Laboratorio di Tecnologie dell'Informazione

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

How the compiler works

Programs and libraries

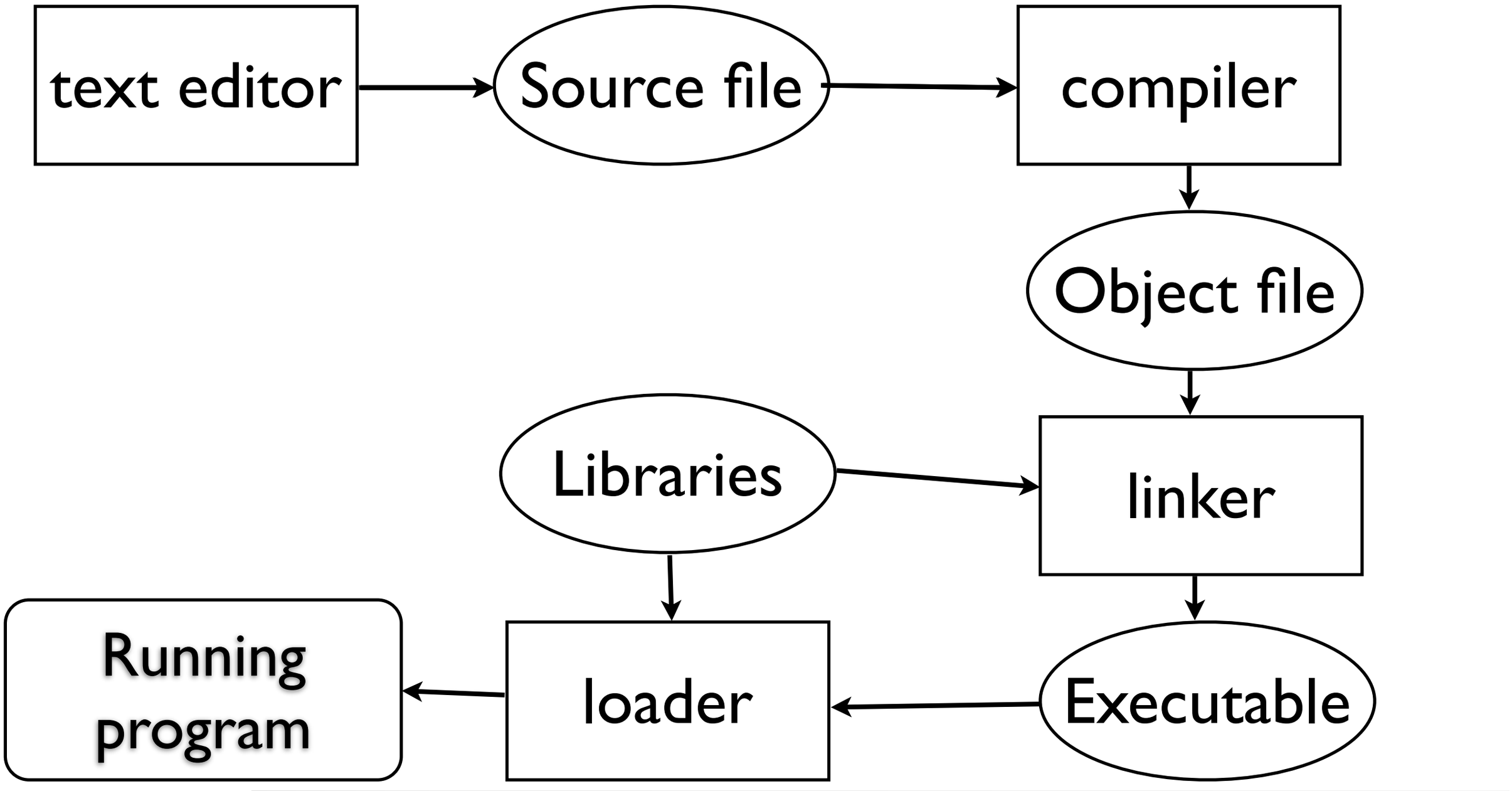
The compiler

"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)

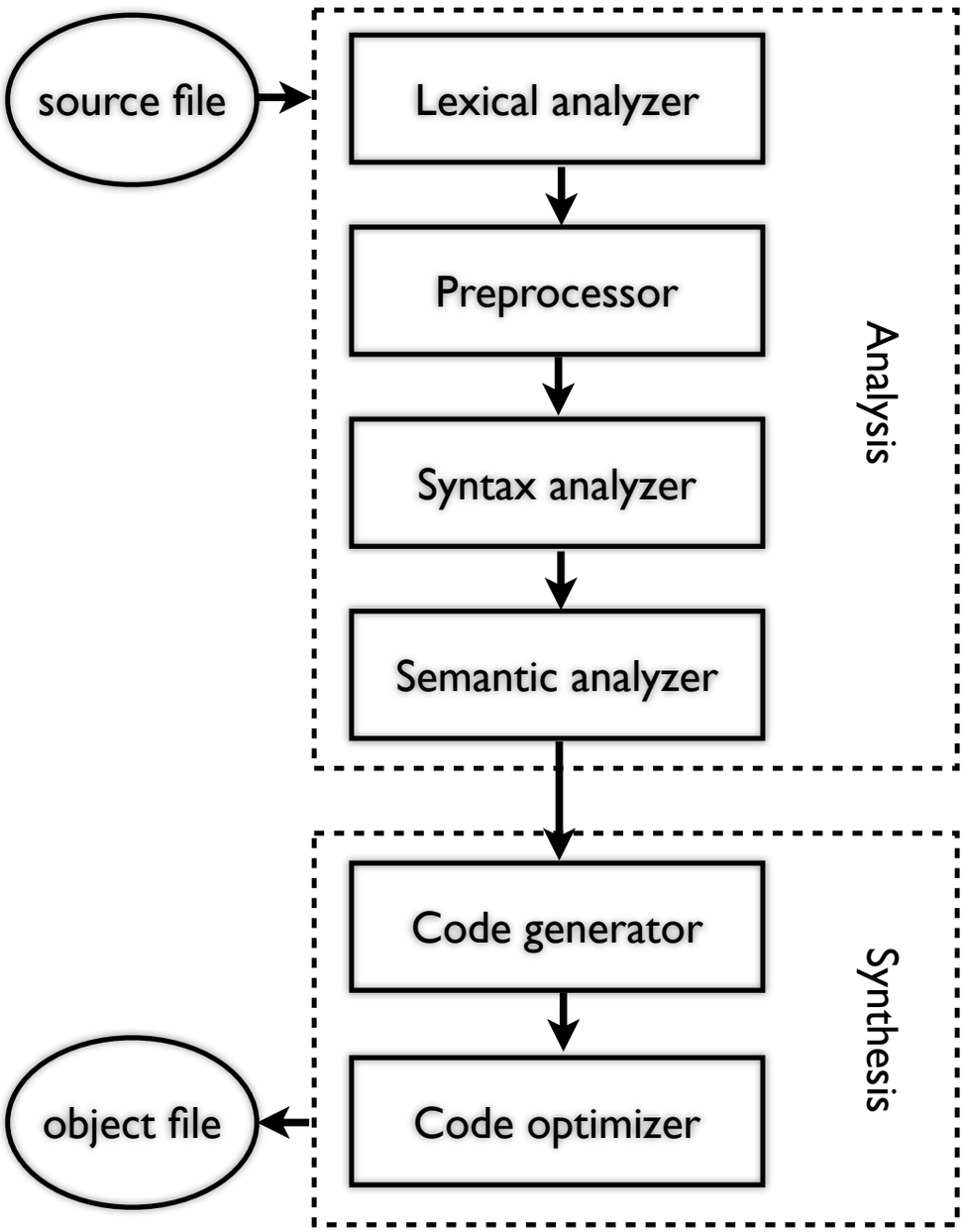
What is a compiler ?

- 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 i

```
#include <

#define AN

using name
// this is
int main()
     cout
Life, 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)

Linking

- Use a linker like ld to link the libraries to the object file:

```
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:

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

If you use OSX add:

```
-lSystem -lcrt1.10.6.o
```

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

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

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

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) for fa optimized size

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

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... more information when studying C++ templates.

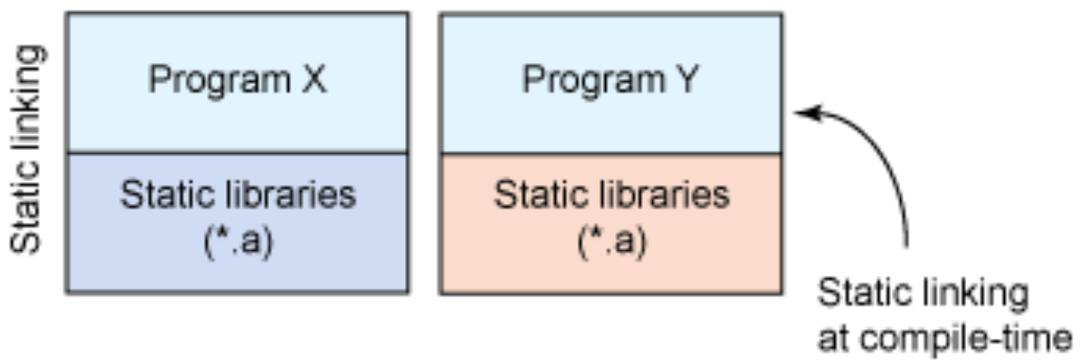
The C++ Standard Library

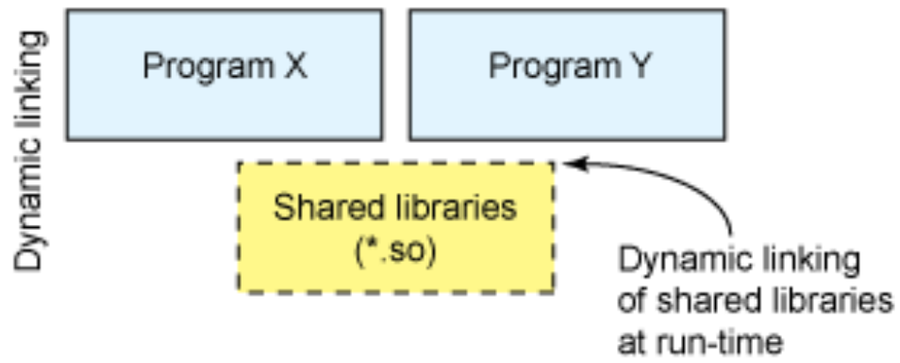
- In C++, the C++ Standard Library is a collection of classes and functions, which are written in the core language and part of the C++ ISO Standard itself.
- The C++ Standard Library provides
  - several generic containers, functions to utilise and manipulate these containers;
  - generic strings and 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

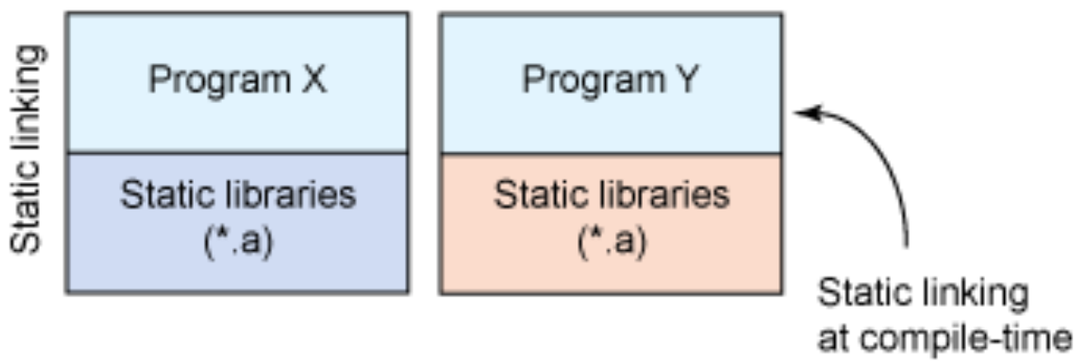


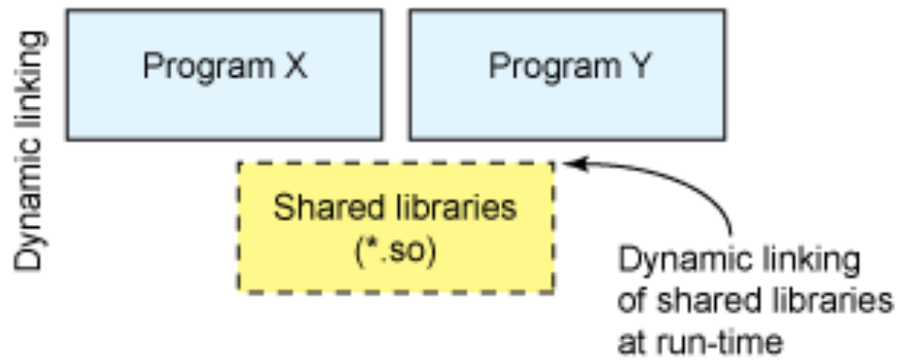


- To check if a program is statically or dynamically linked, and see what dynamic libraries are linked use `ldd` (Linux) or `otool` (OS X):

```
[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



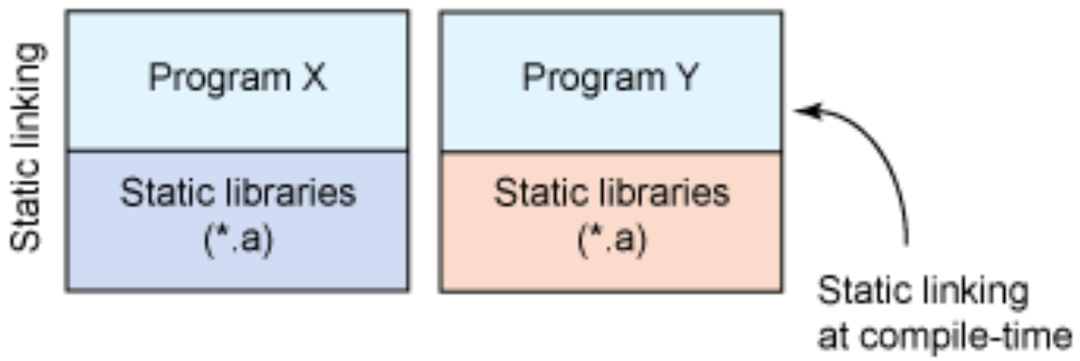


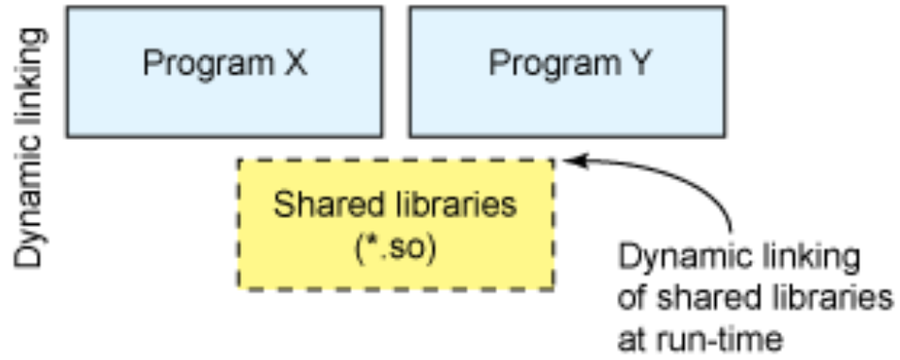
- To check if a program is statically or dynamically linked, and see what dynamic libraries are linked use `ldd` (Linux) or `otool` (OS X):

```
[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.
  - D.L. 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*.

