Building SW Packages Overview: make & cmake

Latin American Introductory School on Parallel Programming and Parallel Architecture for HPC

J. Manuel Solano-Altamirano
Facultad de Ciencias Químicas
Benemérita Universidad Autónoma de Puebla

Code design

- Frequently, the software is a single large program.

```
1  //reading the input
2  ...
3  //processing the input
4  ...
5  //writing the output
6  ...
```

- OK for small projects, but not if the program becomes large, or too complex.
- Break it down into small chunks.
- Each piece should have defined tasks.

Code design

```
1  class ReaderClass {
2  ...
3  };
4  class DataClass {
5  ...
6  };
7  ...
8  int main() {
9     ReaderClass reader;
10    reader.read(...,data,...);
11
12    DataClass data;
13    OperationsClass ops;
14    ops.Function1(data);
15    ops.ProcessA(data);
16
17    WriterClass writer;
18    writer.SaveFile(...,data,...);
19    return EXIT_SUCCESS;
20 }
```

- Easier to identify problems.
- Improves code reusability.

Compilation

- Compilation of a single program

```
1 $g++ myprogram.cpp -o myprogram -O3
```

- Split the code: main.cpp, readerclass.h/cpp, dataclass.h/cpp

```
1 $g++ myprogram.cpp readerclass.cpp dataclass.cpp ... -o myprogram -O3
```

- Split the compilation as well

```
1 $g++ myprogram.cpp -O3 -c -o myprogram.o
2 $g++ readerclass.cpp -O3 -c -o readerclass.o
3 $g++ dataclass.cpp -O3 -c -o dataclass.o
4 $ ...
5 $ld myprogram.o readerclas.o dataclass.o ... -o myprogram
```

- If you change one class, then there is no need to recompile the others*.

