6
Linking with CMake

You might think that after we have successfully compiled the source code into a binary file, our job as build engineers is done. That's almost the case – binary files contain all the code for a CPU to execute, but the code is scattered across multiple files in a very complex way. Linking is a process that simplifies things and makes machine code neat and quick to consume.

A quick glance at the list of commands will tell you that CMake doesn't provide that many related to linking. Admittedly, `target_link_libraries()` is the only one that actually configures this step. Why dedicate a whole chapter to a single command then? Unfortunately, almost nothing is ever easy in computer science, and linking is no exception.

To achieve the correct results, we need to follow the whole story – understand how exactly a linker works and get the basics right. We'll talk about the internal structure of *object files*, how the relocation and reference resolution works, and what it is for. We'll discuss how the final executable differs from its components and how the process image is built by the system.

Then, we'll introduce you to all kinds of libraries – static, shared, and shared modules. They all are called libraries, but in reality, they are almost nothing alike. Building a correctly linked executable heavily depends on a valid configuration (and taking care of such minute details as **position-independent code** (**PIC**).

We'll learn about another nuisance of linking – the **One Definition Rule** (**ODR**). We need to get the amount of definitions exactly right. Dealing with duplicated symbols can sometimes be very tricky, especially when shared libraries come into play. Then, we'll learn why linkers sometimes can't find external symbols, even when the executable is linked with the appropriate library.

Finally, we'll discover how we can save time and use a linker to prepare our solution for testing with dedicated frameworks.

In this chapter, we're going to cover the following main topics:

- Getting the basics of linking right
- Building different library types
- Solving problems with the One Definition Rule
- The order of linking and unresolved symbols
- Separating `main()` for testing

Technical requirements

You can find the code files that are present in this chapter on GitHub at `https://github.com/PacktPublishing/Modern-CMake-for-Cpp/tree/main/examples/chapter06`.

To build examples provided in this book always use recommended commands:

```
cmake -B <build tree> -S <source tree>
cmake --build <build tree>
```

Be sure to replace placeholders `<build tree>` and `<source tree>` with appropriate paths. As a reminder: **build tree** is the path to target/output directory, **source tree** is the path at which your source code is located.

Getting the basics of linking right

We discussed the life cycle of a C++ program in *Chapter 5*, *Compiling C++ Sources with CMake*. It consists of five main stages – writing, compiling, linking, loading, and execution. After correctly compiling all the sources, we need to put them together into an executable. *Object files* produced in a compilation can't be executed by a processor directly. But why?

To answer this, let's take a look at how a compiler structures an *object file* in the popular ELF format (used by Unix-like systems and many others):

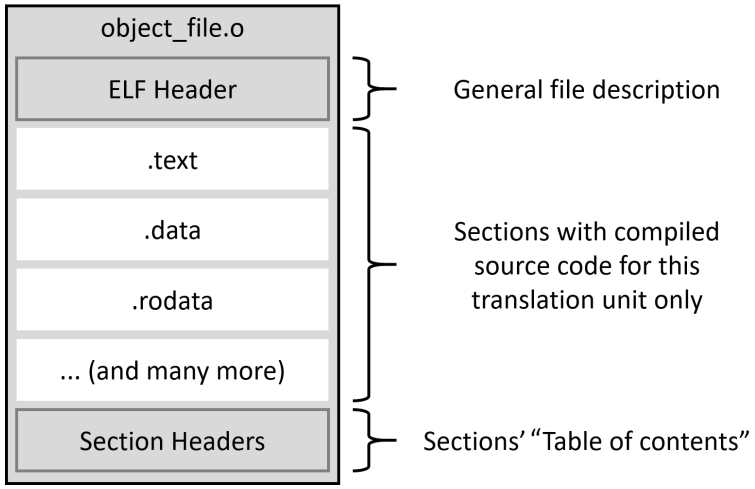

Figure 6.1 – The structure of an object file

The compiler will prepare an *object file* for every unit of translation (for every `.cpp` file). These files will be used to build an in-memory image of our program. *Object files* contain the following elements:

- An ELF header identifying the target operating system, ELF file type, target instruction set architecture, and information on the position and size of two header tables found in ELF files – the program headers table (not present in *object files*) and the section headers table.
- Sections containing information grouped by type (described next).
- A section headers table, containing information about the name, the type, flags, the destination address in memory, the offset in the file, and other miscellaneous information. It is used to understand what sections are in this file and where they are, just like a table of contents.

