第②章

保证稳定性和兼容性

作为 C 语言的嫡亲，C++ 有一个众所周知的特性——对 C 语言的高度兼容。这样的兼容性不仅体现在程序员可以较为容易地将 C 代码“升级”为 C++ 代码上，也体现在 C 代码可以被 C++ 的编译器所编译上。新的 C++11 标准也并不例外。在 C++11 中，设计者总是保证在不破坏原有设计的情况下，增加新的特性，以充分保证语言的稳定性与兼容性。本章中的新特性基本上都遵循了该设计思想。

2.1 保持与 C99 兼容

☞类别：部分人

在 C11 之前最新的 C 标准是 1999 年制定的 C99 标准。而第一个 C++ 语言标准却出现在 1998 年（通常被称为 C++98），随后的 C++03 标准也只对 C++98 进行了小的修正。这样一来，虽然 C 语言发展中的大多数改进都被引入了 C++ 语言标准中，但还是存在着一些属于 C99 标准的“漏网之鱼”。所以 C++11 将对以下 C99 特性的支持也都纳入了新标准中：

- ❑C99 中的预定义宏
- ❑__func__ 预定义标识符
- ❑_Pragma 操作符
- ❑不定参数宏定义以及 __VA_ARGS__
- ❑宽窄字符串连接

这些特性并不像语法规则一样常用，并且有的 C++ 编译器实现也都先于标准地将这些特性实现，因此可能大多数程序员没有发现这些不兼容。但将这些 C99 的特性在 C++11 中标准化无疑可以更广泛地保证两者的兼容性。我们来分别看一下。

2.1.1 预定义宏

除去语法规范等，包括标准库的接口函数定义、相关的类型、宏、常量等也都会被发布在语言标准中。相较于 C89 标准，C99 语言标准增加一些预定义宏。C++11 同样增加了对这些宏的支持。我们可以看一下表 2-1。

表 2-1 C++11 中与 C99 兼容的宏

宏 名 称	功 能 描 述
__STDC_HOSTED__	如果编译器的目标系统环境中包含完整的标准 C 库，那么这个宏就定义为 1，否则宏的值为 0
__STDC__	C 编译器通常用这个宏的值来表示编译器的实现是否和 C 标准一致。C++11 标准中这个宏是否定义以及定成什么值由编译器来决定
__STDC_VERSION__	C 编译器通常用这个宏来表示所支持的 C 标准的版本，比如 1999mmL。C++11 标准中这个宏是否定义以及定成什么值将由编译器来决定
__STDC_ISO_10646__	这个宏通常定义为一个 yyyymmL 格式的整数常量，例如 199712L，用来表示 C++ 编译环境符合某个版本的 ISO/IEC 10646 标准

使用这些宏，我们可以查验机器环境对 C 标准和 C 库的支持状况，如代码清单 2-1 所示。

代码清单 2-1

```
#include <iostream>
using namespace std;

int main() {
    cout << "Standard Clib: " << __STDC_HOSTED__ << endl;    // Standard Clib: 1
    cout << "Standard C: " << __STDC__ << endl;              // Standard C: 1
    // cout << "C Stardard version: " << __STDC_VERSION__ << endl;
    cout << "ISO/IEC " << __STDC_ISO_10646__ << endl;        // ISO/IEC 200009
}

// 编译选项 :g++ -std=c++11 2-1-1.cpp
```

在我们的实验机上，__STDC_VERSION__ 这个宏没有定义（也是符合标准规定的，如表 2-1 所示），其余的宏都可以打印出一些常量值。

预定义宏对于多目标平台代码的编写通常具有重大意义。通过以上的宏，程序员通过使用 #ifdef/#endif 等预处理指令，就可使得平台相关代码只在适合于当前平台的代码上编译，从而在同一套代码中完成对多平台的支持。从这个意义上讲，平台信息相关的宏越丰富，代码的多平台支持越准确。

不过值得注意的是，与所有预定义宏相同的，如果用户重定义（#define）或 #undef 了预定义的宏，那么后果是“未定义”的。因此在代码编写中，程序员应该注意避免自定义宏与预定义宏同名的情况。

2.1.2 __func__ 预定义标识符

很多现实的编译器都支持 C99 标准中的 __func__ 预定义标识符功能，其基本功能就是

返回所在函数的名字。我们可以看看下面这个例子，如代码清单 2-2 所示。

代码清单 2-2

```
#include <string>
#include <iostream>
using namespace std;
const char* hello() { return __func__; }
const char* world() { return __func__; }

int main(){
    cout << hello() << ", " << world() << endl; // hello, world
}

// 编译选项 :g++ -std=c++11 2-1-2.cpp
```

在代码清单 2-2 中，我们定义了两个函数 hello 和 world。利用 __func__ 预定义标识符，我们返回了函数的名字，并将其打印出来。事实上，按照标准定义，编译器会隐式地在函数的定义之后定义 __func__ 标识符。比如上述例子中的 hello 函数，其实际的定义等同于如下代码：

```
const char* hello() {
    static const char* __func__ = "hello";
    return __func__;
}
```

__func__ 预定义标识符对于轻量级的调试代码具有十分重要的作用。而在 C++11 中，标准甚至允许其使用在类或者结构体中。我们可以看看下面这个例子，如代码清单 2-3 所示。

代码清单 2-3

```
#include <iostream>
using namespace std;

struct TestStruct {
    TestStruct () : name(__func__) {}
    const char *name;
};

int main() {
    TestStruct ts;
    cout << ts.name << endl;     // TestStruct
}
// 编译选项 :g++ -std=c++11 2-1-3.cpp
```

从代码清单 2-3 可以看到，在结构体的构造函数中，初始化成员列表使用 __func__ 预定义标识符是可行的，其效果跟在函数中使用一样。不过将 __fun__ 标识符作为函数参数的默

认值是不允许的，如下例所示：

```
void FuncFail( string func_name = __func__) {};// 无法通过编译
```

这是由于在参数声明时，__func__ 还未被定义。

2.1.3 _Pragma 操作符

在 C/C++ 标准中，#pragma 是一条预处理的指令（preprocessor directive）。简单地说，#pragma 是用来向编译器传达语言标准以外的一些信息。举个简单的例子，如果我们在代码的头文件中定义了以下语句：

```
#pragma once
```

那么该指令会指示编译器（如果编译器支持），该头文件应该只被编译一次。这与使用如下代码来定义头文件所达到的效果是一样的。

```
#ifndef THIS_HEADER
#define THIS_HEADER
// 一些头文件的定义
#endif
```

在 C++11 中，标准定义了与预处理指令 #pragma 功能相同的操作符 _Pragma。_Pragma 操作符的格式如下所示：

```
_Pragma （字符串字面量）
```

其使用方法跟 sizeof 等操作符一样，将字符串字面量作为参数写在括号内即可。那么要达到与上例 #pragma 类似的效果，则只需要如下代码即可。

```
_Pragma("once");
```

而相比预处理指令 #pragma，由于 _Pragma 是一个操作符，因此可以用在一些宏中。我们可以看看下面这个例子：

```
#define CONCAT(x) PRAGMA(concat on #x)
#define PRAGMA(x) _Pragma(#x)
CONCAT( ..\concat.dir )
```

这里，CONCAT(..\concat.dir) 最终会产生 _Pragma(concat on "..\concat.dir") 这样的效果（这里只是显示语法效果，应该没有编译器支持这样的 _Pragma 语法）。而 #pragma 则不能在宏中展开，因此从灵活性上来讲，C++11 的 _Pragma 具有更大的灵活性。

2.1.4 变长参数的宏定义以及 __VA_ARGS__

在 C99 标准中，程序员可以使用变长参数的宏定义。变长参数的宏定义是指在宏定义中参数列表的最后一个参数为省略号，而预定义宏 __VA_ARGS__ 则可以在宏定义的实现部分

替换省略号所代表的字符串。比如：

```
#define PR(...) printf(__VA_ARGS__)
```

就可以定义一个 printf 的别名 PR。事实上，变长参数宏与 printf 是一对好搭档。我们可以看如代码清单 2-4 所示的一个简单的变长参数宏的应用。

代码清单 2-4

```
#include <stdio.h>

#define LOG(...)  {\
    fprintf(stderr,"%s: Line %d:\t", __FILE__, __LINE__);\
    fprintf(stderr, __VA_ARGS__);\
    fprintf(stderr,"\n");\
}

int main() {
    int x = 3;
    // 一些代码 ...
    LOG("x = %d", x); // 2-1-5.cpp: Line 12:     x = 3
}
// 编译选项 :g++ -std=c++11 2-1-5.cpp
```

在代码清单 2-4 中，定义 LOG 宏用于记录代码位置中一些信息。程序员可以根据 stderr 产生的日志追溯到代码中产生这些记录的位置。引入这样的特性，对于轻量级调试，简单的错误输出都是具有积极意义的。

2.1.5 宽窄字符串的连接

在之前的 C++ 标准中，将窄字符串（char）转换成宽字符串（wchar_t）是未定义的行为。而在 C++11 标准中，在将窄字符串和宽字符串进行连接时，支持 C++11 标准的编译器会将窄字符串转换成宽字符串，然后再与宽字符串进行连接。

事实上，在 C++11 中，我们还定义了更多种类的字符串类型（主要是为了更好地支持 Unicode），更多详细的内容，读者可以参见 8.3 与 8.4 节。

2.2 long long 整型

☞类别：部分人

相比于 C++98 标准，C++11 整型的最大改变就是多了 long long。但事实上，long long 整型本来就离 C++ 标准很近，早在 1995 年，long long 就被提议写入 C++98 标准，却被 C++ 标准委员会拒绝了。而后来，long long 类型却进入了 C99 标准，而且也事实上也被很多编译器支持。于是辗转地，C++ 标准委员会又掉头决定将 long long 纳入 C++11 标准。

long long 整型有两种：long long 和 unsigned long long。在 C++11 中，标准要求 long long 整型可以在不同平台上有不同的长度，但至少有 64 位。我们在写常数字面量时，可以使用 LL 后缀（或是 ll）标识一个 long long 类型的字面量，而 ULL（或 ull、Ull、uLL）表示一个 unsigned long long 类型的字面量。比如：

```
long long int lli = -9000000000000000000LL;
unsigned long long int ulli = -9000000000000000000ULL;
```

