## CMake入门05-编译阶段：

#### 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**

#### 本节主要的内容：

* 编译的基础
* 预处理器配置
* 配置优化器-性能优化；
* 管理编译过程；

#### 编译的基础

##### 什么是编译

在 C++ 中，我们依赖于静态编译——整个程序在执行之前必须被翻译成本地代码

编译可以粗略地描述为将用高级编程语言编写的指令转换为低级机器代码的过程。

这使我们能够使用抽象概念（如类和对象）创建应用程序，而无需为特定于处理器的汇编语言的繁琐细节而烦恼。

我们不需要直接使用 CPU 寄存器，不需要思考汇编里面的短跳或长跳（JUMP指令），也不需要管理堆栈帧。

使得编程语言更具表现力、可读性、安全性，并且培养了更易于维护的代码（但仍然尽可能提高性能）。

文本

中度可信度描述已自动生成

##### 静态语言和动态语言的区别？

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

##### 创建和运行C++程序的过程？

1. 设计应用程序并编写源代码；
2. 将单个.cpp实现文件（称为翻译单元）编译为目标文件。
3. 在单个可执行文件中将目标文件链接在一起，并添加所有其他依赖项——动态库和静态库。
4. 要运行该程序，操作系统将使用称为 loader 的工具将其机器代码和所有必需的动态库映射到虚拟内存。然后，加载器读取程序头以检查程序的启动位置并将控制权移交给代码。
5. C++运行时启动;执行一个特殊的\_start函数来收集命令行参数和环境变量。它启动线程，初始化静态符号，并注册清理回调。只有这样，它才会调用 main（），它由程序员填充代码。

由于我们的编译方式，在程序运行时可能会失败（即使编译步骤本身已成功通过）。编译器不可能在工作期间检查所有边缘情况。此外，还有许多在编译时无法检测到的错误，如逻辑错误和运行时错误等。例如，如果程序有一个数组越界错误，这种错误可能只会在程序运行时才会出现，并且编译器无法在编译时检测出来。同样，如果程序逻辑有误，可能会导致程序不按预期工作，因此在程序运行时也可能出现错误。

##### 编译各阶段的工作

如前所述，编译是将高级语言翻译成较低级语言的过程 - 具体而言，

通过以特定于给定平台的二进制对象文件格式生成机器代码（特定处理器可以直接执行的指令）。

* 在Linux上，最流行的格式是可执行和可链接格式（ELF）。
* Windows 使用 PE/COFF 格式规范。
* 在macOS上，我们会找到Mach对象（Mach-O格式）。

对象文件是单个源文件的直接编译。它们中的每一个都必须单独编译，

然后通过链接器加入到一个可执行文件或库中。

因此，当您更改代码时，可以通过仅重新编译受修改过的文件来节省时间。

编译器必须执行以下阶段才能创建对象文件：

* 预处理；（处理宏（#if、#elif 和#endif），#include包含等）
* 语言分析（检查C++词法，和语义）
* 汇编 （根据平台可用的指令集将这些标记转换为特定于 CPU 的指令）
* 优化 （负责最大限度地减少寄存器的使用并删除未使用的代码，比如优化inline函数）
* 代码发射（将优化后的机器代码写入目标文件）

预处理：（尽管大多数编译器会自动调用）被认为是实际编译的第一步。它的作用是以非常基本的方式操作源代码;它执行#include指令，将标识符替换为定义的值（#define指令和 -D 标志），调用简单宏，并根据 #if、#elif 和 #endif 指令有条件地包含或排除部分代码。预处理器不知道实际的C++代码，一般来说，只是一个稍微高级的查找和替换工具。然而，它的工作对于构建高级程序至关重要;将代码分解为多个部分并在多个翻译单元之间共享声明的能力是代码可重用性的基础。

