This is a sample chapter of
The Meson Manual

You can purchase a full copy at

http://meson-manual.com

Chapter 2
How compilation works

Compiling source code into executables looks fairly simple on the surface but gets more and more complicated the lower down the stack you go. It is a testament to the design and hard work of toolchain developers that most developers don't need to worry about those issues during day to day coding.

There are (at least) two reasons for learning how the system works behind the scenes. The first one is that learning new things is fun and interesting *an sich*. The second one is that having a grasp of the underlying system and its mechanics makes it easier to debug the issues that inevitably crop up as your projects get larger and more complex.

This chapter aims outline how the compilation process works starting from a single source file and ending with running the resulting executable. The information in this chapter is not necessary to be able to use Meson. Beginners may skip it if they so choose, but they are advised to come back and read it once they have more experience with the software build process.

The treatise in this book is written from the perspective of a build system. Details of the process that are not relevant for this use have been simplified or omitted. Entire books could (and have been) written about subcomponents of the build process. Readers interested in going deeper are advised to look up more detailed reference works such as chapters 41 and 42 of [10].

2.1 Basic term definitions

compile time All operations that are done before the final executable or library is generated are said to happen during *compile time*. Some people use the term informally and include linking in compile time. Others are more strict and use the term *link time* to distinguish between the two.

run time All operations that happen once a built executable is run are said to happen during *run time* (sometimes also called *runtime*). In this chapter we are mostly interested in the run time behaviour of symbol resolution via dynamic linking.

source file Source files contain the actual source code that programs are made of. They usually have file name extensions such as `.c`, `.java` or `.cpp`.

header file Some languages have separate header files that contain things such as function and variable declarations (but not their definitions). In `C++` code that uses templates this gets a bit murkier. It is possible, and in fact quite common, to have code inside header files, but for the purposes of this chapter we can mostly ignore it.

object file An object file is the intermediate step between a source file and an executable or library. The compiler converts one source file into one object file, which contains machine executable binary code. An object file is not usable on its own until it is linked to a build target.

compiler A compiler's job is to take source files, parse their contents and generate corresponding binary code. It is also responsible for optimising the output, printing warnings, generating debug information and sometimes even doing static analysis on the source code.

linker The task of taking built object files and dependency libraries and assembling that into a cohesive whole, usually either a shared library or an executable, falls to the linker. Few people need to deal with the linker directly, and on most platforms it is invoked via the compiler.

static linker A static linker is a tool that produces static libraries from object files.

symbol Many things in binary code have names by which they can be identified. Examples include functions and global variables. These names are called *symbols*.

static library A static library is an archive file containing only object files.

shared library A shared library is a fully built piece of code that other programs can use. When a program (or library) is linked against a shared library no code is copied. Instead the linker stores the name of the dependency library in the target it is building to be used later by the dynamic linker at runtime.

executable An executable is a program that can be run.

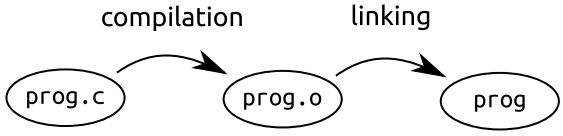

Figure 2.1: The compilation steps needed for a program consisting of one source file.

dynamic linker An executable using shared libraries can not be executed directly. Some process must find the libraries it needs and map all symbols used by the executable to their respective locations. This task is handled by the dynamic linker. It does not come from the compiler toolchain, but is a core component provided by the operating system.

2.2 Building the Hello World application manually

A simple way to get started is to compile a simple program manually. For this we'll use the helloworld application presented in Figure 1.1. As discussed above, the build process can be split into two separate parts: the *compilation step* and the *linking step*. The workflow is visualised in Figure 2.1. The compilation is done by invoking the compiler.

```
$ c++ -c -o hello.o hello.cpp
```

By default most compilers want to compile and link the entire application in one step. We have to use the `-c` command line argument since we only want to compile. The output goes to the object file `hello.o`.