就定义了一个有符号的 long long 变量 lli 和无符号的 unsigned long long 变量 ulli。事实上，在 C++11 中，还有很多与 long long 等价的类型。比如对于有符号的，下面的类型是等价的：long long、signed long long、long long int、signed long long int；而 unsigned long long 和 unsigned long long int 也是等价的。

同其他的整型一样，要了解平台上 long long 大小的方法就是查看 <climits>（或 <limits.h> 中的宏）。与 long long 整型相关的一共有 3 个：LLONG_MIN、LLONG_MAX 和 ULLONG_MIN，它们分别代表了平台上最小的 long long 值、最大的 long long 值，以及最大的 unsigned long long 值。可以看看下面这个例子，如代码清单 2-5 所示。

代码清单 2-5

```
#include <climits>
#include <cstdio>
using namespace std;

int main() {
    long long ll_min = LLONG_MIN;
    long long ll_max = LLONG_MAX;
    unsigned long long ull_max = ULLONG_MAX;

    printf("min of long long: %lld\n", ll_min); // min of long long:
    -9223372036854775808
    printf("max of long long: %lld\n", ll_max); // max of long long:
    9223372036854775807
    printf("max of unsigned long long: %llu\n", ull_max);    // max of unsigned
    long long: 18446744073709551615
}
// 编译选项 :g++ -std=c++11 2-2-1.cpp
```

在代码清单 2-5 中，将以上 3 个宏打印了出来，对于 printf 函数来说，输出有符号的 long long 类型变量可以用符号 %lld，而无符号的 unsigned long long 则可以采用 %llu。18446744073709551615 用 16 进制表示是 0xFFFFFFFFFFFFFFFF（16 个 F），可知在我们的实验机上，long long 是一个 64 位的类型。

2.3 扩展的整型

☞ **类别：部分人**

程序员常会在代码中发现一些整型的名字，比如 UINT、__int16、u64、int64_t，等等。这些类型有的源自编译器的自行扩展，有的则是来自某些编程环境（比如工作在 Linux 内核代码中），不一而足。而事实上，在 C++11 中一共只定义了以下 5 种标准的有符号整型：

- ❑signed char
- ❑short int
- ❑int
- ❑long int
- ❑long long int

标准同时规定，每一种有符号整型都有一种对应的无符号整数版本，且有符号整型与其对应的无符号整型具有相同的存储空间大小。比如与 signed int 对应的无符号版本的整型是 unsigned int。

在实际的编程中，由于这 5 种基本的整型适用性有限，所以有时编译器出于需要，也会自行扩展一些整型。在 C++11 中，标准对这样的扩展做出了一些规定。具体地讲，除了标准整型（standard integer type）之外，C++11 标准允许编译器扩展自有的所谓扩展整型（extended integer type）。这些扩展整型的长度（占用内存的位数）可以比最长的标准整型（long long int，通常是一个 64 位长度的数据）还长，也可以介于两个标准整数的位数之间。比如在 128 位的架构上，编译器可以定义一个扩展整型来对应 128 位的的整数；而在一些嵌入式平台上，也可能需要扩展出 48 位的整型；不过 C++11 标准并没有对扩展出的类型的名称有任何的规定或建议，只是对扩展整型的使用规则做出了一定的限制。

简单地说，C++11 规定，扩展的整型必须和标准类型一样，有符号类型和无符号类型占用同样大小的内存空间。而由于 C/C++ 是一种弱类型语言[⊖]，当运算、传参等类型不匹配的时候，整型间会发生隐式的转换，这种过程通常被称为整型的提升（Integral promotion）。比如如下表达式：

```
(int) a + (long long)b
```

通常就会导致变量 (int)a 被提升为 long long 类型后才与 (long long)b 进行运算。而无论是扩展的整型还是标准的整型，其转化的规则会由它们的“等级”（rank）决定。而通常情况，我们认为有如下原则：

- ❑长度越大的整型等级越高，比如 long long int 的等级会高于 int。

⊖ 关于 C/C++ 是强类型语言还是弱类型语言存在一些争议，请参见 http://stackoverflow.com/questions/430182/is-c-strongly-typed。

- 长度相同的情况下，标准整型的等级高于扩展类型，比如 long long int 和 _int64 如果都是 64 位长度，则 long long int 类型的等级更高。
- 相同大小的有符号类型和无符号类型的等级相同，long long int 和 unsigned long long int 的等级就相同。

而在进行隐式的整型转换的时候，一般是按照低等级整型转换为高等级整型，有符号的转换为无符号。这种规则其实跟 C++98 的整型转换规则是一致的。

在这样的规则支持下，如果编译器定义一些自有的整型，即使这样自定义的整型由于名称并没有被标准收入，因而可移植性并不能得到保证，但至少编译器开发者和程序员不用担心自定义的扩展整型与标准整型间在使用规则上（尤其是整型提升）存在着不同的认识了。

比如在一个 128 位的构架上，编译器可以定义 _int128_t 为 128 位的有符号整型（对应的无符号类型为 _uint128_t）。于是程序员可以使用 _int128_t 类型保存形如 +92233720368547758070 的超长整数（长于 64 位的自然数）。而不用查看编译器文档我们也会知道，一旦遇到整型提升，按照上面的规则，比如 _int128_t a，与任何短于它的类型的数据 b 进行运算（比如加法）时，都会导致 b 被隐式地转换为 _int128_t 的整型，因为扩展的整型必须遵守 C++11 的规范。

2.4 宏 __cplusplus

☞ 类别：部分人

在 C 与 C++ 混合编写的代码中，我们常常会在头文件里看到如下的声明：

```
#ifdef __cplusplus
extern "C" {
#endif
// 一些代码
#ifdef __cplusplus
}
#endif
```

这种类型的头文件可以被 #include 到 C 文件中进行编译，也可以被 #include 到 C++ 文件中进行编译。由于 extern "C" 可以抑制 C++ 对函数名、变量名等符号（symbol）进行名称重整（name mangling），因此编译出的 C 目标文件和 C++ 目标文件中的变量、函数名称等符号都是相同的（否则不相同），链接器可以可靠地对两种类型的目标文件进行链接。这样该做法成为了 C 与 C++ 混用头文件的典型做法。

鉴于以上的做法，程序员可能认为 __cplusplus 这个宏只有“被定义了”和“未定义”两种状态。事实上却并非如此，__cplusplus 这个宏通常被定义为一个整型值。而且随着标准变化，__cplusplus 宏一般会是一个比以往标准中更大的值。比如在 C++03 标准中，__cplusplus

的值被预定为 199711L，而在 C++11 标准中，宏 __cplusplus 被预定义为 201103L。这点变化可以为代码所用。比如程序员在想确定代码是使用支持 C++11 编译器进行编译时，那么可以按下面的方法进行检测：

```
#if __cplusplus < 201103L
    #error "should use C++11 implementation"
#endif
```

这里，使用了预处理指令 #error，这使得不支持 C++11 的代码编译立即报错并终止编译。读者可以使用 C++98 编译器和 C++11 的编译器分别实验一下其效果。

2.5 静态断言

☞ **类别：库作者**

2.5.1 断言：运行时与预处理时

断言（assertion）是一种编程中常用的手段。在通常情况下，断言就是将一个返回值总是需要为真的判别式放在语句中，用于排除在设计的逻辑上不应该产生的情况。比如一个函数总需要输入在一定的范围内的参数，那么程序员就可以对该参数使用断言，以迫使在该参数发生异常的时候程序退出，从而避免程序陷入逻辑的混乱。

从一些意义上讲，断言并不是正常程序所必需的，不过对于程序调试来说，通常断言能够帮助程序开发者快速定位那些违反了某些前提条件的程序错误。在 C++ 中，标准在 <cassert> 或 <assert.h> 头文件中为程序员提供了 assert 宏，用于在运行时进行断言。我们可以看看下面这个例子，如代码清单 2-6 所示。

代码清单 2-6

```
#include <cassert>
using namespace std;
// 一个简单的堆内存数组分配函数
char * ArrayAlloc(int n) {
    assert(n > 0);  // 断言，n 必须大于 0
    return new char [n];
}

int main (){
    char* a = ArrayAlloc(0);
}
// 编译选项 :g++ 2-5-1.cpp
```

在代码清单 2-6 中，我们定义了一个 ArrayAlloc 函数，该函数的唯一功能就是在堆上分配字节长度为 n 的数组并返回。为了避免意外发生，函数 ArrayAlloc 对参数 n 进行了断言，

要求其大于0。而main函数中对ArrayAlloc的使用却没有满足这个条件，那么在运行时，我们可以看到如下结果：

```
a.out: 2-5-1.cpp:6: char* ArrayAlloc(int): Assertion `n > 0' failed.
Aborted
```

在C++中，程序员也可以定义宏NDEBUG来禁用assert宏。这对发布程序来说还是必要的。因为程序用户对程序退出总是敏感的，而且部分的程序错误也未必会导致程序全部功能失效。那么通过定义NDEBUG宏发布程序就可以尽量避免程序退出的状况。而当程序有问题时，通过没有定义宏NDEBUG的版本，程序员则可以比较容易地找到出问题的位置。事实上，assert宏在<cassert>中的实现方式类似于下列形式：

```
#ifdef  NDEBUG
# define assert(expr)            (static_cast<void> (0))
#else
...
#endif
```

可以看到，一旦定义了NDBUG宏，assert宏将被展开为一条无意义的C语句（通常会被编译器优化掉）。

在2.4节中，我们还看到了#error这样的预处理指令，而事实上，通过预处理指令#if和#error的配合，也可以让程序员在预处理阶段进行断言。这样的用法也是极为常见的，比如GNU的cmathcalls.h头文件中（在我们实验机上，该文件位于/usr/include/bits/cmathcalls.h），我们会看到如下代码：

```
#ifndef _COMPLEX_H
#error "Never use <bits/cmathcalls.h> directly; include <complex.h> instead."
#endif
```

