Exercise 1: Programming Tools in Linux/UNIX

Sistemi Operativi ed in Tempo Reale
AA 2021/2022

SORT Exercises

- Programming tools in Linux/UNIX
- Strings, pointers, linked lists, TCP/IP sockets
- Exercises: Concurrent programming with message passing paradigm

Exercise 1

- Tar: file compression
- Pre-processor
- Compiling
- Linking
- Makefile
- CMake

Introduction

- Log in: account needed (username+password)
- Create a new directory

```
$ mkdir sisop
$ cd sisop
```

- Directory content

```
$ ls
```

- Copy file

```
$ cp ../examples.tar.gz .
```

- Text editors: gedit, vim, Xemacs, Kwrite ecc..

Command *tar*

- Software (including SORT material) is often in .tar format
- Archives may store multiple files
- Example: create archive with directory *dir1* and file f*ile2*

```
$ tar cf esempio.tar dir1 file2
```

- Extracting from foo.tar:

```
$ tar xvf foo.tar
```

- Vebose option -*v* for more output info
- To display the content foo.tar:

```
$ tar tf foo.tar
```

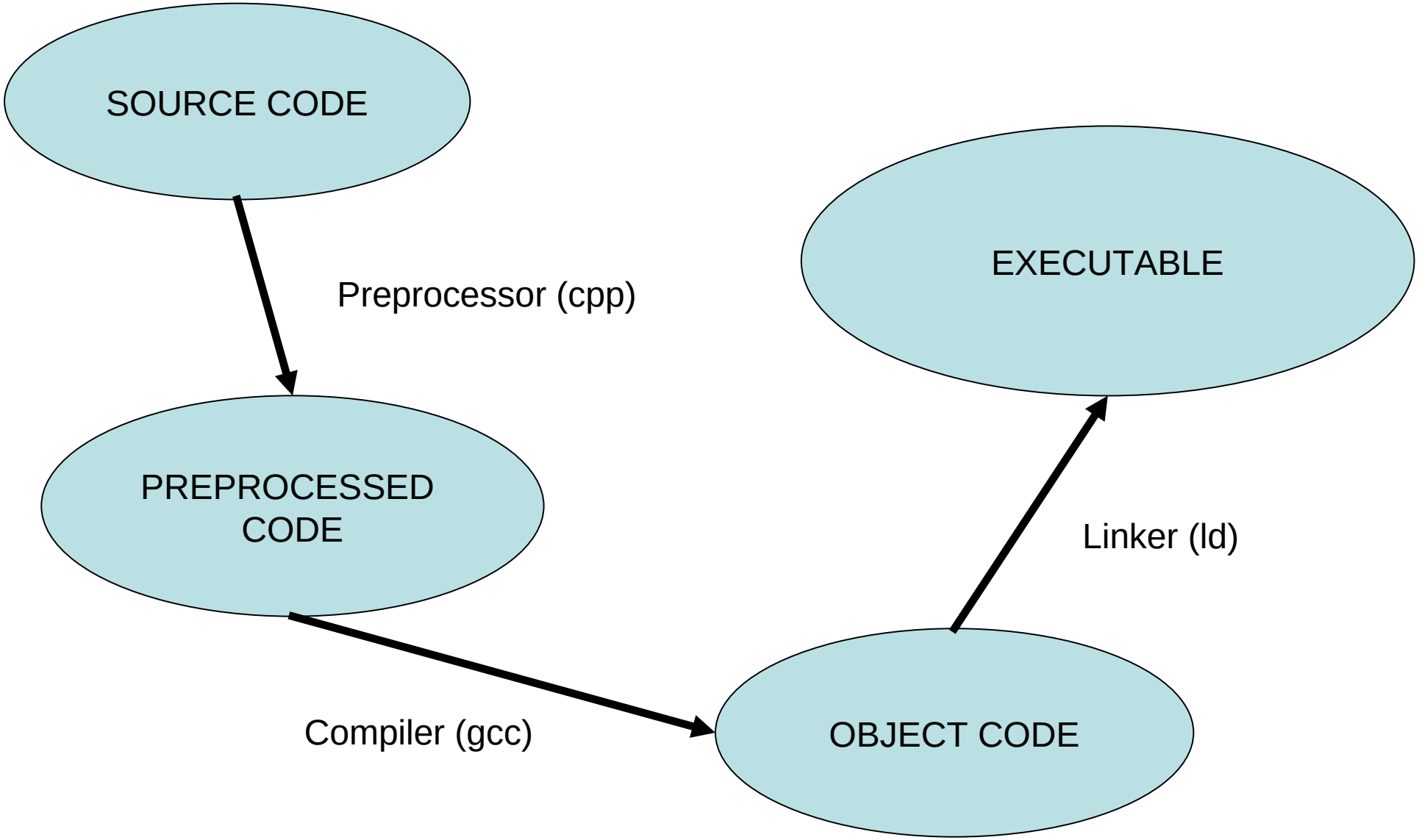
Command *tar*

- Archives can be compressed using *gzip*
- Extensions .tar.gz or .tgz used for compressed tar
- Expand archives using:

  $ tar xvf foo.tar

  $ tar xzvf foo.tgz

Compiling


SOURCE CODE
Preprocessor (cpp)
PREPROCESSED CODE
Compiler (gcc)
OBJECT CODE
Linker (ld)
EXECUTABLE

C Preprocessor

- Tool to transform code before compiling it
- Preprocessor searches and expands **directives**, special instructions in source code
- A **directive** starts with char **'#'**, consists of a single line (although it can continue to next line with '\') and has no terminal char
- Preprocessor creates a copy of original source code where each directive has been **substituted**
  - No binary code with preprocessor

C Preprocessor

- Examples of directives: **#include** , **#define** , **#undef** e **direttive condizionali**.

- Directive #define

  #define NAME expansion
  E.g.: #define MAX 10

- All instances of MAX substituted by string 10 in the code
- It allows definition of constants
- By convention macros are in capital letters