Linking is just as simple.

```
$ c++ -o hello hello.o
```

We don't call the linker binary (which on this platform is called `ld`) but instead use the compiler to do the linking for us. The reason for this becomes fairly obvious if we look at the command line needed to link the program with plain `ld`.

```
ld -o hello \
  -dynamic-linker /lib64/ld-linux-x86-64.so.2 \
  /usr/lib/x86_64-linux-gnu/Scrt1.o \
  /usr/lib/x86_64-linux-gnu/crti.o \
```

```
  /usr/lib/gcc/x86_64-linux-gnu/7/crtbeginS.o \
  hello.o \
12 -L/usr/lib/x86_64-linux-gnu \
  -L/usr/lib/gcc/x86_64-linux-gnu/7/ \
  -lstdc++ -lgcc -lc -lgcc_s \
  /usr/lib/gcc/x86_64-linux-gnu/7/crtendS.o \
  /usr/lib/x86_64-linux-gnu/crtn.o
```

These command line arguments specify all sorts of functionality needed to talk with the core operating system and by the `C++` language runtime. Unless you are working on the compiler toolchain or other such low level component, it is unlikely you'll ever need to deal with the linker manually.

The built executable can now be run.

```
$ ./hello
Hello, world.
```

2.3 Basic symbol resolution

From the developer point of view, compilation is fairly straightforward and easy to comprehend. Source code goes in and a binary artefact comes out. Linking, on the other hand, is a lot more vague. The most user visible operation that happens during linking is *symbol resolution*. In order to understand it, we must first examine what symbols are in the compilation context.

Symbol resolution happens at a very low level, and thus it is necessary to go all the way down to assembly code to understand its behaviour. A simple source file and its corresponding assembly output can be seen in Figure 2.2. Understanding exactly what the individual assembly instructions do is not necessary, a rough understanding of the overall structure is sufficient.

A symbol is nothing more than a string which specifies a name of a thing in the program. To keep things from getting too simple, not all symbols have a name and some names do not have a corresponding symbol. The sample code has three different named elements: the `print_number` and `printf` functions and the `number` variable. The first two of these have a symbol name but the variable name does not. This is because linking only with elements that are in global scope, that is, functions and global variables and constants. Both of these names can be found in the assembly output.

The element that does have a symbol but not a name is the character array `"Number %d\n."`. This may seem surprising given that the character array is only used inside the function just like the `number` variable. What happens behind the scenes is that the compiler elevates the character array to a global constant

```
#include<stdio.h>
void print_number(int number) {
    printf("Number %d.\n", number);
}
```

```
.LC0:
  .string "Number %d.\n"
print_number:
  mov esi, edi
  xor eax, eax
  mov edi, OFFSET FLAT:.LC0
  jmp printf
```


Figure 2.2: A simple `C` function (left) and the result of compiling it to `x86_64` assembly (right).

and gives it a secret symbol name, which in this case is `.LC0`, as can be seen at the beginning of the assembly output. Effectively it is as if the compiler had compiled a program that looks like this:

```
#include<stdio.h>

/* Leading dot removed, because variable names can not
 * have the character "." in them.
 */
const char LC0[] = "Number %d.\n";

void print_number(int number) {
    printf(LC0, number);
}
```

At this point the compiler's job is finished and the object files are handed to the linker. Its job can be most easily understood by looking at the last line of the assembly code which is `jmp printf`. This is the call to the `printf` function which is part of the standard library. Sadly processors do not understand textual labels, they can only jump to specific memory addresses. The main task of the linker is to go through the compiled code and replace all references to symbols with numerical addresses that point to the corresponding functions and global variables. If all symbols used by the program are found the program can be generated and run. In case any piece of code tries to use a symbol that the linker can not find, the linker will abort with an error.

