Exercise 1: Programming Tools in Linux/UNIX

Sistemi Operativi ed in Tempo Reale
AA 2023/2024

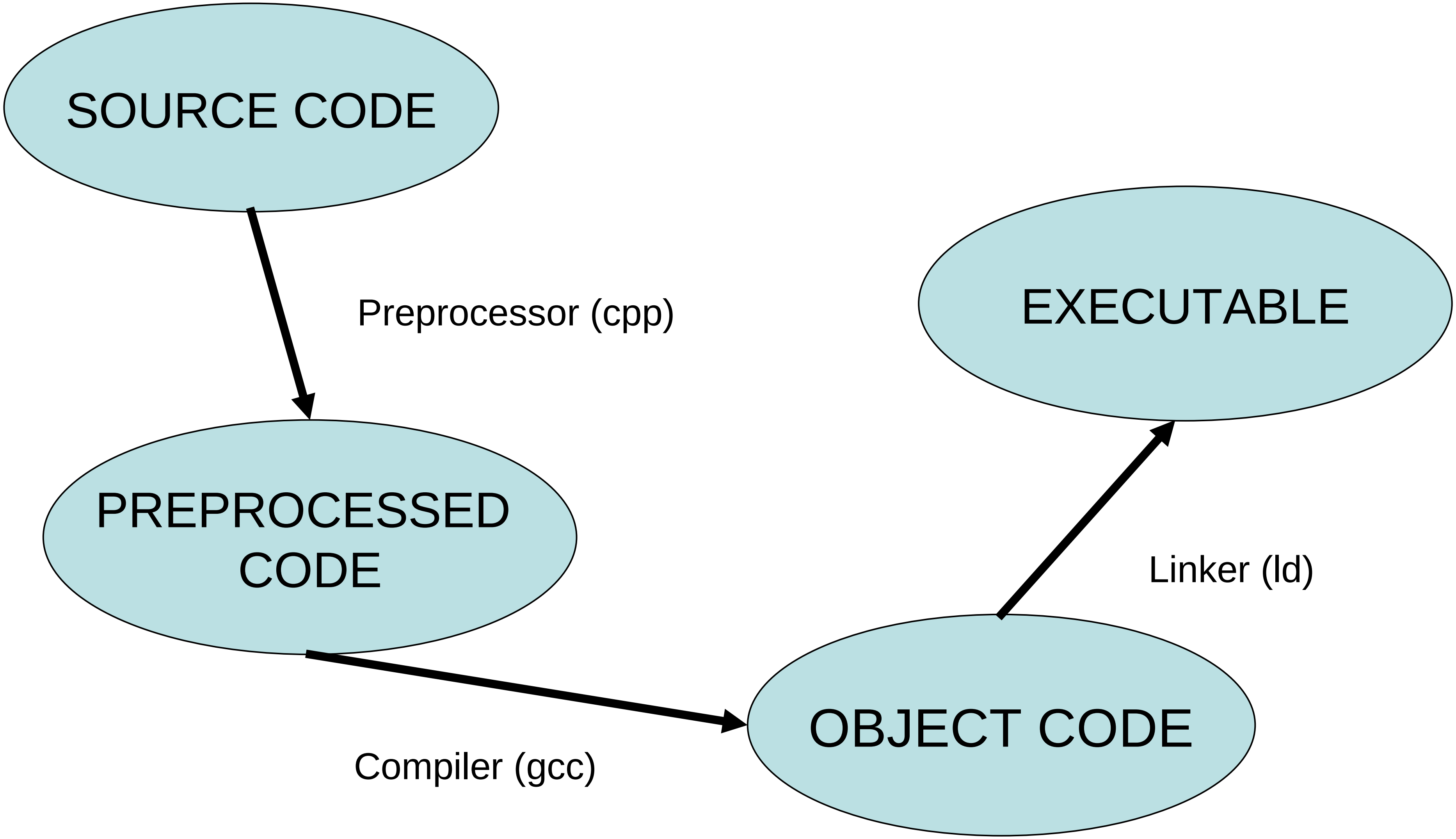
Outline

- Development tools in UNIX
- Compiling process
  - Preprocessor
  - Compiling
  - Linking
- Make and CMake