语言分析：这是更有趣的事情发生的地方。编译器将逐个字符扫描文件（包含预处理器包含的所有标头）并执行词法分析，将它们分组为有意义的标记 - 关键字、运算符、变量名称等。然后，将令牌分组到令牌链中，并验证它们的顺序和存在是否符合C++规则——此过程称为语法分析或解析（通常，就打印错误而言，这是最响亮的部分）。最后，执行语义分析 - 编译器尝试检测文件中的语句是否真正有意义。例如，它们必须满足类型正确性检查（不能为字符串变量赋值整数）。

汇编：只不过是将这些令牌转换为基于平台可用指令集的特定于 CPU 的指令。一些编译器实际上创建了一个汇编器输出文件，该文件稍后传递给专用汇编程序以生成 CPU 可以执行的机器代码。其他人直接从内存中生成相同的机器代码。通常，此类编译器包含一个选项，用于生成人类可读汇编代码的文本输出（尽管，仅仅因为您可以阅读它，并不意味着它值得）。

优化：发生在整个编译过程中，一点一点，在每个阶段。在生成第一个汇编版本之后有一个明确的阶段，负责最大限度地减少寄存器的使用并删除未使用的代码。一个有趣且重要的优化是内联扩展或内联。编译器将“剪切”一个函数的主体并“粘贴”它而不是它的调用（标准没有定义在什么情况下会发生这种情况——它取决于编译器的实现）。这个过程加快了执行速度并减少了内存使用，但对调试有很大的缺点（执行的代码不再在原来的行）。

代码发射：包括根据目标平台指定的格式将优化后的机器代码写入目标文件。这个目标文件还没有准备好执行——它必须被传递给下一个工具，链接器，它将适当地重新定位我们的目标文件的部分并解析对外部符号的引用。这是从 ASCII 源代码到处理器可消化的二进制目标文件的转换。

这些阶段中的每一个都很重要，可以配置为满足我们的特定需求。让我们看看如何使用CMake管理此过程

#### 与编译阶段相关的cmake命令：

* target\_compile\_features()：需要具有特定功能的编译器来编译此目标。
* target\_sources()：将源文件添加到已定义的目标。
* target\_include\_directories()：设置预处理器包含头文文件路径。
* target\_compile\_definitions()：设置预处理器定义。
* target\_compile\_options()：编译器特定选项。
* target\_precompile\_headers()：优化外部头文件的编译。

前面的所有命令都接受类似的参数：

图形用户界面

低可信度描述已自动生成

这意味着它们支持属性传播，如前一章所述，并且可用于可执行文件和库。

另外，这里提醒一下——所有这些命令都支持生成器表达式。

#### 初始配置

##### 设置编译器的特性（target\_compile\_features）

如第 3 章 “设置您的第一个 CMake 项目”中的检查支持的编译器功能部分所述，为出现问题做好准备，旨在为软件用户提供明确的信息 - 可用的编译器 X 不提供必需的功能 Y。这比用户破译他们可能拥有的不兼容工具链产生的任何错误要好得多。我们不希望用户认为您的代码有问题，而不是他们过时的环境。

图形用户界面, 应用程序

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

CMake 了解这些compiler\_ids C++标准和支持的编译器功能：

* AppleClang: Apple Clang for Xcode versions 4.4+
* Clang: Clang Compiler versions 2.9+
* GNU: GNU Compiler versions 4.4+
* MSVC: Microsoft Visual Studio versions 2010+
* SunPro: Oracle Solaris Studio versions 12.4+
* Intel: Intel Compiler versions 12.1+



