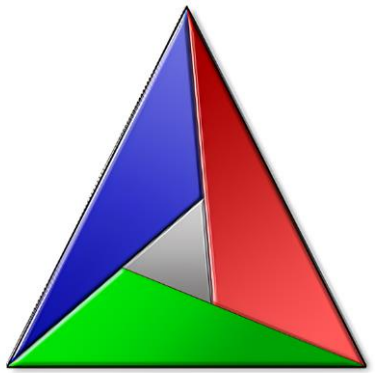
致　谢

- 本次汇报关于CMake部分主要来源于B站UP主小彭老师（双笙子佯谬 https://space.bilibili.com/263032155/）在教学视频（BV16P4y1g7MH）中的分享，PPT主要参考了小彭老师（双笙子佯谬）的教学课件，开源地址为（https://github.com/parallel101/course）。他深入浅出的教学为我初次接触编程领域解答了很多疑惑，欢迎大家前往小彭老师（双笙子佯谬）的视频进行学习。
- 本次学习汇报关于C++11的部分主要参考了《C++ primer plus》一书。

学习进度汇报–TENON项目

——王壹



CMake

Cross-platform Make

- Cmake简介
- C++11内容简介

编译器

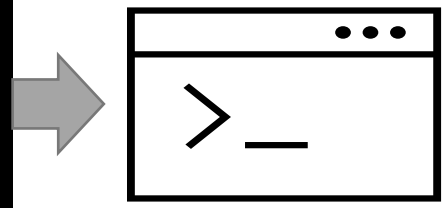
- **编译器**，是一个根据**源代码**生成**机器码**的程序。
- > g++ main.cpp -o a.out
- a.out 是可执行文件，操作系统通过执行机器码从而实现程序功能。

厂商	C	C++
GNU	gcc	g++
LLVM	clang	clang++

main.cpp

```cpp
#include <cstdio>

int main() {
    printf("Hello, world!\n");
    return 0;
}
```



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

单文件编译的局限

1. 代码冗余，不利于构建和阅读。
2. 改动时需要整体重新编译，浪费算力。

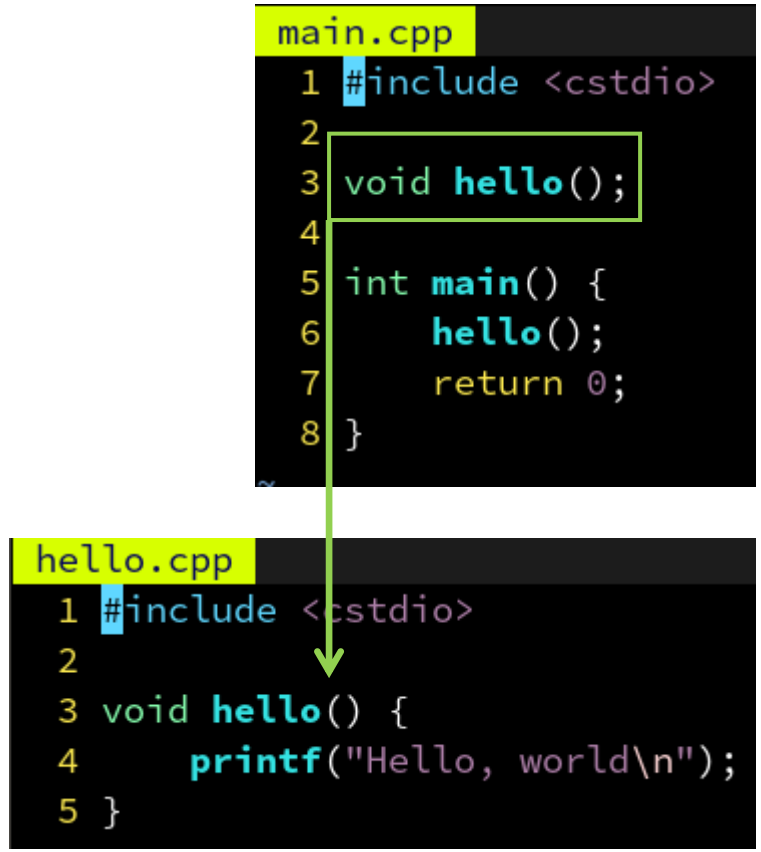
使用多文件编译

> g++ hello.cpp main.cpp –o a.out

多文件编译

- **一般的多文件编译仍旧导致改动时需要整体重新编译。**
- > g++ hello.cpp main.cpp –o a.out

- 可以使用 –c 指定的临时文件main.o来减少编译成本。
- > g++ -c hello.cpp -o hello.o
- > g++ -c main.cpp -o main.o
- 通过临时文件进行链接得到可执行程序。
- > g++ hello.o main.o -o a.out



