Setting-up Visual Studio Code default configuration and brief introduction to the integrated debugger

Mingfei Yu for CS-119 (c) and COM-112(a)

May 25, 2023

1 Introduction

So far, we are using *geany* as the editor to write c++ programs in this course, as it provides us a convenient way to set up the compiling commands.

However, with more and more difficult coding tasks assigned to you, you may already find it time-consuming and annoying to detect the bugs in your program: through inefficiently printing out immediate results and comparing them with your expected value, debugging can cost you more time than developing a program!

In other words, congratulation! You have become a more advanced programmer and therefore need a more powerful helper to face greater challenges! We propose to instead use *Visual Studio Code(VSC)*, a code editor developed by Microsoft, as it supports easy-to-use debuggers, as well as a variety of extensions that can possibly help you write code more efficiently.

VSC is already installed in the virtual machine, and considering its popularity, many of you may have been working with it. In this tutorial, we introduce how to get your *VSC* configured once and for ever!

For extensions, our flow here only cover the installation of two for CS-119(c) and two others for COM-112(a), which we believe are helpful:

1) CS-119(c) / sem1: *C/C++* for debugging and formating, and *Code Runner* for a convenient way to compile and execute your code.

2) COM-112(a) / sem2: *Makefiles Tools* for launching the compilation of the project code, and *Live Share* for a convenient way to edit and compile your code simultaneously with your project partner.

2 Setting-up VSCode default configuration on the VM

If you use the course VM, you only need to do the following once. If you are not using the course VM, please refer instead to **Section 2.2**.

2.1 Configuration through running script

Step 1 From Moodle course page, download the provided script *setup.sh*.

Step 2 Open the terminal application via, for example, the ”show application” button at the bottom of the sidebar. Assume that the downloaded file is under the directory: ”~/Download/”, then enter following commands to run the script:

```
bash ~/Download/vs_setup.sh
```

If, unfortunately, there is error message printed out, try to find a solution or post a question through *EdStem*; Otherwise, we are almost there!

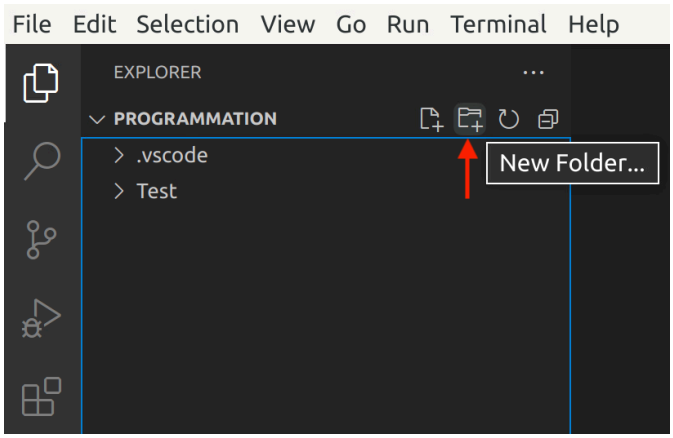
Step 3 Close the terminal window and then launch *VSC*.

For the pop-up "Get Started" window, either do the customization as you wish or directly close it.

From the tool bar on the upside of the GUI, select: "File - Open Folder - Programmation". In this way, *Programmation* would be regarded as the working space.

The script has automatically generated a *.vscode* file folder, as well as 3 configuration files(json files) under "Programmation/.vscode" for C++ compiler, debugger and *VSC* extensions. These configurations would be automatically loaded and enabled by *VSC*.

Step 4 To create a new project, we suggest you to create a new file folder under "Programmation".



For example, to create a project *Helloworld*, which consists of only one cc file *helloworld.cc*, you can: click the "New Folder" button to create a folder named *Helloworld* under *Programmation*, select the created *Helloworld* folder and click the "New File" button to create *helloworld.cc*. Then, when compiling and debugging *helloworld.cc*, the mentioned configurations would take effect automatically, which means you can easily start playing with the compiler and debugger!

Step 5 Now you have get *VSC* ready! Please refer to **Section 3** for a quick hand-on.

