1. （p362）当用户将一个源文件（source file）提交给编译器后，首先对文件进行预处理，即，处理宏（见12.6节）以及将#include指令指定的头文件包含进来（见2.4.1节和15.2.2节）。预处理的结果称为编译单元（translation unit）。编译单元是编译器真正处理的内容，也是C++语言规则所描述的内容。在本书中，仅当需要区分程序员所看到的内容和编译器所处理的内容时才会区分源文件和编译单元。
2. 链接（p363）

除非已显示声明为局部名字，否则函数名、类名、模板名、名字空间名、枚举名以及枚举值名的使用必须跨所有编译单元保持一致。

程序员必须保证每个名字空间、类、函数等必须在其出现的每个编译单元中都正确声明，且对应相同实体的声明是一致的。例如，考虑下面两个文件：

//file1.cpp

int x = 0;

int f() { /\*…\*/ }

//file2.cpp

extern int x;

int f();

void g() { x = f(); }

file2.cpp中的g()使用的x和f()就是file1.cpp中所定义的实体。关键字extern指出file2.cpp中x的声明仅仅是一个声明而已，而非一个定义（见6.3节）。假如x已初始化，extern将会被忽略，因为带初始值的声明总是被看做一个定义。对象在程序中只能定义一次，它可以声明很多次，但类型必须完全一致。例如：

//file1.cpp

int x = 1;

int b = 1;

extern int c;

//file2.cpp

int x; //意味着“int x = 0;”

extern double bl

extern int c;

这个程序有3个错误：x被定义了两次，b的声明类型不一致，c被声明了两次但没有别定义。如果编译器只能同时处理一个文件，就无法检查这些错误（链接错误）。但大多数这种错误能被链接器检查出来。例如，我所知道的所有正确的C++实现都能检测出x的双重定义。但是，流行的C++实现都捕获不到b的声明不一致的问题，遗漏c的定义这一错误通常也只有在c被使用时才能捕获到。

注意，如果全局作用域中或名字空间中的变量定义不带初始值，则该变量会使用默认初始值（见6.3.5.1节）。非static局部变量或创建在自由存储上的对象（见11.2节）则不会使用默认初始值。

在类体外，实体必须先声明后使用（见6.3.4节）。例如：

//file1.cpp:

int g() { return f() + 7; } //错误：f()（尚）未声明

int f() { return x; } //错误：x（尚）未声明

int x;

如果一个名字在其定义处之外的编译单元中也可以使用，我们称其具有外部链接（external linkage）。前一个例子中的所有名字都具有外部链接。如果一个名字只能在其所定义的编译单元中被引用，我们称其具有内部链接（internal linkage）。例如：

static int x1 = 1; //内部链接：其编译单元中不可访问

const char x2 = ‘a’; //内部链接：其编译单元中不可访问

在名字空间作用域（包括全局作用域，见14.2.1节）中使用关键字static（有些不逻辑）辨识“不能再其他源文件中访问”（即内部链接）。如果你希望在其他文件中也能访问x1（“具有外部链接”），就应该去掉static。关键字const暗示默认内部链接，因此如果你希望x2具有外部链接，就需要在其定义前加上extern：

int x1 = 1; //外部链接：其编译单元中可访问

extern const char x2 = ‘a’; //外部链接：其编译单元中可访问

链接器看不到的名字，例如局部变量名，被称为无链接（no linkage）。

inline函数（见12.1.3节和16.2.8节）在其应用的所有编译单元中都必须有完全等价的定义（见15.2.3节）。因此下面这个例子不仅风格糟糕，而且是不合法的：

//file1.cpp

inline int f(int i) { return i; }

//file2.cpp

inline int f(int i) { return i+1; }

不幸的是，C++实现很难捕获这种错误。而下面的例子中外部链接和内部链接的组合虽然完全符合逻辑，但却被禁止的：

//file1.cpp

extern inline int g( int i );

int h(int i) { return g(i); } //错误：此编译单元中无g()定义

//file2.cpp

extern inline int g( int i ) { return i+1; }

//…

我们可以通过使用头文件来保持inline函数的异质性（见15.2.2节），例如：

//h.h

extern inline int next( int i ) { return i+1; }

//file1.cpp

#include “h.h”

int h(int i) { return next(i); } //正确

默认情况下，名字空间中的const对象（见7.5节）constexpr对象（10.4节）、类型别名（6.5节）以及任何声明为static的实体（见6.3.4节）都具有内部链接。因此，下面这个例子是合法的（虽然可能让人困惑）：

//file1.cpp

using T = int;

const int x = 7;

constexpr T c2 = x+1;

