Introduction to Programming

— Practical #1 —

1 Programming environment

1.1 Hello, World !

1. *Connection :*
   Log in on your machine.
2. *Project :*
   Download archive `Tp1_Initial.zip` from the course web page, decompress it. The file `CMakeLists.txt` describes the project. The line of interest to us is the following :

   ```
   add_executable(Tp1 Tp1.cpp)
   ```

   It indicates that our program will be called `Tp1` (`Tp1.exe` under Windows) and that the source code to build it is the file `Tp1.cpp`.

   Qt Creator is the programming environment that we will use, it has the advantage to work on all main platforms (Linux, Windows or Mac) and to control directly CMake. [1] To start, launch `Qt Creator` and choose "Open Project" to fetch the file `CMakeLists.txt`. Look at the window bottom tab 6 General Messages, [2] that should print a lot of messages : it is the result of the file read and its interpretation by `CMake`.
3. *Programming :* insert a comment line with your name at the top of file `Tp1.cpp`. Comments are introduced with // and go until the line end. They are ignored by the compiler, but useful for the programmer. Insert then the command

   ```
       cout << "Hello" << endl;
   ```

   on a line before `return 0;` in file `Tp1.cpp`.
4. *Generation :* Push the hammer button at bottom left to generate the program. Look in tab 4 "Compile Output", it says it created the file `Tp1.cpp.o`. Check with the file explorer (outside Qt Creator) that the file is indeed present in the "build" folder. Observe that the program `Tp1` is also there.
5. *Execution :*
   (a) Launch the program with the green arrow in Qt Creator, observe the result in tab 3 "Application Output".
   (b) Check that we created an independent program that we can launch from the command line :
   — Open a terminal (under Windows : Start menu, type the command `cmd`)
   — In the terminal, type the command : `"cd C:\...\build-Tp1..."`
   (learn to take profit of automatic completion with tabulation key `TAB`, just above Caps lock). Under Linux and Mac, the path is different, and replace all \ with / to delimit folders.
   — Check the presence of `Tp1.exe` or `Tp1` with the command `"dir"` (Windows) or `"ls"` (Linux).
   — Type `"./Tp1"`.
6. *Compress :*
   With a right mouse click on folder `Tp1`, build a compressed archive `Tp1.zip` (or `Tp1.7z` if proposed). Such an archive, comprising all files necessary to compilation, will be the format used to upload your exercises and practical session returns under Educnet.

   Note that with CMake we have two folders :
   — The *source* folder containing files `Tp1.cpp` and `CMakeLists.txt` ;
   — The *build* folder, with name chosen by Qt Creator.

1. Most other environments, such as Visual Studio (Windows) and Xcode (Mac), do not know CMake, and we must use beforehand `CMake` to convert into a native project for the environment.
2. If you do not see it, you can display it with the small arrow at the right of the tabs

The most important one is the first, since the second can always be generated again with CMake. Send your instructor only the source folder, the build will be done on his machine. Your *build* folder is useless on another machine if its system is not the same. When you have finished a project, do not hesitate to clean up by removing your *build* folder, but keep cautiously your *source* folder, it is the result of your hard work. Though CMake allows using the same folder for both, it is better to avoid it, to separate clearly sources and automatically generated files. To understand a little how these tools coordinate, you can read Appendix C of the lecture notes.

> **The precious folder containing your work is the *source* folder, `Tp1_Initial`. The disposable one is the *build* folder, `build-Tp1_Initial-...`.**

7. Check that everything could be restarted from scratch : quit Qt Creator, erase your source and build folders, extract the source folder from your archive ; if a file `CMakeLists.txt.user` remains, it was created by Qt Creator, remove it before launching Qt Creator and opening the project.

1.2 First errors

You will make many,[3] it is perfectly normal. Let's get used to it right now.

1. *Modify the program :*
   Modify the program by changing for example the displayed sentence.
   (a) Test a new generation/execution. Check that Qt Creator saves the file automatically (or proposes to save it) before generating.

   > **Launching directly an execution saves and builds automatically.**

   Knowing that, the hammer tool may seem useless. However, it is good when coding to check simply compilation very frequently, even before taking care of execution because there is nothing to observe yet.
2. *Compilation errors*
   Provoke, observe and learn to recognize some compilation errors :
   (a) `includ` instead of include
   (b) `iostrem` instead of iostream
   (c) Forget the `;` after std
   (d) `inte` instead of int
   (e) `cou` instead of cout
   (f) Forget the double quotes `"` ending the string "Hello␣... "
   (g) Add a line `i=3;` before return.

