## CMake入门06-链接阶段：

### 测试环境：

**Windows 10（Ubuntu）+MinGw（gcc/g++/Ninja.exe）+ Visual Studio 2019 +Clion（克利翁）**

**插图以及内容来自书籍《Modern CMake For C++》**

**辅助工具：ChatGPT**

### 链接阶段涉及到的cmake命令：

target\_link\_libraries()

将目标文件与库文件进行链接，将库文件链接到目标文件中，使得目标文件可以使用库文件中的函数。

### 本节主要的内容：

* 链接的基本知识；
* 构建不同的库（静态库，共享库，共享模块）
* 使用唯一性规则解决问题
* 链接的顺序和未找到符号
* 分离 main() 进行测试

#### 编译后的目标文件

让我们看一下编译器如何以流行的 ELF 格式（由类 Unix 系统和许多其他系统使用）构建目标文件：

图示

描述已自动生成

编译器将为每个翻译单元（每个 .cpp 文件）准备一个目标文件。这些文件将用于构建我们程序的内存映像。目标文件包含以下元素：

* 标识目标操作系统的 ELF 标头、ELF 文件类型、目标指令集体系结构以及有关 ELF 文件中两个头表的位置和大小的信息——程序头表（不存在于目标文件中）和节头表.
* 包含按类型分组的信息的部分（如下所述）。
* 节头表，包含有关名称、类型、标志、内存中的目标地址、文件中的偏移量和其他杂项信息的信息。它用于了解此文件中有哪些部分以及它们在哪里，就像目录一样。

当编译器处理您的源代码时，它将收集到的信息分组到几个单独的容器中，这些容器将放在它们自己的单独部分中。其中一些如下：

.text section 机器代码，包含处理器要执行的所有指令;

.data section 初始化的全局和静态对象（变量）的所有值

.bss section 未初始化的全局和静态对象（变量）的所有值，将在程序启动时初始化为零

.rodata section 常量的所有值（只读数据）

.strtab section 包含所有常量字符串的字符串表，例如我们放入基本 hello.cpp 示例中的 Hello World

.shstrtab section 包含所有节名称的字符串表

可以直接运行吗？

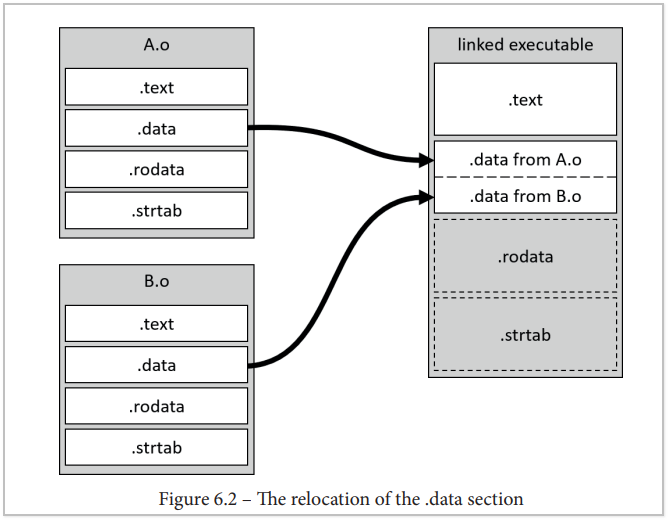
这些组非常类似于可执行文件的最终版本，它将放在 RAM 中以运行我们的应用程序。但是，我们不能只是按原样将此文件加载到内存中。 这是因为每个对象文件都有自己的一组节。如果我们只是将它们连接在一起，我们会遇到各种各样的问题。我们会浪费大量的空间和时间，因为我们需要更多的RAM页面。指令和数据将更难复制到 CPU 缓存。整个系统必须复杂得多，并且会浪费宝贵的周期在运行时跳过许多（可能是数万个）.text，.data和其他部分。

编译后的目标文件，可以存在在库中，或是直接链接进可执行文件；

#### 链接：

##### 重定位：

因此，我们要做的是获取对象文件的每个部分，并将其与所有其他对象文件中相同类型的部分放在一起。此过程称为重定位（这就是为什么 ELF 文件类型对于对象文件是可重定位的）。



