Laboratorio di Programmazione

Prof. Marco Bertini
marco.bertini@unifi.it
http://www.micc.unifi.it/bertini/

How the compiler works

Programs and libraries

The compiler

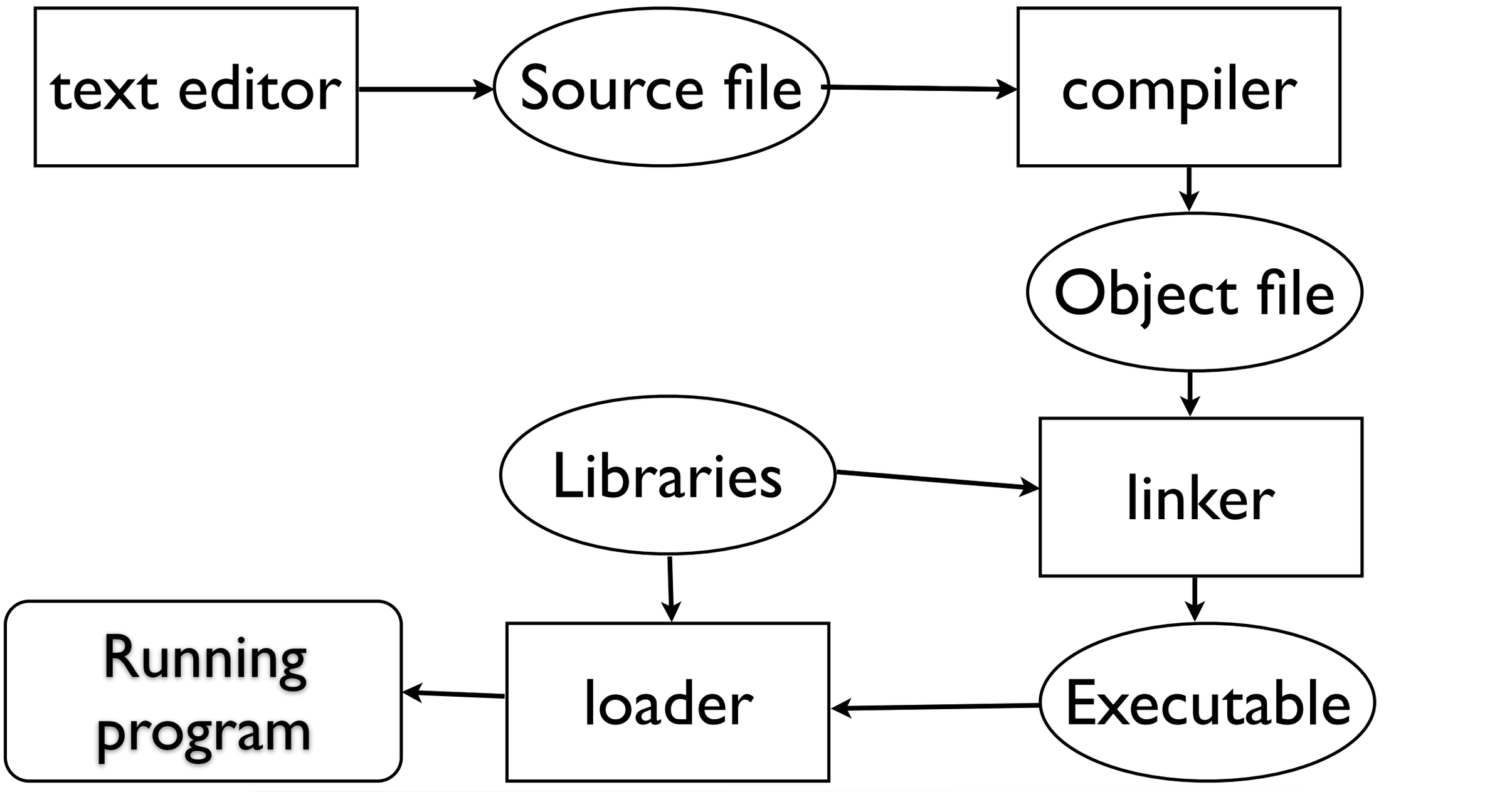
"In C++, everytime someone writes ">> 3" instead of "/ 8", I bet the compiler is like, "OH DAMN! I would have never thought of that!"

- Jon Shiring (Call of Duty 4 / MW2)

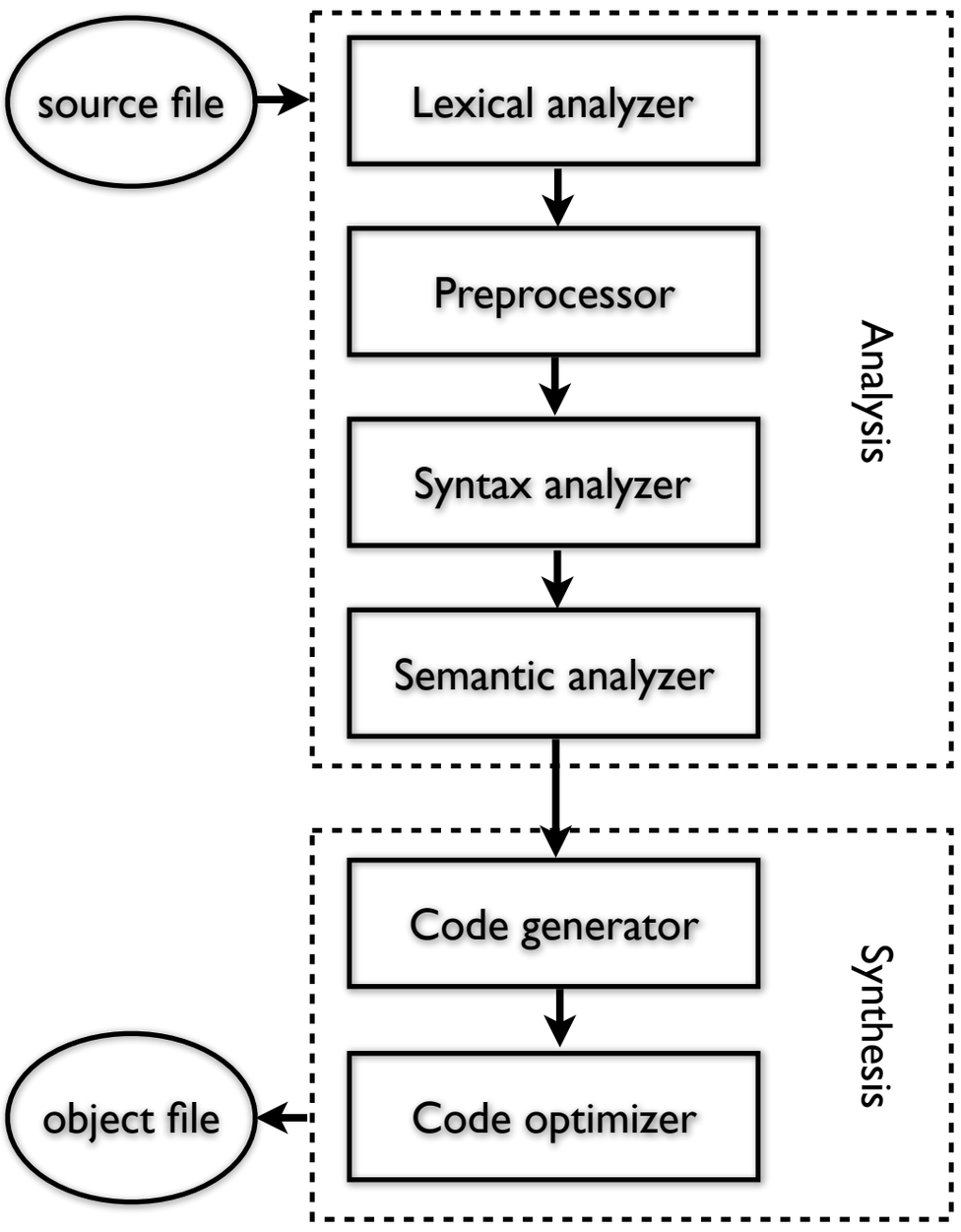
What is a compiler ?

- A compiler is a computer program (or set of programs) that translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine code).
- A compiler typically performs: lexical analysis (tokenization), preprocessing, parsing, semantic analysis, code generation, and code optimization.

From source code to a running program



From source to object file



- Lexical analysis: breaks the source code text into small pieces called tokens. Each token is a single atomic unit of the language, for instance a keyword, identifier or symbol name.
- Preprocessing: in C/C++ macro substitution and conditional compilation
- Syntax analysis: token sequences are parsed to identify the syntactic structure of the program.
- Semantic analysis: semantic checks such as type checking (checking for type errors), or object binding (associating variable and function references with their definitions), or definite assignment (requiring all local variables to be initialized before use), rejecting incorrect programs or issuing warnings
- Code generation: translation into the output language, usually the native machine language of the system. This involves resource and storage decisions (e.g. deciding which variables to fit into registers and memory), and the selection and scheduling of appropriate machine instructions and their associated addressing modes. Debug data may also need to be generated to facilitate debugging.
- Code optimization: transformation into functionally equivalent but faster (or smaller) forms.

