Compiling

Matteo Caldana

28/09/2023

Code organization

A typical C++ program is organised in header and source files.

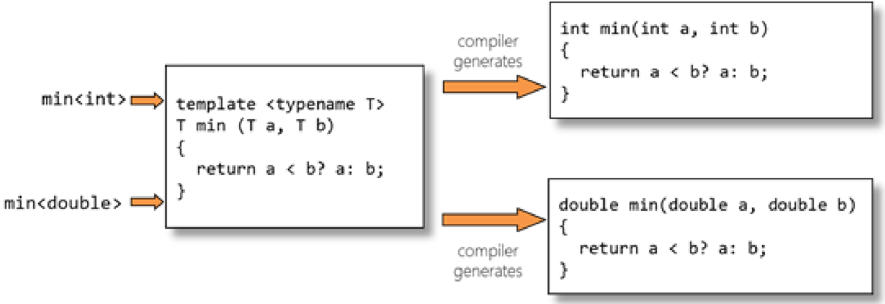
Header files contains the information that describes the public interface of your code: declarations, definition of function class templates, etc. C++ header files have (normally) extension `.hpp` and are eventually stored in special directories with name `include`.

Source files contain the implementations (definitions of variables/functions/methods) and are normally collected under the directory `src`. Only one source files contains the `main()` program.

Libraries can be static or dynamic (shared). For the moment let's think the library as a collection of compiled files that can be used (linked) by an external program.

Template libraries

Template libraries are a special case that consists only of header files, so go in `include/`. Since they are not pre-compiled this makes life much simpler, but compilation time longer. A template function is not *real code* until it is *specialized* (used).



Why the separation into header and source?

The reason is that all source files that uses an item (they call a function for example) need to see the declaration of the item in order to compile. This is done by including the header file containing the declarations. Only one source file will provide the definition of the item: it's the source file whose compilation will produce the actual machine code for that item. Indeed, you cannot have more than one definition (one definition rule, also called odr.)

Remember however that a definition is also a declaration.

Translation unit

A C++ translation unit (also called compilation unit) is formed by a source file and all the (recursively) `include`d header files it contains.
A program is normally formed by more translation units, one and only one of which is the main program.

The important concept to be understood is that during the compilation process each translation unit is treated separately until the last step (linking stage).

The compilation steps (simplified)

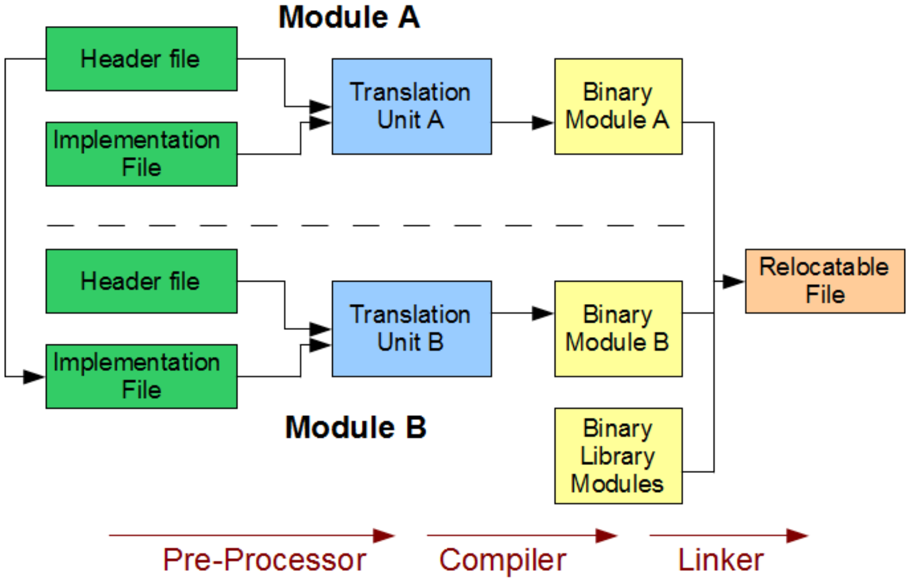
Compiling an executable is in fact a multistage process formed by different components. The main ones are

- Preprocessing. Each translation unit is transformed into an intermediate source by processing C++ directives (`-E` flag);
- Compilation proper. Each preprocessed translation unit is converted into an object file (`-c` flag). Most optimization is done at this stage;
- Linking Object files are assembled and unresolved symbols resolved, eventually by linking external libraries, and an executable is produced.

When you launch the executable, you have an additional step:

- Loading Possible dynamic (shared) library are loaded to complete the linking process. The program is then loaded in memory for execution.