#### 管理目标的源文件：【重要】（target\_sources()）

我们已经知道如何告诉 CMake 哪些源文件构成了单个目标 - 可执行文件或库。

每当我们使用 add\_executable（） 或 add\_library（） 时，我们都会提供源文件列表。

随着解决方案的增长，每个目标的文件列表也会增加。我们最终可能会得到一些非常冗长的add\_......（） 命令。

我们该如何处理？一种诱惑可能是在 GLOB 模式下使用 file（） 命令

——它可以从子目录中收集所有文件并将它们存储在变量中。

我们可以将其作为参数传递给目标声明，而无需再次打扰列表文件：

图形用户界面, 文本, 应用程序, 聊天或短信

描述已自动生成

但是，不建议使用前面提到的方法。让我们弄清楚为什么。 CMake 根据列表文件中的更改生成构建系统，因此如果不进行任何更改，您的构建可能会在没有任何警告的情况下中断（正如我们从长时间调试中知道的那样，这是最严重的破坏类型）。除此之外，在目标声明中没有列出所有源代码将破坏 IDE 中的代码检查，例如 CLion（CLion 仅解析某些命令来理解您的项目）。

思考：

如果不建议在目标声明中使用变量，我们如何有条件地添加源文件，

例如，在处理特定于平台的实现文件（如 gui\_linux.cpp 和 gui\_windows.cpp）时？

我们可以使用 target\_sources（） 命令将文件附加到之前创建的目标：

图形用户界面, 文本, 应用程序

描述已自动生成

这样，每个平台都会获得自己的一组兼容文件。

这很好，但是一长串的来源呢？好吧，我们只需要接受有些事情还不完美，并继续手动添加它们。

现在我们已经确定了有关编译的关键事实，让我们仔细看看第一步 - 预处理。

与计算机科学中的所有事物一样，魔鬼在细节中。

预处理器在构建过程中起着巨大的作用。考虑到其功能是多么简单和有限，也许这有点令人惊讶。在以下部分中，我们将介绍如何提供包含文件的路径和使用预处理器定义。我们还将解释如何使用 CMake 配置包含的标头。

#### 配置预处理器

##### 提供包含文件的路径【重要】（target\_include\_directories）

预处理器的最基本功能是能够在 #include 指令中包含 .h/.hpp 头文件。它有两种形式：

* #include <path-spec>：尖括号形式
* #include 'path-spec'：引用形式

正如我们所知，预处理器将用路径规范中指定的文件内容替换这些指令。

查找这些文件可能是个问题。我们搜索哪些目录以及以什么顺序搜索？

不幸的是，C++ 标准并没有明确规定；我们需要查看我们使用的编译器的手册。

通常，尖括号形式将检查标准包含目录，包括标准 C++ 库和标准 C 库头文件在系统中存储的目录。

引用形式将开始在当前文件的目录中搜索包含的文件，然后检查尖括号形式的目录。

CMake 提供了一个命令来操作搜索包含文件的路径：

图形用户界面, 文本, 应用程序, Word

描述已自动生成

我们可以添加希望编译器检查的自定义路径。 CMake 会将它们添加到生成的构建系统中的编译器调用中。将为它们提供适合特定编译器的标志（通常是 -I）。

使用 BEFORE 或 AFTER 确定路径是否应该添加到目标 INCLUDE\_DIRECTORIES 属性中。仍然由编译器决定是否在默认目录之前或之后检查此处提供的目录（通常是之前）。

SYSTEM 关键字通知编译器所提供的目录是标准系统目录（与尖括号形式一起使用）。对于许多编译器，此值将作为 -isystem 标志提供。

##### 预处理器定义（target\_compile\_definitions）

还记得我在描述编译阶段时如何提到预处理器的#define 和#if、#elif 和#endif 指令吗？让我们考虑以下示例：

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

这看起来很简单，但是如果我们想根据外部因素（例如操作系统、体系结构或其他因素）来调节这些部分，会发生什么情况？好消息！您可以将值从 CMake 传递到 C++ 编译器，而且一点也不复杂。

图形用户界面, 文本

描述已自动生成

前面的代码的行为与两个#define 语句完全一样，

但我们可以自由使用 CMake 的变量和生成器表达式，我们可以将命令放在条件块中。

##### 单元测试私有类领域中的常见陷阱

图片包含 图表

描述已自动生成

虽然这个用例非常方便（它允许测试直接访问私有成员），但它不是很干净的代码。单元测试应该只测试公共接口中的方法是否按预期工作，并将底层实现视为黑盒机制。我建议您仅将其用作最后的手段。

