SAPIENZA
UNIVERSITÀ DI ROMA

Robot Programming

Compilers and Build Systems

Giorgio Grisetti

Programs

Piece of software executed by

- An interpreter program (Interpreters)
- An emulator program (Virtual Machine)
- Directly by the CPU

For something to happen it has to be executed by the CPU.

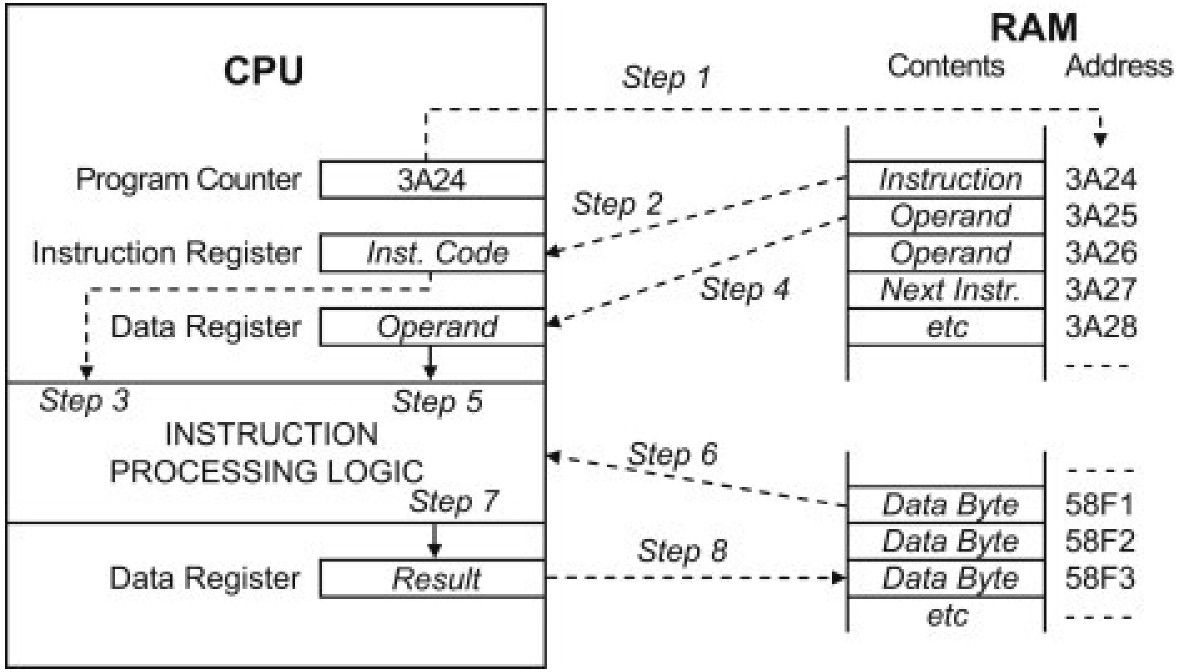
Machines: Native

The CPU processes instructions sequentially

Each instruction consists of one or more binary digits

The CPU fetches instructions and operands from memory

The machine instructions have a 1 to 1 correspondence with architecture specific alphanumeric opcodes (Assembly language)



Executing Bytecode

Some compilers/languages are designed with poerability among architectures in mind.

To this extent, they produce machine language for a virtual processor

To run these binaries, one needs a native program emulating the virtual CPU

For some code chunks the process can be done once, to generate native machine code that is then run faster (Just In time Compilation)

```
i = j + k;            ILOAD j      // i = j + k       Byte Offset
if (i == 3)           ILOAD k                         0     0x15 0x02
    k = 0;            IADD                            2     0x15 0x03
else                  ISTORE i                        4     0x60
    j = j – 1;        ILOAD i      // if (i < 3)      5     0x36 0x01
                      BIPUSH 3                        7     0x15 0x01
                      IF_ICMPEQ L1                    9     0x10 0x03
                      ILOAD j      // j = j – 1       11    0x9F 0x00 0x0D
                      BIPUSH 1                        14    0x15 0x02
                      ISUB                            16    0x10 0x01
                      ISTORE j                        18    0x64
                      GOTO L2                         19    0x36 0x02
                  L1: BIPUSH 0     // k = 0           21    0xA7 0x00 0x07
                      ISTORE k                        24    0x10 0x00
                  L2:                                 26    0x36 0x03
                                                      28
```

Running an interpreter script

Interpreters are programs that directly execute a source file, performing the parsing/code execution on the spot