由于目标文件在编译时不知道其最终的内存地址，链接器需要将目标文件中的指令和数据的地址调整为最终的实际地址，这个过程就是重定位。相关的指令和数据的地址信息被保存在目标文件的重定位表(Relocation Table)中，链接器会根据这个表将目标文件中的符号地址替换成最终的地址

##### 符号解析

其次，链接器需要解析引用。每当一个翻译单元中的一段代码引用另一个翻译单元中定义的符号时（例如，通过包含其标头或使用 extern 关键字），编译器都会读取声明并相信该定义在某个地方，并且稍后会提供。链接器负责收集对外部符号的此类未解析引用，在合并到可执行文件中后查找并填充它们所在的地址。图 6.3 显示了参考分辨率的简单示例：

图示

描述已自动生成

多个源文件产生相同的函数或变量名时，链接器需要解决符号冲突问题，将不同的符号名引用指向正确的声明。这个过程就是符号解析，它通过查找符号表(Symbol Table)来找到函数和变量的声明，以便将其对应的引用与定义进行匹配和链接。

可能导致的问题：

1. 可能会得到找不到其外部符号的未解析引用
2. 我们提供了太多定义，链接器不知道该选择哪一个。

##### 对齐(Alignment)

链接器需要将链接的目标文件进行对齐，以保证可执行文件的正确性。

##### 压缩(Compression)

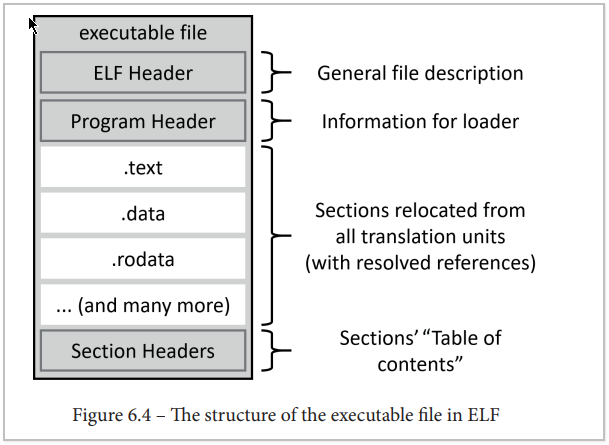
有些链接器可以对代码段进行代码压缩，以减小可执行文件的尺寸。

##### 加载(Loading)

链接器将可执行文件加载到内存中，供操作系统调用执行。

##### 最终的可执行文件

最终的可执行文件看起来与目标文件非常相似;它包含带有解析引用的重新定位的部分，一个部分标题表，当然还有描述整个文件的ELF标题。主要区别在于程序头的存在（如图 6.4 所示）。



程序头位于 ELF 标头之后。系统加载程序将读取此标头以创建进程映像。程序头包含一些常规信息和内存布局的说明。布局中的每个条目都表示一个称为段的内存片段。条目指定将读取哪些部分、按什么顺序读取、读取到虚拟内存中的哪些地址、它们的标志是什么（读取、写入或执行）以及其他一些有用的详细信息。