Programming Language C

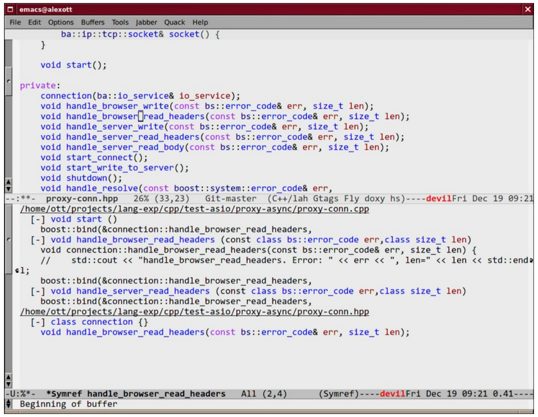
- Most of the kernels of common OS (Windows, Linux, MacOS, Android, iOS, …) are written in C
- System call libraries written in C and inspiring the APIs of other high level languages
- Linux systems have standard development tools
  - GNU Compiler `gcc`
  - Debugger `gdb`
  - Project management `make`, `cmake`
- Although other update applications, C and development tools are a *lingua franca* among programmers

Compiling Process

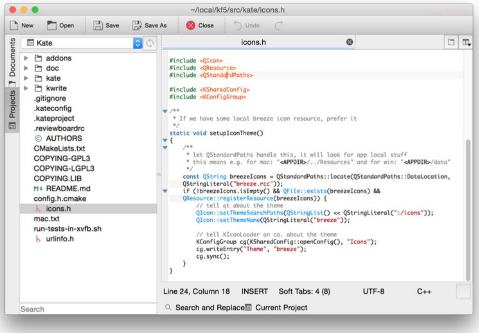


Editor

- Simple Editors
  - Emacs/Xemacs
  - Kate/Kwrite [ambiente grafico KDE]
  - Gedit [ambiente grafico Gnome]
  - Vi/Vim

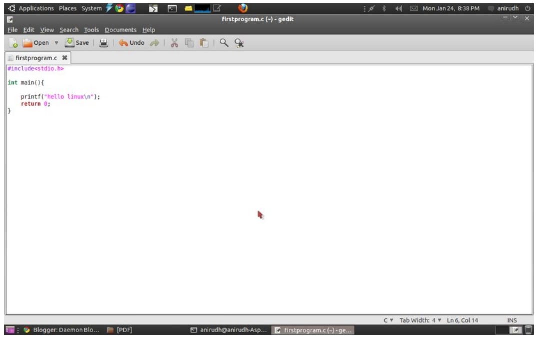


```
        ba::ip::tcp::socket& socket() {
    }

    void start();

private:
    connection(ba::io_service& io_service);
    void handle_browser_write(const bs::error_code& err, size_t len);
    void handle_browser_read_headers(const bs::error_code& err, size_t len);
    void handle_server_write(const bs::error_code& err, size_t len);
    void handle_server_read_headers(const bs::error_code& err, size_t len);
    void handle_server_read_body(const bs::error_code& err, size_t len);
    void start_connect();
    void start_write_to_server();
    void shutdown();
    void handle_resolve(const boost::system::error_code& err,
--:**-  proxy-conn.hpp   26% (33,23)   Git-master  (C++/lah Gtags Fly doxy hs)----devilFri Dec 19 09:21
/home/ott/projects/lang-exp/cpp/test-asio/proxy-async/proxy-conn.cpp
  [-] void start ()
    boost::bind(&connection::handle_browser_read_headers,
  [-] void handle_browser_read_headers (const class bs::error_code err,class size_t len)
    void connection::handle_browser_read_headers(const bs::error_code& err, size_t len) {
    //    std::cout << "handle_browser_read_headers. Error: " << err << ", len=" << len << std::end
l;
    boost::bind(&connection::handle_browser_read_headers,
  [-] void handle_server_read_headers (const class bs::error_code err,class size_t len)
    boost::bind(&connection::handle_browser_read_headers,
/home/ott/projects/lang-exp/cpp/test-asio/proxy-async/proxy-conn.hpp
  [-] class connection {}
    void handle_browser_read_headers(const bs::error_code& err, size_t len);

-U:%*-  *Symref handle_browser_read_headers   All (2,4)     (Symref)----devilFri Dec 19 09:21 0.41----
Beginning of buffer
```



```
#include <QIcon>
#include <QResource>
#include <QStandardPaths>

#include <KSharedConfig>
#include <KConfigGroup>

/**
 * If we have some local breeze icon resource, prefer it
 */
static void setupIconTheme()
{
    /**
     * let QStandardPaths handle this, it will look for app local stuff
     * this means e.g. for mac: "<APPDIR>/../Resources" and for win: "<APPDIR>/data"
     */
    const QString breezeIcons = QStandardPaths::locate(QStandardPaths::DataLocation,
    QStringLiteral("breeze.rcc"));
    if (!breezeIcons.isEmpty() && QFile::exists(breezeIcons) &&
    QResource::registerResource(breezeIcons)) {
        // tell qt about the theme
        QIcon::setThemeSearchPaths(QStringList() << QStringLiteral(":/icons"));
        QIcon::setThemeName(QStringLiteral("breeze"));

        // tell KIconLoader an co. about the theme
        KConfigGroup cg(KSharedConfig::openConfig(), "Icons");
        cg.writeEntry("Theme", "breeze");
        cg.sync();
    }
}
```