The execution is done **during** the parsing

Examples

- BASIC
- BASH
- PYTHON
- Matlab/Octave

```
giorgio@frisbi2:~$ octave-cli
GNU Octave, version 4.2.2
Copyright (C) 2018 John W. Eaton and others.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.  For details, type 'warranty'.

Octave was configured for "x86_64-pc-linux-gnu".

Additional information about Octave is available at http://www.octave.org.

Please contribute if you find this software useful.
For more information, visit http://www.octave.org/get-involved.html

Read http://www.octave.org/bugs.html to learn how to submit bug reports.
For information about changes from previous versions, type 'news'.

octave:1> A=[1,2,3; 4,5,6]
A =

   1   2   3
   4   5   6

octave:2> 
```

Program Files and Execution

A program file contains

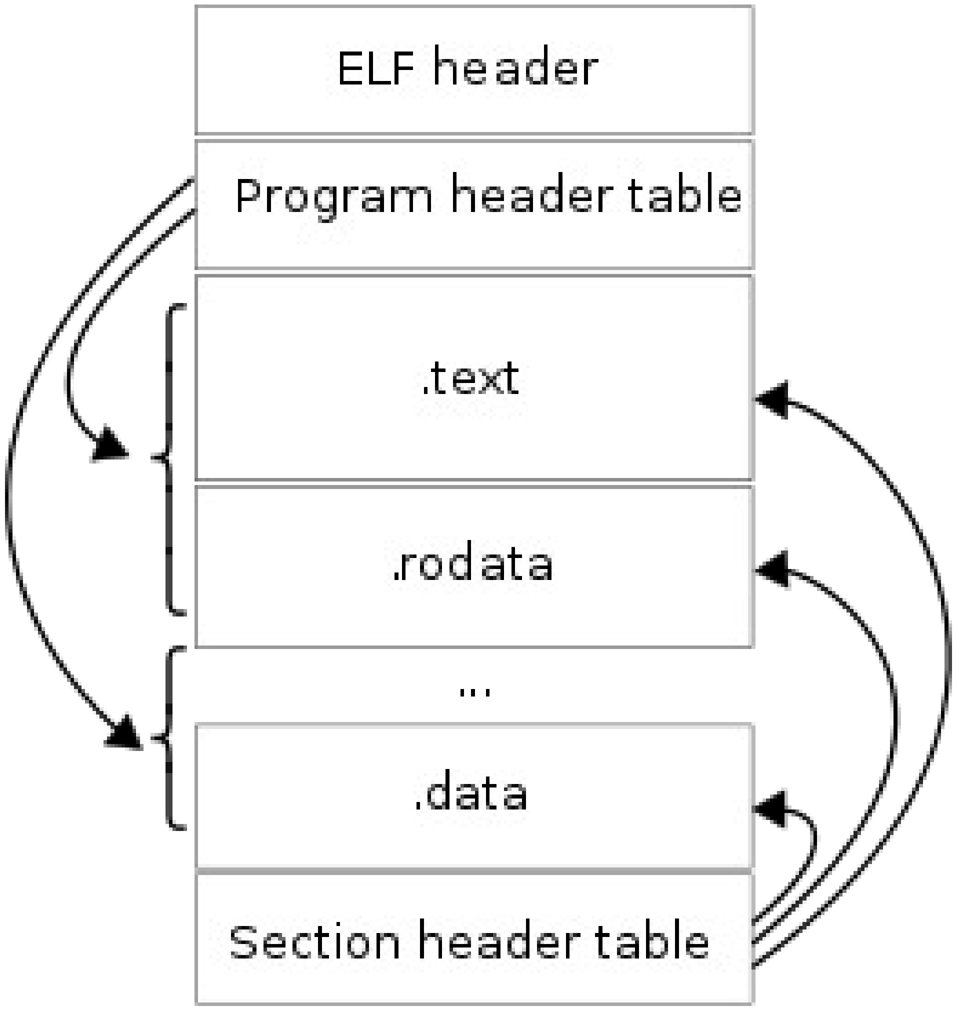
- machine code and
- meta-information

to generate a memory image suitable for CPU execution

Some OS/Environment support is needed to make a memory image of a program file.