How the compilation works

- Consider the following C++ program (e.g. stored in hello.cpp):

```
#include <iostream>

#define ANSWER 42

using namespace std;
// this is a C++ comment
int main() {
   cout << "The Answer to the Ultimate Question of Life,
the Universe, and Everything is " << ANSWER << endl;
   return 0;
}
```

How the compilation works

- Consider
stored ir

```
#include <

#define AN

using name
// this is
int main()
   cout <<
the Universe, and Everything is " << ANSWER << endl;
   return 0;
}
```

Check the result of the preprocessor using:

```
g++ -E hello.cpp
```

the iostream file is included at the beginning of the output, then there's the code without comments and with substituted define.

Assembly/object creation

- Check the assembly output of the program with:

```
g++ -S hello.cpp
```

- The object file is created with:

```
g++ -c hello.cpp
```

Assembly/object creation

- Check the assembly output of the program with:

```
g++ -S hello.cpp
```

- The object file is created with:

```
g++ -c hello.cpp
```

Performs all the compilation steps (also preprocessing)

Optimize and debug

- Add debug information to the output: it will help when debugging a program: when using g++ add the `-g` flag
- Request g++ optimization with the flags `-0x` (x=1...3) for fast execution or `-0s` for optimized size

Optimize and debug

Always use it when developing and testing a program !

- Add debug information to the output: it will help when debugging a program: when using g++ add the `-g` flag
- Request g++ optimization with the flags `-0x` (x=1...3) for fast execution or `-0s` for optimized size

Optimize and debug

Always use it when developing and testing a program !

- Add debug information to the output: it will help when debugging a program:
  when using g++ add the `-g` flag
- Request g++ optimization with the flags `-0x` (x=1...3)
  optimized siz

Compilation becomes slower and slower...
Typically optimization is set when releasing a program

Linking

- Use a linker like `ld` to link the libraries to the object file; on Ubuntu try:

```
ld -lstdc++ hello.o -o hello
```

- or use g++ linking (will add required standard libraries):

```
g++ hello.o -o hello
```

Linking

- Use a linker like `ld` to link the libraries to the object file; on Ubuntu try:

```
ld -lstdc++ hello.o -o hello
```

- or use *g++* linking (will add required standard libraries):

```
g++ hello.o -o hello
```

add `-v` to *g++* linking to see what's going on with `ld`

Linking

- The linker will merge the object files of various sources, e.g. if the program was split in more than one translation unit
- You must tell where object files and libraries are stored
  - the linker will check some default directories for libraries

How the IDE works

Managing build

- An IDE like CLion manages projects and how they are built.
- It creates the instructions for compiler and linker, managing the compilation of all required files and linking of all required libraries
- A common language is that of Makefile
- CLion creates Makefile programs from CMake programs

CMake

- Makefiles are O.S. dependent, so there is need to adapt them to different systems
- CMake creates Makefiles that are specific for each platform and system
- CLion creates CMake files from which Makefile files are created

Makefile

- Standard language for defining a compilation and linking process
- The make program understands if there is need to compile a source or if the last compilation is up-to-date w.r.t. the source code.
- It can contain different actions, like cleaning, building, installing

CLion

- Each time we build our project with Clion it:
  - checks that CMake is up-to-date or if needed uses CMake to create Makefile
  - executes Makefile, that in turns start the real compilation
- CLion builds targets outside the source code directory to keep it clean.

Libraries

What is a library ?

- A software library is a set of software functions used by an application program.
- Libraries contain code and data that provide services to independent programs.
- This encourages the sharing and changing of code and data in a modular fashion, and eases the distribution of the code and data.

Using libraries in C/C++

- To use a library in C/C++ you need to:
  1. Include the headers that provide the prototypes of functions and classes that you need in your code
  2. Tell the linker where are the library files (and which files - if the library is composed by more than one) that are needed by your code
- In C++ some libraries are made only by header files... e.g. template-based libraries.

The C++ Standard Library

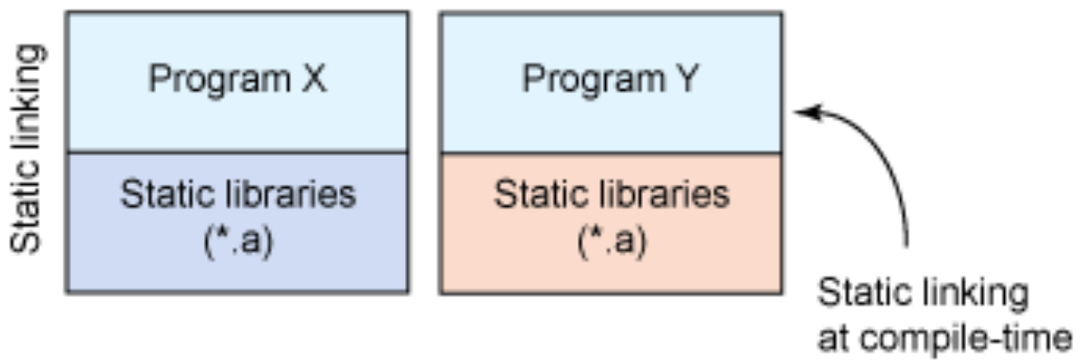
- In C++, the C++ Standard Library is a collection of classes and functions, which are written in the core language and part of the C++ ISO Standard itself.

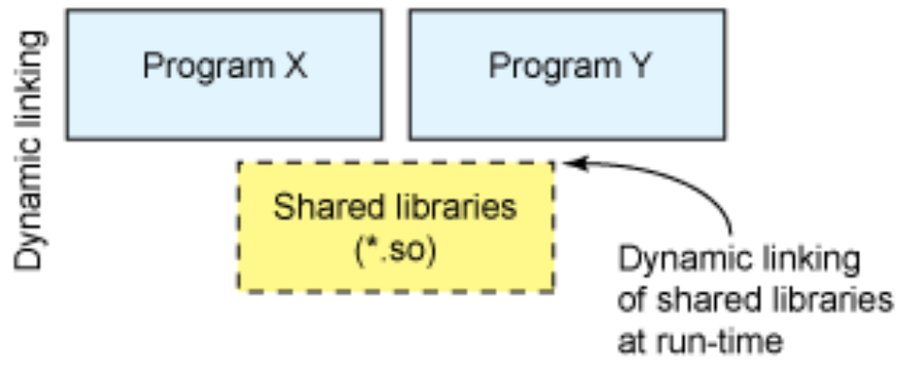
- The C++ Standard Library provides
  - several generic containers, functions to utilise and manipulate these containers;
  - generic strings and streams (including interactive and file I/O);
  - support for some language features, and math.

Types of libraries

- O.S.es like Linux, OS X and Windows support two types of libraries, each with its own advantages and disadvantages:
- The static library contains functionality that is bound to a program statically at compile time.
- The dynamic/shared library is loaded when an application is loaded and binding occurs at run time.
- In C/C++ you also have header files with the prototypes of the functions/classes that are provided by the library

Static vs. Dynamic linking

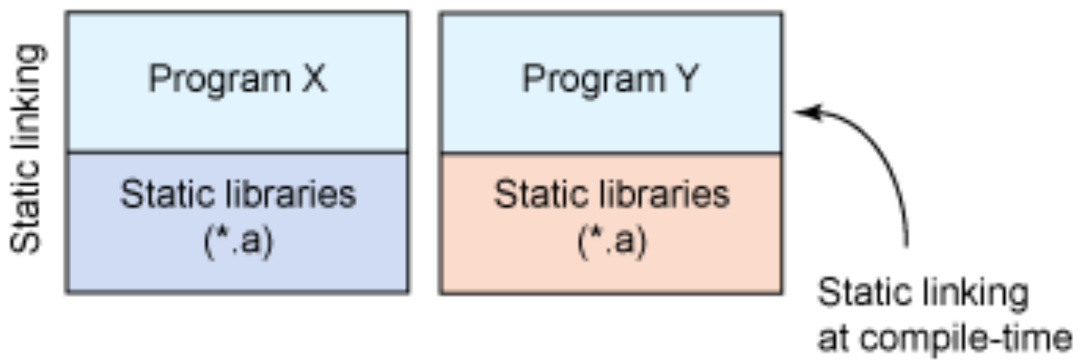


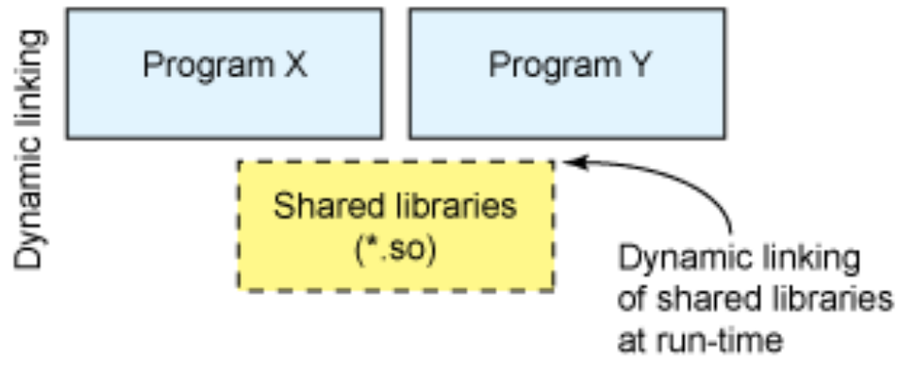


- To check if a program is statically or dynamically linked, and see what dynamic libraries are linked use `ldd` (Linux) or `otool` (OS X):

```
[ian@echidna ~]$ ldd /sbin/sln /sbin/ldconfig /bin/ln
/sbin/sln:
        not a dynamic executable
/sbin/ldconfig:
        not a dynamic executable
/bin/ln:
        linux-vdso.so.1 =>  (0x00007fff644af000)
        libc.so.6 => /lib64/libc.so.6 (0x00000037eb800000)
        /lib64/ld-linux-x86-64.so.2 (0x00000037eb400000)
```

