Universität Zürich[UZH]
Physik-Institut

Lecture 1: Introduction to programming

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. Since its first introduction there are now many different variations of Linux operating systems.
The physics institute if University of Zurich uses the OpenSUSE linux operating system. CERN mainly uses CENTOS7, 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)

In 1970 Bourne again shell was developed by Ken Thompson and since then all Unix distribution is built on a Bourne again shell or "bash". Bash is a command line shell which can be accessed as "terminal" or "console" (The KDE version of console is called konsole). Each time when Bourne again shell is loaded, it runs the commands written in the "bash run command (bashrc)" file

```
$HOME/.bashrc
```

once. The dot in front of bashrc denotes "hidden" files. From your home directory, to see all of these hidden files using command. Note that some of your accounts on spinorX.physik.uzh.ch may be using the "z" shell 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 exactly. If you see 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.

```
ls -a
```

list subcomponents -all.
Any customization you would like to have can be added into the following run command file:

```
$HOME/.bashrc
```

Some of the customization include alias (to run multiple commands in one single command) and re-defining existing shell environments (such as the location of home directory). 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 with a new one. It would be best if you can make multiple backups indicating when you made the backup. We will later go on creating a "time-stamp". For now, it is important to note that there are a two environment variables that are most important:

- PATH: Path to your programs and scripts.
  If you define this wrong, your system will break.
- LD_LIBRARY_PATH: Path to your library to load on default.
  If you define this wrong, your system will not be able to compile and/or execute some of the features.

You can modify these variables by declaring as such:

```
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 at your current directory (./). "Adding" is denoted with seperator (:). Now Add this line to your text editor (kate, gedit, emacs or etc.)

```
$HOME/.bashrc
```

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

```
alias superclear="clear; clear; clear; clear; clear"
```

the command "clear" is a default command in the bash environment. If you try using this command now, you will realize that while your terminal is cleared when you scroll up in your terminal session, you will see your previous command input and output. If you don't want to see any of your previous works in your terminal session, you must execute the command "clear" multiple times. You can also tell your environment to instead "clear many times" you can define a new command (alias) to do so. Here the semi-colon(;) denotes "end of one command". Alias can also be added into your

```
$HOME/.bashrc
```

so that you don't have to define the aliases everytime when you start a new shell environment.

1.2 Scripting

Outside of $HOME/.bashrc, you can execute any number of commands in sequence by writing the commands into a file. This file is called a script. However you need to tell your Bourne again shell "this is a script" before you can use as a script. Please see "Script" below:

```
1 #!/bin/sh
2 #The above tells your shell that this is a script and anything below other
  ↪  than comments
3 #Should be treated as a command.
4 echo "This is a script"
```

The above script can be saved anything. 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 Script. You can give yourself permission by using the following command:

```
chmod u+x Script1
```

Now you can run it using the command:

```
Script1
```

1.3 C programming language

Scripting is considered as "light-weight" programming which allows you to use the computer in many different ways easily but it is limited in features and speed. Since 1972 Dennis Ritchie has been a new programming that is "better than B programming language" and he named C. C may not be as simple and easy as a simple bash script, it is much more powerful than B 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 from bash scripting language is that when 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 command and execute it.

2 Compiling

Compiling allows your computer to assemble your code and all the necessary tools required within the operating system and build a program. One of the best advantage of the compilation is that a proper program can be compiled in any platform (Linux, Windows or MacOS) using local systems libraries. This means your code can be compiled in any computer and can be executed.
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 environment.
- cmake 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

Simplest way to compile is from command line using gcc/g++ as seen below:

```
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 it can gets. To test a small program quickly, this is a perfect. However most program will have more than one source code, and need to include external libraries. See the example below:

```
g++ -c source_code2.cxx -o source_code2.cxx.o
#-c compile the source file source_code2.cxx by itself knowing that it is missing components.
#-o output the partially compiled output as source_code2.cxx.o
g++ -c source_component.cxx -o source_component.cxx.o
#Compile the partial source code and its component together into compiled_program.exe
g++ -o compiled_program.exe source_component.cxx.o source_code2.cxx.o
```

Obviously all of the above command lines can be called from a script. You can create and save "compilescript" and activate it to compile your packages.

```
#!/bin/sh
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
```

In fact, programmers used a script to compile their code so often, there is a default compiling capability built into B-shell and you can invoke it by stating

```
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 built-in make function is impressive, it can compile all codes within all directories and subdirectories from where make is called if it has been programmed properly. It also has a built-in parallel compilation function.
When make is invoked, B-shell will look for a file named "Makefile" it at the current directory. Below 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 systems variable such as "remove" from system and "timestamp"
RM=rm
TIMESTAMP=$(shell date +"%Y_%m_%d_T-%H_%M" )

