Programming 2

Ayman Makki (ayman.makki@uni.lu)
Semen Yurkov (semen.yurkov@uni.lu)

Lab 1 – Hello, C! Hello, Git!

1 *Hello, World !* in C

✎ **Virtual Machine**
In the context of this course, we use the same Ubuntu VM from last semester. It can be found on the `C:\` drive of the workstations in the lab rooms. Add this VM to the VirtualBox Manager by clicking on the corresponding `.vbox` file (the one with the blue icon). To use the VM on your own machines, please find the download link on the Moodle page of this course.

Exercise 1 – "Hello, World !" – Raw Style
The exercice aims to let you write a C program printing `"Hello World!"` to standard output, as seen in the lecture. In a first time, we will come back in the earlier times of programming, where we did not have a whole set of tools for easing and accelerating software development. Following the guidelines to come, you should be able to write, compile and run a C program with **the only usage of the command line terminal**.
If it might seem to you redundant or an inefficient way of programming, be aware that a lot of programmers still have to use this style of programming methodology (especially when you are working on a server, like in the HPC case, where you don't have access to graphical interfaces such as text editors or IDEs). Of course, they learned all keyboard shortcuts to become even more productive than a "classical programmer" using a mouse and/or an IDE.

1° Boot the VM image provided for this course.

2° Open the terminal and use the `touch` command to create a new file

Meet your new best friend : the `man` command (short for *manual*)
In Unix-based programming, there is a huge amount of resources and you will constantly run into new ones. One of the first elementary knowledge you must have is how to read a Unix command manual. Most resources in Ubuntu have a very well documented manual and explain you how to use each command. However, it is not always meant to be read by an unexperienced programmer and being able to extract useful information out of a manual will allow you to not only to be faster, but to have a better understanding of the tools you use.

— In order to display the manual for a command X, you should type : `$ man X`.

— Doing so with the `touch` command yields among the first lines :
SYNOPSIS :
touch [OPTION]... FILE...
This is the global syntax of the `touch` command : first the command, then the options to be applied, and then the name of the file on which to apply the options. To create a new file named "main.c", we will simply type :
`$ touch main.c`.

— As a first approach on Unix manuals, try to guess the original use of the `touch` command in the manual.

3° In order to edit the file we just created, we will use the `nano` command line editor. You can open the file you just created by simply typing `$ nano main.c` which will open the text editor for you. Why not also having a look at the manual ? (`man nano`)

4° Now, you should be able to write the content on the program by yourself (following the lecture notes), as well as compiling it and running it. Please be aware of the keyboards shortcuts for `nano` at the bottom of the screen, they will allow you to save your changes to the file. Make sure it correctly prints the message on the terminal when you run the program.

Exercise 2 – "Hello, World !" – Pro style

As you may have experienced, "raw" programming can become overwhelming when working on big projects with dozens of files and hundreds of code lines in each file. In order to ease the programmer effort, several IDEs have raised : some are specialized for professional workflows, version management etc.. and some other are more general and dedicated to all levels of experience. You might have been using Visual Studio Code on the first semester and you can keep using it if you feel comfortable with it. Otherwise, you might want to have a look at Atom, Eclipse etc. None is the best for everything and each has its own advantages and drawbacks.

Write again a C program printing `"Hello World!"` to standard output, but this time in your preferred IDE.

You should be able to compile and run it as previously without any difficulties. However, let's introduce two very useful tools :

— **Make** is a build automation tool that allows you to automatically build and compile your projects. Currently, while we have very simple programs, it only seems to avoid you having to type the compiling line in your command line. However, you will see that, as soon as you build more complex programs (later on in this course), it can come to be very handy.

— **GDB**, a.k.a GNU Debugger, is a debugging tool than you can use to debug C programs. When your program has been wrongly written (believe me, there will be plenty of occasions to wrongly write your program), either the compiler detects it and let you know about it alongside useful information about your error, either the compiler does not notice the error which will only raises at runtime (meaning when you execute your program). In the latter case, you might need hours or days to find the error, since the program won't give you any clue about where is your error.