目标文件也可以捆绑在库中，库是可以在最终可执行文件或其他库中使用的中间产品。

#### 总结：程序生成过程

文本

描述已自动生成

### 构建不同的库类型

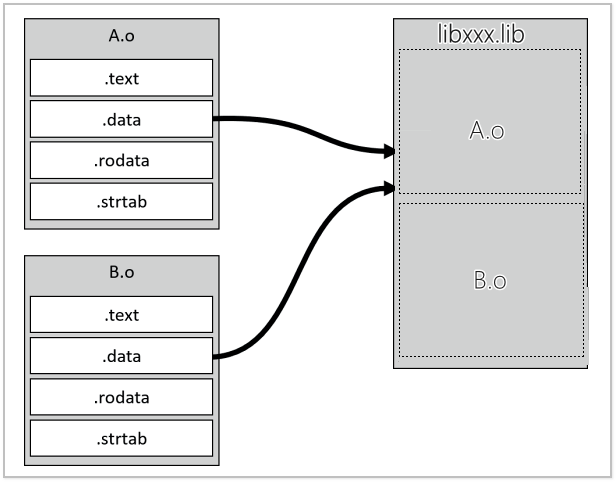
编译源代码后，我们可能希望避免在同一平台上再次编译它，甚至尽可能与外部项目共享它。当然，您可以简单地提供最初创建的所有对象文件，但这有一些缺点。分发多个文件并将它们单独添加到构建系统中比较困难。这可能很麻烦，特别是如果它们很多。相反，我们可以简单地将所有对象文件合并到单个对象中并共享。CMake 对这个过程有很大帮助。我们可以使用一个简单的 add\_library（） 命令（与 target\_link\_libraries（） 命令一起使用）创建这些库。按照惯例，所有库都有一个通用的前缀 lib，并使用特定于系统的扩展来表示它们是哪种类型的库：

* 静态库在类Unix系统上具有.a扩展名，在Windows上具有.lib扩展名。
* 共享库在类 Unix 系统上具有 .so 扩展名，在 Windows 上具有.dll扩展名

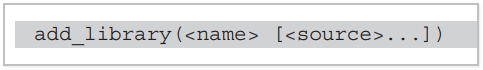
#### 静态库

静态库是在编译阶段生成的。静态库是一组已经编译好的目标文件的集合，这些目标文件通常是由源代码编译得到的，在编译时被链接器静态地链接到可执行文件中，因此可以在运行时直接使用，不需要额外的动态链接。

在 C++ 中，静态库通常由一组 .o 或 .obj 文件生成，这是在编译源代码时生成的目标文件。然后，使用静态库管理器工具将这些目标文件组合成一个 .lib 或 .a 文件，作为静态库的输出。