   Talking about errors, it is useful to discover that :

   > **Double-clicking on an error message (Tab 1 "Issues") shows in the editor the code line where it occurs.**

   Note that tab 1 is only an automatic and clickable extract of tab 4, the full error message is found in tab 4. Sometimes, the automatic extraction may have defects and the error message in tab 1 is not clear since incomplete.
3. *Linker errors*
   It is a bit early to have the linker fail,[4] but it will happen at some point. It is still necessary to know the difference between linker and compiler errors. In general, the linker will notify an error if it does not find a function or some variables because some object file or a library is missing. It is also an error if it finds twice the same function...
   (a) Add a line `f(2);` before return and build. It is for now a *compilation* error (the compiler does not know what f designates).
   (b) Fix the compilation error by adding the line (for the moment *"magic"*) `void f(int i);` before the line with main. Build the program : the *linker* complains about the absence of the function `f()` used by main().
   (c) Remove now the call to f in main(). Everything works again. Rename main as `main2`. The absence of function `main` will be noticed by the linker, and it will refuse to build the program.
4. *Indents :*
   With all these changes, the code may not correctly "indented" anymore. It is still essential for a good understanding and to spot some parenthesis error, curly brace, etc. The menu `Edit/Advanced` includes an option to reindent automatically.

3. Even the expert programmer will make some.
4. It will happen more often when we separate our source in several files.

To spot errors, always indent. Ctrl+I = indent the selected zone. Ctrl+A,Ctrl+I = indent all file.

5. *Warnings of the compiler*
   Modifying the main(), provoke some warnings : [5]
   (a)
   ```
   int i;
   i=2.5;
   cout << i << endl;
   ```
   Execute to see the result.
   (b)
   ```
   int i;
   i=4;
   if(i=3) cout << "hello"<< endl;
   ```
   Launch !
   (c) Add `exit;` as first instruction of `main`. Trying to call a function without arguments can happen (and does nothing) ! Launch to see. Correct by putting exit(0); (The function exit() ends the program at once !)

   Some warnings are not not meaningful (false positives) and are not programmer errors. The problem with not correcting them is that they can drown the true positives, you should thus try to fix *all* of them.

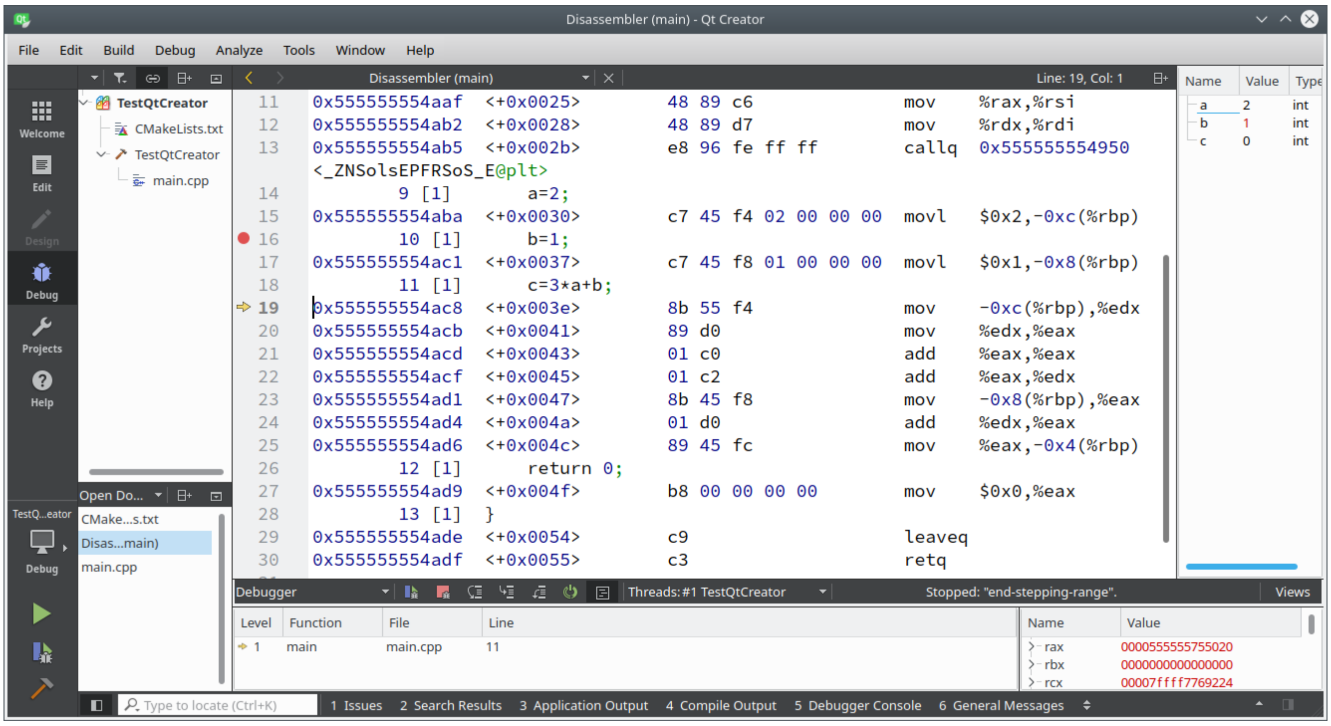
   It is advised not to leave warnings in your code ! Correct them as they appear. A compiler option can even propose to consider them as errors and prevent the compilation to go through !

