C++多文件编程

鸟x

目录

从单文件到多文件

首先来看一段写在单文件里的C++程序：

```cpp
#include <iostream>

// int add(int x, int y);

int main()
{
    std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n';
    return 0;
}

int add(int x, int y)
{
    return x + y;
}
```

显然，这个程序会引发编译器报错，当编译器看到主函数第一行的标识符add时，并不知道它是什么（因为我们在主函数后才定义它）。我们要么使用前向声明，要么调整代码顺序。

现在，我们来看一个与之相似的多文件程序：

```cpp
// add.cpp
int add(int x, int y)
{
    return x + y;
}
// main.cpp
#include <iostream>

int main()
{
    std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n'; // compile error
    return 0;
}
```

系统由两个cpp源文件构成。编译时，编译器要么先编译add.cpp，要么先编译main.cpp，无论是哪一种情况都会引发报错，同样因为编译器不知道主函数的add标识符是什么意思。（注意，编译器编译某一文件时，既不会知晓其他文件的内容，也不会记住曾经编译过的文件的内容）

```
// MSVC
error C3861: 'add': identifier not found
// GCC
error: 'add' was not declared in this scope
```

我们的解决方案仍然与之前一样，然而，由于add在另一个文件里，调整代码顺序是不可能的，我们只能求助于前向声明：

```cpp
#include <iostream>

int add(int x, int y); // needed so main.cpp knows that add() is a function defined elsewhere

int main()
{
    std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n';
    return 0;
}
```

现在，编译器就知道add是一个函数，不会再报错。接下来就由链接器将主函数对add的调用与add的定义连接。如果发现add并没有被定义，就会引发链接器报错。

```
// MSVC
error LNK2019: unresolved external symbol "int __cdecl add(int,int)" (?add@@YAHHH@Z)
referenced in function _main
// GCC
undefined reference to `add(int, int)'
```

头文件

随着程序越来越大（并使用更多的文件），对想使用的定义在不同文件中的每个函数进行前向声明变得越来越乏味。如果你能把所有的前向声明放在一个地方，然后在需要的时候导入它们，那不是很好吗？

C++源文件.cpp并不是C++程序中唯一常见的文件。另一种类型的文件被称为头文件。头文件的扩展名通常是.h，但你偶尔也会看到它们的扩展名是.hpp，或者根本就没有扩展名。头文件的主要目的是将声明传播到代码文件，允许我们把声明放在一个地方而在我们需要的时候导入它们。

最常见的例子，`#include <iostream>`中的iostream就是头文件，包含它在内的一系列头文件从属于标准库，`std::cout`就是在标准库中声明的。

当涉及到函数和变量时，值得注意的是，**头文件通常只包含函数和变量的声明**，而不是函数和变量的定义（否则可能导致违反单一定义规则，见下）。`std::cout`在iostream头文件中是向前声明的，但定义为C++标准库的一部分，它在链接器阶段会自动链接到你的程序中。

为什么头文件不包含定义

来看这个例子：

```
// a.h
int a = 5;// defination
// a.cpp
#include "a.h"
// main.cpp
#include "a.h"
#include <iostream>

int main()
{
    std::cout << a;
    return 0;
}
```

此时会引发链接错误。原因在于，a.cpp与main.cpp都包含了a.h，于是两个源文件都含有a的定义，导致重定义错误。在多文件编程中，多个源文件包含一个头文件的情况是很常见的，如果在头文件中进行定义，就不免会发生上面的情况。

自定义头文件示例

以前面的add为例，演示如何写自己的头文件并使用。

一般而言，头文件会与对应的源文件对应（显而易见，因为头文件里是声明，而源文件里是对应的定义）。于是我们创建add.h与add.cpp如下：

```
// 1) We really should have a header guard here, but will omit it for simplicity (we'll cover header guards below)

// 2) This is the content of the .h file, which is where the declarations go
int add(int x, int y); // function prototype for add.h -- don't forget the semicolon!
#include "add.h" // Insert contents of add.h at this point.  Note use of double quotes here.

int add(int x, int y)
{
    return x + y;
}
```

现在，我们就可以在main.cpp中使用add函数：

```
#include "add.h" // Insert contents of add.h at this point.  Note use of double quotes here.
#include <iostream>

int main()
{
    std::cout << "The sum of 3 and 4 is " << add(3, 4) << '\n';
    return 0;
}
```

多文件编程中务必牢记的书写规范

不要再写using namespace std了!

namespace std是标准命名空间，其中包括了cout、cin、max、min等一系列很有用处的函数。根据命名空间的使用规则（::），我们可以写出这样的代码：

```
#include <iostream>

int main()
{
    std::cout << "Hello world!"; // when we say cout, we mean the cout defined in the std namespace
    return 0;
}
```

using的意义是，在整个文件中对应的函数名都默认在其后的命名空间中，using namespace std就是在整个文件范围内指定使用标准命名空间（如果不存在歧义）：

```
#include <iostream>

using namespace std; // this is a using directive that allows us to access names in the std namespace with no namespace prefix

int main()
{
    cout << "Hello world!";
    return 0;
}
```