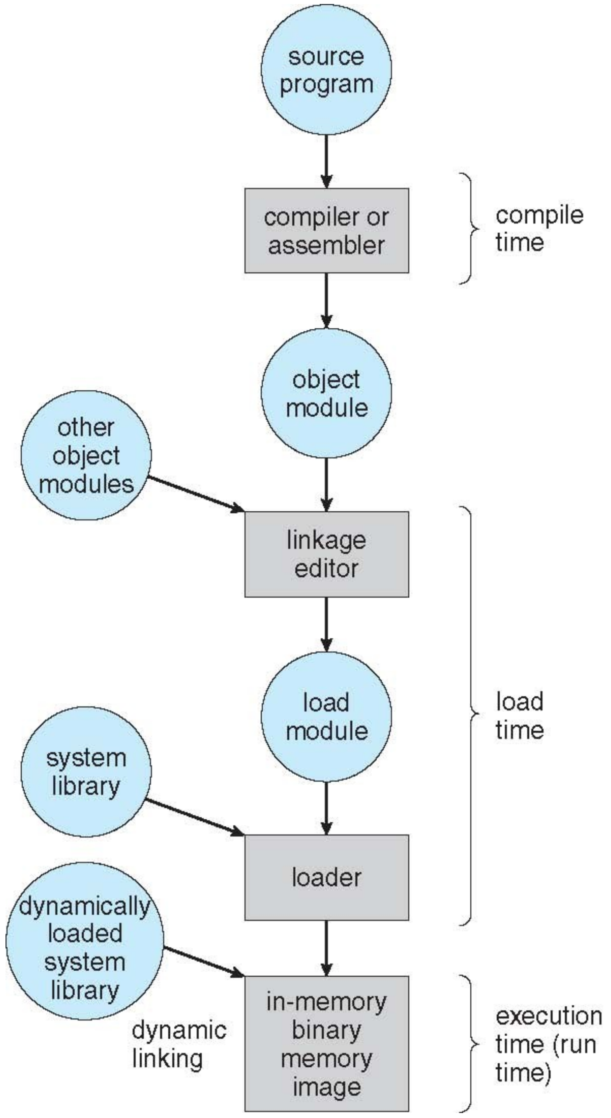
A running program is a process.

The same program file can be loaded multiple times, leading to different processes.



Building a Program File

- Files
  - .h .hpp: Headers
  - .c .cpp .S: Source Files
  - .o: object files
  - .a: static libraries
  - .so: shared libraries
  - Executables (no extensions on unix)
- Compiling
- Linking



Building an Executable Program

Example:

- compiler+linker+assembler: g++/gcc
- Source file: **hello_world.cpp**

```cpp
// hello_world.cpp
#include <iostream>
using namespace std;

int main(int argc,
const char** argv) {
  cout << "hello world" << endl;
}
```

Commands

- **g++ <options> -c <source file>**

  generates an object file from a cpp

  e.g hello_world.cpp → hello_world.o

- **g++ <options> -o <name> <files>**

  links together (and compiles if needed) all files in the command line to generate a program file called <name>

The same machinery holds also if the C compiler (gcc) is used

Building an Executable Program

Example:

– compiler+linker+assembler: g++/gcc

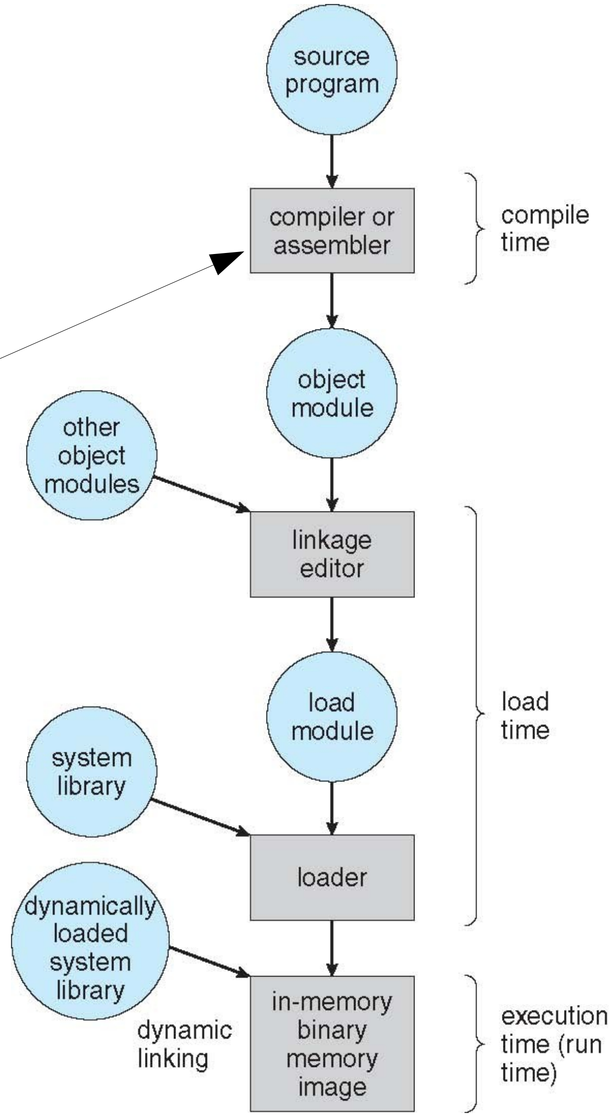
– Source file: hello_world.cpp

Commands

– **`g++ <options> -c <source file>`**
  generates an object file from a cpp
  e.g hello_world.cpp → hello_world.o