2.2 Configuration by hands if you are NOT using the provided VM

This section is for those who are not using the course VM. Please note that we cannot support you more than the content of this section as we focus offering support for the course VM. Make sure that *g++* and *gdb*(for Windows) / *lldb*(for MacOS) are **already available** on your laptop. For Windows, *g++* and *gdb* are included in *MinGW*; For MacOS, *g++* and *lldb* come with Apple's XCode tools package. If you have already got configuration done following **Section 2.1**, please skip to **Section 3**.

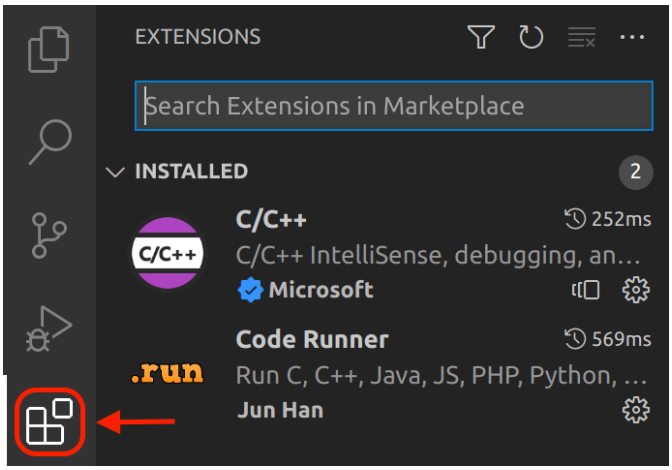
Step 1 Based on the operating system you are using, download the corresponding versions of configuration files from Moodle course page. There would be 3 json files in total: *launch.json*, *tasks.json* and *settings.json*.

Step 2 Launch *VSC*, and open the folder you would like to specify as the working place. Assume you are putting all your codes in a folder named *Programmation*, from the tool bar on the upside of the GUI, select: "File - Open Folder - Programmation".

Step 3 Create a file folder under *Programmation* and name it as *.vscode*(refer to the figure in **Section 2.1** Step 4 for where the "New Folder" button is). Move the 3 configuration files downloaded in Step 1 to *.vscode*.

Step 4 For your projects, put them under *Programmation*, just next to *.vscode*. No worry, the configurations in *.vscode* would automatically take effect. Better **NOT** to put your projects under *.vscode*, which is not necessary and may make things in a mess.

Step 5 Install extensions by clicking the "extensions" button: As shown in the figure, *C/C++* and *Code Runner* are the two extensions that we recommend. But you can always further install more extensions as you wish.



Step 6 Create a file folder under *Programmation* and name it as *Test*, then put the provided *test.cc* under *Test*.

3 Use Extensions for CS-119(c)/sem1

As you may have already noticed, in the *Programmation* folder, there is a *Test* project created, which consists of one file *test.cc*. In this section, we show some basic but handy usages of the installed extensions, while playing with this simple example.

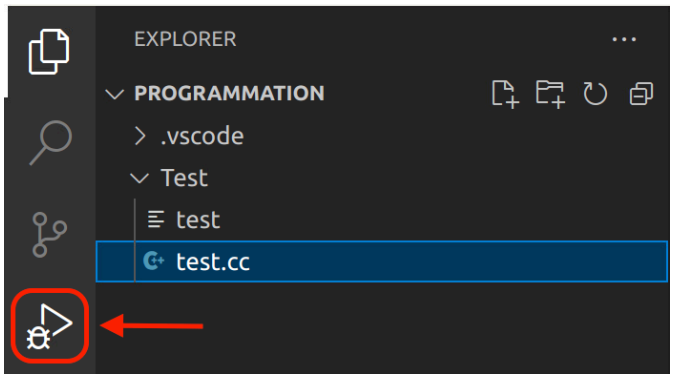
1. As has been highly emphasized during the lectures, it is important to raise the habit of keeping your codes tidy and neat. Now, even if your code is in a mess, *VSC* can help!

