C++编程(7)

唐晓晟
北京邮电大学电信工程学院

第九章 源文件和程序

- 分别编译
- 连接
- 使用头文件
- 程序
- 忠告

9.1 分别编译

- 文件是传统的存储单位和传统的编译单位
- 通常，将一个完整的程序放入一个文件是不可能的(标准库和操作系统)，而且不实际、不方便，将程序组织成一些文件的方式，可以强调它的逻辑结构，帮助读者理解程序
- 一些文件的方式，可帮助编译器去强迫实施这种逻辑结构，也能节省编译时间
- 源文件(source file)提交给编译器后进行预处理(处理宏、#include等)，形成编译单位

分别编译

- 为了使分别编译能够工作，程序员必须提供各种声明，为孤立地分析一个编译单位提供有关程序其他部分的类型信息
- 在一个由许多分别编译的部分组成的程序里，这些声明必须保持一致，连接器(linker有时也被混乱地称作装载器loader)能检查出许多类型的不一致
- 连接工作可以在程序开始运行之前完全做好，另外也允许程序开始运行后为其加入新代码(动态连接)

9.2 连接

- 在所有的编译单位中，对所有函数、类、模板、变量、名字空间、枚举和枚举符号的名字的使用必须保持一致
- 所有名字空间、类、函数等都应该在它们出现的各个编译单位中有适当的声明，而且所有声明都应该一致地引用同一个实体
- 如果一个名字可以在与其定义所在的编译单位不同的地方使用，就说它是具有”外部连接”的，否则就称其为具有”内部连接”的

连接示例(1)

```
//file1.c
    int x = 1;
    int f(){/*...*/}

//file2.c
    extern int x;
    int f();
    void g(){x=f();}

//ok
//file2.c使用了file1.c
//中的x和f()
```

```
//file1.c
    int x = 1;
    int b = 1;
    extern int c;

//file2.c
    int x;
    extern double b;
    extern int c;

//三个错误
//x定义了两次
//b类型不一致
//c只有声明，没有实现
```

```
//file1.c
    int x;
    int f(){ return x; }

//file2.c
    int x;
    int g() {return f();}

//两个错误
//x定义了两次
//file2中无法调用f()
//因为没有声明
```

连接示例(2)

```
//默认情况下
//const和typedef具有“内部连接”
//file1.c
    typedef int T;
    const int x=7;

//file2.c
    typedef void T;
    const int x=8;

//ok
//全局的const和inline一般
//应该放到头文件中去
//extern const int a = 77
//显式声明使const具有“外部连接”
```

```
//无名名字空间的效果很像“内部连接”
//file1.c
namespace {
    class X{/*...*/};
    void f();
    int i;
}
//file2.c
    class X{/*...*/};
    void f();
    int i;
//可以，但是自找麻烦

//关键字static也具有“内部连接”
```

9.2.1 头文件

- 为达到在不同的编译单位中声明的一致性而使用的不完美但是比较简单的方法，使用头文件
- #include机制是一种正文操作的概念，用于将源程序片断收集到一起
- #include “to_be_included”将用文件to_be_included的内容取代这行，该文件内容应该为源代码

头文件示例

```
#include <iostream>
// 来自标准库的包含目录
#include “myheader.h”
// 来自当前目录

#include < iosteam >
// <>或者””中的空格也是
// 有意义的
```

习惯上
头文件采用后缀.h，而包含函数或者数据定义的文件用.c后缀，其他约定，比如.C、.cxx、.cpp和.cc也常常可以看到

```
// 头文件被包含，每次都
// 需要重新编译，不过头
// 文件中只有声明，没有
// 需要被编译器深入分析
// 的代码，此外，很多
// C++实现都提供了头文
// 件的预编译形式
```

头文件中可以包括

命名名字空间	namespace N{/*...*/}
类型定义	struct Point {int x,y;};
模板声明	template <class T> class Z;
模板定义	template <class T> class V{..}
函数声明	extern int strlen(const char*);
数据声明	extern int a;
常量定义	const int a = 10;
名字声明	class Matrix;
包含指令	#include <algorithm>
条件编译	#ifdef _cplusplus
注释	/* check for end of file */
宏定义	#define VERSION 12

头文件中绝对不可以包括

常规的函数定义	char get(char* p){return *p++};
数据定义	int a;
聚集量(aggregate)定义	short tbl[] = {1, 2, 3};
无名名字空间	namespace {/*...*/}
导出的(exported)模板定义	export template <class T> f(T t){/* ... */ }

9.2.2 标准库中的头文件

- 标准库的功能是通过一组标准头文件给出的
- 标准库中的头文件不需要后缀，但并不意味这些头文件采用了某种特殊的存储方式
- 使用标准库中的头文件需要用#include <...>，而不是#include “...”
- 相对于每一个C标准库文件<X.h>，存在着一个与之对应的标准C++头文件<cX>，例如“#include <stdio.h>”和“#include <cstdio>”