As the compiler processes your source code, it groups the collected information into a few separate bins, which will be put in their own separate section. Some of them are as follows:

- `.text` section: Machine code, with all the instructions to be executed by the processor
- `.data` section: All values of the initialized global and static objects (variables)

- `.bss` section: All values of the uninitialized global and static objects (variables), which will be initialized to zero on program start
- `.rodata` section: All values of the constants (read-only data)
- `.strtab` section: A string table containing all constant strings such as *Hello World* that we put in our basic `hello.cpp` example
- `.shstrtab` section: A string table containing the names of all the sections

These groups very closely resemble the final version of the executable, which will be put in the RAM to run our application. However, we can't just load this file to memory as it is. This is because every *object file* has its own set of sections. If we were to just concatenate them together, we'd run into all sorts of issues. We'd be wasting a lot of space and time, as we'd need many more pages of RAM. Instructions and data would be much harder to copy to a CPU cache. An entire system would have to be much more complex and would waste precious cycles jumping around many (possibly tens of thousands) of `.text`, `.data`, and other sections during runtime.

So, what we'll do instead is take each section of the *object file* and put it together with the same type of section from all other *object files*. This process is called **relocation** (that's why the ELF file type is `Relocatable` for *object files*). Apart from just bringing appropriate sections together, it has to update internal associations in the file – that is, addresses of variables, functions, symbol table indexes, or string table indexes. All of these values are local to the *object file*, and their numbering starts from zero. When we bundle files together, we need to offset these values so that they are pointing at the correct addresses in the combined file.

Figure 6.2 shows relocation in action – the `.text` section is relocated, `.data` is being built from all linked files, and `.rodata` and `.strtab` will follow (for simplicity, the figure doesn't contain headers):

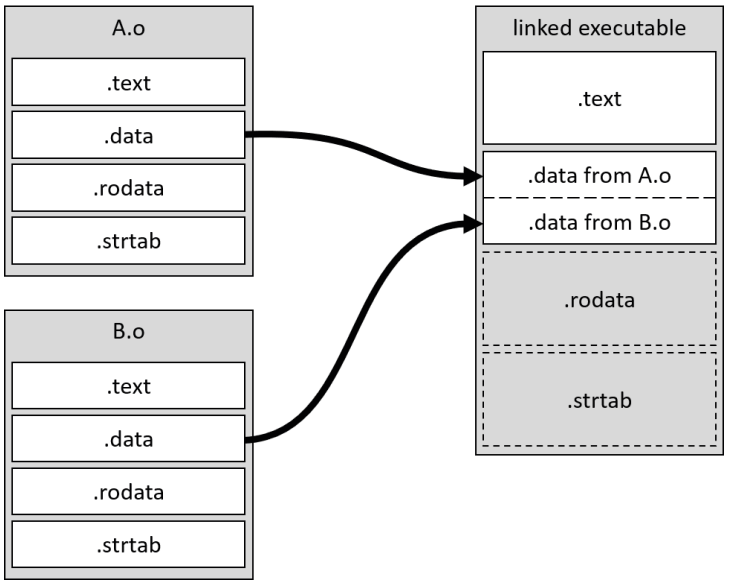

Figure 6.2 – The relocation of the .data section

Secondly, a linker needs to **resolve references**. Whenever a piece of code from one translation unit references a symbol defined in another (such as through including its header or by using the `extern` keyword), the compiler reads the declaration and *trusts* that the definition is somewhere out there and will be provided at a later time. A linker is responsible for collecting such *unresolved references* to external symbols, finding and filling the addresses at which they reside after merging into the executable. *Figure 6.3* shows a simple example of reference resolution:

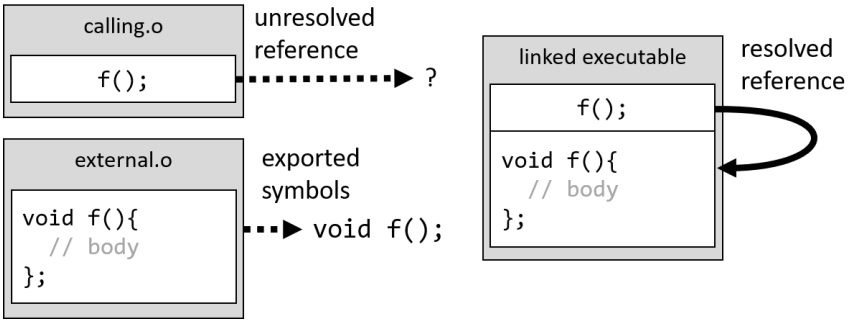

Figure 6.3 – A reference resolution

This part of the linking can be a source of problems if a programmer is unaware of how it works. We may end up with unresolved references that won't find their external symbols, or the opposite – we provided too many definitions and the linker doesn't know which one to pick.

The final *executable file* looks very similar to the *object file*; it contains relocated sections with resolved references, a section headers table, and of course, the ELF Header describing the whole file. The main difference is the presence of the Program Header (as pictured in *Figure 6.4*).

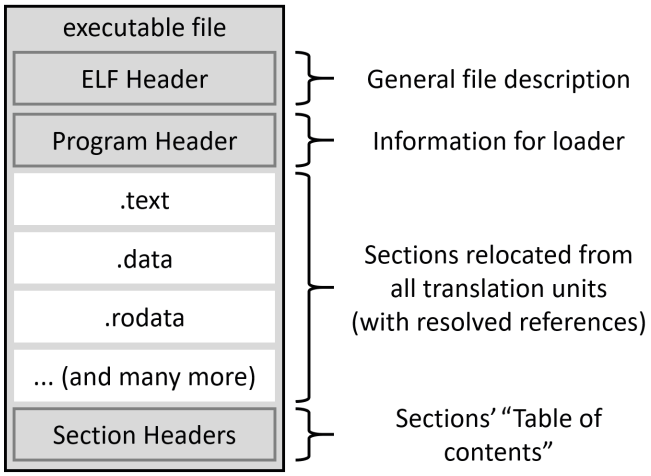

Figure 6.4 – The structure of the executable file in ELF

The Program Header is placed right after the ELF Header. A system loader will read this header to create a process image. The header contains some general information and a description of the memory layout. Each entry in the layout represents one fragment of memory called a **segment**. Entries specify which sections will be read, in what order, to which addresses in the virtual memory, what their flags are (read, write, or execute), and a few other useful details.

Object files may also be bundled in a library, which is an intermediate product that can be used in a final executable or another library. In the next section, we'll discuss three types of libraries.

Building different library types

After source code is compiled, we might want to avoid compiling it again for the same platform or even share it with external projects wherever possible. Of course, you could just simply provide all of your *object files* as they were originally created, but that has a few downsides. It is harder to distribute multiple files and add them individually to a buildsystem. It can be a hassle, especially if they are numerous. Instead, we could simply bring all *object files* into a single object and share that. CMake helps greatly with this process. We can create these libraries with a simple `add_library()` command (which is consumed with the `target_link_libraries()` command). By convention, all libraries have a common prefix, `lib`, and use system-specific extensions that denote what kind of library they are:

- A static library has a `.a` extension on Unix-like systems and `.lib` on Windows.
- Shared libraries have a `.so` extension on Unix-like systems and `.dll` on Windows.

When building libraries (static, shared, or shared modules), you'll often encounter the name *linking* for this process. Even CMake calls it that in the build output of the `chapter06/01-libraries` project:

```
[ 33%] Linking CXX static library libmy_static.a
[ 66%] Linking CXX shared library libmy_shared.so
[100%] Linking CXX shared module libmy_module.so
[100%] Built target module_gui
```

Contrary to how it may seem, a linker isn't used to create all of the preceding libraries. There are exceptions to performing relocation and reference resolution. Let's take a look at each library type to understand how each works.