当然，这能帮我们少做很多琐碎活，如果你可以确保不会使用来自其他命名空间的同名函数，就可以大胆使用using（比如OJ题，它们通常只要求在一个文件完成所有内容）。但除此之外的情况，为了尽可能降低bug出现的概率，不要使用using namespace std！来看这个例子：

```
#include <iostream> // imports the declaration of std::cout

using namespace std; // makes std::cout accessible as "cout"

int cout() // defines our own "cout" function in the global namespace
{
    return 5;
}

int main()
{
    cout << "Hello, world!"; // Compile error!  Which cout do we want here?  The one in the std namespace or the one we defined above?

    return 0;
}
```

现在，我们本打算使用自己定义的cout函数（这看起来很蠢，但别忘了std中还有min、max等经常被用来自己定义函数的函数名字），但由于我们using namespace std，编译器不能分辨我们究竟要使用哪个命名空间的cout，于是就会报错。在多文件编程中，这种错误的出现会让人摸不着头脑，难以定位错误的所在。

源文件应导入它们对应的头文件（如有）

这样做允许编译器在编译期就能发现错误，而非到链接期才能发现。

```
// a.h
int something(int); // return type of forward declaration is int
// a.cpp
#include "a.h"

void something(int) // error: wrong return type
{
}
```

由于源文件导入了头文件，返回值类型不匹配的错误在编译期即可发现（很多IDE的语法检查也可以发现），方便我们debug。

使用双引号来导入自定义的头文件

使用尖括号时，它告诉预处理器这不是我们自定义的头文件。编译器将只在包含目录（include directories）所指定的目录中搜索头文件。包含目录是作为项目/IDE设置/编译器设置的一部分来配置的，通常默认为编译器和/或操作系统所带头文件的目录。编译器不会在你项目的源代码目录中搜索头文件。

使用双引号时，我们告诉预处理器这是我们自定义的头文件。编译器将首先在当前目录下搜索头文件。如果在那里找不到匹配的头文件，它就会在包含目录中搜索。

因此，使用双引号来包含你编写的头文件，或者预期在当前目录中可以找到的头文件。使用尖括号来包含编译器、操作系统或安装的第三方库所附带的头文件。

显式包含所需的所有头文件

每个文件都应该明确地#include它所需要的所有头文件来进行编译。不要依赖从其他头文件中转来的头文件。

如果你包含了a.h，而a.h中包含了b.h，则b.h为隐式包含，a.h为显式包含。你不应该依赖通过这种方式包含的b.h的内容，因为头文件的实现可能会随着时间的推移而改变，或者在不同的系统中是不同的。而这一旦发生，你的代码就可能不能在别的系统上编译，或者将来不能编译。直接同时显式包含a.h和b.h就能解决这个问题。

给自定义头文件加上header guard

前面提到，如果在一个文件中多次定义一个变量/函数会引发重定义报错。现在来看这个例子：

```
// square.h
// We shouldn't be including function definitions in header files
// But for the sake of this example, we will
int getSquareSides()
{
    return 4;
}
// geometry.h
#include "square.h"
// main.cpp
#include "square.h"
#include "geometry.h"

int main()
{
    return 0;
}
```

则在main.cpp中，getSquareSides通过两次包含被定义了两次，将引发重定义问题。关键的问题在于，只站在main.cpp的角度来看，它很无辜，只不过是普通地包含了两个不同的头文件而已。因此，解决这个问题应该从头文件本身下手。这就是我们需要header guard的原因。

header guard的格式如下：

```
#ifndef SOME_UNIQUE_NAME_HERE
#define SOME_UNIQUE_NAME_HERE

// your declarations (and certain types of definitions) here

#endif
```

这样，就可以保证相同的部分只会被包含一次。

现代编译器往往还会提供另一种更简洁的header guard——`#pragma once`。

```
#pragma once

// your code here
```

它的效果与传统的header guard相同，而且更加简单。但是我们仍然建议使用传统的header guard，因为#pragma once在某些编译器上并不受支持。

严禁循环包含

我们来看这样一个例子：

```
// in b.h
#include "a.h"
// ...


// in a.h
#include "b.h"
// ...
```

未使用header guard时，这样的写法会引发连锁反应：b.h包含a.h，后者包含b.h，再包含a.h……

即使使用了header guard消除了这种连锁反应，也会出现错误——如果b包含a，那么a包含b的代码则会因为header guard在预处理阶段就失去作用，于是a没能成功包含b，则a无法看到其需要的b中的声明。

因此，禁止循环包含！（此外， 出现循环包含时有很大的可能就是你的代码设计不好，例如出现了循环依赖，这部分讨论见oop依赖反转原则）

Reference

Learn C++ - Skill up with our free tutorials (learncpp.com)，2-8至2-13

特别鸣谢

刘雪枫学长通读了讲义初版，纠正了其中若干错误。