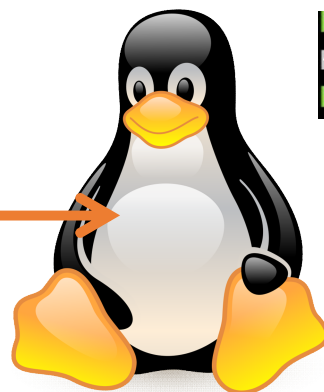


什么是编译器

厂商	C	C++	Fortran
GNU	gcc	g++	gfortran
LLVM	clang	clang++	flang

- 编译器，是一个根据源代码生成机器码的程序。
- `> g++ main.cpp -o a.out`
- 该命令会调用编译器程序g++，让他读取main.cpp中的字符串（称为源码），并根据C++标准生成相应的机器指令码，输出到a.out这个文件中，（称为可执行文件）。
- `> ./a.out`
- 之后执行该命令，操作系统会读取刚刚生成的可执行文件，从而执行其中编译成机器码，调用系统提供的printf函数，并在终端显示出Hello, world。

```
main.cpp
1 #include <stdio>
2
3 int main() {
4     printf("Hello, world!\n");
5     return 0;
6 }
```



```
bate@archer ~/Codes/course/01/01 (master) $ ./a.out
Hello, world!
bate@archer ~/Codes/course/01/01 (master) $
```

多文件编译与链接

The diagram shows two source files side-by-side. The left file, `hello.cpp`, contains a function definition for `hello()` on line 3. The right file, `main.cpp`, contains a function declaration for `hello()` on line 3 and a call to `hello()` on line 6. A yellow arrow points from the `hello()` call in `main.cpp` to the `hello()` definition in `hello.cpp`. A green arrow points from the `hello()` definition in `hello.cpp` to the `hello()` call in `main.cpp`.

```
hello.cpp
1 #include <stdio>
2
3 void hello() {
4     printf("Hello, world\n");
5 }

main.cpp
1 #include <stdio>
2
3 void hello();
4
5 int main() {
6     hello();
7     return 0;
8 }
```

- 单文件编译虽然方便，但也有如下缺点：

1. 所有的代码都堆在一起，不利于模块化和理解。
 2. 工程变大时，编译时间变得很长，改动一个地方就得全部重新编译。
- 因此，我们提出多文件编译的概念，文件之间通过符号声明相互引用。

- > g++ -c hello.cpp -o hello.o

- > g++ -c main.cpp -o main.o

- 其中使用 `-c` 选项指定生成临时的对象文件 `main.o`，之后再根据一系列对象文件进行链接，得到最终的 `a.out`：

- > g++ hello.o main.o -o a.out

为什么需要构建系统（**Makefile**）



- 文件越来越多时，一个个调用g++编译链接会变得很麻烦。
- 于是，发明了 **make** 这个程序，你只需写出不同文件之间的**依赖关系**，和生成各文件的规则。
- **> make a.out**
- 敲下这个命令，就可以构建出 **a.out** 这个可执行文件了。
- 和直接用一个脚本写出完整的构建过程相比，**make** 指明依赖关系的好处：
 1. 当更新了**hello.cpp**时只会重新编译**hello.o**，而不需要把**main.o**也重新编译一遍。
 2. 能够自动**并行**地发起对**hello.cpp**和**main.cpp**的编译，加快编译速度（**make -j**）。
 3. 用通配符批量生成构建规则，避免针对每个**.cpp**和**.o**重复写 **g++** 命令（**%.o: %.cpp**）。
- 但坏处也很明显：
 1. **make** 在 **Unix** 类系统上是通用的，但在 **Windows** 则不然。
 2. 需要准确地指明每个项目之间的依赖关系，有头文件时特别头疼。
 3. **make** 的语法非常简单，不像 **shell** 或 **python** 可以做很多判断等。
 4. 不同的编译器有不同的 **flag** 规则，为 **g++** 准备的参数可能对 **MSVC** 不适用。

```
Makefile+
1 a.out: hello.o main.o
2     g++ hello.o main.o -o a.out
3
4 hello.o: hello.cpp
5     g++ -c hello.cpp -o hello.o
6
7 main.o: main.cpp
8     g++ -c main.cpp -o main.o
```