Static libraries

To build a static library, we can simply use the command we already saw in previous chapters:

```
add_library(<name> [<source>...])
```

The preceding code will produce a static library if the `BUILD_SHARED_LIBS` variable isn't set to `ON`. If we want to build a static library regardless, we can provide an explicit keyword:

```
add_library(<name> STATIC [<source>...])
```

What are static libraries? They are essentially a collection of raw *object files* stored in an archive. On Unix-like systems, such archives can be created by the `ar` tool. Static libraries are the oldest and most basic mechanism to provide a compiled version of code. Use them if you want to avoid separating your dependencies from the executable, at the price of the executable increasing in size and used memory.

The archive may contain some additional indexes to speed up the final linking process. Each platform uses its own methods to generate those. Unix-like systems use a tool called `ranlib` for this purpose.

Shared libraries

It's not surprising to learn that we can build shared libraries with the `SHARED` keyword:

```
add_library(<name> SHARED [<source>...])
```

We can also do it by setting the `BUILD_SHARED_LIBS` variable to `ON` and using the short version:

```
add_library(<name> SHARED [<source>...])
```

The difference from static libraries is significant. Shared libraries are built using a linker, and they will perform both stages of linking. This means that we'll receive a file with proper section headers, sections, and a section header table (*Figure 6.1*).

Shared libraries (also known as shared objects) can be shared between multiple different applications. An operating system will load a single instance of such a library into memory with the first program that uses it, and all subsequently started programs will be provided with the same address (thanks to the complex mechanisms of virtual memory). Only the `.data` and `.bss` segments will be created separately for every process consuming the library (so that each process can modify its own variables without affecting other consumers).

Thanks to this approach, the overall memory usage in the system is better. And if we're using a very popular library, we might not need to ship it with our program. Chances are that it's already available on the target machine. However, if that's not the case, a user is expected to explicitly install it before running the application. This opens up the possibility of some issues when an installed version of a library is different from expected (this type of problem is called *dependency hell*; more information can be found in the *Further reading* section).

Shared modules

To build shared modules, we need to use the `MODULE` keyword:

```
add_library(<name> MODULE [<source>...])
```

This is a version of a shared library that is intended to be used as a plugin loaded during runtime, rather than something that is linked with an executable during compilation. A shared module isn't loaded automatically with the start of the program (like regular shared libraries). This only happens when a program explicitly requests it by making a system call such as `LoadLibrary` (Windows) or `dlopen()/dlsym()` (Linux/macOS).

You shouldn't try to link your executable with a module, as this isn't guaranteed to work on all platforms. If you need to do that, use regular shared libraries.

Position-independent code

All sources for shared libraries and modules should be compiled with a *position-independent code* flag enabled. CMake checks the `POSITION_INDEPENDENT_CODE` property of targets and appropriately adds compiler-specific compilation flags such as `-fPIC` for `gcc` or `clang`.

PIC is a bit of a confusing term. Nowadays, programs are already *position-independent* in a sense, in that they use virtual memory to abstract away actual physical addresses. When calling a function, a CPU uses a **memory management unit** (**MMU**) to translate a virtual address (starting from `0` for every process) to a physical address that was available at the time of allocation. These mappings don't have to point to consecutive physical addresses or follow any other specific order.

PIC is about mapping symbols (references to functions and global variables) to their runtime addresses. During compilation of a library, it is not known which processes might use it. It's not possible to predetermine where in the virtual memory the library will be loaded or in what order. This, in turn, means that the addresses of the symbols are unknown, as is their relative position to the library's machine code.

To deal with that, we need another level of indirection. PIC will add a new section to our output – the **Global Offset Table** (**GOT**). Eventually, this section will become a segment containing runtime addresses for all the symbols needed by shared libraries. The position of the GOT relative to the `.text` section is known during linking; therefore, all symbol references can be pointed (through an offset) to a placeholder GOT at that time. The actual values pointing to symbols in memory will only be filled when an instruction accessing a referenced symbol is first executed. At that time, a loader will set up that particular entry in the GOT (this is where the term *lazy loading* comes from).