– **`g++ <options> -o <name> <files>`**
  links together (and compiles if needed) all files in the command line to generate a program file called <name>

The same machinery holds also if the C compiler (gcc) is used



Building an Executable Program

Example:

- compiler+linker+assembler: g++/gcc
- Source file: hello_world.cpp

Commands

- **`g++ <options> -c <source file>`**

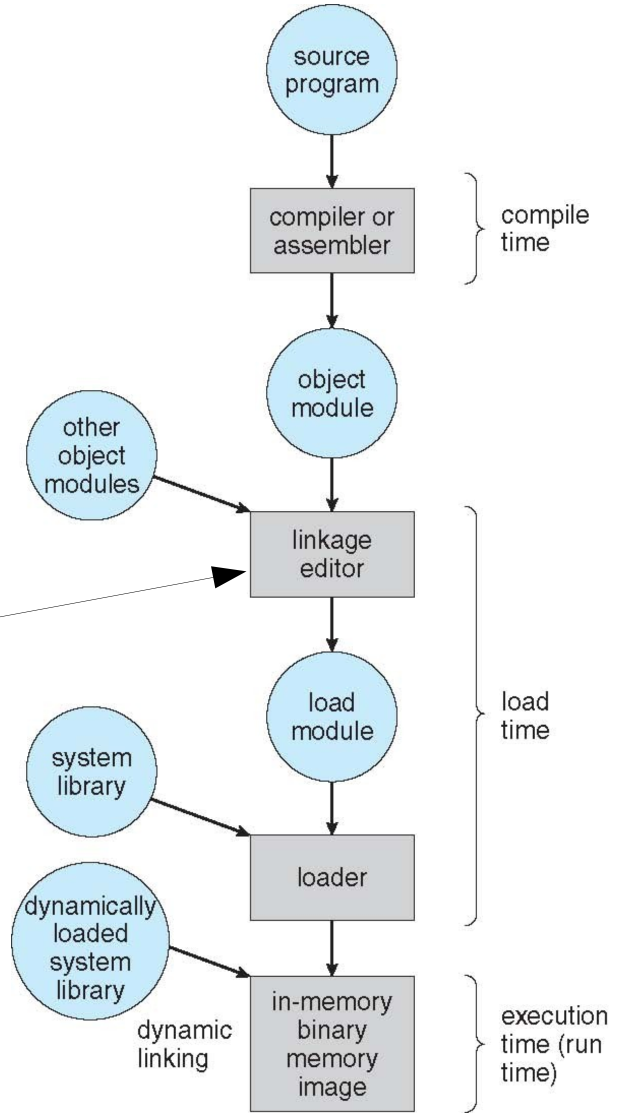
  generates an object file from a cpp

  e.g hello_world.cpp → hello_world.o

- **`g++ <options> -o <name> <files>`**

  links together (and compiles if needed) all files in the command line to generate a program file called <name>