2.4 Static linking

Thus far we have only looked at single executables where all source code is compiled and linked directly. In real world projects this setup is fairly rare. Most

applications use code that has been built separately. A collection of prebuilt code is called a *library*. There are two different kinds of libraries, *static libraries* and *shared libraries* and using code from these on a target is called *static linking* and *shared linking*, respectively. We shall first look at static linking, since it is the simpler of the two.

To demonstrate linking we are going to need two things: a library and an executable using it. We'll create our own library called *messageprinter*. It consists of one file, `messageprinter.c`.

```
#include<stdio.h>

void print_message() {
    printf("I am a library.\n");
}
```

The only thing this function does is print a message to the screen proving that the function has been called. A main program using the library is equally plain.

```
void print_message();

int main(int argc, char **argv) {
    print_message();
    return 0;
}
```

The only thing to note is that `main.c` manually specifies the function prototype at the beginning rather than by including a header. This is merely to simplify the code.

This is all that we need to build and run an executable using static linking. Building the static library takes two commands.

```
$ cc -c -o messageprinter.o messageprinter.c
$ ar csrD libmessageprinter.a messageprinter.o
```

The first command is the familiar compiler invocation. The second command is where the library gets built. It is done with the `ar` command, which is known as the *static linker*. This is actually a misnomer, since `ar` does not do any linking at all. The only thing it does is take all the specified object files and put them together in an archive file. Its behaviour is almost identical to other archive program such as `tar` and `zip`. Because of this the static linker is sometimes called a *static archiver*. The library file name is `libmessageprinter.a`. The

standard way of naming libraries is to have the `lib` prefix and `.a` as the file extension. This is not mandatory, the archive can have any name, but most tools, processes and developers expect this naming scheme so you should use it unless there are strong reasons for doing something else.

The library can be used by adding it on the final executable's link command line.

```
$ cc -c -o main.o main.c
$ cc -o main main.o libmessageprinter.a
$ ./main
I am a library.
```

In addition to passing the library directly, there is an alternative syntax that is used especially for libraries provided by the system.

```
$ cc -o main main.o -L. -lmessageprinter
```

This way of using the library requires two command line arguments. The latter one is `-lmessageprinter` which tells the linker to *find a library called messageprinter, following the standard naming scheme, and link against that.* The standard naming scheme is the one mentioned above. If the library file is not `libmessageprinter.a`, the linker could not find it and linking would fail. By default the linker only does lookups in the *system library directories*. Since our library is not in one of those, we need to add the current directory to the list of lookup directories with the `-L.` command line argument.

The algorithm the linker uses to handle static libraries is straightforward. If it finds that some object file contains a symbol needed by the main program, it will copy out that object file and link all of it with the main program. The behaviour is the same *as if* you had manually specified those object files to be linked like this:

```
$ cc -o main main.o messageprinter.o
```

In this simple case the entire contents of the static library is used. But if the library contains many object files, only the ones whose symbols are actually needed (and their transitive dependencies) end up in the final executable. If only a small fraction of the library's code is needed, this can lead to noticeable space savings in the final executable.

2.5 Shared linking

Building and using a shared library is not very different from static linking.

```
$ cc -o messageprinter.o -fPIC -c messageprinter.c
$ cc -o libmessageprinter.so -shared messageprinter.o
```


The only difference to static linking are the output filename and the two command line arguments. The compiler argument `-fPIC` tells the compiler that the object file will be used in a shared library so it must be built as *position-independent code*. What this means will be explained later in this chapter. On many platforms this argument is not needed as all code is built position-independent by default but we use it here for portability. The linker argument `-shared` tells the linker to produce a shared library as output.

Linking the main program with the shared library is almost identical to using a static library.

```
$ cc -o main main.o libmessageprinter.so
```

You can also use the `-L. -lmessageprinter` syntax, which works in the same way. If you try to run the result, you will get a mysterious crash:

```
$ ./main
./main: error while loading shared libraries:
  libmessageprinter.so:
  cannot open shared object file: No such file or directory
```

This error stems from the main difference between static and shared libraries. Shared libraries are full featured operating system components whereas static libraries are only archives of objects. The former can be used in various ways during runtime but the only thing you can meaningfully do to a static library is link it to an executable or a shared library.