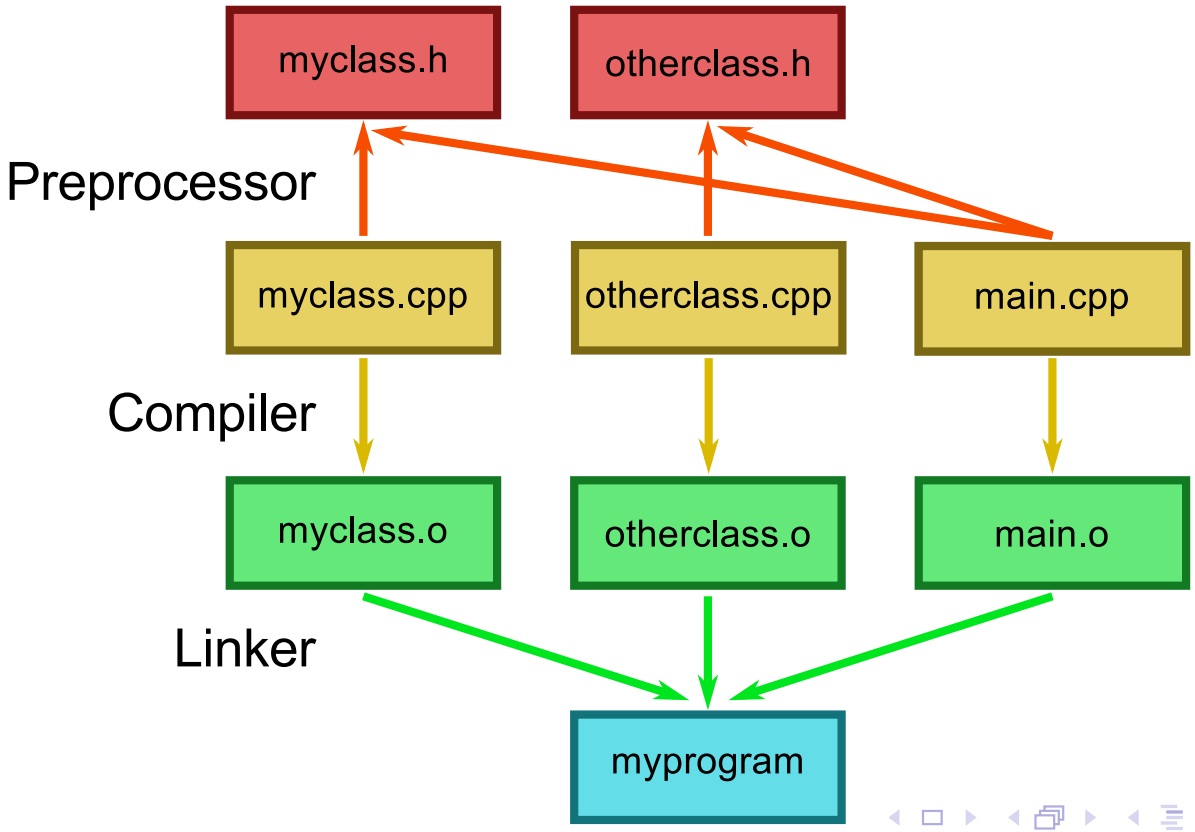
Code design

- Main program looks like this:

```
1  ...
2  #include "readerclass.h"
3  #include "dataclassclass.h"
4  ...
5  int main() {
6     ReaderClass reader;
7     reader.read(...,data,...);
8
9     DataClass data;
10    OperationsClass ops;
11    ops.Function1(data);
12    ops.ProcessA(data);
13
14    WriterClass writer;
15    writer.SaveFile(...,data,...);
16    return EXIT_SUCCESS;
17 }
```

What make/cmake is all about

- Simplify building code projects
- Speed up re-compilation after small changes



What make/cmake is all about

- Consistent build command: make

```
$make
```

- Consistent build command: cmake

```
$mkdir build
$cd build
$cmake ../
$make #or any building tool
```

- Variable definitions (platform/compiler specific configuration)

```
VARIABLE_NAME = value
```

The Makefile

- The magic is requested through a `Makefile`

```
1  target1: dep1
2  [tab]commands1 args1
3
4  target2: dep2a dep2b ...
5  [tab]commands2 args2
6
7    .
8    .
9    .
```

- But the Makefile can be changed

```
1  $make -f MyCustomFile
```

Makefiles syntax

- Targets: what is wanted to be built
- Dependencies: what is needed to build the targets
- Commands: what is executed to build the targets, including options.

```
1  target: dependencies
2  [tab] system commands
```

```
1  myclass.o: myclass.cpp myclass.h
2  [tab] g++ myclass.cpp -O3 -Wall -c -o myclass.o
```

- A target may become a dependency of another target.

How make builds targets

- The target is built if it does not exists.
- The target is re-built if any of the dependencies is newer than the target (*i.e.* a change occurred).
- make checks this for all targets.
- In large projects this saves huge amounts of time, as most of the changes occur in one source at once.
- (One line compilation):

```
1 g++ -o myprogram main.cpp myclass.cpp otherclass.cpp
```

Which compiles all cpp files every time.

Makefiles syntax

- Variables: you can declare variables. Useful for global changes and many other fancy stuff

```
1 CC=g++
2 CFLAGS=-O3 -Wall
3 myclass.o: myclass.cpp myclass.h
4 [tab] $(CC) myclass.cpp $(CFLAGS) -c -o myclass.o
```

- Comments: obviously, you can insert comments.

```
1 CC=g++ #Change to c++, clang if available
2 #set your compiler flags
3 CFLAGS=-O3 -Wall
4 myclass.o: myclass.cpp myclass.h
5 [tab] $(CC) myclass.cpp $(CFLAGS) -c -o myclass.o
```

- Recursive make: make can also call make.

```
1 MY_DIR=/a/custom/path
2 some_target: dependencies
3 [tab] cd $(MY_DIR); make
```

Makefiles syntax

- Automatic variables "$<", "$@", ...

```
1  myclass.o: myclass.cpp myclass.h
2  [tab] $(CC) $(CCFLAGS) -c $< -o $@
```

Here "$<" expands to "`myclass.o`" (the target), and "$@" expands to "`myclass.cpp`" (the first dependency)

```
1  myprogram: object1.o object2.o object3.o
2  [tab] $(CC) $(LDFLAGS) -o $@ $^
```

But "$^" expands to "`object1.o object2.o object3.o`"

Makefiles syntax

- Rules: useful if one is using the same treatment for all files of the same type.

```
1  %.o: %.cpp
2  [tab] $(CC) $(CCFLAGS) -c $< -o $@
```

This will apply the command $(CC) to every *.cpp file and it will produce a *.o file, using the same rule.