The same machinery holds also if the C compiler (gcc) is used



01_building_a_cpp_program

Running an Executable

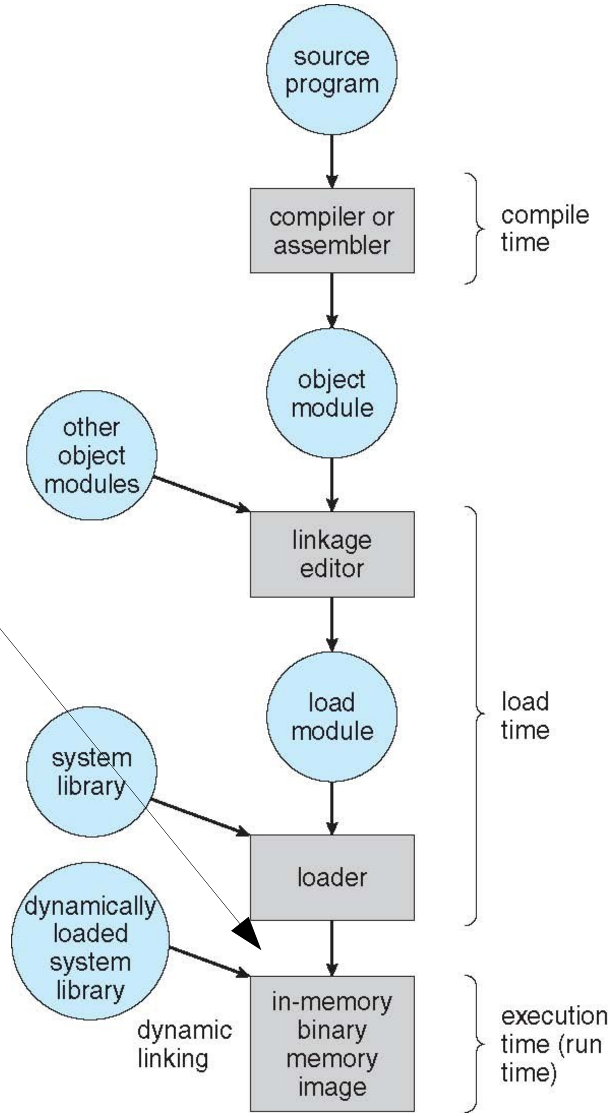
```
$> hello_world
```

The command shell invokes the OS to create a new process and load the process image from the disk in memory

It deals with loading the necessary shared libraryes (.so, the linux dll)

Information about which shared objects to load at runtime should be provided during the compilation process by using the option

```
-l <library_name>
```