In static linking all object code used by the application is copied to the target executable. In shared linking this does not happen. Instead the shared library's name is written in the executable's *dynamic section*. It contains a list of all external shared libraries required to run the program. No code from the shared library is copied inside the executable. When the program is run it is the responsibility of the operating system's *dynamic linker* to find all shared libraries needed by the program and to resolve all missing symbols. This lookup is done every time the program is run.

Just like the static and shared linkers need to be told where to look up libraries, the dynamic linker needs to be told told where to look up shared libraries. Due to security reasons the current directory is not in the library search path by default[1]. We need to add it to the list with the `LD_LIBRARY_PATH` environment variable.

[1] Just like the current directory is not in `PATH` by default.

```
$ LD_LIBRARY_PATH=. ./main
I am a library.
```


One unexplained question about this program remains about the program's use of `printf`. Since it is a symbol and all symbols need to be resolved before a program can be run (both when shared and static linking), where does that symbol come from? To find this out we need to look inside the produced executable. There are many tools available to inspect the contents of programs. We'll use the `ldd` program that lists all libraries needed by an executable.

```
$ LD_LIBRARY_PATH=. ldd main
linux-vdso.so.1 (0x7ffd3e9f7000)
libmessageprinter.so => ./libmessageprinter.so (0x7f741e7f1000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x7f741e400000)
/lib64/ld-linux-x86-64.so.2 (0x7f741ebf5000)
```

Even though we specified one shared library, the final executable ended up with four of them. `linux-vdso.so.1` is a performance optimisation mechanism that makes certain Linux system calls faster. The second line contains the shared library we just built.

The third one is `libc.so.6`. This is where `printf` actually comes from. This library is called the *C runtime library* and contains all functionality needed by `C` programs, such as `malloc` for allocating memory. It also contains code needed for process startup and teardown. Most programming languages have a similar runtime library that they link dynamically against their programs. There are also languages that don't behave in this manner.

The final entry is `/lib64/ld-linux-x86-64.so`. This is the system's dynamic linker. It may seem bizarre that the dynamic linker, whose job is to find an executable's shared dependency libraries is itself provided as a shared library. The answer to this chicken and egg problem is that the dynamic linker is not a dependency library in the traditional sense, `ldd` merely reports it that way. In reality it is set up as the program's *ELF interpreter*. A detailed description of the issue is out of scope for this book, but interested readers can find more information in the ELF reference documentation [18].

2.6 Linking multiple libraries

Linking multiple libraries is only slightly more complex than only one. Basically the linker will proceed through the items on the link line one by one until all linker targets have been processed. Thus far the behaviour of all operating

systems and toolchains has been almost identical, but this is where they start to differ noticeably.

For the purposes of this discussion, let's assume that we have a project that consists of one main object file and two libraries called `one` and `two`. The main object uses functionality from library `one` which in turn uses functionality from library `two`.

2.6.1 The classical Unix linking model

The sample project would eventually be linked using the following command:

```
$ cc -o main main.o libone.a libtwo.a
```

Here use static libraries. The way the linker goes about its job is that it starts by taking the first argument `main.o`. It processes the file and makes a list of all external symbols that it requires. Then it processes the next argument `libone.a`. For each symbol in the missing symbol list it will try to see if any of the object files inside the library provides it. If yes, it will copy the object file out as described in section 2.4.

Once the linker has satisfied as many missing symbols as it possibly can, the static library is discarded. Any object files that were not needed to satisfy symbols are thrown away. After that the linker will go to the next argument `libtwo.a` and repeat the process. If all symbol requirements were found, the linking step is a success, otherwise an error is raised.