如果程序员直接包含头文件<bits/cmathcalls.h>并进行编译，就会引发错误。#error指令会将后面的语句输出，从而提醒用户不要直接使用这个头文件，而应该包含头文件<complex.h>。这样一来，通过预处理时的断言，库发布者就可以避免一些头文件的引用问题。

2.5.2 静态断言与static_assert

通过2.5.1节的例子可以看到，断言assert宏只有在程序运行时才能起作用。而#error只在编译器预处理时才能起作用。有的时候，我们希望在编译时能做一些断言。比如下面这个例子，如代码清单2-7所示。

代码清单2-7

```
#include <cassert>
using namespace std;
```

```
// 枚举编译器对各种特性的支持，每个枚举值占一位
enum FeatureSupports {
    C99          =   0x0001,
    ExtInt       =   0x0002,
    SAssert      =   0x0004,
    NoExcept     =   0x0008,
    SMAX         =   0x0010,
};

// 一个编译器类型，包括名称、特性支持等
struct Compiler{
    const char * name;
    int spp;     // 使用 FeatureSupports 枚举
};

int main() {
    // 检查枚举值是否完备
    assert((SMAX - 1) == (C99 | ExtInt | SAssert | NoExcept));

    Compiler a = {"abc", (C99 | SAssert)};
    // ...
    if (a.spp & C99) {
        // 一些代码 ...
    }
}
// 编译选项 :g++ 2-5-2.cpp
```

代码清单 2-7 所示的是 C 代码中常见的“按位存储属性”的例子。在该例中，我们编写了一个枚举类型 FeatureSupports，用于列举编译器对各种特性的支持。而结构体 Compiler 则包含了一个 int 类型成员 spp。由于各种特性都具有“支持”和“不支持”两种状态，所以为了节省存储空间，我们让每个 FeatureSupports 的枚举值占据一个特定的比特位置，并在使用时通过“或”运算压缩地存储在 Compiler 的 spp 成员中（即 bitset 的概念）。在使用时，则可以通过检查 spp 的某位来判断编译器对特性是否支持。

有的时候这样的枚举值会非常多，而且还会在代码维护中不断增加。那么代码编写者必须想出办法来对这些枚举进行校验，比如查验一下是否有重位等。在本例中程序员的做法是使用一个“最大枚举”SMAX，并通过比较 SMAX - 1 与所有其他枚举的或运算值来验证是否有枚举值重位。可以想象，如果 SAssert 被误定义为 0x0001，表达式 (SMAX - 1) == (C99 | ExtInt | SAssert | NoExcept) 将不再成立。

在本例中我们使用了断言 assert。但 assert 是一个运行时的断言，这意味着不运行程序我们将无法得知是否有枚举重位。在一些情况下，这是不可接受的，因为可能单次运行代码并不会调用到 assert 相关的代码路径。因此这样的校验最好是在编译时期就能完成。

在一些 C++ 的模板的编写中，我们可能也会遇到相同的情况，比如下面这个例子，如代

码清单 2-8 所示。

代码清单 2-8

```
#include <cassert>
#include <cstring>
using namespace std;

template <typename T, typename U> int bit_copy(T& a, U& b){
    assert(sizeof(b) == sizeof(a));
    memcpy(&a,&b,sizeof(b));
};

int main() {
    int a = 0x2468;
    double b;
    bit_copy(a, b);
}
// 编译选项 :g++ 2-5-3.cpp
```

代码清单 2-8 中的 assert 是要保证 a 和 b 两种类型的长度一致，这样 bit_copy 才能够保证复制操作不会遇到越界等问题。这里我们还是使用 assert 的这样的运行时断言，但如果 bit_copy 不被调用，我们将无法触发该断言。实际上，正确产生断言的时机应该在模板实例化时，即编译时期。

代码清单 2-7 和代码清单 2-8 这类问题的解决方案是进行编译时期的断言，即所谓的“静态断言”。事实上，利用语言规则实现静态断言的讨论非常多，比较典型的实现是开源库 Boost 内置的 BOOST_STATIC_ASSERT 断言机制（利用 sizeof 操作符）。我们可以利用“除 0”会导致编译器报错这个特性来实现静态断言。

```
#define assert_static(e) \
    do { \
        enum { assert_static__ = 1/(e) }; \
    } while (0)
```

在理解这段代码时，读者可以忽略 do while 循环以及 enum 这些语法上的技巧。真正起作用的只是 1/(e) 这个表达式。把它应用到代码清单 2-8 中，就会得到代码清单 2-9。

代码清单 2-9

```
#include <cstring>
using namespace std;

#define assert_static(e) \
    do { \
        enum { assert_static__ = 1/(e) }; \
    } while (0)
```

```
template <typename T, typename U> int bit_copy(T& a, U& b){
    assert_static(sizeof(b) == sizeof(a));
    memcpy(&a,&b,sizeof(b));
};

int main() {
    int a = 0x2468;
    double b;
    bit_copy(a, b);
}
// 编译选项 :g++ -std=c++11 2-5-4.cpp
```

结果如我们预期的，在模板实例化时我们会得到编译器的错误报告，读者可以实验一下在自己本机运行的结果。在我们的实验机上会输出比较长的错误信息，主要信息是除零错误。当然，读者也可以尝试一下 Boost 库内置的 BOOST_STATIC_ASSERT，输出的主要信息是 sizeof 错误。但无论是哪种方式的静态断言，其缺陷都是很明显的：诊断信息不够充分，不熟悉该静态断言实现的程序员可能一时无法将错误对应到断言错误上，从而难以准确定位错误的根源。

在 C++11 标准中，引入了 static_assert 断言来解决这个问题。static_assert 使用起来非常简单，它接收两个参数，一个是断言表达式，这个表达式通常需要返回一个 bool 值；一个则是警告信息，它通常也就是一段字符串。我们可以用 static_assert 替换一下代码清单 2-9 中 bit_copy 的声明。

```
template <typename t, typename u> int bit_copy(t& a, u& b){
    static_assert(sizeof(b) == sizeof(a),"the parameters of bit_copy must have
    same width.");
};
```

那么再次编译代码清单 2-9 的时候，我们就会得到如下信息：

```
error: static assertion failed: "the parameters of bit_copy should have same width."
```

这样的错误信息就非常清楚，也非常有利于程序员排错。而由于 static_assert 是编译时期的断言，其使用范围不像 assert 一样受到限制。在通常情况下，static_assert 可以用于任何名字空间，如代码清单 2-10 所示。

代码清单 2-10

```
static_assert(sizeof(int) == 8, "This 64-bit machine should follow this!");
int main() { return 0; }
// 编译选项 :g++ -std=c++11 2-5-5.cpp
```

而在 C++ 中，函数则不可能像代码清单 2-10 中的 static_assert 这样独立于任何调用之外运行。因此将 static_assert 写在函数体外通常是较好的选择，这让代码阅读者可以较容易发

现 static_assert 为断言而非用户定义的函数。而反过来讲，必须注意的是，static_assert 的断言表达式的结果必须是在编译时期可以计算的表达式，即必须是常量表达式。如果读者使用了变量，则会导致错误，如代码清单 2-11 所示。

代码清单 2-11

```
int positive(const int n) {
    static_assert(n > 0, "value must >0");
}
// 编译选项 :g++ -std=c++11 -c 2-5-6.cpp
```

代码清单 2-11 使用了参数变量 n（虽然是个 const 参数），因而 static_assert 无法通过编译。对于此例，如果程序员需要的只是运行时的检查，那么还是应该使用 assert 宏。

2.6 noexcept 修饰符与 noexcept 操作符

☞ 类别：库作者

相比于断言适用于排除逻辑上不可能存在的状态，异常通常是用于逻辑上可能发生的错误。在 C++98 中，我们看到了一套完整的不同于 C 的异常处理系统。通过这套异常处理系统，C++ 拥有了远比 C 强大的异常处理功能。

在异常处理的代码中，程序员有可能看到过如下的异常声明表达形式：

```
void excpt_func() throw(int, double) { ... }
```

在 excpt_func 函数声明之后，我们定义了一个动态异常声明 throw(int, double)，该声明指出了 excpt_func 可能抛出的异常的类型。事实上，该特性很少被使用，因此在 C++11 中被弃用了（参见附录 B），而表示函数不会抛出异常的动态异常声明 throw() 也被新的 noexcept 异常声明所取代。

noexcept 形如其名地，表示其修饰的函数不会抛出异常。不过与 throw() 动态异常声明不同的是，在 C++11 中如果 noexcept 修饰的函数抛出了异常，编译器可以选择直接调用 std::terminate() 函数来终止程序的运行，这比基于异常机制的 throw() 在效率上会高一些。这是因为异常机制会带来一些额外开销，比如函数抛出异常，会导致函数栈被依次地展开（unwind），并依帧调用在本帧中已构造的自动变量的析构函数等。

从语法上讲，noexcept 修饰符有两种形式，一种就是简单地在函数声明后加上 noexcept 关键字。比如：

```
void excpt_func() noexcept;
```

另外一种则可以接受一个常量表达式作为参数，如下所示：

```
void excpt_func() noexcept （常量表达式）;
```

常量表达式的结果会被转换成一个bool类型的值。该值为true，表示函数不会抛出异常，反之，则有可能抛出异常。这里，不带常量表达式的noexcept相当于声明了noexcept(true)，即不会抛出异常。

在通常情况下，在C++11中使用noexcept可以有效地阻止异常的传播与扩散。我们可以看看下面这个例子，如代码清单2-12所示。

代码清单2-12

```
#include <iostream>
using namespace std;
void Throw() { throw 1; }
void NoBlockThrow() { Throw(); }
void BlockThrow() noexcept { Throw(); }

int main() {
    try {
        Throw();
    }
    catch(...) {
        cout << "Found throw." << endl;      // Found throw.
    }

    try {
        NoBlockThrow();
    }
    catch(...) {
        cout << "Throw is not blocked." << endl;     // Throw is not blocked.
    }

    try {
        BlockThrow();    // terminate called after throwing an instance of 'int'
    }
    catch(...) {
        cout << "Found throw 1." << endl;
    }
}
// 编译选项 :g++ -std=c++11 2-6-1.cpp
```

