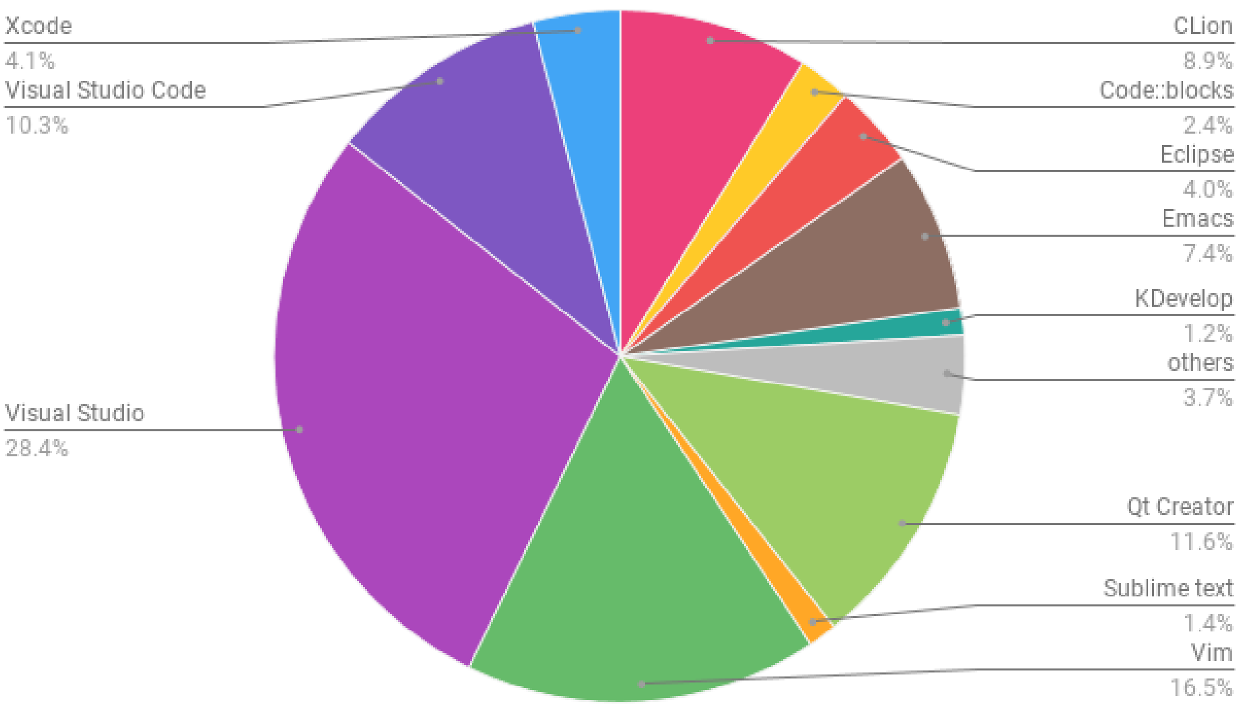
Operating systems: homework 2

An introduction to Development Tools



Bachelor Electronics/ICT

Course coordinator: Bert Lagaisse

Lab coaches:

- Ludo Bruynseels
- Jonas Verbruggen
- Toon Dehaene

```
 ----------------------------------------------------------------
| 1.   Goals of this session                                       |
 ----------------------------------------------------------------
```

Key goal of this homework:

- ⇨ Set up all tools that you will need to do the exercises in this course.
- ⇨ Short tutorial/example on every tool.

Line of departure:

- ⇨ You have performed all the steps from Homework 1.

During this assignment, you will learn the development tools you need to author C programs on your Linux VM.

1. Using *gcc* to compile and link a C program
2. Using *Make* to build C programs
3. Using *GDB* to debug C programs.
4. Using *CLion* to open multi-file C programs with a Makefile.
5. Using *CLion* to compile, run and debug C programs.

C language reference manual: https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html

Manual of the Make utility: https://www.gnu.org/software/make/manual/

```
 ----------------------------------------------------------------
| 2.   First program in "C"                                        |
 ----------------------------------------------------------------
```

In clear words:

1. Function declaration =

 name of a function + return type + list of parameters.