//file2.cpp

using T = double;

const int x = 8;

constexpr T c2 = x+9;

为了确保一致性，应该将别名、const对象、constexpr对象和inline函数放置在头文件中（见15.2.2节）

1. （p365）如果必须使用全局变量，至少应限制他们只在单一源文件中使用，有两种防范实现这种限制：
2. 将声明放在无名名字空间中。
3. 声明实体时使用static。

使用无名名字空间（见14.4.8节）可以令名字成为编译单元的局部名字。无名名字空间的效果非常像内部链接。例如：

//file1.cpp

namespace

{

class X { /\*…\*/ };

void f();

int i;

}

//file2.cpp

class X { /\*…\*/ };

void f();

int i;

file1.cpp中的f()和file2.cpp中的f()不是同一个函数。如果一个名字时一个编译单元的局部名字，我们又用它命名别处的一个觉有外部链接的实体，我们就是自找麻烦。

关键字static（令人困惑地）表示“使用外部链接”（见44.2.3节）。这是早期C语言的遗留问题。

1. （p366）#include机制是一种文本处理方式——将源程序片段收集起来形成单一的编译单元（文件）。考虑如下语句：

#include ”to\_be\_included”

这条#include指令将他所在的这一行替换为文件to\_be\_ included的内容。to\_be\_ included应该是C++源码，因为编译器会继续处理替换后的结果。

1. （p367）编译时断言 static\_assert( 4<=sizeof(int), “small ints” );
2. （p367）头文件中不应该包含以下内容：
3. 普通函数定义 char get( char\*p ) { return \*p++; }
4. 数据定义 int a;
5. 集合定义 short tbl[] = { 1, 2, 3 };
6. 无名子空间定义 short tbl[] = { 1, 2, 3 };
7. using指示 using namespace Foo;

建议将简单常量定义放在头文件中，但不讲几何定义放在头文件中，其原因是C++实现很难避免多个编译单元中重复的集合定义。

尽量减少在头文件中使用非局部的名字（特别是别名）。

1. 单一定义原则（p368）

C++标准中规定类、模板等的定义必须唯一，这一规则是通过一种更为复杂、更为微妙的方式描述的。他通常被称为单一定义规则（one-definition rule,ODR）。即，一个类。模板或内联函数的两个定义可被接受、被认为是相同的唯一定义，当且仅当如下条件成立：

1. 它们出现在不同的编译单元中，且
2. 他们的源码逐单词对应，完全一样，且
3. 这些单词在两个编译单元中的含义完全一样

例如：

//file1.cpp

struct S { int a; char b; };

void f(S\*);

//file2.cpp

struct S { int a; char b; };

void f(S\* p) { /\*…\*/ }

ODR认为这个例子是合法的，S在两个源文件中表示相同的类。但是，像这样一个定义写两次是不明智的。

将共享定义放置在头文件然后#include头文件的技术并不能防止最后一种违反ODR的形式。局部类型别名和宏辉改变#include声明的含义：

//s.h

struct S{ Point a; char b; }

//file1.cpp

#define Point int

#include “s.h”

//…

//file2.cpp

class Point { /\*…\*/ };

#include “s.h”

//…

防止这种错误的最好方法是令头文件尽可能地自包含。

1. 标准头文件

For each C standard-library header <X.h> , there is a corresponding standard C++ header <cX> .For example, #include<cstdio> provides what #include<stdio.h> does. A typical stdio.h will look something like this:

#ifdef \_\_cplusplus // for C++ compilers only (§15.2.5)

namespace std { // the standard librar y is defined in namespace std (§4.1.2)

extern "C" { // stdio functions have C linkage (§15.2.5)

#endif

/\* ... \*/

int printf(const char∗, ...);

/\* ... \*/

#ifdef \_\_cplusplus

}

}

// ...

using std::printf; // make printf available in global namespace

// ...

#endif

That is, the actual declarations are (most likely) shared, but linkage and namespace issues must be addressed to allow C and C++ to share a header. The macro \_\_cplusplus is defined by the C++ compiler (§12.6.2) and can be used to distinguish C++ code from code intended for a C compiler.