For **GDB**, we won't go into much details in the lab sheet, we will prefer live examples during the labs. Here are the most basic stuff you want to know :

1° When you want to debug a program, you should compile it with an additional flag : `-g`. In our case, our compiling command when we aim to debug the program will be `gcc -g main.c -o main.out`.

2° Then, you should run the executable using the **GDB** tool : `gdb main.out`.

3° Then you have full control : you can decide to run your whole program, or tell him to stop at a point where you expect an error to be raised. You can display the content of your variables, set breakpoints etc.. To help you, you can :

— Have a look at `man gdb` which I will strongly encourage you to, even though it won't be crystal clear yet.

— Have a look at the **GDB** cheat sheet that I'll put at the end of the lab sheet in Figure 5.

— Follow a tutorial online to understand the basics of **GDB**. There are plenty of them online and you can find a simple one at https ://www.tutorialspoint.com/gnu_debugger/index.htm for beginning.

— Be careful in class when we will use it ;)

For **Make**, you should understand that its most useful aspect is to automatically build, run and/or debug your program. Through this course and the in your programmer experience, you will run into **a lot** of compilation lines .. and when you have bigger projects, you might need dozens or hundreds of compilation lines to build your program (which is the case for every software, game or programming tool you use). To avoid always copying those lines manually, we use a **Makefile**. Some IDEs also allow you to automatically build your Makefiles, but you should understand the basic of those. The makefile has to be written in a file named *makefile* at the root of the folder in which you have your program. Here is a simple makefile that fits our current needs :

```
# Declaring the variables to be used
FILENAME = main.c
OUT = main

# A rule for compiling our file
compile : $(FILENAME)
  gcc $(FILENAME) -o $(OUT)

# A rule for running our program once compiled
run : $(OUT)
  ./$(OUT)

# A rule for debugging
debug : $(FILENAME)
  gcc -g $(FILENAME) -o $(OUT)
  gdb $(OUT)

# A rule to clear the executable
clear :
  rm -f $(OUT)
```

In blue, you have the rules (commands) that you define. In red, you have the variables that you set at the beginning and that you use later with the decorator $(...). And below each rule in blue you have the detail of the command when you launch it. Eventually, on the right after a command you have the prerequisite for the command to be launched. Let's decompose the first one :

- Command name is *compile*
- Prerequisites on the right are in the variable FILENAME which is the name of our program *main.c*. This means that the command requires this file to be run (in fact, it is hard to compile a file which doesn't exist).
- Then the actual command that will be run will be used with correct variable names, which leads to : **`gcc main.c -o main`**

To launch a command, we simply type **`make command_name`**. Therefore, with this makefile, we just have to type :

- **`make compile`** to compile our program.
- **`make run`** to run the program.
- **`make debug`** to compile the program with the *-g* flag and run the debugger.
- **`make clear`** to clear the executable.

2 Introduction to GitHub Classroom

2.1 Motivation

During this course, the code examples discussed in the lecture as well as the solutions of the practical labs will be put on GitHub Classroom which has been conceived to allow students and teachers to work together on specific assignments. In the meanwhile, the platform will allow us to extend and continue the discussion on code snippets outside the regular lecture and lab slots. The idea is to provide you with a tool where you can comment code snippets, ask questions on what you have not understood so far, discuss alternative approaches and help each other on the journey of learning a programming language. Please feel free to contribute with your questions and discussions, which we will try to answer in a timely and clear way. Also, do not hesitate to engage in a discussion and answer questions of your fellow students.

2.2 One lab session => one assignment on GitHub Classroom

Before each lab session, we will send you an invitation link to join an assignment repository that we made for the lab. It will contain all the code snippets required to proceed through the lab, and potentially a small README.md file that will detail you the main guidelines of the exercises in the lab session.