Shared libraries and modules will have the `POSITION_INDEPENDENT_CODE` property automatically set to `ON` by CMake. However, it is important to remember that if your shared library is linked against another target, such as a static or object library, you need to set this property on that target too. Here's how:

```
set_target_properties(dependency_target
                      PROPERTIES POSITION_INDEPENDENT_CODE
                      ON)
```

Failing to do so will get you into trouble with CMake, as this property is by default checked for conflicts in a manner described in the *Dealing with conflicting propagated properties* section of *Chapter 4*, *Working With Targets*.

Speaking of symbols, there's another problem to discuss. The next section is about name collisions leading to ambiguity and inconsistency in definitions.

Solving problems with the One Definition Rule

Phil Karlton was right on point when he said the following:

> *"There are two hard things in computer science: cache invalidation and naming things."*

Names are difficult for a few reasons – they have to be precise, simple, short, and expressive at the same time. That makes them meaningful and allows programmers to understand the concepts behind the raw implementation. C++ and many other languages impose one more requirement – many names have to be unique.

This is manifested in a few different ways. A programmer is required to follow the ODR. This says that in the scope of a single translation unit (a single `.cpp` file), you are required to *define* it exactly once, even if you *declare* the same name (of a variable, function, class type, enumeration, concept, or template) multiple times.

This rule is extended to the scope of an entire program for all variables you effectively use in your code and non-inlined functions. Consider the following example:

chapter06/02-odr-fail/shared.h

```
int i;
```

chapter06/02-odr-fail/one.cpp

```
#include <iostream>
#include "shared.h"

int main() {
  std::cout << i << std::endl;
}
```

chapter06/02-odr-fail/two.cpp

```
#include "shared.h"
```

chapter06/02-odr-fail/two.cpp

```
cmake_minimum_required(VERSION 3.20.0)
project(ODR CXX)
set(CMAKE_CXX_STANDARD 20)
add_executable(odr one.cpp two.cpp)
```

As you can see, it's very straightforward – we created a `shared.h` header file used in two separate translation units:

- `one.cpp`, which simply prints `i` to the screen
- `two.cpp`, which does nothing except include the header

We then link the two into a single executable and receive the following error:

```
[100%] Linking CXX executable odr
/usr/bin/ld: CMakeFiles/odr.dir/two.cpp.o:(.bss+0x0): multiple
definition of 'i'
; CMakeFiles/odr.dir/one.cpp.o:(.bss+0x0): first defined here
collect2: error: ld returned 1 exit status
```

You can't define these things twice. However, there's a notable exception – types, templates, and extern inline functions can repeat their definitions in multiple translation units if they are exactly the same (that is, they have the same sequence of tokens). We can prove that by replacing a simple definition, `int i;`, with a definition of a class:

chapter06/03-odr-success/shared.h

```
struct shared {
  static inline int i = 1;
};
```

Then, we use it like so:

chapter06/03-odr-success/one.cpp

```
#include <iostream>
#include "shared.h"

int main() {
  std::cout << shared::i << std::endl;
}
```

The other two files, `two.cpp` and `CMakeLists.txt`, remain the same, as in the `02odrfail` example. Such a change will allow the linking to succeed:

```
-- Build files have been written to: /root/examples/
chapter06/03-odr-success/b
[ 33%] Building CXX object CMakeFiles/odr.dir/one.cpp.o
[ 66%] Building CXX object CMakeFiles/odr.dir/two.cpp.o
[100%] Linking CXX executable odr
[100%] Built target odr
```

Alternatively, we can mark the variable as local to a translation unit (it won't be exported outside of the *object file*). To do so, we'll use the `static` keyword, like so:

chapter06/04-odr-success/shared.h

```
static int i;
```

All other files will remain the same, as in the original example, and linking will still succeed. This, of course, means that the variable in the preceding code is stored in separate memory for each translation unit, and changes to one won't affect the other.