Large Builds

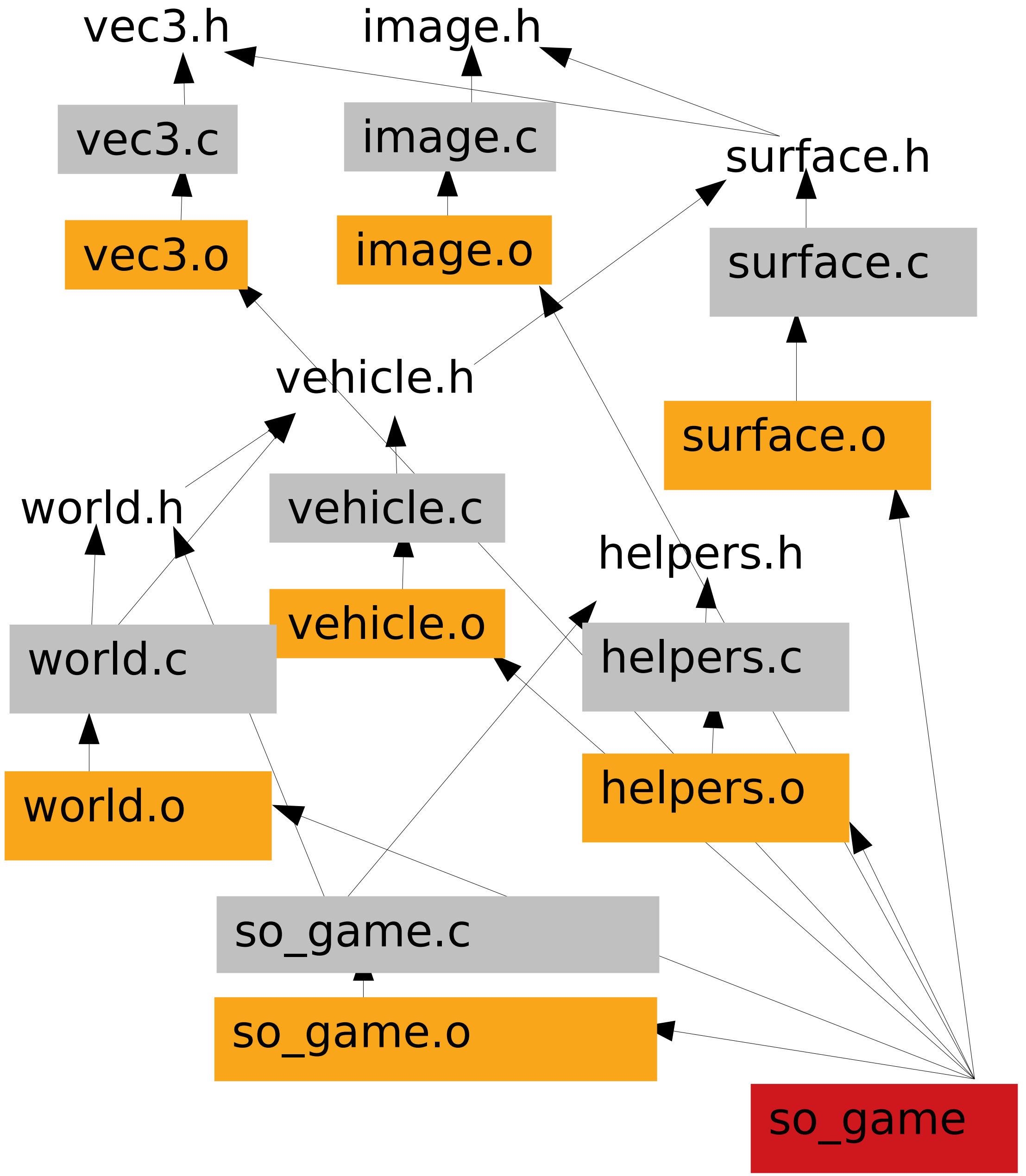
Partition the project in many files

Each file (or set of files), gets compiled in a .o

Upon change of the sources/headers, only the affected parts are recompiled and linked

Dependencies: each **.o** depends o

- the sources that are used to build it and
- the headers included by these sources



02_so_game

Show the build by hand

Make a script

```
gcc -Wall -O3 -std=gnu99 -I/usr/include/GL -c -o vec3.o vec3.c
gcc -Wall -O3 -std=gnu99 -I/usr/include/GL -c -o surface.o surface.c
gcc -Wall -O3 -std=gnu99 -I/usr/include/GL -c -o image.o image.c
gcc -Wall -O3 -std=gnu99 -I/usr/include/GL -c -o vehicle.o vehicle.c
gcc -Wall -O3 -std=gnu99 -I/usr/include/GL -c -o world.o world.c
gcc -Wall -O3 -std=gnu99 -I/usr/include/GL -c -o helpers.o helpers.c
gcc -Wall -O3 -std=gnu99 -I/usr/include/GL -c -o so_game.o so_game.c
gcc -Wall -O3 -std=gnu99 -I/usr/include/GL -o so_game vec3.o surface.o
image.o vehicle.o world.o helpers.o so_game.o -lglut -lGLU -lGL -lm
```

Each time a file change issue the script and compile all from scratch

unacceptable (a clean build of a large project can take hours)

Makefiles

Way to automate the build process

Made of a set of rules, triggered by the modification time

Rules in the form

```
    <target> : [tab] <dependancies>
    <tab>        command
```

e.g

```
vec3.o :    vec3.c vec3.h
            gcc -std=gnu99 -c -o  vec3.o vec3.c
```

Executes gcc if the modification time of **`vec3.o`** is older than the one of **`vec3.c`** or **`vec3.h`**

Dependancies

With

`gcc (or g++) -MM <source file>` you get the “head” of the rule (target and dependancies)

Try executing

```
$> gcc -MM vec3.c
```

Makefiles

```
#var declarations, expanded with $(name)
CC=gcc
LIBS=-lglut -lGLU -lGL -lm
INCLUDES=-I/usr/include/GL
CCOPTS= -Wall -O3 -std=gnu99 $(INCLUDES)

#1st rule <head>: <dependancies>, root of tree
all:  so_game

#to make vec.o you need vec.h. and vec.c, bottom line
# is the command to execute if the rule matches
# <target> : <dependancies>
#             <command>
vec3.o:  vec3.c vec3.h
           gcc $(CCOPTS) -c -o vec3.o vec3.c

surface.o: surface.c surface.h vec3.h image.h
      gcc $(CCOPTS) -c -o surface.o surface.c

image.o:image.c image.h
      gcc $(CCOPTS) -c -o image.o image.c
```

```
vehicle.o: vehicle.c vehicle.h surface.h vec3.h image.h
      gcc $(CCOPTS) -c -o vehicle.o vehicle.c

world.o: world.c world.h image.h surface.h\ vec3.h vehicle.h
helpers.h
     gcc $(CCOPTS) -c -o world.o world.c

helpers.o: helpers.c helpers.h
     gcc $(CCOPTS) -c -o helpers.o helpers.c

so_game.o: so_game.c world.h image.h surface.h\
    vec3.h vehicle.h helpers.h
     gcc $(CCOPTS) -c -o so_game.o so_game.c

# here we call the linker
so_game: vec3.o surface.o image.o vehicle.o \
   world.o helpers.o so_game.o

   gcc $(CCOPTS) -o so_game vec3.o surface.o \
          image.o vehicle.o world.o helpers.o \
     so_game.o $(LIBS)

clean:
    rm -rf *.o *~ cube main so_game
```

Makefiles

Call by issuing the following command in a folder containing a **`Makefile`**

`$make`

It will trigger a object dependant compilation, that verifies which rules to execute based on the modification time