Static vs. Dynamic linking



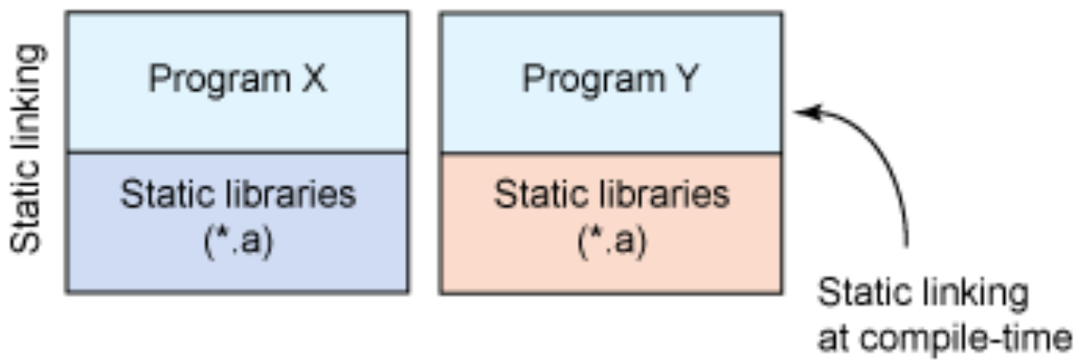


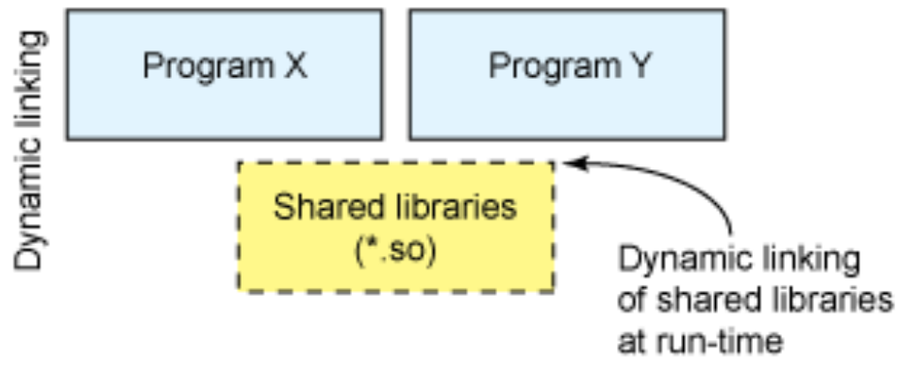
- To check if a program is statically or dynamically linked, and see what dynamic libraries are linked use `ldd` (Linux) or `otool` (OS X):

```
[ian@echidna ~]$ ldd /sbin/sln /sbin/ldconfig /bin/ln
/sbin/sln:
        not a dynamic executable
/sbin/ldconfig:
        not a dynamic executable
/bin/ln:
        linux-vdso.so.1 =>  (0x00007fff644af000)
        libc.so.6 => /lib64/libc.so.6 (0x00000037eb800000)
        /lib64/ld-linux-x86-64.so.2 (0x00000037eb400000)
```

Static linking

Static vs. Dynamic linking





- To check if a program is statically or dynamically linked, and see what dynamic libraries are linked use `ldd` (Linux) or `otool` (OS X):

```
[ian@echidna ~]$ ldd /sbin/sln /sbin/ldconfig /bin/ln
/sbin/sln:
        not a dynamic executable
/sbin/ldconfig:
        not a dynamic executable
/bin/ln:
        linux-vdso.so.1 =>  (0x00007fff644af000)
        libc.so.6 => /lib64/libc.so.6 (0x00000037eb800000)
        /lib64/ld-linux-x86-64.so.2 (0x00000037eb400000)
```

Static linking

Dynamic linking

Dynamic libraries

Static libraries

- Static libraries are simply a collection of ordinary object files; this collection is created using an archiver program (e.g. `ar` in *NIX systems).
  - Conventionally, static libraries end with the ".a" suffix (*NIX system) or ".lib" (Windows).
- Static libraries permit users to link to programs without having to recompile its code, saving recompilation time. There's no need to install libraries along with programs.
- With a static library, every running program has its own copy of the library.

Static libraries

- Statically linked programs incorporate only those parts of the library that they use (not the whole library!).
- To create a static library, or to add additional object files to an existing static library, use a command like this:
- `ar rcs my_library.a file1.o file2.o`
- The library file is used by the linker to create the final program file

Statically linked executables

- Statically linked executables contain all the library functions that they need to execute:
  - all library functions are linked into the executable.
- They are complete programs that do not depend on external libraries to run:
  - there is no need to install prerequisites.

Dynamic/Shared libraries

- Dynamic/Shared libraries are libraries that are loaded by programs when they start.
- They can be shared by multiple programs.
- Shared libraries can save memory, not just disk space. The O.S. can keep a single copy of a shared library in memory, sharing it among multiple applications. That has a pretty noticeable effect on performance.

Shared library versions

- Shared libraries use version numbers to allow for upgrades to the libraries used by applications while preserving compatibility for older applications.
- Shared objects have two different names: the *soname* and the *real name*. The *soname* consists of the prefix "lib", followed by the name of the library, a ".so" followed by another dot, and a number indicating the major version number (in OS X the dotted numbers precede the ".dylib" extension). The *real name* adds to the *soname* a period, a minor number, another period, and the release number. The last period and release number are optional.
- There is also the *linker name*, which may be used to refer to the soname without the version number information. Clients using this library refer to it using the linker name.

Why using library versions ?

- The major/minor number and release number support configuration control by letting you know exactly what version(s) of the library are installed. With a statically linked executable, there is some guarantee that nothing will change on you. With dynamic linking, you don't have that guarantee.

- What happens if a new version of the library comes out? Especially, what happens if the new version changes the calling sequence for a given function?

- Version numbers to the rescue: when a program is linked against a library, it has the version number it's designed for stored in it. The dynamic linker can check for a matching version number. If the library has changed, the version number won't match, and the program won't be linked to the newer version of library.

Shared libraries paths

- Since linking is dynamic the library files should be somewhere they can be found by the O.S. dynamic linker
  - e.g. /usr/lib or /usr/local/lib
  - It's possible to add other directories to the standard library paths (e.g. using LD_LIBRARY_PATH or DYLD_LIBRARY_PATH environment variables)

Dynamically linked executables

- Dynamically linked executables are smaller programs than statically linked executables:
  - they are incomplete in the sense that they require functions from external shared libraries in order to run.
- Dynamic linking permits a package to specify prerequisite libraries without needing to include the libraries in the package.
  - Dynamically linked executables can share one copy of a library on disk and in memory (at running time). Most programs today use dynamic linking.

Libraries and links

- Some technicalities about *NIX systems and libraries:
  generally, a *linker name* is a link to the *soname*.
  And the *soname* is a link to the *real name*.

```
lrwxr-xr-x     1 root     admin          10 20 Set 13:42 libpng.a -> libpng14.a
lrwxr-xr-x     1 root     admin          14 20 Set 13:42 libpng.dylib -> libpng14.dylib
lrwxr-xr-x     1 root     admin          11 20 Set 13:42 libpng.la -> libpng14.la
-rwxr-xr-x     1 root     admin      155204 20 Set 13:42 libpng14.14.dylib
-rw-r--r--     1 root     admin      209512 20 Set 13:42 libpng14.a
lrwxr-xr-x     1 root     admin          17 20 Set 13:42 libpng14.dylib -> libpng14.14.dylib
```

Libraries and links

- Some technicalities about *NIX systems and libraries:
  generally, a *linker name* is a link to the *soname*.
  And the *soname* is a link to the *real name*.