- Complex macros:
  #define identifier(arguments) expression
  E.g.: #define MIN(x,y) ((x<y)?(x):(y))
  #define SQUARE(x) x*x

- MIN(a,b) expended with: ((a<b)?(a):(b))

C Preprocessor

- Directive **#include** for including files in the code, usually .h header files
- **#include <filename>**

  #include "filename"
- Angle brackets **<>**: filename in default path of the project
- Quotation marks **"":** relative path from the directory where #include is called

C Preprocessor

- Conditional compiling: selection of lines to be compiled when some conditions are met

- **#ifdef NAME (#infdef NAME**)

  ...

  #endif

  insert the lines between the macros only if NAME is defined

  E.g.:

```
#define FOO
#ifdef FOO
    … this gets included…
#endif
#ifndef FOO
    … this does NOT get included…
#endif
```

C Preprocessor once more

- Gcc option *'-DMYMACRO'* for definition of macro *MYMACRO* in command line
  Example ('preproc_ex'):

```
#include <stdio.h>
int main (void){
    #ifdef TEST
        printf ("Test mode\n");
    #endif
    printf ("Running...\n");
    return 0;
}
```

- Message "Test mode" printed only when compiling wth command line '-DTEST'

```
$ gcc -Wall -DTEST dtest.c
$ ./a.out
```

- Without '-DTEST' the message "Test mode" is not printed

```
$ gcc -Wall dtest.c
$ ./a.out
```

C Preprocessor once more

- Also macro values can be defined by command line

```
#include <stdio.h>
int main (void) {
    printf("NUM equal to %d\n", NUM);
    return 0;
}
```

```
$ gcc -Wall -DNUM=100 dtestval.c
$ ./a.out
NUM equal to 100

$ gcc -Wall -DNUM="2+2" dtestval.c
$ ./a.out
NUM equal to 4
```

- When macro value is not defined (e.g. gcc -DNUM …) gcc uses default value 1

Compiler

- Compiling object code
  gcc <options> <filename>

  (examples/compilation/ex1)
  $ gcc -c hello.c

- gcc operates as preprocessor, compiler and linker depending on the command line options

- Options for strict check about the code (e.g. compatibility to ANSI):
  -Wall: shows all warnings;
  -pedantic: displays all errors and warnings required by ANSI C standard