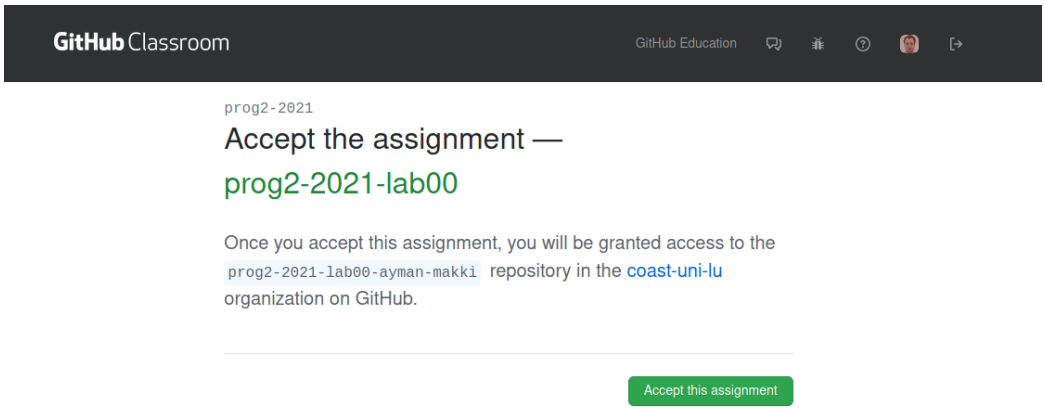

FIGURE 1 – Accepting an assignment invitation creates your own repository

Once you click on the invitation link, after creating you GitHub account or logging into it, you will be asked to accept the assignment, such as in Figure 1 : accepting it will create your own copy of the laboratory repository, in which you'll have access to a copy of all the required files. You can modify any of those files within your repository using the Git-based methods (pull,commit,push,merge etc.). Your repository is private, meaning only you and the teachers have access to it (except if you decide otherwise), and no one else than you and the teacher will be able to see what you upload and/or the questions you ask.

In Figure 2, we represented you the main screen of your own repository, to which you should have access after accepting the assignment. Let us details the most important tools you now have access to :