Special targets

- `.PHONY`. A phony target is one that is not really the name of a file; rather it is just a name for a recipe to be executed when you make an explicit request.
- clean
- all

```
1  .PHONY: clean
2  clean:
3  [tab] rm *.o temp
```

Thus, if a file named "clean" exists, `$make clean` will still be executed.

Libraries

- Libraries must be setup:
  - In source files:

```
1 #include <fftw3.h>
```

  - During compilation (the compiler needs to know where to look for fftw3.h):

```
1 $g++ ... -I/path/to/fftw3/include ...
```

  - At linking (the compiler needs to know where to look for the object file and the name of the library, which usually is an lib*.a or an lib*.so file):

```
1 $g++ ... -L/path/to/fftw3/lib -lfftw3 ...
```

Libraries

- Convention: the functions of the library "LLLLL" (*i.e.* the binary code) are saved into archive file(s) named "libLLLLL.a" (static) and/or "libLLLLL.so" (shared object).
- In makefiles, one may use variables:

```
1  INCLUDES= -I/path/to/fftw3/include -I/path/to/custom/include
2  LIBRARYPATHS=-L/path/to/fftw3/lib -L/path/to/custom/library
3  LIBRARIES= -lcustom -lfftw3
4
5  main.o: main.cpp
6  [tab] $(CC) $(INCLUDES) $(CFLAGS) -o $@ -c $<
7
8  myprogram: main.o object1.o object2.o object3.o
9  [tab] $(CC) $(LIBRARYPATHS) $(LIBRARIES) -o $@ $^
```

Make calling options

- Parallel compilation (large SW packages)

```
1 #calling make with 2 processors.
2 make -j 2
```

- Override variables

```
1 CC := g++
2 myprogram: main.o myclass.o
3 [tab]$(CC) -o $@ $+
```

```
1 $make
2 g++ -o myprogram main.o myclass.o
```

```
1 $make CC=icpc
2 icpc -o myprogram main.o myclass.o
```

Cross-platform

- Variables easy the compilation with different compilers

```
1  CC := gcc #this can be overriden
2  libs = -lfftw3 # common library
3  libs_for_gcc = -lgnu
4  normal_libs =
5
6  ifeq ($(CC),gcc)
7          libs += $(libs_for_gcc) #Adds gnu library to 'libs'!
8  else
9          libs += $(normal_libs)
10 endif
11
12 myprogram: $(objs)
13 [tab] $(CC) $(lib_paths) $(libs) -o $@ $^
```

Mixing languages

- Compilation using different languages

```
1  FFLAGS = #fortran flags (includes)
2  CFLAGS = #c flags (includes)
3  LDFLAGS = #linker options (paths and libraries)
4
5  fortrancode.o: fortrancode.f90
6  [tab] gfortran $(FFLAGS) -o $@ -c $<
7
8  ccode.o: ccode.f90
9  [tab] gcc $(FFLAGS) -o $@ -c $<
10
11 mixedprogram: fortrancode.o ccode.o
12 [tab] gcc $(LDFLAGS) -o $@ $^
```

Any target/dependency sequence

- Latex projects

```
1  chapters = chapter1.tex chapter2.tex
2
3  main.pdf: main.tex $(chapters) image1.pdf
4  [tab] pdflatex $@
5
6  image1.pdf: image1.svg
7  [tab] inkscape --without-gui --file=$< --export-pdf=$@
```

Debug/release versions

- Again, variables can do the trick

```
1  CFLAGS = -Wall -std=c++11
2  DEBUG := 0
3
4  ifeq ($(DEBUG),0)
5          CFLAGS += -O3 #optimization flags, standard flags
6  else
7          CFLAGS += -g
8  endif
9
10 myclass.o: myclass.cpp
11 [tab] $(CC) $(CFLAGS) -o $@ $<
```

```
1  $make DEBUG=1
```

Serial/parallel version

- Yet again, variables can help:

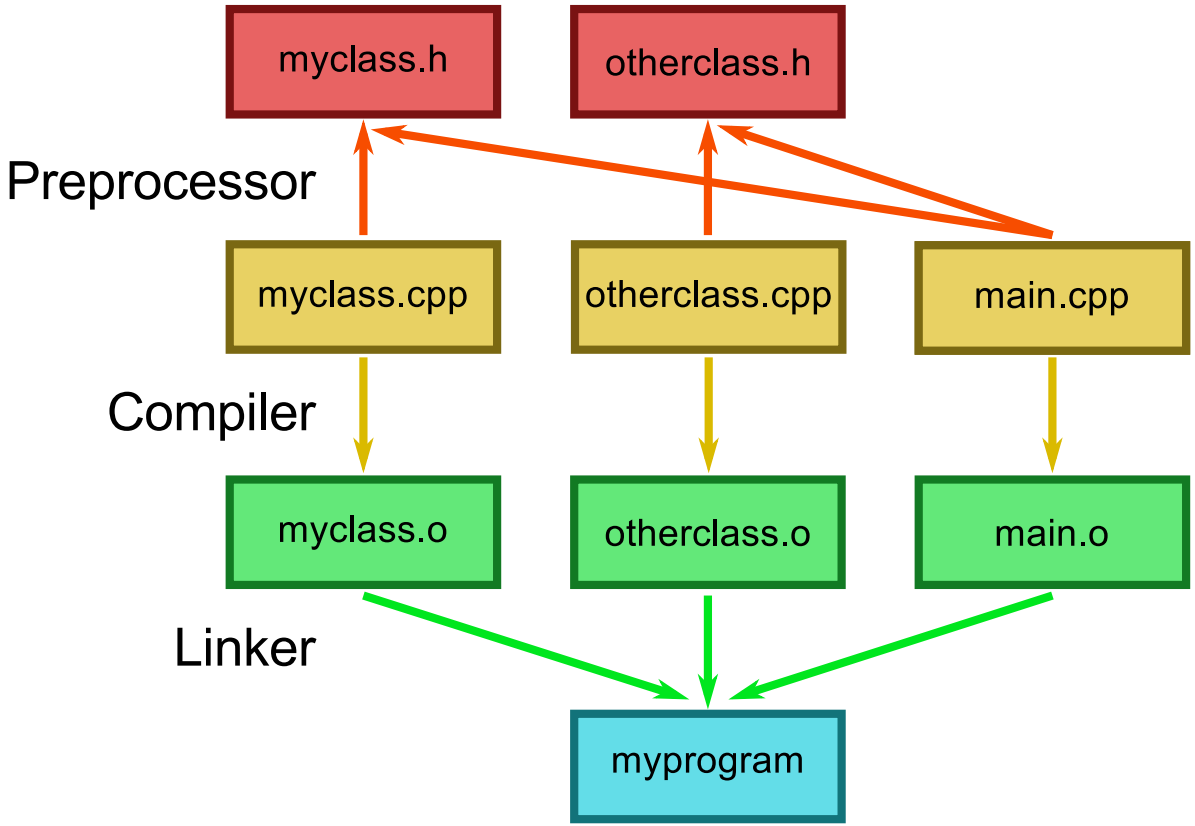
```
1  CC      := g++
2  CFLAGS = -Wall
3  LDFLAGS =
4  PARALLELMPI := 0
5
6  ifeq ($(PARALLELMPI),1)
7    CC = mpic++
8  endif
```

```
1  $make PARALLELMPI=1
```

- Or just override the compiler when executing make (do not forget to load modules if needed).

```
1  module load openmpi/1.10.7/intel/16.0
2  make CC=mpic++
```

Example


myclass.h
otherclass.h
Preprocessor
myclass.cpp
otherclass.cpp
main.cpp
Compiler
myclass.o
otherclass.o
main.o
Linker
myprogram

Example (direct)

```
1  myprogram: myclass.o otherclass.o main.o
2  [tab] g++ main.o myclass.o otherclass.o -o myprogram
3
4  myclass.o: myclass.cpp myclass.h
5  [tab] g++ -c myclass.cpp -o myclass.o
6
7  otherclass.o: otherclass.cpp otherclass.h
8  [tab] g++ -c otherclass.cpp -o otherclass.o
9
10 main.o: main.cpp
11 [tab] g++ -c main.cpp -o main.o
12
13 .PHONY: clean
14 clean:
15 [tab] rm *.o myprogram
```