- To optimize code:
  -O -O1 -O2 -O3: increasing levels of optimization
  -O0: no optimization

- Include debug options: -g

Linker

- Solves symbols among object files, links libraries and generates the executable

```
$ gcc -c hello.c
$ gcc hello.o -o hello
$ ./hello
```

- Example: linking other functions

  (examples/compilation/ex2)

```
$ gcc hello2.c -o hello2        [it may fail]
$ gcc hello2.c -lm -o hello2    [it works]
```

- Function *sqrt()* defined in library file /usr/lib/libm.a

Libraries

- Library: collection of precompiled object files ready to be linked to an executable
  - language or system standard libraries: glibc, math
  - user defined libraries

- To use library include its header file .h

- **Static library**:
  - Extension **.a** (“archive”) in Linux (.lib in Windows)
  - A copy of library is integrated in the executable (no dependency from .a)

- **Dynamic library**:
  - Extension **.so** (“shared object”) in Linux (.ddl in Windows)
  - Library code on external file
  - Avoid too large size of executables

Libraries

- **Dynamic linking**:
  - Executable linked to a shared/dynamic library file only contains a table with the symbols of functions
  - Linking to the code of the function before running the executable

- Saving space and program footprint: a single library copy is shared among multiple executables

- Shared libraries can be updated without recompiling (if the library interface does not change)

Linking once more

- **ldd**: command to show the list of shared libraries required by an executable

```
$ gcc -lm hello2.c
$ ldd a.out

linux-gate.so.1 =>  (0xb7f13000)
libm.so.6 => /lib/tls/libm.so.6 (0xb7eca000)
libc.so.6 => /lib/tls/libc.so.6 (0xb7db2000)
/lib/ld-linux.so.2 (0xb7f14000)
```

The above program depends on *libm* (version 6), C library (*libc*) and dynamic loader *ld*

Solving Paths

- Compiler requires to find where the file is located
  - Standard error with header file:

    FILE.h : No such file or directory

    the file is not in a standard directory checked by gcc
  - Similar issue for libraries:

    /usr/bin/ld: cannot find library

- Options **-I** and **-L** specify to compiler additional path where to search header or libraries
  - Syntax: -I*/path/to/header*, -L*/path/to/library*

```
$ gcc -Wall -I/opt/gdbm-1.8.3/include -L/opt/gdbm-1.8.3/lib dbmain.c -lgdbm
```