方式1：



未指定关键字默认是静态库，或是指定是STATIC

方式2：



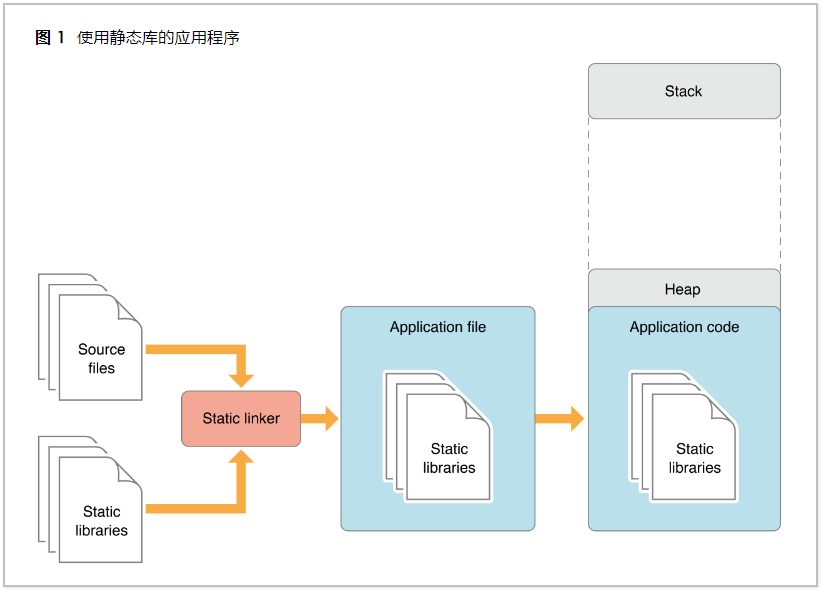
方式3：

如果 BUILD\_SHARED\_LIBS 变量未设置为 ON，则上述代码将生成静态库

缺点：

* 最古老的机制；
* 执行文件变大；
* 占用内存变多；
* 不能在程序运行时被更新

存档可能包含一些额外的索引，以加快最终链接过程。



来自Apple官网；

#### 共享库

动态库是在链接阶段完成的。在C++中，动态库是一种可重定位的目标文件，它包含了一组函数和数据，可以在运行时被加载到内存中。

决定应用程序性能的两个重要因素是它们的启动时间和内存占用量。

减小应用程序可执行文件的大小并在启动后最大限度地减少其内存使用量可使应用程序启动速度更快并在启动后使用更少的内存。

使用动态库而不是静态库可以减少应用程序的可执行文件大小。

它们还允许应用程序仅在需要时而不是在启动时延迟加载具有特殊功能的库。此功能有助于进一步缩短启动时间并提高内存使用效率。

方式1：



方式2：

如果 BUILD\_SHARED\_LIBS 变量设置为 ON，则上述代码将生成动态库

与静态库的区别很大。共享库是使用链接器构建的，它们将执行链接的两个阶段。

这意味着我们将收到一个包含正确节标题、节和节标题表的文件（图 6.1）。

共享库的原理：

共享库（也称为共享对象）可以在多个不同的应用程序之间共享。

操作系统会将此类库的单个实例与第一个使用它的程序一起加载到内存中，

并且所有随后启动的程序都将提供相同的地址（由于虚拟内存的复杂机制）。

仅为每个使用库的进程单独创建 .data 和 .bss 段（以便每个进程可以修改自己的变量而不影响其他消费者）

动态库是加载在堆中

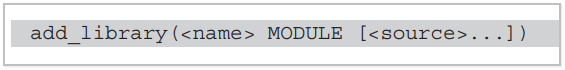
图示

描述已自动生成

来自Apple官网；

#### 3.共享模块

方式1：



这是一个共享库的版本，旨在用作在运行时加载的插件，而不是在编译期间与可执行文件链接的东西。

共享模块不会在程序启动时自动加载（如常规共享库）。

这仅在程序通过进行系统调用（例如 LoadLibrary (Windows) 或 dlopen()/dlsym() (Linux/macOS)）明确请求它时发生。