Linker names

```
lrwxr-xr-x    1 root    admin        10 20 Set 13:42 libpng.a -> libpng14.a
lrwxr-xr-x    1 root    admin        14 20 Set 13:42 libpng.dylib -> libpng14.dylib
lrwxr-xr-x    1 root    admin        11 20 Set 13:42 libpng.la -> libpng14.la
-rwxr-xr-x    1 root    admin    155204 20 Set 13:42 libpng14.14.dylib
-rw-r--r--    1 root    admin    209512 20 Set 13:42 libpng14.a
lrwxr-xr-x    1 root    admin        17 20 Set 13:42 libpng14.dylib -> libpng14.14.dylib
```

Libraries and links

- Some technicalities about *NIX systems and libraries:
  generally, a *linker name* is a link to the *soname*.
  And the *soname* is a link to the *real name*.

Linker names

```
lrwxr-xr-x    1 root    admin        10 20 Set 13:42 libpng.a -> libpng14.a
lrwxr-xr-x    1 root    admin        14 20 Set 13:42 libpng.dylib -> libpng14.dylib
lrwxr-xr-x    1 root    admin        11 20 Set 13:42 libpng.la -> libpng14.la
-rwxr-xr-x    1 root    admin    155204 20 Set 13:42 libpng14.14.dylib
-rw-r--r--    1 root    admin    209512 20 Set 13:42 libpng14.a
lrwxr-xr-x    1 root    admin        17 20 Set 13:42 libpng14.dylib -> libpng14.14.dylib
```

soname

Libraries and links

- Some technicalities about *NIX systems and libraries:
  generally, a *linker name* is a link to the *soname*.
  And the *soname* is a link to the *real name*.

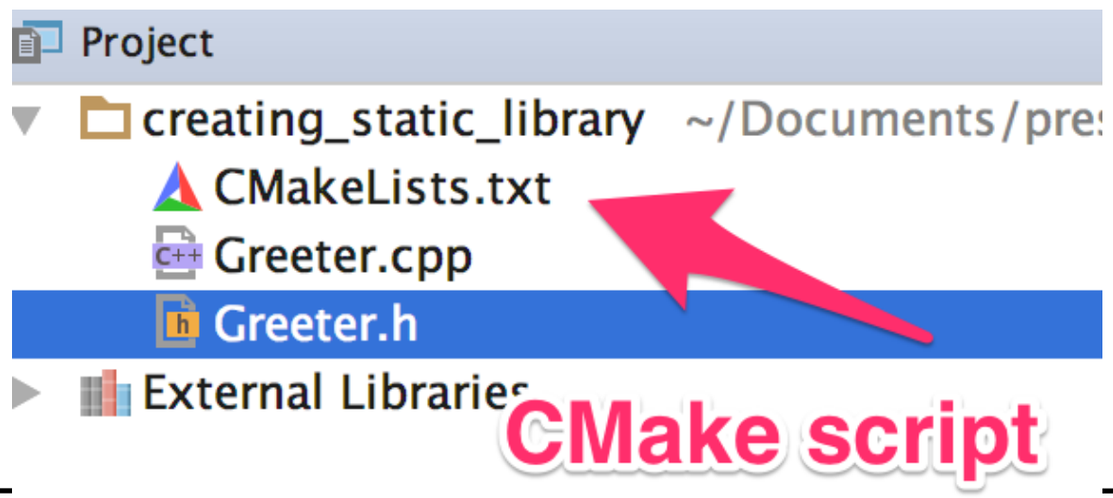
How CLion manages a build

CLion and CMake

- A CLion project must handle the compilation of many files, written by the programmer, and use external libraries.
- CLion uses CMake, a build system that describes the whole compilation and linking process using a specific CMake language.
- CMake then produces platform-specific scripts, that guide the compiler and linker, e.g. using Makefile language
  - CMake is a meta-language

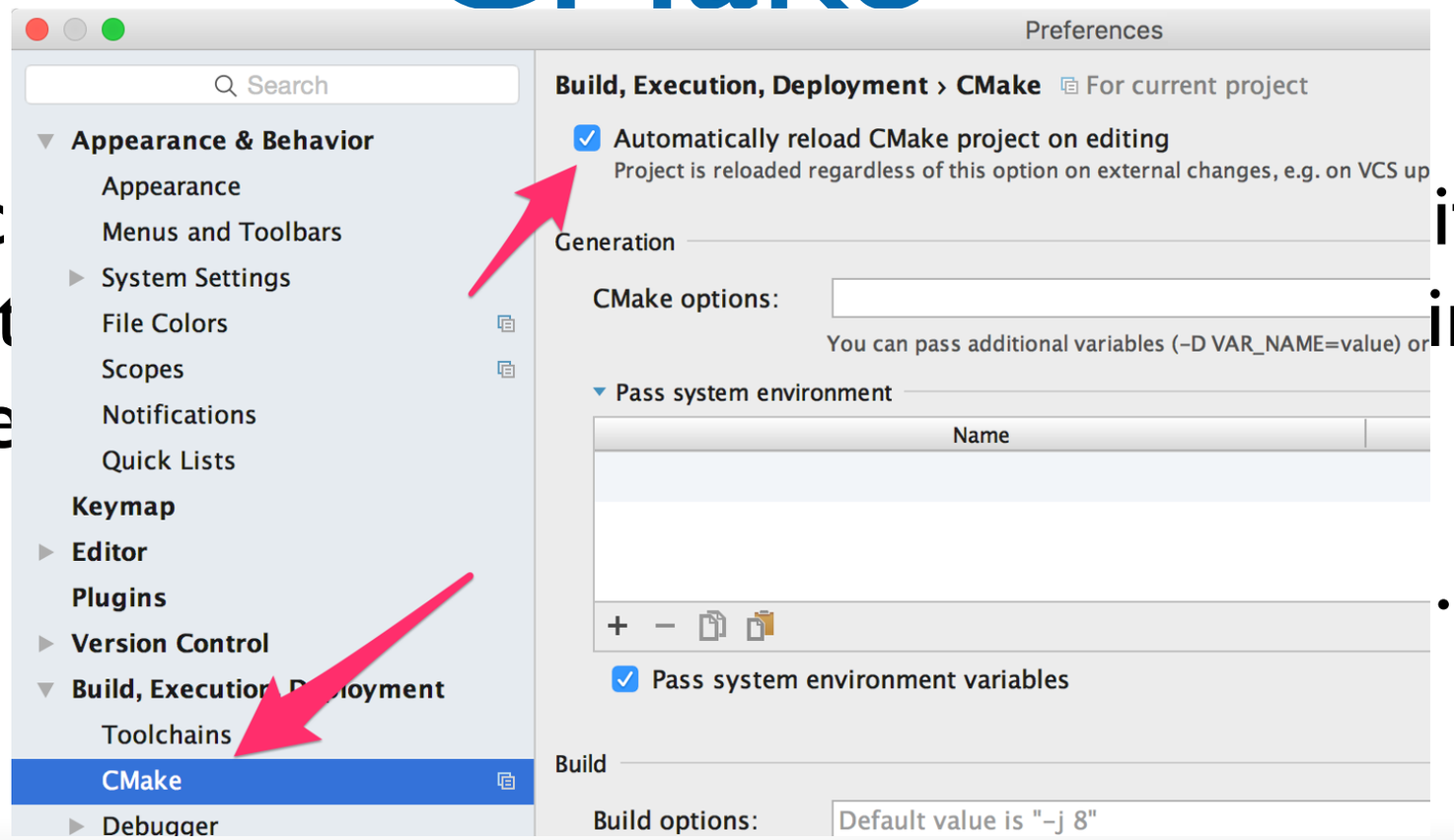
CMake

- The script used to generate the platform specific script that guides the build process is defined in CMakeLists.txt file.
- CLion updates the file as we add more .h and .cpp files to the project…
- …but we need to add manually instructions to perform more complex operations such as using and producing libraries