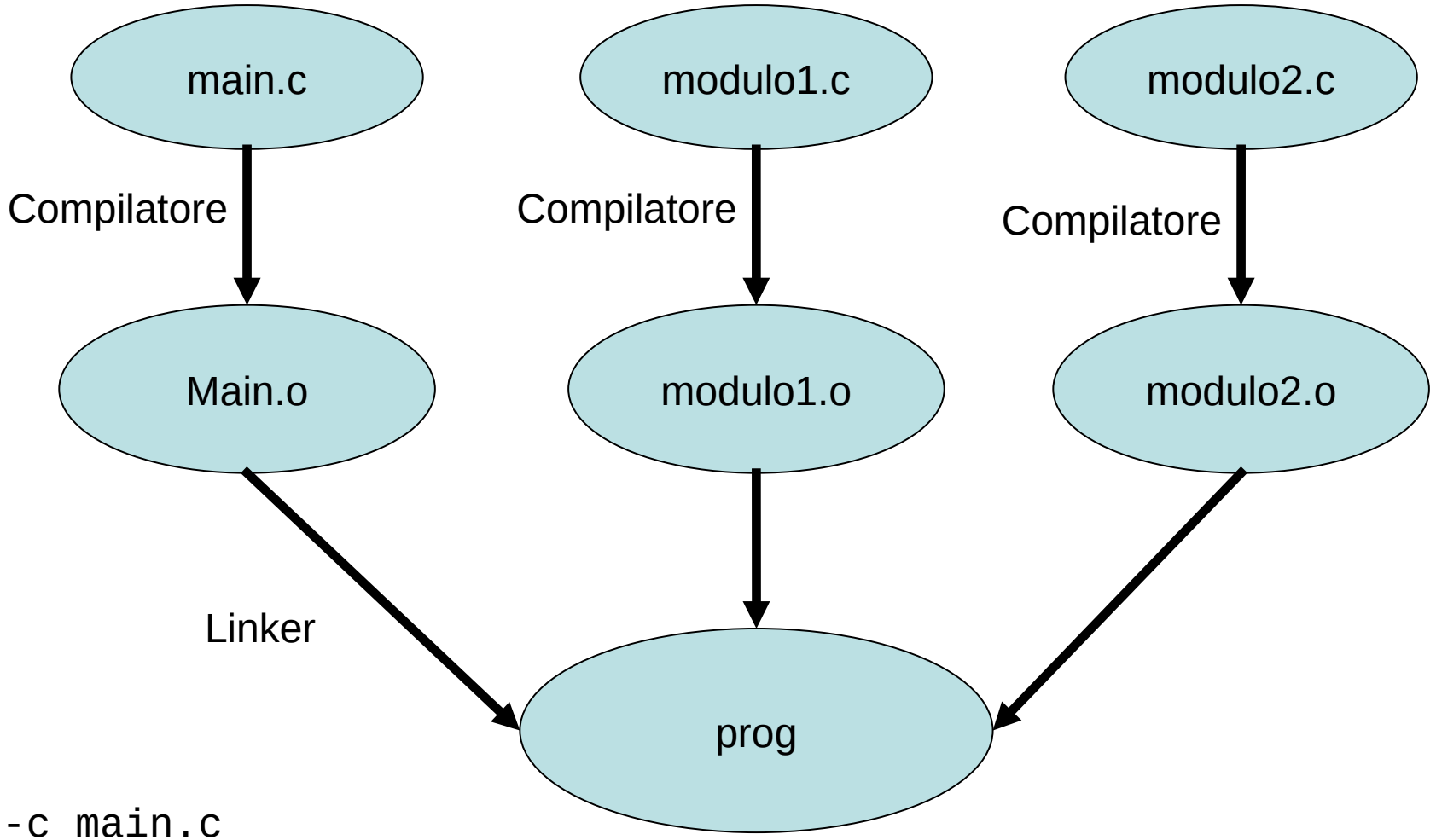
Function Prototypes

- Good practice: declare functions before using them (and before their definition)
- E.g. (examples/compilation/ex3):

```
#include <stdio.h>
/* Prototipo della funzione */
int multiply(int a, int b);

/* Definizione della funzione */
int multiply(int a, int b) {
  return(a*b);
}
```

- Avoid errors

Multi-file Programs




```
gcc -c main.c
gcc -c module1.c
gcc -c module2.c
gcc -o prog main.o module1.o module2.o
```

Multi-file Programs

- Examples:

(examples/compilation/ex4):

```
gcc -c main.c
gcc -c multiply.c
gcc main.o -o example [non funziona]
gcc main.o multiply.o -o example [funziona]
```

File header (.h)

Definition of function interfaces

- Option -Idir to give the compiler the path to header files

- Header contains:
  - prototypes of shared funtions
  - declaration of extern variables
  - typedefs
  - macros
  - structs, enums

File header (.h)

- Using macros to avoid recursive definion

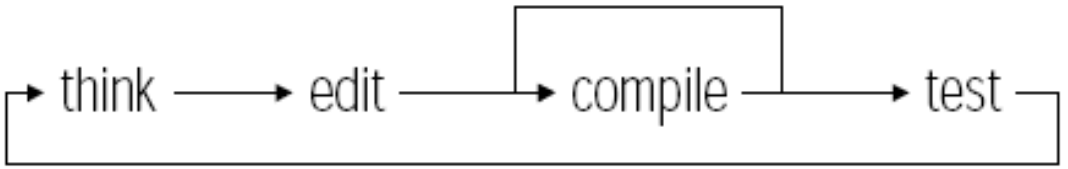
```
#ifndef FOO_H
#define FOO_H

… definition or inclusion of foo …

#endif
```

- Example (examples/compilation/ex5):

```
gcc -c main.c
gcc -c multiply.c
gcc main.o multiply.o -o example
```