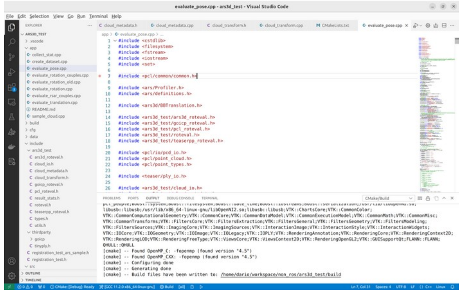
Line 24, Column 18   INSERT   Soft Tabs: 4 (8)   UTF-8   C++


Applications Places System
Mon Jan 24, 8:38 PM
anirudh
firstprogram.c (~) - gedit
File Edit View Search Tools Documents Help
Open
Save
Undo
firstprogram.c
Blogger: Daemon Blo...
[PDF]
anirudh@anirudh-Asp...
firstprogram.c (~) - ge...

```
#include<stdio.h>

int main(){

    printf("hello linux\n");
    return 0;
}
```

C   Tab Width: 4   Ln 6, Col 14   INS

- IDE (Integrated development environment)
  - VS Code
  - Kdevelop
  - Eclipse
  - NetBeans



```
#include <cstdlib>
#include <filesystem>
#include <fstream>
#include <iostream>
#include <set>

#include <pcl/common/common.h>

#include <ars/Profiler.h>
#include <ars/definitions.h>

#include <ars3d/BBTranslation.h>

#include <ars3d_test/ars3d_roteval.h>
#include <ars3d_test/goicp_roteval.h>
#include <ars3d_test/pcl_roteval.h>
#include <ars3d_test/roteval.h>
#include <ars3d_test/teaserpp_roteval.h>

#include <pcl/io/pcd_io.h>
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>

#include <teaser/ply_io.h>

#include <ars3d_test/cloud_io.h>
```

```
libusb::libusb;/usr/lib/x86_64-linux-gnu/libOpenNI2.so;libusb::libusb;VTK::ChartsCore;VTK::CommonColor;
VTK::CommonComputationalGeometry;VTK::CommonCore;VTK::CommonDataModel;VTK::CommonExecutionModel;VTK::CommonMath;VTK::CommonMisc;
VTK::CommonTransforms;VTK::FiltersCore;VTK::FiltersExtraction;VTK::FiltersGeneral;VTK::FiltersGeometry;VTK::FiltersModeling;
VTK::FiltersSources;VTK::ImagingCore;VTK::ImagingSources;VTK::InteractionImage;VTK::InteractionStyle;VTK::InteractionWidgets;
VTK::IOCore;VTK::IOGeometry;VTK::IOImage;VTK::IOLegacy;VTK::IOPLY;VTK::RenderingAnnotation;VTK::RenderingCore;VTK::RenderingContext2D;
VTK::RenderingLOD;VTK::RenderingFreeType;VTK::ViewsCore;VTK::ViewsContext2D;VTK::RenderingOpenGL2;VTK::GUISupportQt;FLANN::FLANN;
QHULL::QHULL
[cmake] -- Found OpenMP_C: -fopenmp (found version "4.5")
[cmake] -- Found OpenMP_CXX: -fopenmp (found version "4.5")
[cmake] -- Configuring done
[cmake] -- Generating done
[cmake] -- Build files have been written to: /home/dario/workspace/non_ros/ars3d_test/build
```