The main problem with this algorithm is that it is fragile and sometimes confusing. It breaks if you get the order of libraries wrong. This will not link:

```
$ cc -o main main.o libtwo.a libone.a
```

The reason is simple. When `libtwo.a` is being processed, none of the symbols it provides are in the list of needed symbols. That means that *everything* in it gets thrown away. When `libone.a` is processed those symbols are added to the list, but they can't be fulfilled any more because `libtwo.a` is gone.

This was a fairly common problem back when Makefiles were written by hand. It is very confusing to be told by the linker that your program has unresolved symbols even though you can clearly see them on the linker command line. This lead to lots of "cargo cult" problem solving where developers would add the same libraries on the command line many times in the hopes that eventually it would work.

Sometimes it is even necessary to have the same dependency library on the command line multiple times. This happens if you have a circular dependency between two libraries. This happens when library A requires symbols from library

B and vice versa. If this ever happens the only reasonable approach is to change the code so the circular dependency is broken. If this is not possible for some reason, then the only workable solution is to tell the linker to first link A, then B and then A again. Or possibly B, A, B depending on how the calling program uses the libraries. In fact for pathological cases there may be an arbitrary number of repetitions needed. It is left as an exercise to the reader to work out how that might come about.

The reason for this behaviour is that linkers were originally designed and implemented in the early 70s. At the time computers were slow and had little memory. Keeping all symbols alive would have required too many resources, and actively reducing the amount of data to keep in memory made sense. Then, as it usually happens, computers got a lot faster so this limitation was no longer an issue, but the behaviour was kept to maintain backwards compatibility.

The most widespread linker in current use that behaves like this is the GNU bfd linker, which is the default on most Linux distributions.

2.6.2 Modern linker model

Modern linkers behave in roughly the same way as the classical Unix linker, except that they don't discard any libraries. This means that symbol resolution happens globally. It does not matter which order the libraries are defined, because the linker will search for symbols in every file.

The exact order in which symbols are resolved depends on each linker. Usually developers do not have to care about it as long as no symbol is repeated. Duplicated symbols are considered an error. Most new linkers behave in this manner, including the Visual Studio linker, macOS linker and the lld linker provided by the LLVM project.

2.7 Which is better, shared or static linking?

This is a common topic of, shall we say, lively debate on the Internet. Both of these approaches have their merits and use cases. Meson does not have a preference, instead it aims to work identically with both library types. Switching between the two library types is simple, as the only change needed is to alter the library target's type.

2.8 Dynamic linker and symbol resolution

Now that we know the difference between static and shared linking we can examine how symbol resolution works in more detail. We will remain at the

conceptual level, though. Readers interested in the actual implementation details are instructed to look up more detailed reference works such as [5].

Resolving symbols in static linking is not particularly complicated. As was discussed in Section 2.3 the compiler will write placeholder code for all symbols outside the current translation unit. In addition it writes a set of *relocation records*. They are merely a list of locations of said placeholders and which symbol's address they should be filled in with. When the final executable is linked, the linker has all the code and thus the addresses of all symbols. It can then overwrite the placeholders with the real addresses.

Dynamic linking is more difficult. Nothing about it known at link time apart from its name. We don't know what address it will end up when the program is run due to *address layout randomisation* or ASLR. This is a security mechanism against various memory corruption vulnerabilities. Whenever a piece of code is loaded into memory, whether it comes from an executable or a shared library, it is placed at a random address in the process' virtual address space. Thus some sort of an indirection mechanism is needed to make things work.

