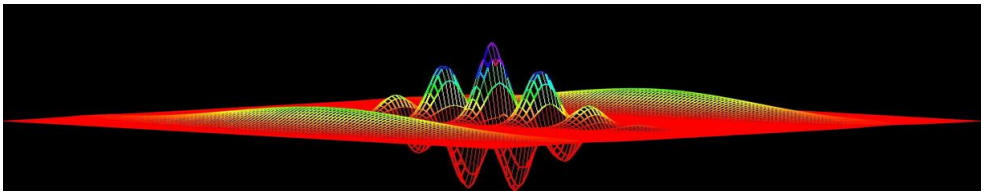
Computational Physics

numerical methods with C++ (and UNIX)



Fernando Barao
Instituto Superior Tecnico, Dep. Fisica
email: barao@lip.pt

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 **LD_LIBRARY_PATH**

```
setenv LD_LIBRARY_PATH <library-dir>
```

  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 th eprefix **lib**. The search is made first for the **.so** and after the **.a** library.

```
g++ -o Readpol.exe Readpol.C -L lib/ -l POL
g++ -o Readpol.exe Readpol.C -Wl,-Bstatic -L lib/ -l POL #pass option to linker t
g++ -o Readpol.exe Readpol.C -Wl,-Bstatic -L lib/ -l POL #pass option to linker t
```

- **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>
```


C++ Makefiles

- UNIX/Linux systems have a powerful tool called **make** that allows you to manage compilation and linking of multiple modules into an executable
- The **make** utility reads a specification file, **makefile**, that describes how the modules of a software system depend on each other
- The several steps to compile a program including also some other actions like directory cleaning can be grouped in this special file *Makefile* or *makefile*. To run it :

```
> make <tag>
```

- **make** options :
  -d display debug information
  -f filename specifies a different name for the specification file
  -h display a brief description of all options
  -n do not run any makefile command, just display them
  -s run in silent mode

C++ Makefiles (cont.)

- The Makefile is organized in blocks with a *target-list*, *dependency-list* and *commands*.
  - Every action is preceded of spaces generated by a **TAB**.
  - A long line can be continued by terminatingit with a backsplash (\) character
  - add comment by using # - we can use shell's filename pattern characters (?, *)

```
# block example
target-list: dependency-list
<TAB>    command-list
```

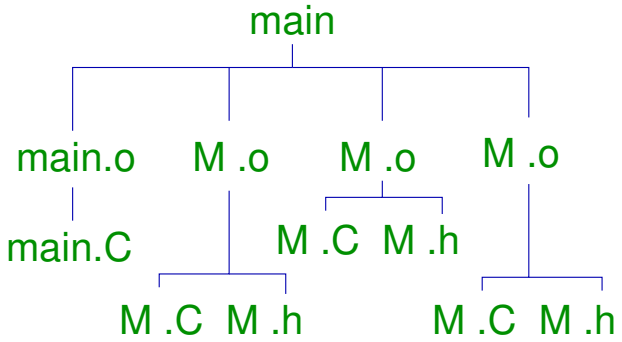

C++ Makefiles (cont.)

- Suppose we have a main program **main.C** and list of C++ modules called **M1.C, M2.C, M3.C**
- The compilation of **main.C** will depend on the compilation of the modules

```
main: main.o M1.o M2.o M3.o
        g++ -o main.exe main.o M1.o M2.o M3.o
```

- ...but we need to produce the object code of the modules and text.C

```
main.o: main.C
        g++ -c main.C
M1.o: M1.C M1.h
        g++ -c M1.C
M2.o: M2.C M2.h
        g++ -c M2.C
M3.o: M3.C M3.h
        g++ -c M3.C
```



C++ Makefiles (cont.)

- To remove all object files of current directory

```
clean:
        rm -f *.o
```

- **Suffix default rules**

  The make rules shown precedingly contain some redundant commands

  The make utility has many predefined make rules, also known as *suffix rules*, that allow the make utility to perform automatic tasks

```
test: test.o M1.o M2.o M3.o
        g++ -o test.exe test.o M1.o M2.o M3.o

M1.o: M1.C M1.h
M2.o: M2.C M2.h
M3.o: M3.C M3.h

.C.o:
        g++ -fPIC -Wall -g -c $<
```


C++ Makefiles (cont.)

- **MACROS**

  The *make* utility supports simple macros that allow text substitution (define before using them)

- **user defined macros**

```
1. macro_name = text
2. define macro_name
   text
   endef
```

- to use the macros : **$ (macro_name)**

- **built-in macros**

  $@ name of current target

  $? list of dependencies that have changed more recently than target

  $< the name of the current dependency that has been modified more rece

  $^ a space-separated list of all dependencies without duplications

C++ Makefiles (cont.)

```
# user macros
define CC
  g++
endef
CFLAGS = -Wall -g -fPIC
LDFLAGS = -lm
OBJS = main.o M1.o M2.o M3.o
SRCS = main.C M1.C M2.C M3.C
HEAS = main.h M1.h M2.h M3.h
# actions
build: test.exe
       @echo building $?
test.exe: $(OBJS)
       $(CC) $(CFLAGS) -o $@ $^
       @echo compiling $?
test.o: $(SRCS) $(HEAS)
.C.o:
        $(CC) $(CFLAGS) -c $<
clean:
        rm -f *.o
```