GNU Compiler gcc

- **Open-Source GNU** project
  - Compiler and Linker (and pre-processor)
  - Supports C/C++ (also other languages when configurated)
- **Command syntax**:

```
gcc <options> <arguments>
```

- Options: list of flags that control the compiler and the linker; there are options for compilation only, for linker only, or both
- Arguments: list of files that gcc reads and process depending on the given options

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`**.
- Directive `#define`

  - **`#define NAME expansion`**
  - e.g.: `#define MAX 10`

- All instances of *MAX* substituted by <u>string</u> 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 as: `((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

```
#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:

```
#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 with command line option '–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

- Compiler: it translates the source code into machine code\
- GNU compiler uses option **-c**
  - Syntax: `gcc -c` *`sourcefile.c`*
  - Example: `gcc -c hello.c`
- Output: the so called object file
  - Object file has the name of source file with changed extension **.o**

    e.g. `hello.c` → `hello.o`
  - Intermediate file according to **Executable and Linkable Format** (**ELF**) defined for executables, libraries, etc.
  - ELF files define symbols to functions
  - Object file may have incomplete references

Object File Symbols: `nm`

- Command nm shows the symbols in object files
- Example: `nm hello.o`

```
0000000000000000 T main
                 U puts
```

  - function `main` is a symbol in the text (**T**) of the source code
  - Function `puts` (included through `printf()`) is undefined (**U**) in the code
- Undefined symbols requires definition, e.g. in another object file or library
  - e.g. `put` is defined in GLIBC with I/O and other standard library functions

Compiler Options

- Main options of compiler
  - **–g**: add information useful for debugging (e.g. variables, symbols, line numbers) in object code
  - **–Wall**: enables all warning messages
  - **–pedantic**: displays all errors and warnings required by ANSI C standard
  - **–O1**, **–O2**, **–O3**: increasing level of optimization
  - **–O0**: no optimization

Linking

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

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

- Executing on executable command `nm hello`:

```
000000000000038c r __abi_tag
0000000000004010 B __bss_start
0000000000004010 b completed.0
                 w __cxa_finalize@GLIBC_2.2.5
0000000000004000 D __data_start
…
                 U __libc_start_main@GLIBC_2.34
0000000000001149 T main
                 U puts@GLIBC_2.2.5
00000000000010c0 t register_tm_clones
0000000000001060 T _start
0000000000004010 D __TMC_END__
```

puts() is still undefined, but there is the dynamic link to GLIBC!

Linking

- Example:

```
#include <stdio.h>
#include <math.h>

int main()
{
  float value = 5.0f;
  printf("The square root of %f is %f\n", value, sqrt(value));
  return 0;
}
```

- This example uses function `sqrt()` defined in library file `/usr/lib/libm.a` (not in GLIBC)
- Needed linking to math library:

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

(*) Newest versions of gcc are able to detect dependency to some standard libraries like libm.a and to implicitly link it

Linking

- **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`
- Note: the above linking step omitted the output file (hello2 in previous slides) and the default name of executables is a.out

Solving Paths

- Compiler needs to know the path where files are 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
- Compiler needs to know the path where files are located
  - Syntax: `-I/path/to/header`, `-L/path/to/library`
  - Example:

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

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 a library you must include its header file(s) .h
- **Static library**:
  - Extension **.a** (“archive”) in Linux (.lib in Windows)
  - A copy of library is integrated in the executable (no dependency from an extenal file .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)

Creating Static Libraries

- Creating a static library with command **`ar`**:

  ```
  ar -rc libname.a file1.o file2.o … fileN.o
  ```

  - File libname.a is an **archive** of functions defined in object files
  - Static library becomes part of executable

    ```
    gcc -o exec exec.o -I/path/to/header
    -L/path/to/libname.a -lname
    ```

- Standard name of libraries: `lib`*`name`*`.a`
  - Prefix: `lib`
  - Library name: *`name`*
  - Extension: `.a`

Creating Dynamic Libraries

- Shared objects **.so** are created from object files as:

```
gcc -shared -o libname.so file1.o ... fileN.o
```

- Dynamic linked library are not part of executable
  - The compiling command is the same as for .a:

```
gcc -o exec exec.o -I/path/to/header
-L/path/to/libname.so -lname
```