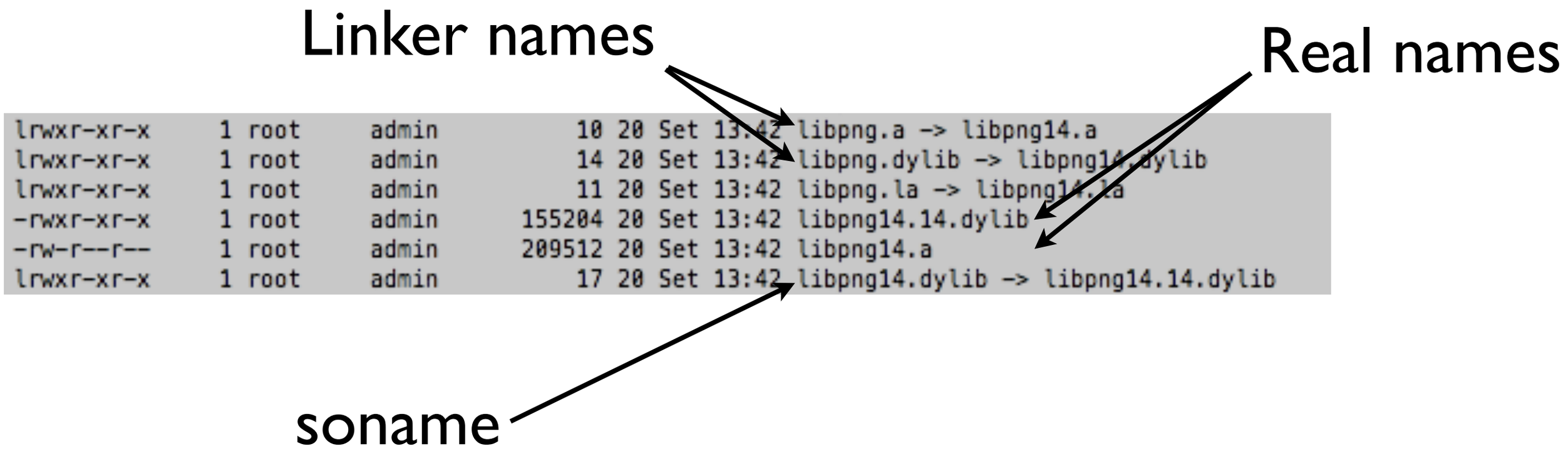
Linker names

```
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
```

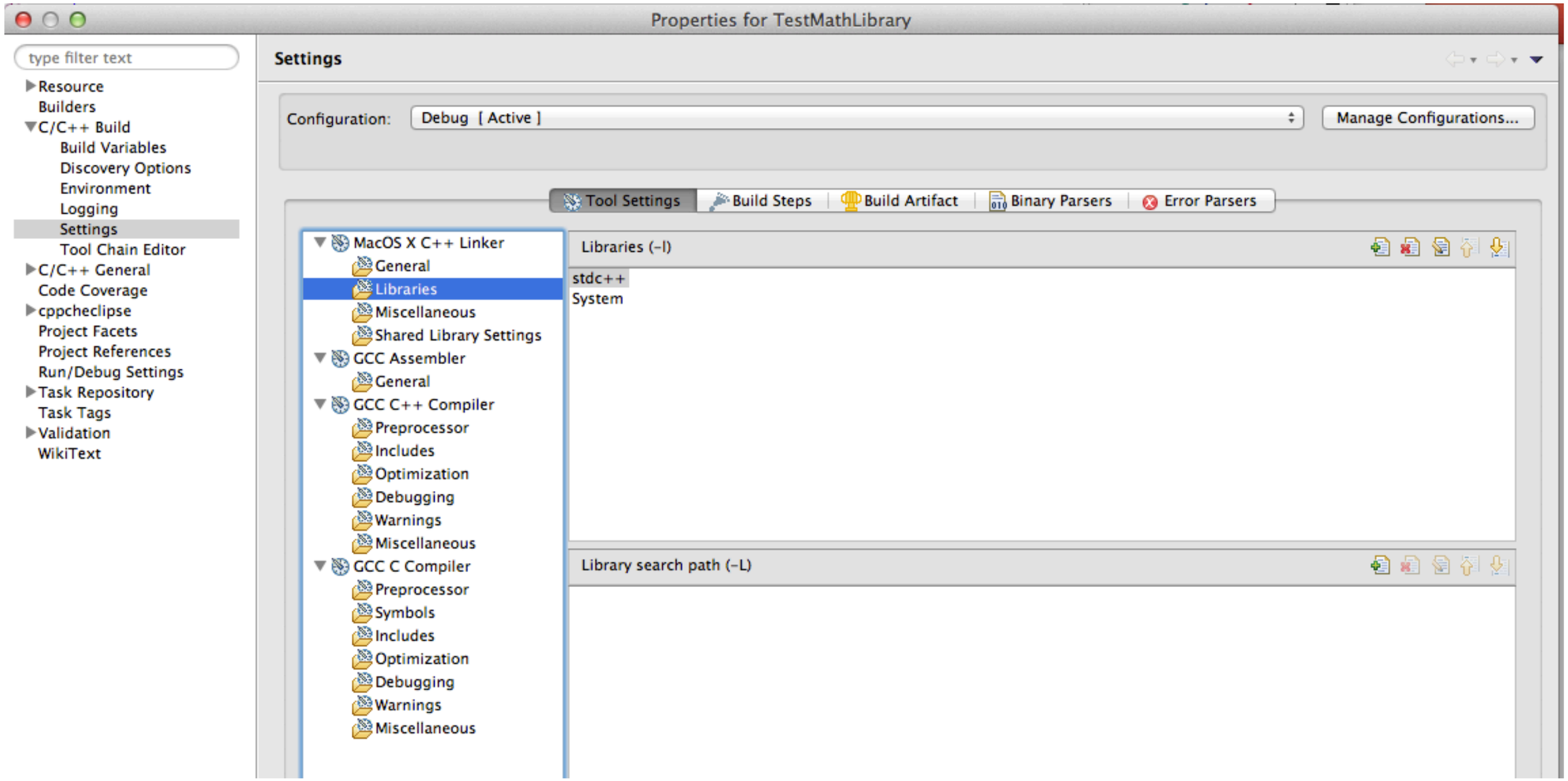
soname

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*.

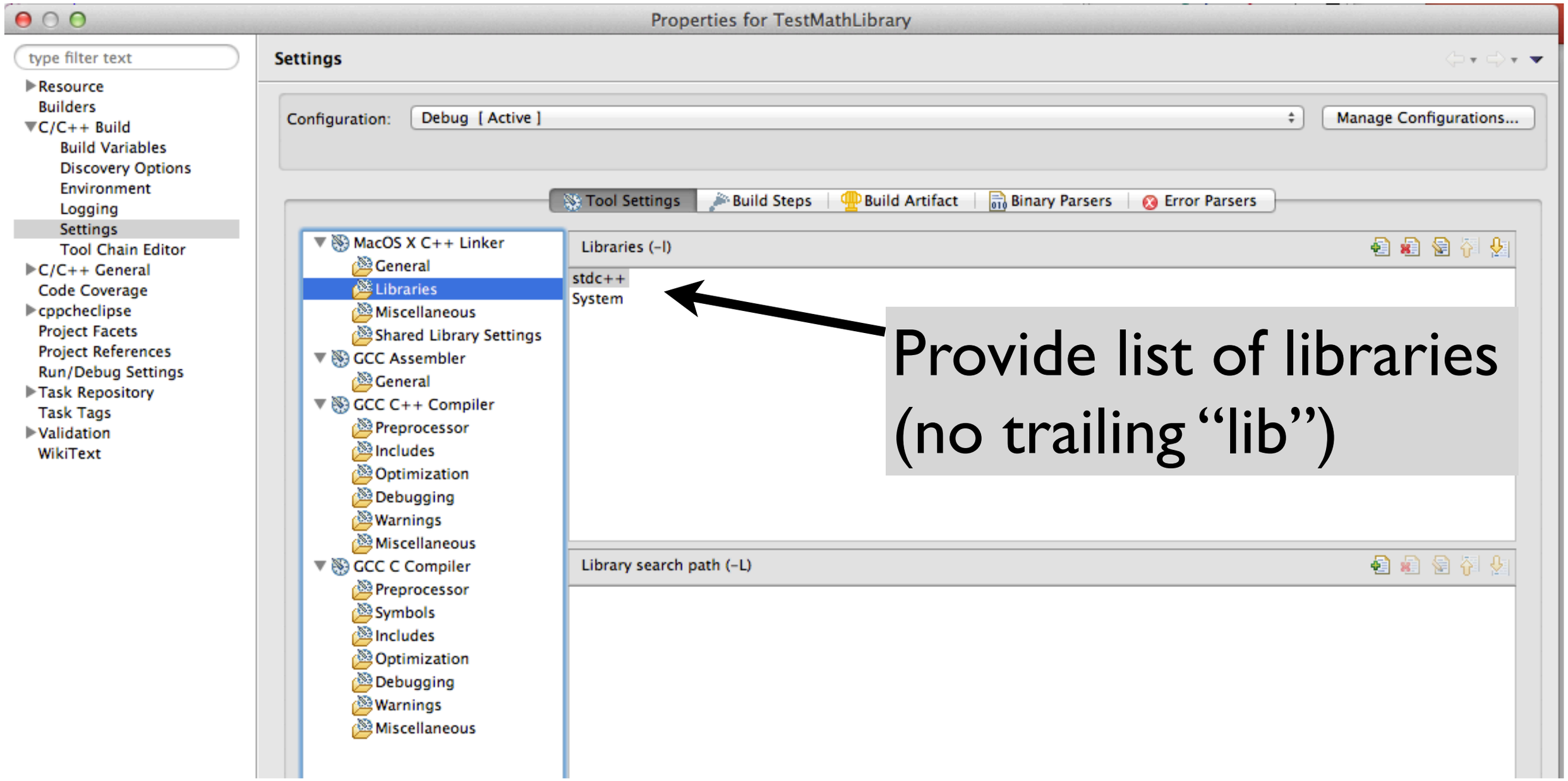


Use libraries in Eclipse



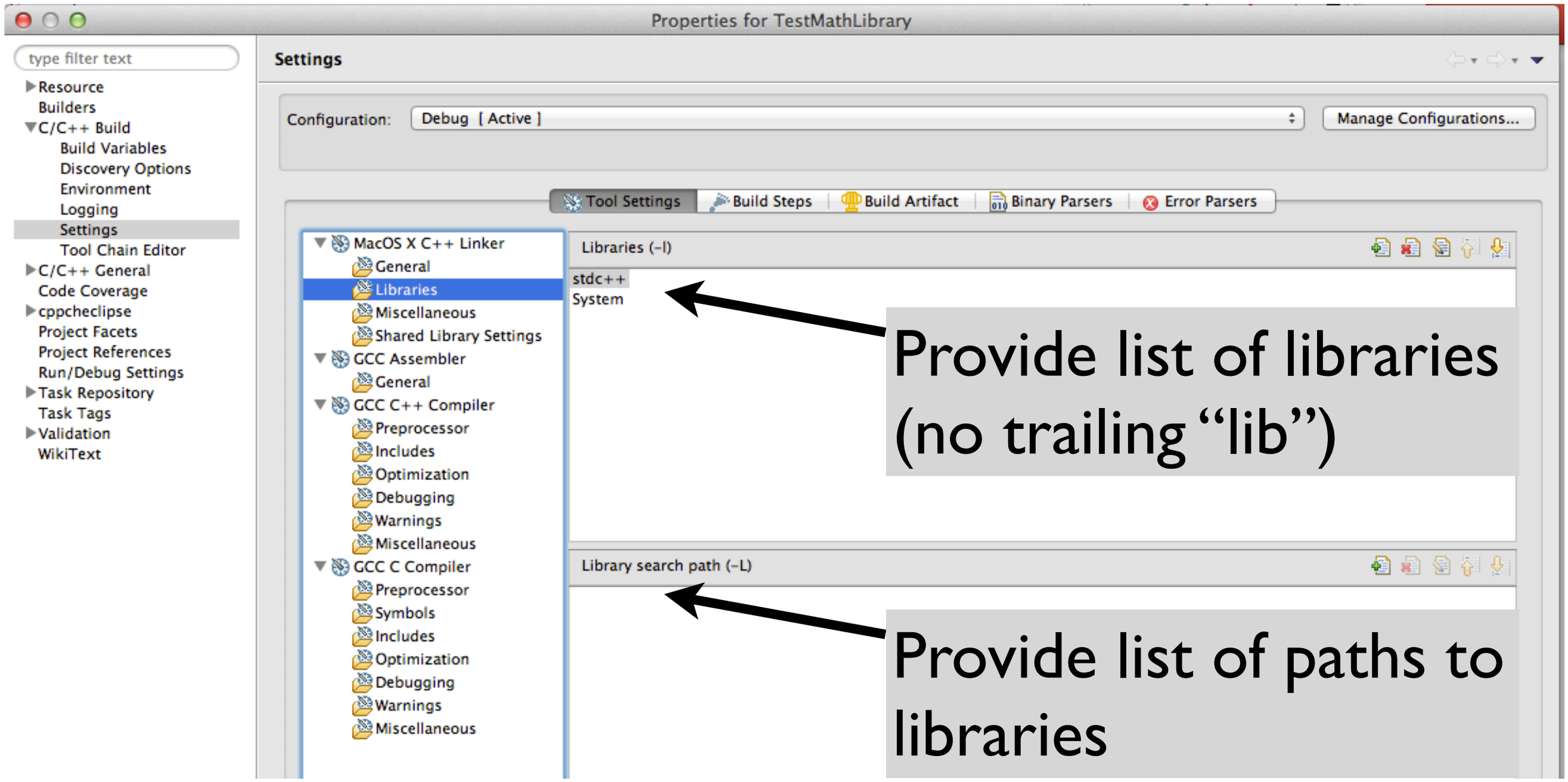
These options are equivalent to command line -l and -L

Use libraries in Eclipse



These options are equivalent to command line -l and -L

Use libraries in Eclipse



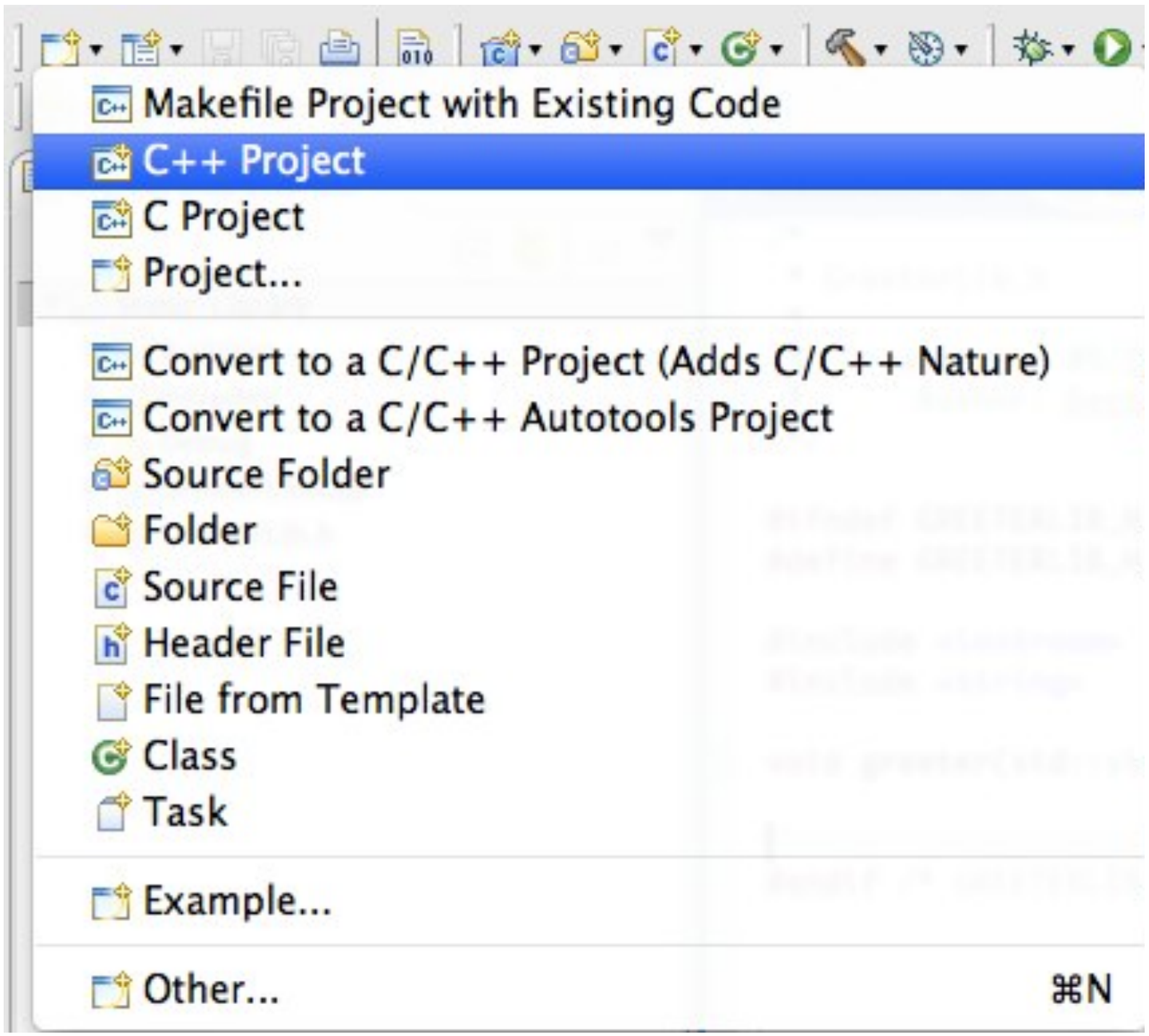
These options are equivalent to command line -l and -L

Creating and using a library

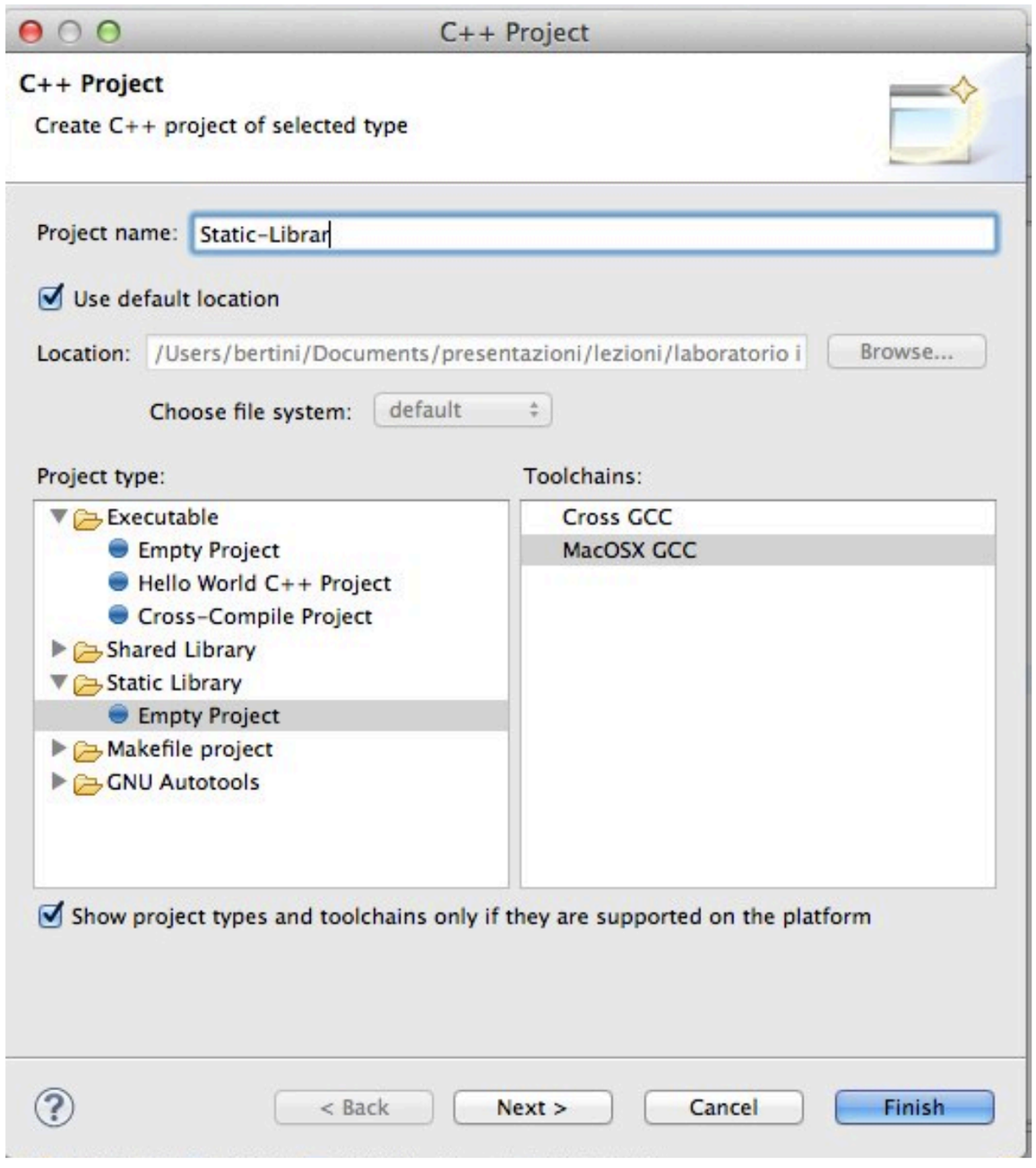
Writing a library

- 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.