   To play with this function, you can at first randomly disorganize the code by, for example, inserting arbitrary numbers of space at the beginning of different lines. Then, type "ctrl+shift+I", or right click and select "Format Document". You would find the code automatically organized! But, keep in mind, don't rely too much on this handy function, and always keep your code in the format requested by the course conventions)

2. To quickly compile your code, use shortcut "ctrl+shift+B". It is recommended to compile your code time to time, especially when you are developing a large-scale project: detecting bugs as early as possible can save you a lot of time.

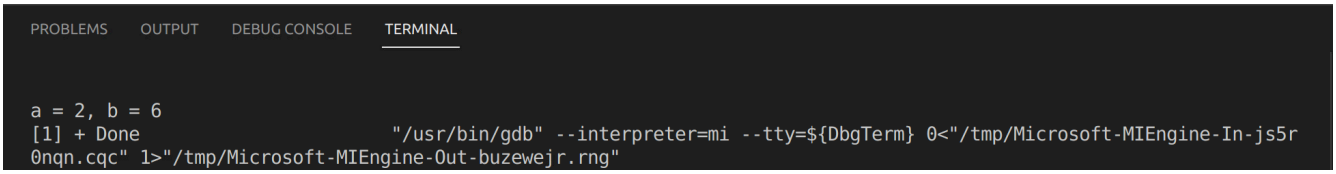
   Also, since we have installed *Code Runner*, when you are quite confident with your code that there are neither syntax error nor semantic error, you can give it a quick run via right click and select "Run Code", or use "alt+ctrl+n" to see the execution result.

3. Time to learn how to debug now! At first, find the "Run and Debug" button on the left hand side of the GUI and click it. Similarly, click the same button when you want to exit the debugging mode.



Then, click the green triangle as shown below, or use shortcut "F5", to start debugging.

After waiting for several seconds, we would find the output of the program in the internal terminal:



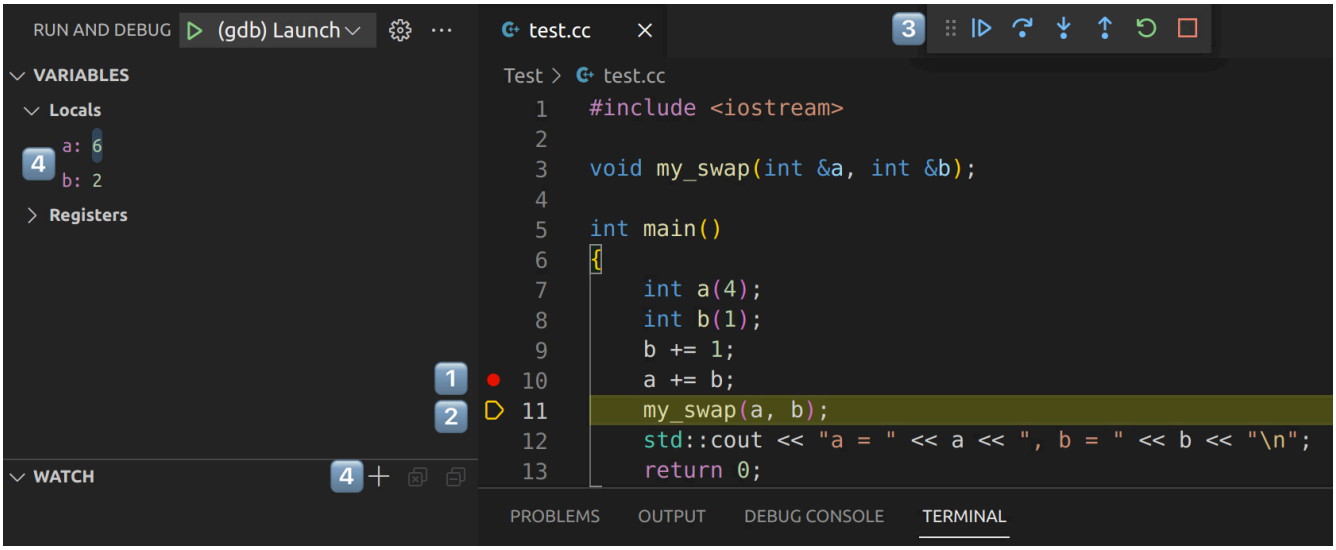
(1) When developing programs, it is usual that the output is not consistent with our expectation(semantic error, but not syntax error). In those situations, an efficient way to debug is to run the code line by line and trace the value of some variables. To do this, set a breakpoint by clicking the space before the line number where you would like the program to stop. As an example, as the red point indicates, we set a breakpoint at line 10.