Dynamically linked duplicated symbols

The ODR rule works exactly the same for static libraries as it does for *object files*, but things aren't so clear when we build our code with `SHARED` libraries. A linker will allow duplicated symbols here. In the following example, we'll create two shared libraries, `A` and `B`, with one `duplicated()` function and two unique `a()` and `b()` functions:

chapter06/05-dynamic/a.cpp

```
#include <iostream>
void a() {
  std::cout << "A" << std::endl;
}
void duplicated() {
  std::cout << "duplicated A" << std::endl;
}
```

The second implementation file is almost an exact copy of the first:

chapter06/05-dynamic/b.cpp

```
#include <iostream>
void b() {
  std::cout << "B" << std::endl;
}
void duplicated() {
  std::cout << "duplicated B" << std::endl;
}
```

Now, let's use each function to see what happens (we'll declare them locally with `extern` for simplicity):

chapter06/05-dynamic/main.cpp

```
extern void a();
extern void b();
```

```
extern void duplicated();

int main() {
  a();
  b();
  duplicated();
}
```

The preceding code will run unique functions from each library and then call a function defined with the same signature in both dynamic libraries. What do you think will happen? Would the linking order matter in this case? Let's test it for two cases:

- `main_1` linked with the `a` library first
- `main_2` linked with the `b` library first

Here's the code for such a project:

chapter06/05-dynamic/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.20.0)
project(Dynamic CXX)

add_library(a SHARED a.cpp)
add_library(b SHARED b.cpp)

add_executable(main_1 main.cpp)
target_link_libraries(main_1 a b)

add_executable(main_2 main.cpp)
target_link_libraries(main_2 b a)
```

After building and running both executables, we'll see the following output:

```
root@ce492a7cd64b:/root/examples/chapter06/05-dynamic# b/main_1
A
B
duplicated A
root@ce492a7cd64b:/root/examples/chapter06/05-dynamic# b/main_2
A
```

```
B
duplicated B
```

Aha! So, a linker does care about the order of the linked libraries. This may create some confusion if we aren't careful. In practice, naming collisions aren't as rare as they seem.

There are some exceptions to this behavior; if we define locally visible symbols, they will take precedence over those available from dynamically linked libraries. Adding the following function to `main.cpp` will change the last line of output of both binaries to **duplicated MAIN**, as shown here:

```
#include <iostream>
void duplicated() {
  std::cout << "duplicated MAIN" << std::endl;
}
```

Always take great care when exporting names from libraries, as you're bound to encounter name collisions sooner or later.

Use namespaces – don't count on a linker

The concept of namespaces was invented to avoid such weird problems and deal with the ODR in a manageable way. It comes as no surprise that it is recommended to wrap your library code in a namespace named after the library. This way, we can escape all the problems of duplicated symbols.

In our projects, we might experience situations where one shared library is linking another and then another in a lengthy chain. These aren't that rare, especially in more complex setups. It is important to remember that simply linking one library to another doesn't imply any kind of namespace inheritance. Symbols in each link of this chain remain unprotected, kept in the namespaces in which they were originally compiled.

The quirks of a linker are interesting and useful to know on a couple of occasions, but let's talk about a not-so-uncommon problem – what to do when correctly defined symbols go missing without an explanation.

The order of linking and unresolved symbols

A linker can often seem whimsical and start complaining about things for no apparent reason. This is an especially difficult ordeal for programmers starting out who don't know their way around this tool. It's no wonder, since they usually try to avoid touching build configuration for as long as they possibly can. Eventually, they're forced to change something (perhaps add a library they worked on) in the executable, and all hell breaks loose.

Let's consider a fairly simple dependency chain – the `main` executable depends on the `outer` library, which depends on the `nested` library (containing the necessary `int b` variable). Suddenly, an inconspicuous message appears on the programmer's screen:

```
outer.cpp:(.text+0x1f): undefined reference to 'b'
```

This isn't such a rare diagnostic – usually, it means that we forgot to add a necessary library to the linker. But in this case, the library is actually added correctly to the `target_link_libraries()` command:

chapter06/06-order/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.20.0)
project(Order CXX)

add_library(outer outer.cpp)
add_library(nested nested.cpp)

add_executable(main main.cpp)
target_link_libraries(main nested outer)
```

What then!? Very few errors can be as infuriating to debug and understand. What we're seeing here is an incorrect order of linking. Let's dive into the source code to figure out the reason:

chapter06/06-order/main.cpp

```
#include <iostream>
extern int a;
int main() {
  std::cout << a << std::endl;
}
```

