Lab: building with make and debugging in VS Code

1 Objective

Move your *Lab shapes* project into the course project template and stop typing gcc commands by hand. At the end of this lab you will have:

- built a multi-module project with `make`, from the terminal and from VS Code;
- seen which files `make` recompiles after a change, and understood why;
- used the *Problems* panel to go from a compiler message to the line of code;
- used the debugger (breakpoints, step by step, variables, call stack) to find a bug that the compiler does not report.

> Estimated time: 60–90 minutes. Keep following the Barr-C style guide, as in *Lab shapes*.

2 Requirements

- *Lab shapes* finished: the files `area.h`, `area.c`, `volume.h`, `volume.c` and `main.c` in the folder `lab_shapes`. If your version does not work, ask your instructor for the reference version.
- The project template `c-template.zip` provided by your instructor.
- The tools `make` and `gdb`. Open the **MSYS2 UCRT64** terminal and run the following command. It installs only the packages that are missing:

```
$ pacman -S --needed mingw-w64-ucrt-x86_64-gdb mingw-w64-ucrt-x86_64-make
```

  Then check them in a new Windows command prompt:

```
C:\...> mingw32-make --version
C:\...> gdb --version
```

> From now on VS Code builds the whole project through `make`, so all the `.c` files are compiled. The ▷ *Run C/C++ File* button is still forbidden: it builds only the active file and ignores the `Makefile`.

3 Starting point

1. Extract the template into your projects folder and rename it `lab_make`. These commands assume that you saved `c-template.zip` in your *Downloads* folder:

```
C:\...> cd %USERPROFILE%\c_projects
C:\...> tar -xf %USERPROFILE%\Downloads\c-template.zip
C:\...> ren c-template lab_make
C:\...> cd lab_make
```

2. Delete the example files of the template and copy your *Lab shapes* files. The `.c` files go to `src`, the `.h` files go to `include`:

```
C:\...\lab_make> del src\main.c src\greeting.c include\greeting.h
C:\...\lab_make> copy ..\lab_shapes\*.c src
C:\...\lab_make> copy ..\lab_shapes\*.h include
```

3. Open the folder in VS Code, either with the command `code .` or with *File → Open Folder…*. If VS Code asks whether you trust the authors of the files, answer *Yes*. If it offers to install the recommended C/C++ extension, accept.
   Your project must look like this:

```
lab_make
├── Makefile
├── README.md
├── .vscode
│   ├── c_cpp_properties.json
│   ├── extensions.json
│   ├── launch.json
│   └── tasks.json
├── include
│   ├── area.h
│   └── volume.h
└── src
    ├── area.c
    ├── main.c
    └── volume.c
```

> The `Makefile` compiles every `.c` file in `src` and looks for headers in `include`. You never need to edit it, nor the files in `.vscode`. If you add a new module, just put its files in the right folder.

4 Part 1: build with make from the terminal

1. Open a terminal inside VS Code with *Terminal → New Terminal*. It starts in the project folder.
2. Build the project:

```
PS C:\...\lab_make> mingw32-make
if not exist build mkdir build
gcc -std=c99 -Wall -Wextra -pedantic -g -Iinclude -MMD -MP -c src/area.c -o
build/area.o
gcc -std=c99 -Wall -Wextra -pedantic -g -Iinclude -MMD -MP -c src/main.c -o
build/main.o
gcc -std=c99 -Wall -Wextra -pedantic -g -Iinclude -MMD -MP -c src/volume.c
-o build/volume.o
gcc build/area.o build/main.o build/volume.o  -o build/app.exe
```

These are the commands you typed by hand in *Lab shapes*: three compilations with `-c` and one link. The new options are `-g`, which adds the information needed by the debugger; `-Iinclude`, which tells gcc where the headers are; and `-MMD -MP`, which you will meet in Part 2. All the generated files go to the `build` folder.

3. Build and run the program. The output must be the same as in *Lab shapes*:

```
PS C:\...\lab_make> mingw32-make run
build\app.exe
Areas
  circle    r = 1.5           :    7.069
  rectangle 4.0 x 2.5         :   10.000
  square    side = 2.0        :    4.000
Volumes
  cylinder  r = 1.5, h = 3.0  :   21.206
  cuboid    4.0 x 2.5 x 3.0   :   30.000
  cube      side = 2.0        :    8.000
```

4. Run `mingw32-make` again, without changing anything:

```
PS C:\...\lab_make> mingw32-make
mingw32-make: 'build/app.exe' is up to date.
```

No command was run. Keep this in mind for question 1.