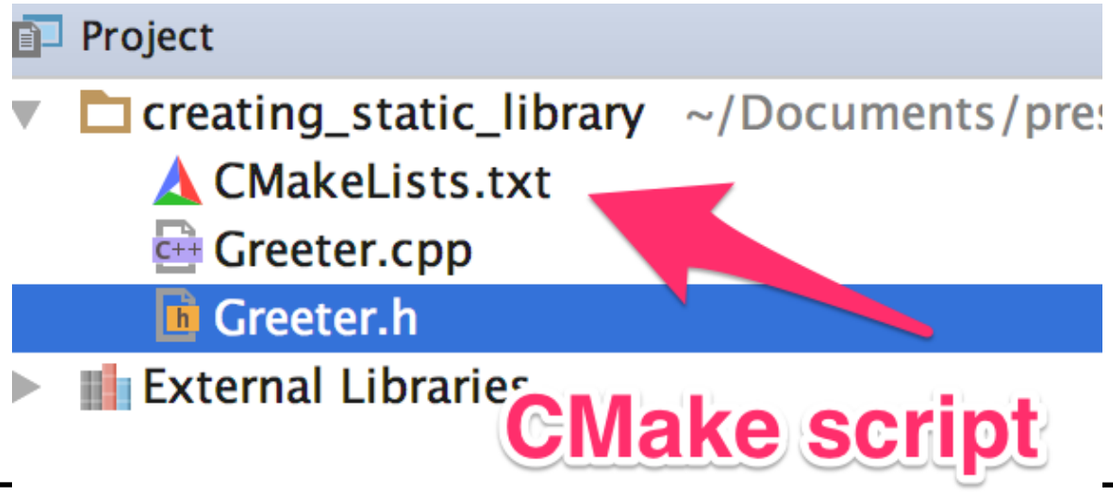


CMake

- The sc ific script t in CMake
- CLion .cpp files to the

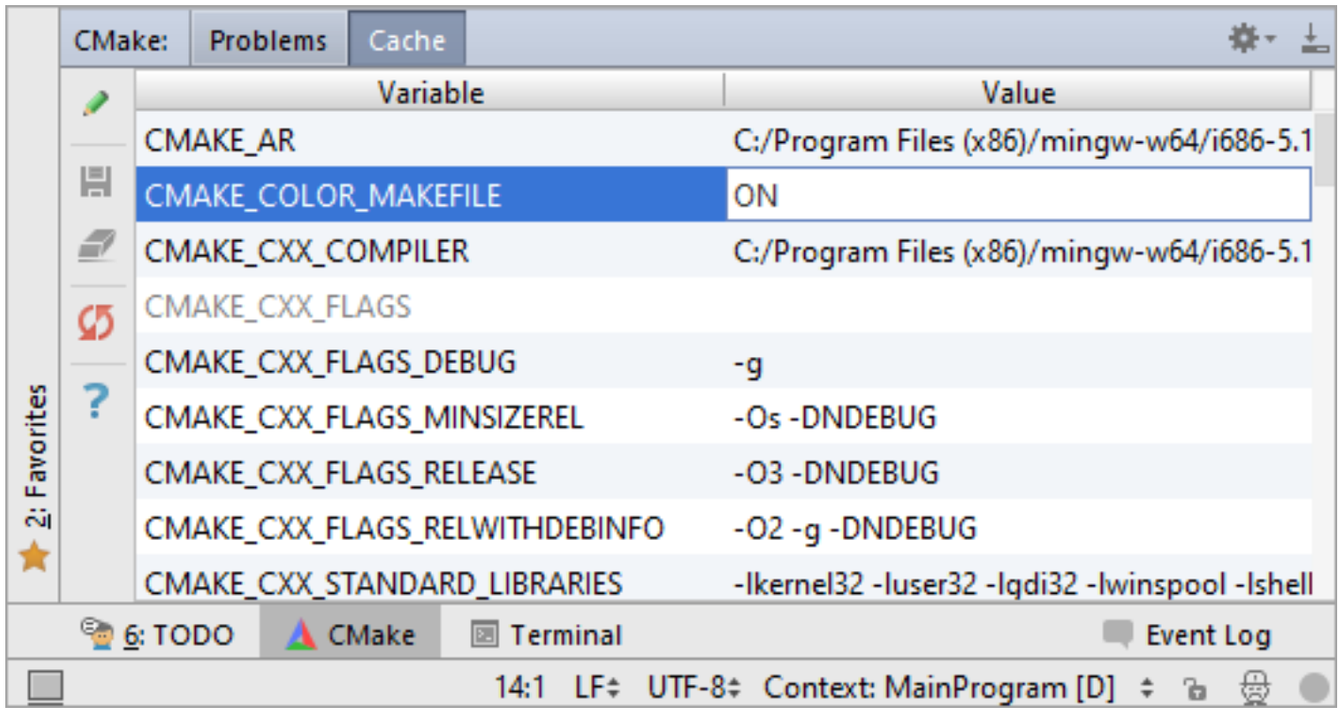


When we modify CMakeLists.txt we need to reload it in CLion, or we can set auto-reload.



CMake

- When you run CMake for the first time, what it will do is assemble the so-called CMake cache, collecting the variables and settings found in the system and storing them so it doesn't have to regenerate them later on.
- In most cases, you should not worry about this cache, but if you want to fine-tune it, you can. CLion gives you an editor window where you can view and edit the values:

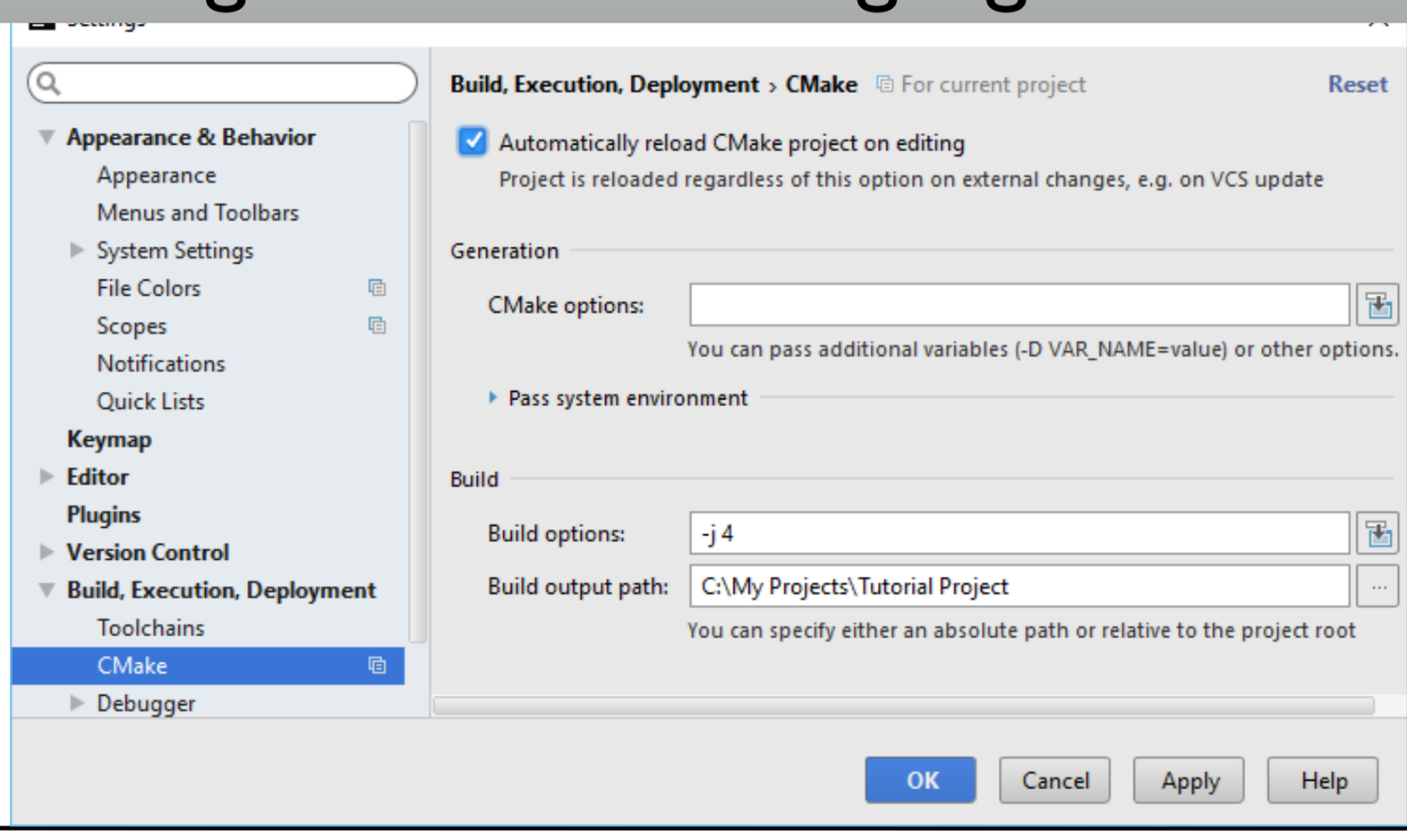


Compilation results

If there is need to locate the generated CMake files:

On the main menu, choose `Tools | CMake | Show Generated CMake Files in Explorer/Finder`