Actualize the change time of **vec3.h** and check what happens by calling make again

`$touch vec3.h`

`$make`

Modify **`vec3.c`**, and see what happens by issiung the same command

Dealing with External Packages

Complex projects tend to rely on other complex projects/libraries

E.g. A simulator might rely on

- the OpenGL system to display the data,
- on a finite element simulation system to compute the dynamics
- on a linear algebra library to do some geometric calculation
- on some tool to load/save formatted files and images and so on

Finding the packages (specially non standard ones) might be annoying

- Different OS/distributions
- Different revisions
- Different architectures
- Different revisions of a package installed
- Packages can depend on each other

Dealing with External Packages

To the extent of a C++ build, a package comes as

- A set of include files (,h, .hpp)
- A set of libraries (.so, .a)

In the .o generation, one needs to specify

- The include paths, that contains the .h files of the project

  `-I<path1> -I<path2>`

- Potential defines, that might trigger library specific behaviors

  `-D<statement>` (equivalent to **`#define <statement>`** in the source file)

In the linking phase, one needs to specify

- The directory of the libraries

  `-L <path>`

- The libraries

  `-l<libname_without_prefix>, e.g. -lm` links **`libm.so`**

pkg-config

How to automate the retrieval of compilation parameters for the installed libraries and packages

Who tells you where are the library files and the includes of a package

- Hardcoding them (usually on the same OS/Release works, but expect plenty of emails)
- Automating the search (given the library package name)

Pkg config:

- A database consisting of text files describing the packages
- For each package the include (-I<…>) and the library (-l<…>) files are provided

try

```
$pkg-config --libs gl
$pkg-config --cflags gl
```

pkg-config

*.pc files contained in **`$PKG_CONFIG_PATH`** variable

They contain the answer to the potential queries made by pkg config

example (roscpp.pc)

```
prefix=/opt/ros/melodic
Name: roscpp
Description: Description of roscpp
Version: 1.14.11
Cflags: -I${prefix}/include -I/usr/include
Libs: -L${prefix}/lib -lroscpp -lpthread
/usr/lib/x86_64-linux-gnu/libboost_chrono.so /usr/lib/x86_64-linux-
gnu/libboost_filesystem.so
/usr/lib/x86_64-linux-gnu/libboost_system.so
Requires: cpp_common message_runtime rosconsole roscpp_serialization
roscpp_traits rosgraph_msgs rostime std_msgs xmlrpcpp
```

pkg-config and makefiles

```
CC=gcc # CC is our compiler, we can change later if we want

# execute pkg-config to find the libraries
LIBS=$(shell pkg-config --libs glut glu gl) -lm
INCLUDES=$(shell pkg-config --libs glut glu gl)

CCOPTS= -Wall -O3 -std=gnu99 $(INCLUDES)

OBJS=vec3.o surface.o image.o vehicle.o world.o helpers.o so_game.o

.phony: clean all

all: $(OBJS)
	$(CC) $(CCOPTS) -o so_game $^ $(LIBS)

# generic rule, to make a .o you need a .c and a .h of the same name
%.o: %.c %.h
	$(CC) $(CCOPTS) -c -o $@ $<

clean:
	rm -rf *.o *~ cube main so_game
```

CMake

A lot of things have been done with make/pkgconfig

- Each package should provide a .pc file
- The compiler arguments/options should be canonized

Sometimes between different revisions/OSes the packages do not match and the compiler options differ

The build process might require different steps (think cross-compilation)

Solution:

CMake: a makefile generator, that requires to write less things

The dependencies are deduced from the sources (by using a compiler toolchain)

Only tell in CMakeLists.txt

- What packages you need
- Where are your sources

For each target:

- The files it requires
- What are the libraries it depends from

To compile

- `$cmake -c <path to CMakeLists.txt>`
- `$make (from the same folder)`

CMake Example

```
cmake_minimum_required (VERSION 2.8.11)
project (so_game)
```

This is the preamble
Defines the minimum cmake version and the project name

```
find_package(OpenGL REQUIRED)
find_package(GLUT REQUIRED)

include_directories(${OPENGL_INCLUDE_DIRS} ${GLUT_INCLUDE_DIRS})

add_executable(so_game
  helpers.c  image.c  so_game.c
  surface.c  vec3.c  vehicle.c  world.c
)

#tell the executable the libraries it needs
target_link_libraries(so_game
  ${OPENGL_gl_LIBRARY} #
  ${OPENGL_glu_LIBRARY}
  ${GLUT_glut_LIBRARY}
  m
 )
```