5. Delete everything that was generated, then build again. All the files are compiled again:

```
PS C:\...\lab_make> mingw32-make clean
PS C:\...\lab_make> mingw32-make
```

5 Part 2: what does make recompile?

1. In VS Code, change only a comment in `src/area.c` and save the file (`Ctrl+S`). Build again and note which commands `make` runs.
2. Now change only a comment in `include/area.h`, save it and build again. Note which commands `make` runs this time.
3. Open the file `build/volume.d` in VS Code. It was written by gcc because of the options `-MMD -MP`. Its first line is:

```
build/volume.o: src/volume.c include/volume.h include/area.h
```

`make` compares the modification dates of the files. A target, such as `build/volume.o`, is rebuilt only when one of the files it depends on is newer than the target. The `.d` files tell `make` which headers each `.c` file includes.

6 Part 3: build from VS Code and read the messages

1. Press `Ctrl+Shift+B`. VS Code runs the *build* task, which is simply `mingw32-make`, and shows its output in the terminal panel.
2. Add this unused variable to `main()`, just after the declaration of `side`, save and press `Ctrl+Shift+B`:

   ```
       int unused;
   ```

   gcc compiles `main.c` again and reports a warning:

   ```
   src/main.c: In function 'main':
   src/main.c:15:9: warning: unused variable 'unused' [-Wunused-variable]
   ```

   Open the *Problems* panel (*View* → *Problems*). The warning is listed there and the variable is underlined in the editor. Click the warning in the panel: VS Code jumps to the line.
3. Press `Ctrl+Shift+B` once more, without changing anything. Is the warning reported again? Keep the answer for question 4. Then delete the line `int unused;` and save.
4. Remove the `;` at the end of the `return` line of `area_rectangle()` in `src/area.c`, save and build. This time it is an error:

   ```
   src/area.c:17:26: error: expected ';' before '}' token
   ...
   mingw32-make: *** [Makefile:46: build/area.o] Error 1
   ```

   The line numbers may be different in your file. `make` stops at the first file that does not compile and the executable is **not** updated. Put the `;` back and build again.

A warning does not stop the build, so the program runs anyway. In this course, a warning must be fixed exactly like an error.

7 Part 4: use the debugger

7.1 Follow the calls step by step

The debugger is controlled with these keys:

- `F5`: start the program in the debugger, or continue until the next breakpoint.
- `F10` (*Step Over*): run the current line; if it calls functions, run them completely.

- `F11` (*Step Into*): run the current line, entering the first function it calls.
- `Shift+F11` (*Step Out*): finish the current function and return to the caller.
- `Shift+F5`: stop the debugger.

1. Open `src/main.c` and set a breakpoint on the line `printf("  cube ...`: click on the left of its line number. A red dot appears.
2. Set a second breakpoint on the line `return 0;`.

> On Windows the program runs in a separate console window, which closes as soon as the program ends. The breakpoint on `return 0;` keeps it open so that you can read the output.

3. Press `F5`. VS Code builds the project and starts the program, which stops on the `cube` line. Look at the *Variables* panel on the left: you see the local variables of `main()` and their values.
4. Press `F11` three times. You enter `volume_cube()`, then `area_square()`, then `area_rectangle()`. After each step, look at the *Variables* and *Call Stack* panels.
5. Stopped in `area_rectangle()`, take a screenshot of VS Code showing the code, the *Variables* panel and the *Call Stack* panel. You will need it for question 5 and for the submission.
6. Press `Shift+F11` twice and watch how you return through the calls. Then press `F5` to continue until `return 0;`, read the output in the console window and press `F5` once more to finish.

7.2 Find a bug that the compiler does not see

1. Add a new solid to the `volume` module. Add this prototype to `volume.h`:

```
double volume_cone(double radius, double height);
```

   Add this function at the end of `volume.c`. Copy it exactly, including its mistake:

```
/* Volume of a cone: one third of the cylinder with the same base. */
double volume_cone(double radius, double height)
{
    double base = area_circle(radius);
    double factor = 1 / 3;

    return factor * base * height;
}
```

   Add these lines to `main.c`, just after the `cube` line:

```
    printf("  cone      r = %.1f, h = %.1f  : %8.3f\n",
           radius, height, volume_cone(radius, height));
```

2. Build with `Ctrl+Shift+B`. There is no warning, but run the program with `mingw32-make run` and look at the last line:

```
    cone       r = 1.5, h = 3.0  :    0.000
```

The correct volume is 7.069.