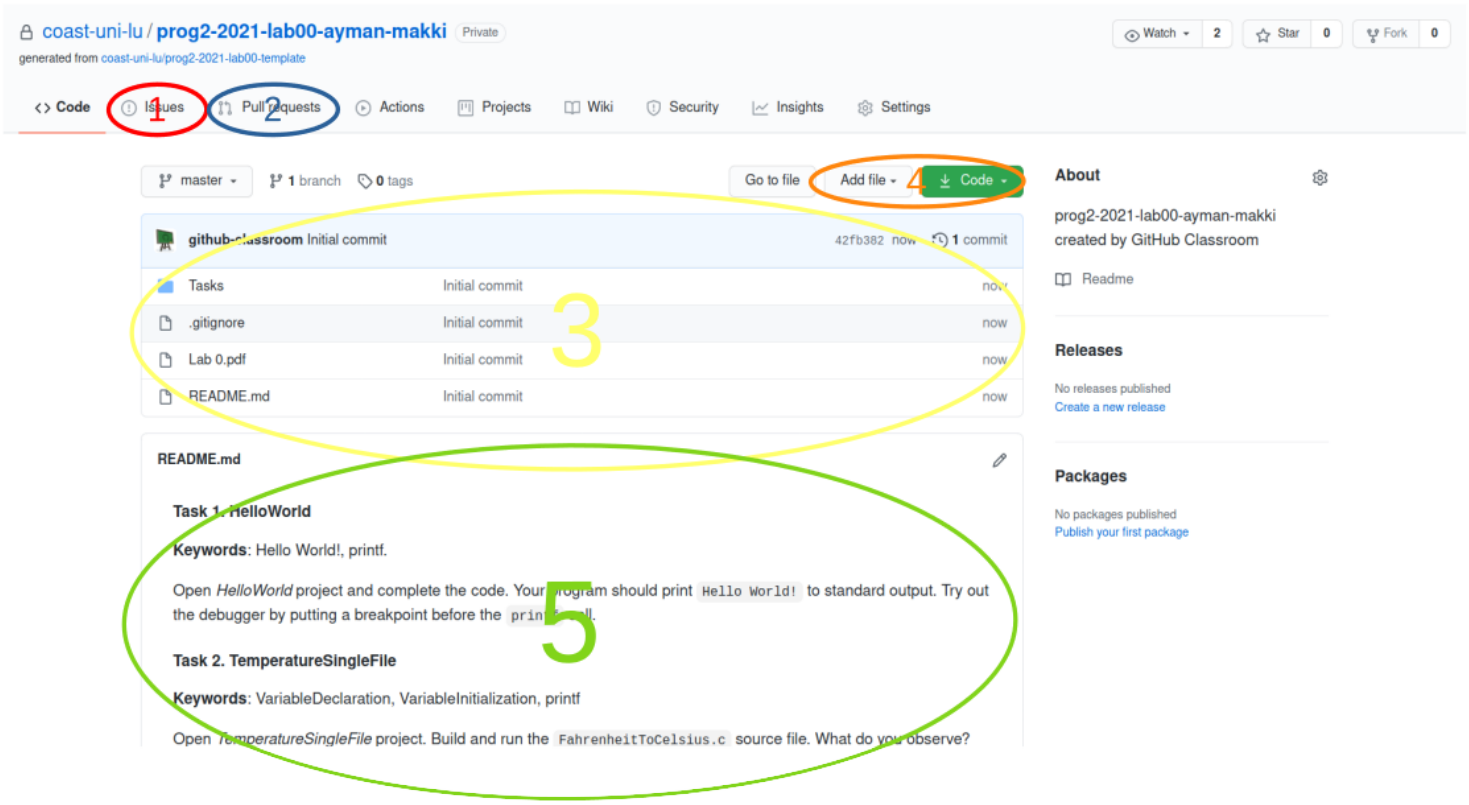

FIGURE 2 – The main page of your assignment repository

— 1 - Issues Submission : this allows you to ask a question directly to the teacher, by adding a new issue in your repository. If you agree too, interesting issues raised will be discussed during lab sessions to benefit to everyone.

— 2 - Pull Requests : This will mostly be used during your projects : it allows you to correctly merge your work with the work of your project mates while making sure you won't erase their changes to the repositories. We'll come back on this later.

— 3 - Repository files : this is as summary of the files in the repository.

— 4 - Manual submission and upload : Those buttons allow you to manually add / clone files from the repository to / from your local repository on your machine.

— 5 - Basic instructions : this contains all the basic instructions for the laboratory assignment which are written in the README file.

2.3 GitHub usage

2.3.1 Clone, add, commit and push !

For a brief introduction that let you use Git tools out of the box, we need to understand three main commands : clone, add, commit and push. Those allow you to carefully synchronize your working repository (on your computer) with your local and remote repositories. A summary of the operations is provided in Figure 3.

