Universität Zürich[UZH]
Physik-Institut



# Lecture 1: Introduction to programming in Linux

## 1 Introduction to Linux environment

The first `Linux` operating systems was developed by Linus Torvald in 1991 August 25. It was originally named `Freeix` as `Free Unix` operating system. A while after its first introduction, there are now many different versions of `Linux` operating systems.
The physics institute of University of Zurich uses the `openSUSE Linux` operating system. CERN mainly uses `ALMA9`, the most popular distribution is `Ubuntu`.
This block course will be dedicated to introduce you on how to program in a `Linux` environment.

### 1.1 Bourne again shell (bash)

The `Unix shell` was first developed by Ken Thompson in the early 1970s, providing a command-line interface that revolutionized how users interact with the operating system. Building on this foundation, the `Bourne again shell` (`bash`) was introduced in 1987 by Brian Fox and has become the default `shell` for most `Unix`-like systems since then. `bash` can be accessed through a *terminal* or *console* (for example, KDE's terminal emulator is called *konsole*). Each time `bash` starts, it executes the commands stored in the *bash run command* (`.bashrc`) file:

```
$HOME/.bashrc
```

The `.` in front of `bashrc` denotes *hidden* files. From your home directory, use the command

```
ls -a
```

to list all of these hidden files.
Note that some of you may be using the `Z Shell` (`zsh`) instead of `bash`. You can check this by running the command

```
$ ps -p $$
  PID TTY          TIME CMD
 5436 pts/3    00:00:00 bash
```

If you see the above, you are using `bash` and you can follow the rest of the instructions literally. If you see the below, you are using `zsh`.

```
$ ps -p $$
  PID TTY          TIME CMD
 7952 pts/0    00:00:00 zsh
```

If you are on `zsh`, you need to replace all instances of `.bashrc` with `.zshrc`. Otherwise the remainder of the instructions will be the same. Any customization you would like to have can be added to your `.bashrc` file. Some of the customizations include `alias` (to define new commands or to run multiple commands in one single command) and re-defining existing `shell` environments (such as the meaning of `$HOME` for the current session). It's important to note that you can very easily break your `shell` environment with a small typo or any changes you make. You should always try to make a backup in a manner similar to this command:

```
cp $HOME/.bashrc $HOME/.bashrc_backup
```

Be aware, however, that if you use the above command, you will replace the existing backup of your `.bashrc` file with a new one. It would be best if you could make multiple backups indicating when you made the backup. We will later go back to creating a *time stamp*. At this stage, note that there are two key environment variables:

- `PATH`: Path to your programs and scripts.
  If you define this wrong, your system will break.
- `LD_LIBRARY_PATH`: Path to your libraries that should be loaded by default.
  If this is set incorrectly, your system may fail to compile or execute certain features.

You can modify these variables by declaring them as follows:

```
export PATH=$PATH:./
```

The above command will tell your `shell` environment to keep the existing `PATH` defined by your operating system, and to also look into your current directory (`./`). *Adding* is denoted with the seperator (`:`). You can add this line with your text editor (`Kate`, `Gedit`, `Emacs`, etc.) to your `.bashrc` file.

Similarly, an `alias` can be defined in the following manner:

```
alias uzh='ssh -X <username>@linux.physik.uzh.ch'
```

With the above setup, you only need to type

```
uzh
```

instead of the full `SSH` command if you want to log into the frontend of the physics institute's `Linux` cluster with your `<username>`. You could define an alias such as

```
alias lt='ls -l -t -ro'
```

so that a simple

```
lt
```

lists all files in a directory in long format (`-l`) sorted by modification time (`-t`), with the most recently changed files appearing at the bottom (`-ro`).

Any `alias` can also be added to your `.bashrc` file, so that you don't have to define it every time you start a new `shell` environment.