多文件编译的局限

1. 每次更改都要按文件重新编译比较麻烦。

使用构建系统 (MakeFile)

> make a.out

文件构建系统

指明不同文件之间的依赖关系来构建的优势在于：

- 可以仅编译更改后的文件以提高编译速度。
- 可以并行地进行多文件编译。
- 利用通配符生成构建规则可以减小工作量。

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
```

MakeFile的局限

- Make更适应与Unix类系统，对其他系统适配一般。
- 要准确的指明项目关系对大型项目不友好。
- 语法较为简单，无法实现判断等复杂的功能。
- 不同编译器的flag规则是不同的，适配性较差。

使用构建系统的构建系统 (CMake)

- 一份CMakeLists.txt就可以实现跨平台。
- 自动监测源文件和头文件的依赖关系。
- 具有相对高级的语法。
- 可以自动监测编译器并添加flag。

CMake的命令与调用

传统的 CMake 软件构建/安装方式

• mkdir build	• 需要先创建 build 目录
• cd build	• 切换到 build 目录
• cmake ..	• 在 build 目录运行 cmake <源码目录> 生成 Makefile
• make -j4	• 执行本地的构建系统 make 并行构建4进程
• sudo make install	• 让本地的构建系统执行安装步骤
• cd ..	• 回到源码目录

CMake的命令与调用

现代的CMake 软件构建/安装方式

- cmake -B build　// 在源码目录创建 build 目录并生成 Makefile
- cmake --build build　// 自动调用本地的构建系统在 build 里构建
- cmake -B build **-DCMAKE_BUILD_TYPE=Release**　//在配置阶段可以使用-D指定缓存变量，发布模式在下次构建时得到保留
- cmake –GNinja –B build　//使用-G指定Ninja构建系统以进行性能优化

CMake的命令与调用

不同构建系统的对比

性能上： Ninja > Makefile > MSBuild

Makefile 启动时会把每个文件都检测一遍，浪费很多时间。

Ninja 在有很多文件，但是实际需要构建的只有一小部分的时候，速度提升就很明显。

然而某些专利公司的 CUDA toolkit 在 Windows 上只允许用 MSBuild 构建，不能用 Ninja。

```
bate@archer ~/Codes/course/11/template (master) $ cmake -G"Unix Makefiles" -B build
-- 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/11/template/build
bate@archer ~/Codes/course/11/template (master) $ time cmake --build build
[ 50%] Building CXX object CMakeFiles/main.dir/main.cpp.o
[100%] Linking CXX executable main
[100%] Built target main

real    0m0.236s
user    0m0.145s
sys     0m0.080s
```

```
bate@archer ~/Codes/course/11/template (master) $ rm -rf build && cmake -GNinja -B build
-- 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/11/template/build
bate@archer ~/Codes/course/11/template (master) $ time cmake --build build
[2/2] Linking CXX executable main

real    0m0.070s
user    0m0.052s
sys     0m0.019s
bate@archer ~/Codes/course/11/template (master) $
```

CMake的使用

一个源文件时

main.cpp CMakeLists.txt

```
1 #include <cstdio>
2
3 int main() {
4     printf("Hello, world!\n");
5 }
```

- 直接添加一个可执行文件作为构建目标

main.cpp CMakeLists.txt

```
1 add_executable(main main.cpp)
```

- 或者先创建目标再添加源文件

main.cpp CMakeLists.txt

```
1 add_executable(main)
2 target_sources(main PUBLIC main.cpp)
```

CMake的使用

多个源文件时

main.cpp+ other.cpp | other.h |

```cpp
#include "other.h"

int main() {
    say_hello();
}
```

main.cpp other.cpp other.h |

```cpp
#include "other.h"
#include <cstdio>

void say_hello() {
    printf("Hello from other.cpp!\n");
}
```

main.cpp | other.cpp other.h

```cpp
#pragma once

void say_hello();
```

- 逐个添加

CMakeLists.txt

```
add_executable(main)
target_sources(main PUBLIC main.cpp other.cpp)
```

- 通过变量储存
- 添加头文件可以帮助在 VS 中显示

CMakeLists.txt