- When the executable is called the path to .so must be known
  - Standard environment variable `LD_LIBRARY_PATH`
  - Check its value on your system

```
export | grep LD_LIBRARY_PATH
```

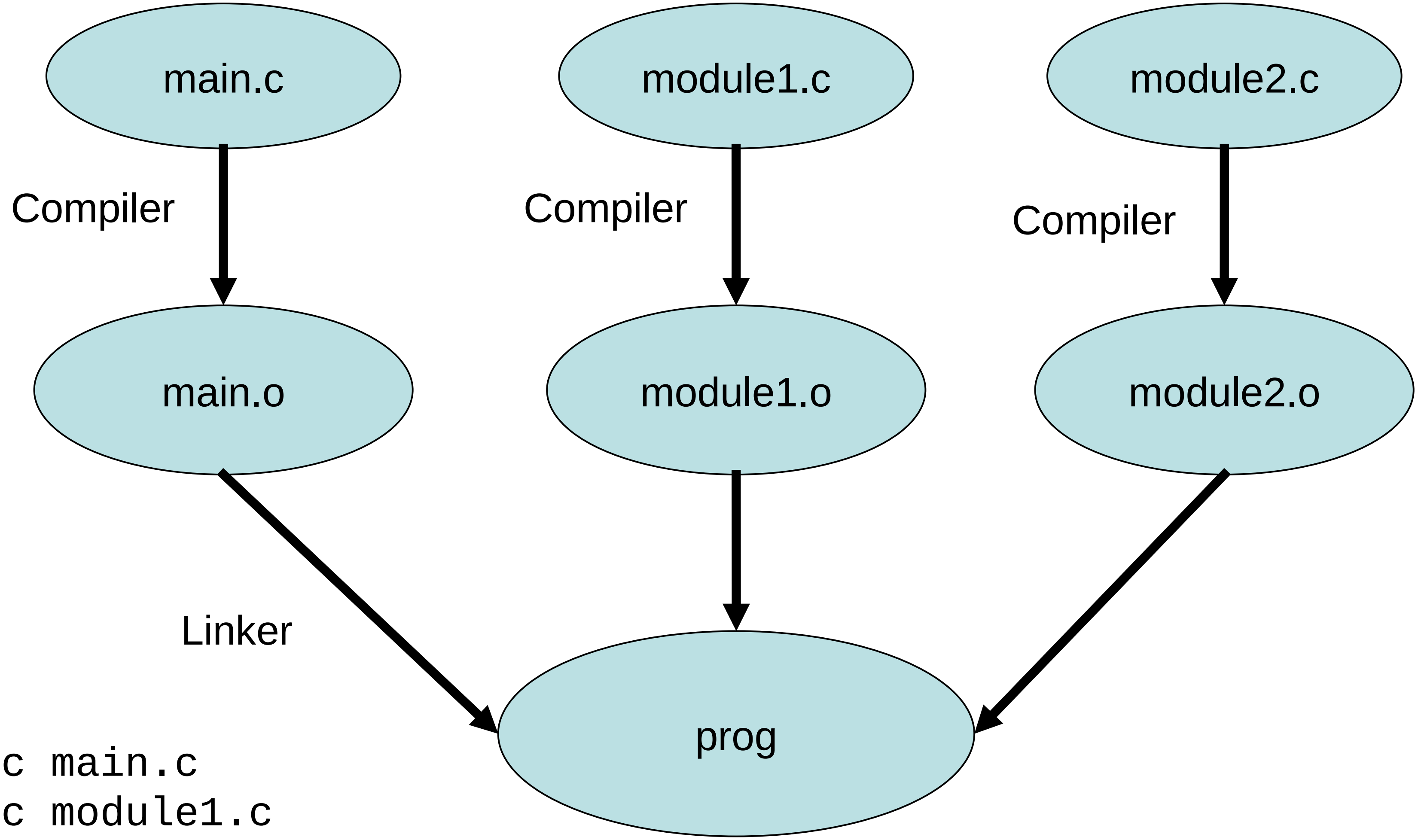
Function Prototypes

- Good practice: declare functions before using them (and before their definition)
- Example:

```
#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

- Example:

```
gcc -c main.c

gcc -c multiply.c

gcc main.o -o example [not working]

gcc main.o multiply.o -o example [working]
```

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

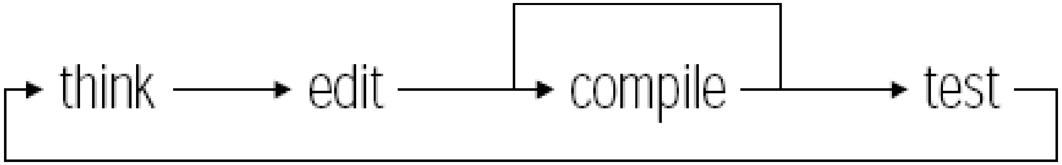
```
#ifndef FOO_H
#define FOO_H

… definition or inclusion of foo …

#endif
```

- Example:

```
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
- Development cycle of a program (repeated multiple times!)

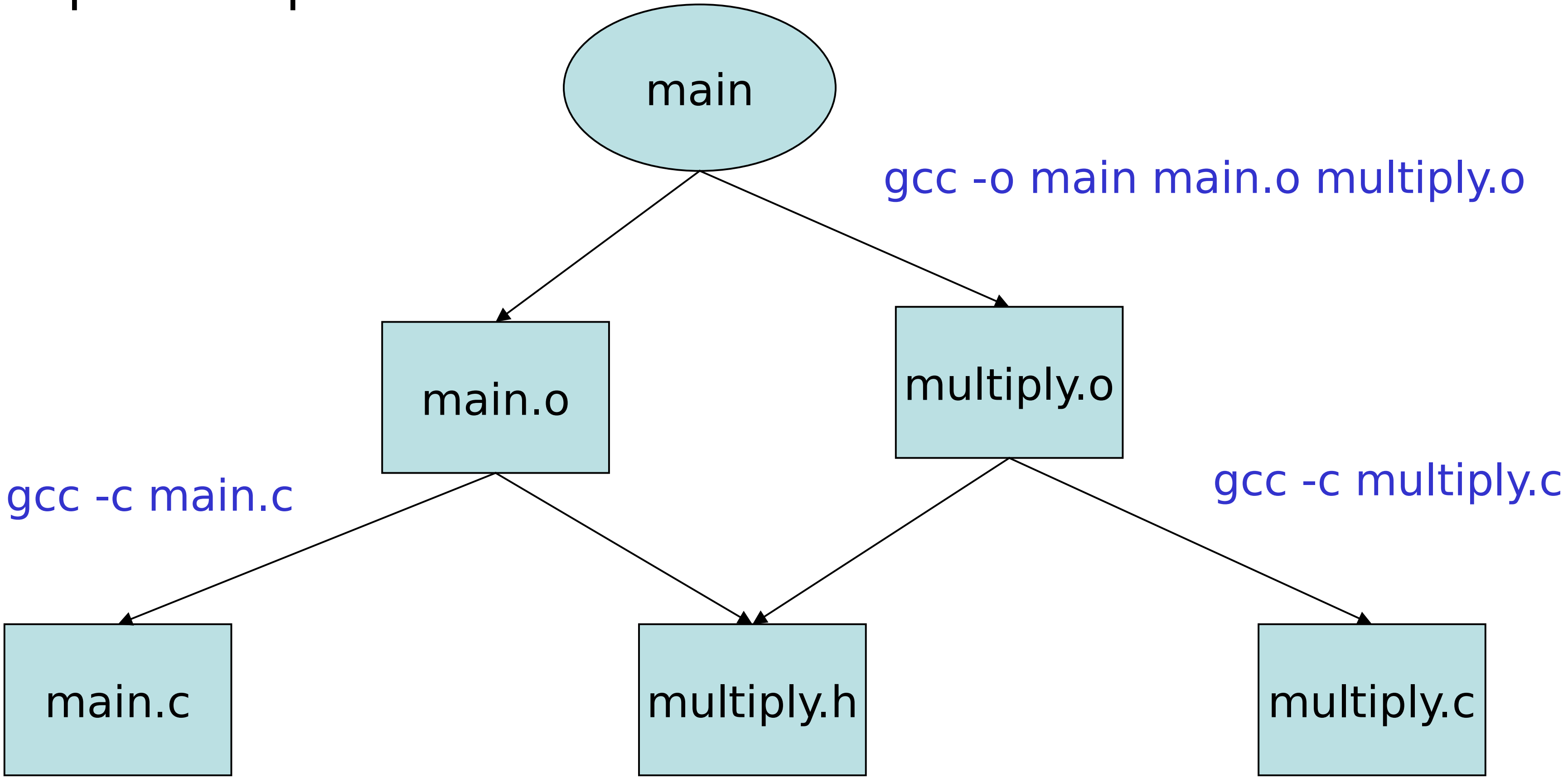


- Issues:
  - you change a file and forget to recompile it
  - interface changes (.h), but you forget to compile all the files depending on it