在代码清单2-12中，我们定义了Throw函数，该函数的唯一作用是抛出一个异常。而NoBlockThrow是一个调用Throw的普通函数，BlockThrow则是一个noexcept修饰的函数。从main的运行中我们可以看到，NoBlockThrow会让Throw函数抛出的异常继续抛出，直到main中的catch语句将其捕捉。而BlockThrow则会直接调用std::terminate中断程序的执行，从而阻止了异常的继续传播。从使用效果上看，这与C++98中的throw()是一样的。

而noexcept作为一个操作符时，通常可以用于模板。比如：

```
template <class T>
  void fun() noexcept(noexcept(T())) {}
```

这里，fun 函数是否是一个 noexcept 的函数，将由 T() 表达式是否会抛出异常所决定。这里的第二个 noexcept 就是一个 noexcept 操作符。当其参数是一个有可能抛出异常的表达式的时候，其返回值为 false，反之为 true（实际 noexcept 参数返回 false 还包括一些情况，这里就不展开讲了）。这样一来，我们就可以使模板函数根据条件实现 noexcept 修饰的版本或无 noexcept 修饰的版本。从泛型编程的角度看来，这样的设计保证了关于“函数是否抛出异常”这样的问题可以通过表达式进行推导。因此这也可以视作 C++11 为了更好地支持泛型编程而引入的特性。

虽然 noexcept 修饰的函数通过 std::terminate 的调用来结束程序的执行的方式可能会带来很多问题，比如无法保证对象的析构函数的正常调用，无法保证栈的自动释放等，但很多时候，“暴力”地终止整个程序确实是很简单有效的做法。事实上，noexcept 被广泛地、系统地应用在 C++11 的标准库中，用于提高标准库的性能，以及满足一些阻止异常扩散的需求。

比如在 C++98 中，存在着使用 throw() 来声明不抛出异常的函数。

```
template<class T> class A {
  public:
    static constexpr T min() throw() { return T(); }
    static constexpr T max() throw() { return T(); }
    static constexpr T lowest() throw() { return T(); }
...
```

而在 C++11 中，则使用 noexcept 来替换 throw()。

```
template<class T> class A {
  public:
    static constexpr T min() noexcept { return T(); }
    static constexpr T max() noexcept  { return T(); }
    static constexpr T lowest() noexcept { return T(); }
...
```

又比如，在 C++98 中，new 可能会包含一些抛出的 std::bad_alloc 异常。

```
void* operator new(std::size_t) throw(std::bad_alloc);
void* operator new[](std::size_t) throw(std::bad_alloc);
```

而在 C++11 中，则使用 noexcept(false) 来进行替代。

```
void* operator new(std::size_t) noexcept(false);
void* operator new[](std::size_t) noexcept(false);
```

当然，noexcept 更大的作用是保证应用程序的安全。比如一个类析构函数不应该抛出异常，那么对于常被析构函数调用的 delete 函数来说，C++11 默认将 delete 函数设置成 noexcept，就可以提高应用程序的安全性。

```
void operator delete(void*) noexcept;
void operator delete[](void*) noexcept;
```

而同样出于安全考虑，C++11 标准中让类的析构函数默认也是 noexcept(true) 的。当然，如果程序员显式地为析构函数指定了 noexcept，或者类的基类或成员有 noexcept(false) 的析构函数，析构函数就不会再保持默认值。我们可以看看下面的例子，如代码清单 2-13 所示。

代码清单 2-13

```
#include <iostream>
using namespace std;

struct A {
    ~A() { throw 1; }
};

struct B {
    ~B() noexcept(false) { throw 2; }
};

struct C {
    B b;
};

int funA() { A a; }
int funB() { B b; }
int funC() { C c; }

int main() {
    try {
        funB();
    }
    catch(...){
        cout << "caught funB." << endl; // caught funB.
    }

    try {
        funC();
    }
    catch(...){
        cout << "caught funC." << endl; // caught funC.
    }

    try {
        funA(); // terminate called after throwing an instance of 'int'
    }
    catch(...){
        cout << "caught funA." << endl;
    }
}
// 编译选项 :g++ -std=c++11 2-6-2.cpp
```

在代码清单 2-13 中，无论是析构函数声明为 noexcept(false) 的类 B，还是包含了 B 类型成员的类 C，其析构函数都是可以抛出异常的。只有什么都没有声明的类 A，其析构函数被默认为 noexcept(true)，从而阻止了异常的扩散。这在实际的使用中，应该引起程序员的注意。

2.7 快速初始化成员变量

☞ **类别：部分人**

在 C++98 中，支持了在类声明中使用等号“=”加初始值的方式，来初始化类中静态成员常量。这种声明方式我们也称之为“就地”声明。就地声明在代码编写时非常便利，不过 C++98 对类中就地声明的要求却非常高。如果静态成员不满足常量性，则不可以就地声明，而且即使常量的静态成员也只能是整型或者枚举型才能就地初始化。而非静态成员变量的初始化则必须在构造函数中进行。我们来看看下面的例子，如代码清单 2-14 所示。

代码清单 2-14

```
class Init{
public:
    Init(): a(0){}
    Init(int d): a(d){}
private:
    int a;
    const static int b = 0;
    int c = 1;           // 成员，无法通过编译
    static int d = 0;    // 成员，无法通过编译
    static const double e = 1.3;         // 非整型或者枚举，无法通过编译
    static const char * const f = "e";   // 非整型或者枚举，无法通过编译
};
// 编译选项 :g++ -c 2-7-1.cpp
```

在代码清单 2-14 中，成员 c、静态成员 d、静态常量成员 e 以及静态常量指针 f 的就地初始化都无法通过编译（这里，使用 g++ 的读者可能发现就地初始化 double 类型静态常量 e 是可以通过编译的，不过这实际是 GNU 对 C++ 的一个扩展，并不遵从 C++ 标准）。在 C++11 中，标准允许非静态成员变量的初始化有多种形式。具体而言，除了初始化列表外，在 C++11 中，标准还允许使用等号 = 或者花括号 {} 进行就地的非静态成员变量初始化。比如：

```
struct init{ int a = 1; double b {1.2}; };
```

在这个名叫 init 的结构体中，我们给了非静态成员 a 和 b 分别赋予初值 1 和 1.2。这在 C++11 中是一个合法的结构体声明。虽然这里采用的一对花括号 {} 的初始化方法读者第一次见到，不过在第 3 章中，读者会在 C++ 对于初始化表达式的改动发现，花括号式的集合

（列表）初始化已经成为 C++11 中初始化声明的一种通用形式，而其效果类似于 C++98 中使用圆括号 () 对自定义变量的表达式列表初始化。不过在 C++11 中，对于非静态成员进行就地初始化，两者却并非等价的，如代码清单 2-15 所示。

代码清单 2-15

```
#include <string>
using namespace std;

struct C {
    C(int i):c(i){};
    int c;
};

struct init {
    int a = 1;
    string b("hello");  // 无法通过编译
    C c(1);             // 无法通过编译
};
// 编译选项 :g++ -std=c++11 -c 2-7-2.cpp
```

从代码清单 2-15 中可以看到，就地圆括号式的表达式列表初始化非静态成员 b 和 c 都会导致编译出错。

在 C++11 标准支持了就地初始化非静态成员的同时，初始化列表这个手段也被保留下来了。如果两者都使用，是否会发生冲突呢？我们来看下面这个例子，如代码清单 2-16 所示。

代码清单 2-16

```
#include <iostream>
using namespace std;

struct Mem {
    Mem() { cout << "Mem default, num: " << num << endl; }
    Mem(int i): num(i) { cout << "Mem, num: " << num << endl; }

    int num = 2;  // 使用 = 初始化非静态成员
};

class Group {
public:
    Group() { cout << "Group default. val: " << val << endl; }
    Group(int i): val('G'), a(i) { cout << "Group. val: " << val << endl; }
    void NumOfA() { cout << "number of A: " << a.num << endl; }
    void NumOfB() { cout << "number of B: " << b.num << endl; }

private:
    char val{'g'};  // 使用 {} 初始化非静态成员
```

```
    Mem a;
    Mem b{19};        // 使用{}初始化非静态成员
};

int main() {
    Mem member;           // Mem default, num: 2

    Group group;          // Mem default, num: 2
                          // Mem, num: 19
                          // Group default. val: g

    group.NumOfA();       // number of A: 2
    group.NumOfB();       // number of B: 19

    Group group2(7);      // Mem, num: 7
                          // Mem, num: 19
                          // Group. val: G

    group2.NumOfA();      // number of A: 7
    group2.NumOfB();      // number of B: 19
}
// 编译选项 :g++ 2-7-3.cpp -std=c++11
```

在代码清单2-16中，我们定义了有两个初始化函数的类Mem，此外还定义了包含两个Mem对象的Group类。类Mem中的成员变量num，以及class Group中的成员变量a、b、val，采用了与C++98完全不同的初始化方式。读者可以从main函数的打印输出中看到，就地初始化和初始化列表并不冲突。程序员可以为同一成员变量既声明就地的列表初始化，又在初始化列表中进行初始化，只不过初始化列表总是看起来“后作用于”非静态成员。也就是说，初始化列表的效果总是优先于就地初始化的。

相对于传统的初始化列表，在类声明中对非静态成员变量进行就地列表初始化可以降低程序员的工作量。当然，我们只在有多个构造函数，且有多个成员变量的时候可以看到新方式带来的便利。我们来看看下面的例子，如代码清单2-17所示。

代码清单 2-17

```
#include <string>
using namespace std;

class Mem {
public:
    Mem(int i): m(i){}

private:
    int m;
};
```

```
class Group {
public:
    Group(){}                          // 这里就不需要初始化 data、mem、name 成员了
    Group(int a): data(a) {}           // 这里就不需要初始化 mem、name 成员了
    Group(Mem m) : mem(m) {}           // 这里就不需要初始化 data、name 成员了
    Group(int a, Mem m, string n): data(a), mem(m), name(n){}

private:
    int data = 1;
    Mem mem{0};
    string name{"Group"};
};
// 编译选项 :g++ 2-7-4.cpp -std=c++11 -c
```