(2) With breakpoints set, if we start debugging, something different would happen: this time, nothing is printed out in the terminal, and the execution stops at line 10. Notice that the line highlighted in yellow(line 11) denotes **the next line to be executed**.

(3) To execute the program line by line, you can make use of the tool bar, which appears only if we are debugging.

Among these buttons: "Continue" would run the code until next breakpoint; Both "Step Over" and "Step Into" allow us to execute the program line by line, but when encountering a line that calls a function, "Step Over" would regard the execution of the function body as a whole, while "Step Into" would execute the function body also in a line by line manner.

In our example, using "Step Over", the line to be executed after line 11 would be line 12, because the execution of the my_swap function(line 17 to line 21) is regarded as one step(line 11); Instead, if we use "Step Into", line 17 would be the next line where the program stops.



(4) Find the temporary value of the variables in the window on the left hand side of the GUI. If the variables that you are concerned about are not there, you can manually add them by clicking the "+" button in the "WATCH" window.

Also, by hovering the cursor over the variable that you are interested in, its temporary value would be available as well.

While, this brief tutorial only offers a quick view to debug c++ code, we recommend you to also check the tutorial provided by Microsoft: `https://learn.microsoft.com/en-us/visualstudio/debugger/getting-started-with-the-debugger-cpp`.

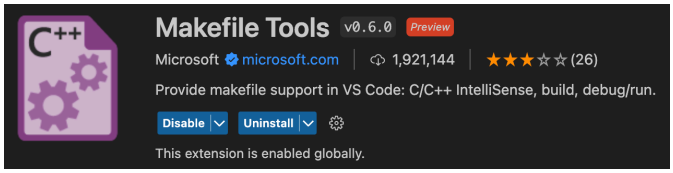
Notice that it is totally fine if you find yourself not able to completely figure out everything in their example, as some points there can be beyond your current knowledge. The target here is just master the most basic usages of a debugger, in order to develop your program more efficiently:)

4 Advanced Usage of Extensions for COM-112(a)/sem2

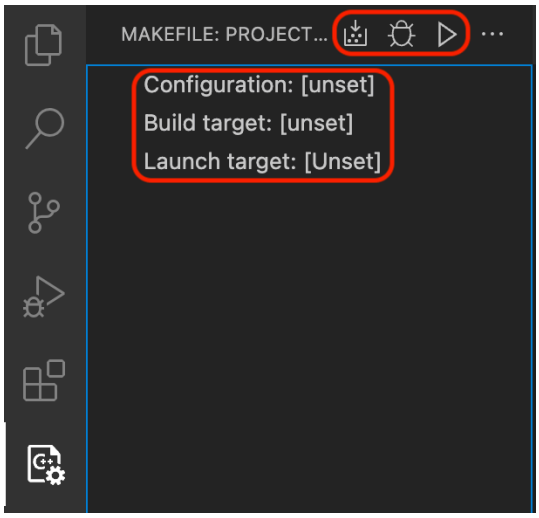
4.1 Use of *Makefiles*

In this semester, since we are learning the concept of *modular programming*, as well as the usage of *Makefile*, it is necessary to think about how to make use of *Makefiles* when developing projects using *VSC*.

The first thing that may come into your mind, which is also what most of you are doing, is that, entering the *make* command into the terminal in *VSC*. This is definitely a convenient and straightforward solution, but the goal here is to **handle Makefiles using the GUI of *VSC***. Our suggestion is to use the following extension, *Makefile Tools*. If you are already used to making use of the terminal to compile your program, it is **not compulsory** to follow the configuration below.



