Libraries and Compilation Environment (II)

Computadors – Grau en Ciència i Enginyeria de Dades – 2019-2020 Q2

Facultat d'Informàtica de Barcelona

Compilers are the tools we use to generate binary executables, as well as other code files like object files and libraries. Besides, executables can be diferent depending on the compiler optimizations. We attach some source codes to perform the following exercises.

In this lab session we will learn how to create libraries (as well as extract object files from libraries), and dynamically and statically link executables. Finally, we will see a brief exemple of the difference between compiled program and an interpreted program.

Compiler command line options

Compilers usually have a large number of command line options. Let's see the ones that are most useful in the GCC C/C++ compilers:

Getting help

```
--help                  # simple help

--help  --verbose       # help of the compiler driver and sub-processes. Most useful!!
```

Common options

```
-S      # generate assembly only

-c      # generate object file(s) only

-o <name>       # name the output file
```

Code generation options *(https://eli.thegreenplace.net/2012/01/03/understanding-the-x64-code-models)*

```
-fpic           # generate position independent code – small mode. Needed to build shared libraries

-fPIC           # generate position independent code – large mode. For shared libraries
```

Optimization options *(https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html)*

```
-O              # equivalent to  -O1

-O0             # do not optimize. Sometimes useful for debugging

-O1             # optimization level 1: basic optimizations that do not take compilation time

                #    merges constants, moves loop invariants…

-O2             # adds some more expensive optimizations:

                #    code alignment, partial inlining, switch conversion, store merging…

-O3             # adds even more expensive optimizations:

                #    inline functions, loop vectorize, loop distribution, loop interchange…
```

Debug support options

```
-g      # generates debug information, useful for gdb, ddd, etc.
```

Linking options

```
-L <path>       # adds path to the list of directories where to find libraries for linking
-l<name>        # adds lib<name>.so for shared linking, and/or lib<name>.a for static linking
-shared         # generates a shared library, instead of a binary executable
-static         # generates a statically linked binary executable
```

Compiling programs and libraries

In the attached codes (FileS6.tar.gz) we have developed two source codes, named *maths.cc* and *inout.cc* with their respective header files. Additionally, we include a main source code called *program.cc* that uses functions of both files.

Exercise 1

Create a new file called "answers.txt" and write the command lines you have to use to build the object files for maths.cc and inout.cc. Finally, write the command line to create the executable ***progex1*** *in which you compile program.cc in conjunction with the other two object files.*

Now we are gonna create a static library (with the extension ".a"). This library will comprise the object files of both maths.cc and inout.cc. To do this you have to use the "ar" command. In this particular case you have to execute the following command line:

```
#>ar –csr libCOMstatic.a maths.o inout.o
```

You can use this "ar" command to extract object files from a given library file. We suggest you read the "-x" and "-p" options of "ar" in the man.

This command line will create the static library library libCOMstatic.a. The flags indicate: (c) create the library; (s) create an index of the files inside the library; and (r) to add files to the library. In this particular case, we reuse the object files created in the previous exercise. Now let's create a program with this library instead of referencing the object files.

Execute "g++ -o **progex2** program.cc –lCOMstatic" or "g++ -o **progex2** program.cc libCOMstatic.a"

Exercise 2

Compare both progex1 and progex2 with "ls –l" and the commands of the previous session. Are they equal/similar or different? Write your findings in the "answers.txt" file.

Now we are gonna create a dynamic library (with the extension ".so"). This library will comprise the object files of both maths.cc and inout.cc.

Exercise 3

To do this you have to use the "g++" compiler with the following steps that you have to write down in your "answers.txt" file:

1) *Create again the object files of maths.cc and inout.cc, but adding the "-fpic" flag.*
2) *Execute the command line "g++ -o libCOMdyn.so –shared maths.o inout.o"*
3) *Compile a new version of the program, called **progex3**, but using the new created library. In this case, you have to introduce the flag "-L$PWD" to indicate the compiler the current working directory is the path to find the library. Remember to also introduce the flag "-lCOMdyn" to link this particular library.*

To double check the shared objects the program depends on, execute the command "ldd" (NOTE: read the man of this command to find out how to use it). That is, the libraries that are dynamically linked at run-time. We suggest you to use the "-v" flag.

Exercise 4

Write in the "answers.txt" file the output of the "ldd" command for progex2 and progrex3. Briefly explain which binary depends on one of our libraries.

If you try to run progex3, it would not work. The problem is the environment variable "LD_LIBRAY_PATH" does not include the path of our library. To solve this issue, you have to update this environment variable by adding the "$PWD" path into this variable (NOTE: check the notes of the first lab session). Afterwards, execute again the binary and it should work.

Statically vs Dynamically linked

Compile again the three binaries, but adding the "-static" flag in the compilation command line, and add the suffix "static" at the end of every new binary.

Exercise 5

Write in the "answers.txt" file the difference of the static versus non-static linked binary. Focus the comparison on: file size (using "ls –l") and the output of the symbol table (NOTE: check the Lab session 5).

Optimization Flags

Compile again the three binaries, but without using our library. In fact, compile again all the source code files as follows:

g++ -o progOx -Ox program.cc maths.cc inout.cc

in such a way, "progOx" represents the binary compiled with the optimization flag "-Ox". That is, -O0, -O1, -O2, -O3. Afterwards, use the "objdump" command, with the option "-d" to extract the assembly code from the program. Do it for every single program created in this Section. Redirect the output to "outOx.asm" files, where "x" can be 0, 1, 2, 3.

Exercise 6

Write in the "answers.txt" file the difference in size of the binaries (using "ls –l") and the output of the "objdump –d" command. In particular, look for the implementation of "_Z8mathloopiii" function (the function of the maths.cc file) and compare the output of the different assembly files.

Interpreted Programs

In this Section we are gonna deal with programs that are interpreted. We suggest you to double check the slides to fully understand the differences between compiled and interpreted programs. In this case we are gonna deal with the interpreted programming language Python, commonly used for Big Data, AI, machine learning. You may find a tutorial in https://docs.python.org/3/tutorial

We attach a file called "hello.py". This file comprises file few code lines that shows a message to the screen, reads an integer from the keyboard and shows it afterwards.

To execute this program you have to use the following command line:
#> python ./hello.py

In this case, we are calling the interpreter "python" to perform the AOT + JIT steps, as shown in the slides.

Exercise 7

Modify the "hello.py" file and include the line "#!/usr/bin/python" in the first line of that file. Modify the permissions of the file and grant execution permissions (NOTE: check the chmod command of the first Lab session). And execute the file as any other program "./hello.py". The added line indicates what interpreter has to read and compile at run-time (interpret) the code. Write down, in the "answers.txt" file, what are the key differences between the binary files compiled in the previous exercices and this interpret code file (NOTE: compare it with the slide of compiler vs interpreter).

Upload the Deliverable

When done with the lab session, to save the changes you can use the tar command as follows:

```
#tar  czvf  session6.tar.gz   answers.txt
```

Now go to RACO and upload this recently created file to the corresponding session slot.