1. 链接非C++语言（p370）

不同语言和同一种语言的不同实现可能在如何用机器的寄存器保存参数、参数在占中的布局、字符串和整数等内置类型的内存布局、编译器传递给链接器的名字的格式以及链接器要求的内存检查等方面都有不同。为了帮助解决此问题，我们可以指定extern声明使用哪种链接（linkage）规范。例如，下面的代码声明了C和C++标准库函数strcpy()并指定它采用C链接规范（系统相关）：

extern “C” char\* strcpy( char\*, const char\* );

此声明的效果与下面的普通“声明”不同：

extern char\* strcpy( char\*, const char\* );

但差别只是调用strcpy()时所采用的链接规范不同。

应为C和C++关系紧密，extern “C”指示特别有用。需要注意的是，extern “C”中的C表示的是链接规范而非语言。extern “C”通常用于将函数链接到恰好符合C实现规范的Fortran和汇编程序。

一个extern “C”指示（仅）指出链接规范，它不影响函数调用的语义。特别是，声明为extern “C”的函数任然遵守C++类型检查和参数转换规则而不是弱一些的C规则。例如：

extern “C” int f();

int g()

{

return f(1); //错误：不需要参数

}

为大量声明添加extern “C”很令人厌烦。因此，C++提供了一种机制为一组声明指定链接规范。例如：

extern “C”

{

char\* strcpy( char\*, const char\* );

int strcmp( const char\*, const char\* );

int strlen( const char\* );

}

这种构造通常称为链接块（linkage block），甚至可用来封装完整C头文件，从而使之适用于C++程序。例如：

extern “C”

{

#include <string.h>

}

程序员通常使用这种技术从C头文件生成C++头文件。另一种创建C和C++公用头文件的的技巧是条件编译（见12.6.1节）：

#ifdef \_\_cplusplus // for C++ compilers only (§15.2.5)

extern "C" // stdio functions have C linkage (§15.2.5)

{

#endif

char\* strcpy( char\*, const char\* );

int strcmp( const char\*, const char\* );

int strlen( const char\* );

// ...

#ifdef \_\_cplusplus

}

#endif

1. 链接和函数指针（p372）

当在一个程序中混合C和C++代码片段时，我们有时虚妄将一种语言定义的函数指针传递给另一种语言定义的函数。如果两种语言的实现共享链接规范和调用机制，这种函数指针的传递就很简单。但是，这种通用性一般很难保证，因此我们必须小心确保一个函数的调用方式符合函数自身的设计预期。

如果在生命中指定了链接方式，则此链接方式会应用于声明中设计的所有函数类型、函数名和变量。这令各种奇怪的（有时也是必须的）链接方式组合成为可能。例如：

typedef int (\*FT)( const void\*, const void\* ); //FT采用C++链接方式

extern “C”

{

typedef int (\*CFT) ( const void\*, const void\* ); //FT采用C链接方式

void qsort( void\* p, size\_t n, size\_t sz, CFT cmp ); //cmp采用C链接方式

}

void isort( void\* p, size\_t n, size\_t sz, FT cmp ); //cmp采用C++链接方式

void xsort( void\* p, size\_t n, size\_t sz, CFT cmp ); //cmp采用C链接方式

extern “C” ysort( void\* p, size\_t n, size\_t sz, FT cmp ); //cmp采用C++链接方式

int compare( const void\*, const void\* ); //compare()采用C++链接方式

extern “C” int ccmp( const void\*, const void\* ); //ccmp()采用C链接方式

void f( char\* v, int sz )

{

qsort( v, sz, 1, & compare ); //错误

qsort( v, sz, 1, & ccmp ); //正确

isort( v, sz, 1, & compare ); //正确

isort( v, sz, 1, & ccmp ); //错误

}

1. （p374）cpp函数定义中，对每个定义使用显示的限定Lexer::，而不是简单地将他们发放在名字空间中;

namespace Lexer { /\*…\*/ }

这能避免意外地向Lexer添加新成员。另一方面，加入我希望向Lexer添加接口之外的成员，我必须重现打开名字空间（见14.2.5节）。

这样使用头文件能确保其中的每个声明都在定义文件中被包含。例如，当编译Lexer.cpp时，编译器会看到如下内容：

namespace Lexer //来自dc.h

{

//…

class Token\_stream

{

public:

Token get();

//…

}

}

//…

Lexer::Token Lexer:: Token\_stream::get() { /\*…\*/ }

这保证编译器能检查出每个名字任何类型上的不一致。例如，加入get()声明为返回一个Token，但定义为返回一个int，Lexer.cpp的编译器就会失败，报个一个类型不匹配的错误。如果程序漏掉了定义，则链接器会发现这个问题。