您不应该尝试将您的可执行文件与模块链接，因为这不能保证在所有平台上都有效。如果您需要这样做，请使用常规共享库。

#### 4.与位置无关的代码

代表编译出来的目标代码是位置无关的，这意味着可以在内存中的任意位置加载和运行该目标文件，而不需要其他支持。

这种代码是与位置关系无关的挂载方式，不管放在哪里，都能够正确地工作。

例如，在编译动态链接库时，应使用-PIC选项。这会告诉编译器在生成的目标文件中添加必要的标记，以便可以将它们装载到内存的任意位置，

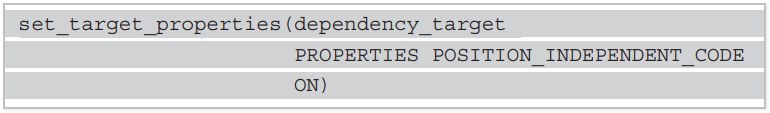
从而使可执行代码块能够正确地引用库函数。

这种技术可以用来防止在运行时出现“代码地址冲突”错误，并提高了可执行代码的安全性和可移植性。

总之，如果将代码标记为位置无关的，可能会使编译后的二进制文件变大。

但是，它提高了代码的兼容性，使得可以在不同的操作系统和处理器架构上使用同一个库。因此，它在开发跨平台应用程序时非常有用。

设置的方法：



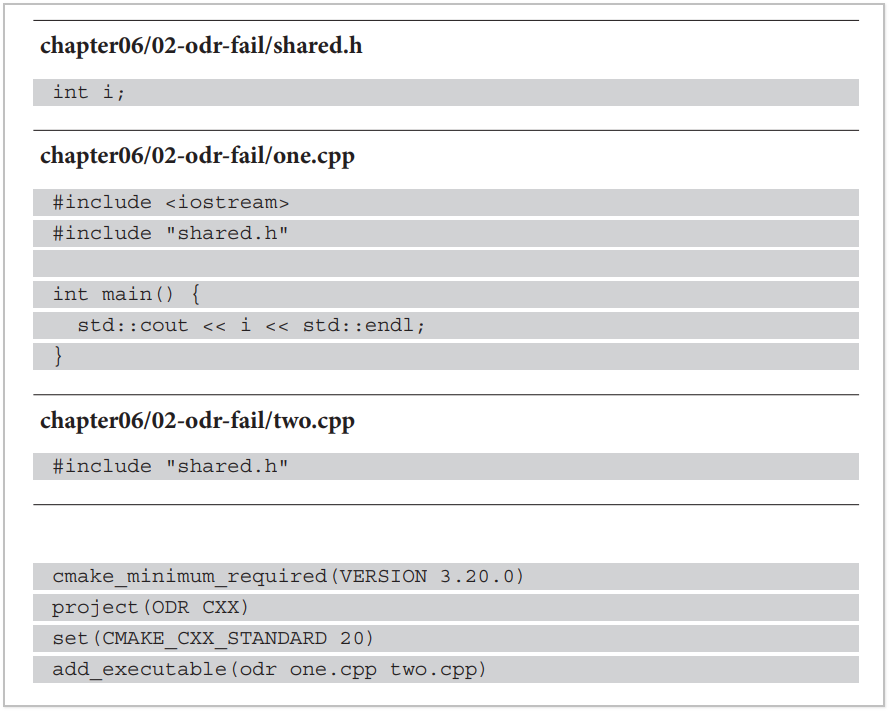
### 使用唯一性规则解决问题

C++ 和许多其他语言强加了一项要求——许多名称必须是唯一的。

