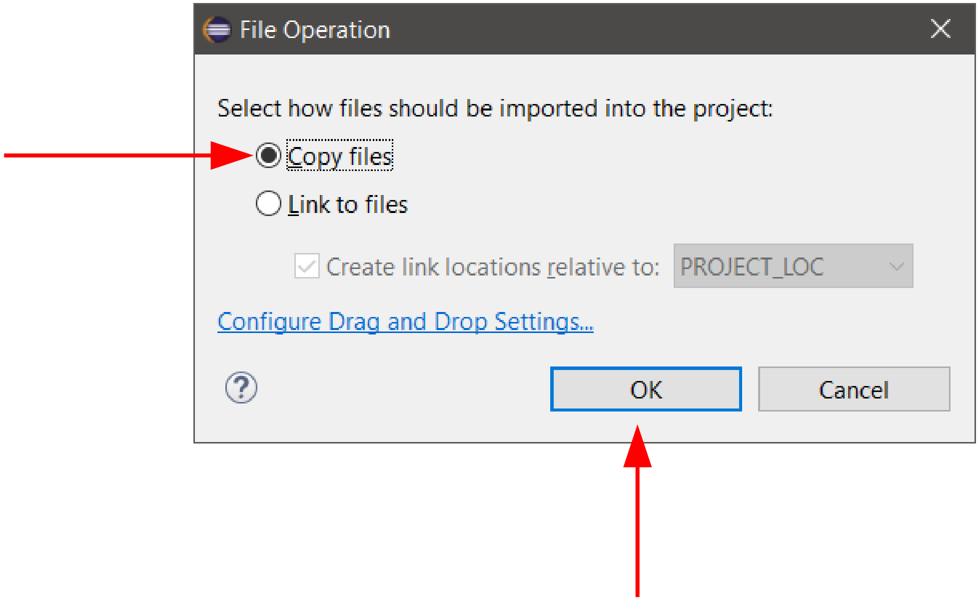
Creating a New Project on System Workbench

- The final step is to move the given file **`stm32l476xx.h`** to the **`inc`** folder. You can do this by clicking and dragging the file.



Creating a New Project on System Workbench

- The IDE will ask if you want to copy or link the file. Click on **`Copy files`** and, then, on **`OK`**.