Make

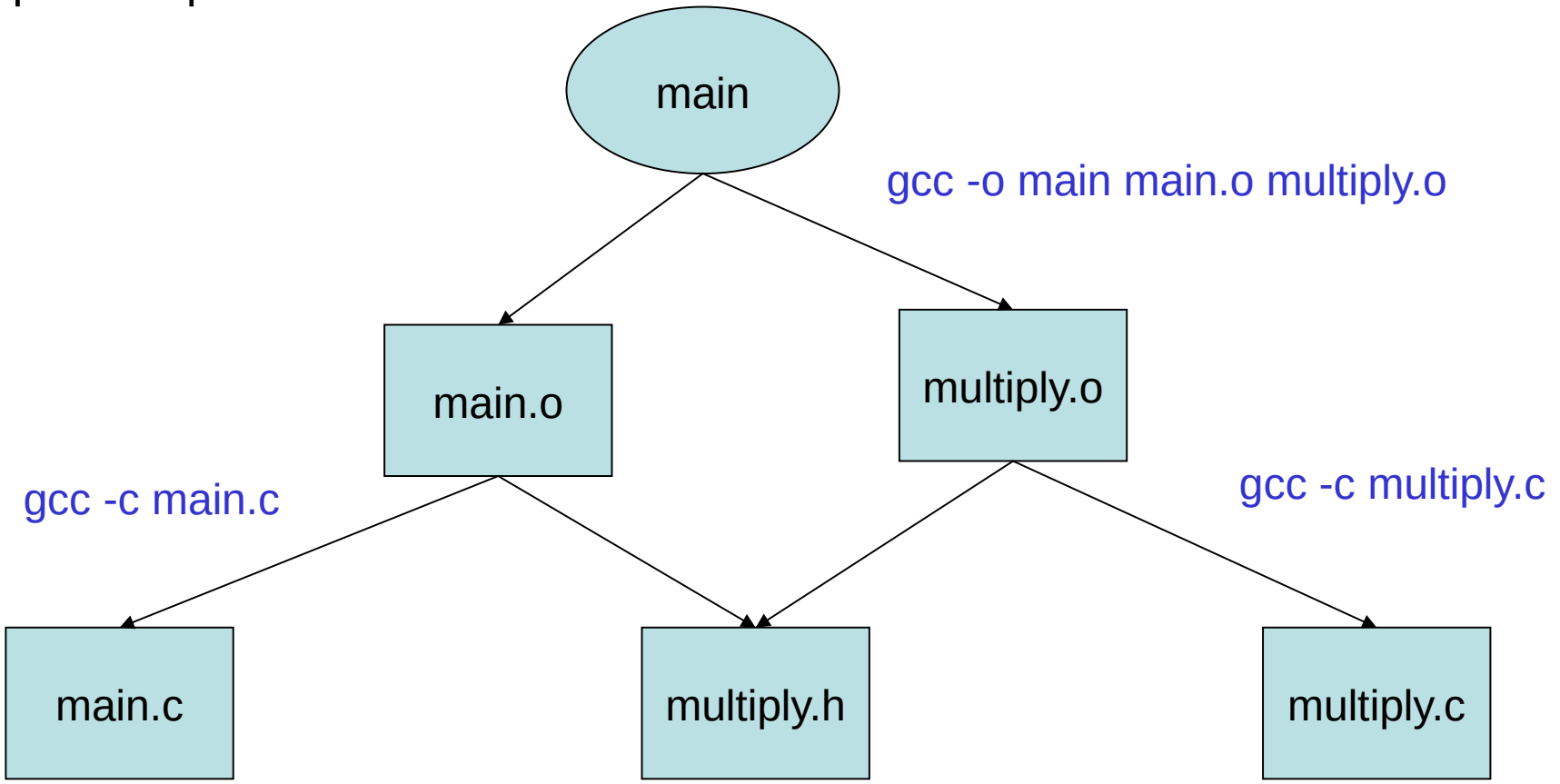
- Compiling multi-file project is tedious and error prone
- Develpment cycle of a program (repeated multiple times!)