Ends with semicolon

```
int foo (int, double);
```

2. Function definition: name, return type, list of parameters. NO semicolon BUT a function BODY

```
int add_values (int x, int y)
{
  return x + y;
}
```

Consequences:

- ⇨ Typically, function declarations are placed in a header file (.h) but not always.
- ⇨ Not mandatory but highly recommended: write a declaration for every function.
- ⇨ Function definitions are placed in a source file (.c in our case).
- ⇨ A function must be declared before it can be used. Therefor get in control **and** write the declaration yourself!

First, make a working directory where you will store all your work for these sessions:

```
$ mkdir osc
$ cd osc/
$ mkdir homework2
$ cd homework2
```

In directory homework2 you can create all the files of the next exercise

```
$ cd homework2
$ touch hellofunc.h
$ touch hellofunc.c
$ touch hellomain.c
$ touch Makefile
```

We will start this tutorial with building a C program that consists of multiple code files: hellofunc.h, hellofunc.c and hellomain.c . The file hellofunc.h contains the function declaration. This is useful for users of your code: they need the declaration.

```
/*
header file
*/
void myPrintHelloMake (char * who);
```

Figure 1: hellofunc.h

The file hellofunc.c contains the function definition Users of your function do not need to know the implementation.

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

void myPrintHelloMake(char * who) {

  printf("Hello %s!\n", who);

  return;
}
```

Figure 2: hellofunc.c

The file hellomain.c uses the function myPrintHelloMake:

```
#include "hellofunc.h"

int main (int argc, char **argv) {
  // call a function in another file
  if (argc == 2) myPrintHelloMake (argv [ 1 ]);
  else myPrintHelloMake ("nobody");
  return 0;
}
```

Figure 3 hellomain.c

You can compile and run your C program using the following gcc command:

```
$ gcc -o hello hellomain.c hellofunc.c
$ ./hello me
```

Your turn to try it all out:

- ⇨ Get the version of gcc on your system by typing `gcc –version` in a terminal.
- ⇨ If gcc is not installed type `sudo apt install gcc`
- ⇨ Type `man gcc` to get an idea of the possible options supported by the gcc command.
- ⇨ make a folder somewhere in your folder tree, create these 3 files with an editor of choice and build and run the program.

```
 -----------------------------------------------------------------
| 3.  Building with make: defining the Makefile.                  |
 -----------------------------------------------------------------
```

```
Target1:  prerequisites ….
          Recipe
          …
          …
Target2:  prerequisites ….
          Recipe
          …
          …
…
```

Figure 4 a makefile consists of rules (targets)

First, we will create a make file that has multiple build targets.

1. One target "`all`" where we build the program hello at once, just like in the manual command above.

2. One target “`stepwise`” where we build each file to a separate object file, which we then link together to the program hello.
3. A target to `clean` up the output above.

You can find the different build targets in the Makefile underneath.

```
all: hellomain.c hellofunc.c hellofunc.h
      gcc -o hello hellomain.c hellofunc.c

stepwise: hellomain.o hellofunc.o hellofunc.h
      gcc -o hello hellomain.o hellofunc.o
hellofunc.h

hellomain.o: hellomain.c
      gcc -c hellomain.c

hellofunc.o: hellofunc.c hellofunc.h
      gcc -c hellofunc.c hellofunc.h

clean:
      rm -rf *.o
```

Figure 5 content of Makefile

It is your turn again

- **`$ make –version`**
- If make is not installed: **`$ sudo apt install make`**
- With your editor, create the Makefile as above.
- Do experimentation with following commands. What happens in your working folder, which files are created, deleted, …? All commands must work. If not, be persistent and make them work!