##### 使用 git commit 跟踪编译后的版本

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

##### 配置头文件【重要】

如果我们有多个变量，通过 target\_compile\_definitions() 传递定义可能会有一些开销。我们不能只提供一个带有引用各种变量的占位符的头文件，然后让 CMake 填充它们吗？我们当然可以！使用 configure\_file(<input> <output>) 命令，我们可以从这样的模板生成新文件：

configure\_file命令：

图形用户界面, 文本

描述已自动生成

图形用户界面, 应用程序

中度可信度描述已自动生成

#### 配置优化器-性能优化

##### 优化器的作用：

优化器将分析前面阶段的输出并使用大量技巧，程序员会认为这些技巧很脏，因为他们不遵守干净代码原则。没关系——优化器的关键作用是提高代码的性能（即使用较少的 CPU 周期、较少的寄存器和较少的内存）。当优化器遍历源代码时，它会对其进行大量转换，以至于它几乎变得无法识别。它变成了专门为目标 CPU 准备的版本。

优化器不仅会决定哪些函数可以被删除或压缩；它还会四处移动代码甚至显着复制它！如果它可以完全确定某些代码行是无意义的，它会将它们从一个重要函数的中间删除（您甚至不会注意到）。它将重用内存，因此多个变量可以在不同时间段占用同一个槽。如果这意味着它可以在这里和那里削减几个周期，它将把你的控制结构转换成完全不同的结构

每个编译器都有自己的技巧，与它所遵循的平台和理念保持一致。我们将看看GNU GCC和LLVM Clang中最常见的，以便我们能够理解什么是有用的和可能的。

事情是这样的 - 许多编译器默认情况下不会启用任何优化（包括GCC）。这在某些情况下是可以的，但在其他情况下则不然。当你可以走得快时，为什么要走慢？要更改内容，我们可以使用 target\_compile\_options（） 命令并准确指定我们想要从编译器中获取的内容。

图形用户界面, 应用程序

描述已自动生成

我们提供了要添加的目标命令行选项，并指定了传播关键字。执行此命令时，CMake 会将给定的选项附加到目标的相应COMPILE\_OPTIONS变量。可选的 BEFORE 关键字可用于指定我们希望在它们前面加上它们。在某些情况下，顺序很重要，所以我们可以选择是件好事。

##### 优化等级：（target\_compile\_options）

大多数编译器提供四个基本级别的优化，从 0 到 3。

* -O0 表示不优化，通常是编译器的默认级别（相当于debug模式）
* -O1 级别，它可以启用合理数量的优化机制，而不会过多地减慢编译速度
* -O2 被认为是完全优化，它生成高度优化的代码但以最慢的编译时间为代价。
* -O3，它是完全优化，就像 -O2 一样，但对子程序内联和循环向量化采用更积极的方法（相当于Relese模式）

还有一些优化的变体将针对生成文件的大小（不一定是速度）进行优化 – -Os。有一个超级激进的优化，-Ofast，这是一个不严格遵守C++标准的-O3优化。 最明显的区别是 -ffast-math 和 -ffinite-math 标志的使用，这意味着如果你的程序是关于精确计算的（就像大多数一样），你可能想避免它。

设置debug或是release模式：

文本

描述已自动生成

如您所见，调试配置不启用任何优化，发布配置直接针对 O3。如果您愿意，可以直接使用 set（） 命令更改它们，或者只添加一个目标编译选项，这将覆盖此默认行为。其他两个标志（-g、-DNDEBUG）与调试相关 – 我们将在为调试器提供信息部分中讨论它们。

诸如CMAKE\_<LANG>\_FLAGS\_<CONFIG> 等变量是全局的——它们适用于所有目标。建议通过属性和命令（如 target\_compile\_options（））来配置目标，而不是依赖全局变量。这样，您就可以以更高的粒度控制目标。