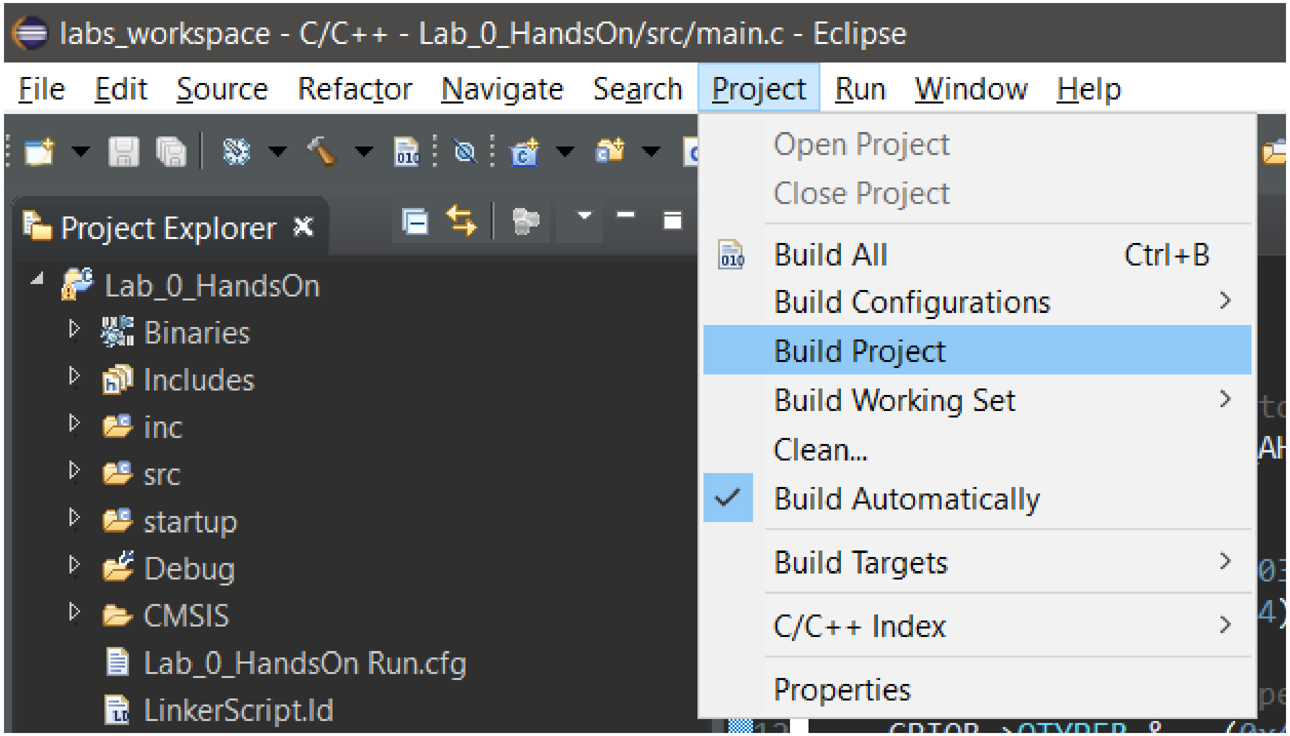


Creating a New Project on System Workbench

- Now, you can double click on the file **`main.c`** and start writing your code! Finally!

Compiling your code on System Workbench

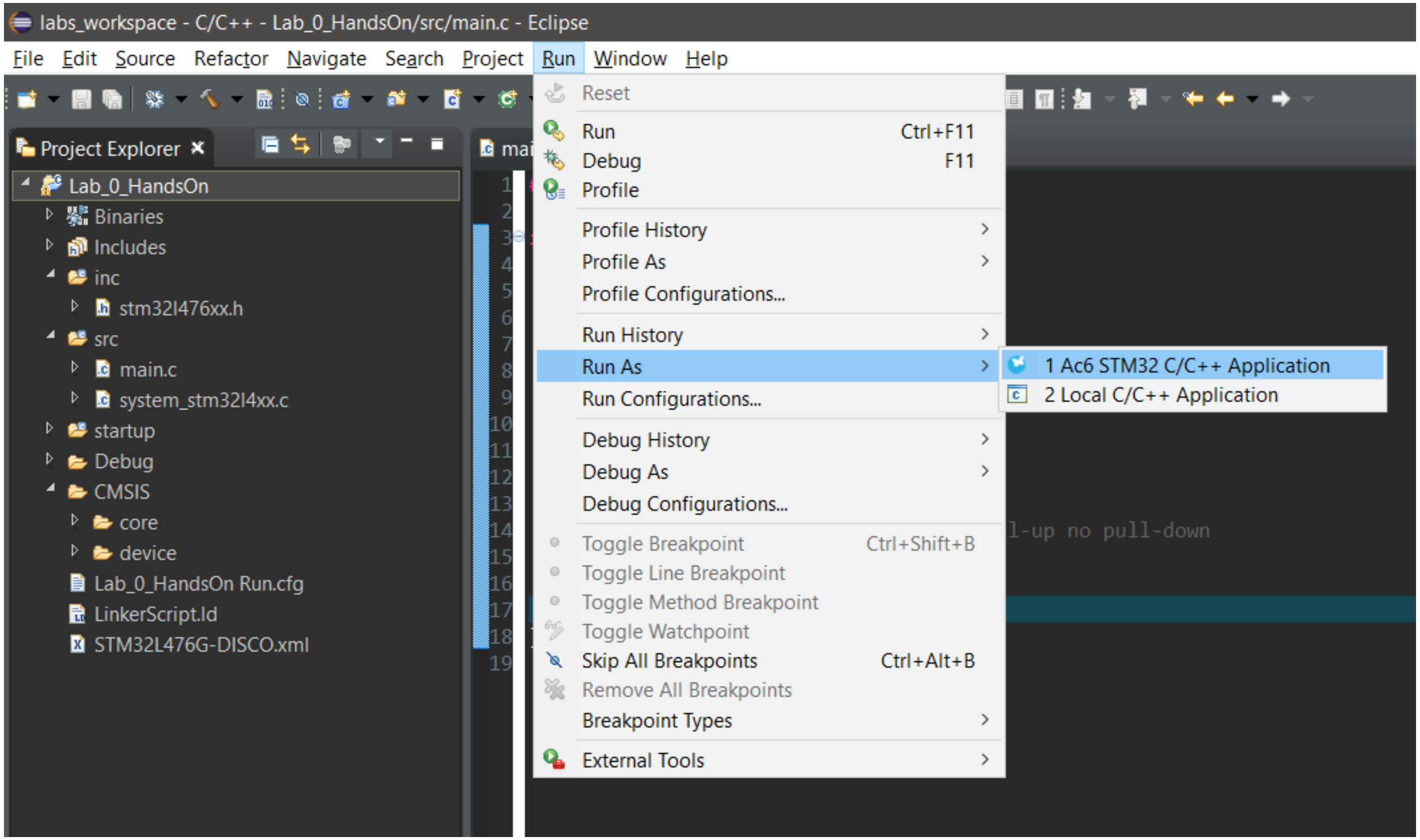
- After you're done writing your code, you will need to **compile** it, and **upload** it to the development kit.
  - To compile, go to **`Project → Build Project`**.