```
$ make all
$ make clean
$ make stepwise
$ ./hello
$ ./hello me
```

```
------------------------------------------------------------------
| 4.   Configuring Git (local repository)                         |
------------------------------------------------------------------
```

Test if git is installed on your system: **`$ git –version`**

To install: **`$ sudo apt install git`**

Basic configuration:

```
$ git config –global user.email "you.name@student.kuleuven.be"
$ git config –global user.name "your name"
$ git config –global init.defaultBranch "main"
$ git config --global –list
```

Git wants to know who you are because anonymous commits are not allowed. Every commit must leave its fingerprints. There you have to set user.name and user.email.

The 3th setting makes sure that in any new repository's default branch is named 'main'. Later on, we will link your local repository to a remote repository on github and they name the default branch also 'main'. It avoids a few problems if we do it the first time right.

The last one lists all the settings you have made. (they are in a hidden file ~/.gitconfig)

Now cd into the directory you created a few paragraphs ago. If you followed our suggestions, that is ~/osc/ and initialize a new repository.

```
$ cd ~/osc/
$ git init
```

You can see that your directory now contains a valid git repository: **`$ ls -a`** will show that there is a hidden .git directory.

Next we do want to check in our files. To do that, issue following commands.

```
$ git status  /* lists all the files that are not yet under source
control or modified. */

$ git add . /* copies all added and modified to the index. This means
that are staged for commit but not committed yet. */

$ git commit -m "my commit message".  /*Finally commits from the index
to the repository. A message is mandatory. If you omit the -m switch,
git will ask you for a message.*/

$ git status  /* your working folder is now clean */
```

Useful git commands:

git clone // clone (copy) a repository from a remote to a local repository.

git push // push your commits to a remote repository

git pull // get changes from a remote repository.

git status // check what happened in your working directory: new files, deleted files, modified files.

git add // add modified files to the index ('.' Means all changed files)

git commit // commits staged changes to the repository. Message is mandatory.

git remote // confige remotes

git init // initializes a new repository in the current working directory.

For complete documentation on GIT: https://git-scm.com/doc

Knowing these commands can be very useful when your repository is not behaving as you expect. By learning these commands, you learn the basics of git on an atomic level. Typically git interfaces in IDE's like Visual Studio, IntelliJ, Android Studio and Clion help you very well but expose too many features at once. This can confuse you and even lead to errors. So, first learn to walk and run later.