- Issues:
  - you change a file and forget to recompile it
  - interface changes (.h), but you forget to compile all the files depending on it
- **Make**: automatic execution of compiling instructions

Make

Graph of dependencies



– each node is a file
– every node is associated to a command executed by make in bottom-up fashion

Makefile

Makefile:

- file that provide the dependency graph
- the commands associated to each node of the graph
- The operations

targets: dipendencies
[tab] comands

- Comands must start with a <tab>
- Example:

```
main: main.o multiply.o
    gcc -o main main.o multiply.o
main.o: main.c multiply.h
    gcc -c main.c
multiply.o: multiply.c multiply.h
    gcc -c multiply.c
```

Makefile

To run make: `make <target>`

```
make
make multiply.o
make main
```

- Without arguments it executes the first target in makefile:

Esempio (/examples/make/ex1)

```
$ make
gcc -c main.c
gcc -c multiply.c
gcc -o main main.o multiply.o
$ touch multiply.c
$ make
gcc -c multiply.c
gcc -o main main.o multiply.o
```

Makefile

- Make allows to define macros to handle generalizations and parameters in makefile

```
OBJECTS = data.o main.o io.o
CC=gcc
project1: $(OBJECTS)
    $(CC) -o project1 $(OBJECTS)
data.o: data.c data.h
    $(CC) -c data.c
main.o: data.h io.h main.c
    $(CC) -c main.c
io.o: io.h io.c
    $(CC) -c io.c
```

Dummy Targets

- Dummy targets for operation that are not stricly part of compiling

```
install: a.out
    cp a.out main

clean:
    rm *.o a.out main
```

`make clean` removes files “.o” and executable

- Dummy targets for management of project

```
clean install print
release submit test
```

Dynamic Macros in Make

- Make supports macros to automatize targets:
  - **$@** name of current target
  - **$?** list of outdated dependencies
  - **$<** name of first dependency
  - **$*** target name without suffix/extension
  - **$^** list of all the dependencies

Es (examples/compilation/ex1):

```
hello: hello.o
	gcc -o $@ $<
hello.o: hello.c
	gcc -c $<
```

Options:

`make` **`-n`** shows commands to be executed without executing them

`make` **`-k`** Continue as much as possible when error occurs

`make` **`-f`** `<filename>` Make uses <filename> instead of default file *makefile* o *Makefile*

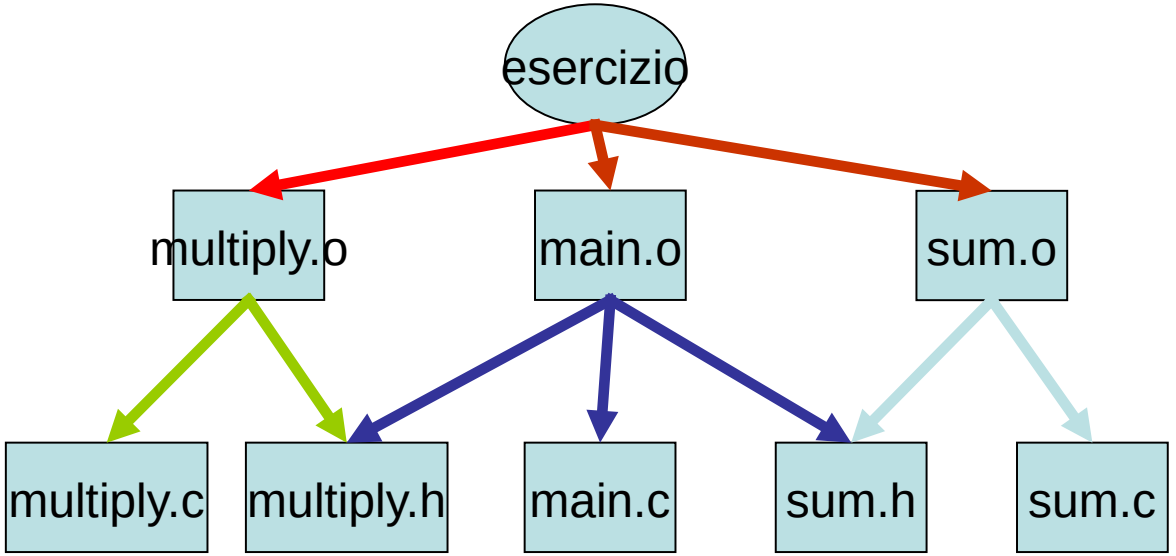
Exercise

- examples/make/ex2

  multiply.c
  multiply.h
  sum.c
  sum.h
  main.c

Exercise: write the Makefile to compile the project (+ target clean)