## 1.2 Scripting

Also outside of your `.bashrc` file you can execute any number of commands in sequence by writing the commands into a file. Such a file is called a script. However, you need to tell your `bash` *this is a script* before you can use it as a script, as in the script below:

```
1 #!/bin/bash
2 # The line above informs your shell that this is a bash script, so everything that
3 # follows - except comments - will be interpreted as a command.
4 echo "This is a script"
```

The above script can be saved under any name. Let's call it `Script1` and save it. If you try to execute `Script1`, you will get a *Permission denied* error.

```
$ ./Script1
bash: ./Script1: Permission denied
```

This is because you haven't told your file system that you have a permission to execute `Script1`. You can give yourself permission by using the following command:

```
chmod u+x Script1
```

Now the above command to run `Script1` will be executed successfully.

### 1.3 C programming language

Scripting is considered as *light-weight* programming, which allows you to use the computer easily in many different ways, but is limited in features and speed. Since 1972 Dennis Ritchie has been developing a new programming language, intended as an improvement over the `B` programming language (developed in 1969, as a simplification of `BCPL` (*Basic Combined Programming Language*) from 1966), which he named `C`. `C` may not be as simple and easy as a `bash` script, but is much more powerful and has many more features. *Power* in context of programming is the amount of control that your language gives you over the computer.
The main difference between `C` programming language and `bash` scripting language is that if you write a `bash` script, it can be executed as soon as you give your environment permission. For a `C` code, you must *compile* it with libraries so that the computer can understand your commands and execute them.

## 2 Compiling

Compiling allows your computer to assemble your code and all the necessary tools required within the operating system to build a program. One of the biggest advantages of the compilation is that a proper program can be compiled on any platform (`Linux`, `Windows` or `MacOS`) using local systems libraries. This means your code can be compiled and executed on any computer.
There are many options of compilers available to many different operating systems. Some examples are:

- `VisualStudio` has a built-in `C/C++` compiler for Microsoft `Windows`.
- `gcc/g++` are `C/C++` compilers used by `Unix` environments.
- `Clang` is a cross-platform compiler.

For this block course, we will only use `gcc` and `g++` in a `Linux` environment.

### 2.1 Command-line and script compiling

The simplest way to compile is from the command line using `gcc/g++` as follows:

```
g++ source_code.cxx -o compiled_program.exe
# Here, g++ will take the source_code.cxx and
# output a program (-o) specified as compiled_program.exe
```

This is as simple as it can get. To test a small program quickly, this is perfectly fine. However, most programs will have more than one file with source code and need to include external libraries, as in the following example:

```
g++ -c source_code2.cxx -o source_code2.cxx.o
# Compiles the source file source_code2.cxx into an object file source_code2.cxx.o
# (-c: compile only, do not link; -o: specify output filename)
g++ -c source_component.cxx -o source_component.cxx.o
# Compiles source_component.cxx into source_component.cxx.o
g++ -o compiled_program.exe source_component.cxx.o source_code2.cxx.o
# links both object files into an executable compiled_program.exe
```

Obviously, all of the above command lines can be called from a script. You can create and save the following `compilescript` and run it to compile your packages:

```
#!/bin/bash
# compilescript
g++ -c source_code2.cxx -o source_code2.cxx.o
g++ -c source_component.cxx -o source_component.cxx.o
g++ -o compiled_program.exe source_component.cxx.o source_code2.cxx.o
```

Because programmers used scripts to compile their code so often, a separate tool called `make` was developed. It automates compilation based on instructions in a `Makefile`.

```
$ make
make: *** No targets specified and no makefile found.  Stop.
```

Of course, you need to build your `Makefile` before you can use this feature.