```
 -------------------------------------------------------------
| 5.   Create a repository in github                          |
 -------------------------------------------------------------
```

If you haven't done so, create an account on www.github.com

Log in and create a new repository as per the following settings:

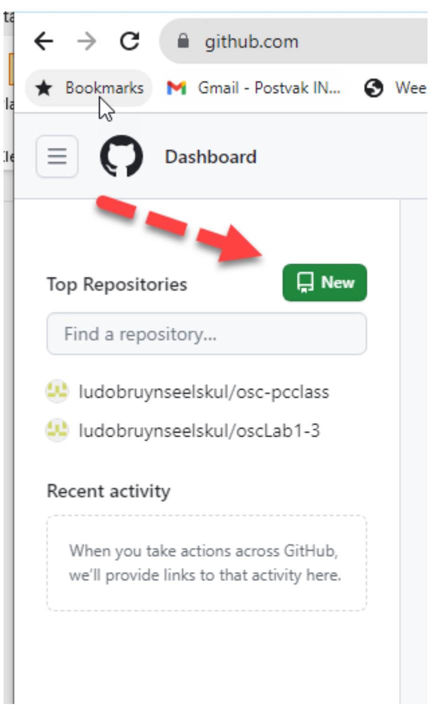


Figure 6: create a new repository

Make your repository private and include a .gitignore file based on the C template. The '.' in '.ignore' indicates that it is a hidden file. To see it in the output of the ls command, you need to need to add the -a switch: `$ ls -a`

.gitignore controls which elements are excluded from version control. You only need to put in version control what you need to rebuild the project.

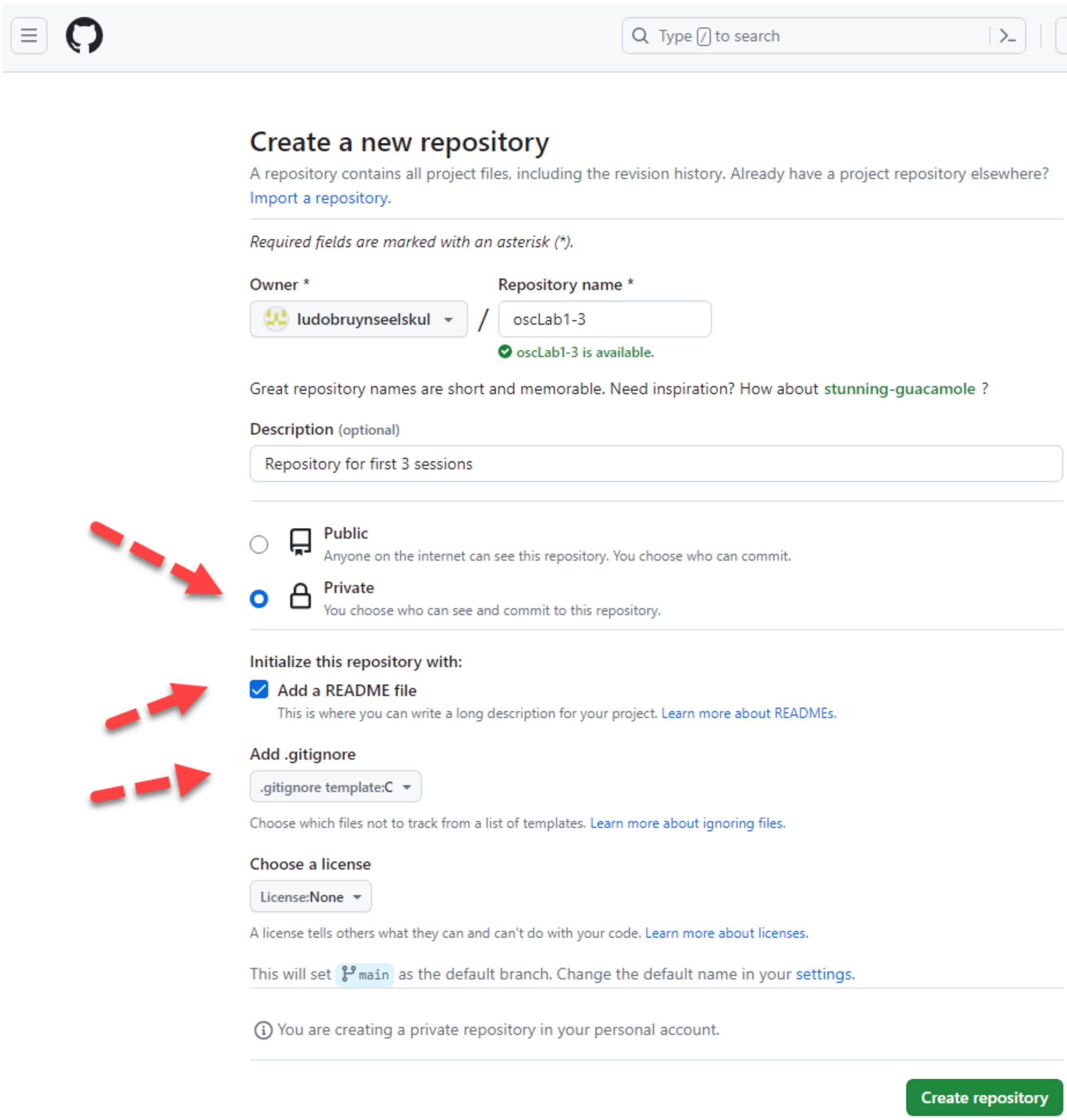


Figure 7: settings for new repository

Now you need to set up the public/private key pair on your machine and in github.