构建系统的构建系统（**CMake**）

```
CMakeLists.txt  
1 add_executable(a.out main.cpp hello.cpp)
```

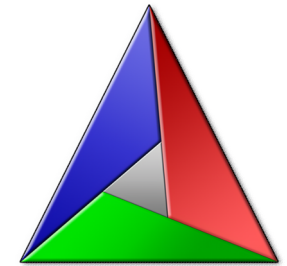
↑
输出的可执行文件

↙ ↘
输入的多个源文件

- 为了解决 **make** 的以上问题，跨平台的 **CMake** 应运而生！
- ~~**make** 在 Unix 类系统上是通用的，但在 Windows 则不然。~~
- 只需要写一份 **CMakeLists.txt**，他就能在调用时生成当前系统所支持的构建系统。
- ~~需要准确地指明每个项目之间的依赖关系，有头文件时特别头疼。~~
- **CMake** 可以自动检测源文件和头文件之间的依赖关系，导出到 **Makefile** 里。
- ~~**make** 的语法非常简单，不像 **shell** 或 **python** 可以做很多判断等。~~
- **CMake** 具有相对高级的语法，内置的函数能够处理 **configure**，**install** 等常见需求。
- ~~不同的编译器有不同的 **flag** 规则，为 **g++** 准备的参数可能对 **MSVC** 不适用。~~
- **CMake** 可以自动检测当前的编译器，需要添加哪些 **flag**。比如 **OpenMP**，只需要在 **CMakeLists.txt** 中指明 `target_link_libraries(a.out OpenMP::OpenMP_CXX)` 即可。




CMake 的命令行调用



CMake

- 读取当前目录的 CMakeLists.txt，并在 build 文件夹下生成 build/Makefile:
- > cmake -B build
- 让 make 读取 build/Makefile，并开始构建 a.out:
- > make -C build
- 以下命令和上一个等价，但更跨平台:
- > cmake --build build
- 执行生成的 a.out:
- > build/a.out

```
1 cmake_minimum_required(VERSION 3.12)
2 project(hellocmake LANGUAGES CXX)
3
4 add_executable(a.out main.cpp hello.cpp)
```



```
-- The CXX compiler identification is GNU 11.1.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/bate/Codes/course/01/05/build
[ 33%] Building CXX object CMakeFiles/a.out.dir/main.cpp.o
[ 66%] Building CXX object CMakeFiles/a.out.dir/hello.cpp.o
[100%] Linking CXX executable a.out
[100%] Built target a.out
Hello, world
```

为什么需要库（**library**）



- 有时候我们会有多个可执行文件，他们之间用到的某些功能是相同的，我们想把这些共用的功能做成一个库，方便大家一起共享。
- 库中的函数可以被可执行文件调用，也可以被其他库文件调用。
- 库文件又分为静态库文件和动态库文件。
- 其中静态库相当于直接把代码插入到生成的可执行文件中，会导致体积变大，但是只需要一个文件即可运行。
- 而动态库则只在生成的可执行文件中生成“插桩”函数，当可执行文件被加载时会读取指定目录中的.dll文件，加载到内存中空闲的位置，并且替换相应的“插桩”指向的地址为加载后的地址，这个过程称为重定向。这样以后函数被调用就会跳转到动态加载的地址去。
- Windows: 可执行文件同目录，其次是环境变量%PATH%
- Linux: ELF格式可执行文件的RPATH，其次是/usr/lib等

CMake 中的静态库与动态库

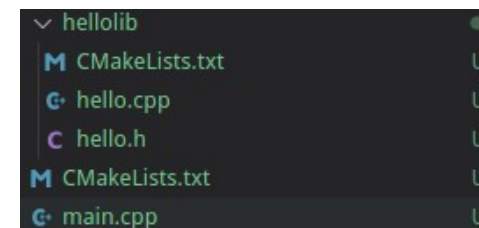
- CMake 除了 `add_executable` 可以生成可执行文件外，还可以通过 `add_library` 生成库文件。
- `add_library` 的语法与 `add_executable` 大致相同，除了他需要指定是动态库还是静态库：
- `add_library(test STATIC source1.cpp source2.cpp)` # 生成静态库 `libtest.a`
- `add_library(test SHARED source1.cpp source2.cpp)` # 生成动态库 `libtest.so`
- 动态库有很多坑，特别是 Windows 环境下，初学者自己创建库时，建议使用静态库。
- 但是他人提供的库，大多是作为动态库的，我们之后会讨论如何使用他人的库。
- 创建库以后，要在某个可执行文件中使用该库，只需要：
- `target_link_libraries(myexec PUBLIC test)` # 为 `myexec` 链接刚刚制作的库 `libtest.a`
- 其中 `PUBLIC` 的含义稍后会说明（CMake 中有很多这样的大写修饰符）

CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.12)
2 project(hellocmake LANGUAGES CXX)
3
4 add_library(hellolib STATIC hello.cpp)
5 add_executable(a.out main.cpp)
6 target_link_libraries(a.out PUBLIC hellolib)
```

```
-- The CXX compiler identification is GNU 11.1.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/bate/Codes/course/01/06/build
[ 25%] Building CXX object CMakeFiles/hellolib.dir/hello.cpp.o
[ 50%] Linking CXX static library libhellolib.a
[ 50%] Built target hellolib
[ 75%] Building CXX object CMakeFiles/a.out.dir/main.cpp.o
[100%] Linking CXX executable a.out
[100%] Built target a.out
Hello, world
```

CMake 中的子模块



- 复杂的工程中，我们需要划分子模块，通常一个库一个目录，比如：
- 这里我们把 **hellolib** 库的东西移到 **hellolib** 文件夹下了，里面的 **CMakeLists.txt** 定义了 **hellolib** 的生成规则。
- 要在根目录使用他，可以用 **CMake** 的 **add_subdirectory** 添加子目录，子目录也包含一个 **CMakeLists.txt**，其中定义的库在 **add_subdirectory** 之后就可以在外面使用。
- 子目录的 **CMakeLists.txt** 里路径名（比如 **hello.cpp**）都是相对路径，这也是很方便的一点。

CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.12)
2 project(hellocmake LANGUAGES CXX)
3
4 add_subdirectory(hellolib)
5
6 add_executable(a.out main.cpp)
7 target_link_libraries(a.out PUBLIC hellolib)
```

h/CMakeLists.txt

```
1 add_library(hellolib STATIC hello.cpp)
```


子模块的头文件如何处理

- 因为 `hello.h` 被移到了 `hellolib` 子文件夹里，因此 `main.cpp` 里也要改成：
- 如果要避免修改代码，我们可以通过 `target_include_directories` 指定
- `a.out` 的头文件搜索目录：(其中第一个 `hellolib` 是库名，第二个是目录)

```
6 add_executable(a.out main.cpp)
7 target_link_libraries(a.out PUBLIC hellolib)
8 target_include_directories(a.out PUBLIC hellolib)
```

- 这样甚至可以用 `<hello.h>` 来引用这个头文件了，因为通过 `target_include_directories` 指定的路径会被视为与系统路径等价：

```
main.cpp
1 #include <stdio>
2 #include <hello.h>
3
4 int main() {
5     hello();
6     return 0;
7 }
```

```
main.cpp
1 #include <stdio>
2
3 #include "hellolib/hello.h"
4
5 int main() {
6     hello();
7     return 0;
8 }
```

子模块的头文件如何处理（续）

- 但是这样如果另一个 `b.out` 也需要用 `hellolib` 这个库，难道也得再指定一遍搜索路径吗？
- 不需要，其实我们只需要定义 `hellolib` 的头文件搜索路径，引用他的可执行文件 `CMake` 会自动添加这个路径：

```
h/CMakeLists.txt
1 add_library(hellolib STATIC hello.cpp)
2 target_include_directories(hellolib PUBLIC .)
```

- 这里用了 `.` 表示当前路径，因为子目录里的路径是相对路径，类似还有 `..` 表示上一层目录。
- 此外，如果不希望让引用 `hellolib` 的可执行文件自动添加这个路径，把 **PUBLIC** 改成 **PRIVATE** 即可。这就是他们的用途：决定一个属性要不要在被 `link` 的时候传播。