标准库中的头文件示例

```
#ifdef __cplusplus
// 只为了C++编译器
namespace std {
    extern “C” {
    // 见9.2.4
#endif

    /* ..... */
    int printf(const char* ...);

#ifdef __cplusplus
    }
}
using namespace std;
#endif
```

左边代码为一个典型的<stdio.h>看起来的样子

实际声明都是共享的，但连接和名字空间问题需要另行处理，以便C和C++能共享这个头文件

9.2.3 单一定义规则

- 在一个程序里，任一个类、枚举和模板等都必须只定义唯一的一次，这种规则需要以一种复杂和精细的方式来描述，被称为“单一定义规则”(One-Definition Rule, ODR)
- 也就是说，一个类、模板或者内联函数的两个定义能够被接受为同一个唯一定义的实例，当且仅当
  - 它们出现在不同的编译单位里
  - 它们按一个个单词对应相同
  - 这些单词的意义在两个编译单位中也完全一样

单一定义规则(ODR)示例

```
//file1.c
    struct S{int a; char b;};
    void f(S*);

//file2.c
    struct S{int a; char b;};
    void f(S* p){/*...*/}

// ok
// S在两个源文件中被声明了两次
// 缺点是：
// 维护file2.c的人们不见得知道S
// 在不同文件中被声明了两次
// 或许认为可以自由去修改它
```

```
//s.h
    struct S{int a; char b;};
    void f(S*);

//file1.c
    #include “s.h”
    // 使用 f()

//file2.c
    #include “s.h”
    void f(S* p) {/*...*/}

// 基于ODR的设计
```

单一定义规则的错误示例

```
//file1.c
    struct S1{ int a; char b; };
    struct S1{ int a; char b; };
// Error 重复定义
//file1.c
    struct S2{int a; char b;};
//file2.c
    struct S2{int a; char bb;};
// Error 成员不同
//file1.c
    typedef int X;
    struct S3{X a;char b;};
//file2.c
    typedef char X;
    struct S3{X a;char b;};
//Error X的类型被改变了
```

```
//s.h
    struct S{Point a; char b;};

//file1.c
    #define Point int
    #include “s.h”
    //...
//file2.c
    class Point{/*...*/};
    #include “s.h”
    //...

// 局部typedef和宏都可能改变通过
// #include包含进来的声明的意义
// 防范此类问题的方法是尽可能将头
// 文件做得自给自足
```

9.2.4 与非C++代码的连接

- 一个C++程序可能由多个部分组成，这些部分不见得都使用C++语言编写
- 不同语言写出的程序片断、 或者是同一种语言，但是由不同的编译器编译的片断之间的协作比较困难，为了能让C++使用，可以在一个extern声明中给出有关的连接约定
- 这种声明能够提醒编译器应该如何去连接这个片断(比如说：寄存器使用方式、参数入栈的顺序、整数等内部类型的布局)

与非C++代码的连接示例(1)

```
extern “C” char* strcpy(char*, const char*);
// 与 extern char* strcpy(char*, const char*);
// 的区别仅在于连接约定不同
```

extern “C”指令描述的只是一种连接约定，并不影响函数的语义，特别的，声明为extern “C”的函数仍要遵守C++的类型检查和参数转换规则，而不是C的较弱的规则

```
extern “C” int f();
int g() {  return f(1); }  // Error f()函数不应该有参数

extern “C” {
    char* strcpy(char*, const char*);
    int strcmp(const char*, const char*);
    int strlen(const char*);
} // 这种结构经常被称为连接块
```

与非C++代码的连接示例(2)

```
extern “C” {
    #include <string.h>
}

// 用extern “C”将整个C头文件包裹起来
// 以便整个文件能适合C++使用

#ifdef __cplusplus
extern “C” {
#endif
    char* strcpy(char*, const char*);
    int strcmp(const char*, const char*);
#ifdef __cpluscplus
}
#endif //__cplusplus是一个预定义的宏名
```

9.2.5 连接与指向函数的指针

- 当一个程序中混合使用C和C++代码片断时，有时会希望把在一个语言里定义的函数指针传递到另一个语言中定义的函数里
- 在这种情况下，注意不同语言之间的连接约定是否一致

连接与指向函数的指针示例

```
typedef int(*FT)(const void*, const void*);  // FT: C++连接
extern “C” {
    typedef int (*CFT)(const void*, const void*); // CFT: C连接
    void qsort(void* p, size_t n, size_t sz, CFT cmp); // cmp: C连接
}
void isort(void* p, size_t n, size_t sz, FT cmp);  // cmp: C++连接
void xsort(void* p, size_t n, size_t sz, CFT cmp); // cmp: C连接
extern “C” void ysort(void* p, size_t n, size_t sz, FT cmp); // cmp: C++
int compare(const void*, const void*);  // compare: C++连接
extern “C” int ccmp(const void*, const void*); // ccmp: C连接
void f(char* v, int sz)
{
    qsort(v, sz, 1, &compare);    // Error
    qsort(v, sz, 1, &ccmp);       // ok
    isort(v, sz, 1, &compare);    // ok
    isort(v, sz, 1, &ccmp); }     // Error
```