Clients can communicate in 2 ways with github servers: SSH and HTTPS. Both are application layer (layer 5) protocols but authenticate in a different way.

HTTPS could use username/password to authenticate but github does not accept that. Instead they use personal access tokens. We will discuss that when we discuss the use of Clion.

In this installment we will only use SSH.

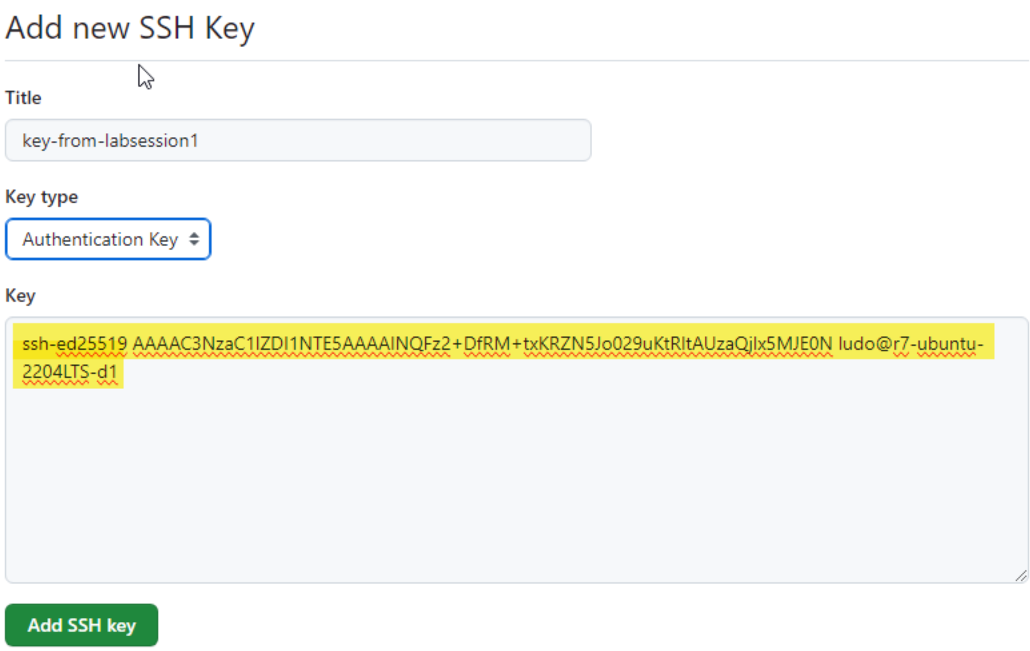
SSH operates on a client-server model. In our use case github is the server, our workstation is the client. To authenticate, a pair of keys is used: a private key on the client and a public key on the server. Clients can authenticate without any exchange of usernames or passwords.

In homework 1, you already generated a pair of keys. Now we have to put the public key on the server. (You will tell the private key to anyone; after all, is it a private key). To get to the public key type following commands:

```
$ cd ~/.ssh  (do not forget the dot '.')
$ ls
$ cat id_ed25519.pub
```

```
ludo@r7-ubuntu-2204LTS-d1:~$ cd ~/.ssh
ludo@r7-ubuntu-2204LTS-d1:~/.ssh$ ls
id_ed25519  id_ed25519.pub  known_hosts  known_hosts.old
ludo@r7-ubuntu-2204LTS-d1:~/.ssh$
ludo@r7-ubuntu-2204LTS-d1:~/.ssh$ cat id_ed25519.pub
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAINQFz2+DfRM+txKRZN5Jo029uKtRltAUzaQjlx5MJE0N
 ludo@r7-ubuntu-2204LTS-d1
ludo@r7-ubuntu-2204LTS-d1:~/.ssh$
```

In the above image, the public key value is highlighted. Copy that to your clipboard. Next, open your account settings in Github: click on your avatar in the upper right corner and select "settings". In the settings menu (left pane), click on SSH and GPG keys. Next, guess what! click on "New SSH key" and paste the key text into the appropriate field:



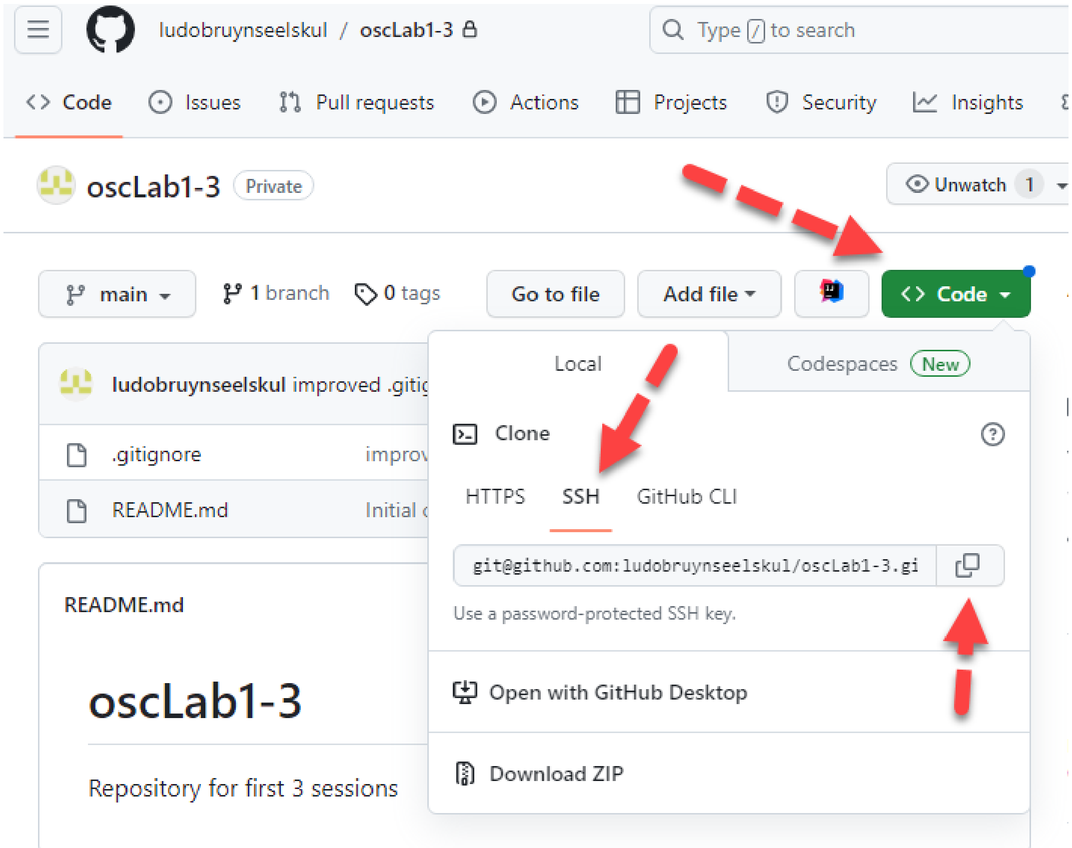

```
 --------------------------------------------------------------
| 6.   Set up a remote in the local repository.                |
 --------------------------------------------------------------
```

We could do version control in our local repository: all versions of all files are kept. All features of git are available. But we would be missing the possibility to cooperate with other developers (not needed in this course). But one word on cooperation: the assignment and grading is individual. But it is not forbidden to speak to each other and exchange ideas to deepen your understanding of the matter. To support that: do not exchange zip files by e-mail. You can invite others to your remote.

As a side effect, pushing to a repo in the cloud is also a very effective and simple means of a backup.

```
$ git remote add origin git@github.com:yourname/yourrepositoryname
$ git branch -M main
$ git push -u origin main
```

The url can be pasted from github:



7. Debugging C programs with GDB

First of all, type `gdb -version`, to check if gdb is installed on your system. If not, do `sudo apt install gdb` to install it. Create a new directory 'debugging' in the home directory and create a new C source file containing the following code. You can also make a makefile. It is not needed here but practice leads to success.

```
#include   <stdio.h>
int   factorial ( int   number );
int   main ( void ) {
     printf ( "Calculating 5!... \n " );
     int  result =  factorial ( 5 );
     printf ( "5! =  %d \n " , result);
     return   0 ;
}
int   factorial ( int   number ) {
     int  current = number;
     int  result =  1 ;
     while (current >  0 ) {
        result *= number;
        number--;
    }
      return result;
}
```

Figure 8 factorial.c

Compile this file for debugging using the -g flag. Inserting the -g flag tells the compiler to produce debugging information. This information can be used to debug your program. If ever you are not able to debug your code, the most likely cause is that you omitted the -g switch after the gcc command.

```
~/debugging$ gcc –g program.c –o program
```

Next, start the gdb debugger:

```
~/debugging$ gdb program
```

To view the source code window of the program, enter the command:

```
(gdb) layout src
```

Start the program using the `run` command. We can see the output "Calculating 5!...", but then the program seems to be stuck.