```
add_executable(main)
set(sources main.cpp other.cpp)
target_sources(main PUBLIC ${sources})
```

CMake的使用

多个源文件时

- 可以使用**GLOB**自动查找指定扩展名文件实现批量添加

CMakeLists.txt

```
1 add_executable(main)
2 file(GLOB sources *.cpp *.h)
3 target_sources(main PUBLIC ${sources})
```

- 利用**CONFIGURE_DEPENDS**选项自动更新变量

```
▾ mylib/
    other.cpp
    other.h
  CMakeLists.txt
  main.cpp
```

CMakeLists.txt

```
1 add_executable(main)
2 file(GLOB sources CONFIGURE_DEPENDS *.cpp *.h)
3 target_sources(main PUBLIC ${sources})
```

- 使用**aux_source_directory**自动搜索后缀名

CMakeLists.txt

```
1 add_executable(main)
2 aux_source_directory(. sources)
3 aux_source_directory(mylib sources)
4 target_sources(main PUBLIC ${sources})
```

CMake的使用

- 模式的选择：**Release** or **Debug**

CMakeLists.txt+

```
1 if (NOT CMAKE_BUILD_TYPE)
2     set(CMAKE_BUILD_TYPE Release)
3 endif()
```

- project：初始化信息并设置根目录
- LANGUAGES 设置语言，也可以多选

CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.15)
2 project(hellocmake LANGUAGES CXX)
3
4 add_executable(main main.c)
```

- 可以设置c++标准

CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.15)
2
3 set(CMAKE_CXX_STANDARD 17)
```

一个CMakeLists.txt

0/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.15)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

project(zeno LANGUAGES C CXX)

if (PROJECT_BINARY_DIR STREQUAL PROJECT_SOURCE_DIR)
    message(WARNING "The binary directory of CMake cannot be the same as source directory!")
endif()

if (NOT CMAKE_BUILD_TYPE)
    set(CMAKE_BUILD_TYPE Release)
endif()

if (WIN32)
    add_definitions(-DNOMINMAX -D_USE_MATH_DEFINES)
endif()

if (NOT MSVC)
    find_program(CCACHE_PROGRAM ccache)
    if (CCACHE_PROGRAM)
        message(STATUS "Found CCache: ${CCACHE_PROGRAM}")
        set_property(GLOBAL PROPERTY RULE_LAUNCH_COMPILE ${CCACHE_PROGRAM})
        set_property(GLOBAL PROPERTY RULE_LAUNCH_LINK ${CCACHE_PROGRAM})
    endif()
endif()
```

C++11的一些功能

统一的初始化

- C++11扩大了用大括号括起的列表（初始化列表）的适用范围，使其可用于所有内置类型和用户定义的类型（即类对象）。
- 可以使用 = 也可以不使用 。

```
int x = {5};
double y{2.72};
short quar[5] = {4, 5, 2, 76, 1};
int *ar = new int[4]{2, 4, 6, 7};
```

- 也可以使用列表来创建对象并调用构造函数。

C++11的一些功能

统一的初始化

- 初始化列表可以防止缩窄:

```
char c1 = 459585821;   //允许
char c1 = {459585821}; //不允许
```

- 可以使用initializer_list作为构造函数的参数

```
double sum(std::initializer_list<double> il)
{
    double tot = 0;
    for (auto p:il)
        tot += p;
    return tot;
}
```

```
double total = sum({2.5,3.1,4});
```

C++11的一些功能

声明

- 使用auto来简化模板声明

```
for (std::initializer_list<double>::iterator p = il.begin();
     p != il.end(); p++)
```

```
for (auto p = il.begin(); p != il.end(); p++)
```

- 使用decltype来将变量的类型指定为表达式的类型

```
decltype(x*n) q; //double
decltype(&x) pd; //double *
```

```
template<typename T, typename U>
void ef (T t, U u)
{
    decltype(T*U) tu;
}
```

C++11的一些功能

声明

- 返回类型后置

```
template<typename T, typename U>
auto eff (T t, U u) -> decltype(t*u)
{
    decltype(t*u) tu;
    ...
}
```

- 使用 using= 来实现模板别名（可以用于模板部分的具体化）

```
using itType = std::vector<std::string>::iterator;
```

- 使用 nullptr 来表示空指针。

C++11的一些功能

模板和STL方面的修改

- 基于范围的 for 循环 （如需修改可以使用引用）

```
1 double prices[5] = {4.99, 10.99, 6.87, 7.99, 8.49};
2 for (auto x : prices)
3     std::cout << x << std::endl;
```