1.3 Debugger

Knowing how to use the debugger is fundamental. It should be the first reflex in front of an incorrect program. It is a true investigative tool, simpler and more powerful than stuffing the code with additional instructions to spy its run.

1. Type the following function `main`.
   ```
   int main()
   {
       int i,j,k;
       i=2;
       j=3*i;
       if(j==5)
           k=3;
       else
           k=2;
       return 0;
   }
   ```
2. To use the debugger with Qt Creator, we need to compile in Debug mode. The mode is chosen by clicking on the small terminal on top of the green start triangle. Observe that the variable `CMAKE_BUILD_TYPE` takes then value `Debug` in tab "Projects".
3. Launch the program "under the debugger control" with the green triangle button with superimposed bug. The program executes but nothing seems to happen.
4. Put a "breakpoint" by clicking on the left margin at the level of line i=2; then relaunch the debugger.
5. Advance in "`Step over`" with key `F10` or the corresponding button and observe the values of variables (in the special window on the top right or by placing the mouse cursor without click on the variable in source code).
6. Place a second breakpoint and note that `F5` continues the program until the next reached breakpoint.
7. Add i=max(j,k); before the return. Use "`Step into`" or `F11` or the corresponding button when the cursor is on this line. Note the difference with `F10` : `max` is a function call and we indicate the debugger we want to follow what happens inside.

5. Warnings are not errors : they may be present to warn the programmer about possible mistakes, but the sense for the compiler is clear. Hence, depending on compiler options, the warnings may not appear.



```
0x555555554aaf  <+0x0025>         48 89 c6              mov    %rax,%rsi
0x555555554ab2  <+0x0028>         48 89 d7              mov    %rdx,%rdi
0x555555554ab5  <+0x002b>         e8 96 fe ff ff        callq  0x555555554950
<_ZNSolsEPFRSoS_E@plt>
           9 [1]        a=2;
0x555555554aba  <+0x0030>         c7 45 f4 02 00 00 00  movl   $0x2,-0xc(%rbp)
          10 [1]        b=1;
0x555555554ac1  <+0x0037>         c7 45 f8 01 00 00 00  movl   $0x1,-0x8(%rbp)
          11 [1]        c=3*a+b;
0x555555554ac8  <+0x003e>         8b 55 f4              mov    -0xc(%rbp),%edx
0x555555554acb  <+0x0041>         89 d0                 mov    %edx,%eax
0x555555554acd  <+0x0043>         01 c0                 add    %eax,%eax
0x555555554acf  <+0x0045>         01 c2                 add    %eax,%edx
0x555555554ad1  <+0x0047>         8b 45 f8              mov    -0x8(%rbp),%eax
0x555555554ad4  <+0x004a>         01 d0                 add    %edx,%eax
0x555555554ad6  <+0x004c>         89 45 fc              mov    %eax,-0x4(%rbp)
          12 [1]        return 0;
0x555555554ad9  <+0x004f>         b8 00 00 00 00        mov    $0x0,%eax
          13 [1]    }
0x555555554ade  <+0x0054>         c9                    leaveq
0x555555554adf  <+0x0055>         c3                    retq
```

Name	Value	Type
a	2	int
b	1	int
c	0	int

Level	Function	File	Line
1	main	main.cpp	11

Name	Value
rax	0000555555755020
rbx	0000000000000000
rcx	00007ffff7769224

FIGURE 1 – Qt Creator showing machine code and registers.

8. At last, see how machine code looks like : execute until some breakpoint then activate menu `Debug/Operate by Inst`
   We can also in the same menu see the registers of the microprocessor. It can happen we fall in the window "machine code" unwillingly when we debug a program for which we do not have the source file anymore. This display is actually very useful to the programmer to check what the compiler produced and check it optimizes well.
9. The registers are visible by going in the menu Window then Views. See Figure 1.

Useful keys :	**F5**	=	▶	=	**Debug**
	F10	=	↳	=	**Step over**
	F11	=	↘	=	**Step inside**

1.4 If there is time left

Download the supplementary program `Tp1_Final.zip` on the course web page
(`http://imagine.enpc.fr/~monasse/Info`)
play with it... and complete it with the right rebounds!

1.5 Install Imagine++ at home

Go to `http://imagine.enpc.fr/~monasse/Imagine++`. Install on your laptop computer and try the supplementary program with Qt Creator.