The compilation steps (simplified)



Let's try this on a Hello World

What should a header file contain

Named namespaces	namespace LinearAlgebra
Type declarations	class Matrix{...}
Extern variables declatations	extern double a;
Constant variables	const double pi=4*atan(1.0)
Constant expressions	constexpr double h=2.1
Constexpr functions	constexpr double fun(double x){...}
Enumerations	enum bctype {...}
Function declarations	double norm(...);
Forward declarations	class Matrix;
Includes	#include<iostream>
Preprocessor directives	#ifdef AAA
Template functions	template <class T> fun(T x){...};
Inline functions	inline fun(){...}
constexpr functions	constexpr fun(){...}
automatic functions	auto fun(auto x){...}
type alias	using Real=double;

What a header file should not contain

Function definitions[1]	`double norm(){ ..}`
Method definitions	`double Mat::norm(){ ..}`
Definition of non-constexpr variables	`double bb=0.5;`
Definition of static class members	`double A::b;`
C array definitions	`int aa[3]={1,2,3};`
Array definitions	`std::array<double,3> a{1,2,3};`
Unnamed namespaces	`namespace { ...}`

[1] unless inline, constexpr functions or template functions

Useful flags

Command `g++ -std=c++17 -o prog prog.cc` would execute both *compilation* and *linking* steps.

MAIN g++ OPTIONS (some are in fact preprocessor or linker options)

-g	For debugging	-O[0-3] -Ofast	Optimization level
-Wall, -Wextra	Activates warnings	-Idirname	Directory of header files
-DMACRO	Activate MACRO	-Ldirname	Directory of libraries
-o file	output in file	-lname	link library
-std=c++14	activates c++14 features	-std=c++20	activates c++20 features
-std=c++17	activates c++17 features	-c	create only object file
-S	output assembly code	-E	create only pre-processed files
-v	verbose compile	-g	keep debugging symbols

The default C++ version for g++ versions 9 and 10 is `c++14`. In version 11 is `c++17`.

The compilation process

But in fact the situation is usually more complex, we normally have more than one translation units (source files)

```
g++ -std=c++20 -c a.cpp b.cpp
g++ -std=c++20 -c main.cpp
```

produce the object files `a.o`, `b.o` and `main.o`. Only one of them contains the main program (`int main()...`).
Then,

```
g++ -std=c++20 -o main main.o a.o b.o
```

produces the executable `main` (linking stage).

Each translation unit is compiled separately, even if they are in the same compiler command.

All in one go

Of course it is possible to do all in one go

```
g++ -std=c++20 main.cpp a.cpp b.cpp -o main
```

But it is normally better to keep compilation and linking stages separate. If you modify `a.cpp` you have to recompile only `a.o` and repeat the linking.

Note however, that the compiler will in any case treat the translation units `a.cpp`, `b.cpp` and `main.cpp` separately.

Let's look at the 3 stages in more detail.

The preprocessor

To understand the mechanism of the header files correctly it is necessary to introduce the C preprocessor (`cpp`). It is launched at the beginning of the compilation process and it modifies the source file producing another source file (which is normally not shown) for the actual compilation.

The operations carried out by the preprocessor are guided by directives characterized by the symbol # in the first column. The operations are rather general, and the C preprocessor may be used non only for C or C++ programs but also for other languages like FORTRAN!.

The preprocessor

Very rarely one calls the preprocessor explicitly, yet it may be useful to have a look at what it produces
To do that one may use the option `-E` of the compiler:

```
g++ -E [-DVAR1] [-DVAR2=xx] [-Iincdir] file >pfile
```

Note: `-DXX` and `-I<dirname>` compiler options are in fact **cpp options**.

The first indicates that the preprocessor macro variable `XX` is set, the second indicates a directory where the compiler may look for header files.

Main cpp directives

All preprocessor directives start with a hash (#) at the first column.

```
#include<filename>
```

Includes the content of `filename`. The file is searched fist in the directories possibly indicated with the option `-Idirname`, then in the system directories (like `/usr/include`).

```
#include "filename"
```

Like before, but first the current directory is searched for `filename`, then those indicated with the option `-Idir`, then the system directories.

`#define VAR`

Defines the *macro variable* `VAR`. For instance `#define DEBUG`. You can test if a variable is defined by `#ifdef VAR` (see later). The preprocessor option `-DVAR` is equivalent to put `#define VAR` at the beginning of the file. Yet it overrides the corresponding directive, if present.

`#define VAR=nn`

It assigns value `nn` to the (*macro variable*) `VAR`. `nn` is interpreted as an alphanumeric string. Example: `#define VAR=10`. Not only the test `#ifdef VAR` is positive, but also any occurrence of `VAR` in the following text is replaced by `10`. The corresponding cpp option is `-DVAR=10`.

```
#ifdef VAR
code block
#endif
```

If `VAR` is undefined code block is ignored. Otherwise, it is output to the preprocessed source.

```
#ifndef VAR
code block
#endif
```

If `VAR` is defined code block is ignored. Otherwise, it is output to the preprocessed source.