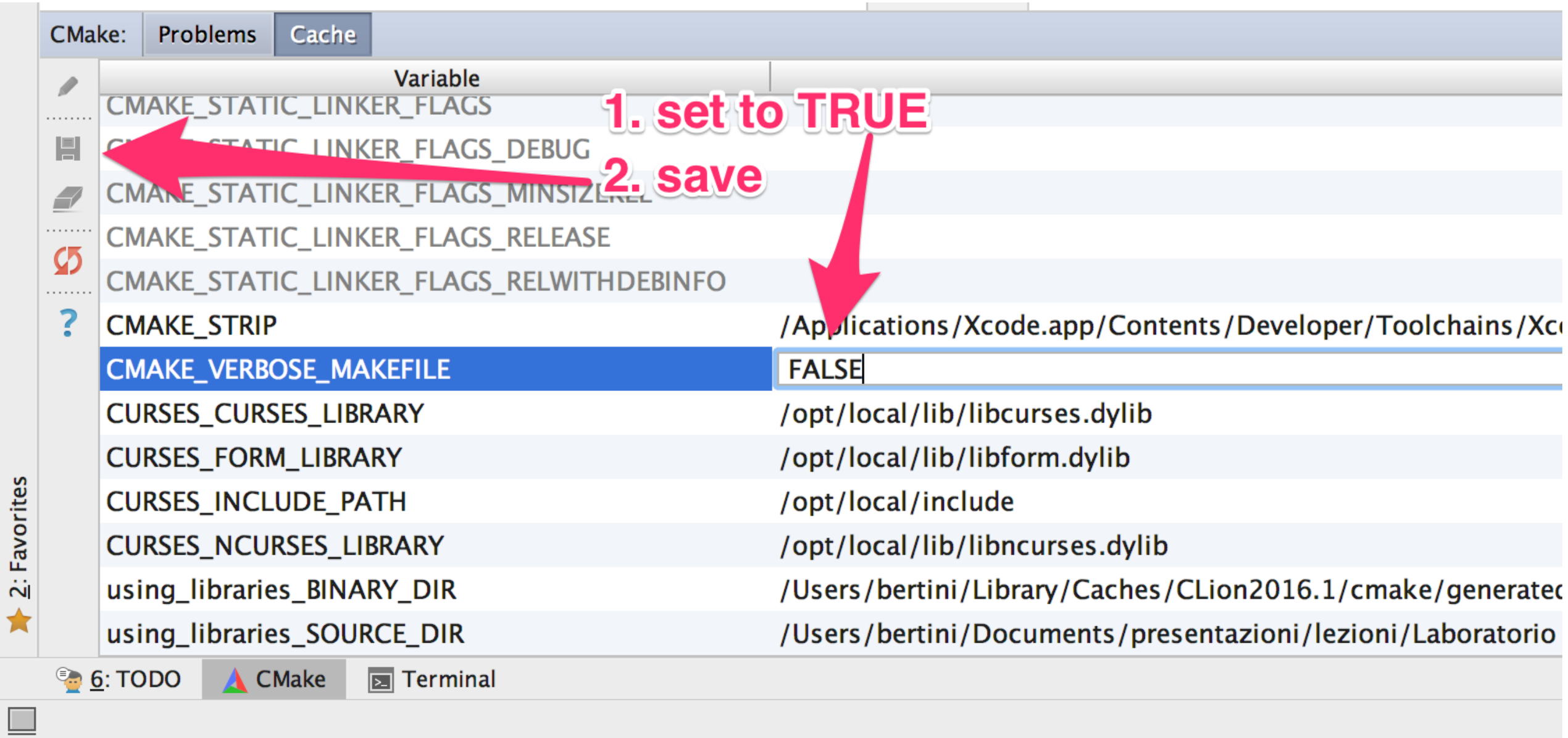
CLion will open the Explorer/Finder with the folder of generated files highlighted.



Use libraries in CLion

- The build process in CLion is based on CMake:
  - we need to work on the CMakeLists.txt file that contains the instructions used to manage the project
  - set tre verbosity of build to TRUE to better observe what happens
- CMake (most of the times) will find the desired libraries for us, adding path to include, library files and names of libraries to be linked

Use libraries in CLion



Use libraries in CLion

CMakeLists.txt × | main.cpp ×

```
1   cmake_minimum_required(VERSION 3.5)
2   project(using_libraries)
3
4   set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11" )
5
6   set(SOURCE_FILES main.cpp)
7   add_executable(using_libraries ${SOURCE_FILES})
8
9   set(CURSES_NEED_NCURSES, TRUE)
10  find_package( Curses REQUIRED )
11  include_directories( ${CURSES_INCLUDE_DIRS} )
12  target_link_libraries( using_libraries ${CURSES_LIBRARIES} )
```

1. We want to use either Curses or NCurses
2. We need the library, if not available abort
3. Add the path of the include files (-I)
4. Add path to libraries (-L) and library names (-l)

These options are equivalent to manually adding to command line -I, -L and -l

Use libraries in CLion

Messages Build

```
/Applications/CLion.app/Contents/bin/cmake/bin/cmake -E cmake_progress_start /Users/bertini/Library/Caches/CLion2016.1
 .1/cmake/generated/usi                                  /a1dbb825/Debug/CMakeFiles/progress.marks
/Applications/Xcode.app/Contents/Developer/usr/bin/make -f CMakeFiles/Makefile2 all
/Applications/Xcode.app/Contents/Developer/usr/bin/make -f CMakeFiles/using_libraries.dir/build.make CMakeFiles/using_
cd /Users/bertini/Library/Caches/CLion2016.1/cmake/generated/using_libraries-a1dbb825/a1dbb825/Debug && /Applications/
 "/Users/bertini/Documents/presentazioni/lezioni/Laboratorio di Programmazione 2015-2016/workspace/using_libraries" "/
  /Users/bertini/Library/Caches/CLion2016.1/cmake/generated/using_libraries-a1dbb825/a1dbb825/Debug /Users/bertini/Lib
 /Users/bertini/Library/Caches/CLion2016.1/cmake/generated/using_libraries-a1dbb825/a1dbb825/Debug/CMakeFiles/using_l
Scanning dependencies of target using_libraries
/Applications/Xcode.app/Contents/Developer/usr/bin/make -f CMakeFiles/using_libraries.dir/build.make CMakeFiles/using_
[ 50%] Building CXX object CMakeFiles/using_libraries.dir/main.cpp.o
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/c++    -I/opt/local/include    -s
 "/Users/bertini/Documents/presentazioni/lezioni/Laboratorio di Programmazione 2015-2016/workspace/using_libraries/ma
[100%] Linking CXX executable using_libraries
/Applications/CLion.app/Contents/bin/cmake/bin/cmake -E cmake_link_script CMakeFiles/using_libraries.dir/link.txt --ve
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/c++    -std=c++11 -g -Wl,-searc
/opt/local/lib/libcurses.dylib /opt/local/lib/libform.dylib
[100%] Built target using_libraries
/Applications/CLion.app/Contents/bin/cmake/bin/cmake -E cmake_progress_start /Users/bertini/Library/Caches/CLion2016.1
```

1. Compiling
2. Linking

- Now we can use the library our program:

1. Include the required headers
2. The libraries are managed by Makefile

Use libraries in CLion

```
#include <ncurses.h>
#include <unistd.h>

int main() {

    initscr();
    noecho();
    curs_set(FALSE);

    mvprintw(10, 10, "Hello, world!");
    refresh();

    sleep(1);
    getch();
    endwin();
}
```

Use libraries in CLion

- Do not Run the program with the "Run" icon/command
- Open the terminal, `cd` to temp directory of the project (see in "Messages")
- execute the program

```
    initscr();
    noecho();
    curs_set(FALSE);

    mvprintw(10, 10, "Hello, world!");
    refresh();

    sleep(1);
    getch();
    endwin();
}
```

Use libraries in CLion

- Do not Run the program with the "Run" icon/command
- Open the terminal, cd to temp directory of the project (see in "Messages")
- execute the program

Messages Build

```
/Applications/CLion.app/Contents/bin/cmake/bin/cmake --build /Users/bertini/Library/Caches/CLion2016.1/cmake/generated/using_libraries-a1dbb825/a1dbb825/Debug
/Applications/CLion.app/Contents/bin/cmake/bin/cmake -H"/Users/bertini/Documents/presentazioni/lezioni/Laboratorio_di_Programmazione_2015-2016/workspace/using_li
 .1/cmake/generated/using_libraries-a1dbb825/a1dbb825/Debug --check-build-system CMakeFiles/Makefile.cmake 0
/Applications/CLion.app/Contents/bin/cmake/bin/cmake -E cmake_progress_start /Users/bertini/Library/Caches/CLion2016.1/cmake/generated/using_libraries-a1dbb825/a
.1/cmake/generated/using_libraries-a1dbb825/a1dbb825/Debug/CMakeFiles/progress.marks
/Applications/Xcode.app/Contents/Developer/usr/bin/make -f CMakeFiles/Makefile2 all
/Applications/Xcode.app/Contents/Developer/usr/bin/make -f CMakeFiles/using_libraries.dir/build.make CMakeFiles/using_libraries.dir/depend
cd /Users/bertini/Library/Caches/CLion2016.1/cmake/generated/using_libraries-a1dbb825/a1dbb825/Debug && /Applications/CLion.app/Contents/bin/cmake/bin/cmake -E c
```

```
    mvprintw(10, 10, "Hello, world!");
    refresh();

    sleep(1);
    getch();
    endwin();
}
```