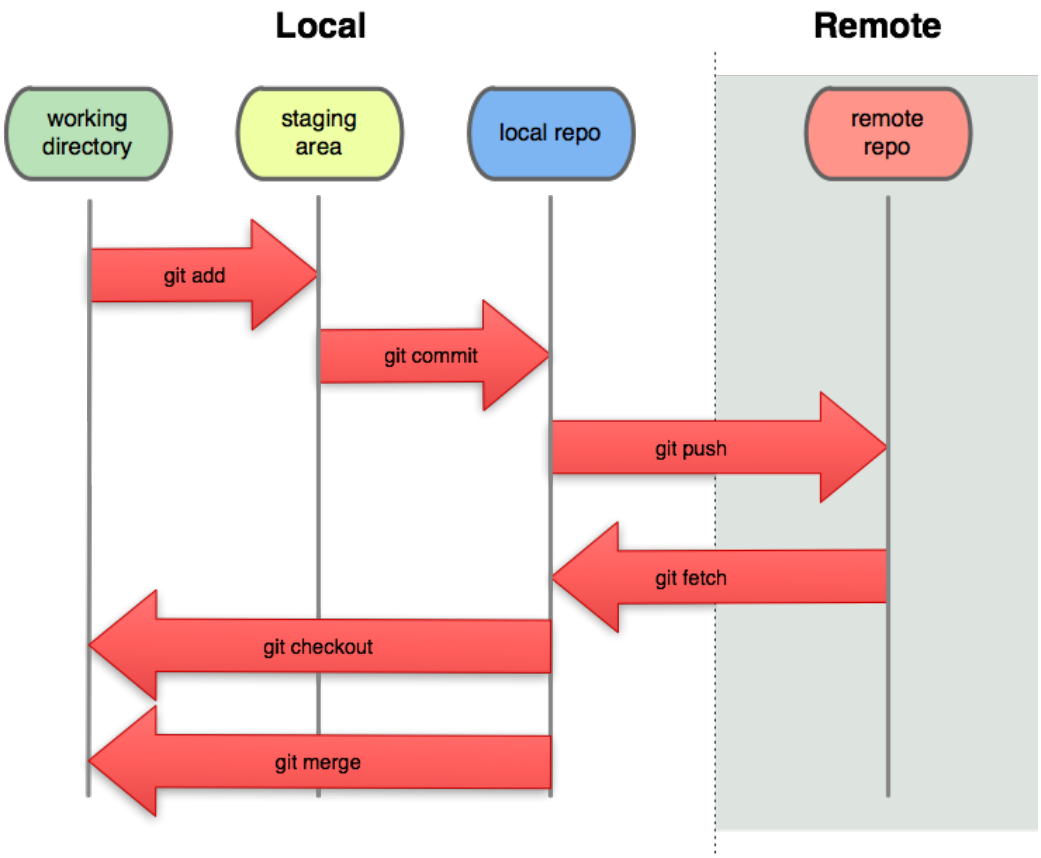

FIGURE 3 – GitHub workflow

In a nutshell, you should consider three main concepts : the working directory, the local repository and the remote repository.

— The remote repository (on GitHub) will always be synchronized with your local repository, never directly with the working directory.

— The working directory is the one where you usually work, save your files and compile your programs.

— All you have to do when you want to update your changes to GitHub, is making sure that your local repository contains everything you wish to update from your working directory to the remote one on GitHub. This is done with the `add` and `commit` commands.

— Once your local repository is up to date, you can synchronize it with your remote repository on GitHub using the `push` command.

You will find below a detailed explanations for each command :

1° **Clone :** (make sure to change the adress with the one from your repository, cf Figure 2, 4 - Manual submission and upload)

```
git clone https://github.com/coast-uni-lu/prog2-2021-lab00-ayman-makki.git
```

This command allows you to copy all the files in your remote repository on your computer(provided you change the adress here to the one provided in your own repository). This will create a new folder on your computer with the name of the repository. If you do not want to use the repository name for the folder in your laptop, you can simply add the name of the folder you would like to use at the end of the command (git clone https ://.....git myfolder). **This is to be used only once when starting to work on a Git project.**

2° **Add :**

```
git add [FILE or . for everything]
```

By default, Git only synchronizes the files that were in the original remote repository. Meaning that, if you changed one of those files, you won't need to "add" it to the local repository since it is already there : it will get updated with a simple **`commit and push`**. However, if you want to add a new file to the repository, you must do so **before committing**. Either you add a specific file withe a **`git add filename`** command, or use the "." to specify you want to add the whole current directory. You can also make a smart use of the **.gitignore** file to prevent Git to synchronize files when using the "." path that synchronizes your whole working directory.

