

C++ 编译链接

最近，有同事向我多次问及 C++ 关于编译链接方面的问题，包括如下：

- 1: 什么样的函数以及变量可以定义在头文件中
- 2: `extern "C"`的作用
- 3: 防止重复包含的宏的作用
- 4: 函数之间是怎么链接起来的

我认为，这些问题不难，书上基本上都有，但要是没有真正思考过，就凭死记硬背，也就是只能“嘴上说说”而已，遇到问题还真棘手，所以我觉得有必要说一下。

C/C++ 的编译链接过程

其实，“编译”这个词大多数时候，我们指的是由一堆 `.h`, `.c`, `.cpp` 文件生成链接库或者可执行文件的过程。但是拿 C/C++ 来说，其实这是很模糊的，由一堆 C/C++ 文件生成应用程序包括 **预处理---编译文件---链接**(写的比较粗糙，不影响本文论述)。

首先，要明白什么是编译单元，一个编译单元可以认为是一个 `.c` 或者 `.cpp` 文件，每一个编译单元首先会经过预处理得到一个临时的编译单元，这里称为 `tmp.cpp`，预处理会把 `.c` 或者 `.cpp` 直接或者间接包含的其它文件(不只局限于 `.h` 文件，只要是 `#include` 即可)的内容替换进来，并展开宏调用等。

下面首先看一个例子：

a.h

[\[html\]](#) [view plain copy](#)

```
1.  #ifndef A_H_
2.  #define A_H_
```

```
3.  
4. static int a = 1;  
5. void fun();  
6.  
7. #endif
```

a.cpp

[cpp] view plain copy

```
1. #include "a.h"  
2.  
3.  
4. static void hello_world()  
5. {  
6. }
```

只有 a.h 和 a.cpp 这两个文件，及其简单。首先通过 g++ 的 -E 参数得到 a.cpp 预处理之后的内容

[plain] view plain copy

```
1. coderchen@coderchen:~/c++$ g++ -E a.cpp > tmp.cpp
```

查看 tmp.cpp

[cpp] view plain copy

```
1. # 1 "a.cpp"  
2. # 1 "<built-in>"  
3. # 1 "<command-line>"  
4. # 1 "a.cpp"  
5. # 1 "a.h" 1  
6.  
7.  
8.  
9. static int a = 1;  
10. void fun();  
11. # 2 "a.cpp" 2  
12.
```

```
13.  
14. static void hello_world()  
15. {  
16. }
```

tmp.cpp 就是只经过预处理得到的文件，这个文件才是编译器能够真正看到的文件。这个过程就是预处理。

其中#define A_H_的作用是防止重复包含 a.h 这个头文件，很多人都知道这一点，但是再仔细问，我见过大多数人都说不清楚。

这种宏是为了防止一个编译单元(cpp 文件)重复包含同一个头文件。它在预处理阶段起作用，预处理器发现 a.cpp 内已经定义过 A_H_ 这个宏的话，在 a.cpp 中再次发现#include "a.h"的时候就不会把 a.h 的内容替换进 a.cpp 了。

编译器看到 tmp.cpp 的时候，会编译成一个 obj 文件，最后由链接器对这一个对 obj 文件进行链接，从而得到可执行程序。

编译错误和连接错误

编译错误指的是一个 cpp 编译单元在编译时发生的错误，这种错误一般都是语法错误，拼写错误，参数不匹配等。

以 main.cpp 为例(只有一个 main 函数)

[cpp] view plain copy


```
1. int main()  
2. {  
3.     hello_world();  
4. }
```

编译(加-c 参数表示只编译不链接)

[cpp] view plain copy


1. coderchen@coderchen:~/c++\$ g++ -c -o main.o main.cpp
2. main.cpp: In function 'int main()':
3. main.cpp:4: error: 'hello_world' was not declared in this scope

这种错误就是编译，原因是 `hello_world` 函数未声明，把 `void hello_world();` 这条语句加到 `main` 函数前面，再次编译

[plain] view plain copy 

1. coderchen@coderchen:~/c++\$ g++ -c -o main.o main.cpp
2. coderchen@coderchen:~/c++\$

编译成功，虽然我们调用了 `hello_world` 函数，却没有定义这个函数。好，接下来，我们把这个 `main.o` 文件链接下，

[cpp] view plain copy 

1. coderchen@coderchen:~/c++\$ g++ -o main main.o
2. main.o: In function 'main':
3. main.cpp:(.text+0x7): undefined reference to 'hello_world()'
4. collect2: ld returned 1 exit status