通过选择具有 -O<级别>的优化级别，我们间接设置了一长串标志，每个标志控制特定的优化行为。然后，我们可以通过附加更多标志来微调优化，如下所示：

* 使用 -f 选项启用它们：-finline-functions。
* 使用 -fno 选项禁用它们：-fno-inline-functions

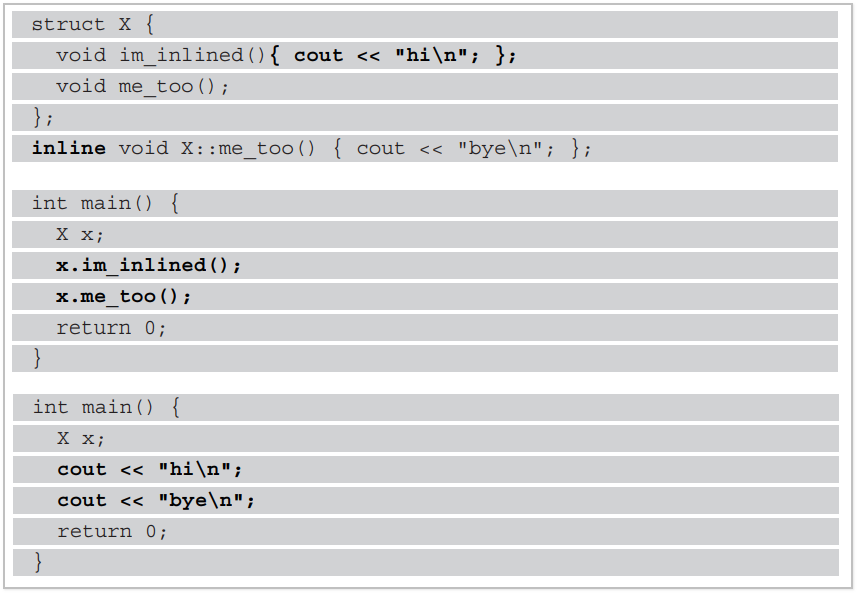
图形用户界面, 应用程序

描述已自动生成



##### 内联函数

由编译器决定是否内联函数。如果启用了内联并且函数在单个地方使用（或者是在几个地方使用的相对较小的函数），那么内联很可能会发生。



内联函数的原理：

编译器这样做是为了节省时间；它不必经历新调用框架的创建和拆卸，

它不必查找要执​​行（并返回）的下一条指令的地址，并且它可以更好地缓存指令附近。

特别注意：

这不是一个精确的表示，因为内联发生在汇编或机器代码级别（而不是源代码），但它传达了一个一般情况。

弊端-1，文件体积变大；

当然，内联有一些重要的副作用；如果该函数被多次使用，

则必须将其复制到所有地方（意味着更大的文件大小和更多的内存被使用）。

如今，这可能不像过去那么重要，但它仍然很重要，

因为我们不断开发必须在低端设备上运行而没有太多 RAM 的软件。

弊端-2，不容易调试

除此之外，当我们调试我们编写的代码时，它会严重影响我们。内联代码不再位于最初编写的行号处，因此跟踪起来不那么容易（有时甚至不可能）。这就是为什么放置在内联函数中的调试器断点永远不会被命中（尽管代码仍然以某种方式执行）的确切原因。为避免此问题，我们只需禁用调试版本的内联（代价是不测试与发布版本完全相同的版本）。

怎么禁用内联函数：

您可以使用 -fno-inline-.... 明确禁用内联。无论如何，有关详细信息，请参阅特定版本编译器的文档。

##### 循环展开

文本, 表格

中度可信度描述已自动生成

循环展开的原理：

循环展开是一种优化技术，也称为循环展开。一般的做法是将循环转化为一组达到相同效果的语句。通过这样做，我们将以程序大小换取执行速度，因为我们将减少或消除控制循环的指令——指针算术或循环结束测试。