在代码清单 2-17 中，Group 有 4 个构造函数。可以想象，如果我们使用的是 C++98 的编译器，我们不得不在 Group()、Group(int a)，以及 Group(Mem m) 这 3 个构造函数中将 data、mem、name 这 3 个成员都写进初始化列表。但如果使用的是 C++11 的编译器，那么通过对非静态成员变量的就地初始化，我们就可以避免重复地在初始化列表中写上每个非静态成员了（在 C++98 中，我们还可以通过调用公共的初始化函数来达到类似的目的，不过目前在书写的复杂性及效率性上远低于 C++11 改进后的做法）。

此外，值得注意的是，对于非常量的静态成员变量，C++11 则与 C++98 保持了一致。程序员还是需要到头文件以外去定义它，这会保证编译时，类静态成员的定义最后只存在于一个目标文件中。不过对于静态常量成员，除了 const 关键字外，在本书第 6 章中我们会看到还可以使用 constexpr 来对静态常量成员进行声明。

2.8 非静态成员的 sizeof

☞ **类别：部分人**

从 C 语言被发明开始，sizeof 就是一个运算符，也是 C 语言中除了加减乘除以外为数不多的特殊运算符之一。而在 C++ 引入类（class）类型之后，sizeof 的定义也随之进行了拓展。不过在 C++98 标准中，对非静态成员变量使用 sizeof 是不能够通过编译的。我们可以看看下面的例子，如代码清单 2-18 所示。

代码清单 2-18

```
#include <iostream>
using namespace std;

struct People {
public:
    int hand;
    static People * all;
```

```
};

int main() {
    People p;
    cout << sizeof(p.hand) << endl;          // C++98 中通过，C++11 中通过
    cout << sizeof(People::all) << endl;     // C++98 中通过，C++11 中通过
    cout << sizeof(People::hand) << endl;    // C++98 中错误，C++11 中通过
}
// 编译选项 :g++ 2-8-1.cpp
```

注意最后一个 sizeof 操作。在 C++11 中，对非静态成员变量使用 sizeof 操作是合法的。而在 C++98 中，只有静态成员，或者对象的实例才能对其成员进行 sizeof 操作。因此如果读者只有一个支持 C++98 标准的编译器，在没有定义类实例的时候，要获得类成员的大小，我们通常会采用以下的代码：

```
sizeof(((People*)0)->hand);
```

这里我们强制转换 0 为一个 People 类的指针，继而通过指针的解引用获得其成员变量，并用 sizeof 求得该成员变量的大小。而在 C++11 中，我们无需这样的技巧，因为 sizeof 可以作用的表达式包括了类成员表达式。

```
sizeof(People::hand);
```

可以看到，无论从代码的可读性还是编写的便利性，C++11 的规则都比强制指针转换的方案更胜一筹。

2.9 扩展的 friend 语法

☞ **类别：部分人**

friend 关键字在 C++ 中是一个比较特别的存在。因为我们常常会发现，一些面向对象程序语言，比如 Java，就没有定义 friend 关键字。friend 关键字用于声明类的友元，友元可以无视类中成员的属性。无论成员是 public、protected 或是 private 的，友元类或友元函数都可以访问，这就完全破坏了面向对象编程中封装性的概念。因此，使用 friend 关键字充满了争议性。在通常情况下，面向对象程序开发的专家会建议程序员使用 Get/Set 接口来访问类的成员，但有的时候，friend 关键字确实会让程序员少写很多代码。因此即使存在争论，friend 还是在很多程序中被使用到。而 C++11 对 friend 关键字进行了一些改进，以保证其更加好用。我们可以看看下面的例子，如代码清单 2-19 所示。

代码清单 2-19

```
class Poly;
typedef Poly P;

class LiLei {
```

```
    friend class Poly;   // C++98 通过，C++11 通过
};

class Jim {
    friend Poly;         // C++98 失败，C++11 通过
};

class HanMeiMei {
    friend P;            // C++98 失败，C++11 通过
};
// 编译选项 :g++ -std=c++11 2-9-1.cpp
```

在代码清单 2-19 中，我们声明了 3 个类型：LiLei、Jim 和 HanMeiMei，它们都有一个友元类型 Poly。从编译通过与否的状况中我们可以看出，在 C++11 中，声明一个类为另外一个类的友元时，不再需要使用 class 关键字。本例中的 Jim 和 HanMeiMei 就是这样一种情况，在 HanMeiMei 的声明中，我们甚至还使用了 Poly 的别名 P，这同样是可行的。

虽然在 C++11 中这是一个小的改进，却会带来一点应用的变化——程序员可以为类模板声明友元了。这在 C++98 中是无法做到的。比如下面这个例子，如代码清单 2-20 所示。

代码清单 2-20

```
class P;
template <typename T> class People {
    friend T;
};

People<P> PP;    // 类型 P 在这里是 People 类型的友元
People<int> Pi; // 对于 int 类型模板参数，友元声明被忽略
// 编译选项 :g++ -std=c++11 2-9-2.cpp
```

从代码清单 2-20 中我们看到，对于 People 这个模板类，在使用类 P 为模板参数时，P 是 People<P> 的一个 friend 类。而在使用内置类型 int 作为模板参数的时候，People<int> 会被实例化为一个普通的没有友元定义的类型。这样一来，我们就可以在模板实例化时才确定一个模板类是否有友元，以及谁是这个模板类的友元。这是一个非常有趣的小特性，在编写一些测试用例的时候，使用该特性是很有好处的。我们看看下面的例子，该例子源自一个实际的测试用例，如代码清单 2-21 所示。

代码清单 2-21

```
// 为了方便测试，进行了危险的定义
#ifdef UNIT_TEST
#define private public
#endif
class Defender {
```

```
public:
    void Defence(int x, int y){}
    void Tackle(int x, int y){}

private:
    int pos_x = 15;
    int pos_y = 0;
    int speed = 2;
    int stamina = 120;
};

class Attacker {
public:
    void Move(int x, int y){}
    void SpeedUp(float ratio){}

private:
    int pos_x = 0;
    int pos_y = -30;
    int speed = 3;
    int stamina = 100;
};

#ifdef UNIT_TEST
class Validator {
public:
    void Validate(int x, int y, Defender & d){}
    void Validate(int x, int y, Attacker & a){}
};

int main() {
    Defender d;
    Attacker a;
    a.Move(15, 30);
    d.Defence(15, 30);
    a.SpeedUp(1.5f);
    d.Defence(15, 30);
    Validator v;
    v.Validate(7, 0, d);
    v.Validate(1, -10, a);
    return 0;
}
#endif
// 编译选项 :g++ 2-9-3.cpp -std=c++11 -DUNIT_TEST
```

在代码清单 2-21 所示的这个例子中，测试人员的目的是在一系列函数调用后，检查 Attacker 类变量 a 和 Defender 类变量 d 中成员变量的值是否符合预期。这里，按照封装的思想，所有成员变量被声明为 private 的。但 Attacker 和 Defender 的开发者为了方便，并没有为每个成员写 Get 函数，也没有为 Attacker 和 Defender 增加友元定义。而测试人员为了能够

快速写出测试程序，采用了比较危险的做法，即使用宏将 private 关键字统一替换为 public 关键字。这样一来，类中的 private 成员就都成了 public 的。这样的做法存在 4 个缺点：一是如果侥幸程序中没有变量包含 private 字符串，该方法可以正常工作，但反之，则有可能导致严重的编译时错误；二是这种做法会降低代码可读性，因为改变了一个常见关键字的意义，没有注意到这个宏的程序员可能会非常迷惑程序的行为；三是如果在类的成员定义时不指定关键字（如 public、private、protect 等），而使用默认的 private 成员访问限制，那么该方法也不能达到目的；四则很简单，这样的做法看起来也并不漂亮。

不过由于有了扩展的 friend 声明，在 C++11 中，我们可以将 Defender 和 Attacker 类改良一下。我们看看下面的例子，如代码清单 2-22 所示。

代码清单 2-22

```
template <typename T> class DefenderT {
public:
    friend T;
    void Defence(int x, int y){}
    void Tackle(int x, int y){}

private:
    int pos_x = 15;
    int pos_y = 0;
    int speed = 2;
    int stamina = 120;
};

template <typename T> class AttackerT {
public:
    friend T;
    void Move(int x, int y){}
    void SpeedUp(float ratio){}

private:
    int pos_x = 0;
    int pos_y = -30;
    int speed = 3;
    int stamina = 100;
};

using Defender = DefenderT<int>;     // 普通的类定义，使用 int 做参数
using Attacker = AttackerT<int>;

#ifdef UNIT_TEST
class Validator {
public:
    void Validate(int x, int y, DefenderTest & d){}
    void Validate(int x, int y, AttackerTest & a){}
```

```
};

using DefenderTest = DefenderT<Validator>;   // 测试专用的定义，Validator 类成为友元
using AttackerTest = AttackerT<Validator>;

int main() {
    DefenderTest d;
    AttackerTest a;
    a.Move(15, 30);
    d.Defence(15, 30);
    a.SpeedUp(1.5f);
    d.Defence(15, 30);
    Validator v;
    v.Validate(7, 0, d);
    v.Validate(1, -10, a);
    return 0;
}
#endif
// 编译选项 :g++ 2-9-4.cpp -std=c++11 -DUNIT_TEST
```

在代码清单 2-22 中，我们把原有的 Defender 和 Attacker 类定义为模板类 DefenderT 和 AttackerT。而在需要进行测试的时候，我们使用 Validator 为模板参数，实例化出 DefenderTest 及 AttackerTest 版本的类，这个版本的特点是，Validator 是它们的友元，可以任意访问任何成员函数。而另外一个版本则是使用 int 类型进行实例化的 Defender 和 Attacker，按照 C++11 的定义，它们不会有友元。因此这个版本保持了良好的封装性，可以用于提供接口用于常规使用。

值得注意的是，在代码清单 2-22 中，我们使用了 using 来定义类型的别名，这跟使用 typedef 的定义类型的别名是完全一样的。使用 using 定义类型别名是 C++11 中的一个新特性，我们可以在 3.10 节中看到相关的描述。