看到了吧，链接器 `ld` 报出了链接错误，原因是 `hello_world` 这个函数找不到。这个例子很简单，基本上可以区分出编译错误和链接错误。我们再添加一个 `hello_world.cpp`

[html] view plain copy 


1. void hello_world()
2. {
3. }

编译

[cpp] view plain copy 

1. coderchen@coderchen:~/c++\$ g++ -c -o hello_world.o hello_world.cpp

链接

[plain] view plain copy 

```
1. coderchen@coderchen:~/c++之所以$ g++ -o main main.o hello_world.o
```

ok, 我们的 main 程序已经生成了, 我们经历了预处理---编译---链接的过程。

有的人说为什么不需要写一个 hello_world.h 的头文件, 声明 hello_world 函数, 然后再让 main.cpp 包含 hello_world.h 呢? 这样写自然是标准的做法, 不过预处理过后, 和我们现在写的一样的, 预处理会把 hello_world.h 的内容替换到 main.cpp 中。

问题: 在链接的时候, main.o 怎么知道 hello_world 函数定义在 hello_world.o 中呢?

答案: main.o 不知道 hello_world 函数定义在那个 obj 文件中, 每个 obj 文件都有一个导出符号表, 对于这个例子, hello_world.o 的导出符号表中有 hello_world 这个函数, 而 main.o 需要用到这个函数, 可以想象就像几个插槽一样。链接器通过扫描 obj 文件发现这个函数定义在 hello_world.o 中, 然后就可以链接了。

问题: 为什么函数不能定义在头文件中?

这个问题是不恰当的, 因为用 inline 和 static 修饰的函数可以定义在头文件中, 而 inline 修饰的函数必须定义在头文件中。

如果函数定义在头文件中, 并且有多个 cpp 文件都包含了这个头文件的话, 那么这些 cpp 文件生成的 obj 文件的导出符号表中都有这个头文件中定义的函数, 单文件编译的时候是不会出错的, 但是链接的时候就会报错。链接器发现了多个函数实体, 但却无法确定应该使用哪一个。这是一个链接错误。

inline 修饰的函数, 通常都不会存在函数实体, 即便编译器没有对其内联, 那么 obj 文件也不会导出 inline 函数, 所以链接不会出错。

static 修饰的函数, 只能由定义它的编译单元调用, 也不会导出。如果头文件中顶一个 static 修饰的函数, 就相当于多个 obj 文件中都顶一个了一个一模一样的函数, 大家各用各的, 互补干扰。

问题: 什么样的变量可以定义在头文件中?

其实变量于函数很类似, 由 static 或 const 修饰的变量可以定义在头文件中。


static 修饰的变量于 static 修饰的函数一样, 道理同上。

`const` 修饰的变量默认是不会进入导出符号表的，相当于每个 `obj` 中都定义了一个一模一样的 `const` 变量，各用各的。而 `const` 可以再用 `extern` 修饰，如果用 `extern const` 修饰的变量定义在头文件中，那么就会出现链接错误，原因就是“想一想 `extern` 是干嘛的”

问题：extern "C"是干嘛的？

如果有人回答“兼容 C 和 C++”，我只能说“这是一个正确答案，但我不知道你是否真的知道”。

首先要知道 C 不支持重载，C++支持重载，C++为了支持重载，引入了函数重命名的机制，就像下面这样：

[cpp] view plain copy 

```
1. int hello_world(type1 param);
2. int hello_world(type2 param);
```

通常第一个函数会被编译成 `hello_world_type1` 这样子，第二个函数会被编译成 `hello_world_type2` 这样子。不管是定义的地方还是调用的地方，都会把函数改成同样的名字，所以链接器可以正确的找到函数实体。

而我们写 C++程序的时候，通常会引入由 `c` 编写的库(gcc 编译的 `c` 文件)，而 `c` 不支持重载，自然不会对函数重命名。而我们在 C++中调用的地方很可能会重命名，这就造成了调用的地方(C++编译)和定义的地方(C 编译)函数名不一致的情况，这也是一种链接错误。

所以我们经常会看到在 C++中用 `extern "C" { #include "some_c.h" }` 这种代码。这就是告诉 `c++` 编译器，`some_c.h` 中的函数要按照 `c` 的方式编译，不要重命名，这样在链接的时候就 `ok` 了。