# The basics of compiling and running

The classic book [KR] popularized the use of the `Hello World!` program. Its only purpose is to learn how to compose, compile an run a simple program writing on the screen the homonymous message.

In general, compilation and execution are not platform independent. We restrict ourselves to Linux systems (or, more properly, Unix-like). There are also a lot of IDEs (Integrated Development Environments) to ease the management of large projects. We only consider console commands.

See [hw] and [lp] for a more complete list.

```
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
```

`C: Hello World` Type a file called `helloworld.c` with your favorite text editor containing the following lines:

```
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
```

To compile the program type in a console terminal (opened in the same path as the file)

```
gcc helloworld.c -o helloworld.out -lm
```

The options after the name of the file are not mandatory. The first one `-o helloworld.out` indicates the name of the resulting executable file. If it is not specified the output file by default is `a.out`. On the other hand, the option `-lm` lnks the `math` library. It is not necessary in our case but it is required to use mathematical functions like sine or exponential. Running the program reduces to type in the console

```
./helloworld.out
```

`C++: Hello World` In some sense C++ programming language is an extension and improvement of C largely caompatible with it. The compile and run process parallels the steps described above. Now the analog file named `helloworld.cpp`

```
#include <iostream>
int main(){
    std::cout << "Hello World!\n";
    return 0;
}
```

The prefix `std::` is cumbersome. The using directive inserted as `using namespace std;` after the first line allows to replace `std::cout` by `cout` and the same for the rest of functions in the library `iostream`. This gives

```
#include <iostream>
using namespace std;
int main(){
    cout << "Hello World!\n";
    return 0;
}
```

To compile the program use

```
g++ helloworld.cpp -o helloworld.out -lm
```

and to run it, proceed as before.

`Java: Hello World` Type the following lines in a text file named `helloworld.java`

```
public class helloworld{
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

The command to compile is

```
javac helloworld.java
```

It generates `helloworld.class`, a program to be interpreted and executed by the so-called Java Virtual Machine (abbreviated as JVM) that is launched by

```
java helloworld
```

This is a standalone (an offline) Java program. There are also Java applets, i.e., Java programs embedded in web pages.

`Java applet: Hello World` Let us say that we name `helloworldapplet.java` to the following lines

```
import java.awt.Graphics;
public class helloworldapplet extends java.applet.Applet {
    public void paint(Graphics g) {
        g.drawString("Hello world!", 40, 20);
    }
}
```

We compile the code as before to get `helloworld.class`. But if we try to run it the JVM will complain saying that there is no `main` in the program. We need a web page (in the same path) acting as a launcher including

```
<applet code="helloworldapplet.class"></applet>
```

at some point of its HTML code. In fact it is convenient to include the atributes `height` and `width` overall if (as in our case) the size of the canvas is not fixed in the applet. The full code of a web page `applet.html` containing the previous applet could be

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html><head>
  <title>An applet</title>
</head><body>
This is the applet:
<applet code="helloworldapplet.class" height="25" width="200"></applet>
</body></html>
```

Naturally `applet.html` will be opened by a java enabled web navigator. For debugging purposes the command

```
appletviewer applet.html
```

allows to visualize the result.

`Maple: Hello World` First of all we need Maple running. There are two modes: the console mode and the graphic mode, usually corresponding to the commands `maple` and `xmaple`, respectively. In the second case we select “Start with Blank Worksheet” in the popup window showing the startup menu (at least in Maple 10) to start from scratch.

In both modes one can type a program line by line, indeed this is the easier in our case, but in general it becomes tedious, overall in the graphic mode (the one that allows to plot functions) whose WYSIWYG (What You See Is What You Get) interface takes too many decisions by itself. For programs exceeding a simple calculation, the best option in both modes is to type in a separate text file. In our case `helloworld.txt` with a single line:

```
printf("\tHello World!\n");
```

To load and run this program, type beside the prompt (`>`) in Maple

```
read "helloworld.txt";
```

The full path is needed if Maple was opened in a different location. When using the graphics mode the interface will try to close the double quotes by itself and the chances of typing by mistake a third pair increase.