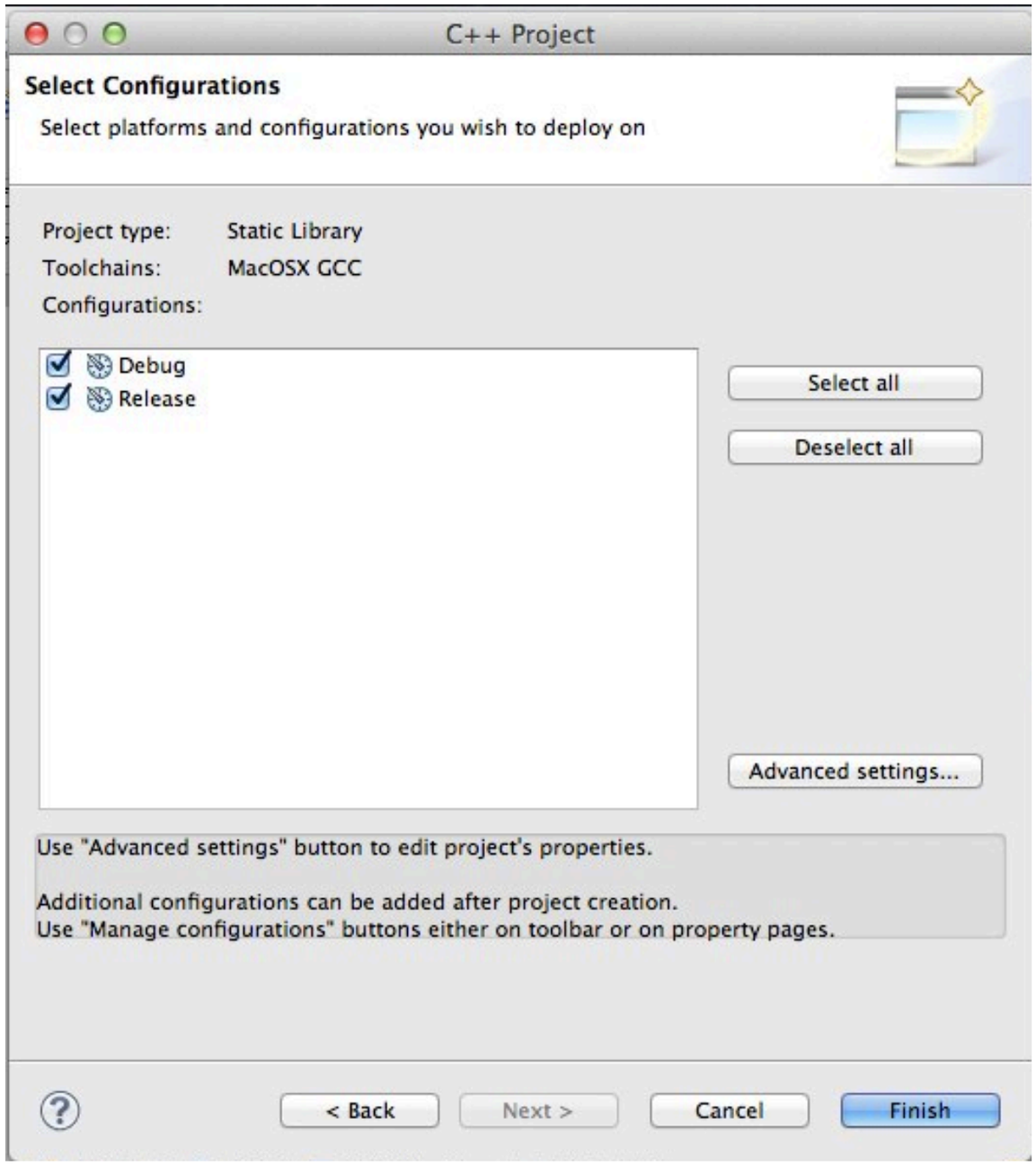
Creating a static library with Eclipse



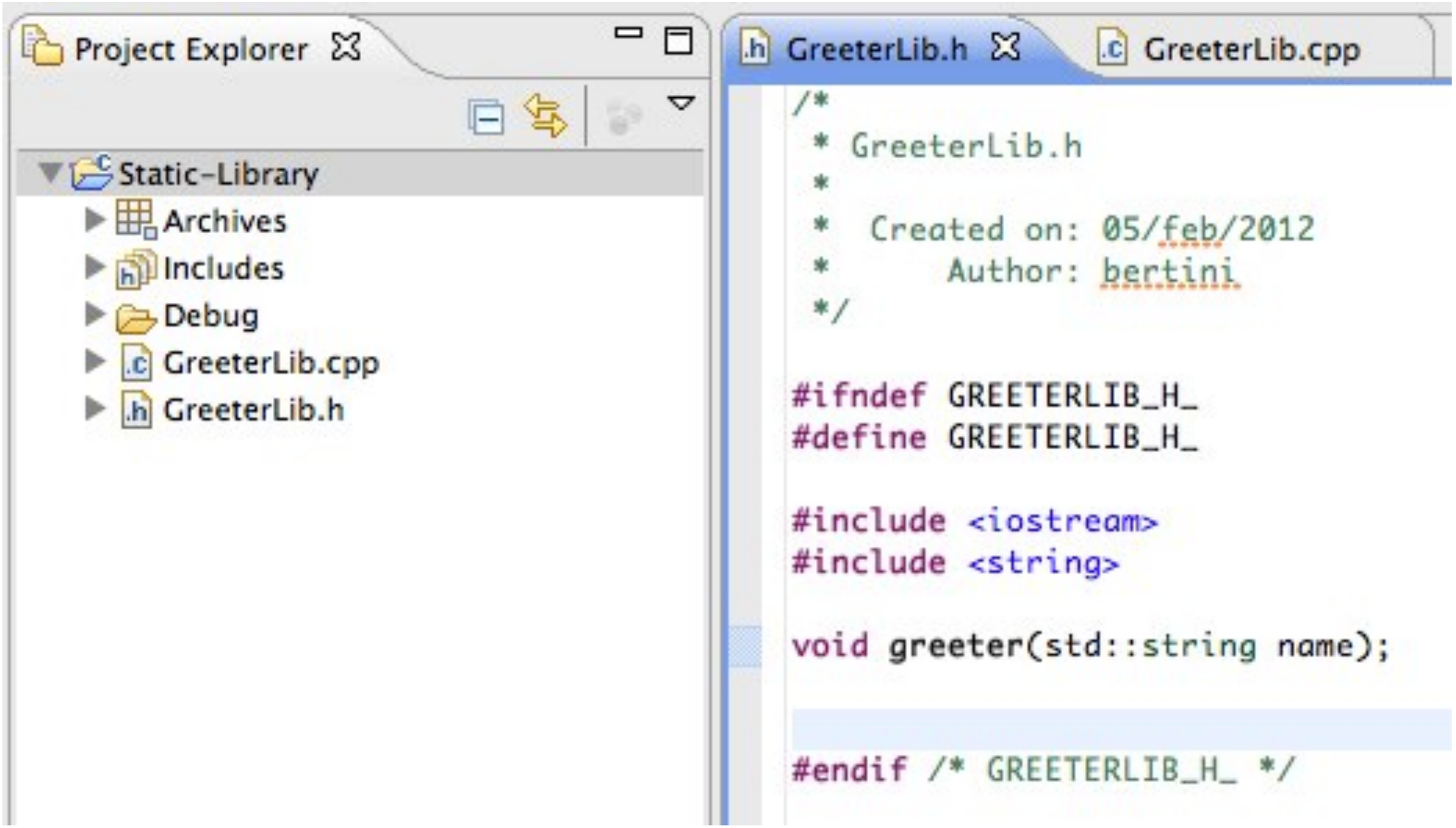
Creating a static library with Eclipse



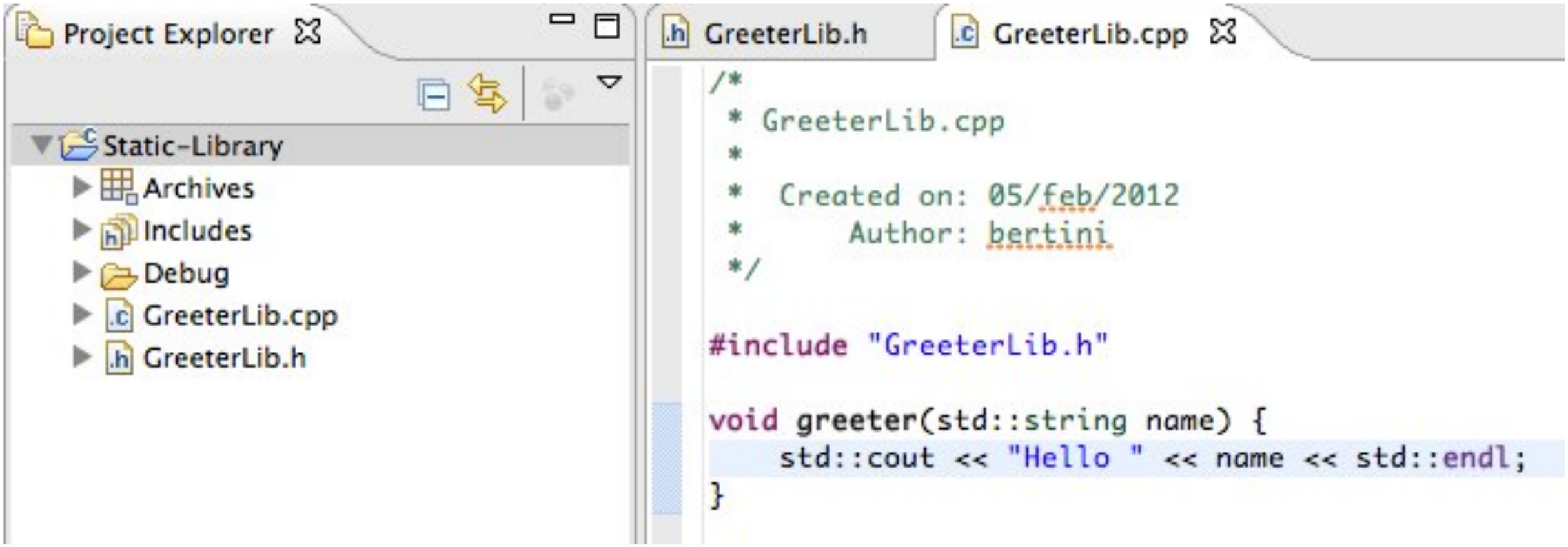
Creating a static library with Eclipse



Creating a static library with Eclipse



Creating a static library with Eclipse



Creating a static library with Eclipse

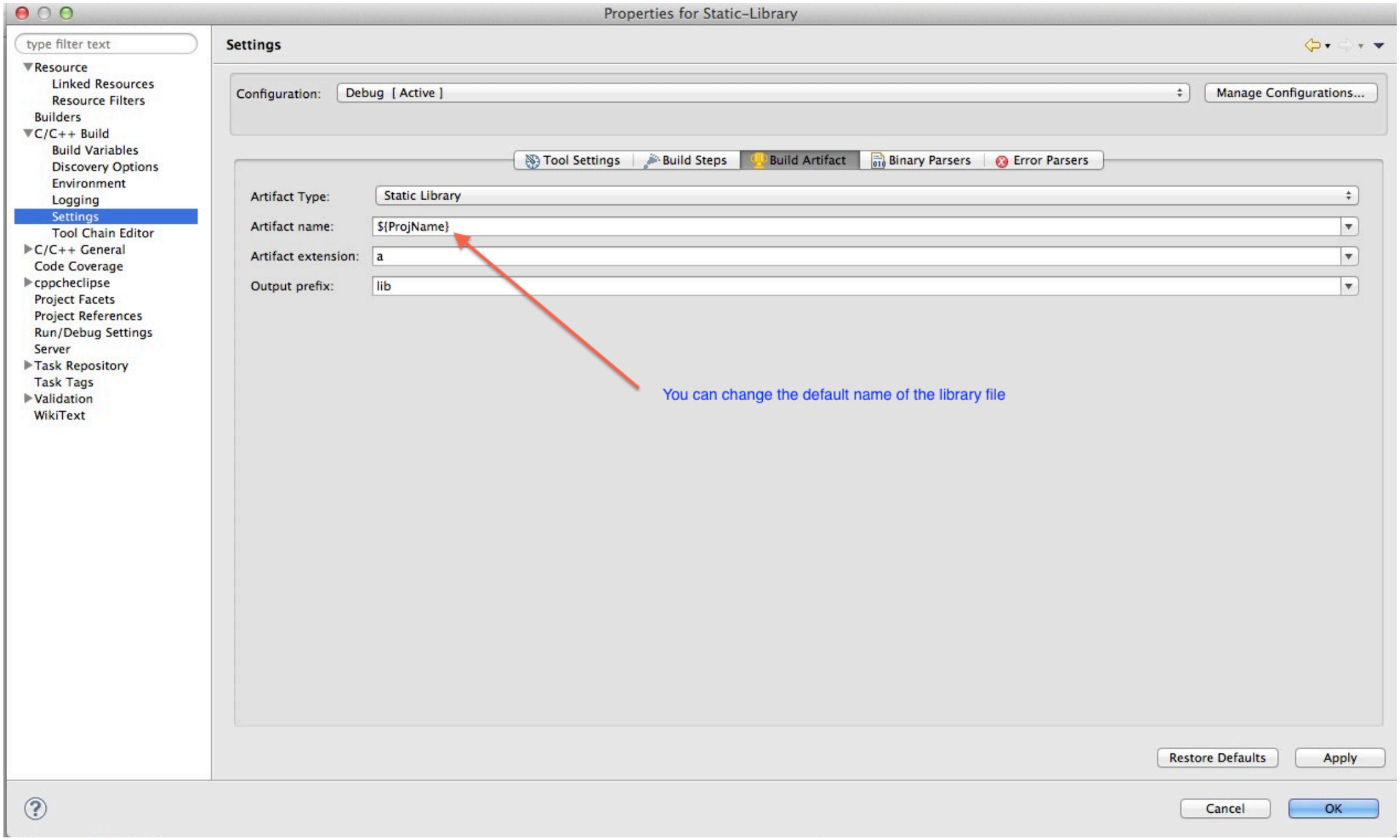
CDT Build Console [Static-Library]

```
**** Build of configuration Debug for project Static-Library ****

make all
Building file: ../GreeterLib.cpp
Invoking: GCC C++ Compiler
g++ -O0 -g3 -Wall -c -fmessage-length=0 -MMD -MP -MF"GreeterLib.d" -MT"GreeterLib.d" -o "GreeterLib.o" "../GreeterLib.cpp"
Finished building: ../GreeterLib.cpp

Building target: libStatic-Library.a
Invoking: GCC Archiver
ar -r  "libStatic-Library.a"  ./GreeterLib.o
ar: creating archive libStatic-Library.a
Finished building target: libStatic-Library.a


**** Build Finished ****
```