After installation, the icon of this extension can be found in the sidebar:



There are three buttons on the top side of the window, respectively corresponding to the *build*, *debug* and *run* commands, which are quite similar to the "Run and Debug" function of *VSC* that is already introduced in Section 3. Additionally, there are three different items of configurations: *Configuration*, *Build target* and *Launch target*, by customizing which we can instruct *VSC* to make use of the Makefile in the way we want.

Hereafter, we focus on figuring out **the duties/usages of these three configuration items**. The example here is the source code of the project *prog*, which is provided in exercise 0.

1. *Configuration* is for **passing arguments to the *make* utility**, i.e., **setting the configuration for build**. It refers to the *make* command configurations defined in **.vscode/settings.json**.

   When there is no such configuration made in the json file, when clicking the pencil button in this item, "Default" would be the only available option. As an example, we define the following configurations in **.vscode/settings.json**:

```
"makefile.configurations": [
    {
        "name": "Default",
        "makeArgs": []
    },
    {
        "name": "Print make version",
        "makeArgs": ["--version"]
    }
]
```

   One more *make* command configuration, *Print make version*, now becomes available, which adds the "–version" argument to the make utility. With this make command configuration, every time we build our program using *Makefile Tools*, the version of *make* would be printed out. In my case, the following information is displayed in the terminal:

```
GNU Make 3.81
Copyright (C) 2006  Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
```

"Default" is possibly the most common *make* command configuration, but you can explore different arguments for your own need. A list of valid arguments can be found in `https://www.gnu.org/software/make/manual/html_node/Options-Summary.html`.

2. *Build target* is a concept that we have learned about in exercise 0 at the beginning of this semester. In our example, there are three targets defined in the Makefile: *prog*, *depend* and *clean*, which are the three optional configurations here. In this example, to generate the executable file, we select *prog* as the build target.

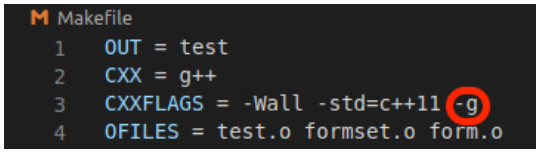
3. *Launch target* requires us to point out which executable file would be run if we click the debug or run button. In our example, *prog* is apparently the only option for this configuration.

In conclusion, the *Makefile Tools* extension enables us to **make use of Makefiles using the GUI of *VSC***, eliminating the necessity of explicitly entering the *make* command into terminals.

4.2 Call the Debugger using *Makefile Tools*

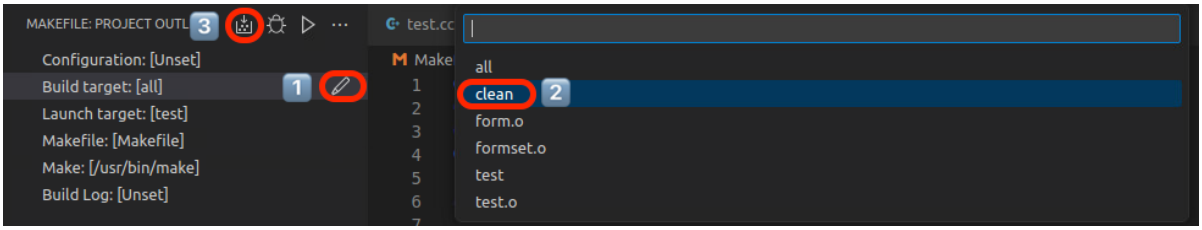
In this section, we introduce how to call the debugger in *VSC* using the *Makefile Tools* extension. To provide an easy-to-follow tutorial, we take the project *type_parametre_form* in Week4's exercise as an example, the source code of which can be found at: `https://moodle.epfl.ch/mod/resource/view.php?id=1016209`.

1. First of all, make sure that the "-g" flag is included in the "CXXFLAGS" list of your Makefile. In our example, we configure the "CXXFLAGS" list as: The "-g" flag is important if we want to use



the debugger because it tells the compiler to generate those intermediate files that *gdb* requires.