这表示在单个翻译单元（单个 .cpp 文件）的范围内，您需要恰好定义一次，

即使您声明相同的名称（变量、函数、类类型、枚举、概念或模板）多次。

错误的方式：



正确的方式1：

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

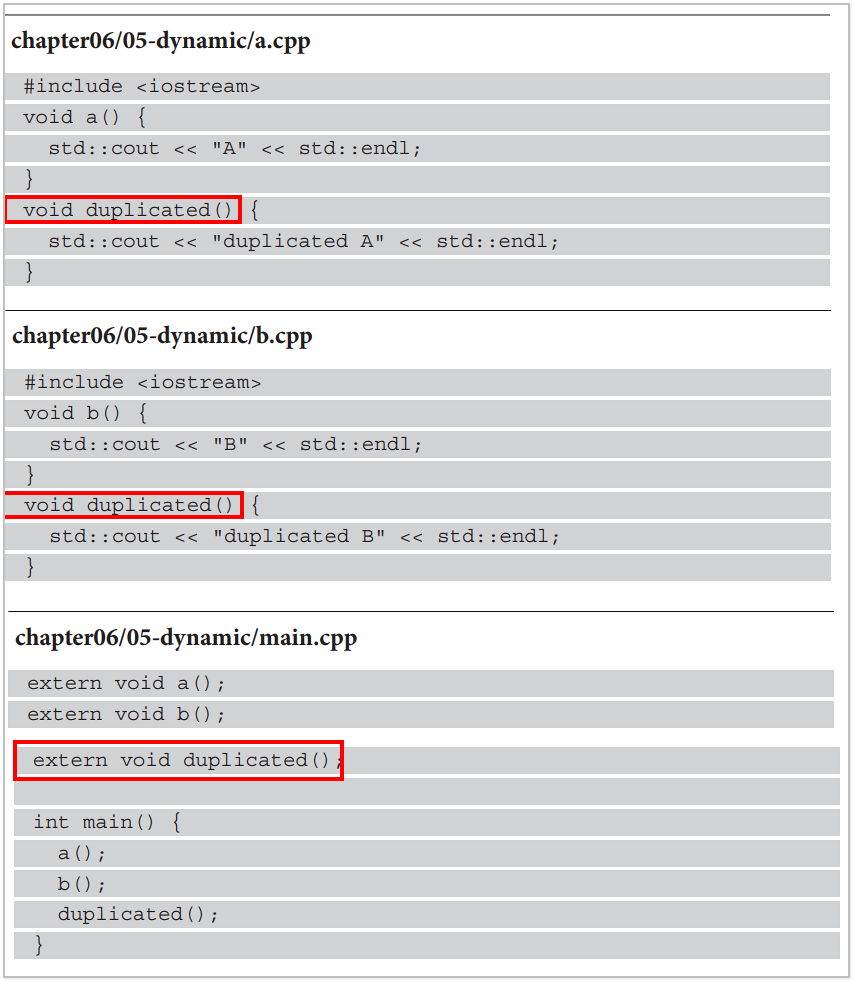
方式2：

图形用户界面, 表格

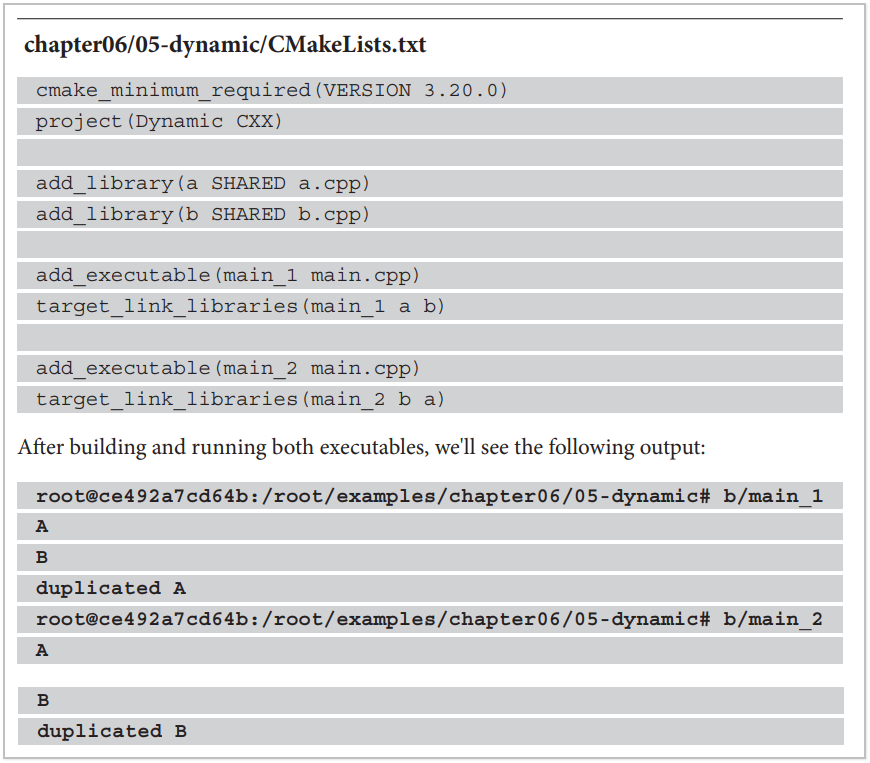
中度可信度描述已自动生成

#### 1.动态链接的重复符号

例子：



链接顺序不同导致的问题：



如果我们定义局部可见的符号，它们将优先于动态链接库中可用的符号。

将以下函数添加到 main.cpp 会将两个二进制文件的最后一行输出更改为重复的 MAIN，如下所示：

图形用户界面, 文本, 应用程序

描述已自动生成

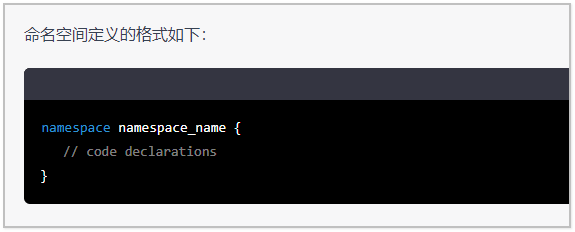
#### 2.使用命名空间——不要指望链接器

在我们的项目中，我们可能会遇到这样的情况：一个共享库链接另一个共享库，然后在一个冗长的链中链接另一个共享库。这些并不罕见，