Solution




```
esercizio: main.o multiply.o sum.o
    gcc -o esercizio main.o multiply.o sum.o
main.o: main.c multiply.h sum.h
    gcc -c main.c
multiply.o: multiply.c multiply.h
    gcc -c multiply.c
sum.o: sum.h sum.c
    gcc -c sum.c
clean:
    rm *.o esercizio
```

Measuring Execution Times

- **gprof**: GNU tool to measure performances of programs
  - It tracks all the calls to functions and assessment of their execution times
  - developers can find functions with high processing time and focus on their
- Calling grof:
  - compile with option ***-pg***

```
$ gcc -Wall -c -pg main.c
$ gcc -Wall -pg main.o
```

  - this executable is *instrumented*: it contains additional instruction to register function calls
  - run the executable: *./a.out*
  - results written in file *gmon.out* that can be analysed with tool gprof

```
$ gprof a.out
```

```
$ gprof a.out
Flat profile:
Each sample counts as 0.01 seconds.
 %     cumul.    self              self    total
time seconds seconds      calls us/call us/call name
68.59    2.14    2.14 62135400    0.03     0.03 step
31.09    3.11    0.97   499999    1.94     6.22 nseq
 0.32    3.12    0.01                           main
```

Again on Libraries

- ***ar***: archiver that collects object files into a single archive file .a (static library)
- Illustrated by example ax_ex
  - source files *saluti.h*, *hello.c* and *bye.c*
  - first object file *hello.o* from file *hello.c*, the second *bye.o* from *bye.c*

    $ gcc -Wall -c hello.c

    $ gcc -Wall -c bye.c
  - a common header file *saluti.h* for the library interface
  - create library *libsalut.a* with command ***ar*** with options ***rc***

    $ ar cr libsaluti.a hello.o bye.o

    options: **c** = create, **r** = replace
  - Linking the library to an executable with source code *main.c*:

    $ gcc -Wall main.c -L. -lsaluti -o program

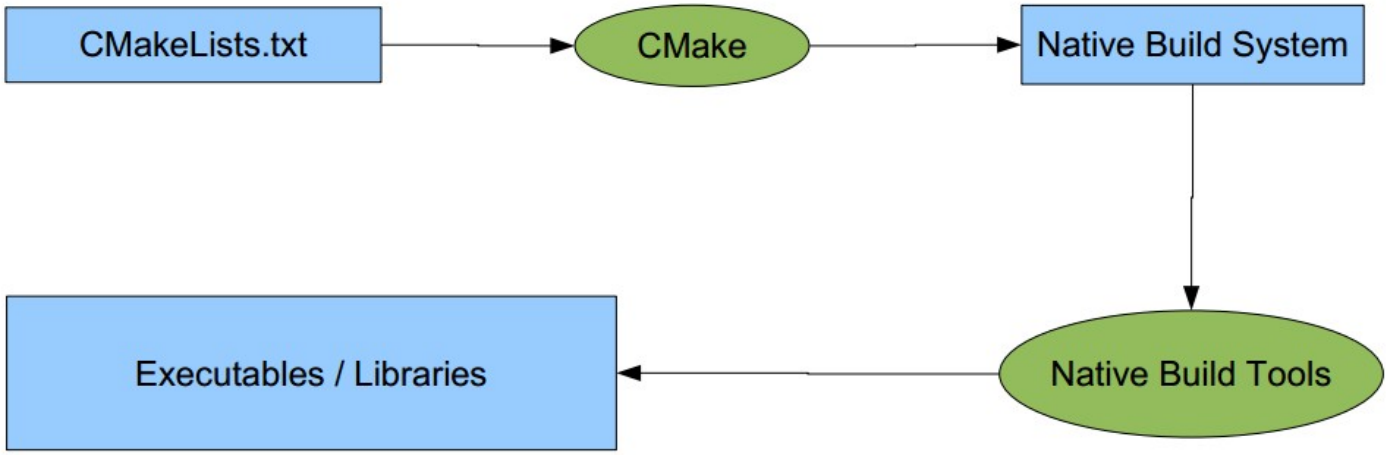
CMake

- **CMake**: open source and crossplatform tool for compiling
  - It is a “make makefile”
- Designed to be portable on different OS: it supports different project formats like Makefiles and MS Visual Studio project files
- Solving dependencies from other libraries
  - specific library scripts mylibrary.cmake (usually installed in system dirs */usr/share* or */usr/local/share*, or locally in cmake/)
  - it finds paths to header and library directories and list of library components
  - it finds the dependencies of dependencies (if script are well written!)
- Cmake supports many programming languarges: C, C++, Fortan, Java, Perl, Python..
  - … but it is commonly used in C/C++ projects
- Cmake does not list execute compiling: it creates the Makefile for compiling

CMake

Using CMake:

1. Write the source code (e.g. divided in *include/* and *src/*)
2. Write script file '*CMakeLists.txt*' in main source directory
3. Run *cmake* (usually in a specific directory *build*) to generate the Makefile
4. Run make to compile the project



CMake

- Example: project feature_cv_example using library OpenCV

```
cmake_minimum_required(VERSION 2.4.6)
project(feature_cv_example)
add_definitions(-std=c++0x)     # add specific command line options of compiler
set(CMAKE_BUILD_TYPE RelWithDebInfo)

# Solve dependency on external library OpenCV: results in variables
# ${OpenCV_INCLUDE_DIRS}, ${OpenCV_LIBS}
find_package(OpenCV REQUIRED)
include_directories(${OpenCV_INCLUDE_DIRS})
include_directories(src)                # local header file

add_executable(matchFeatures src/matchFeatures.cpp src/ParamMap.cpp)
target_link_libraries(matchFeatures ${OpenCV_LIBS})
```

CMake

- Enter project directory with file *CMakeLists.txt*

  cd example_image_features

- Create compiling directory *build/* and run cmake

  mkdir build

  cd build

  cmake ..

- Run make in *build/*

  make

```
-- The C compiler identification is GNU 9.3.0
-- The CXX compiler identification is GNU 9.3.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found OpenCV: /usr (found version "4.2.0")
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dario/robotica_ws/src/ra-
teaching/material/non_ros/example_image_features/build
```

CMake

- Other Cmake commands:

SET(VAR value [CACHE TYPE DOCSTRING [FORCE]])
ADD_EXECUTABLE
ADD_LIBRARY
MESSAGE
LIST(APPEND|INSERT|LENGTH|GET|REMOVE_ITEM|REMOVE_AT|SORT ...)
FIND_FILE
FIND_LIBRARY
FIND_PROGRAM
FIND_PACKAGE
EXEC_PROGRAM(bin [work_dir] ARGS <..> [OUTPUT_VARIABLE var] [RETURN_VALUE var])
OPTION(OPTION_VAR “description string” [initial value])

- IF() ... ELSE()/ELSEIF() ... ENDIF()
  - Very useful: IF(APPLE); IF(UNIX); IF(WIN32)
- WHILE() ... ENDWHILE()
- FOREACH() ... ENDFOREACH()

References

pagine web ufficiali del Progetto GNU dedicate a GCC
http://www.gnu.org/software/gcc/

Using GCC (for GCC version 3.3.1) di Richard M. Stallman e la Comunità di Sviluppatori di GCC (pubblicato da GNU Press, ISBN 1882114396)

Herbert Schildt - C: The Complete Reference, 4th Ed. (McGraw Hill)

The C Programming Language (ANSI edition) Brian W. Kerighan, Dennis Ritchie (ISBN 0131103628)

Richard Stevens - UNIX Network Programming Vol 1, 2nd Ed. (Prentice Hall)

Robert Sedgewick - Algorithms (Addison-Wesley)

CMake, the cross-platform, open-source build system, http://www.cmake.org/