## 2.2 Makefile

The `make` utility is a powerful build automation tool. When properly configured with a suitable `Makefile`, it can compile source code spread across multiple directories and subdirectories from the location where it is invoked. It also supports parallel compilation to speed up builds on multi-core systems. When `make` is run, the `shell` launches the `make` program, which looks for a file named `Makefile` (or `makefile`) in the current directory by default. The following is an example of a `Makefile`:

```
# This is the directory of your source code.
sourcedirectory=./

# These are your source codes and components
first_part=source_code.cxx
second_part=source_code2.cxx
component=source_component.cxx

# Here, we're defining the compilers.
CC=gcc
CPP=g++
NVCC=nvcc

# We're defining system variables such as "rm" (remove from system) and "timestamp"
RM=rm
TIMESTAMP=$(shell date +"%Y_%m_%d_T-%H_%M_%S" )

# When a Makefile is executed, by default it tries the option "all"
all: clean first second third
# We tell the Makefile to clean, and then to perform the first, the second,
# and eventually the third compilation step.

# Let's define what "first" is
first:
# Here, we define what "first" will do.
# Using @ before a command hides it from the terminal output.
	@echo Compiling $(sourcedirectory)$(first_part)
	$(CPP) -o $(first_part).exe $(sourcedirectory)$(first_part)
```

```
29          @echo Successfully compiled $(sourcedirectory)$(first_part)
30          @echo The executable is $(first_part).exe
31  second:
32  # We compile the respective source code files in "second".
33          @echo Compiling $(sourcedirectory)$(second_part)
34          $(CPP) -c -o $(second_part).o $(sourcedirectory)$(second_part)
35          @echo Successfully compiled $(sourcedirectory)$(second_part)
36          @echo The unlinked compiled code is $(second_part).o
37          @echo Compiling $(sourcedirectory)$(component)
38          $(CPP) -c -o $(component).o $(sourcedirectory)$(component)
39          @echo Successfully compiled $(sourcedirectory)$(component)
40          @echo The unlinked compiled code is $(component).o
41  third:
42  # In "third" we link the object files from "second" to an executable.
43          @echo Linking $(component).o and $(second_part).o
44          $(CPP) -o compiled_program.exe $(sourcedirectory)$(component).o
            ↪  $(sourcedirectory)$(second_part).o
45          @echo Successfully compiled $(sourcedirectory)$(component)
46          @echo Everything is linked and compiled into compiled_program.exe
47  clean:
48          @echo $(TIMESTAMP)
49          @echo "Making old/$(TIMESTAMP) directory"
50          $(shell mkdir -p old/$(TIMESTAMP) )
51          @echo "Copying the source to the old directory"
52          $(shell cp -r $(sourcedirectory)/*.h old/$(TIMESTAMP) )
53          $(shell cp -r $(sourcedirectory)/*.cxx old/$(TIMESTAMP) )
54          @echo "Moving all .exe to the old directory"
55          $(shell mv *.exe old/$(TIMESTAMP) )
56          $(shell mv *.o old/$(TIMESTAMP) )
57          @echo "Copying the Makefile to the old directory"
58          $(shell cp Makefile old/$(TIMESTAMP) )
```

Note that indentation in a `Makefile` is mandatory and must be done with a `tab` character, not spaces, to conform to `make` syntax. Using spaces instead of a tab will cause `make` to fail.

Please inspect the above materials for the practical programming session. Please do not blindly copy-and-paste, since that would defeat the purpose of the practical session. The first practical part of the course will be on getting familiar with `bash` and compiling a given code. Please do type everything from scratch, or at least go thoroughly through every single line of code.

**We will now start a 60-minute practical programming session.**

# 3 Practical Session – Part I

### Compile `one.cxx` in three different ways

1. Create a file called `one.cxx` and fill it with the following lines:

```cpp
// one.cxx
#include <iostream> // "include" input/output stream.
#include <string> // "include" the string data type explicitly
using namespace std; // using standard namespace
// Otherwise, every instance of commands such as cout needs a prefix: std::cout .
int main (){ // Beginning of the function "integer" main
  string name = "You";
  cout << "Hello " << name << endl;
  return 0; // Return and completing of the function.
}
```

   This is a `C` code that uses `C++` standard namespace and `C++` libraries. `C++` is actually a large repository of functions created for `C` language.

2. Compile `one.cxx` into `one.exe` from command line using `g++`.

3. Execute `one.exe`, and if it runs, delete it.

4. Write a `bash` script to compile `one.cxx` as `s_one.exe` and use it.

5. Create a `Makefile` that has the following options:

   – `install`: Compile `one.cxx` with standard `C++` flags and create an executable called `m_one.exe`

   – `clean`: Create a directory called `old`, then a subdirectory therein with the time stamp `yyyy-mm-dd-hh-mm-ss`, and copy `one.cxx` and `Makefile` into the new subdirectory. `clean` must also remove `m_one.exe` if it exists.