Compiling source to object file

Creating library

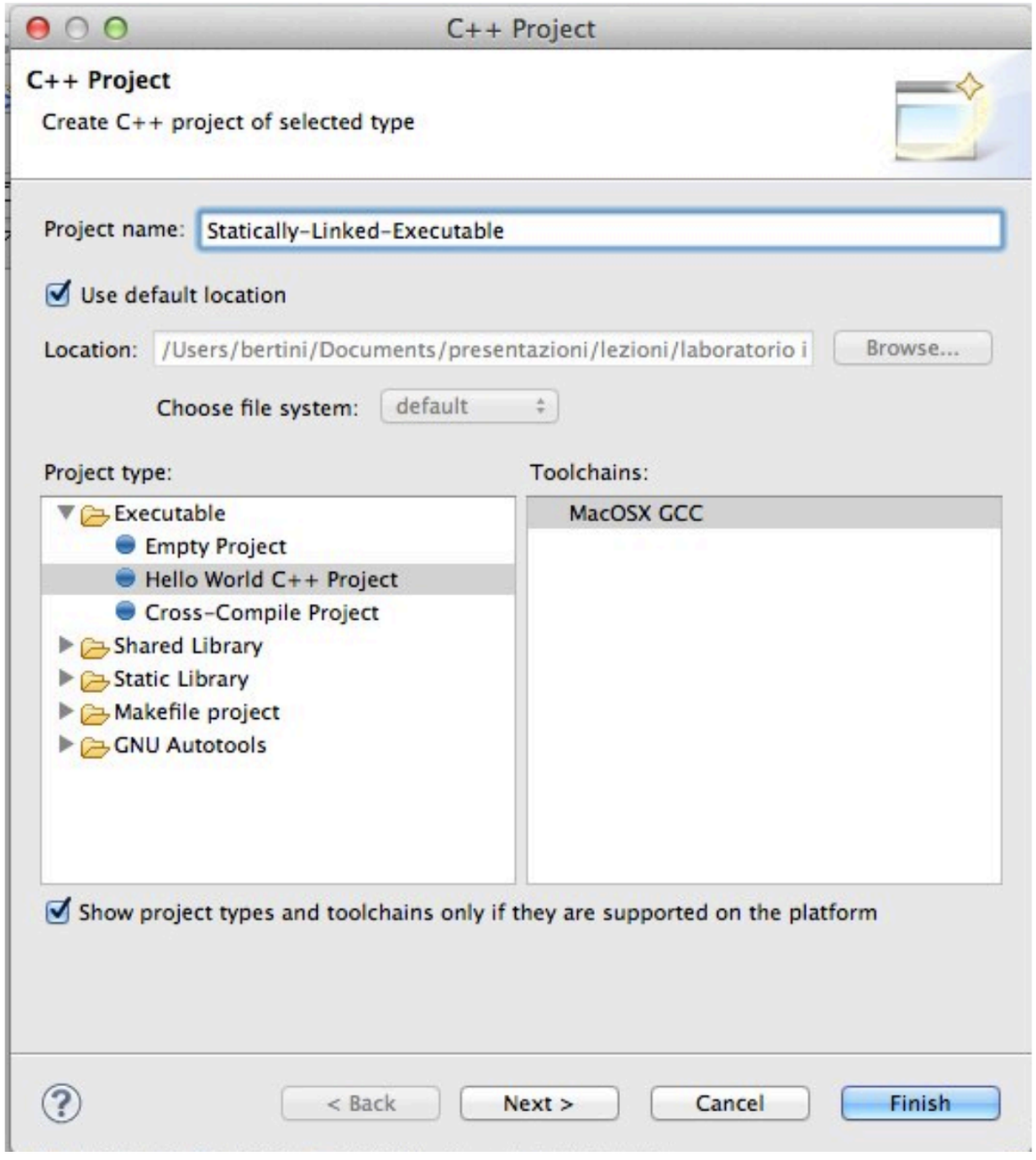
Creating a static library with Eclipse



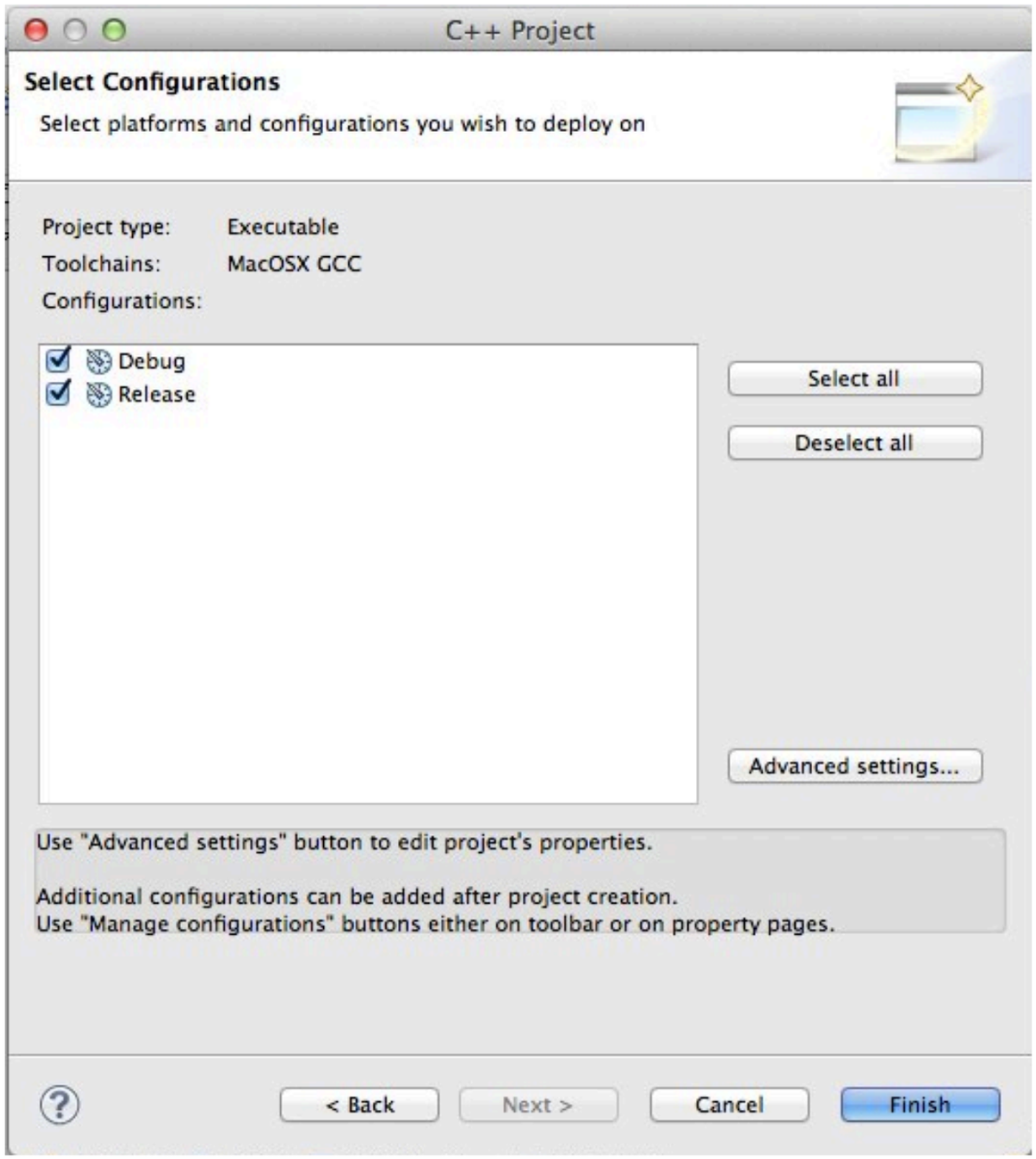
Using a static library with Eclipse

- 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 !)
- Eclipse will use this information to create the required makefile