Special macros

The compiler set special macros depending on the options used, the programming language etc. Some of the macros are compiler dependent, other are rather standard:

- `__cplusplus` It is set to a value if we are compiling with a c++ compiler. In particular, it is set to `201103L` if we are compiling with a C++11 compliant compiler.
- `NDEBUG` is a macros that the user may set it with the `-DNDEBUG` option. It is used when compiling "production" code to signal that one DOES NOT intend to debug the program. It may change the behavior of some utilities, for instance `assert()` is deactivated if `NDEBUG` is set. Also some tests in the standard library algorithms are deactivated. Therefore, you have a more efficient program.

The header guard

To avoid multiple inclusion of a header file the most common technique is to use the header guard, which consists of checking if a macro is defined and, if not, defining it!

```
#ifndef HH_MYMAT0__HH
#define HH_MYMAT0__HH
... Here the actual content
#endif
```

The variable after the `ifndef` (`HH_MYMAT0__HH` in the example) is chosen by the programmer. It should be a long name, so that it is very unlikely that the same name is used in another header file!
Some IDEs generate it for you!

The compilation proper

After the preprocessing phase the translation unit is translated into an `object code`, typically stored in a file with extension `.o`.

Object code, however, is not executable yet. The executable is produced by gathering the functionalities contained in several object files (and/or libraries).

```
g++ -std=c++20 -c -Wall a.cpp b.cpp
```

run preprocessig+compilation proper and produces the object files `a.o` and `b.o`.

The linking process

The process to create an executable from object files is done by calling the linker using *the same name of the compiler used in the compilation process*.

```
g++ main.o a.o b.o -lmylib -o myprogram
```

The linker is called with the same name of the compiler (g++ in this case) so it knows which system libraries to search! Here, it will search the c++ standard library. If you call the standalone linker, called `ld` you need to specify yourself where the c++ standard library resides!

You have to indicate possible other libraries used by your code. In this case the library `libmylib`.

Handle all this complexity

If all this complexity seems a little bit over bearing to you, do not worry, there are tools that help you automate the builds of large projects. Here a couple:

- The make utility is a tool to produce files according to user defined, or predefine, rules. The rules are written on a file, usually called Makefile. See `https://makefiletutorial.com/`, `https://www.gnu.org/software/make/manual/make.html`.
- The CMake is cross-platform software for build automation with support for many IDEs. It is easy to learn since it is a much higher level tool than make. See `https://cmake.org/`.

Back to libraries

Static libraries are the oldest and most basic way of integrating “third party” code. They are basically a collection of object file stored in a single archive. At the linking stage of the compilation processes the symbols (which identify objects used in the code) that are still unresolved (i.e. they have not been defined in that translation unit) are searched in the indicated libraries, and the corresponding code is inserted in the executable.

With shared libraries, the linker just makes sure that the symbols that are still unresolved are indeed provided by the library, with no ambiguities. But the corresponding code is not inserted, and the symbols remains unresolved. Instead, a reference to the library is stored in the executable for later use by the loader.

Advantages and disadvantages of static libraries

PROS

The resulting executable is *self contained*, i.e. it contains all the instructions required for its execution.

CONS

- To take advantage of an update of an external library we need to recompile the code (at least to replicate the linking stage), so we need the availability of the source (or at least of the object files);
- We cannot load symbols dynamically, on the base of decisions taken run-time (it's an advanced stuff, we will deal with it in another lecture);
- The executable may become large.

Advantages and disadvantages of shared libraries

PROS

- Updating a library has immediate effect on all codes linking the library. No recompilation is needed.
- Executable is smaller since the code in the library is not replicated;
- We can load libraries and symbols run time (plugins).

CONS

- Executables depend on the library. (Can't delete the library!)
- You may have different versions of a library. This may cause headaches.
- Code must be compiled with `-fPIC -shared` to be machine independent.
- Must tell the loader where to look for the library

Where does the loader search for shared libraries?

It looks in `/lib`, `/usr/lib`, in all the directories contained in `/etc/ld.conf` and in all `.conf` contained in the `/etc/ld.conf.d/` directory (so the search strategy is different than that of the linker!)

If I want to permanently add a directory in the search path of the loader I need to add it to `/etc/ld.conf`, or add a conf file in the `/etc/ld.conf.d/` directory with the name of the directory, and then launch `ldconfig`).

The command `ldconfig` rebuilds the data base of the shared libraries and should be called every time one adds a new library (of course `apt` does it for you, and moreover `ldconfig` is launched at every boot of the computer).