   – `all`: performs `clean`, then `install`.

### Compile `two.cxx` with `three.cxx`

1. Create the files `two.cxx`, `three.cxx`, and `three.h`:

```cpp
// two.cxx
#include <iostream> // "include" input/output stream.
#include <string> // "include" the string data type explicitly
#include "three.h" // This time, you are including a local package called "three.h"
using namespace std; // using standard namespace
// Otherwise, every instance of commands such as cout needs a prefix: std::cout .
int main (){ // Beginning of the function "integer" main
  string name = "You";
  cout << "Hello " << name << endl;
  greet(name);
  return 0; // Return and completing of the function.
}
```

```cpp
// three.cxx
#include <iostream> // "include" input/output stream.
#include <string> // "include" the string data type explicitly
#include "three.h" // This time, you are including a local package called "three.h"
using namespace std;
void greet(string input){
  cout << "Hi " << input << endl;
}
```

```cpp
// three.h
#include <string> // "include" the string data type explicitly
using namespace std;
void greet(string input);
```

The file `three.h` is a *header* file related to `three.cxx`. The name of the header file does not need to match that of the corresponding `.cxx` file, but it is recommended. Note that `three.h` has an *empty* version of the `greet` function. This empty version is called a constructor in object-oriented programming. Further note that `two.cxx` must include `three.h`, whereas `three.cxx` does not have to necessarily (but it is common practice), and no header corresponding to `two.cxx` is required. The lecture part will give further explanations; for now, let's just compile them.

2. Compile `two.cxx` with `three.cxx` from command line using `g++` as `four.exe`.
3. Execute `four.exe`, and if it runs, delete it.
4. Write a `bash` script to compile `two.cxx` with `three.cxx` as `s_four.exe` and use it.
5. Create a `Makefile` that has the following options:
   - `install`: Compile `two.cxx` with `three.cxx` and standard `C++` flags, then create an executable called `m_four.exe`.
   - `clean`: Create a directory called `old`, then a subdirectory therein with time stamp `yyyy-mm-dd-hh-mm-ss`, and copy `two.cxx`, `three.cxx`, `three.h` and `Makefile` into the new subdirectory. `clean` must also remove `m_four.exe` if it exists.
   - `all`: performs `clean`, then `install`

**Compile `one.cxx` with `three.cxx` with separate assembler/linker steps**

1. Compile `three.cxx` as linker `three.cxx.o`, then compile `three.cxx.o` with `one.cxx` to `five.exe` using `g++`.
2. Execute `five.exe`, and if it runs, delete it.
3. Do the same using a `bash` script to produce `s_five.exe`.
4. Create a `Makefile` that has the following options:
   - `assembler`: Creates the `three.cxx`.o object file.
   - `linker`: Compiles `one.cxx` and links it with `three.cxx`.o to produce the executable `m_five.exe`.
   - `clean`: Create a directory called `old`, then a subdirectory therein with time stamp `yyyy-mm-dd-hh-mm-ss`, and copy `one.cxx`, `three.cxx`, `three.h` and `Makefile` into the new subdirectory. `clean` must also remove `m_five.exe` if it exists.
   - all: performs `clean`, `assembler`, then `linker`.

---

## We will now continue with a 15-minute theory session.

---

# 4 Programming in C language

Until now, you have been actually working in `bash` to compile a couple of `C++` codes. We will now go over to programming in `C`. You may be wondering why we are starting with `C`. The simple answer is that there is no difference between `C` and `C++`. The `C++` project is an on-going effort to add new features and to improve the existing `C` language. With `C`, you have the capability of building any new objects or functions into packages, and you can use them in any of your future works. `C++` are literally additional packages (`++`) available for `C`. We will program in `C` language, but will be using `C++` packages. In simpler terms, we will program in `C++`.