Use libraries in CLion

- Do not Run the program with the "Run" icon/command
- Open the terminal, cd to temp directory of the project (see in "Messages")
- execute the program

Messages Build

```
/Applications/CLion.app/Contents/bin/cmake/bin/cmake --build /Users/bertini/Library/Caches/CLion2016.1/cmake/generated/using_libraries-a1dbb825/a1dbb825/Debug
/Applications/CLion.app/Contents/bin/cmake/bin/cmake -H"/Users/bertini/Documents/presentazioni/lezioni/Laboratorio_di_Programmazione_2015-2016/workspace/using_li
 .1/cmake/generated/using_libraries-a1dbb825/a1dbb825/Debug --check-build-system CMakeFiles/Makefile.cmake 0
/Applications/CLion.app/Contents/bin/cmake/bin/cmake -E cmake_progress_start /Users/bertini/Library/Caches/CLion2016.1/cmake/generated/using_libraries-a1dbb825/a
.1/cmake/generated/using_libraries-a1dbb825/a1dbb825/Debug/CMakeFiles/progress.marks
/Applications/Xcode.app/Contents/Developer/usr/bin/make -f CMakeFiles/Makefile2 all
/Applications/Xcode.app/Contents/Developer/usr/bin/make -f CMakeFiles/using_libraries.dir/build.make CMakeFiles/using_libraries.dir/depend
cd /Users/bertini/Library/Caches/CLion2016.1/cmake/generated/using_libraries-a1dbb825/a1dbb825/Debug && /Applications/CLion.app/Contents/bin/cmake/bin/cmake -E c
nirvana:using_libraries bertini$ cd /Users/bertini/Library/Caches/CLion2016.1/cmake/generated/using_libraries-a1dbb825/a1dbb825/Debug
nirvana:Debug bertini$ ls -la
total 104
drwxr-xr-x   8 bertini  staff    272 Apr 22 13:14 .
drwxr-xr-x   8 bertini  staff    272 Apr 22 12:58 ..
-rw-r--r--   1 bertini  staff  14179 Apr 22 13:05 CMakeCache.txt
drwxr-xr-x  15 bertini  staff    510 Apr 22 13:14 CMakeFiles
-rw-r--r--   1 bertini  staff   5426 Apr 22 13:05 Makefile
-rw-r--r--   1 bertini  staff   1386 Apr 22 12:58 cmake_install.cmake
-rwxr-xr-x   1 bertini  staff  13300 Apr 22 13:14 using_libraries
-rw-r--r--   1 bertini  staff   6437 Apr 22 13:05 us     ries.cbp
nirvana:Debug bertini$ ./using_libraries
nirvana:Debug bertini$
```

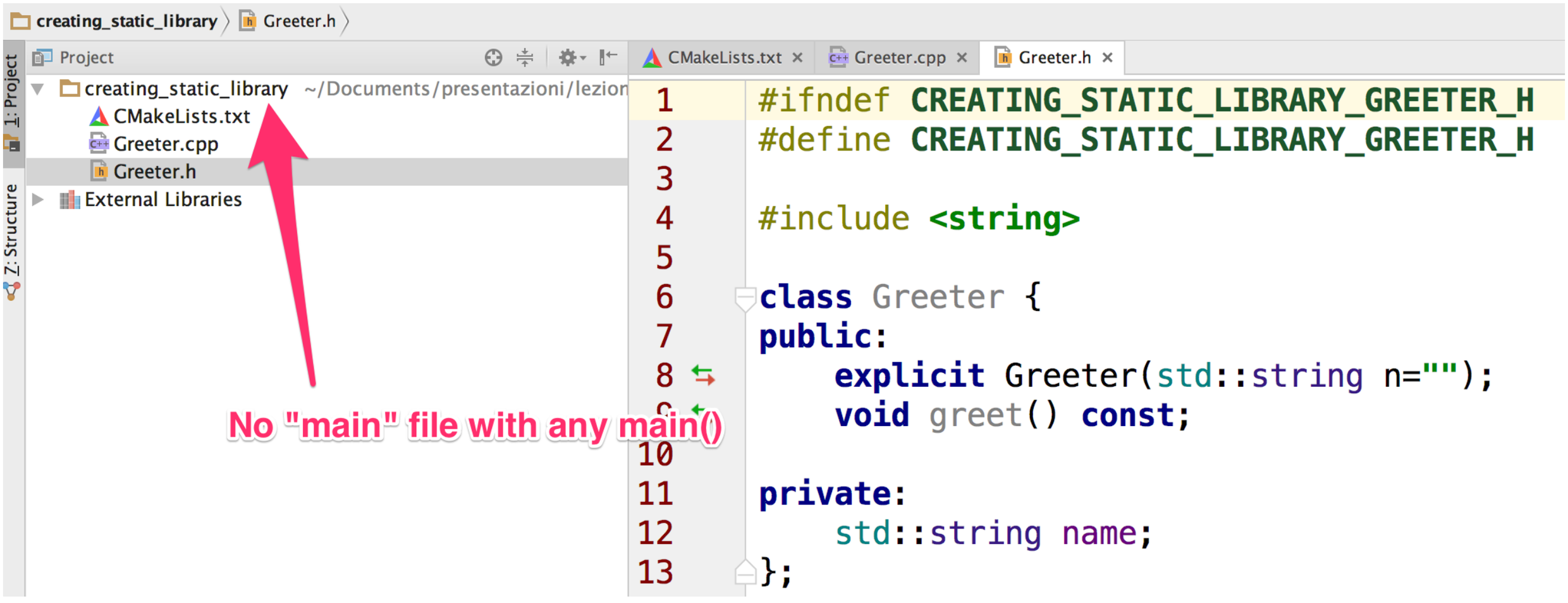
1. cd to directory
2. execute program

Creating and using a library

Writing a library

- There are basically two files that have to be written for a usable library:
- The first is a header file, which declares all the functions/classes/types exported by the library.
  - It will be included by the client in the code.
- The second is the definition of the functions/ classes to be compiled and placed as the shared object.
  - the object file created through compilation will be used by the linker, to create the library.

Creating a static library with CLion



Creating a static library with CLion

creating_static_library 〉 CMakeLists.txt 〉

Project

creating_static_library ~/Documents/presentazioni/lezion
- CMakeLists.txt
- Greeter.cpp
- Greeter.h

External Libraries

CMakeLists.txt × | Greeter.cpp × | Greeter.h ×

```
1  cmake_minimum_required(VERSION 3.5)
2  project(creating_static_library)
3
4  set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11")
5
6  set(SOURCE_FILES Greeter.cpp Greeter.h)
7  add_library(greeter STATIC ${SOURCE_FILES})
```

Set target to static library creation

Terminal

```
-rw-r--r--   1 bertini  staff   6877 Apr 22 16:18 creating_static_library.cbp
-rw-r--r--   1 bertini  staff  87576 Apr 22 16:18 libcreating_static_library.a
-rw-r--r--   1 bertini  staff  87576 Apr 22 16:19 libgreeter.a
nirvana:Debug bertini$ rm libcreating_static_library.a
nirvana:Debug bertini$ cd /Users/bertini/Library/Caches/CLion2016.1/cmake/generated/creating_static_library-cd6aeb7/cd6aeb7/Debug
nirvana:Debug bertini$ ls -la
total 248
drwxr-xr-x   8 bertini  staff    272 Apr 22 16:19 .
drwxr-xr-x   8 bertini  staff    272 Apr 22 13:26 ..
-rw-r--r--   1 bertini  staff  13625 Apr 22 16:18 CMakeCache.txt
drwxr-xr-x  16 bertini  staff    544 Apr 22 16:19 CMakeFiles
-rw-r--r--   1 bertini  staff   5351 Apr 22 16:18 Makefile
-rw-r--r--   1 bertini  staff   1400 Apr 22 13:26 cmake_install.cmake
-rw-r--r--   1 bertini  staff   6877 Apr 22 16:18 creating_static_library.cbp
-rw-r--r--   1 bertini  staff  87576 Apr 22 16:19 libgreeter.a
nirvana:Debug bertini$
```

.a file

Creating a dynamic library

creating_static_library › CMakeLists.txt

Project

creating_static_library ~/Documents/presentazioni/lezion
- CMakeLists.txt
- Greeter.cpp
- Greeter.h
- External Libraries

CMakeLists.txt × | Greeter.cpp × | Greeter.h ×

```
1  cmake_minimum_required(VERSION 3.5)
2  project(creating_static_library)
3
4  set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11")
5
6  set(SOURCE_FILES Greeter.cpp Greeter.h)
7  add_library(greeter SHARED ${SOURCE_FILES})
```