9.3 使用头文件

- 单一头文件方式
- 对将一个程序划分成几个文件的最简单解决方案就是将所有定义放入合适数目的.c文件里，在一个头文件里声明这些.c文件之间通信所需要的类型，并让它们中的每个都#include该头文件
- 对于小的程序，这种风格的物理划分就很合适，但是如果程序大一些，这种方式就无法工作了，对公共头文件的修改就将导致对整个程序的重新编译，此外，几个程序员去修改一个头文件也极容易导致错误

多个头文件方式

- 让每一个逻辑模块有自己的头文件，其中定义它所提供的功能
- 这样，每个.c文件用一个对应的.h文件描述它所提供的东西(界面)，其中：为用户提供的界面放在它的.h文件中，为实现部分所用的界面放在以_impl.h(这个后缀可以随便起名)为后缀的文件里，而模块中的函数、变量等的定义则放在.c文件里

多个头文件方式

- 多个头文件的组织方式能够适应于比我们玩具式的分析器大几个数量级的模块，以及比我们的计算器大几个数量级的程序，基本原因就是它提供了我们所关心的更好的局部性
- 实际应用中，具体选择哪一种头文件的组织方式要根据实际情况来决定，过多过少都不好

9.3.3 包含保护符

- 为了避免某个头文件被多次包含，可以使用包含保护符的方式解决此类问题
- 为了避免潜在的宏的名称的冲突，包含保护符通常取比较长的非常难看的名字

```
//error.h
#ifndef CALC_ERROR_H
#define CALC_ERROR_H

namespace  Error{ /* ... */ }

#endif //CALC_ERROR_H
```

9.4 程序

- 一个程序就是由连接器组合到一起的一组分别编译单位
- 在这一集合中所使用的每个函数、对象、类型等都必须有唯一的定义
- 程序中必须包含恰好一个名字为main的函数，作为程序执行的入口，程序通过此函数的返回而结束，main函数返回的int数值被传递给调用main的系统，作为程序的结果

9.4.1 非局部变量的初始化

- 原则上说，在所有函数之外定义的变量(即那些全局的、名字空间的合类的static变量)应该在main()的调用之前完成初始化，在一个编译单位内的这种非局部变量将按照它们的定义顺序进行初始化，若某个变量没有显式的初始式，它将被初始化为有关类型的默认值

```
double x = 2;
double y; // 初始化为0
double sqx = sqrt(x+y); // sqx = sqrt(2)
```

非局部变量的初始化

- 不同编译单位里的全局变量，其初始化的顺序没有任何保证，此外，也没有办法捕捉到由全局变量的初始化式抛出的异常，一般来说，尽量减少全局变量的使用

```
int& use_count()
{
    static int uc = 0;
    return uc;
}

void f(){
    cout << ++use_count();
}
```

//对use_count()的调用在行为上就像是一个全局变量，除了它是在第一次被使用时才被初始化之外

9.4.1.1 程序终止

- 程序可能以多种方式终止
  - 通过从main()返回
  - 通过调用exit()，函数类型void exit(int)
  - 通过调用abort()
  - 通过抛出一个未被捕捉的异常
- 若程序调用exit()终止，所有已经构造起来的静态对象的析构函数都将被调用(调用它的函数及其调用者里面的局部变量的析构函数都不会被执行)，然而，如果程序使用abort()终止，那么析构函数就不会被调用
- 注意：析构函数里面调用exit()有可能导致无穷递归

程序终止

- 调用exit(int)时，其参数将被作为程序的值返回给系统(一般用0表示程序成功结束)
- 一般情况下，最好通过抛出一个异常的方式脱离一个环境，让异常处理器来决定应该做什么
- 可以通过使用atexit函数来让程序在终止之前执行一些代码
- exit(),abort(),atexit()在<cstdlib>里

9.5 忠告

- [1] 利用头文件去表示界面和强调逻辑结构
- [2] 用#include将头文件包含到实现有关功能的源文件里
- [3] 不要在不同编译单位里定义具有同样名字、意义类似但又不同的全局实体
- [4] 避免在头文件里定义非inline函数
- [5] 只在全局作用域和名字空间里使用#include
- [6] 只用#include包含完整的定义
- [7] 使用包含保护符

忠告

- [8] 用#include将C头文件包含到名字空间里，以避免全局名字
- [9] 将头文件做成自给自足的
- [10] 区分用户界面和实现界面
- [11] 区分一般用户界面和专家用户界面
- [12] 在有意向用于非C++程序组成部分的代码中，应避免需要运行时初始化的非局部对象