2. If you have already compiled your program before, do not forget *make clean* before proceeding to the following steps. To do a *make clean*, you can either (a) directly execute the *make clean* command in a terminal, **or** (b) click the three items highlighted using red squares in the figure below in order:

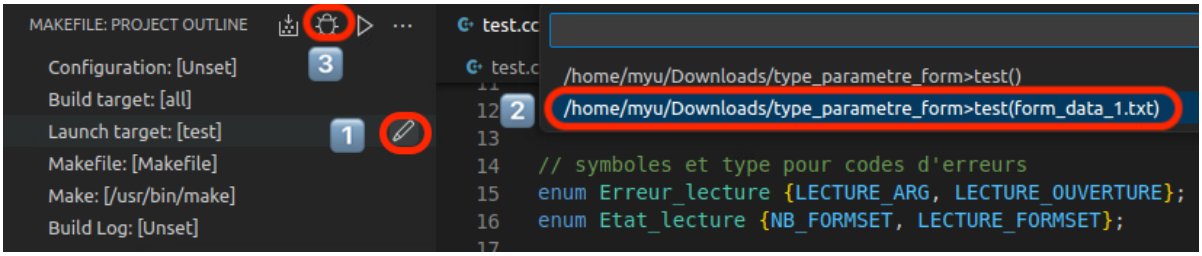


3. If you do *make clean* in the (b) manner in the previous step, do not forget to reset the *build target* back to *all*. Now, if you press the *Debug the selected binary target* button, your program would be built first, and the debugger would be activated sequentially — then you can play with the same tricks introduced in Section 3 to debug your program using the debugger!

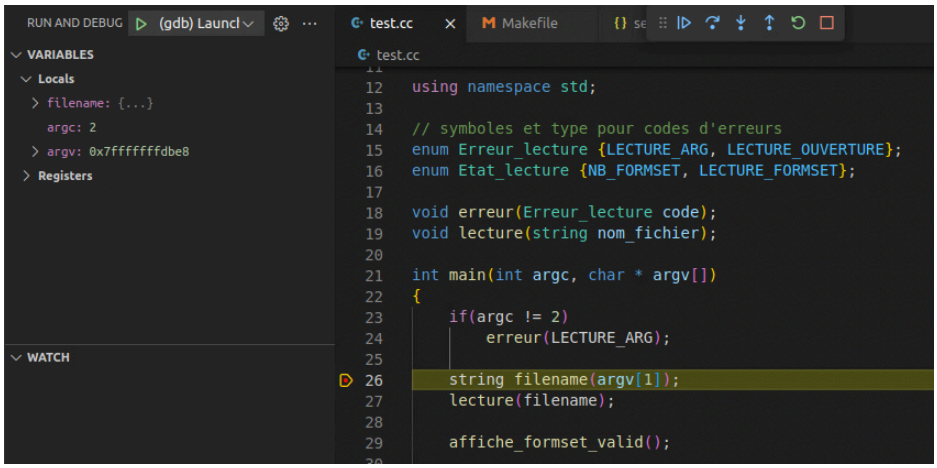
4. If you need to pass an argument/arguments to your executable file: configure the *settings.json* under the *.vscode* folder by writing down the argument(s) you want to parse in the "binaryArgs" entry of "makefile.launchConfigurations":

In our example, since our executable file *test* takes in the name of a file as the argument, I fulfill the square brackets with *"form_data_1.text"*. If you would like to parse more than one argument, put them all between the square brackets and separate them using commas.