- **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

    target: dependencies
    [tab] commands

- Commands 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:

Example (/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*

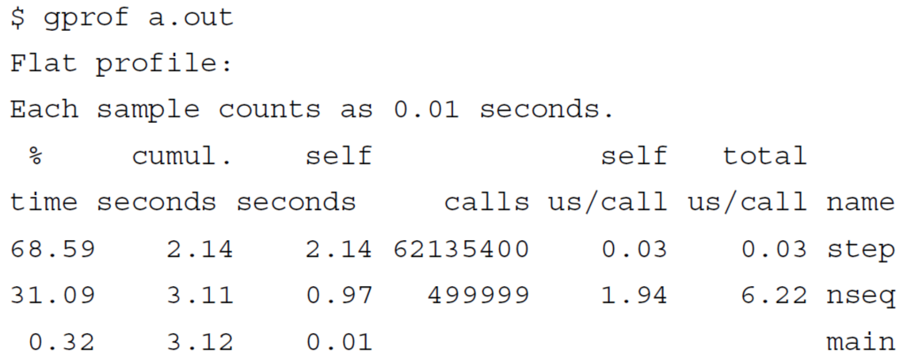
Measuring Execution Times

- **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
```

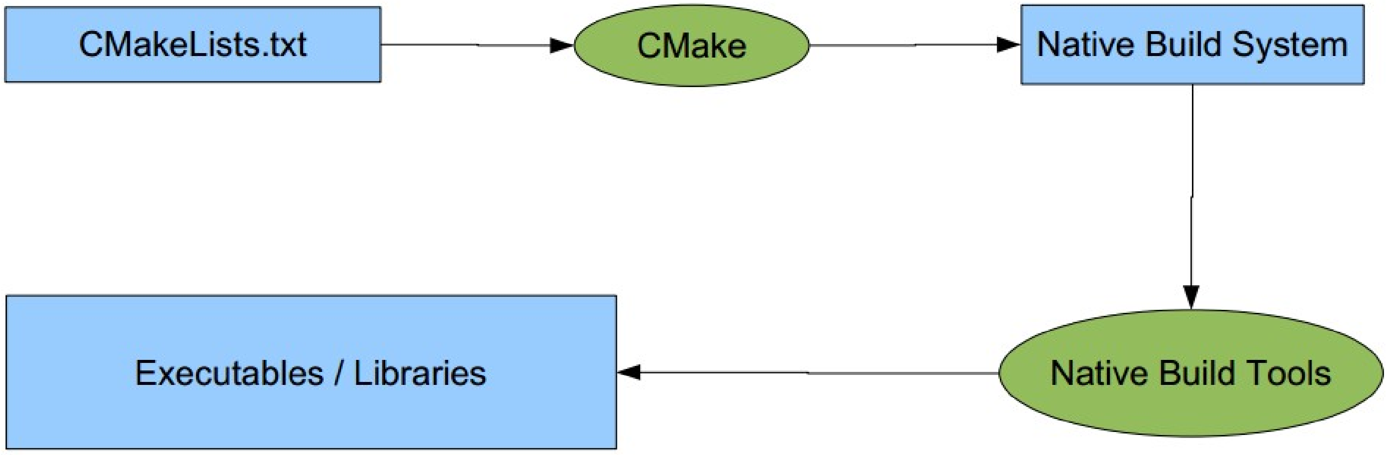
CMake

- **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: C++ 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
```

Exercise

- Multi-file project in `ex4_multifile/`:
  - Files: `main.c`, `fast_trigo.h`, `fast_trigo.c`, `test_func.h`, `util.h`
- Fix the errors:
  - Function re-definition: use the proper pre-processor macros
  - Separate function declaration in *.h* and definition in *.c*
- Create a library instead from auxilary files to be linked to the main file
- Create a *Makefile* for the whole compiling process