目标的一些其他选项

- 除了头文件搜索目录以外，还有这些选项，PUBLIC 和 PRIVATE 对他们同理：
- `target_include_directories(myapp PUBLIC /usr/include/eigen3)` # 添加头文件搜索目录
- `target_link_libraries(myapp PUBLIC hellolib)` # 添加要链接的库
- `target_add_definitions(myapp PUBLIC MY_MACRO=1)` # 添加一个宏定义
- `target_add_definitions(myapp PUBLIC -DMY_MACRO=1)` # 与 `MY_MACRO=1` 等价
- `target_compile_options(myapp PUBLIC -fopenmp)` # 添加编译器命令行选项
- `target_sources(myapp PUBLIC hello.cpp other.cpp)` # 添加要编译的源文件
- 以及可以通过下列指令（不推荐使用），把选项加到所有接下来的目标去：
- `include_directories(/opt/cuda/include)` # 添加头文件搜索目录
- `link_directories(/opt/cuda)` # 添加库文件的搜索路径
- `add_definitions(MY_MACRO=1)` # 添加一个宏定义
- `add_compile_options(-fopenmp)` # 添加编译器命令行选项

第三方库 - 作为纯头文件引入

- 有时候我们不满足于 C++ 标准库的功能，难免会用到一些第三方库。
- 最友好的一类库莫过于纯头文件库了，这里是一些好用的 **header-only** 库：
 1. **nothings/stb** - 大名鼎鼎的 **stb_image** 系列，涵盖图像，声音，字体等，只需单头文件！
 2. **Neargye/magic_enum** - 枚举类型的反射，如枚举转字符串等（实现方式很巧妙）
 3. **g-truc/glm** - 模仿 **GLSL** 语法的数学矢量/矩阵库（附带一些常用函数，随机数生成等）
 4. **Tencent/rapidjson** - 单纯的 **JSON** 库，甚至没依赖 **STL**（可定制性高，工程美学经典）
 5. **ericniebler/range-v3** - **C++20 ranges** 库就是受到他启发（完全是头文件组成）
 6. **fmtlib/fmt** - 格式化库，提供 **std::format** 的替代品（需要 **-DFMT_HEADER_ONLY**）
 7. **gabime/spdlog** - 能适配控制台，安卓等多后端的日志库（和 **fmt** 冲突！）
- 只需要把他们的 **include** 目录或头文件下载下来，然后 **include_directories(spdlog/include)** 即可。
- 缺点：函数直接实现在头文件里，没有提前编译，从而需要重复编译同样内容，编译时间长。

第三方库 - 作为子模块引入

- 第二友好的方式则是作为 CMake 子模块引入，也就是通过 `add_subdirectory`。
- 方法就是把那个项目（以fmt为例）的源码放到你工程的根目录：
- 这些库能够很好地支持作为子模块引入：

1. fmtlib/fmt - 格式化库，提供 `std::format` 的替代品
2. gabime/spdlog - 能适配控制台，安卓等多后端的日志库
3. ericniebler/range-v3 - C++20 ranges 库就是受到他启发
4. g-truc/glm - 模仿 GLSL 语法的数学矢量/矩阵库
5. abseil/abseil-cpp - 旨在补充标准库没有的常用功能
6. bombela/backward-cpp - 实现了 C++ 的堆栈回溯便于调试
7. google/googletest - 谷歌单元测试框架
8. google/benchmark - 谷歌性能评估框架
9. glfw/glfw - OpenGL 窗口和上下文管理
10. libigl/libigl - 各种图形学算法大合集

CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.12)
2 project(hellocmake LANGUAGES CXX)
3
4 add_subdirectory(fmt)
5
6 add_executable(a.out main.cpp)
7 target_link_libraries(a.out PUBLIC fmt)
```