This Hello World! program is not the simplest, just `"Hello world!";` in the command line or a separate file does the job. The based on a C `printf` command allows to control format. In our case `\t` refers to the tab character that indents the message with respect to the left margin.

`Matlab/Octave: Hello World` As in Maple, one could type a program line by line in the Matlab or Octave interfaces but reasonably involved programs require separate files. In our case we type in `helloworld.m`

```
fprintf('Hello world!\n');
```

The command

```
helloworld
```

(note the absence of the extension) typed in Matlab or Octave loads and runs the program.

`Python: Hello World` In a text file `helloworld.py` we type

```
# -*- coding: iso-8859-15 -*-
print 'Hello World!\n'
```

The first line is optional and indicates the codification that the programmer is going to use. To run it simply type in a console

```
python helloworld.py
```

Another way of running a Python program is typing `python` in a terminal and after the prompt

```
import helloworld
```

This load the functions defined in the file (in our case there are none) and run the program. This is interesting to add formerly defined functions to our project (reusing is part of Python philosophy). If we try to `import` again a file the program will not be executed because the functions are already supposed to be loaded, then it is not the usual way of running a program.

In Unix-like systems one can convert a Python program in a directly executable command adding the so-called shebang line (according to the Thesaurus dictionary this means an entire system; used in the phrase 'the whole shebang'). It is a heading line indicating how to run Python. In our case the file turns into

```
#!/usr/bin/env python
# -*- coding: iso-8859-15 -*-
print 'Hello World!\n'
```

The shebang line indicates that Python interpreter should be searched at some place in `/usr/bin`. More specific valid paths like `/usr/bin/python` or `/usr/local/bin/python` could be (more) platform dependent. Modifying the file attributes of the new `helloworld.py`, if necessary, to obtain an executable file (`chmod a+x helloworld.py`) it can be run in console with

```
./helloworld.py
```

$\boxed{\texttt{Sage: Hello World}}$ Sage is Python-based then after launching Sage (typically with the console command `sage`) one can type simple Python programs (like `print 'Hello World!\n'`). Following the previous philosophy, we explain how to load a program in a file that in this case we name `helloworld.sage` and contains

```
print 'Hello World!\n'
```

After Sage prompt (`sage:`) to run the program it is enough to write

```
attach "helloworld.sage"
```

It is also valid the more naturally named sentence

```
load "helloworld.sage"
```

This works in part like a kind of analog of `import`. It reads the functions only once (but runs always). Then attach is more convenient when debugging code.

A broadly used way of typing, presenting and running Sage programs is through a web based interface. To access to it, launch Sage with `sage -notebook` or type `notebook()` after the Sage's prompt. A new tab will open in the web browser by default. After logging in (if necessary), select New Worksheet and a name. Click on the box (it will become blue) and type the program. To run it use `evaluate` option or press $\boxed{\text{Ctrl}}+\boxed{\text{enter}}$. There is a menu with different options to save

Like in Python it is possible to turn a Sage program in a script directly runnable in console inserting a shebang line:

```
#!/usr/local/sage-4.5.1/sage

print 'Hello World!\n'
```

that is executed (after changing the attributes if necessary) with

```
./helloworld_2.sage
```

This creates automatically a Python file with the same name and `.py` extension.

There are several other forms of running Sage code through Python code and viceversa or compiling Sage programs to improve performance even loading external C functions. They are detailed in Chapter 5 of [sag].

## References

[hw] HelloWiki! `http://hellowiki.org/`

[KR] B.W. Kernighan, D.M. Ritchie. The C programming language, Prentice Hall, 1978.

[lp] LiteratePrograms `http://en.literateprograms.org/Hello_World`

[sag] Sage Tutorial Release 4.4.4 (by The Sage Development Team), `http://www.sagemath.com/pdf/SageTutorial.pdf` June 24, 2010.