CMake Example

```
cmake_minimum_required (VERSION 2.8.11)
project (so_game)

find_package(OpenGL REQUIRED)
find_package(GLUT REQUIRED)

include_directories(${OPENGL_INCLUDE_DIRS} ${GLUT_INCLUDE_DIRS})

add_executable(so_game
  helpers.c  image.c  so_game.c
  surface.c  vec3.c  vehicle.c  world.c
)

#tell the executable the libraries it needs
target_link_libraries(so_game
  ${OPENGL_gl_LIBRARY}
  ${OPENGL_glu_LIBRARY}
  ${GLUT_glut_LIBRARY}
  m
 )
```

This invokes the routines to find a package called OpenGL.
If found, some variables, typically OPENGL_LIBRARIES and OPENGL_INCLUDE_DIR are set

If not found the generation aborts with an error

CMake Example

```
cmake_minimum_required (VERSION 2.8.11)
project (so_game)

find_package(OpenGL REQUIRED)
find_package(GLUT REQUIRED)

include_directories(${OPENGL_INCLUDE_DIRS} ${GLUT_INCLUDE_DIRS})

add_executable(so_game
  helpers.c  image.c  so_game.c
  surface.c  vec3.c  vehicle.c  world.c
)

#tell the executable the libraries it needs
target_link_libraries(so_game
  ${OPENGL_gl_LIBRARY}
  ${OPENGL_glu_LIBRARY}
  ${GLUT_glut_LIBRARY}
  m
 )
```

This adds the to the include path the content of the OPENGL_INCLUDE_DIRS and GLUT_INCLUDE_DIRS set by the find package

CMake Example

```
cmake_minimum_required (VERSION 2.8.11)
project (so_game)

find_package(OpenGL REQUIRED)
find_package(GLUT REQUIRED)

include_directories(${OPENGL_INCLUDE_DIRS} ${GLUT_INCLUDE_DIRS})

add_executable(so_game
  helpers.c  image.c  so_game.c
  surface.c  vec3.c  vehicle.c  world.c
)

#tell the executable the libraries it needs
target_link_libraries(so_game
  ${OPENGL_gl_LIBRARY}
  ${OPENGL_glu_LIBRARY}
  ${GLUT_glut_LIBRARY}
  m
 )
```

This defines an executable **target**, named **`so_game`** built from the source files **`helpers.c`** … till **`world.c`**.
The dependencies between the objects, the sources and the header files are determined automatically by using the compiler toolchain (**`gcc -MM`**, in our case)

CMake Example

```
cmake_minimum_required (VERSION 2.8.11)
project (so_game)

find_package(OpenGL REQUIRED)
find_package(GLUT REQUIRED)

include_directories(${OPENGL_INCLUDE_DIRS} ${GLUT_INCLUDE_DIRS})

add_executable(so_game
  helpers.c  image.c  so_game.c
  surface.c  vec3.c  vehicle.c  world.c
)
```

```
#tell the executable the libraries it needs
target_link_libraries(so_game
  ${OPENGL_gl_LIBRARY}
  ${OPENGL_glu_LIBRARY}
  ${GLUT_glut_LIBRARY}
  m
 )
```

This specifies that to generate the target **`so_game`**, certain libraries are needed. Each item in the list will result in a flag **`-l <name>`** appended to the command line

CMake: Targets

A target is a binary executable or library.

```
add_executable (<name> <files…>)
```

adds an executable named **`<name>`**

```
add_library(<name> [SHARED] <files…>)
```

adds a library, if **`SHARED`** is omitted the library is static

to specify link dependancies for a target use

```
target_link_libraries (target_name <libraries…>)
```

where **`<libraries>`** are the target names of the libraries

CMake: Syntax

CMake relies on variables that can be lists or atoms. Variables are global.

Variables are expanded with

```
${<variable_name>}
```

Variables can be assigned with

```
set (<variable name> value) //plain var
set (<variable_name> <value1> <value2> … <value N>)//list
set (<variable_name> ${<variable_name>} <value>) //appends value to a list
```

Conditional constructs such as if/if-else are supported, and they can enclose all constructs.

To handle projects with nested folders, in the higher level CMake use

```
add_subdirectory(<folder_name>)
```

And in folder name put another `CMakeLists.txt` that specifies the targets in that folder.

Exercises

1. given the the files stored in the exercise folder of this lesson, write a CMakeLists.txt file that compiles them

2. create your own git account on one of the following public services

   - Gitlab
   - Github
   - Bitbucket

and push a robot_programming repo, where you will add the files and the CMakeLists you created