3° **Commit :**

```
git commit [-m "Message about new changes"]
```

Ok, you did some changes in your code and it's now time to upload them on GitHub. You've already made sure to include new files with the **`add`** command and are now ready for uploading. Simply type the commit command with an additional message that stipulates the changes you brought to the code (useful for remembering what you did). If you do not provide a message to the command line when committing, it will take you into a terminal text editor where it will expect you to put a message to notify your changes. However, even though it is pretty useful to keep traces of what you did, it's not mandatory and you can simply leave a blank if you do not which to leave a message.

4° **Push :**

```
git push -u origin master
```

It's over ! Pushing allows you to synchronize your local repository with the remote repository hosted on GitHub. The first name "origin" means your local repository and master is the "branch" in the remote repository you want to push your changes to. No need to go in further details now, you have the essential to make a good use of Git.

2.3.2 Commenting Code

As already mentioned, the main purpose is to enable you to ask questions on aspects you have not understood so far and discuss alternative approaches with the lecturer, the teaching assistants and even your fellow students. However, no one is inside your mind and might be able to guess what you meant by different parts of your code. Commenting your code is a standard in all open source projects and should be a good habit that you take starting from this year. It can be painful for a classmate, or a teacher, to spend

twenty minutes trying to **understand** your code before trying to debug it. Even for yourself, when you come back to code you wrote years ago, you won't have a so clear vision of it as you did when you wrote it. So, please, spare yourself, your teacher, your classmates and your future manager : **COMMENT YOUR CODE :)**.

"Code never lies, comments sometimes do." - Anonymous

2.3.3 Notifications

When a comment is written on a code snippet, the teaching team will be notified to be able to answer your questions in a timely manner. The questions can be asked either in your private repository, or in the teacher's repository where everyone will be able to see your question. Of course, you are also kindly invited to participate in the discussion initiated by a fellow student. To be notified on a new comment from any project member, you have to change the notification level in the repository of the teacher (or in your own) as seen in Figure 4. **Remember : There is no such thing as a stupid question !**

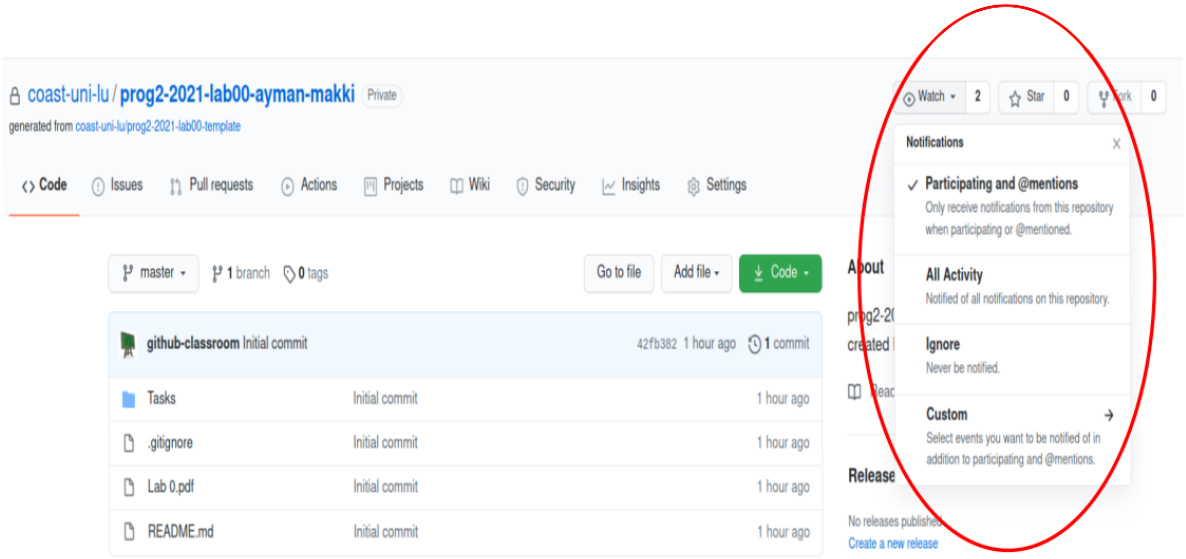

FIGURE 4 – Notification Levels within your repository

Exercise 3 – Temperature Conversion #c :VariableDeclaration #c :VariableInitialization #c :printf

In this exercise, you will take first steps with Git and GitLab.