The preceding code seems easy enough – we'll print an `a` external variable, which can be found in the `outer` library. We're declaring it ahead of time with the `extern` keyword. Here is the source for that library:

chapter06/06-order/outer.cpp

```
extern int b;
int a = b;
```

This is quite simple too – `outer` is depending on the `nested` library to provide the `b` external variable, which gets assigned to the `a` exported variable. Let's see the source of `nested` to confirm that we're not missing the definition:

chapter06/06-order/nested.cpp

```
int b = 123;
```

So indeed, we have provided the definition for `b`, and since it's not marked as local with the `static` keyword, it's correctly exported from the `nested` target. As we saw previously, this target is linked with the `main` executable in `CMakeLists.txt`:

```
target_link_libraries(main nested outer)
```

So where does the `undefined reference to 'b'` error come from?

Resolving undefined symbols works like this – a linker processes the binaries from left to right. As the linker iterates through the binaries, it will do the following:

1. Collect all undefined symbols exported from this binary and store them for later
2. Try to resolve undefined symbols (collected from all binaries processed so far) with symbols defined in this binary
3. Repeat this process for the next binary

If any symbols remain undefined after the whole operation is completed, the linking fails.

This is the case in our example (CMake puts the *object files* of the executable target before the libraries):

1. We processed `main.o`, got an undefined reference to `a`, and collected it for future resolution.
2. We processed `libnested.a`, no undefined references were found, so there was nothing to resolve.
3. We processed `libouter.a`, got an undefined reference to `b`, and resolved a reference to `a`.

We did correctly resolve the reference to the `a` variable, but not for `b`. All we need to do is reverse the order of linking so that `nested` comes after `outer`:

```
target_link_libraries(main outer nested)
```

Another less elegant option is to repeat the library (which is useful for cyclic references):

```
target_link_libraries(main nested outer nested)
```

Finally, we can try using linker-specific flags such as `--start-group` or `--end-group`. Go to the documentation of your linker for details, as these specifics are outside of the scope of this book.

Now that we know how to solve common problems, let's talk about how we could use the linker to our advantage.

Separating main() for testing

As we established so far, a linker enforces the ODR and makes sure that all external symbols provide their definitions in the process of linking. One interesting problem that we might encounter is the correct testing of the build.

Ideally, we should test exactly the same source code that is being run in production. An exhaustive testing pipeline should build the source code, run its tests on produced binary, and only then package and distribute the executable (without the tests themselves).

But how do we actually make this happen? Executables have a very specific flow of execution, which often requires reading command-line arguments. C++'s compiled nature doesn't really support pluggable units that can be temporarily injected into the binary for test purposes only. It seems like we'll need a very complex approach to solve this.

Luckily, we can use a linker to help us deal with this in an elegant manner. Consider extracting all logic from your program's `main()` to an external function, `start_program()`, like so:

chapter06/07-testing/main.cpp

```
extern int start_program(int, const char**);
int main(int argc, const char** argv) {
  return start_program(argc, argv);
}
```

It's reasonable to skip testing this new `main()` function now; it is only forwarding arguments to a function defined elsewhere (in another file). We can then create a library containing the original source from `main()` wrapped in a new function – `start_program()`. In this example, I'm going to use a simple program to check whether the command-line argument count is higher than `1`:

chapter06/07-testing/program.cpp

```
#include <iostream>
int start_program(int argc, const char** argv) {
  if (argc <= 1) {
    std::cout << "Not enough arguments" << std::endl;
    return 1;
  }
  return 0;
}
```

We can now prepare a project that builds this application and links together those two translation units:

chapter06/07-testing/CMakeLists.cpp

```
cmake_minimum_required(VERSION 3.20.0)
project(Testing CXX)

add_library(program program.cpp)

```

```
add_executable(main main.cpp)
target_link_libraries(main program)
```

The `main` target is just providing the required `main()` function. It's the `program` target that contains all the logic. We can now test it by creating another executable with its own `main()` containing the test logic.