5. Then, if we click the pencil icon next to *Launch target*, we would find one more option for the launch target, which corresponds to our configuration in the previous step. Select the launch target that you would like to debug and then press the debug button as before. An example, if we put a breaking



point at line 26, we witness that we have successfully invoked the debugger, and the execution of the program is stopped exactly at line 26, together with the familiar debugging tools that we have seen before in Section 3:

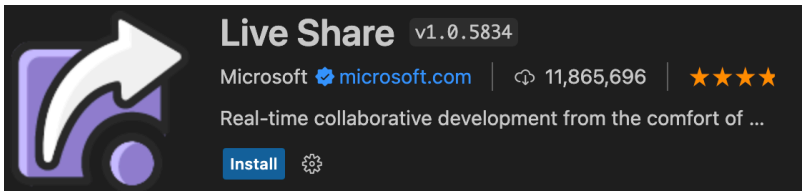


4.3 Collaborate with your project partner using *Live Share*

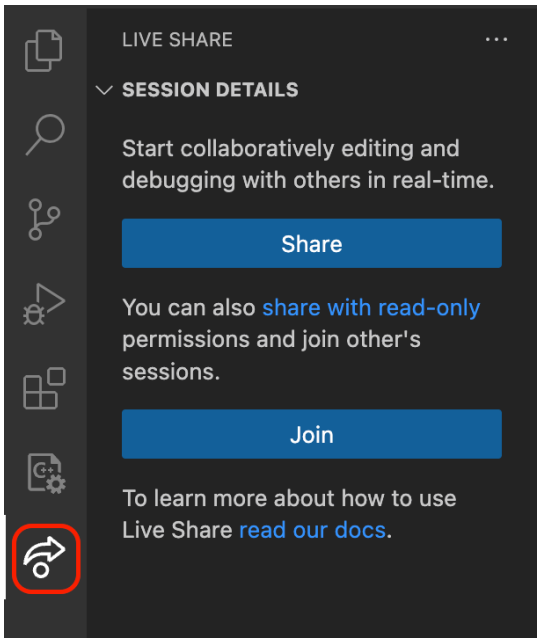
As the semester progressed, it is glad to notice that you have become more experienced programmers. However, since the projects you are developing are becoming more and more complicated, **how to collaborate with your project partner more efficiently** is a natural question.

While there might be numerous solutions, we introduce one of them here: *Visual Studio Live Share*. Again, please notice that we are just offering an option and it is **not compulsory** for you to follow this suggestion.

(1) To get started, search and download the *Live Share* extension in *VSC*.



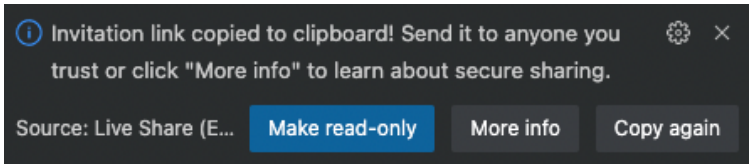
After installing the extension, a new icon would appear in your sidebar:



As can be learned from the figure, using *Live Share*, we can either: generate a link to invite your project partner to join your session; or, join someone's session using an invitation link from him/her.

(2) To create a session for collaboration, let us click the "Share" button shown in the previous figure. You have the option to make this session *ready-only*, if you would like to prevent your invitee from editing the code for some reasons. You can also configure the session to be read-only later. For a ready-only guest, although he/she cannot edit the codes, he/she is still able to debug the program.

VSC may ask you for your Microsoft or Github account, if you have not yet signed in. Then, a message would pop up, notifying you that the invitation link to your session is ready:

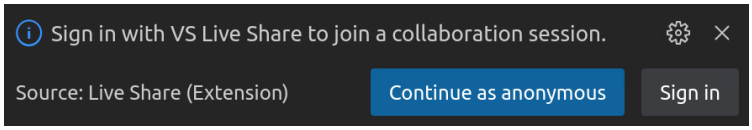


As shown in the message, the invitation link is already in the clipboard of your computer and you can easily distribute it to your project partner by email, social software, etc.

Hereafter, let us go through a typical usage of *Live Share*. To give an example, I am regarding my laptop as the host, and my VM as the guest who has received the invitation link from the host.