```

```
#When a Makefile is executed, by default it tries the option "all"
all: clean first second third
#We will tell the makefile to clean, compile the first component, second then the third component.

#Let's define what the first one is
first:
#Here, we define what "first_one" will do.
#At sign @ will silence the command appering in the terminal
	@echo Compiling $(sourcedirectory)$(first_part)
	@$(CPP) -o $(first_part).exe $(sourcedirectory)$(first_part)
	@echo Successfully compiled $(sourcedirectory)$(first_part)
	@echo The executable is $(first_part).exe
second:
#Let's now compile the second part.
	@echo Compiling $(sourcedirectory)$(second_part)
	@$(CPP) -c -o $(second_part).o $(sourcedirectory)$(second_part)
	@echo Successfully compiled $(sourcedirectory)$(second_part)
	@echo The unliked compiled code is $(second_part).o

	@echo Compiling $(sourcedirectory)$(component)
	@$(CPP) -c -o $(component).o $(sourcedirectory)$(component)
	@echo Successfully compiled $(sourcedirectory)$(component)
	@echo The unliked compiled code is $(component).o
third:
#Let's link the second part
	@echo Linking $(component).o and $(second_part).o
	@$(CPP) -o compiled_program.exe $(sourcedirectory)$(component).o $(sourcedirectory)$(second_part).o
	@echo Successfully compiled $(sourcedirectory)$(component)
	@echo Everything is linked and compiled into compiled_program.exe
clean:
	@echo $(TIMESTAMP)
	@echo "Making old/$(TIMESTAMP) directory"
	$(shell mkdir -p old/$(TIMESTAMP) )
	@echo "Copying the source to the old directory"
	$(shell cp -r $(sourcedirectory)/*.h old/$(TIMESTAMP) )
	$(shell cp -r $(sourcedirectory)/*.cxx old/$(TIMESTAMP) )
	@echo "Moving all .exe to the old directory"
	$(shell mv *.exe old/$(TIMESTAMP) )
	$(shell mv *.o old/$(TIMESTAMP) )
	@echo "Copying the Makefile to the old directory"
	$(shell cp Makefile old/$(TIMESTAMP) )
```

Please inspect the above materials for the practical programming session. Please do not copy and paste, as that defeats the purpose of the practical session. The first practical part of the course will be on getting familiar with Bourne again shell and compiling a given code. Please do type everything from scratch.

= The practical programming part of this course will now begin for 60 minutes. =

3 Compile one.cxx in three different ways

Create a file called "one.cxx" and write in the following lines:

```
//one.cxx
#include <iostream> //"including" input/output stream.
#include <stdio.h> //"including" headers from default library directory
using namespace std; //using standard namespace, otherwise every instances
↪ of command such as cout need the prefix std:cout or std:endl.
int main (){ //Beginning of the function "integer" main
string name="You";
cout<<"Hello "<<name<<endl;
return 0; //Return and completing the function.
}
```

This is a C code that uses C++ namespace and C++ library. C++ is really a large repository of functions created for C language.

3.1 First exercise

1. Compile one.cxx into one.exe from command line using g++.
2. Execute one.exe and if it runs, delete one.exe
3. Write a bash script to compile one.cxx as s_one.exe and compile.
4. Create a Makefile that will have the following options:
   - install: Compile one.cxx with standard c++ flags and creates an executable called m_one.exe
   - clean: Create a directory called "old" then create a sub directory with time stamp (yyyy-mm-dd-hh-mm-ss) and copy one.cxx and Makefile into the new sub directory. Clean must also remove m_one.exe if it exists.
   - all: performs clean then install

4 Compile two.cxx with three.cxx

Inspect the file "two.cxx"

```
//two.cxx
#include <iostream> //"including" input/output stream.
#include <stdio.h> //"including" headers from default library directory
#include "three.h"//This time, you are including a local package called
↪ "three.h"
using namespace std; //using standard namespace, otherwise every instances
↪ of command such as cout need the prefix std:cout or std:endl.
int main (){ //Beginning of the function "integer" main
string name="You";
cout<<"Hello "<<name<<endl;
greet(name);
return 0; //Return and completing the function.
}
```

Inspect the file "three.h"

```
//three.h
#include <iostream> //"including" input/output stream.
#include <stdio.h> //"including" headers from default library directory
using namespace std;
void greet(string input);
```

"three.h" is a "header" file which will be read ahead of three.cxx. the name of the header files must same as the cxx package. Now, let's inspect "three.cxx".

```
//three.cxx
#include <iostream> //"including" input/output stream.
#include <stdio.h> //"including" headers from default library directory
#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;
}
```

Note that "three.h" has an "empty" version of "greet" function. The empty version is called a constructor in object oriented programming. Also note that all three files must include "three.h" but no header is required for two.cxx. The lecture part will explain further. For now, let's compile them.