- 新增了STL容器forward_list、unordered_map等。还新增了模板array，它不能修改大小如push_back()，但是可以使用begin()和end()等方法。
- 新增了方法cbegin()和cend()。这些新方法将元素视为const。
- 对模板valarray添加了两个函数begin()和end()。
- 允许在声明嵌套模板时不使用空格将尖括号分开：

```
1 std::vector<std::list<int>> v1;
```

C++11的一些功能

新的智能指针模板类

- 智能指针模板~~auto_ptr~~、unique_ptr、和shared_ptr都定义了类似指针的对象，可以将new获得（直接或间接）的地址赋给这种对象。
- 当智能指针过期时，其析构函数将使用delete来释放内存。

```
1 auto_ptr<string> ps (new string("I reigned lonely as a cloud.");
```

- auto_ptr存在的问题：指向同一个对象时可能释放两遍

```
1 auto_ptr<string> ps (new string("I reigned lonely as a cloud.");
2 auto_ptr<string> vocation;
3 vocation = ps;
```

C++11的一些功能

新的智能指针模板类

- 解决方案：

1. 建立所有权概念的unique_ptr，对于特定的对象，只能有一个只能指针可拥有它，这样只有拥有对象的智能指针的析构函数会删除该对象。下面的语句中#6非法！

   注：unique_ptr有使用new[]和delete[]的版本

```
1  unique_ptr<string> p3(new string(“auto”));  //#4
2  unique_ptr<string> p4;                      //#5
3  p4 = p3;                                    //#6
```

2. 创建智能更高的指针，跟踪引用特定对象的智能指针数。这称为引用计数。仅当最后一个指针过期时，才调用delete。这是shared_ptr采用的策略。

- 多个指向同一个对象的指针，应选择shared_ptr；如果程序不需要多个指向同一个对象的指针，则可使用unique_ptr。

C++11的一些功能

可变参数模板

- C++11提供了用省略号表示的元运算符，能够声明表示模板参数包的标识符，模板参数包基本上是一个类型列表。

```
1  template<typename... Args>
2  void show_list(Args... args)
3  {
4      //...
5  }
```

```
template <typename... Args>
void show_list1(Args... args)
{
    show_lists1(args...);
}
show_list1(5, 'L', 0.5);
```

- 右侧的程序会导致无限递归，因此存在缺陷。

C++11的一些功能

可变参数模板

- 可以通过递归来正确的展开参数包

```
template<typename T, typename... Args>
void show_list(T value, Args... args)
{
    std::cout << value << ",";
    show_list(args...);
}
```

- 可以为最后一项单独定义模板

```
template<typename T>
void show_list(T value)
{
    std::cout << value << '\n';
}
```

- 这样在args包缩短到只有一项时将调用这个版本，打印换行符而不是逗号。

Lambda函数

对于接受函数指针或函数符的函数，可使用匿名函数定义（Lambda)作为其参数。

- 对于表达式完全由一条返回语句组成或没有返回语句的Lambda函数，其返回值类型自动推断。

```
1 [](int x) {return x % 3 == 0;}
```

- 表达式由多条语句组成时，需要使用返回类型后置语法

```
1 [](double x)->double(int y = x; return x - y;}
```

- 也可以为Lambda表达式指定一个名称

```
1 auto mod3 = [](int x){return x % 3 == 0;}
2 count1 = std::cout_if(n1.begin(), n1,end(), mod3);
```

Lambda函数

lambda表达式允许捕获一定范围内的变量:

- [] 不捕获任何变量
- [&] 引用捕获，捕获外部作用域所有变量，在函数体内当作引用使用
- [=] 值捕获，捕获外部作用域所有变量，在函数内内有个副本使用
- [=, &a] 值捕获外部作用域所有变量，按引用捕获a变量
- [a] 只值捕获a变量，不捕获其它变量
- [this] 捕获当前类中的this指针

```
1  int a = 0;
2  auto f1 = [=](){ return a; }; // 值捕获a
3  cout << f1() << endl;
4
5  auto f2 = [=]() { return a++; }; // 修改按值捕获的外部变量，error
6  auto f3 = [=]() mutable { return a++; };
```

Lambda函数

- 一个完整的Lambda函数

```
auto func = [capture] (params) opt -> ret { func_body; };
```

- Lambda函数的常用场景

```
int count3 = 0;
std::for_each(numbers.begin(), numbers.end(),  [&count3](int x){count3 += x % 3 == 0;});
```

参考资料

https://github.com/parallel101/course

C++ Primer Plus