Computational Physics

Compiling and linking

multimodule software

Fernando Barao, Phys Department IST (Lisbon)

Multimodule software

- Usually the software we develop is divided in multiple source files or modules

  easier to manage and maintain (edit, correct, compile, test, debug)

- A modular structure also allows recompilation of only those source files that have been modified, rather than the entire software system

- Disadvantages:
  - you need to know the interdependencies between files
  - compilation line risks to be very long

    a simple shell script can be created to have the compilation command...or a Makefile!

C++ Compiling chain

- The C++ program (source code) consists in a set of symbolic instructions and data *(test.C)*
- The language compiler *(C++ → g++)* produces the object code *(test.o)* which is a translation of the source code into the machine language that can be understood by the CPU
  the compiler assigns memory addresses to variables and translates arithmetic and logical operations into machine-language instructions
  > g++ -c test.C
- Thirdly, the object code *(test.o)* is linked with other codes installed or with system binary libraries called by the user code, producing the executable *(test.exe)*
  > g++ -o test.exe test.C
- Additional flags can be used in the compiler process:
  -g - turn on debugging (so GDB gives more friendly output)
  -Wall - turns on most warnings
  -O or -O2 - turn on optimizations
  -o <name> - name of the output file
  -c - output an object file (.o)
  -I<include path> - specify an include directory
  -L<library path> - specify a lib directory
  -l<library> - link with library lib<library>.a

C++ libraries

A library is a collection of object files grouped together into a single file and indexed.

Libraries are used in the compiler with an argument of the form **-l library-name** .

The directory where they are is provided through **-L library-dir** .

The libraries must be listed in the g++ command after the object or source files that contain calls to the functions they include.

- **static libraries**

  The object code included in this kind of libraries is included in the executable through the compilation process. The advantage is that the built executable is portable and autonomous.

```
g++ -I <include_dir> -c Tpol.C # compile C++ code
ar ruv lib/libPOL.a Tpol.o # making the static library
ranlib lib/libPOL.a # make symbol table
```

C++ libraries (cont.)

✔ shareable libraries

The dynamic libraries (.so) include C++ code that is not included in the executable when the linking process happens. One gets a smaller executable! The library is loaded in memory and its location is provided to the executable though the environment variable
LINUX: **LD_LIBRARY_PATH**
MacOS: **DYLD_LIBRARY_PATH**

```
setenv LD_LIBRARY_PATH <library-dir>:<library-dir> #cshell
export LD_LIBRARY_PATH=<library-dir>:<library-dir> #bash
```

The option -fPIC (Position Independent Code) tells the compiler to compile without specifying memory addresses.

```
g++ -I <include_dir> -fpic -c Tpol.C # compile C++ code with position indep code
g++ -shared -o lib/libPOL.so TPol.o # created shared library
```

C++ libraries (cont.)

✔ using the libraries

The directory where the library **lib***POL.so* or **lib***POL.a* is provided using the **-L** option and the name of the library is provided by the **-l** option without the prefix **lib**.

The search is made first for the .so and after the .a library

```
# default: shareable lib used
g++ -o Readpol.exe Readpol.C -L lib/ -l POL
# pass option to linker to force use of static lib
g++ -o Readpol.exe Readpol.C -Wl,-Bstatic -L lib/ -l POL
# pass option to linker to force use of dynamic lib
g++ -o Readpol.exe Readpol.C -Wl,-Bdynamic -L lib/ -l POL
```

✔ check what symbols is included in the library

```
nm -C lib/libAddNumbers.a

nm lib/libAddNumbers.a | c++filt
```

✔ check which shareable libraries are used by the executable

```
ldd <executable>
```