1° Begin by *cloning* the Git repository of the lecture as we saw in the last section :

> ✎ **Easy, huh ?**
>
> You will see later than you can deal with Git repositories through an IDE. For now, let's focus on the command line usage as we detailed expansively in the previous section which is easier and faster :
>
> ```
> git clone [your repository gitfile adress]
> ```
>
> Now you have access to the raw code for all three exercises.

2° Build and run the `FahrenheitToCelsius.c` source file. What do you observe ? What might be the issue and what solution do you propose ? You can either try to debug it yourself or try to use the **GDB** tool introduced earlier. Feel free to adapt the Makefile from previous exercises to make your development or debugging easier.

3° One you have fixed the code and made sure it was working properly, please push your changes to your GitHub repository for Lab 01 as explained in the previous section on Git usage. As a message for the `commit`, you can put "Fixed Temperature Conversion".

Exercise 4 – Temperature Conversion – Reloaded #c :Typedef #c :FunctionDefinition #c :PreprocessorInclude

Using the tags attached to this exercise, consult material on these topics to create a new C console application that converts temperatures between Celsius and Fahrenheit units. Particularly :

— *define a new type* `Temperature` as a synonym for the already existing type `float` ;

— write two *functions*

  — `Temperature toFahrenheit(Temperature celsius)`

  — `Temperature toCelsius(Temperature fahrenheit)`

  that calculate the respective conversion ;

— put the *type definition* and function *prototypes* in a separate *header file* `temperature.h`, the function implementations in a separate source file `temperature.c` and the `main` function, which demonstrates the correct behavior calling the previous two functions, in a separate source file `main.c`

— Adapt your makefile to make it work with multiple files compilation.

gdb cheat-sheet for reverse-engineering

Nota bene: character ‘ is a backquote (AltGr+7) !

Starting GDB

`gdb`	start GDB, with no debugging files
`gdb` *program*	begin debugging *program*
`gdb --args` *prg args*	begin debugging *prg args*

Stopping GDB

`quit`	exit GDB; also `q` or `EOF` (eg `C-d`)
`INTERRUPT`	(eg `C-c`) terminate current command, or send to running process

Getting Help

`help`	list classes of commands
`help` *class*	one-line descriptions for commands in *class*
`help` *command*	describe *command*

Executing your Program

`r[un]` *arglist*	start your program with *arglist*
`r[un] $(`*cmd*`)`	start your program with the output of command *cmd* as an argument
`r[un] ‘`*cmd*`‘`	
`r[un] ... <`*inf* `>`*outf*	start your program with I/O redirected
`r[un] ... <<<` *str*	start your program with *str* as standard input content
`r[un] ... < <(`*cmd*`)`	start your program with the output of command *cmd* as standard input content
`r[un] ... <<< $(`*cmd*`)`	
`r[un] ... <<< ‘`*cmd*`‘`	
`kill`	kill running program
`set args` *arglist*	specify *arglist* for next `run`
`set args`	specify empty argument list
`show args`	display argument list
`set disable-randomization [on|off]`	disable ASLR

Breakpoints and Watchpoints

`break` [*file*:]*line*	set breakpoint at *line* number [in *file*]
`b` [*file*:]*line*	eg: `break main.c:37`
`break` [*file*:]*func*	set breakpoint at *func* [in *file*]
`break [+|-]`*offset*	set break at *offset* lines from current stop
`break *`*addr*	set breakpoint at address *addr*
`break`	set breakpoint at next instruction
`catch` *event*	break at *event*, which may be `catch`, `throw`, `exec`, `fork`, `vfork`, `load`, or `unload`.
`info break`	show defined breakpoints
`delete` [*n*]	delete breakpoints