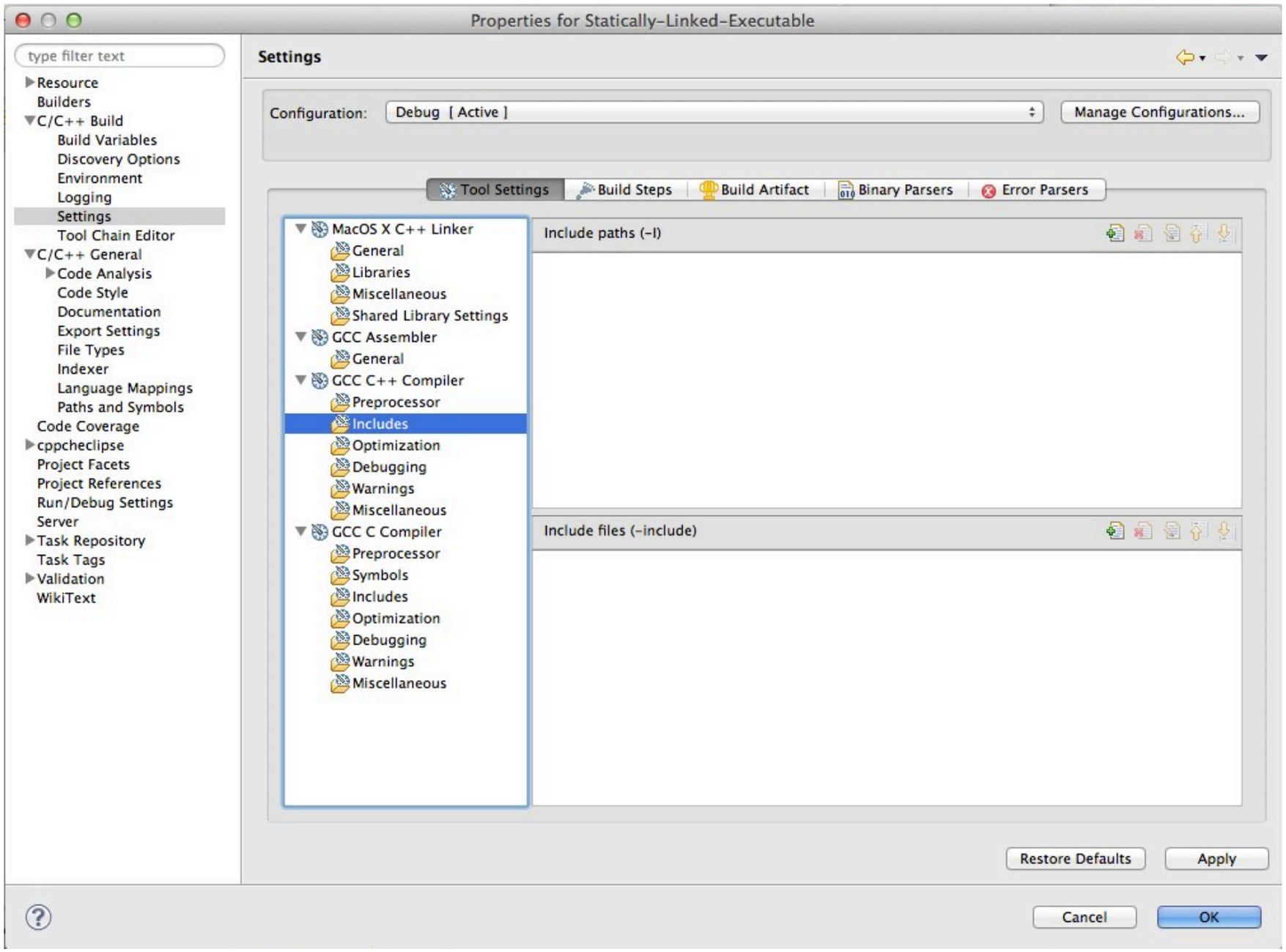
Using a static library with Eclipse



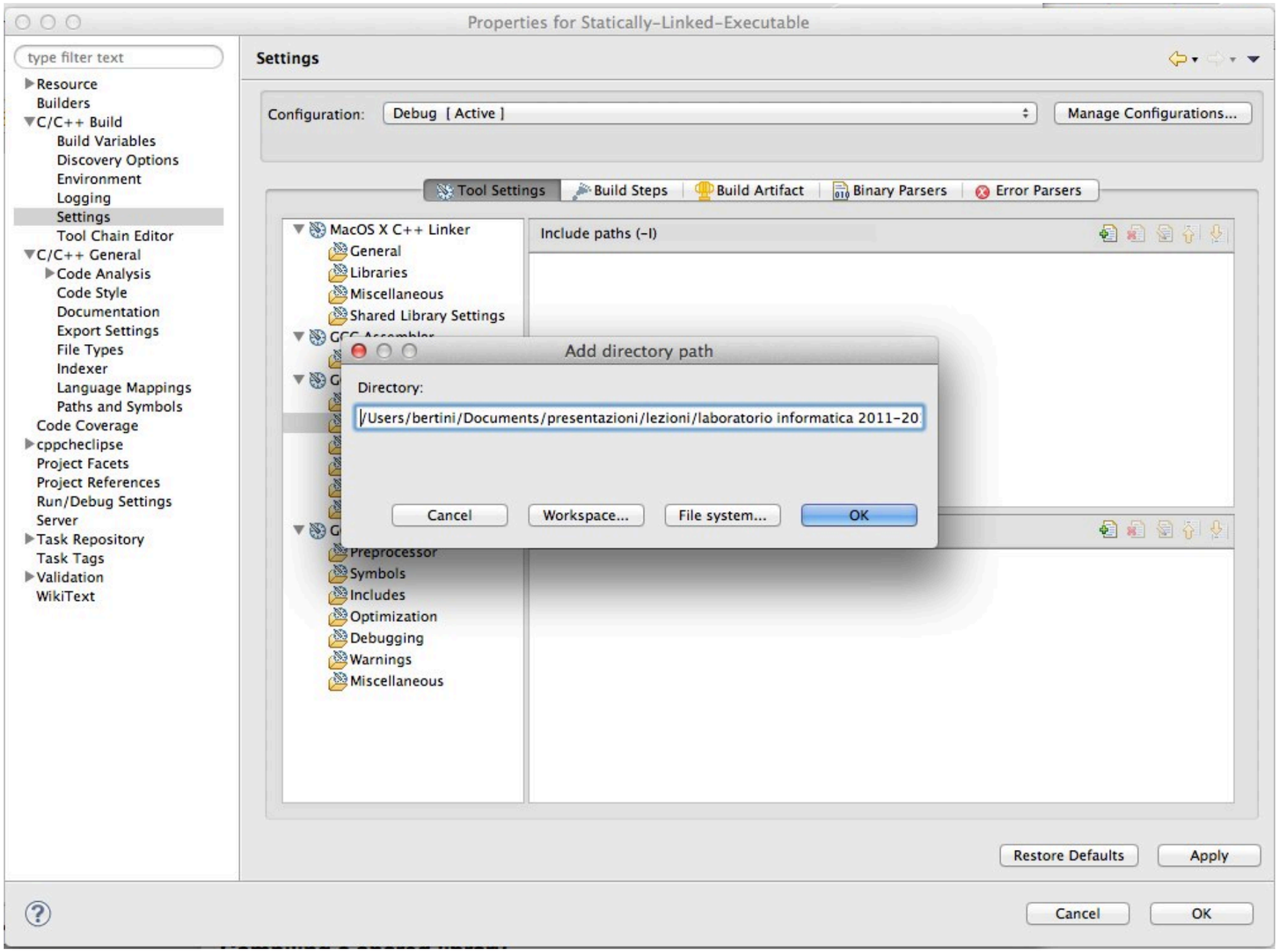
Using a static library with Eclipse



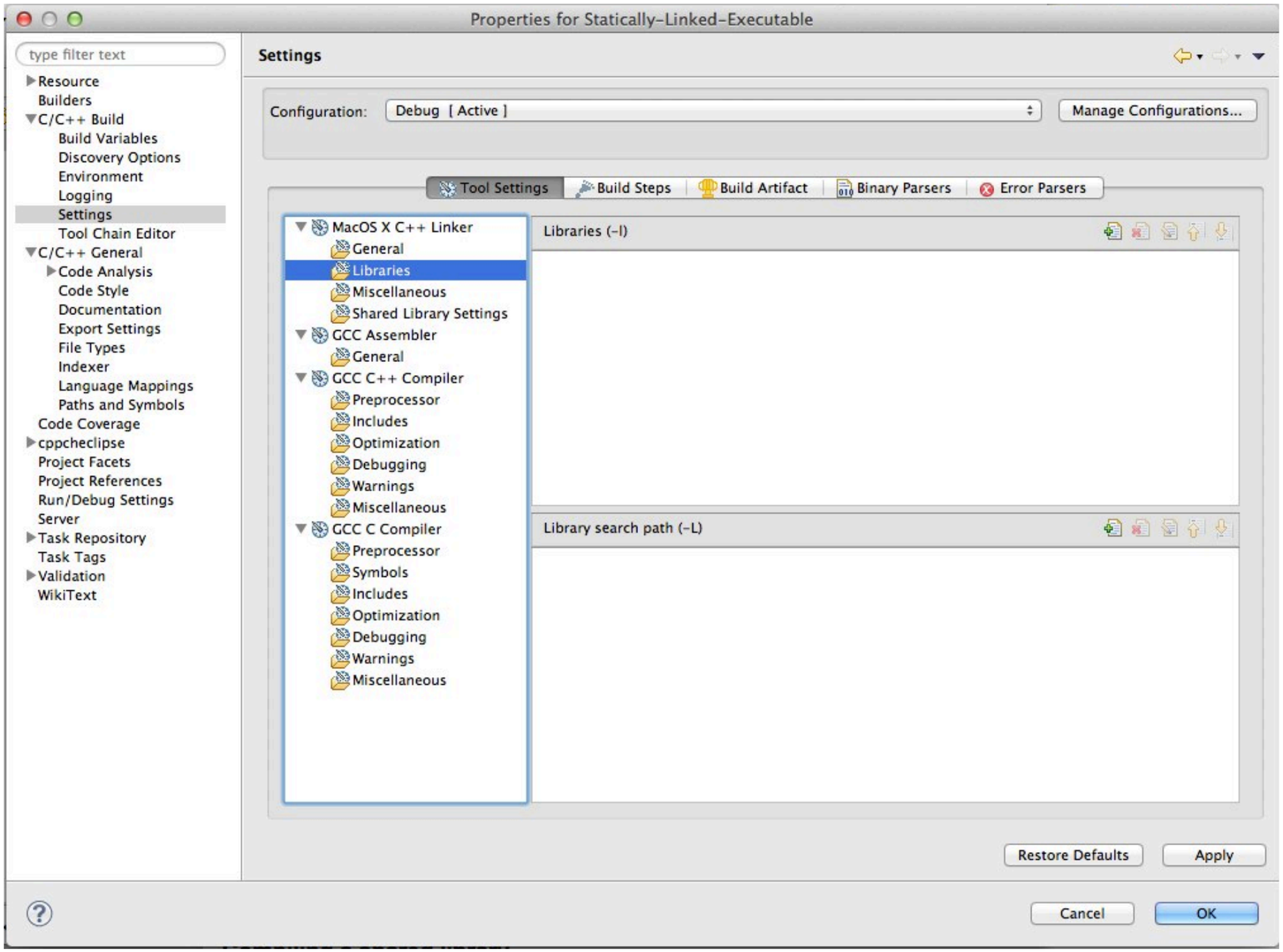
Using a static library with Eclipse



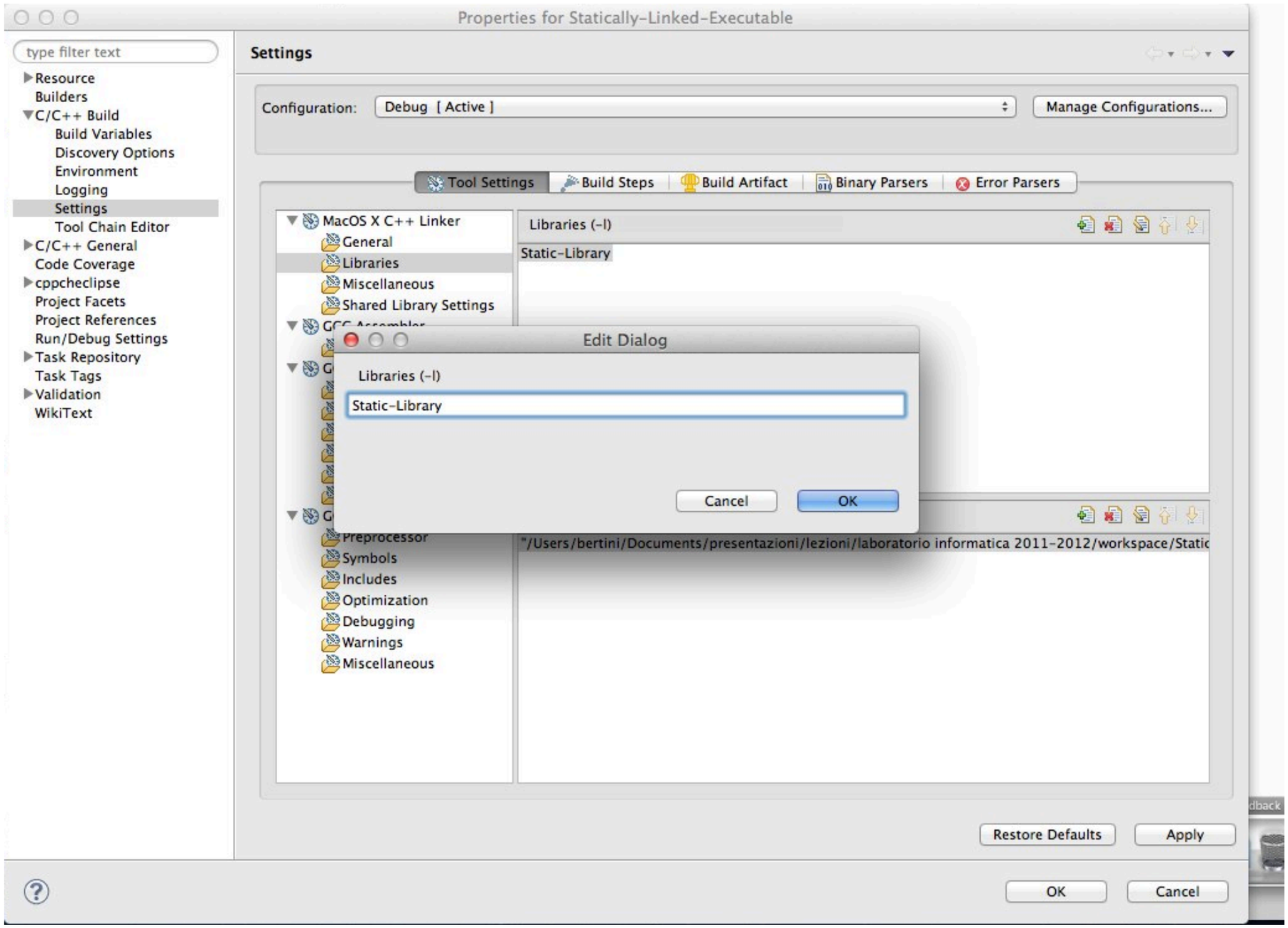
Using a static library with Eclipse



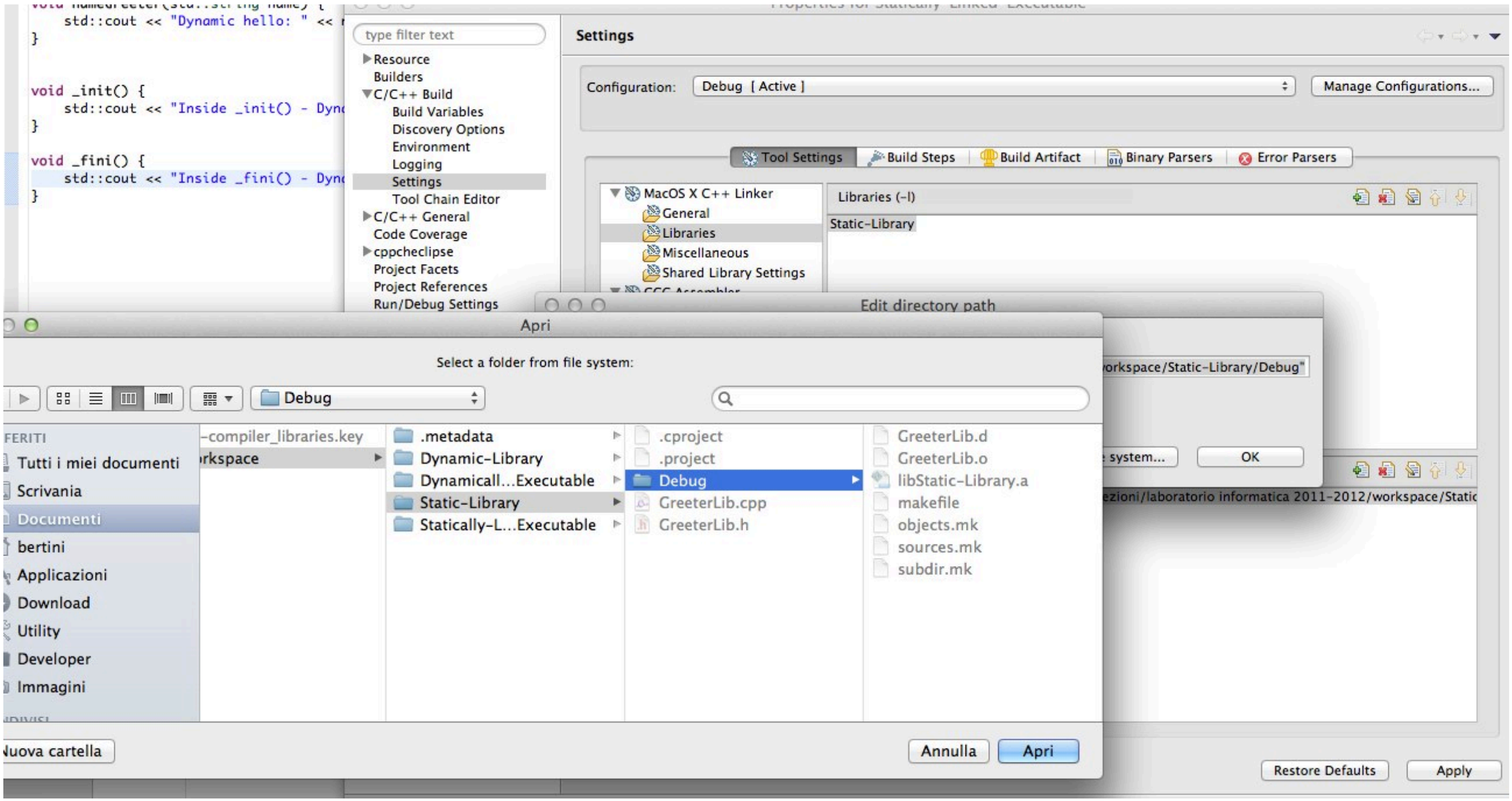
Using a static library with Eclipse



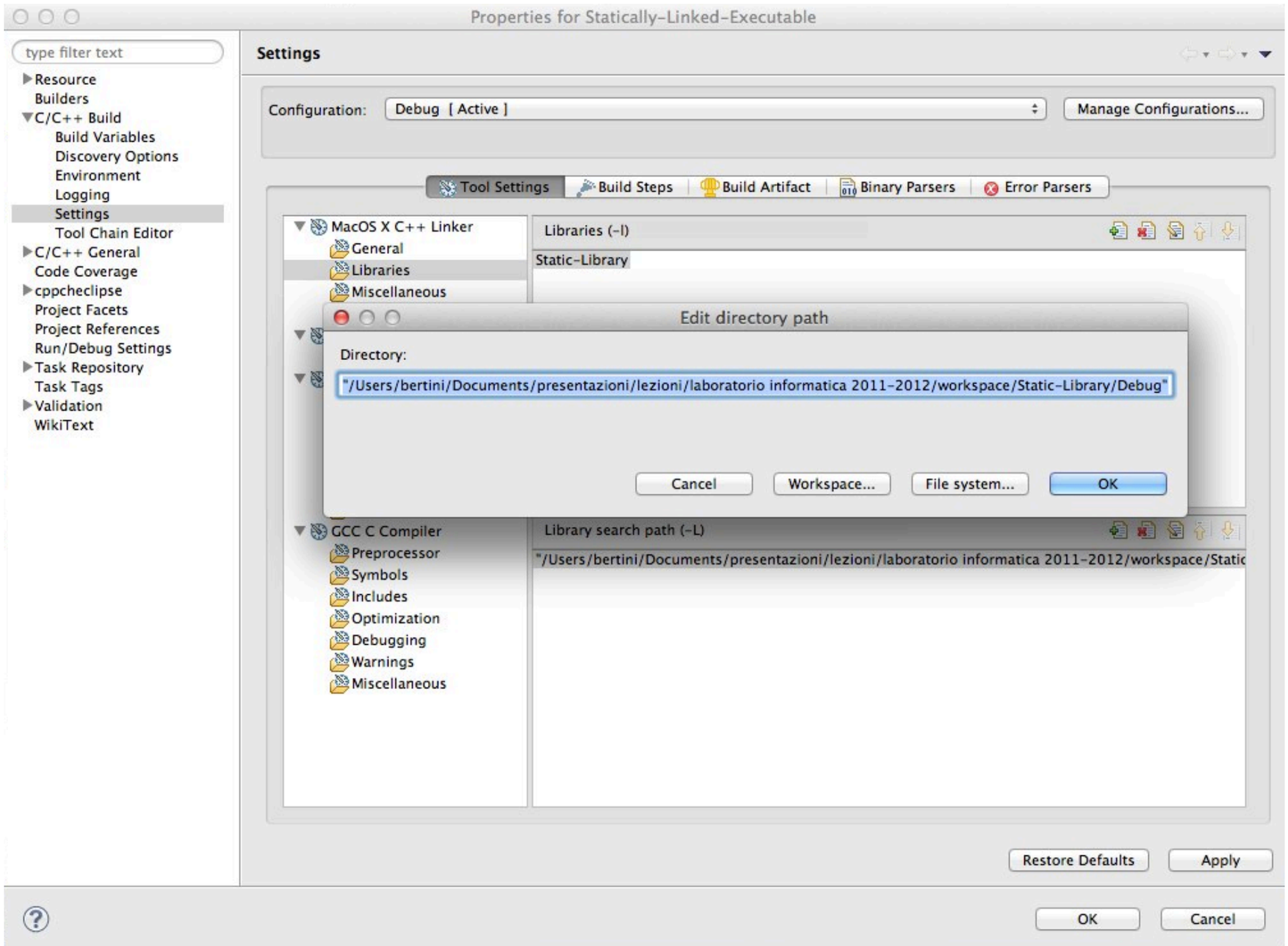
Using a static library with Eclipse



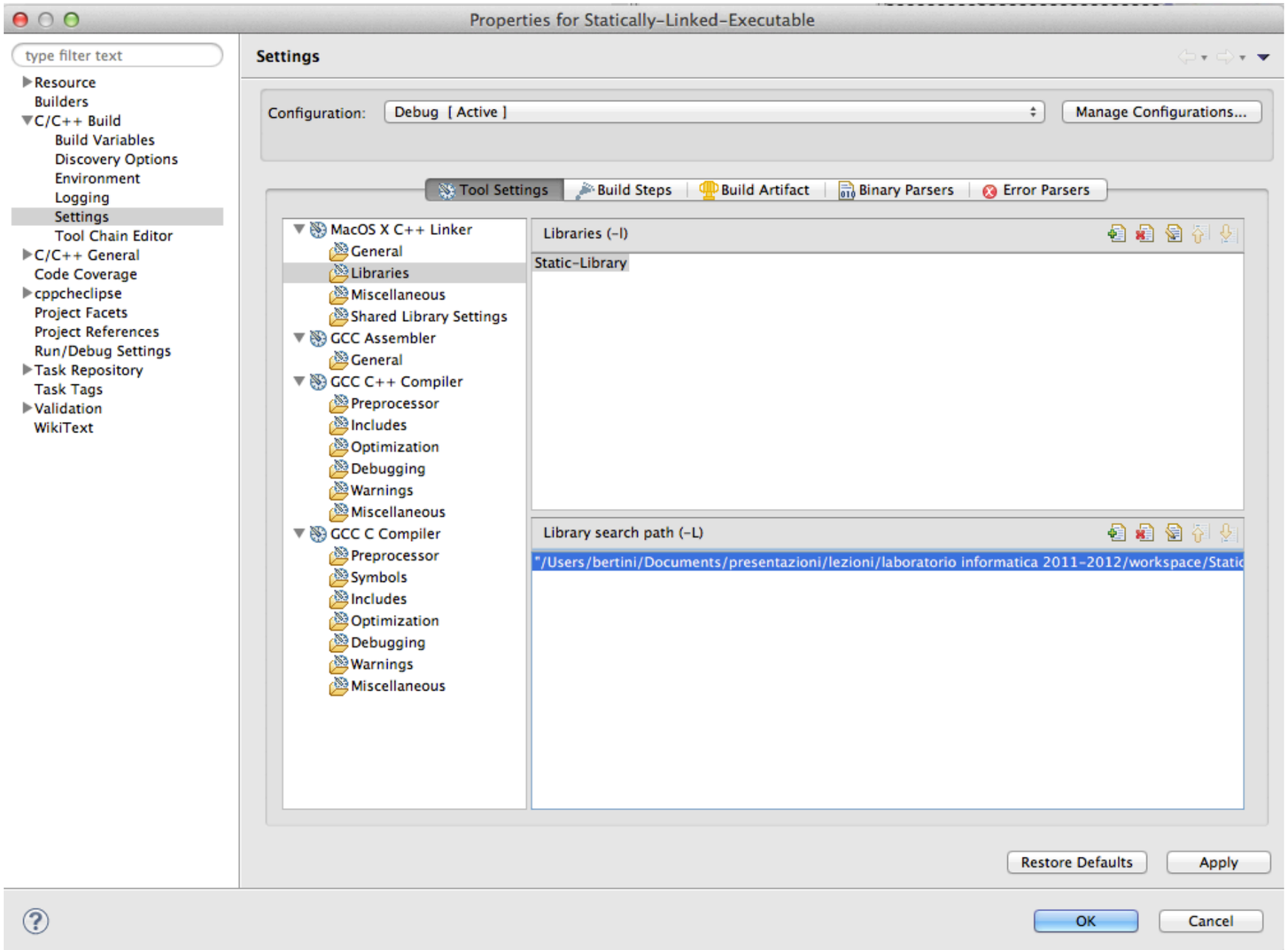
Using a static library with Eclipse



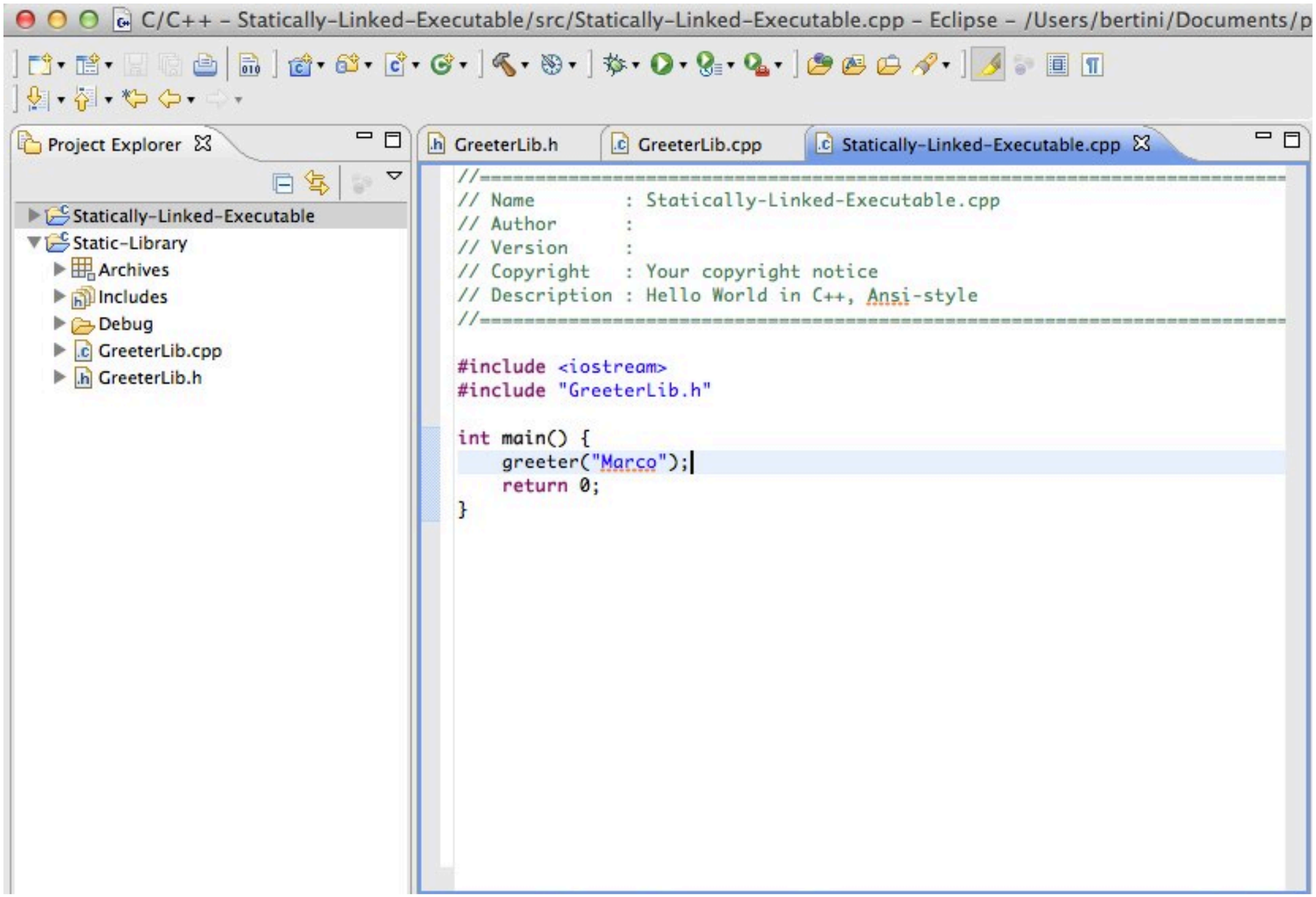
Using a static library with Eclipse



Using a static library with Eclipse



Using a static library with Eclipse



Using a static library with Eclipse

Problems | Tasks | Console | Properties

CDT Build Console [Statically-Linked-Executable]

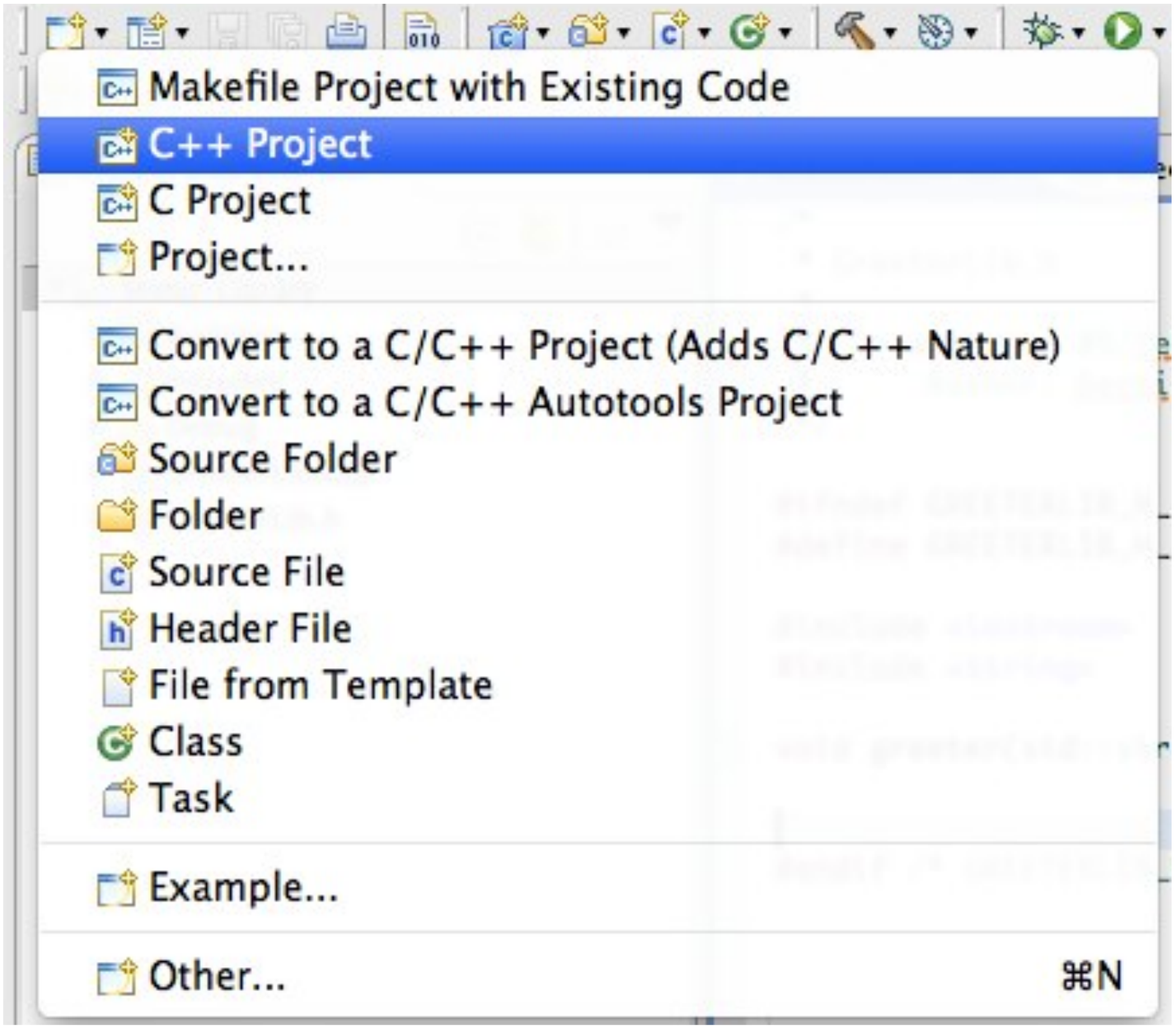
```
**** Build of configuration Debug for project Statically-Linked-Executable ****

make all
Building file: ../src/Statically-Linked-Executable.cpp
Invoking: GCC C++ Compiler
g++ -I"/Users/bertini/Documents/presentazioni/lezioni/laboratorio informatica 2011-2012/workspace/Static-Library" -O0 -g3 -Wall -c -
fmessage-length=0 -MMD -MP -MF"src/Statically-Linked-Executable.d" -MT"src/Statically-Linked-Executable.d" -o "src/Statically-Linked-
Executable.o" "../src/Statically-Linked-Executable.cpp"
Finished building: ../src/Statically-Linked-Executable.cpp

Building target: Statically-Linked-Executable
Invoking: MacOS X C++ Linker
g++ -L"/Users/bertini/Documents/presentazioni/lezioni/laboratorio informatica 2011-2012/workspace/Static-Library/Debug" -o "Statically-
Linked-Executable"  ./src/Statically-Linked-Executable.o   -lStatic-Library
Finished building target: Statically-Linked-Executable


**** Build Finished ****
```