Add SHARED to the target

Terminal

```
-rw-r--r--   1 bertini  staff   6877 Apr 22 16:18 creating_static_library.cbp
-rw-r--r--   1 bertini  staff  87576 Apr 22 16:19 libgreeter.a
nirvana:Debug bertini$ ls
CMakeCache.txt                  CMakeFiles                      Makefile
nirvana:Debug bertini$ ls -la
total 296
drwxr-xr-x   9 bertini  staff    306 Apr 22 16:22 .
drwxr-xr-x   8 bertini  staff    272 Apr 22 13:26 ..
-rw-r--r--   1 bertini  staff  13625 Apr 22 16:22 CMakeCache.txt
drwxr-xr-x  16 bertini  staff    544 Apr 22 16:22 CMakeFiles
-rw-r--r--   1 bertini  staff   5351 Apr 22 16:22 Makefile
-rw-r--r--   1 bertini  staff   1400 Apr 22 13:26 cmake_install.cmak
-rw-r--r--   1 bertini  staff   6885 Apr 22 16:22 creating_static_library.cbp
-rw-r--r--   1 bertini  staff  87576 Apr 22 16:19 libgreeter.a
-rwxr-xr-x   1 bertini  staff  23108 Apr 22 16:22 libgreeter.dylib
nirvana:Debug bertini$
```

Dynamic library file

Deploy the library

- Deploying the library means to copy the header and library files to a position where they can be used to build other programs.
- CMake has specific instructions for this
- First let's reorganize our code to tell which header files are going to be deployed (and needed by other programs):

```
set(HEADER_FILES Greeter.h)
set(SOURCE_FILES Greeter.cpp ${HEADER_FILES})
```

Deploy the library files

- Then add the CMake instructions to tell where to copy the results of our compilation:

```
install(TARGETS greeter
        ARCHIVE DESTINATION /tmp/greeter/lib
        LIBRARY DESTINATION /tmp/greeter/lib)
install(FILES ${HEADER_FILES} DESTINATION /tmp/greeter/include)
```

- ARCHIVE indicates static library and LIBRARY indicates dynamic library destinations.

Deploying from CLion

- At present it's not possible to install from CLion
- Open a terminal in the same directory where CLion has built the library and issue:

```
make install
```

Deploying from CLion

```
nirvana:Debug bertini$ cd  /Users/bertini/Library/Caches/CLion2016.1/cmake/generated/creating_static_library-cd6aeb7/cd6aeb7/Debug
nirvana:Debug bertini$ ls
total 328
drwxr-xr-x  11 bertini  staff    374 Apr 22 17:
drwxr-xr-x   8 bertini  staff    272 Apr 22 13:26 ..
-rw-r--r--@  1 bertini  staff   6148 Apr 22 17:15 .DS_Store
-rw-r--r--   1 bertini  staff  13625 Apr 22 17:41 CMakeCache.txt
drwxr-xr-x  16 bertini  staff    544 Apr 22 17:42 CMakeFiles
-rw-r--r--   1 bertini  staff   7159 Apr 22 17:41 Makefile
-rw-r--r--   1 bertini  staff   3186 Apr 22 17:41 cmake_install.cmake
-rw-r--r--   1 bertini  staff  11145 Apr 22 17:41 creating_static_library.cbp
-rw-r--r--   1 bertini  staff     64 Apr 22 17:42 install_manifest.txt
-rw-r--r--   1 bertini  staff  87576 Apr 22 17:30 libgreeter.a
-rwxr-xr-x   1 bertini  staff  23108 Apr 22 17:41 libgreeter.dylib
nirvana:Debug bertini$ make install
[100%] Built target greeter
Install the project...
-- Install configuration: "Debug"
-- Up-to-date: /tmp/greeter/lib/libgreeter.dylib
-- Up-to-date: /tmp/greeter/include/Greeter.h
nirvana:Debug bertini$
```

CMake | Terminal | 0: Messages | 6: TODO

1. cd to output directory

2. make install

Deploying from CLion

- Alternatively add a new target like this:

```
add_custom_target(install_${PROJECT_NAME}
        make install
        DEPENDS greeter
        COMMENT "Installing ${PROJECT_NAME}")
```

- and build this new configuration.

Messages Build

```
/Applications/CLion.app/Contents/bin/cmake/bin/cmake --build /Users/bertini/Library/Caches/CL
[ 66%] Built target greeter
[100%] Installing creating_static_library
[100%] Built target greeter
Install the project...
-- Install configuration: "Debug"
-- Installing: /tmp/greeter/lib/libgreeter.dylib
-- Installing: /tmp/greeter/include/Greeter.h
Built target install_creating_static_library
```

Using a static library with CLion

- We need to tell the compiler where are the header files of the library
- We need to include the files in our client code
- We need to tell the linker where is the library file (“.a”) and the name of the library (remind the convention used !)
- CLion will use this information to create the required Makefile from CMake

Using a static library with CLion

- In CMakeLists.txt tell where to look for library files and includes:

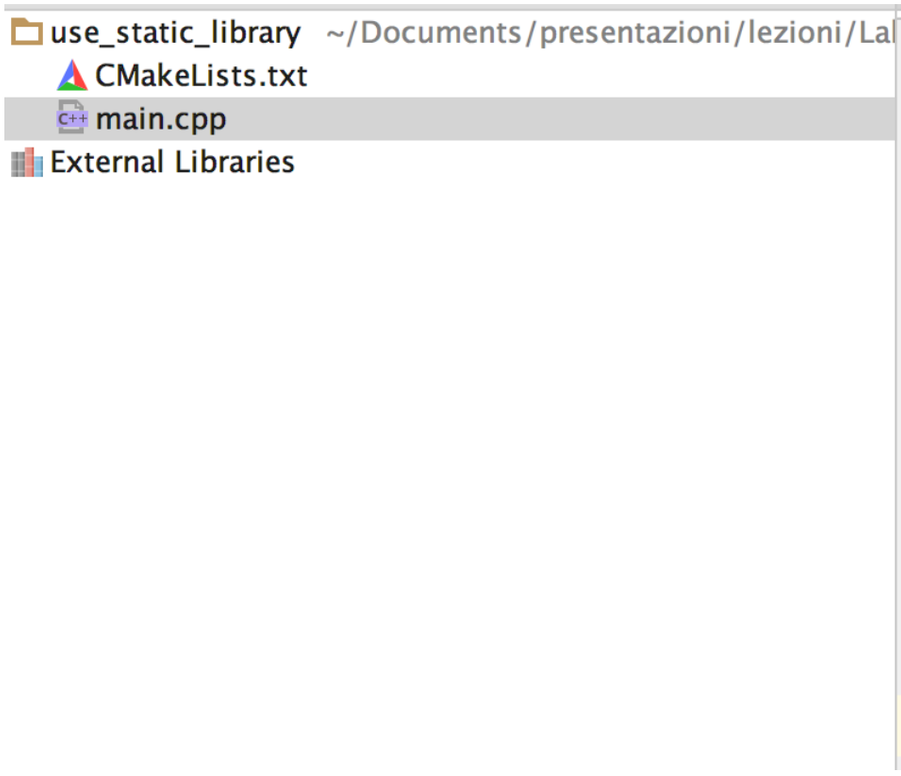
```
include_directories(/tmp/greeter/include)
link_directories(/tmp/greeter/lib)

add_executable(use_static_library ${SOURCE_FILES})
```

- Add the library name (no trailing lib):

```
target_link_libraries(use_static_library greeter)
```

Using a static library with CLion




```cpp
#include <iostream>
#include <string>

// include library header(s)
#include "Greeter.h"

int main() {
    std::string name = "Marco";
    Greeter g(name); // use library objects
    g.greet();
    return 0;
}
```

Using a dynamic library with CLion

- We need to tell the compiler where are the header files of the library
- We need to include the files in our client code
- We need to tell the linker where is the library file (“.so” / “.dylib”) and the name of the library (remind the convention used !)
- CLion will use this information to create the required Makefile from CMake

Using a dynamic library with CLion

- Use exactly the same CMake commands seen before
- If both static and dynamic libraries are in the same directory then CMake selects the static one as default.
  - Use full name with extension to specify which one to use (e.g. add .a/.dylib/.so).

Executing a dynamically linked program

- Remind that dynamically linked programs need to access the library (actually it is the dynamic linker that needs this)
- Either copy the library to a path used by the dynamic linker (check info of your O.S.) or copy it in the same directory of the executable-

References and sources

These slides are based on the following articles

Suggested reading: dynamic/ shared libraries

- Learn Linux, 101: Manage shared libraries: http://www.ibm.com/developerworks/linux/library/l-lpic1-v3-102-3/
- Anatomy of Linux dynamic libraries: http://www.ibm.com/developerworks/linux/library/l-dynamic-libraries/
- Dissecting shared libraries: http://www.ibm.com/developerworks/linux/library/l-shlibs/

Suggested reading: writing dynamic/shared libraries

- Program Library HOWTO http://www.linuxdoc.org/HOWTO/Program-Library-HOWTO/
- Shared objects for the object disoriented! http://www.ibm.com/developerworks/library/l-shobj/
- Writing DLLs for Linux apps http://www.ibm.com/developerworks/linux/library/