## 4.1 Main

As you may have noticed in previous works, there is a basic structure you must follow in a `C` program. First you must have a `main` code. Inspect below a simple `main.cxx` code.

```
// main.cxx
#include <iostream> // "include" input/output stream.
#include <stdio.h> // "include" headers from default library directory
int main (){
  return 0;
}
```

The file `main.cxx` will do the following: when the program is executed, it will `return` the *integer* (`int`) 0 to the command line. The two included *headers* `iostream` and `stdio.h` are redundant here, but we need them as soon as we want to use any input or output.
For `C`-style functions, we need to make sure that the compiler reads the *standard input/output header* (`stdio.h`) ahead of the main code. In many cases this header file is located in:

```
/usr/include/stdio.h
```

This is found because your `bash` is looking into the `PATH` variable, which contains

```
/usr/bin/
```

and from there, your shell gets an instruction to look into

```
/usr/include/
```

whenever you write

```
#include
```

To make use of the `C++` *input/output* functions, you need to tell the compiler *I am giving you a stream of input and I expect an output.* To do so, you need to include the *header* for *input/output stream* (`iostream`) by placing

```
#include <iostream>
```

at the top of your code. We will typically use the `C++` *input/output* functions, but occasionally also some `C` functions. In that case, we are recommended to use the `cstdio` header instead of `stdio.h` as it is more consistent with `C++` style. For our simple examples it will usually not make any difference.

Note that, while all functions inside your `main` code end with a semicolon (`;`), the `include` lines do not. This is because your computer is not reading these codes as `C` code, as they are *preprocessor directives*.

Let's start with the simplest example with output in C++ style, namely Helloworld1.cxx:

```
// Helloworld1.cxx
#include <iostream> // "include" input/output stream.
int main (){
  std::cout << "Hello World" << std::endl;
  return 0;
}
```

Note that, in standard syntax, *character out* (cout) and *end line* (endl) need to have the prefix std::. You can avoid typing std:: each time by specifying that your program will use names from the C++ *standard* namespace (std), like in Helloworld2.cxx:

```
// Helloworld2.cxx
#include <iostream> // "include" input/output stream.
using namespace std;
int main (){
  cout << "Hello World" << endl;
  return 0;
}
```

By using the namespace provided by the *standard* (std) package, we no longer have to declare std:: for many simple functions.

## 4.2 Functions

You do not have to code everything in main. It is also not advisable because the reason we program is to get the computer to do the same task many times and quickly. If we write everything in main, we have to write this section of code each time we need it. We can instead write this subsection of code somewhere else, in a *function*, like in Helloworld3.cxx:

```
// Helloworld3.cxx
#include <iostream> // "include" input/output stream.
using namespace std;
void sayhello(){
  cout << "Hello world" << endl;
}
int main (){
  sayhello();
  return 0;
}
```

We can also let the function take some input, like in Helloworld4.cxx:

```
// Helloworld4.cxx
#include <iostream> // "include" input/output stream.
#include <string> // "include" the string data type explicitly
using namespace std;
void sayhelloto(string input){
  cout << "Hello " << input << endl;
}
int main (){
  sayhelloto("world");
  sayhelloto("Steven");
  return 0;
}
```

Since we are using the data type string explicitly here, it is good practise to include its respective header file explicitly, even though it is already implicitly included through iostream.

A function can also have several *arguments* and/or a *return value*, like in adding.cxx below:

```cpp
// adding.cxx
#include <iostream> // "include" input/output stream.
using namespace std;
int add(int first, int second){
  return first + second;
}
int main (){
  cout << add(2, 4) << endl;
  return 0;
}
```

You can also choose to put the functions below `main`, like in `Helloworld5.cxx`:

```cpp
// Helloworld5.cxx
#include <iostream> // "include" input/output stream.
#include <string> // "include" the string data type explicitly
using namespace std;
void sayhelloto(string input);
int main (){
  sayhelloto("world");
  sayhelloto("Steven");
  return 0;
}
void sayhelloto(string input){
  cout << "Hello " << input << endl;
}
```

You may notice that the function is declared before `main` via:

```cpp
void sayhelloto(string input);
```

This is called a *constructor*. The constructor tells the computer: *There is a function within this code called* `sayhelloto`*, which takes in a* `string` *as an input. If you see* `sayhelloto`*, don't panic and look for it first.*

Note that in `C++` functions can be *overloaded*, i.e. you can define multiple functions with the same name but different parameter lists (different number or types of parameters). Each version is a separate function, and the compiler automatically selects the appropriate one at compile time based on the arguments provided. Vice versa, this means that you need to call a function with parameters that match those declared in the function, since otherwise you would try to call a function that does not exist (for this set of input parameters).