2.10 final/override 控制

☞ **类别：部分人**

在了解 C++11 中的 final/override 关键字之前，我们先回顾一下 C++ 关于重载的概念。简单地说，一个类 A 中声明的虚函数 fun 在其派生类 B 中再次被定义，且 B 中的函数 fun 跟 A 中 fun 的原型一样（函数名、参数列表等一样），那么我们就称 B 重载（overload）了 A 的 fun 函数。对于任何 B 类型的变量，调用成员函数 fun 都是调用了 B 重载的版本。而如果同时有 A 的派生类 C，却并没有重载 A 的 fun 函数，那么调用成员函数 fun 则会调用 A 中的版本。这在 C++ 中就实现多态。

在通常情况下，一旦在基类 A 中的成员函数 fun 被声明为 virtual 的，那么对于其派生类

B 而言，fun 总是能够被重载的（除非被重写了）。有的时候我们并不想 fun 在 B 类型派生类中被重载，那么，C++98 没有方法对此进行限制。我们看看下面这个具体的例子，如代码清单 2-23 所示。

代码清单 2-23

```
#include <iostream>
using namespace std;

class MathObject{
public:
    virtual double Arith() = 0;
    virtual void Print() = 0;
};

class Printable : public MathObject{
public:
   double Arith() = 0;
   void Print() // 在 C++98 中我们无法阻止该接口被重写
   {
       cout << "Output is: " << Arith() << endl;
   }
};

class Add2 : public Printable {
public:
    Add2(double a, double b): x(a), y(b) {}
    double Arith() { return x + y; }
private:
    double x, y;
};

class Mul3 : public Printable {
public:
    Mul3(double a, double b, double c): x(a), y(b), z(c) {}
    double Arith() { return x * y * z; }
private:
    double x, y, z;
};
// 编译选项 :g++ 2-10-1.cpp
```

在代码清单 2-23 中，我们的基础类 MathObject 定义了两个接口：Arith 和 Print。类 Printable 则继承于 MathObject 并实现了 Print 接口。接下来，Add2 和 Mul3 为了使用 MathObject 的接口和 Printable 的 Print 的实现，于是都继承了 Printable。这样的类派生结构，在面向对象的编程中非常典型。不过倘若这里的 Printable 和 Add2 是由两个程序员完成的，Printable 的编写者不禁会有一些忧虑，如果 Add2 的编写者重载了 Print 函数，那么他所期望的统一风格的打印方式将不复存在。

对于 Java 这种所有类型派生于单一元类型（Object）的语言来说，这种问题早就出现了。因此 Java 语言使用了 final 关键字来阻止函数继续重写。final 关键字的作用是使派生类不可覆盖它所修饰的虚函数。C++11 也采用了类似的做法，如代码清单 2-24 所示的例子。

代码清单 2-24

```
struct Object{
    virtual void fun() = 0;
};

struct Base : public Object {
    void fun() final;    // 声明为 final
};

struct Derived : public Base {
    void fun();      // 无法通过编译
};
// 编译选项 :g++ -c -std=c++11 2-10-2.cpp
```

在代码清单 2-24 中，派生于 Object 的 Base 类重载了 Object 的 fun 接口，并将本类中的 fun 函数声明为 final 的。那么派生于 Base 的 Derived 类对接口 fun 的重载则会导致编译时的错误。同理，在代码清单 2-23 中，Printable 的编写者如果要阻止派生类重载 Print 函数，只需要在定义时使用 final 进行修饰就可以了。

读者可能注意到了，在代码清单 2-23 及代码清单 2-24 两个例子当中，final 关键字都是用于描述一个派生类的。那么基类中的虚函数是否可以使用 final 关键字呢？答案是肯定的，不过这样将使用该虚函数无法被重载，也就失去了虚函数的意义。如果不想成员函数被重载，程序员可以直接将该成员函数定义为非虚的。而 final 通常只在继承关系的“中途”终止派生类的重载中有意义。从接口派生的角度而言，final 可以在派生过程中任意地阻止一个接口的可重载性，这就给面向对象的程序员带来了更大的控制力。

在 C++ 中重载还有一个特点，就是对于基类声明为 virtual 的函数，之后的重载版本都不需要再声明该重载函数为 virtual。即使在派生类中声明了 virtual，该关键字也是编译器可以忽略的。这带来了一些书写上的便利，却带来了一些阅读上的困难。比如代码清单 2-23 中的 Printable 的 Print 函数，程序员无法从 Printable 的定义中看出 Print 是一个虚函数还是非虚函数。另外一点就是，在 C++ 中有的虚函数会“跨层”，没有在父类中声明的接口有可能是祖先的虚函数接口。比如在代码清单 2-23 中，如果 Printable 不声明 Arith 函数，其接口在 Add2 和 Mul3 中依然是可重载的，这同样是在父类中无法读到的信息。这样一来，如果类的继承结构比较长（不断地派生）或者比较复杂（比如偶尔多重继承），派生类的编写者会遇到信息分散、难以阅读的问题（虽然有时候编辑器会进行提示，不过编辑器不是总是那么有效）。而自己是否在重载一个接口，以及自己重载的接口的名字是否有拼写错误等，都非常

不容易检查。

在 C++11 中为了帮助程序员写继承结构复杂的类型，引入了虚函数描述符 override，如果派生类在虚函数声明时使用了 override 描述符，那么该函数必须重载其基类中的同名函数，否则代码将无法通过编译。我们来看一下如代码清单 2-25 所示的这个简单的例子。

代码清单 2-25

```
struct Base {
    virtual void Turing() = 0;
    virtual void Dijkstra() = 0;
    virtual void VNeumann(int g) = 0;
    virtual void DKnuth() const;
    void Print();
};

struct DerivedMid: public Base {
    // void VNeumann(double g);
    // 接口被隔离了，曾想多一个版本的 VNeumann 函数
};

struct DerivedTop : public DerivedMid {
    void Turing() override;
    void Dikjstra() override;              // 无法通过编译，拼写错误，并非重载
    void VNeumann(double g) override;      // 无法通过编译，参数不一致，并非重载
    void DKnuth() override;                // 无法通过编译，常量性不一致，并非重载
    void Print() override;                 // 无法通过编译，非虚函数重载
};
// 编译选项 :g++ -c -std=c++11 2-10-3.cpp
```

在代码清单 2-25 中，我们在基类 Base 中定义了一些 virtual 的函数（接口）以及一个非 virtual 的函数 Print。其派生类 DerivedMid 中，基类的 Base 的接口都没有重载，不过通过注释可以发现，DerivedMid 的作者曾经想要重载出一个“void VNeumann(double g)”的版本。这行注释显然迷惑了编写 DerivedTop 的程序员，所以 DerivedTop 的作者在重载所有 Base 类的接口的时候，犯下了 3 种不同的错误：

- 函数名拼写错，Dijkstra 误写作了 Dikjstra。
- 函数原型不匹配，VNeumann 函数的参数类型误做了 double 类型，而 DKnuth 的常量性在派生类中被取消了。
- 重写了非虚函数 Print。

如果没有 override 修饰符，DerivedTop 的作者可能在编译后都没有意识到自己犯了这么多错误。因为编译器对以上 3 种错误不会有任何的警示。这里 override 修饰符则可以保证编译器辅助地做一些检查。我们可以看到，在代码清单 2-25 中，DerivedTop 作者的 4 处错误都无法通过编译。

此外，值得指出的是，在 C++ 中，如果一个派生类的编写者自认为新写了一个接口，而实际上却重载了一个底层的接口（一些简单的名字如 get、set、print 就容易出现这样的状况），出现这种情况编译器还是爱莫能助的。不过这样无意中的重载一般不会带来太大的问题，因为派生类的变量如果调用了该接口，除了可能存在的一些虚函数开销外，仍然会执行派生类的版本。因此编译器也就没有必要提供检查“非重载”的状况。而检查“一定重载”的 override 关键字，对程序员的实际应用则会更有意义。

还有值得注意的是，如我们在第 1 章中提到的，final/override 也可以定义为正常变量名，只有在其出现在函数后时才是能够控制继承 / 派生的关键字。通过这样的设计，很多含有 final/override 变量或者函数名的 C++98 代码就能够被 C++ 编译器编译通过了。但出于安全考虑，建议读者在 C++11 代码中应该尽可能地避免这样的变量名称或将其定义在宏中，以防发生不必要的错误。

2.11 模板函数的默认模板参数

☞ **类别：所有人**

在 C++11 中模板和函数一样，可以有默认的参数。这就带来了一定的复杂性。我可以通过代码清单 2-26 所示的这个简单的模板函数的例子来回顾一下函数模板的定义。

代码清单 2-26

```
#include <iostream>
using namespace std;

// 定义一个函数模板
template <typename T> void TempFun(T a) {
    cout << a << endl;
}

int main() {
    TempFun(1);      // 1, (实例化为 TempFun<const int>(1))
    TempFun("1");    // 1, (实例化为 TempFun<const char *>("1"))
}
// 编译选项 :g++ 2-11-1.cpp
```

在代码清单 2-26 中，当编译器解析到函数调用 fun(1) 的时候，发现 fun 是一个函数模板。这时候编译器就会根据实参 1 的类型 const int 推导实例化出模板函数 void TempFun<const int>(int)，再进行调用。相应的，对于 fun("1") 来说也是类似的，不过编译器实例化出的模板函数的参数的类型将是 const char *。

函数模板在 C++98 中与类模板一起被引入，不过在模板类声明的时候，标准允许其有默认模板参数。默认的模板参数的作用好比函数的默认形参。然而由于种种原因，C++98 标准

却不支持函数模板的默认模板参数。不过在 C++11 中，这一限制已经被解除了，我们可以看看下面这个例子，如代码清单 2-27 所示。

代码清单 2-27

```
void DefParm(int m = 3) {}  // c++98 编译通过，c++11 编译通过
template <typename T = int>
    class DefClass {};      // c++98 编译通过，c++11 编译通过
template <typename T = int>
    void DefTempParm() {};  // c++98 编译失败，c++11 编译通过
// 编译选项 :g++ -c -std=c++11 2-11-1.cpp
```