Note: all this operations require you act as superuser, for instance with the `sudo` command.

Alternative ways of directing the loader

- Setting the environment variable `LD_LIBRARY_PATH`. If it contains a comma-separated list of directory names the loader will first look for libraries on these directories (analogous to `PATH` for executables):

  ```
  export LD_LIBRARY_PATH+=:dir1:dir2
  ```

- With the special flag `-Wl,-rpath=directory` during the compilation of the executable, for instance

  ```
  g++ main.cpp -o main -Wl,-rpath=/opt/lib  -L. -lsmall
  ```

  Here the loader will look in `/opt/lib` before the standard directories. You can use also relative paths.
- Launching the command `sudo ldconfig -n directory` which adds `directory` to the loader search path (superuser privileges are required). This addition remains valid until the next reboot of the computer. **Note**: prefer the other alternatives!

mk

`mk` are set of executables (compiler, linker and bash commands) and shared libraries which constitutes a close environment for the development of software. The toolchain is independent from the hosting OS (although it must be capable of talking with the specific kernel and with the CPU instruction set).

A module is basically a set of instructions that define a specific environment for a specific software: `Bash` environmental variables define paths for executables and libraries.

- Installation path: `"mk" + Modulename + "Prefix"`, *e.g.* `${mkOctavePrefix}`
- Headers: `"mk" + Modulename + "Inc"`, *e.g.* `${mkEigenInc}`
- Libraries: `"mk" + Modulename + "Lib"`, *e.g.* `${mkSuitesparseLib}`

The module system takes also care of defining the environment variable `LD_LIBRARY_PATH`.

How to use a third-party library

Basic compile/link flags:

```
$ g++ -I${mkLibrarynameInc} -c main.cpp
$ g++ -L${mkLibrarynameLib} -llibraryname main.o -o main
```

Warning: by mistake, one can include headers and link against libraries related to different installations/versions of the same library! The compile, link and loading phase may succeed, but the executable may crash, resulting in a very subtle yet painful error to debug!
Warning: By default if the linker finds both the static and shared version of a library it gives precedence to the shared one. If you want to by sure to link with the static version you need to use the `-static` linker option.

Naming scheme of shared libraries (Linux/Unix)

We give some nomenclature used when describing a shared library

- Link name. It's the name used in the linking stage when you use the `-lmylib` option. It is of the form `libmylib.so`. The normal search rules apply. Remember that it is also possible to give the full path of the library instead of the `-l` option.
- `soname` (shared object name). It's the name looked after by the *loader*. Normally it is formed by the link name followed by the version. For instance `libfftw3.so.3`. It is *fully qualified* if it contains the full path of the library.
- real name. It's the name of the actual file that stores the library. For instance `libfftw3.so.3.3.9`

How does it work?

The command `ldd` lists the shared libraries used by an object file.
For example:

```
$ ldd ${mkOctavePrefix}/lib/octave/6.2.0/liboctave.so | grep fftw3
libfftw3.so.3 => /full/path/to/libfftw3.so.3
```

It means that the version of `Octave` I have has been linked (by its developers) against version 3 of the `libfftw3` library, as indicated by the `soname`. Indeed `libfftw3.so` provides a `soname`. If we wish we can check it:

```
$ objdump libx.so.1.3 -p | grep SONAME
SONAME   libfftw3.so.3
```

Which release? Well, lets take a closer look at the file

```
$ ls -l ${mkFftwLib}/libfftw3.so.3
/full/path/to/libfftw3.so.3 -> libfftw3.so.3.3.9
```

Got it?

The executable (`octave`) contains the information on which shared library to load, including version information (its `soname`). This part has been taken care by the developers of `Octave`.

When I launch the program the loader looks in special directories, among which `/usr/lib` for a file that matches the `soname`. This file is typically a symbolic link to the real file containing the library.

If I have a new release of `fftw3` version 3, let's say 3.4.1, I just need to place the corresponding shared library file, reset the symbolic links and automagically `octave` will use the new release (this is what `apt` does when installing a new update in a Debian/Ubuntu system, for example).

No need to recompile anything!

Got it?

Once the library has been located, the symbol must be loaded. To see the symbols contained in a library (or in an object file, or in an executable) you may use the command nm –demangle (possibly piped with grep).

```
$ nm --demangle libopenblas.so | grep dgemm
0000000000123840 T cblas_dgemm
...
```

The T in the second column indicates that the function is actually defined (resolved) by the library. While U is referenced but undefined, meaning you need another library, or object file, where it is defined.

```
$ nm --demangle libopenblas.so | grep tan
                  U atan2@GLIBC_2.2.5
```