Programming in GNU/Linux

Ivan Marković Matko Orsag Damjan Miklić
(Srećko Jurić-Kavelj)

University of Zagreb, Faculty of Electrical Engineering and Computing,
Departement of Control and Computer Engineering

2015

University of Zagreb
Faculty of Electrical Engineering
and Computing

Before we begin

- Go to `Ubuntu Software Center`→`Edit`→`Software Sources` and make sure that under `Downloadable from the Internet` the `main` and `universe` checkboxes are checked
- Now run the update

```
$ sudo apt-get update
```

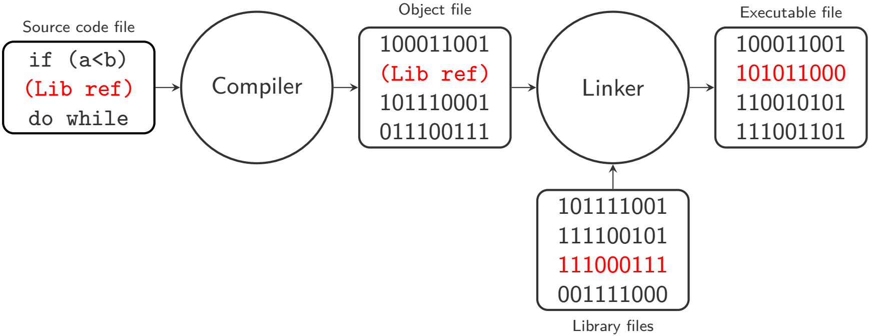
- Be sure to install the following for the class

```
$ sudo apt-get install cmake g++
```

- For homework you will also need OpenCV (probably already installed!)

```
$ sudo apt-get install ros-indigo-vision-opencv
```

Developing a program



http://www.aboutdebian.com/compile.htm

Writing a program in Linux

- Let's write a program called `hello.c` in C

  ```c
  #include <stdio.h>
  int main()
  {
    printf("Hello!\n");

    return 0;
  }
  ```

- Use your favorite text editor to make the source code

  ```
  $ nano hello.c
  $ gedit hello.c
  ```

Compiling proper

- People often under ‘compilation’ mean the entire build process (compiling and linking), so to emphasize the literal step we say ‘compiling proper’
- To compile the source code we call the C compiler (gcc)

```
$ gcc -c hello.c
```

- The result is an object file `hello.o`, which is nothing but a machine-readable source code version with references to library functions

Linking

- To link the object file with certain libraries which contain ‘built-in’ functions (like `printf`) we execute

```
$ gcc -o hello hello.o
```

- This step replaces the references in the object file with functions from library files (for static libraries)
- The result is a binary file `hello` that we can run (since usually not in `PATH` we must specify the location with `./`)

```
$ ./hello
Hello!
```

- Possibly the binary will not be executable — check with `ls -la` and change if needed with `chmod`

Splitting the code

- Let's write a function in a separate file `print_time.c` that will print the date and time, and which will be called from `hello.c`

```
#include <stdio.h>
#include <time.h>

void print_time(void)
{
  time_t now;
  time(&now);

  printf("Today is %s\n", ctime(&now));
}
```

Splitting the code

- To be able to use it in `hello.c` we need to write also the header file `print_time.h` with the function prototype

  ```c
  void print_time(void);
  ```

- Now we modify the `hello.c`

  ```c
  #include <stdio.h>
  #include "print_time.h"

  int main()
  {
  printf("Hello!\n");
  print_time();

  return 0;
  }
  ```

Splitting the code

- Building and execution goes as follows

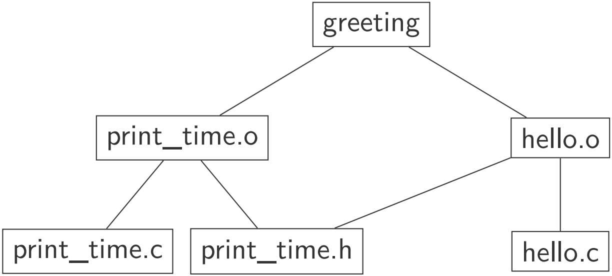
```
$ gcc -c hello.c print_time.c
$ gcc -o greeting hello.o print_time.o
$ ./greeting
Hello!
Today is Thu Sep 27 10:17:01 2012
```

- If only `print_time.c` changes we do not need to compile the whole project again
- How to build and take care of dependencies automatically?