```
main.c
        4  int main(void) {
        5      printf("Calculating 5!...\n");
        6      int result = factorial(5);
        7      printf("5! = %d\n", result);
        8      return 0;

exec No process In:                                                    L??   PC: ??
(gdb) run
Starting program: /home/brent/debugging/program
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Calculating 5!...
```

Interrupting the program using `Ctrl+C` shows that the program is currently inside of the while loop in the factorial function. Step through the program line by line using the `step` command (or use the shorter version: s). This shows that the program keeps looping in the while loop. We can see that the while loop will stop if the variable current is lower than or equal to zero. Use the `print current` command to print out the value of the variable `current`.

```
main.c
       11      int current = number;
       12      int result = 1;
  >    13      while(current > 0) {
       14          result *= number;
       15          number--;
       16      }

multi-thre Thread 0x7ffff7d8a7 In: factorial          L13   PC: 0x5555555551d9
$6 = 5
(gdb) s
(gdb) s
(gdb) s
(gdb) print current
$7 = 5
(gdb)
```

Stepping through the program and printing the value of current reveals that the variable `current` is never updating. Instead, it is the variable `number` which is updating.

Quit the debugger using the command quit (or q) and fix this bug using the nano editor. Compile the program again and check its output.

More info on GDB

GDB is a very important tool. Take some time to learn the basics of it. The above paragraph are not enough to learn you how to use gdb. But you are lucky to live in the 21st century: the internet comes to help us. If you let Google do a search on 'GDB Quick Reference Sheet' for instance, you get plenty of information on gdb's commands and how to use them.

In itself gdb is not a difficult tool but debugging is a skill that requires a lot of practice. Some seem to think that all you to know to debug is the printf statement. They are wrong. Printf can help you but you must be able to set breakpoints in your programs, inspect the value of variables, … If you think the only thing you need is printf you are wrong.

Useful GDB commands:

- run : start program
- run *arglist* start program with arguments.
- kill : kill running program
- set a breakpoint: b + linenr or function name. (b or break)