The program is compiled

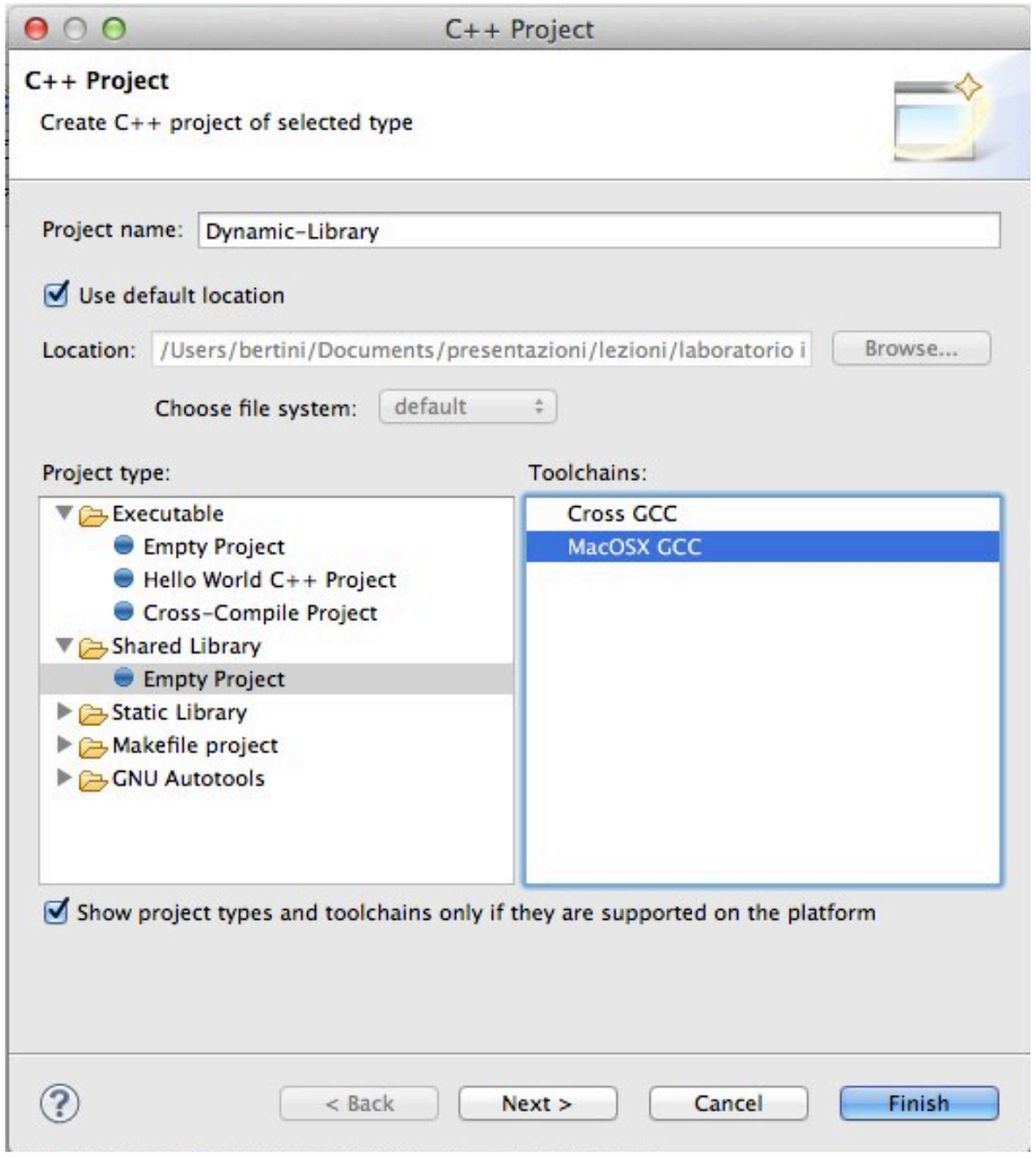
The library is linked

The executable is linked and created

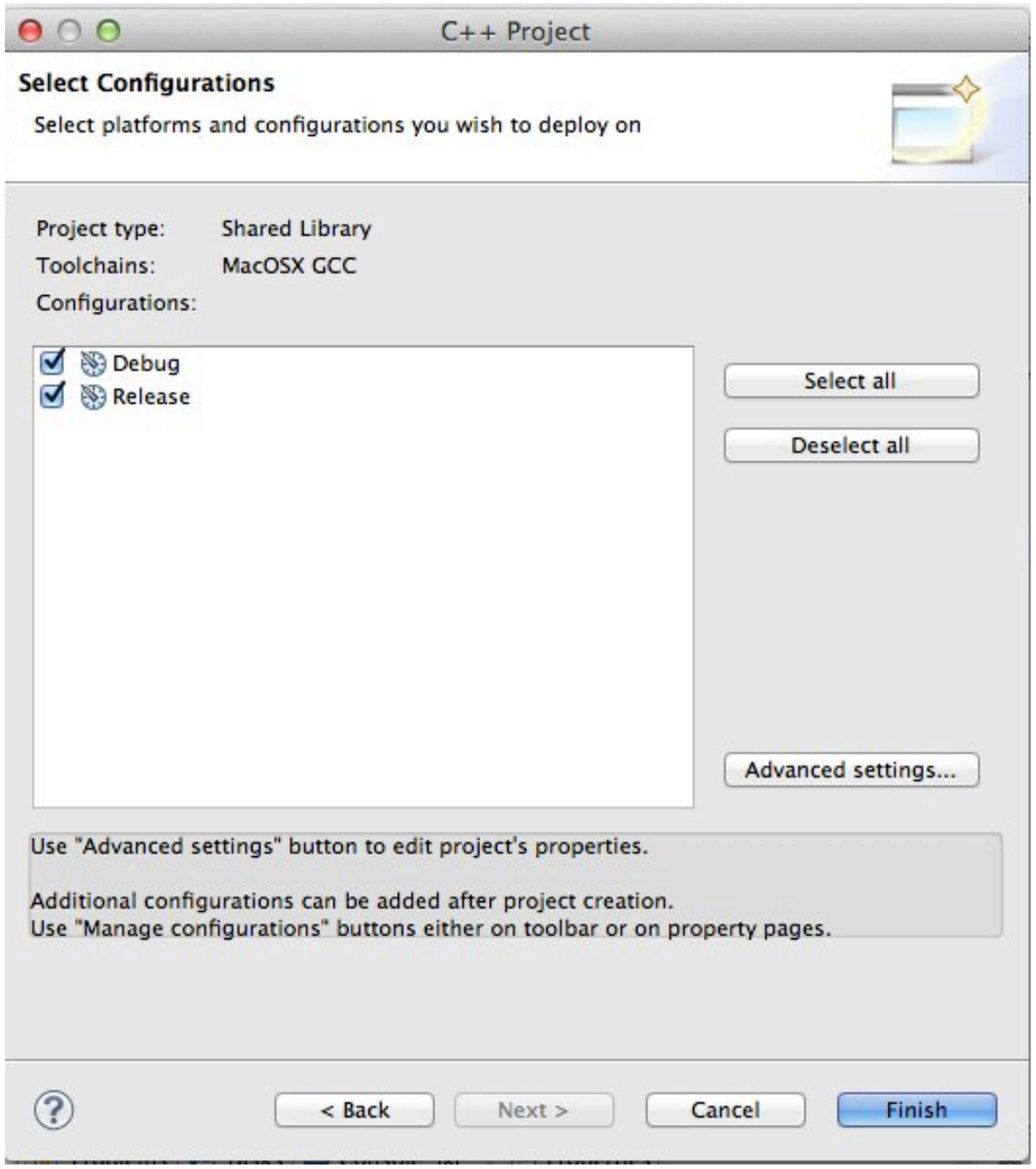
Creating a dynamic library with Eclipse



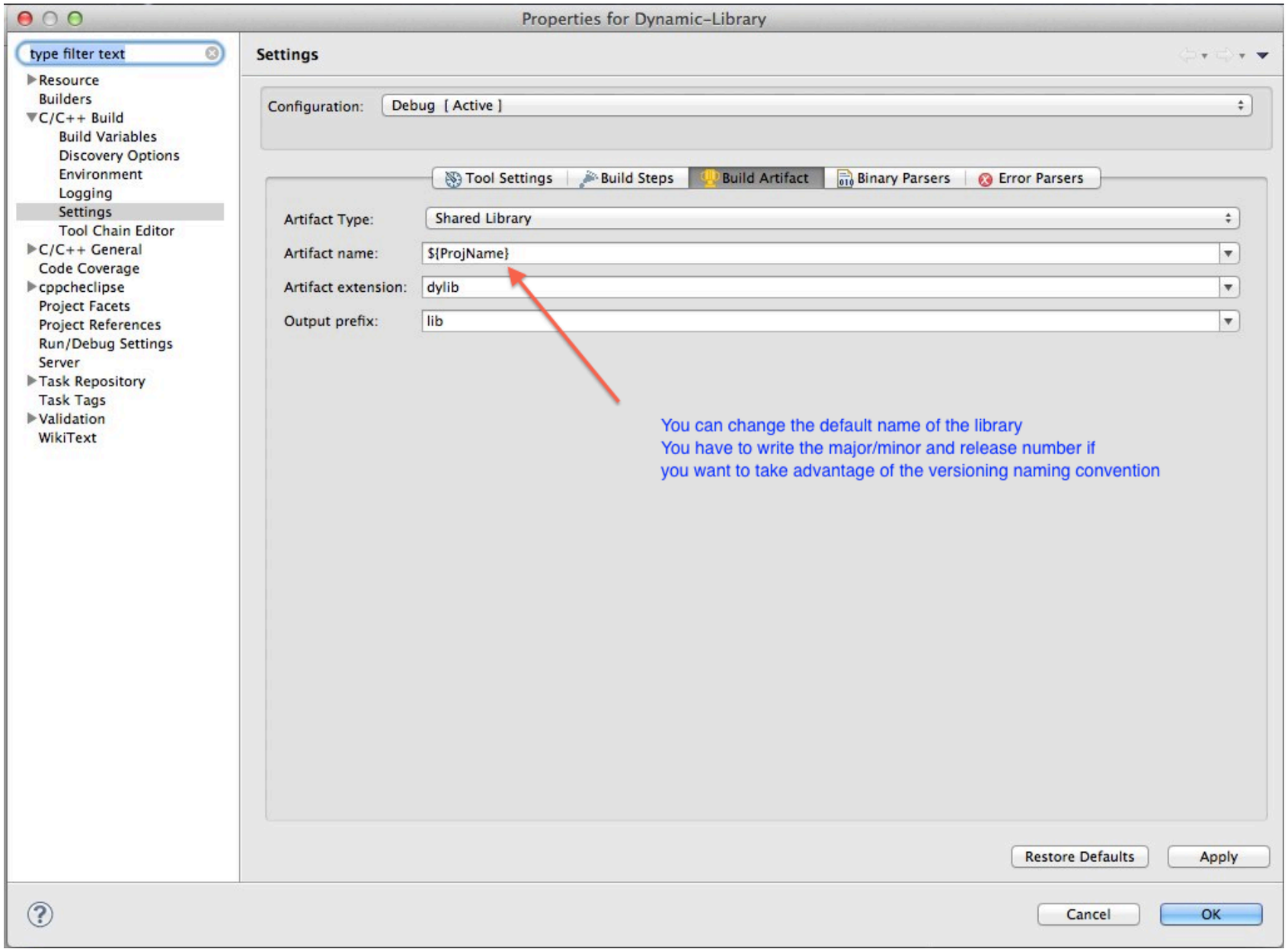
Creating a dynamic library with Eclipse



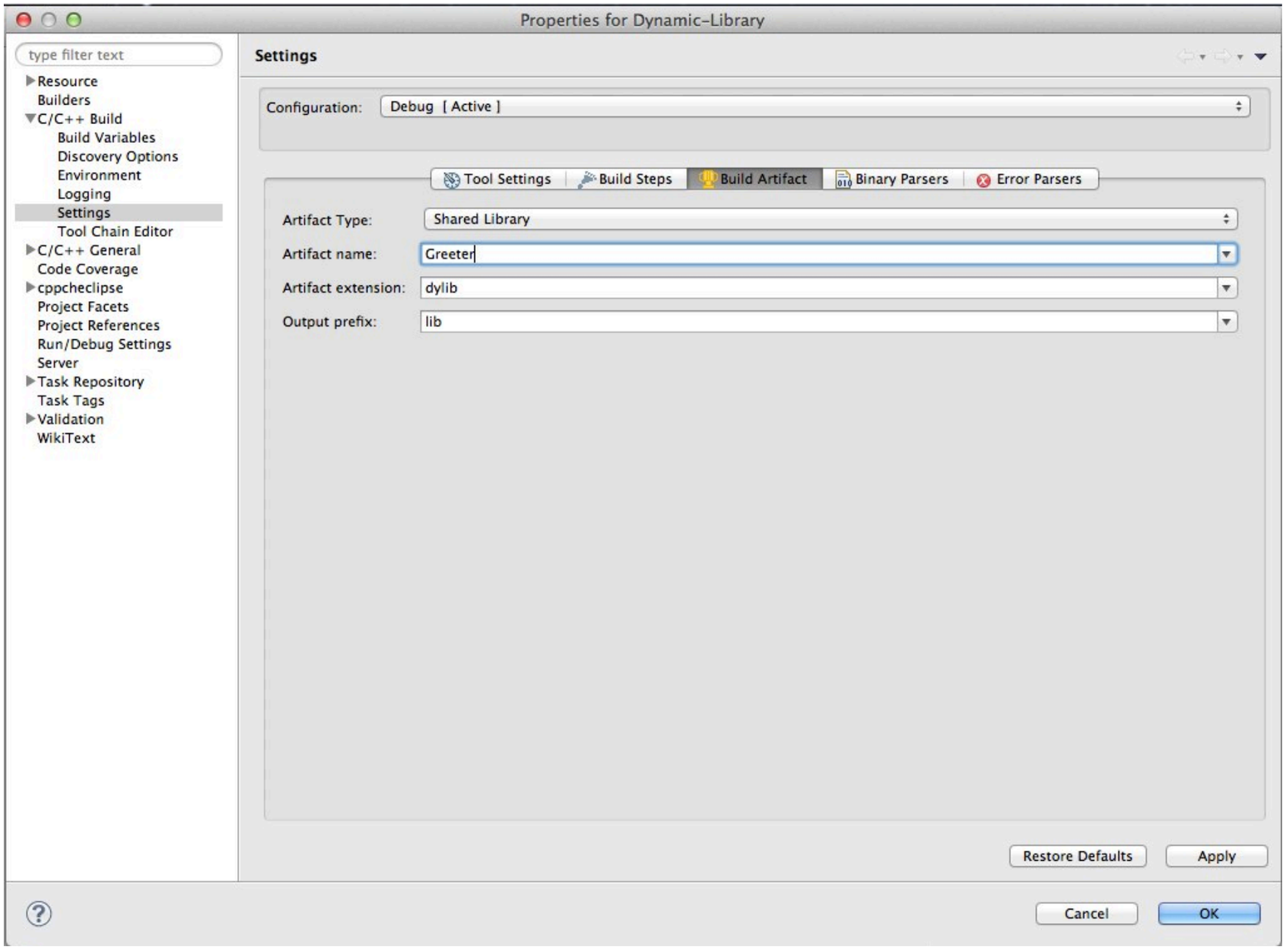
Creating a dynamic library with Eclipse



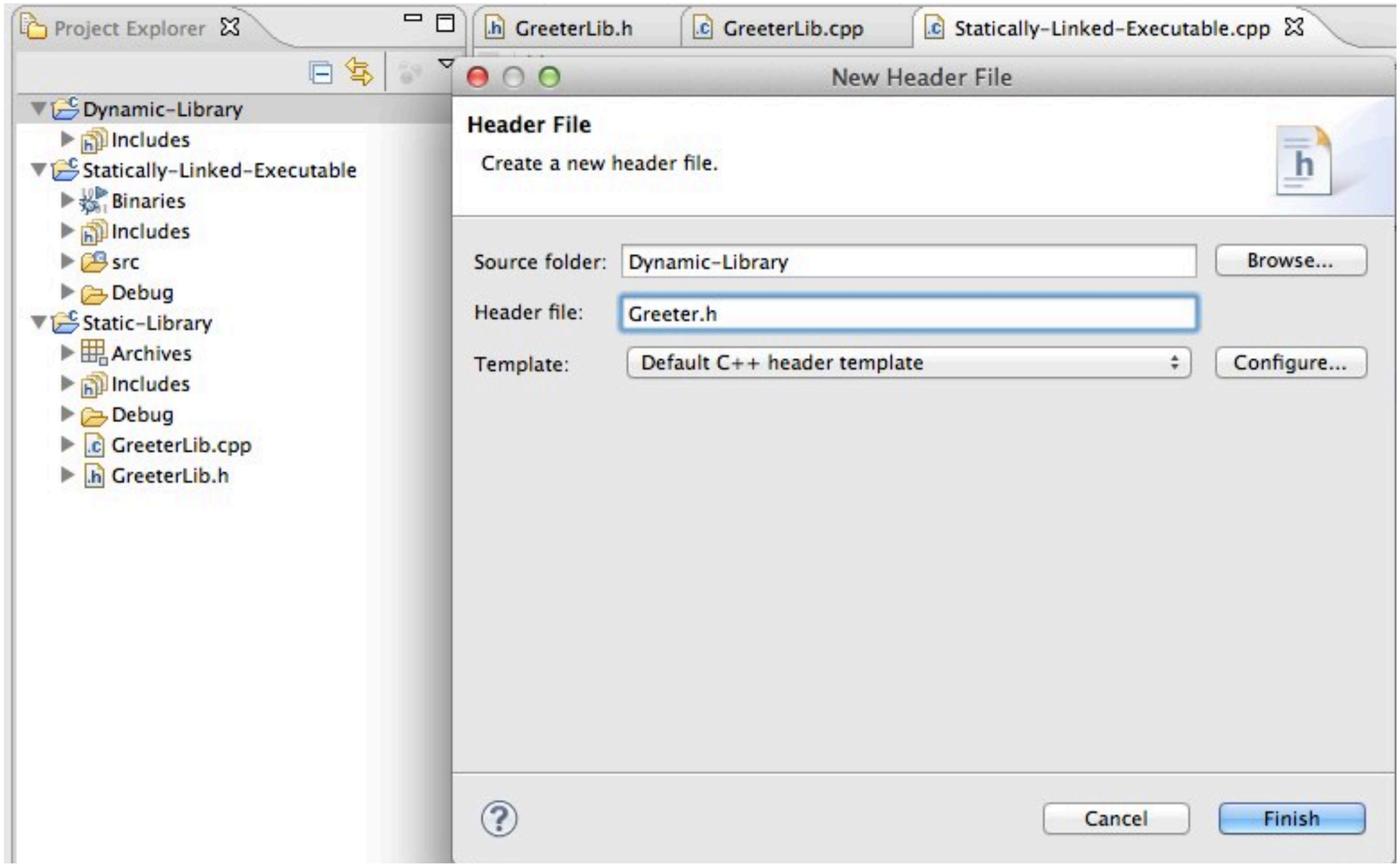
Creating a dynamic library with Eclipse



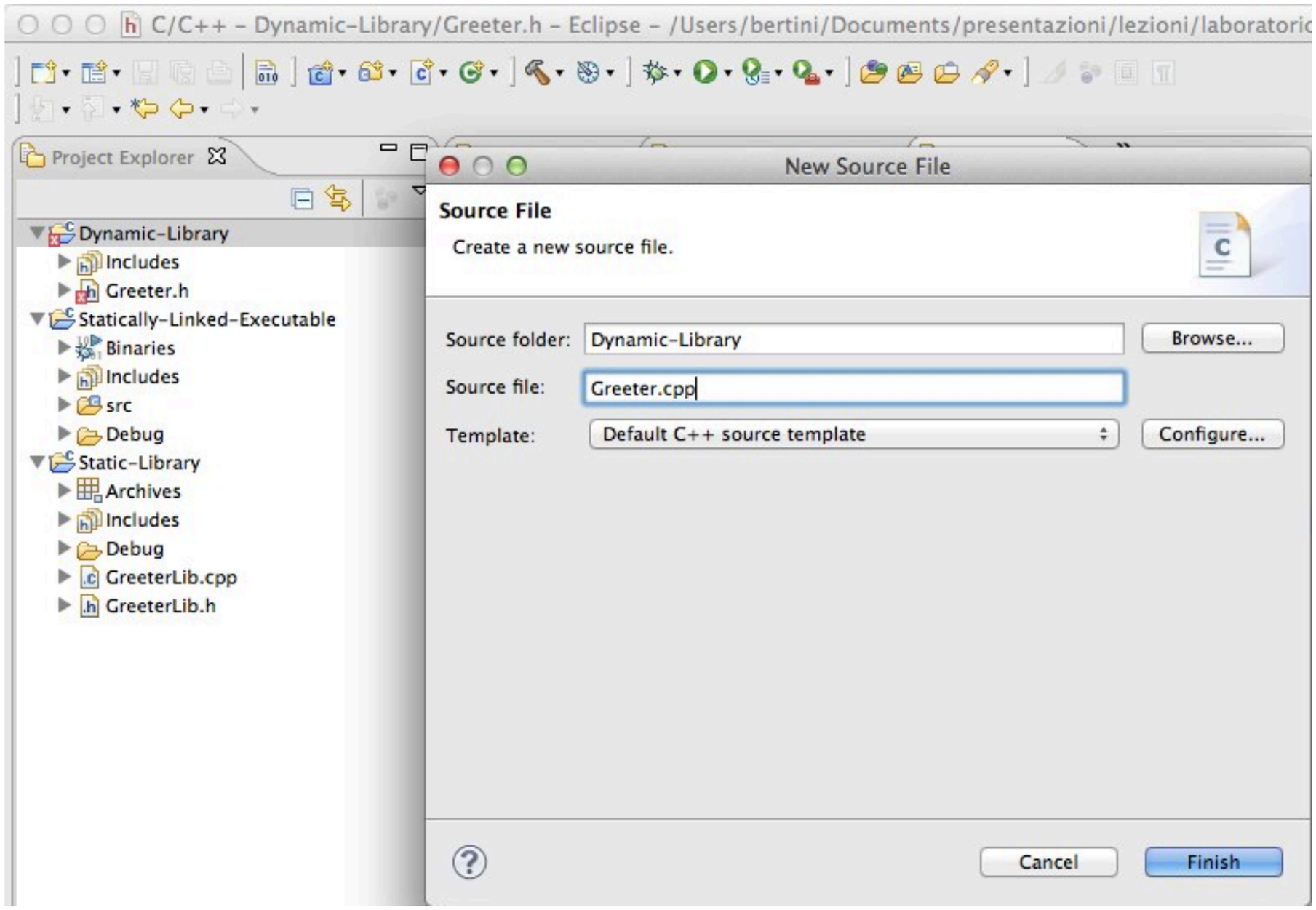
Creating a dynamic library with Eclipse



Creating a dynamic library with Eclipse



Creating a dynamic library with Eclipse



Creating a dynamic library with Eclipse

Creating a dynamic library with Eclipse

CDT Build Console [Dynamic-Library]

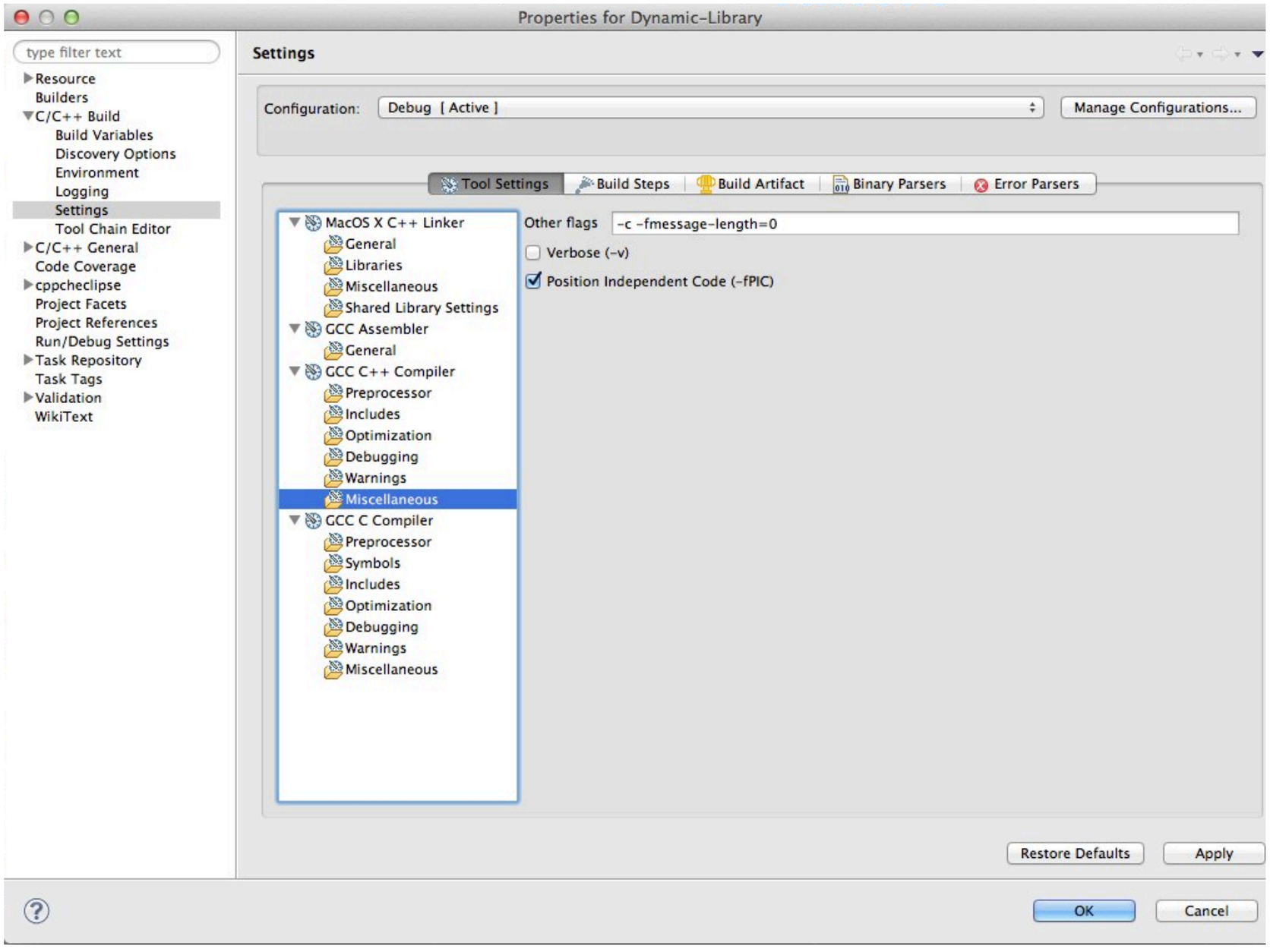
```
**** Build of configuration Debug for project Dynamic-Library ****

make all
Building file: ../Greeter.cpp
Invoking: GCC C++ Compiler
g++ -O0 -g3 -Wall -c -fmessage-length=0 -fPIC -MMD -MP -MF"Greeter.d" -MT"Greeter.d" -o "Greeter.o" "../Greeter.cpp"
Finished building: ../Greeter.cpp

Building target: libGreeter.dylib
Invoking: MacOS X C++ Linker
g++ -dynamiclib -o "libGreeter.dylib"  ./Greeter.o
Finished building target: libGreeter.dylib


**** Build Finished ****
```

Library source code is compiled, creating an object file

the -fPIC parameter is used to create Position-Independent Code.
It's better to use it to improve performance of the s/w using the library.

