Introduction to Modern C++

Lab Session 1
"Hello world" and the compilation process

In this first lab, we will familiarize ourselves with a working environment to write and test programs. We will develop the typical "Hello world" program and explore how the C++ compiler turns C++ code into something that your computer understands.

Important: Make mistakes, break things, and understand what went wrong.

1 Hello world program

The "Hello world" program writes "Hello world" on the screen. It is a minimal working example and will give matter for us to discuss important aspects of C++ programming.

1.1 How to write code?

A C++ program is simply a collection of text files. It is customary to give the `.cpp` extension to these files. To code in C++, you just need a **text editor**. You can use any text editor that you like, but I recommend those that provide some syntax colouring, e.g. `vim` or `emacs`. If you don't know about them, you can use `gedit`.

For the simple "Hello world" program we will only have one file. Large programs (e.g. Firefox) can have several hundreds!

Task 1. Create a file `hello.cpp` and write this code inside:

```cpp
#include <iostream>

int main() {
  std::cout << "Hello world" << std::endl;
}
```

Then save the file, and exit your text editor.

1.2 How to test code?

We created a `hello.cpp` file, but for now the computer cannot use it. We first need to **compile** this program, i.e. translate it into a language that computers understand.

Task 2. Run the following command in a terminal: `g++ hello.cpp -o compiled.exe`

Task 3. Run the `compiled.exe` program by running the following command in a terminal: `./compiled.exe`

We will look back in more detail about this "compiling" business in a moment. In the meantime, you can already write and test your own programs! (yay!)

You can try `clang++` instead of `g++`. I recommend that you play with both.

1.3 Break it, make it, change it – Technology

We will use that to understand the `hello.cpp` program. By playing with it (and breaking it).

Don't forget to compile your program after you modified it, and **read what happens**. Try to understand the error messages.

Task 4. Modify the program to print "Hello C++" instead (write, compile, and run). What happens if you remove the line that starts with `std::cout`? If you replace the << symbol by something else? What if you forget to use the quotation marks `"..."` around the message?

Task 5. Modify the program to print something more interesting, for instance a poem. Can you make it print the message `m\rm`? What happens? Fix it.

Task 6. Modify the program to print "Hello from" on one line, then "the other side" on the next line. What is the purpose of `std::endl`? What happens if you remove the semi-colon `;` ?

Task 7. Modify the program to change `main()` into something else, like `pied()`. What happens? What happens if you forget `{` or `}` ? What does `int` mean in this context?

Task 8. Modify the program by removing the line that starts with `#include`. What happens?

You should now be able to explain every single aspect of the original `hello.cpp` file.

Remark 1. Note that the compiler tries to help. But it is after all a stupid program, and sometimes it misses errors. We will have a complete lecture and lab about that!

Remark 2. When errors happen, it usually means something interesting about how things work. Never underestimate this — do not try to ignore/hide errors from sight. We're learning, and doing mistakes is part of learning.

You can ask the compiler to be very explicit about all possible mistakes (`W` is for "warning"):
`g++ myfile.cpp -o executable.exe -Wall -Wextra`

• **Task 9.** Make a program that asks for your age and prints "You are x years old", where x is your answer. You can use:

```
int age;

std::cin >> age;
```

Test your program extensively (be creative!)

• **Task 10.** Make a program that asks for my birth year and tells me how old I am.

2 The compilation process

Now you can write code and play with it – and we'll do that during the 10 next lectures. But let's focus a moment on the compilation process. It traditionally goes like this:

$$\texttt{.cpp files} \xrightarrow{\text{Pre-processing}} \xrightarrow{\text{Compiling}} \texttt{.s files} \xrightarrow{\text{Assembling}} \texttt{.o files} \xrightarrow{\text{Linking}} \text{Executable file}$$

Today we will understand all these steps.

2.1 Pre-processing

Task 11. Create a file `macro.cpp` and write this code inside:

```
#define MACRO 12345
#define SOMETHING blah

void main() {

  // This is a comment
  // You can add comments too!

  int SOMETHING = MACRO;
  int blah = 12 + 15 * 3 + 4;
}
```

Then run the following command: `g++ -E macro.cpp > macropp.cpp`

This only runs the pre-processor, and puts the result in the file `macrocpp.cpp`. Open this file and understand what happened.

• **Task 12.** Look at what the pre-processor does to you `hello.cpp`. Explain what the `#include` directive does and why you would like to use it.

Remark 3. All commands that start with a pound symbol (#) are pre-processor directives.

Remark 4. Normally, the pre-processor does not create a new file.

2.2 Compiling

This step actually translates C++ code into elementary operations that your computer can understand.

Task 13. Run the command `g++ -S hello.cpp` and look into the newly created file `hello.s`

Remark 5. This language is called Assembly, and depends on your CPU (e.g. Intel, etc.)

2.3 Assembling

The `.s` file of the previous task was easy to read for humans. But it is not easy to read for computers. The assembling step just writes the exact same information in a format that the CPU can directly eat.

Task 14. Run the command `g++ -c hello.cpp` and look into the newly created file `hello.o`.

2.4 Linking

Task 15. Try compiling with `clang` instead of `clang++`. What happens?

3 The toolchain: Chicken or the egg?

• **Question:** How were `g++` and `clang++`, i.e. the preprocessor, compiler, assembler and linker, created?