循环展开的弊端：

仅当编译器知道或可以有效估计迭代量时，循环展开才有效。其次，循环展开可能会对现代 CPU 产生不良影响，因为增加的代码大小可能会阻止有效的缓存。

##### 循环矢量化

表格

描述已自动生成

循环矢量化的原理：

通常，前面的代码会循环 128 次，但有了功能强大的 CPU，我们可以通过同时计算数组的两个或多个元素来更快地执行代码。这是有效的，因为连续元素之间没有依赖关系，数组之间也没有数据重叠。智能编译器可以将前面的循环转换为类似于以下内容的内容（在程序集级别发生）：

GCC 将在 -O3 处启用循环的这种自动矢量化。Clang默认启用它。

#### 管理编译过程

作为程序员和构建工程师，我们还需要考虑编译的其他方面 - 完成所需的时间，以及在构建解决方案的过程中发现和修复所犯错误的难易程度。

##### 减少编译时间

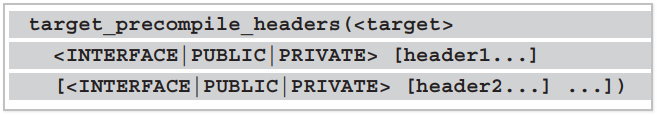
预编译头文件（target\_precompile\_headers）

在每天（或每小时）需要数十次重新编译的繁忙项目中，尽可能快地进行编译至关重要。这不仅会影响代码-编译-测试循环的紧密程度，还会影响您的注意力和工作流程。 幸运的是，由于单独的翻译单元，C++ 已经非常擅长管理编译时间。CMake 将负责仅重新编译受最近更改影响的源文件。但是，如果我们需要进一步改进，可以使用几种技术 - 标头预编译和统一构建。

预编译头文件：

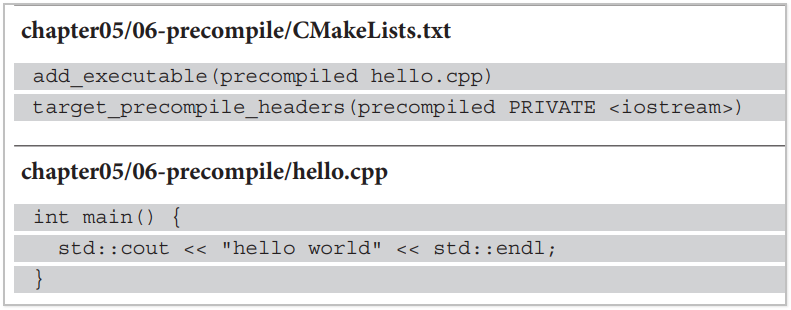
头文件 （.h） 在实际编译开始之前由预处理器包含在翻译单元中。这意味着每次.cpp实现文件更改时都必须重新编译它们。最重要的是，如果多个翻译文件使用相同的共享标头，则必须在每次包含时对其进行编译。这是浪费，但很长一段时间以来都是这样。

幸运的是，从版本 3.16 开始，CMake 提供了一个命令来启用标头预编译。 这允许编译器将标头与实现文件分开处理，并加快编译速度。这是所提供命令的语法



添加的标头列表存储在 PRECOMPILE\_HEADERS 目标属性中。正如您将从第 4 章 “使用目标”中了解到的那样，我们可以使用传播的属性通过使用 PUBLIC 或 INTERFACE 关键字与任何依赖的目标共享标头;但是，对于使用 install（） 命令导出的目标，不应执行此操作。其他项目不应该被迫使用我们的预编译标头（因为它是非常规的）。

CMake 会将所有标头的名称放在一个 cmake\_pch.h|xx 文件中，然后将其预编译为扩展名为 .pch、.gch 或 .pchi 的编译器特定二进制文件