Makefiles

- When project consists of multiple files (`*.c`, `*.h`) manual compiling and linking quickly becomes tedious
- Makefiles with the `make` utility can automatically build and manage the project
- Let's look at the dependency tree of the `greeting` project

Makefiles


greeting
print_time.o
hello.o
print_time.c
print_time.h
hello.c

Makefiles

- Makefile has the following format

```
target: source file(s)
  command (must be preceded by a tab)
```

- The greeting project makefile (connect it with the dependency tree)

```
greeting: hello.o print_time.o
  gcc -o greeting hello.o print_time.o
hello.o: hello.c
  gcc -c hello.c
print_time.o: print_time.c
  gcc -c print_time.c
```

Makefiles

- When we call `make` the utility will read the `makefile`

```
$ make
gcc -c hello.c
gcc -c print_time.c
gcc -o greeting hello.o print_time.o
```

- Often practical to include is `clean` target to get rid of built files (no dependencies are stated)

```
clean:
	@rm -rf *o greeting
```

- When executed as `make clean` it will delete all `*.o` files and the `greeting` binary
- If `@` is placed then there will be no output on the terminal

Libraries

- Libraries can be static (`*.a`) or shared (*.so)
- At linking stage the static library gets placed in the final program, while in the case of shared library only a reference is placed inside
- Program having references to shared libraries must be able to see them during execution
- Libraries are usually located in `/lib, /usr/lib` and `/usr/local/lib`

CMake

- A cross-platform, open-source build system
- In other words, a tool for making makefiles (in case of Linux) which wraps around native build system (e.g. if on another computer we don't have `gcc` but `gcc-xyz` we would have to replace it in the `makefile`)
- Instead of explicitly writing dependencies and commands as in the case of a `makefile`, with CMake we will describe how to make a `makefile`
- Of course, the final result will be an automatically generated `Makefile`

CMake

- Let's build our previous project greeting (without the print_time.c for now) with CMake
- Make a text file called CMakeLists.txt (this is what CMake will read to do its magic)
- In our project directory with the source hello.c make a folder called build where all our build files will be stored

CMake

- Edit the CMakeLists.txt as follows

```
# Specify the version being used
cmake_minimum_required(VERSION 2.8)
# We name our project
project(greeting)
# This tells CMake to compile hello.c and name it hello
add_executable(greeting hello.c)
```

- Navigate to the build folder and run

```
$ cmake ..
$ make
```

- We tell CMake where the sources are and then we make the project

CMake

- Let's now build the greeting project, but with the print_time.c
- Copy files print_time.c and print_time.h in your folder along with hello.c
- In a way, what we did before was to include the print_time.c function as a static library directly in the hello binary
- Now, we will do it explicitly and create a static library

CMake

- Add the following lines to your `CMakeLists.txt`

```
# We add the file as a static library called TimeLibrary
add_library(TimeLibrary STATIC print_time.c)
# This tells CMake to link greeting with the TimeLibrary
target_link_libraries(greeting TimeLibrary)
```

- after running `cmake` and `make` you should see `libTimeLibrary.a` in your `build` folder

Exercize

Try moving your `greeting` binary to another folder and running it. Does it need to see `libTimeLibrary.a`? Add `TimeLibrary` as a shared library in your `CMakeLists.txt` and `cmake` and `make` your project again. Move your `greeting` binary to another folder and run it. Now cut/copy `libTimeLibrary.so` to the same folder as `greeting` binary. What happens? Can `greeting` see the library now? (see here for help)

Homework

In this assignment you will build a face detector using the OpenCV library. Unpack `homework6.zip` which contains the source code and parameters files for this task. Your job is to write the `CMakeLists.txt` and compile the program with CMake.

Assignments

1. Check if you have under ROS the `objdetect`, `highgui` and `imgproc` OpenCV libraries installed (required for the program). In which folder are these shared libraries situated?
2. In the same folder as `objecDetection.cpp` copy a `.jpg` image with a human face (detection does not work well with animals, we already tried it!)
3. Write the `CMakeLists.txt` file (hint: look for some ROS projects that used OpenCV and see how they have setup the file)

Homework (continued)

Assignments

5. Build the program with CMake (do not forget about the build folder)
6. If a face was detected program will create `face_detection.jpg` image with the detected face and eyes
7. Send us the created `CMakeLists.txt` file and the image with the detection results