[ ] surround optional arguments ... show one or more arguments

Program Stack

`backtrace` [*n*]	print trace of all frames in stack; or of *n* frames—innermost if *n*>0, outermost if *n*<0
`bt` [*n*]	
`frame` [*n*]	select frame number *n* or frame at address *n*; if no *n*, display current frame
`up` *n*	select frame *n* frames up
`down` *n*	select frame *n* frames down
`info frame` [*addr*]	describe selected frame, or frame at *addr*
`info args`	arguments of selected frame
`info locals`	local variables of selected frame
`info reg` [*rn*]...	register values [for regs *rn*] in selected frame; `all-reg` includes floating point
`info all-reg` [*rn*]	

Execution Control

`continue` [*count*]	continue running; if *count* specified, ignore this breakpoint next *count* times
`c` [*count*]	
`step` [*count*]	execute until another line reached; repeat *count* times if specified
`s` [*count*]	
`s[tep]i` [*count*]	step by machine instructions
`next` [*count*]	execute next line, including any function calls
`n` [*count*]	
`n[ext]i` [*count*]	next machine instruction
`until` [*location*]	run until next instruction (or *location*) or the current stack frame returns
`finish`	run until selected stack frame returns
`return` [*expr*]	pop selected stack frame without executing [setting return value]
`jump` *line*	resume execution at specified *line* number or *address*
`jump *`*address*	
`set var=`*expr*	evaluate *expr* without displaying it;

Working Files

`file` [*file*]	use *file* for both symbols and executable; with no arg, discard both
`core` [*file*]	read *file* as coredump; or discard
`exec` [*file*]	use *file* as executable only; or discard
`symbol` [*file*]	use symbol table from *file*; or discard
`load` *file*	dynamically link *file* and add its symbols
`add-sym` *file addr*	read additional symbols from *file*, dynamically loaded at *addr*
`info files`	display working files and targets in use
`path` *dirs*	add *dirs* to front of path searched for executable and symbol files
`show path`	display executable and symbol file path
`info share`	list names of shared libraries currently loaded

Display

`print` [*/f*] [*expr*]	show value of *expr* [or last value `$`] according to format *f*:
`p` [*/f*] [*expr*]	
`x`	hexadecimal
`d`	signed decimal
`u`	unsigned decimal
`o`	octal
`t`	binary
`a`	address, absolute and relative
`c`	character
`f`	floating point
`call` [*/f*] *expr*	like `print` but does not display `void`
`x` [*/Nuf*] *expr*	examine memory at address *expr*; optional format spec follows slash
N	count of how many units to display
u	unit size; one of `b` individual bytes, `h` halfwords (two bytes), `w` words (four bytes), `g` giant words (eight bytes)
f	printing format. Any `print` format, or `s` null-terminated string, `i` machine instructions
`disassem` [*addr*]	display memory as machine instructions

Debugging Targets

`target` *type param*	connect to machine, process, or file; e.g. `target remote | sshpass -p`*pw* `ssh -T` [`-p` *port*] [user`@`]*host* `gdbserver -` *prog* [*args*]
`attach` *param*	connect to another process
`detach`	release target from GDB control

Source Files

`dir` *names*	add directory *names* to front of source path
`dir`	clear source path
`show dir`	show current source path
`list`	show next ten lines of source
`list -`	show previous ten lines
`list` *lines*	display source surrounding *lines*, specified as:
[*file*:]*num*	line number [in named file]
[*file*:]*function*	beginning of function [in named file]
`+`*off*	*off* lines after last printed
`-`*off*	*off* lines previous to last printed
`*`*address*	line containing *address*
`list` *f*`,`*l*	from line *f* to line *l*

FIGURE 5 – Original file can be found at : http ://morgan.perso.enseeiht.fr/assets/supports/gdb-cheat-sheet.pdf