3. Set a breakpoint on the first line of `volume_cone()` and start the debugger with `F5`. Press `F10` line by line and watch the values of `base` and `factor` in the *Variables* panel.
4. In the *Watch* panel, click **+** and add the expressions `1 / 3` and `1.0 / 3.0`. Compare their values.
5. Fix `volume_cone()`, build and check that the last line of the output is now:

```
    cone       r = 1.5, h = 3.0  :    7.069
```

8 Part 5: questions

Answer these questions in a plain text file named `answers.txt`, in the project folder. Short answers are fine, but justify them.

1. In Part 1, step 4, `make` did not run any command. How does it decide that `build/app.exe` is up to date?
2. In Part 2, after changing `area.c`, `make` recompiled `area.c` and linked the program, but it did not recompile `volume.c`, even though `volume.c` calls functions of `area.c`. Why is that correct?
3. In Part 2, after changing `area.h`, the three `.c` files were recompiled. Explain why using the content of `build/volume.d`. What could go wrong if `make` did not know which headers each `.c` file includes, and you changed a prototype in `area.h`?
4. In Part 3, step 3, was the warning reported again? Why? What should you run before submitting a project to be sure that it has no warnings at all?
5. Stopped in `area_rectangle()` (Part 4), which functions appear in the *Call Stack* panel, and in which order? What are the values of `width` and `height`, and why are they equal?
6. In `volume_cone()`, what was the value of `factor`, and why? Why did gcc not warn you, even with `-Wall -Wextra`? Give two different ways of writing the line correctly.

9 Going further (optional)

Add a new module, `surface`, that computes the total surface area of two solids:

- `double surface_cuboid(double length, double width, double height)` returns the total surface of a cuboid (six rectangular faces).
- `double surface_cube(double side)` returns the total surface of a cube (six square faces).

Each function must call the functions of the `area` module. Call both from `main()` with the same values used for the volumes. The expected results are 59.000 for the cuboid and 24.000 for the cube.

- Did you have to change the `Makefile`? Look at the output of `mingw32-make` and explain in

`answers.txt` how the new file was found.

10 Checklist and submission

Your lab is complete if:

- ☐ `mingw32-make clean` followed by `mingw32-make` builds the project without any warning.
- ☐ `volume_cone()` is fixed and the program prints the expected output, cone included.
- ☐ The `Makefile` and the files in `.vscode` have not been modified.
- ☐ `answers.txt` answers the six questions.
- ☐ You have the screenshot of Part 4, stopped in `area_rectangle()`.

The final output of the program must be:

```
Areas
  circle    r = 1.5           :    7.069
  rectangle 4.0 x 2.5         :   10.000
  square    side = 2.0        :    4.000
Volumes
  cylinder  r = 1.5, h = 3.0  :   21.206
  cuboid    4.0 x 2.5 x 3.0   :   30.000
  cube      side = 2.0        :    8.000
  cone      r = 1.5, h = 3.0  :    7.069
```

> Submit the folders `src` and `include`, the file `answers.txt` and the screenshot of Part 4, as indicated by your instructor. Do not submit the `build` folder: run `mingw32-make clean` first.

11 Troubleshooting

- `'mingw32-make' is not recognized ...`**.** The package is not installed, or `C:\msys64\ucrt64\bin` is not in your `Path`. If you have just changed the `Path`, close VS Code completely and open it again: its terminal keeps the old one.
- `multiple definition of 'main'`**.** There are two `.c` files with a `main()` function in `src`, for example a copy of `main.c`. Every `.c` file in `src` is compiled and linked: keep only one.
- `undefined reference to ...`**.** This is a linker error: a `.c` file is not inside `src` (the `Makefile` only looks there), or a function is declared but never defined.
- `fatal error: ...h: No such file or directory`**.** The header is not in `include`, or its name is misspelled.
- `The preLaunchTask 'build' terminated with exit code 2`**.** The build failed when you pressed `F5`. Choose *Abort*, not *Debug Anyway*, which would debug the old executable. Fix the errors shown in the *Problems* panel.
- **The debugger does not start and mentions** `gdb.exe`**.** `gdb` is not installed, or MSYS2 is not in `C:\msys64`. Ask your instructor before changing `miDebuggerPath` in `launch.json`.

- **The console window closes before you can read the output.** Set a breakpoint on `return 0;`, or run the program with `mingw32-make run`.
- **The `.vscode` files changed by themselves.** You probably pressed ▷ *Run C/C++ File*. Copy the `.vscode` folder again from the template.