4.1 Second exercise

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

4.2 Third exercise

1. Compile three.cxx as linker three.cxx.o then compile three.cxx.o with one.cxx as five.exe using g++
2. Execute five.exe and if it runs, delete five.exe
3. Do the same using bash script as s_five.exe and compile.
4. Create a Makefile that will have the following options:
   - linker: Creates three.cxx.o linker.
   - assemble: Compiles three.cxx.o with one.cxx
   - clean: Create a directory called "old" then create a sub directory with time stamp (yyyy-mm-dd-hh-mm-ss) and copy one.cxx, three.cxx, three.h and Makefile into the new sub directory. Clean must also remove m_five.exe if it exists.
   - all: performs clean, linker then assemble

= The theoretical lecture part of this course will now continue for 15 minutes. =

5 Programming in C language

Until now, you have been really working in Bourne again shell to compile a couple of C++ codes. We will now go over 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 improve on existing C language. With C, you have the capability of building any new objects or functions into a 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 we will be using C++ packages. In a simpler way, we will program in C++.

5.1 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.

```
1 //main.cxx
2 #include <iostream> //"including" input/output stream.
3 #include <stdio.h>
4 int main (){
5 return 0;
6 }
```

main.cxx will do the following: when the program is executed, it will return an integer 0. In C world, "return" means, return to command line. However, no one tells the computer what "int" or "return means, so we must tell the computer to make sure it reads "standard input/output (stdio)" ahead of the main code. In many cases this header file is located in:

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

This is because your Bourne again shell is looking at the PATH

```
/usr/bin/
```

and from there, your shell gets an instruction to look at:

```
/usr/include/
```

whenever you write

```
1 #include
```

You should also note that you need to tell computer "I am giving you a stream of input and I expect an output." To do so, you need to include the input/output stream (iostream) by placing

```
1 #include <iostream>
```

at the top of your code. Also notice that while all functions inside your "main" code ends with a semicolon(;), they are not present for the "include" lines this is because your computer is not reading these codes as "C" code yet. Now, let's continue and look at another simple code.
Now, let's inspect Helloworld1.cxx below:

```
//Helloworld1.cxx
#include <iostream> //"including" input/output stream.
#include <stdio.h> //"including" headers from default library directory
int main (){
std::cout<<"Hello World"<<std::endl;
return 0;
}
```

Notice that a standard syntax for "c out" and "end line" need to have prefix std:: You can avoid typing std:: everytime by declaring that you will be using names from c++ std name package. Inspect Helloworld2.cxx below:

```
//Helloworld2.cxx
#include <iostream> //"including" input/output stream.
#include <stdio.h> //"including" headers from default library directory
using namespace std;
int main (){
cout<<"Hello World"<<endl;
return 0;
}
```

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

5.2 Methods

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 the section of code everytime we need to. We can instead write this subsection of code somewhere else. Inspect Helloworld3.cxx below:

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

We can also make sure that the method will take an input.

```
//Helloworld4.cxx
#include <iostream> //"including" input/output stream.
#include <stdio.h> //"including" headers from default library directory
using namespace std;
void sayhelloto(string input){
        cout<<"Hello "<<input<<endl;
}
int main (){
sayhelloto("world");
```

```
sayhelloto("Steven");
return 0;
}
```

You can also expect a return.

```
//adding.cxx
#include <iostream> //"including" input/output stream.
#include <stdio.h> //"including" headers from default library directory
using namespace std;
int add(int first, int second){
        return first+second;
}
int main (){
cout<<add(2,4)<<endl;
return 0;
}
```

This can get very messy very quickly. You can also choose to put the functions below main.

```
//Helloworld5.cxx
#include <iostream> //"including" input/output stream.
#include <stdio.h> //"including" headers from default library directory
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 before main

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

is declared. This is called a "constructor". The constructor tells the computer that, "Hey, 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.

5.3 Package building

Obviously writing everything into a single main code gets messy quickly. Furthermore, no function you build will be accessible by any other programs. You can fix these problems quickly by writing the function somewhere else separately.
You need to make sure you compile the external function with the main function and also make sure that the compiler is aware that there are external functions. Your main code must look as the following

```
//Helloworld6.cxx
#include <iostream> //"including" input/output stream.
#include <stdio.h> //"including" headers from default library directory
#include "myfunctions.h" // " is to used to denote "local" library.
```

```
using namespace std;
int main (){
sayhelloto("world");
sayhelloto("Steven");
return 0;
}
```

Of course you need to build your myfunctions.h

```
//myfunctions.h
#include <iostream> //"including" input/output stream.
#include <stdio.h>
using namespace std;
void sayhelloto(string input);
```

and your myfunctions.cxx

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

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

= The practical programming part of this course will now begin for 60 minutes. =

6 Hello world

1. Reproduce, compile and run all of the codes shown from theory section.
2. Create a main function that accepts a string by

```
std::cin
```

   and uses myfunctions.cxx to print

```
"hello " inputstring.
```

3. Create a new functions package that performs simple add and subtraction operations for 2 integers each.
4. Compile myfunctions.cxx, the new function and main together and to return values of 2+3, then returns 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.

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

7 Conclusion

You have now learned and programmed some basic codes that will come useful for all of your future works in C++. As part of homework, I strongly recommend you to try to impement in argc and char* argv[] in your code such that you can execute your program with a number of arguments.

```
~>addintegers.exe 2 3 4
9
```