### 4.3 Package building

Obviously, writing everything into a single file gets messy quickly. Furthermore, no function you build will be accessible by other programs. You can fix these problems by writing functions somewhere else separately.
You need to make sure that you compile the external functions with the `main` function, and also that the compiler is aware of the fact that there are external functions. Your `main` code should then look like in `Helloworld6.cxx`:

```cpp
// Helloworld6.cxx
#include <iostream> // "include" input/output stream.
#include <string> // "include" the string data type explicitly
#include "myfunctions.h" // "include" the "local" function package.
using namespace std;
int main (){
  sayhelloto("world");
  sayhelloto("Steven");
  return 0;
}
```

Of course, you need to create the *header* file `myfunctions.h` that is included:

```
// myfunctions.h
#include <string> // "include" the string data type explicitly
using namespace std;
void sayhelloto(string input);
```

and the separate file `myfunctions.cxx` where the content of the function is contained:

```
// myfunctions.cxx
#include <iostream> // "include" input/output stream.
#include <string> // "include" the string data type explicitly
#include "myfunctions.h"
using namespace std;
void sayhelloto(string input){
  cout << "Hello " << input << endl;
}
```

Note that all standard `C++` headers have internal *include guards* (or equivalent mechanisms) to prevent multiple inclusion from causing redefinition errors. This is not automatically the case for your own headers. While it is not an issue so far, you are recommended to use *include guards* right from the beginning, to avoid issues in future work. The following — otherwise equivalent — version of `myfunctions.h` will not give rise to redefinition errors if included multiple times:

```
// myfunctions.h - include-guarded version
#ifndef MYFUNCTIONS_H
#define MYFUNCTIONS_H

#include <string> // "include" the string data type explicitly
using namespace std;
void sayhelloto(string input);

#endif // MYFUNCTIONS_H
```

Eventually, if you want to accept input from the command line in your code, you can do so using `cin` (from `iostream`, in contained `namespace std`), like in `Helloworld7.cxx`:

```
// Helloworld7.cxx
#include <iostream> // "include" input/output stream.
#include <string> // "include" the string data type explicitly
#include "myfunctions.h" // "include" the "local" function package.
using namespace std;
int main (){
  string name;
  cout << "Who should be greeted? ";
  cin >> name;
  sayhelloto(name);
  return 0;
}
```

This concludes the theory section for today. Now, let's practice.

**We will now start a 60-minute practical programming session.**

## 5 Practical Session – Part II

### Hello world

1. Reproduce, compile, and run all of the codes shown in the previous theory section.
2. Create a `main` function that accepts a `string` from the command line and uses the `myfunctions.cxx` package to print

```
Hello <inputstring>
```

3. Extend the `myfunctions.cxx` package to perform basic addition and subtraction operations on two integers.
4. Compile `myfunctions.cxx` together with a `main` that first returns the value of $2 + 3$, then the value of $4 - 7$, and then says hello to you.
5. Modify your code such that your program takes 10 integers and returns the sum of all of them.
6. Research and read about input and output arguments for the `main` function:

```
1 int main (int argc, char *argv[])
```

**We will now conclude with a 15-minute question session.**

## 6 Conclusion

You have now learned and programmed some basic codes that will be useful for all of your future works in `C++`. In the example with 10 integers you will have realised that loops and containers of variables could be convenient. We will cover this in tomorrow's lecture.

As part of homework, I strongly recommend you to try to implement `argc` and `char* argv[]` in your code such that you can execute your program with a number of arguments, like:

```
$ addintegers.exe 2 3 4
9
```