The linker creates a dynamic library

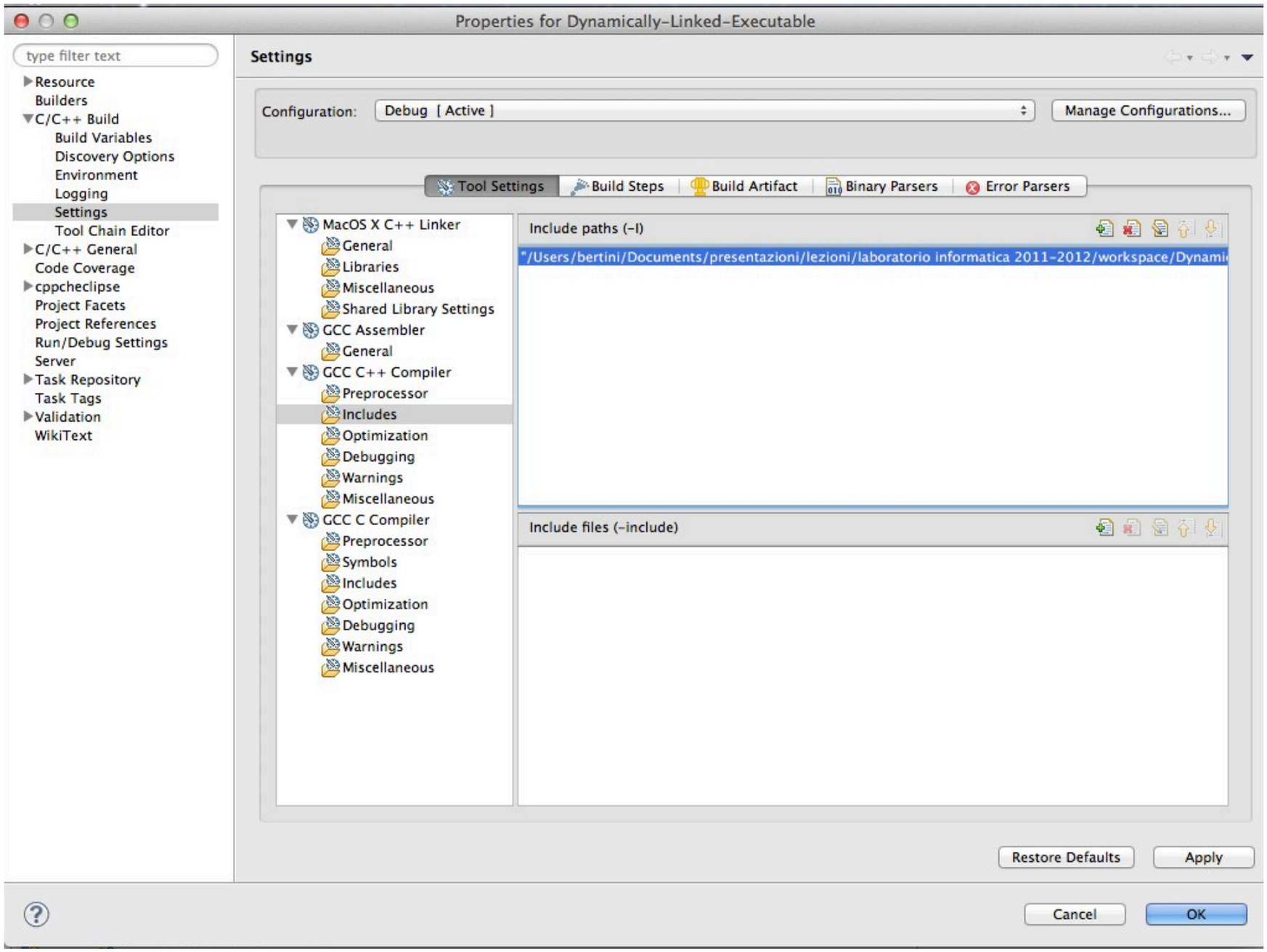
Creating a dynamic library with Eclipse



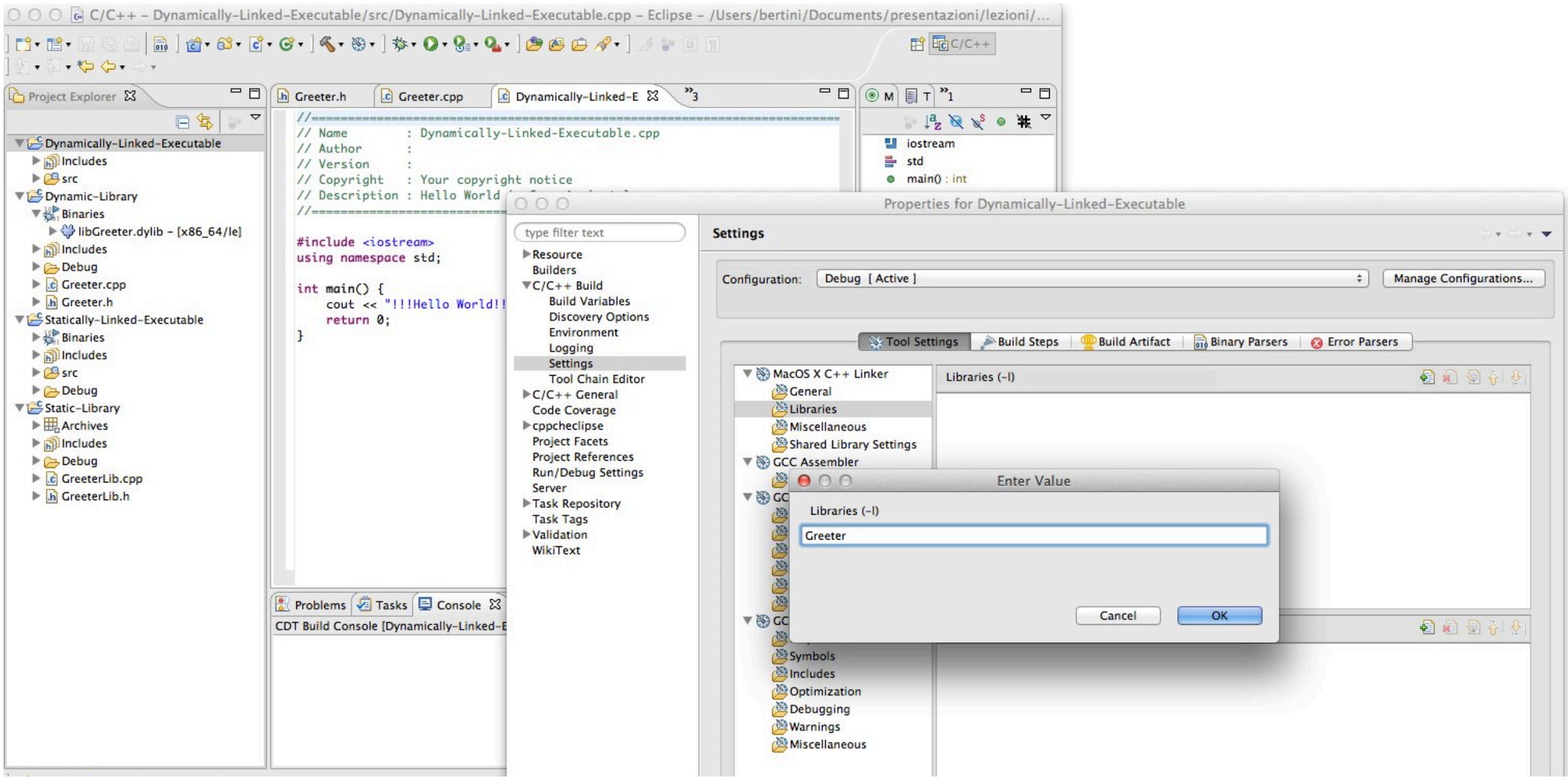
Using a dynamic library with Eclipse

- 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 !)
- Eclipse will use this information to create the required makefile

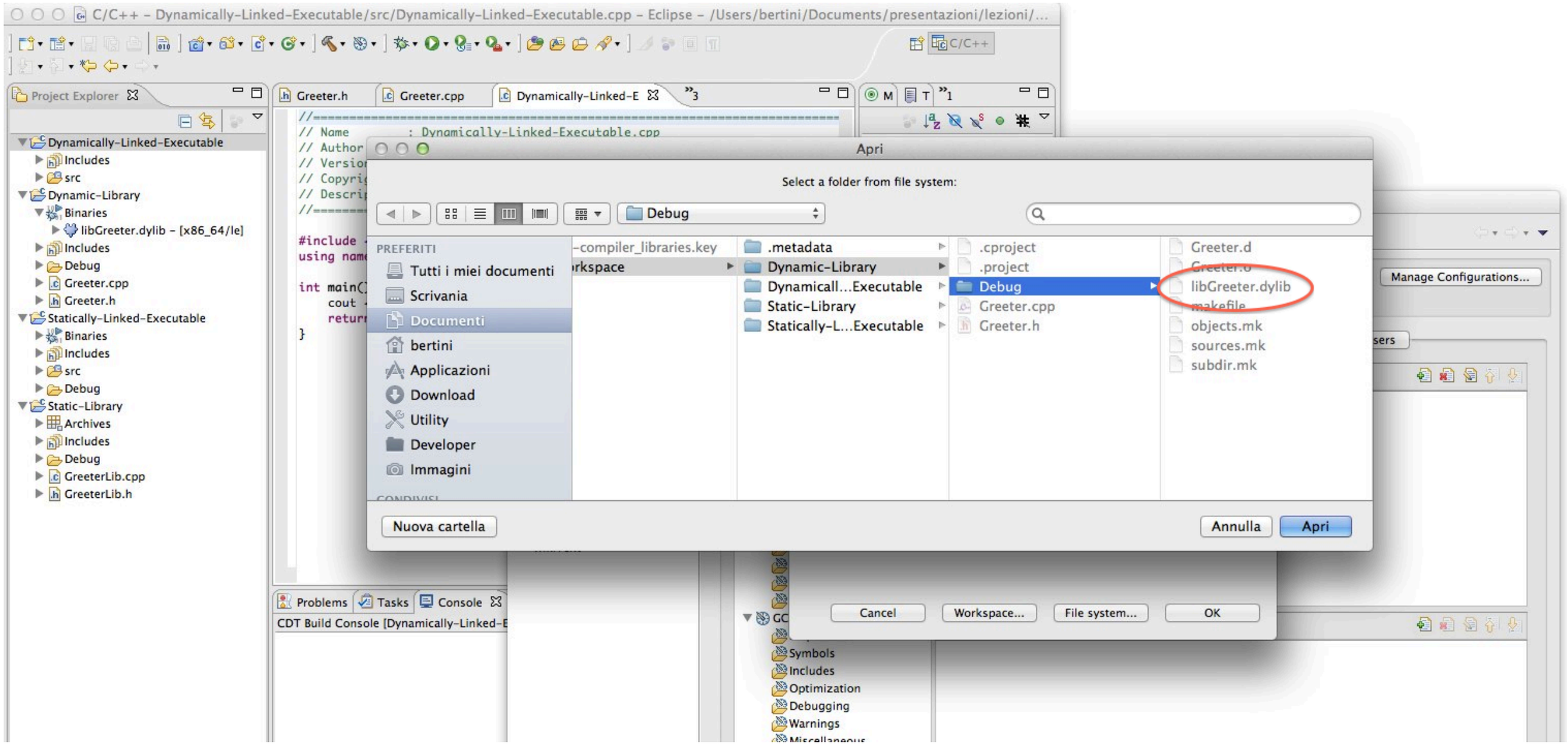
Using a dynamic library with Eclipse



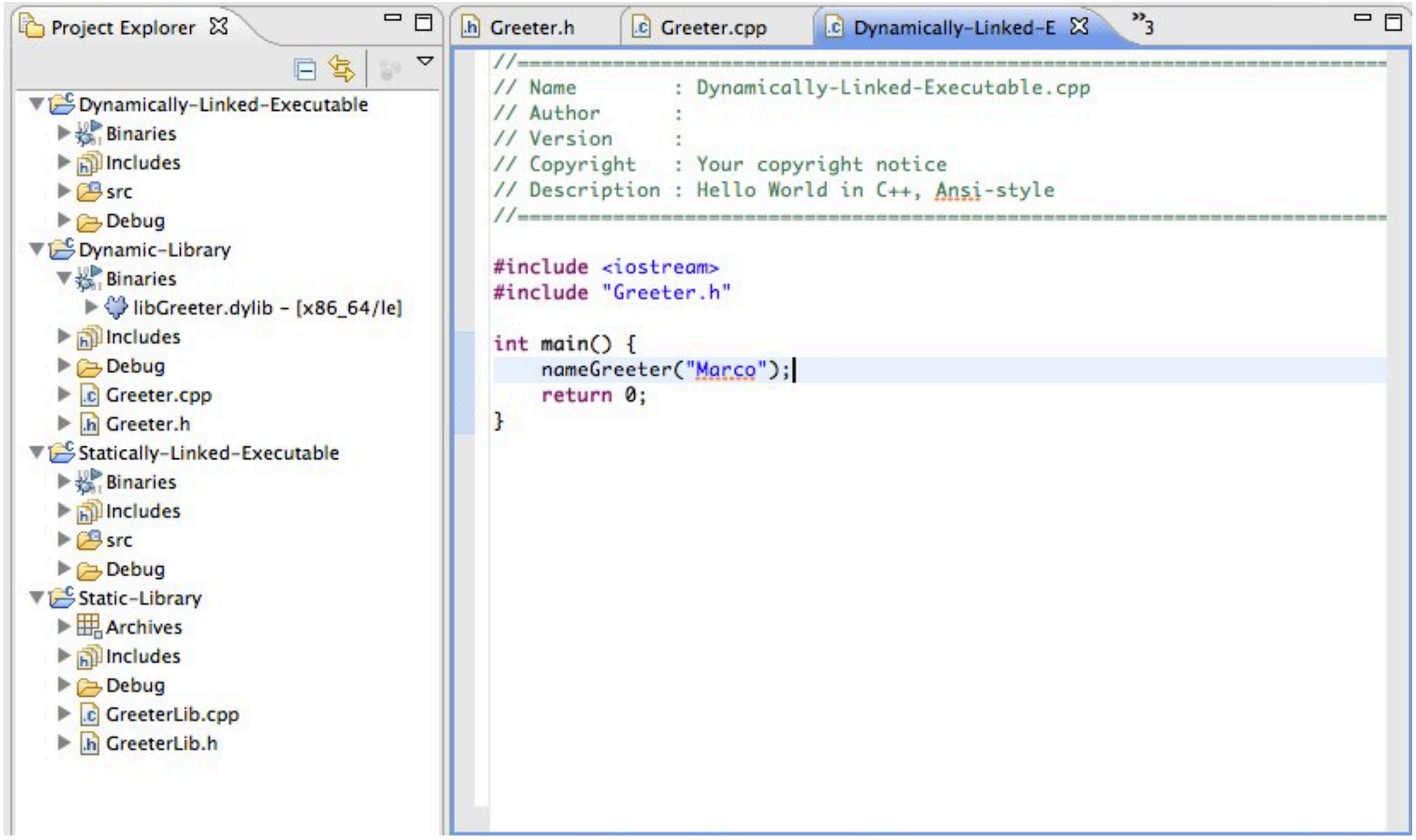
Using a dynamic library with Eclipse



Using a dynamic library with Eclipse



Using a dynamic library with Eclipse



Using a dynamic library with Eclipse

CDT Build Console [Dynamically-Linked-Executable]

```
**** Build of configuration Debug for project Dynamically-Linked-Executable ****

make all
Building file: ../src/Dynamically-Linked-Executable.cpp
Invoking: GCC C++ Compiler
g++ -I"/Users/bertini/Documents/presentazioni/lezioni/laboratorio informatica 2011-2012/workspace/Dynamic-Library" -O0 -g3 -Wall -c -
fmessage-length=0 -MMD -MP -MF"src/Dynamically-Linked-Executable.d" -MT"src/Dynamically-Linked-Executable.d" -o "src/Dynamically-Linked-
Executable.o" "../src/Dynamically-Linked-Executable.cpp"
Finished building: ../src/Dynamically-Linked-Executable.cpp

Building target: Dynamically-Linked-Executable
Invoking: MacOS X C++ Linker
g++ -L"/Users/bertini/Documents/presentazioni/lezioni/laboratorio informatica 2011-2012/workspace/Dynamic-Library/Debug" -o
"Dynamically-Linked-Executable"  ./src/Dynamically-Linked-Executable.o   -lGreeter
Finished building target: Dynamically-Linked-Executable


**** Build Finished ****
```

Program source code is compiled into an object file

The library is linked by the linker

The program is linked and the executable is created

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/l-dll/