In a real-world scenario, frameworks such as **GoogleTest** or **Catch2** will provide their own `main()` method that can be used to replace your program's entry point and run all the defined tests. We'll dive deep into the subject of actual testing in *Chapter 8*, *Testing Frameworks*. For now, let's focus on the general principle and write our own tests in another `main()` function:

chapter06/07-testing/test.cpp

```
#include <iostream>
extern int start_program(int, const char**);
using namespace std;
int main() {
  auto exit_code = start_program(0, nullptr);
  if (exit_code == 0)
    cout << "Non-zero exit code expected" << endl;

  const char* arguments[2] = {"hello", "world"};
  exit_code = start_program(2, arguments);
  if (exit_code != 0)
    cout << "Zero exit code expected" << endl;
}
```

The preceding code will call `start_program` twice, with and without arguments, and check whether the returned exit codes are correct. This unit test leaves much to be desired in terms of clean code and elegant testing practices, but at least it's a start. The important thing is that we have now defined `main()` twice:

- In `main.cpp` for production use
- In `test.cpp` for test purposes

We'll add the second executable to the bottom of our `CMakeLists.txt` now:

```
add_executable(test test.cpp)
target_link_libraries(test program)
```

This creates another target, which is linked against the exact same binary code as the production, but it grants us the freedom to call all exported functions however we like. Thanks to this, we can run all code paths automatically and check whether they work as expected. Great!

Summary

Linking in CMake does seem simple and insignificant, but in reality, there's much more to it than meets the eye. After all, linking executables isn't as simple as putting puzzle pieces together. As we learned about the structure of *object files* and libraries, we discovered that things need to move around a bit before a program is runnable. These things are called sections and they have distinct roles in the life cycle of the program – store different kinds of data, instructions, symbol names, and so on. A linker needs to combine them together in the final binary accordingly. This process is called relocation.

We also need to take care of symbols – resolve references across all the translation units and make sure that nothing's missing. Then, a linker can create the program header and add it to the final executable. It will contain instructions for the system loader, describing how to turn consolidated sections into segments that make up the runtime memory image of the process.

We also discussed three different kinds of libraries (static, shared, and shared modules), and we explained how they differ and which scenarios fit some better than others. We also touched on the subject of PIC – a powerful concept that allows for the lazy binding of symbols.

The ODR is a C++ concept, but as we already know, it's heavily enforced by linkers. After introducing this subject, we briefly explored how to deal with the most basic symbol duplication, in both static and dynamic libraries. This was followed by some short advice to use namespaces wherever possible and not to rely on a linker too much when it comes to preventing symbol collisions.

For such a seemingly straightforward step (CMake offers only a few commands dedicated to a linker), it sure has a lot of quirks! One tricky thing to get right is the order of linking, especially when libraries have nested dependencies. We now know how to handle some basic situations and what other methods we could research to deal with more complex ones.

Lastly, we investigated how to take advantage of a linker to prepare our program for testing – by separating the `main()` function into another translation unit. This enabled us to introduce another executable, which ran tests against the exact same machine code that will be run in production.

Now that we know how to link, we can retrieve external libraries and use them in our CMake projects. In the next chapter, we'll study how to manage dependencies in CMake.

Further reading

For more information on the topics covered in this chapter, you can refer to the following:

- *The structure of ELF files:*

  `https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`

- *The CMake manual for* `add_library()`:

  `https://cmake.org/cmake/help/latest/command/add_library.html`

- *Dependency hell:*

  `https://en.wikipedia.org/wiki/Dependency_hell`

- *The differences between modules and shared libraries:*

  `https://stackoverflow.com/questions/4845984/difference-between-modules-and-shared-libraries`

7

Managing Dependencies with CMake

It doesn't really matter whether your solution is big or small; as it matures, you'll eventually decide to bring in external dependencies. It's important to avoid the costs of creating and maintaining code using prevailing business logic. This way, you can devote your time to things that matter to you and your customers.

External dependencies are used not only to provide frameworks and features and solve quirky problems. They can also play an important part in the process of building and controlling the quality of your code – whether it is in the form of special compilers such as **Protobuf** or testing frameworks such as **GTest**.

Whether you're working with open source projects or using projects written by other developers in your company, you still need a good, clean process to manage external dependencies. Solving this on your own would take countless hours of setup and a lot of additional support work later. Fortunately, CMake does an excellent job in accommodating different styles and historical approaches to dependency management while keeping up with the constant evolution of industry-approved standards.