请注意，在我们的 main.cpp 文件中，我们不需要包含 cmake\_pch.h 或任何其他标头 - CMake 将使用特定于编译器的命令行选项强制包含它。

使用其他的目标的编译：

图形用户界面, 文本, 应用程序

描述已自动生成

统一构建

CMake 3.16 还引入了另一个编译时间优化功能 – Unity builds，也称为统一构建或巨型构建。Unity 构建将多个实现源文件与 #include 指令相结合（毕竟，编译器不知道它是否包含标头或实现）。这有一些有趣的含义——有些是真的有用，有些是潜在的有害的。

图形用户界面, 文本, 应用程序

描述已自动生成

当这两个源都包含#include的“header.h”行时，由于包含守卫（假设我们没有忘记添加这些），它只会被解析一次。这不像预编译标头那样优雅，但它是一种选择。

文本

描述已自动生成

##### 提供错误信息 （target\_compile\_options）

作为程序员，我们花了很多时间寻找错误。这是一个可悲的事实。发现错误并解决它们通常会深入我们的皮肤，特别是如果它需要很长时间。当我们盲目飞行时，没有仪器来帮助我们在风暴中导航时，情况就更加困难了。这就是为什么我们应该非常小心地设置我们的环境，使这个过程尽可能容易和可以忍受。我们通过用 target\_compile\_options（） 配置编译器来做到这一点。那么哪些编译选项可以帮助我们呢？

配置错误和警告

调试生成

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

为调试器提供信息

图形用户界面, 文本

描述已自动生成

机器代码是以二进制格式编码的指令和数据的一个神秘列表。它不传达任何意义或目标。这是因为CPU不关心程序的目标是什么，也不关心所有指令的意义是什么。唯一的要求是代码的正确性。编译器会将上述所有内容转换为 CPU 指令的数字标识符、一些用于初始化内存的数据以及数千个内存地址。换句话说，最终的二进制文件不需要包含实际的源代码、变量名、函数签名或程序员关心的任何其他细节。这是编译器的默认输出 - 原始和干燥。 这样做主要是为了节省空间并在没有太多开销的情况下执行。

相反，我们可以要求编译器将源代码存储在生成的二进制文件中，以及包含编译代码和原始代码之间引用的映射。然后，我们可以将调试器挂接到正在运行的程序，并查看在任何给定时刻正在执行哪个源代码行。这在我们处理代码时是必不可少的，

#### 总结：

毫无疑问，编译是一个复杂的过程。由于其所有边缘情况和特定要求，如果没有好的工具，可能很难管理。值得庆幸的是，CMake 在这里为我们提供支持方面做得很好。 到目前为止，我们学到了什么？我们首先讨论了什么是编译，以及它在操作系统中构建和运行应用程序的更大故事中的位置。 然后，我们检查了编译的阶段以及管理它们的内部工具。这对于解决我们可能遇到的更高级情况下的所有问题非常有用。

然后，我们研究了如何要求 CMake 验证主机上可用的编译器是否满足构建代码的所有必要要求。正如我们已经建立的那样，对于我们解决方案的用户来说，看到一条要求他们升级的友好消息是一种更好的体验，而不是由旧编译器打印的一些晦涩错误，该错误被语言的新功能所混淆。

我们很快讨论了如何将源添加到已经定义的目标，并转到预处理器的配置。这是一个相当大的主题，因为这个阶段将所有代码位放在一起，并决定忽略其中的哪些位。我们讨论了提供文件的路径以及将自定义定义添加为单个参数和批量（以及一些用例）。

然后，我们讨论了优化器;我们探索了所有一般的优化级别以及它们所暗示的标志类型，但我们也详细介绍了其中的一些 - Finline，Floop-Unroll和Ftree-vectorize

最后，是时候再次研究更大的图景，研究如何管理编译的可行性了。我们在这里解决了两个主要方面——减少编译时间（并进一步加强程序员的注意力）和发现错误。后者对于发现损坏的内容以及如何损坏非常重要。 正确设置工具并理解事情发生的原因对于确保代码的质量（以及我们的心理健康）大有帮助。