Example (with (automatic) variables)

```
1  CC := g++
2  CFLAGS = -O3 -Wall
3
4  objects = myclass.o otherclass.o main.o
5  executable = myprogram
6
7  $(executable): $(objects)
8  [tab] $(CC) -o $@ $^
9
10 myclass.o: myclass.cpp myclass.h
11 [tab] $(CC) -o $@ -c $<
12
13 otherclass.o: otherclass.cpp otherclass.h
14 [tab] $(CC) -o $@ -c $<
15
16 main.o: main.cpp
17 [tab] $(CC) -o $@ -c $<
18
19 .PHONY: clean
20 clean:
21 [tab] rm $(objects) $(executable)
```

Example (with (automatic) variables and rules)

```
1  CC := g++
2  CFLAGS = -O3 -Wall
3
4  objects = myclass.o otherclass.o main.o
5  executable = myprogram
6
7  $(executable): $(objects)
8  [tab] $(CC) -o $@ $^
9
10 .cpp.o:
11     $(CC) $(CFLAGS) -c $< -o $@
12
13 .PHONY: clean
14 clean:
15 [tab] rm $(objects) $(executable)
```

Cmake

- Cross platform compilation
- Cmake creates makefile's (linux/MacOSX), xcodeproj's (MacOSX) or vcxproj's (Windows), which are suitable for building software on different OS's.

Cmake

- The magic file: `CMakeLists.txt`

```
1  cmake_minimum_required (VERSION 3.0)
2  add_executable(myprogram
3     main.cpp
4     myclass.cpp
5     otherclass.cpp
6     )
```

```
1  $mkdir build
2  $cd build
3  $cmake .. #CMakeLists.txt is in ../
4  $make
```

```
1  $mkdir build
2  $cd build
3  $cmake -G Xcode ..
4  $xcodebuild
```

Cmake

- Variables are defined with `set(...)`

```
1  cmake_minimum_required (VERSION 3.0)
2  set(top_path .)
3  add_executable(myprogram
4     ${top_path}/main.cpp
5     ${top_path}/myclass.cpp
6     ${top_path}/otherclass.cpp
7     )
```

- Many packages are cmake-friendly:

```
1     find_package(Qt5OpenGL REQUIRED)
2     find_package(Armadillo REQUIRED)
```

Adding the fftw3 library

```
1  INCLUDE_DIRECTORIES(/path/to/fftw3/include)
2  LINK_DIRECTORIES(/path/to/fftw3/lib)
3
4  ADD_EXECUTABLE(myprogram path0/main.cpp myclass.cpp otherclass.cpp)
5
6  TARGET_LINK_LIBRARIES(myprogram fftw3)
```

```
1  $mkdir build
2  $cd build
3  $cmake .. #CMakeList.txt is in ../
4  $make
```

Libraries on different OS's

```
1
2  if (UNIX OR APPLE)
3          set(libname fftw3)
4          include_directories(/path/to/fftw3/include)
5          link_directories(/path/to/fftw3/lib)
6  endif (UNIX OR APPLE)
7
8  if (WIN32)
9          set(libname libfftw3.lib)
10         include_directories(C:\Windows\path\to\include)
11         link_directories(C:\Windows\path\to\library)
12 endif (WIN32)
13
14 add_executable(myprogram path0/main.cpp path1/object1.cpp path2/object2.cpp)
15
16 target_link_libraries(myprogram ${libname})
```

```
1  $mkdir build
2  $cd build
3  $cmake .. #CMakeList.txt is in ../
4  $make
```

Compiling with mpi

```
1  cmake_minimum_required (VERSION 3.0)
2  find_package(MPI REQUIRED)
3
4  include_directories(${MPI_INCLUDE_PATH})
5
6  set(SRC_DIR .)
7  add_executable(myprogram
8     ${SRC_DIR}/main.cpp
9     ${SRC_DIR}/myclass.cpp
10    ${SRC_DIR}/otherclass.cpp
11    )
12
13 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS}-Wall -pedantic")
14 set(CMAKE_CXX_FLAGS_RELEASE "${CMAKE_CXX_FLAGS} -O3")
15
16 if(MPI_COMPILE_FLAGS)
17   set_target_properties(myprogram PROPERTIES
18     COMPILE_FLAGS "${MPI_COMPILE_FLAGS}")
19 endif()
20
21 if(MPI_LINK_FLAGS)
22   set_target_properties(myprogram PROPERTIES
23     LINK_FLAGS "${MPI_LINK_FLAGS}")
24 endif()
```