可以看到，DefTempParm 函数模板拥有一个默认参数。使用仅支持 C++98 的编译器编译，DefTempParm 的编译会失败，而支持 C++11 的编译器则毫无问题。不过在语法上，与类模板有些不同的是，在为多个默认模板参数声明指定默认值的时候，程序员必须遵照“从右往左”的规则进行指定。而这个条件对函数模板来说并不是必须的，如代码清单 2-28 所示。

代码清单 2-28

```
template<typename T1, typename T2 = int> class DefClass1;
template<typename T1 = int, typename T2> class DefClass2;   // 无法通过编译

template<typename T, int i = 0> class DefClass3;
template<int i = 0, typename T> class DefClass4;            // 无法通过编译

template<typename T1 = int, typename T2> void DefFunc1(T1 a, T2 b);
template<int i = 0, typename T> void DefFunc2(T a);
// 编译选项 :g++ -c -std=c++11 2-11-2.cpp
```

从代码清单 2-28 中可以看到，不按照从右往左定义默认类模板参数的模板类 DefClass2 和 DefClass4 都无法通过编译。而对于函数模板来说，默认模板参数的位置则比较随意。可以看到 DefFunc1 和 DefFunc2 都为第一个模板参数定义了默认参数，而第二个模板参数的默认值并没有定义，C++11 编译器却认为没有问题。

函数模板的参数推导规则也并不复杂。简单地讲，如果能够从函数实参中推导出类型的话，那么默认模板参数就不会被使用，反之，默认模板参数则可能会被使用。我们可以看看下面这个来自于 C++11 标准草案的例子，如代码清单 2-29 所示。

代码清单 2-29

```
template <class T, class U = double>
void f(T t = 0, U u = 0);

void g() {
```

```
    f(1, 'c');        // f<int,char>(1,'c')
    f(1);             // f<int,double>(1,0), 使用了默认模板参数 double
    f();              // 错误 : T 无法被推导出来
    f<int>();         // f<int,double>(0,0), 使用了默认模板参数 double
    f<int,char>();    // f<int,char>(0,0)
}
// 编译选项 :g++ -std=c++11 2-11-3.cpp
```

在代码清单 2-29 中，我们定义了一个函数模板 f，f 同时使用了默认模板参数和默认函数参数。可以看到，由于函数的模板参数可以由函数的实参推导而出，所以在 f(1) 这个函数调用中，我们实例化出了模板函数的调用应该为 f<int,double>(1,0)，其中，第二个类型参数 U 使用了默认的模板类型参数 double，而函数实参则为默认值 0。类似地，f<int>() 实例化出的模板函数第二参数类型为 double，值为 0。而表达式 f() 由于第一类型参数 T 的无法推导，从而导致了编译的失败。而通过这个例子我们也可以看到，默认模板参数通常是需要跟默认函数参数一起使用的。

还有一点应该强调一下，模板函数的默认形参不是模板参数推导的依据。函数模板参数的选择，总是由函数的实参推导而来的，这点读者在使用中应当注意。

2.12 外部模板

☞ **类别：部分人**

2.12.1 为什么需要外部模板

“外部模板”是 C++11 中一个关于模板性能上的改进。实际上，“外部”（extern）这个概念早在 C 的时候已经就有了。通常情况下，我们在一个文件中 a.c 中定义了一个变量 int i，而在另外一个文件 b.c 中想使用它，这个时候我们就会在没有定义变量 i 的 b.c 文件中做一个外部变量的声明。比如：

```
extern int i;
```

这样做的好处是，在分别编译了 a.c 和 b.c 之后，其生成的目标文件 a.o 和 b.o 中只有 i 这个符号[⊖]的一份定义。具体地，a.o 中的 i 是实在存在于 a.o 目标文件的数据区中的数据，而在 b.o 中，只是记录了 i 符号会引用其他目标文件中数据区中的名为 i 的数据。这样一来，在链接器（通常由编译器代为调用）将 a.o 和 b.o 链接成单个可执行文件（或者库文件）c 的时候，c 文件的数据区也只会有一个 i 的数据（供 a.c 和 b.c 的代码共享）。

而如果 b.c 中我们声明 int i 的时候不加上 extern 的话，那么 i 就会实实在在地既存在于 a.o 的数据区中，也存在于 b.o 的数据区中。那么链接器在链接 a.o 和 b.o 的时候，就会报告

⊖ 符号（symbol）是编译器 / 链接器的术语，读者可以简单地将它想象为一个变量名字。

错误，因为无法决定相同的符号是否需要合并。

而对于函数模板来说，现在我们遇到的几乎是一模一样的问题。不同的是，发生问题的不是变量（数据），而是函数（代码）。这样的困境是由于模板的实例化带来的。

注意 这里我们以函数模板为例，因为其只涉及代码，讲解起来比较直观。如果是类模板，则有可能涉及数据，不过其原理都是类似的。

比如，我们在一个 test.h 的文件中声明了如下一个模板函数：

```
template <typename T> void fun(T) {}
```

在第一个 test1.cpp 文件中，我们定义了以下代码：

```
#include "test.h"
void test1() { fun(3); }
```

而在另一个 test2.cpp 文件中，我们定义了以下代码：

```
#include "test.h"
void test2() { fun(4); }
```

由于两个源代码使用的模板函数的参数类型一致，所以在编译 test1.cpp 的时候，编译器实例化出了函数 fun<int>(int)，而当编译 test2.cpp 的时候，编译器又再一次实例化出了函数 fun<int>(int)。那么可以想象，在 test1.o 目标文件和 test2.o 目标文件中，会有两份一模一样的函数 fun<int>(int) 代码。

代码重复和数据重复不同。数据重复，编译器往往无法分辨是否是要共享的数据；而代码重复，为了节省空间，保留其中之一就可以了（只要代码完全相同）。事实上，大部分链接器也是这样做的。在链接的时候，链接器通过一些编译器辅助的手段将重复的模板函数代码 fun<int>(int) 删除掉，只保留了单个副本。这样一来，就解决了模板实例化时产生的代码冗余问题。我们可以看看图 2-1 中的模板函数的编译与链接的过程示意。

不过读者也注意到了，对于源代码中出现的每一处模板实例化，编译器都需要去做实例化的工作；而在链接时，链接器还需要移除重复的实例化代码。很明显，这样的工作太过冗余，而在广泛使用模板的项目中，由于编译器会产生大量冗余代码，会极大地增加编译器的编译时间和链接时间。解决这个问题的方法基本跟变量共享的思路是一样的，就是使用“外部的”模板。

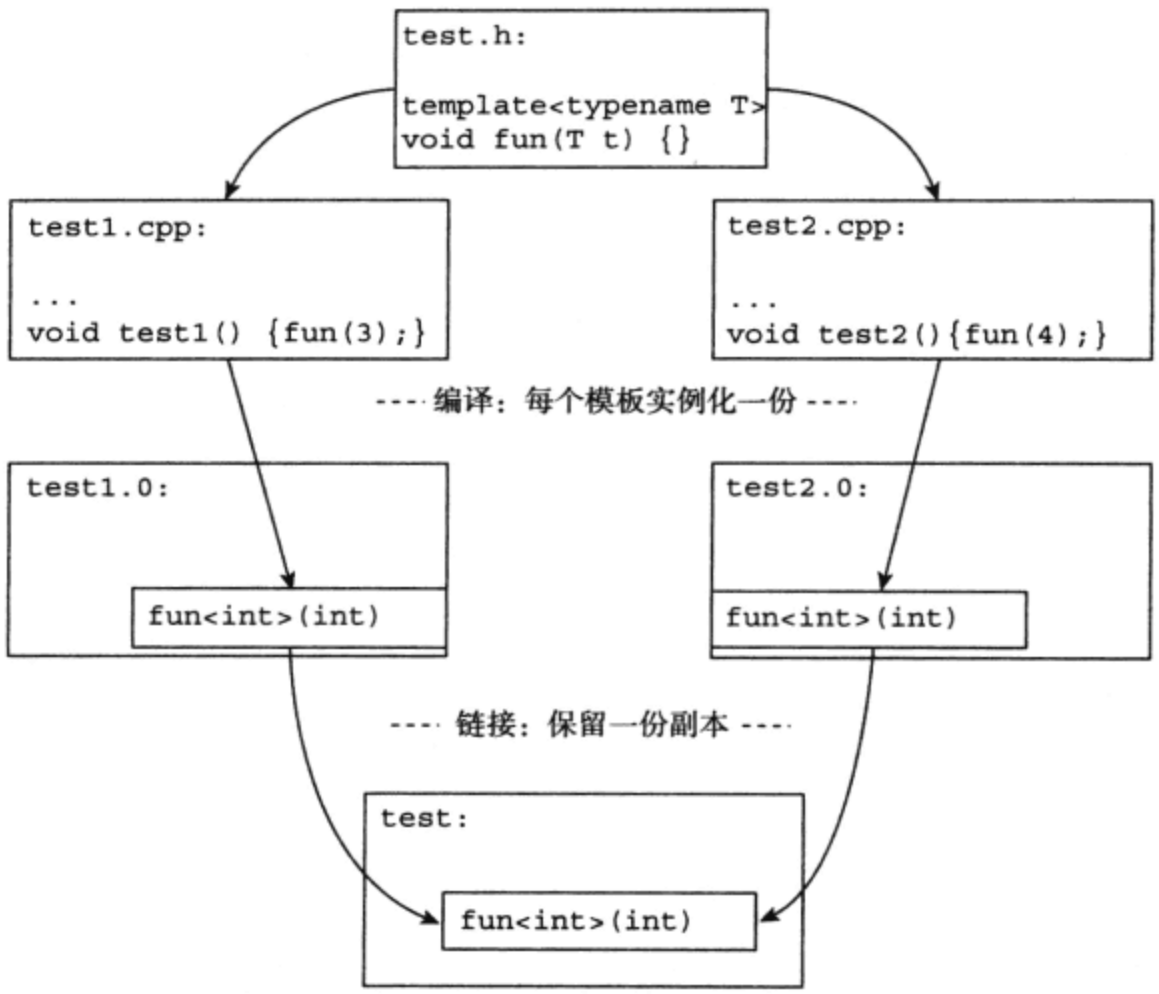

图 2-1 模板函数的编译与链接

2.12.2 显式的实例化与外部模板的声明

外部模板的使用实际依赖于C++98中一个已有的特性，即显式实例化（Explicit Instantiation）。显式实例化的语法很简单，比如对于以下模板：

```
template <typename T> void fun(T) {}
```