  - If everything is correct with your code, you will see the message **`Build Finished`** and no errors in the **`Console`** window.

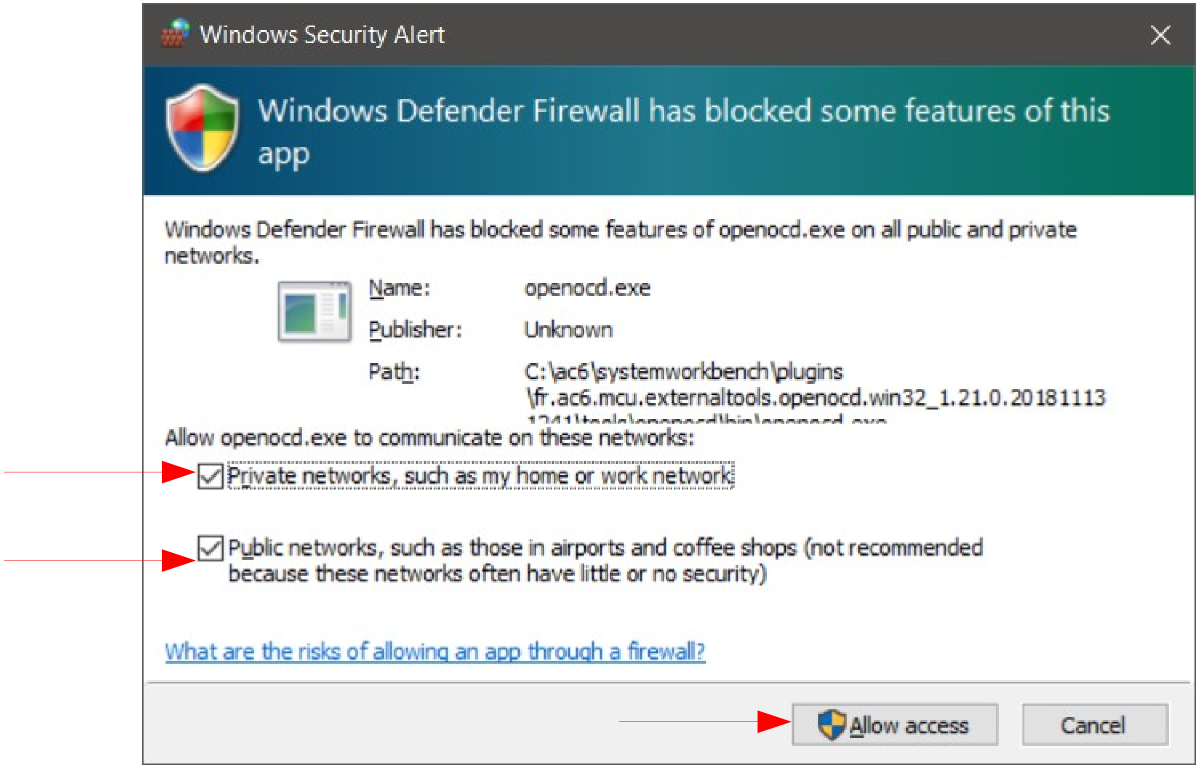
Uploading your code using System Workbench

- To upload your newly compiled code, go to **`Run → Run As → Ac6 STM32 C/C++ Application`**.
- This will upload your compiled code and reset the development kit.



Uploading your code using System Workbench

- When uploading, the application may ask for permission to use the network. Make sure you allow access.



Manually Programming and Debugging the Board

- The **`debug`** interface of most development boards, such as **`STM32L4 Discovery Kit`**, often provides a **`USB mass storage interface`**. When a board is connected to a computer, it is automatically mounted as a USB drive.
- To program the board, we only need to copy the generated **`.bin`** file to the mounted USB drive.

Manually Programming and Debugging the Board

- **`OpenOCD`** (**`Open On-Chip Debugger`**) is an open-source software that is widely used for debugging and downloading executables to microprocessors. **`OpenOCD`** runs as a server (also known as a **`daemon`**) on a host computer and serves two purposes:
  - It receives commands from either **`Telnet`** or **`gdb`** via TCP/IP connection.
  - It translates commands received to **`JTAG/SW`** commands, and sends them to the target ARM Cortex-M processor via the hardware debugger.