命名空间的概念的发明是为了避免这种奇怪的问题，

建议将库代码包装在以库命名的命名空间中。

这样，我们可以避免重复符号的所有问题。



### 链接的顺序和未找到符号

**target\_link\_libraries()**函数中链接库的顺序是非常重要的，因为库之间可能存在依赖关系。

应该尽可能地将被依赖的库文件放在后面，以确保连接器能够正确地解决所有未解决的符号。

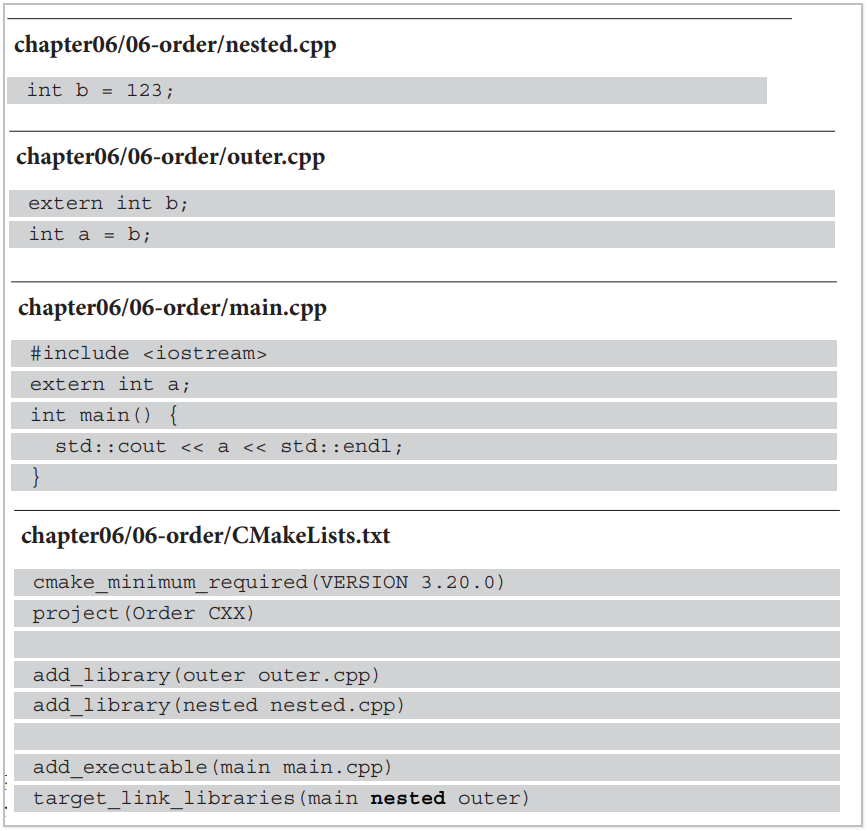
**思考：这个target\_link\_libraries应该怎么写？**

图示

描述已自动生成

来自Apple官网；

例子：



让我们考虑一个相当简单的依赖链 – 【主可执行文件】依赖于【外部】库，【外部】库依赖于【嵌套库】（包含必要的 int b 变量）

图示

描述已自动生成

Draw BY MO；

链接器的解析流程：

1. 收集从此二进制文件导出的所有未定义符号并存储它们以供以后使用
2. 尝试使用此二进制文件中定义的符号解析未定义的符号（从到目前为止处理的所有二进制文件中收集）
3. 对下一个二进制文件重复此过程

target\_link\_libraries(main nested outer)