```
bate@archer ~/Codes/course/01/12 (master) $ git clone https://github.com/fmtlib/
fmt.git --depth=1
Cloning into 'fmt'...
remote: Enumerating objects: 238, done.
remote: Counting objects: 100% (238/238), done.
remote: Compressing objects: 100% (229/229), done.
remote: Total 238 (delta 4), reused 139 (delta 1), pack-reused 0
Receiving objects: 100% (238/238), 846.96 KiB | 909.00 KiB/s, done.
Resolving deltas: 100% (4/4), done.
bate@archer ~/Codes/course/01/12 (master) $ ls
CMakeLists.txt  fmt  main.cpp  run.sh
bate@archer ~/Codes/course/01/12 (master) $ ls fmt
ChangeLog.rst  CONTRIBUTING.md  include  README.rst  support
CMakeLists.txt  doc  LICENSE.rst  src  test
bate@archer ~/Codes/course/01/12 (master) $
```

fmt - 使用这个神奇的格式化库

- `fmt::format` 的用法和 Python 的 `str.format` 大致相似:

```
main.cpp+
1 #include <fmt/core.h>
2 #include <iostream>
3
4 int main() {
5     std::string msg = fmt::format("The answer is {}.\\n", 42);
6     std::cout << msg << std::endl;
7     return 0;
8 }
```

```
main.cpp
1 #include <fmt/core.h>
2
3 int main() {
4     fmt::print("The answer is {}.\\n", 42);
5     return 0;
6 }
```

```
[ 20%] Building CXX object fmt/CMakeFiles/fmt.dir/src/format.cc.o
[ 40%] Building CXX object fmt/CMakeFiles/fmt.dir/src/os.cc.o
[ 60%] Linking CXX static library libfmt.a
[ 60%] Built target fmt
[ 80%] Building CXX object CMakeFiles/a.out.dir/main.cpp.o
[100%] Linking CXX executable a.out
[100%] Built target a.out
The answer is 42.
```


CMake - 引用系统中预安装的第三方库

- 可以通过 `find_package` 命令寻找系统中的包/库：
- `find_package(fmt REQUIRED)`
- `target_link_libraries(myexec PUBLIC fmt::fmt)`
- 为什么是 `fmt::fmt` 而不是简单的 `fmt`？
- 现代 CMake 认为一个包 (package) 可以提供多个库，又称组件 (components)，比如 TBB 这个包，就包含了 `tbb`, `tbbmalloc`, `tbbmalloc_proxy` 这三个组件。
- 因此为避免冲突，每个包都享有一个独立的名字空间，以 `::` 的分割（和 C++ 还挺像的）。
- 你可以指定要用哪几个组件：
- `find_package(TBB REQUIRED COMPONENTS tbb tbbmalloc REQUIRED)`
- `target_link_libraries(myexec PUBLIC TBB::tbb TBB::tbbmalloc)`

CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.12)
2 project(hellocmake LANGUAGES CXX)
3
4 add_executable(a.out main.cpp)
5
6 find_package(fmt REQUIRED)
7 target_link_libraries(a.out PUBLIC fmt::fmt)
```

第三方库 - 常用 **package** 列表

1. `fmt::fmt`
2. `spdlog::spdlog`
3. `range-v3::range-v3`
4. `TBB::tbb`
5. `OpenVDB::openvdb`
6. `Boost::iostreams`
7. `Eigen3::Eigen`
8. `OpenMP::OpenMP_CXX`

- 不同的包之间常常有着依赖关系，而包管理器的作者为 `find_package` 编写的脚本（例如 `/usr/lib/cmake/TBB/TBBConfig.cmake`）能够自动查找所有依赖，并利用刚刚提到的 `PUBLIC PRIVATE` 正确处理依赖项，比如如果你引用了 `OpenVDB::openvdb` 那么 `TBB::tbb` 也会被自动引用。
- 其他包的引用格式和文档参考：
<https://cmake.org/cmake/help/latest/module/FindBLAS.html>

安装第三方库 - 包管理器

- Linux 可以用系统自带的包管理器（如 **apt**）安装 C++ 包。
- `> pacman -S fmt`
- Windows 则没有自带的包管理器。因此可以用跨平台的 **vcpkg**:
<https://github.com/microsoft/vcpkg>
- 使用方法：下载 **vcpkg** 的源码，放到你的项目根目录，像这样：
- `> cd vcpkg`
- `> .\bootstrap-vcpkg.bat`
- `> .\vcpkg integrate install`
- `> .\vcpkg install fmt:x64-windows`
- `> cd ..`
- `> cmake -B build -DCMAKE_TOOLCHAIN_FILE="%CD%/vcpkg/scripts/buildsystems/vcpkg.cmake"`