Suppose we build an executable called `proggy` in the current directory and that it uses a shared library `thingy` which resides in directory `subdir`. The program would be built with the following command line invocation:

```
$ gcc -g -o proggy main.c subdir/libthingy.so
```

The linker adds an entry to the executable that it requires `libthingy.so` to run. This can be verified with the `ldd` command.

```
$ ldd proggy
  ...
  subdir/libthingy.so (0x00007f5bfe106000)
```

As can be seen, the entry also contains the subdirectory where the library resides in. To keep things from being too simple and straightforward, this depends on the compiler flags used. If the executable is linked with the alternative link syntax like this:

```
$ gcc -g -o proggyL main.c -Lsubdir -lthingy
```

then only the filename is written in the executable:

```
$ ldd proggy
  ..
  libthingy.so => not found
```

This behaviour is confusing and is probably inherited from the 70s and can not be changed due to backwards compatibility. The solution to this is an entry called *soname*, which is a "virtual file name" that can be defined for each shared library. There are exact rules on how sonames should be determined but Meson will do that automatically.

When an executable that uses shared libraries is run, it is the responsibility of the dynamic linker, sometimes also called a *loader*, which starts with the main executable and the list of sonames that it requires. It will search for shared libraries matching the sonames on the system in a platform specific way. Any libraries required by the found shared libraries are also looked up in the same way. If any of the libraries can not be found, then the process is not run, but instead exits with an error.

Now the dynamic linker is almost at the same position as we were when linking the executable statically. It has all the symbol names and knows the corresponding runtime addresses. It could, in theory, write the actual addresses in the code that has been loaded in memory, but it turns out that this can't be done. All code loaded from files is mapped to memory as *read-only* so the actual code can't be changed. This is due to performance and security reasons. Thus an additional piece of functionality is needed.

For function calls this is done with a data structure called the *procedure linkage table* or *PLT*.[2] A slightly simplified way of looking at it is to consider it as a table of function pointers, one for each symbol needed. Once the table is filled, the code can call any function it needs to execute. Yet, the tables are not filled yet.

The ELF file format used by most unixes is very powerful and flexible and supports many different ways of loading, using and interposing symbols. We shall not look at them in detail, but what is important for this discussion is that symbol loading is done *lazily*. That is, the actual location of any symbol is only determined when someone actually calls it. This also improves program startup times, since it is not uncommon for programs to only use a subset of all symbols at runtime. Symbol lookup takes time, and resolving all symbols up front would be slow. An outline of the lookup process can be seen in Figure 2.3.

Symbol resolution starts by the executable calling a function that resides in some shared library. This is implemented by calling the function pointer in the PLT that corresponds to the desired function. This reduces to calling a function pointer in the PLT (which is an array) with an offset. All of this information was available when the executable was built, so this can be done directly.

As discussed earlier, the PLT does not contain function pointers to the actual code. Instead all pointers in the PLT have been set to point to a symbol resolution

[2]Global variables are looked up in an analogous fashion using a table called *global offset table* or GOT.

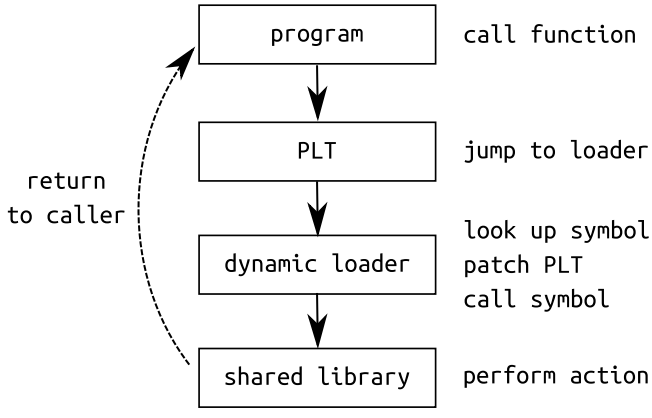

Figure 2.3: How the dynamic linker looks up symbols.

function inside the dynamic loader and to pass it an argument specifying which function in the table initiated the call. The dynamic loader now knows which function was called and can look up its actual runtime address. It writes this address to the PLT entry and then jumps to that function. From the outside it looks as if the program had called the correct function in the shared library directly.

When the function is called a second time, the entry in the PLT already points to the correct location. The expensive resolution operation is thus done only once and only for those symbols that have actually been used.