解析顺序是从左到右解析的；

我们处理了main.o，获得了对a的未定义引用，并收集了它以供将来解决。

我们处理了libnested.a，没有找到未定义的引用，所以没有什么需要解决的。

我们处理了libouter.a，得到了对b的未定义引用，并解析了对a的引用。

需要把被依赖的库放在依赖的库后面

处理方式1：

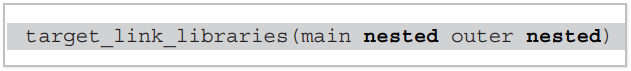
我们需要做的就是颠倒链接的顺序，使【嵌套】在【外部】之后：

图形用户界面, 文本, 应用程序, 聊天或短信

描述已自动生成

处理方式2：

另一个不太优雅的选择是重复库（这对于循环引用很有用）：



方法3：

最后，我们可以尝试使用特定于链接器的标志，例如 --start-group 或 --end-group



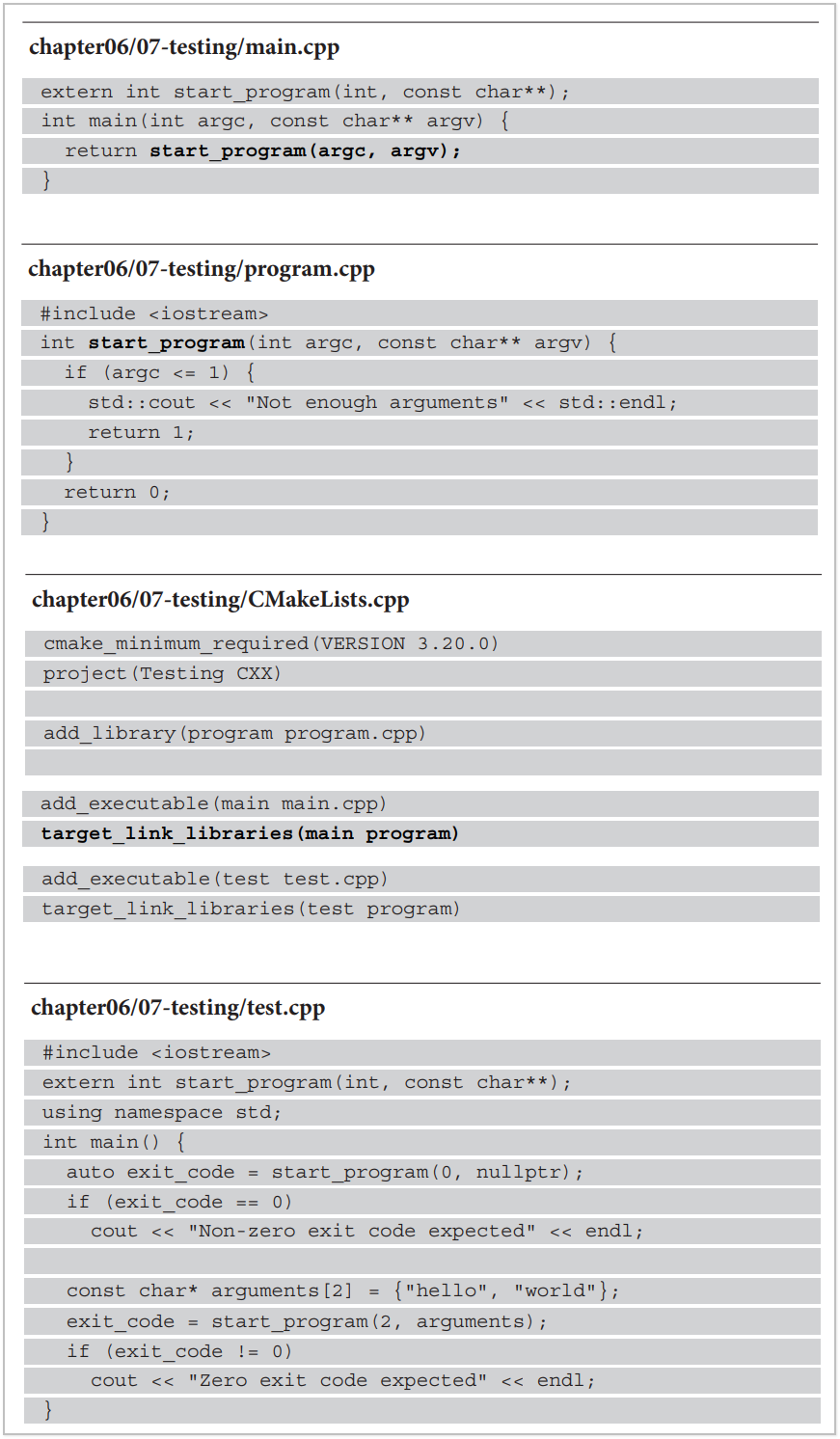
图形用户界面, 文本, 应用程序

描述已自动生成

### 分离 main() 进行测试

我们应该测试在生产环境中运行的完全相同的源代码。

详尽的测试管道应构建源代码，在生成的二进制上运行其测试，然后才打包和分发可执行文件



### 总结