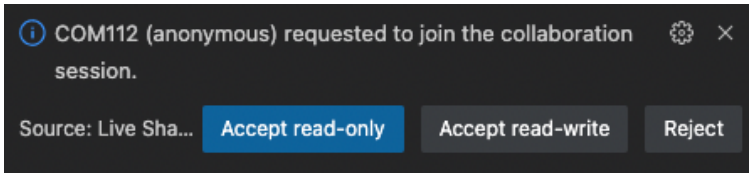
(3) The simplest way for you (as a guest) to join the shared session might be opening the link using a web browser. A request would pop up, asking for your permission to open the link. After confirming the request, you can choose to work with either the web-based or desktop-based version of *VSC*. If the *Live Share* extension is not yet installed, the installation would start automatically.

As a guest, it is not required to log in your Github or Microsoft account and you can contribute anonymously:



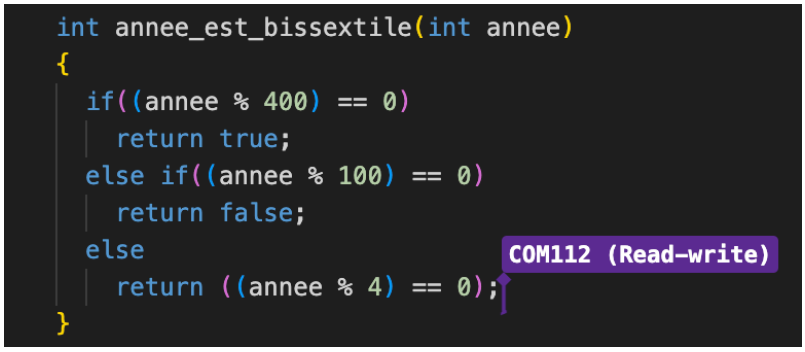
For clarity, in this example, we join the session as anonymous and specify the name as "COM112".

At the same time, on the host's side, a message would pop up, asking the host to approve the join request from COM112, as well as specify the authority of the guest (*read-only* or *read-write*).



If the host gives the green light to the join request, we reach the stage where there are two parties (the host and the guest, COM112) in this shared session, ready to collaborate with each other.

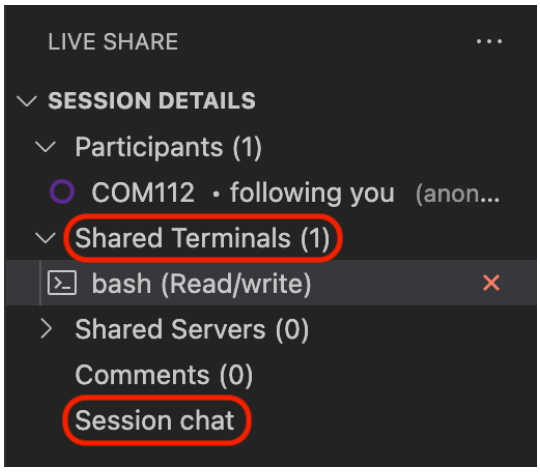
(4) Now, as long as this session is not terminated by the host, the guest can access the program as if it is locally available. Still use the source code of the project *prog* as an example:



Each party in the session can learn which lines are others working on in real-time: in our example, the purple cursor and name label tell the host that COM122 is working on this *annee_est_bissextile* function.

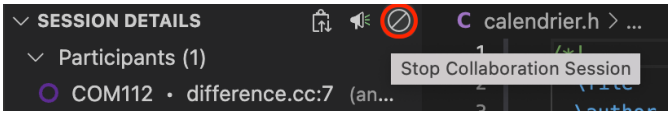
Similarly, any configuration for debugging, such as the insertion of breakpoints and adding variables to the watch window introduced in Section 3, would be **simultaneously visible** to all the parties in this shared session.

Besides, by clicking the "Share terminal..." option, the host can create a shared terminal:



Using this shared terminal, the team can jointly evaluate the program by giving it various inputs. Additionally, the "Session chat" option in the menu provides the team a convenient chat room to exchange ideas efficiently.

(5) To terminate the shared session, the host just have to click the following button:



Then, all the changes made to the program are **still available to the host**, but the guests cannot get access to the codes any longer.

Hope this brief introduction to the *Live Share* extension can help you collaborate with your teammate on the project more efficiently. For those interested in learning more about this extension, please refer to `https://code.visualstudio.com/learn/collaboration/live-share`.