1. Multiple-Header Organization(p450)（p376）

An alternative physical organization lets each logical module have its own header defining the facilities it provides. Each .cpp file then has a corresponding .h file specifying what it provides (its interface). Each .cpp file includes its own .h file and usually also other .h files that specify what it needs from other modules in order to implement the services advertised in the interface. This physical organization corresponds to the logical organization of a module. The interface for users is putinto its .h file, the interface for implementers is put into a file suffixed \_impl.h , and the module’s definitions of functions, variables, etc., are placed in .cpp files. In this way, the parser is represented by three files. The parser’s user interface is provided by parser.h :

// parser.h:

namespace Parser { // interface for users

double expr(bool get);

}

The shared environment for the functions expr() , prim() , and term() , implementing the parser is presented by parser\_impl.h :

// parser\_impl.h:

#include "parser.h"

#include "error.h"

#include "lexer.h"

using Error::error;

using namespace Lexer;

namespace Parser { // interface for implementers

double prim(bool get);

double term(bool get);

double expr(bool get);

}

The distinction between the user interface and the interface for implementers would be even clearer had we used a Parser\_impl namespace (§14.3.3).

The user’s interface in header parser.h is #included to giv e the compiler a chance to check consistency (§15.3.1).

The functions implementing the parser are stored in parser.cpp together with #include directives for the headers that the Parser functions need:

// parser.cpp:

#include "parser\_impl.h"

#include "table .h"

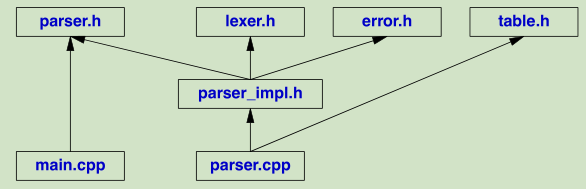
using Table::table;

double Parser::prim(bool get) { /\* ... \*/ }

double Parser::term(bool get) { /\* ... \*/ }

double Parser::expr(bool get) { /\* ... \*/ }

Graphically, the parser and the driver’s use of it look like this:



1. （p379）符号表应该是自包含的，标准库头文件<map>可以将所有感兴趣的内容包含进来已实现一个搞笑的map模板类；

// table.h:

#include <map>

#include <string>

namespace Table

{

extern std::map<std::string,double> table;

}

由于假设没个头问件都可能被#include多个.cpp文件中，我们必须将table的声明与其定义分离开来：

// table.cpp:

#include "table .h"

std::map<std::string,double> Table::table;

1. （p457）（p382）

Several techniques exist for enforcing an order of initialization of global variables in different translation units. However, none are both portable and efficient. In particular, dynamically linked libraries do not coexist happily with global variables that have complicated dependencies.

Often, a function returning a reference is a good alternative to a global variable. For example:

int& use\_count()

{

static int uc = 0;

return uc;

}

A call use\_count() now acts as a global variable except that it is initialized at its first use (§7.7). For example:

void f()

{

cout << ++use\_count(); // read and increment

// ...

}

Like other uses of static , this technique is not thread-safe. The initialization of a local static is thread-safe (§42.3.3). In this case, the initialization is even with a constant expression (§10.4), so that it is done at link time and not subject to data races (§42.3.3). However, the ++ can lead to a data race.

1. 初始化和并发（p382）

The flaw in this argument is that if multiple threads are used (§5.3.1, §42.2), each will do the run-time initialization. No mutual exclusion is implicitly provided to prevent a data race. Then, sqrt(++x) in one thread may happen before or after the other thread manages to increment x . So, the value of y may be sqrt(4) or sqrt(5) .

To avoid such problems, we should (as usual):

• Minimize the use of statically allocated objects and keep their initialization as simple as possible.

• Avoid dependencies on dynamically initialized objects in other translation units (§15.4.1).

In addition, to avoid data races in initialization, try these techniques in order:

[1] Initialize using constant expressions (note that built-in types without initializers are initialized to zero and that standard containers and string s are initialized to empty by linktime initialization).

[2] Initialize using expressions without side effects.

[3] Initialize in a known single-threaded ‘‘startup phase’’ of computation.

[4] Use some form of mutual exclusion (§5.3.4, §42.3).

1. 程序终止

A program can terminate in several ways:

[1] By returning from main()

[2] By calling exit()

[3] By calling abort()

[4] By throwing an uncaught exception

[5] By violating noexcept

[6] By calling quick\_exit()

In addition, there are a variety of ill-behaved and implementation-dependent ways of making a program crash (e.g., dividing a double by zero).

If a program is terminated using the standard-library function exit() , the destructors for constructed static objects are called (§15.4.1, §16.2.12). However, if the program is terminated using the standard-library function abort() , they are not. Note that this implies that exit() does not terminate a program immediately. Calling exit() in a destructor may cause an infinite recursion.