- bt : backtrace. When you hit a seg fault. Bt unwinds the stack so you can see at which point in the execution the seg fault occurred.
- p (or print) + expression: get the value of the expression.
- n : step 1 line, step over function calls
- s : next line, step into function calls.
- gdb *program* : start gdb and load *program*
- *quit : leave gdb*

Experiment with setting breakpoints, inspect variables, and so on. Eventually, introduce a seg fault and see what happens. There are many more commands that can help you. See

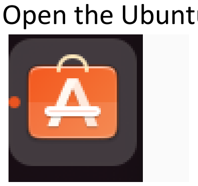
https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/documents/gdbref.pdf

If you like it concise, try **`$ man gdb`**

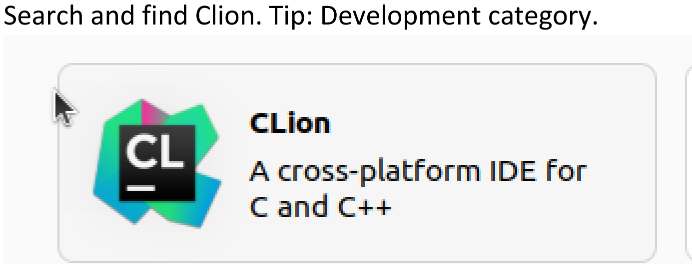
```
 -----------------------------------------------------------------
| 8.  Using the CLion IDE to develop, build and run your project.  |
 -----------------------------------------------------------------
```

To install Clion in your Ubuntu Desktop

- Open the Ubuntu software catalog



- Search and find Clion. Tip: Development category.



- Install – next – ok and all those things.
- Activate it with your academic JetBrains license like for other JetBrains tools (IntelliJ, Android Studio, …)

9. Opening an existing Makefile project with CLion

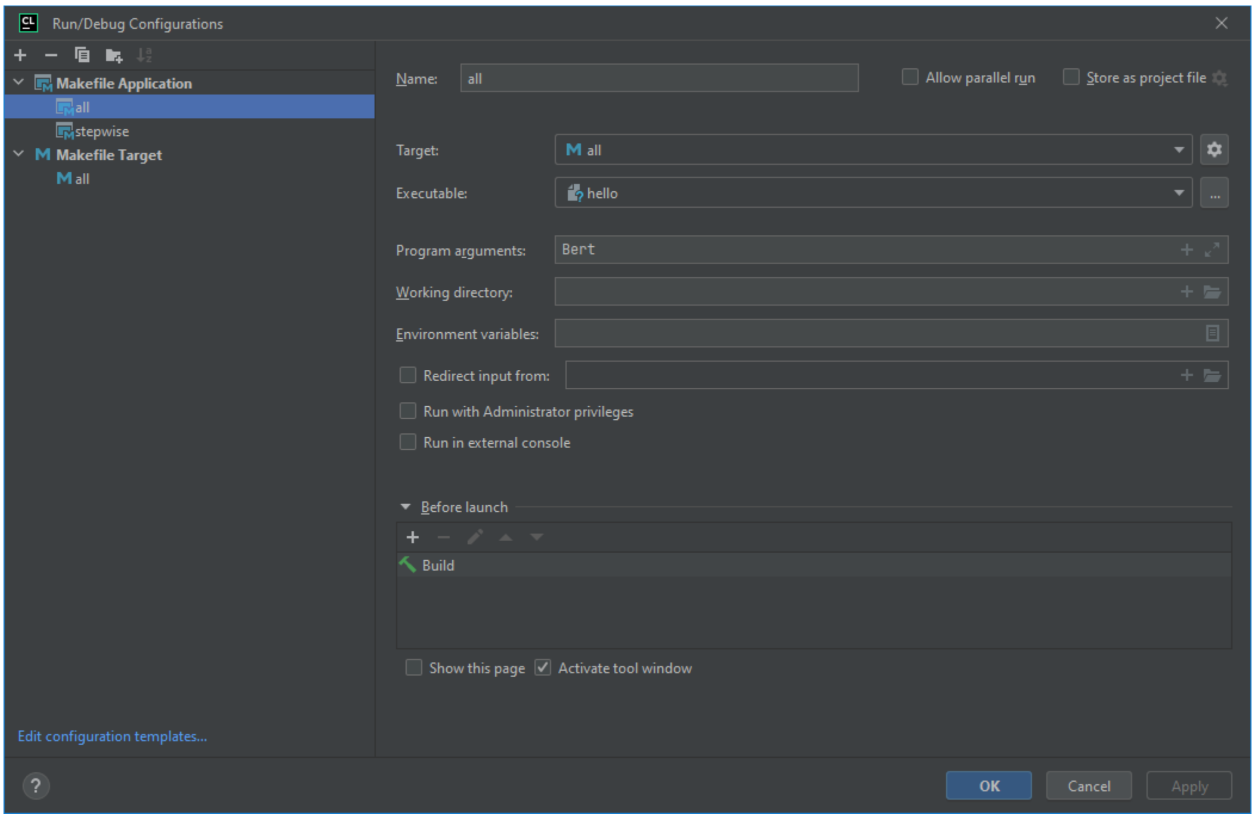
In this tutorial everything you need to know about CLion and makefiles is explained very well. Reading this tutorial is a good investment of your time:

https://www.jetbrains.com/help/clion/makefiles-support.html

CLion expects by default that targets "`all`" and "`clean`" exit in the Makefile. So you have to rename the "`atonce`" target to "`all`". before building and running this project.

To run and debug the resulting executable, open the configurations window (Run -> edit configuration). As depicted below, set the executable in the Make application "all" to the "hello" binary, and add a runtime argument with a name of your choice (e.g. Bert). You should be able to run the application. To debug the application, you will need to edit and reload the Makefile with the -g debug option for gcc.

Some of the Makefile targets (e.g., clean) might not execute from the CLion GUI.



Congratulations, you have now reached the level of Padawan in C development tools. You should now be familiar with different ways of building, running and debugging C programs on a Linux VM.

- On the commandline.
- Using CLion.

Use your new powers wisely depending on the task at hand.