我们只需要声明：

```
template void fun<int>(int);
```

这就可以使编译器在本编译单元中实例化出一个 fun<int>(int) 版本的函数（这种做法也被称为强制实例化）。而在 C++11 标准中，又加入了外部模板（Extern Template）的声明。语法上，外部模板的声明跟显式的实例化差不多，只是多了一个关键字 extern。对于上面的例子，我们可以通过：

```
extern template void fun<int>(int);
```

这样的语法完成一个外部模板的声明。

那么回到一开始我们的例子，来修改一下我们的代码。首先，在 test1.cpp 做显式地实例化：

```
#include "test.h"
template void fun<int>(int); // 显示地实例化
void test1() { fun(3); }
```

接下来，在 test2.cpp 中做外部模板的声明：

```
#include "test.h"
extern template void fun<int>(int); // 外部模板的声明
void test1() { fun(3); }
```

这样一来，在 test2.o 中不会再生成 fun<int>(int) 的实例代码。整个模板的实例化流程如图 2-2 所示。

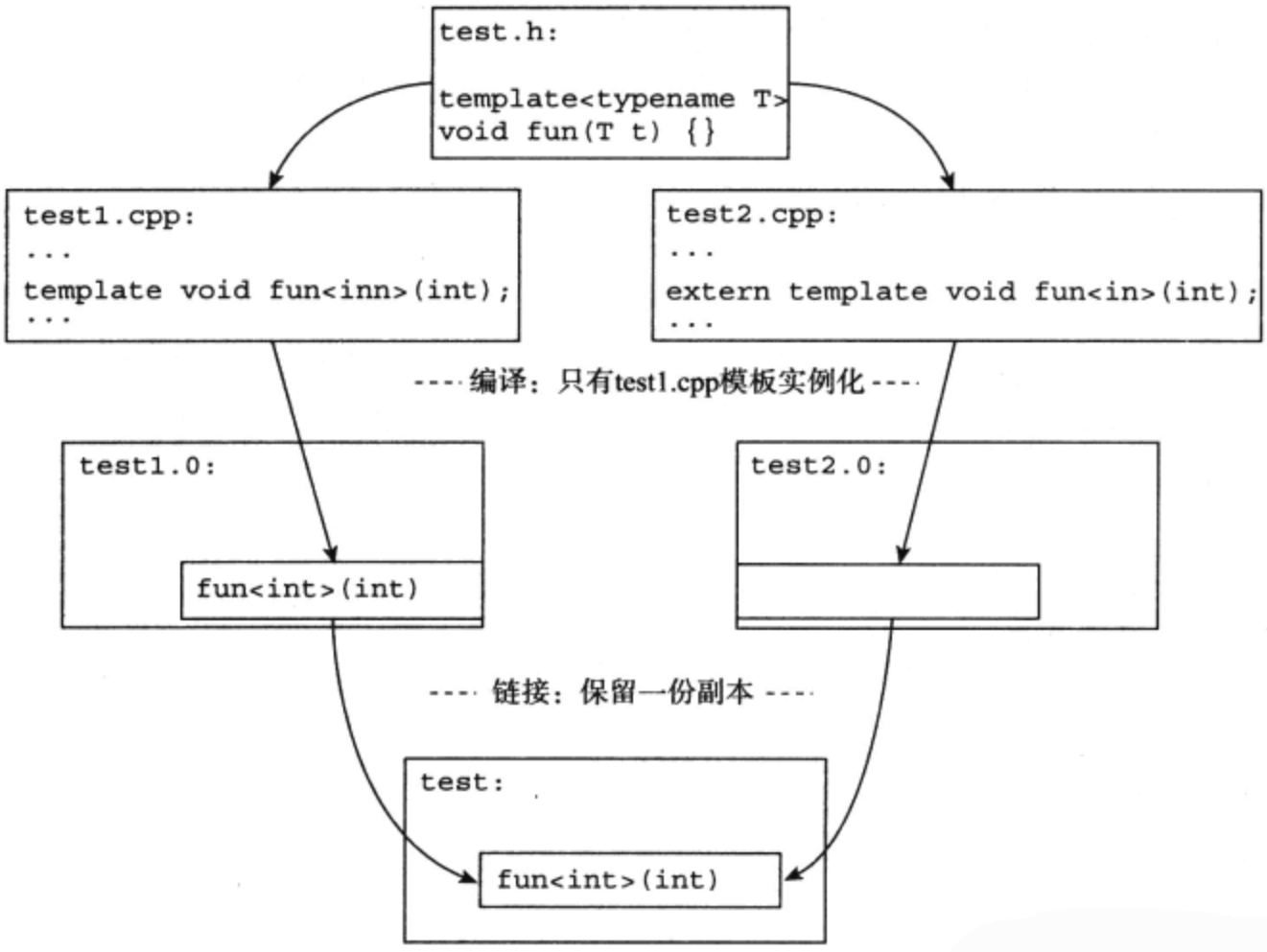

图 2-2 模板函数的编译与链接（使用外部模板声明）

可以看到，由于 test2.o 不再包含 fun<int>(int) 的实例，因此链接器的工作很轻松，基本跟外部变量的做法是一样的，即只需要保证让 test1.cpp 和 test2.cpp 共享一份代码位置即可。而同时，编译器也不用每次都产生一份 fun<int>(int) 的代码，所以可以减少编译时间。这里也可以把外部模板声明放在头文件中，这样所有包含 test.h 的头文件就可以共享这个外部模板声明了。这一点跟使用外部变量声明是完全一致的。

在使用外部模板的时候，我们还需要注意以下问题：如果外部模板声明出现于某个编译单元中，那么与之对应的显示实例化必须出现于另一个编译单元中或者同一个编译单元的后

续代码中；外部模板声明不能用于一个静态函数（即文件域函数），但可以用于类静态成员函数（这一点是显而易见的，因为静态函数没有外部链接属性，不可能在本编译单元之外出现）。

在实际上，C++11 中“模板的显式实例化定义、外部模板声明和使用”好比“全局变量的定义、外部声明和使用”方式的再次应用。不过相比于外部变量声明，不使用外部模板声明并不会导致任何问题。如我们在本节开始讲到的，外部模板定义更应该算作一种针对编译器的编译时间及空间的优化手段。很多时候，由于程序员低估了模板实例化展开的开销，因此大量的模板使用会在代码中产生大量的冗余。这种冗余，有的时候已经使得编译器和链接器力不从心。但这并不意味着程序员需要为四五十行的代码写很多显式模板声明及外部模板声明。只有在项目比较大的情况下。我们才建议用户进行这样的优化。总的来说，就是在既不忽视模板实例化产生的编译及链接开销的同时，也不要过分担心模板展开的开销。

2.13 局部和匿名类型作模板实参

☞ **类别：部分人**

在 C++98 中，标准对模板实参的类型还有一些限制。具体地讲，局部的类型和匿名的类型在 C++98 中都不能做模板类的实参。比如，如代码清单 2-30 所示的代码在 C++98 中很多都无法编译通过。

代码清单 2-30

```
template <typename T> class X {};
template <typename T> void TempFun(T t){};
struct A{} a;
struct {int i;}b;              // b 是匿名类型变量
typedef struct {int i;}B;      // B 是匿名类型

void Fun()
{
    struct C {} c;             // C 是局部类型

    X<A> x1;     // C++98 通过，C++11 通过
    X<B> x2;     // C++98 错误，C++11 通过
    X<C> x3;     // C++98 错误，C++11 通过
    TempFun(a); // C++98 通过，C++11 通过
    TempFun(b); // C++98 错误，C++11 通过
    TempFun(c); // C++98 错误，C++11 通过
}
// 编译选项 :g++ -std=c++11 2-13-1.cpp
```

在代码清单 2-30 中，我们定义了一个模板类 X 和一个模板函数 TempFun，然后分别用普通的全局结构体、匿名的全局结构体，以及局部的结构体作为参数传给模板。可以看到，

使用了局部的结构体 C 及变量 c，以及匿名的结构体 B 及变量 b 的模板类和模板函数，在 C++98 标准下都无法通过编译。而除了匿名的结构体之外，匿名的联合体以及枚举类型，在 C++98 标准下也都是无法做模板的实参的。如今看来这都是不必要的限制。所以在 C++11 中标准允许了以上类型做模板参数的做法，故而用支持 C++11 标准的编译器编译以上代码，代码清单 2-30 所示代码可以通过编译。

不过值得指出的是，虽然匿名类型可以被模板参数所接受了，但并不意味着以下写法可以被接受，如代码清单 2-31 所示。

代码清单 2-31

```
template <typename T> struct MyTemplate { };

int main() {
    MyTemplate<struct { int a; }> t; // 无法编译通过，匿名类型的声明不能在模板实参位置
    return 0;
}
// 编译选项 :g++ -std=c++11 2-13-2.cpp
```

在代码清单 2-31 中，我们把匿名的结构体直接声明在了模板实参的位置。这种做法非常直观，但却不符合 C/C++ 的语法。在 C/C++ 中，即使是匿名类型的声明，也需要独立的表达式语句。要使用匿名结构体作为模板参数，则可如同代码清单 2-30 一样对匿名结构体作别名。此外在第 4 章我们还会看到使用 C++11 独有的类型推导 decltype，也可以完成相同的功能。

2.14 本章小结

在本章中，我们可以看到 C++11 大大小小共 17 处改动。这 17 处改动，主要都是为保持 C++ 的稳定性以及兼容性而增加的。

比如为了兼容 C99，C++11 引入了 4 个 C99 的预定的宏、__func__ 预定义标识符、_Pragma 操作符、变长参数定义，以及宽窄字符连接等概念。这些都是错过了 C++98 标准，却进入了 C99 的一些标准，为了最大程度地兼容 C，C++ 将这些特性全都纳入 C++11。而由于标准的更新，C++11 也更新了 __cplusplus 宏的值，以表示新的标准的到来。而为了稳定性，C++11 不仅纳入了 C99 中的 long long 类型，还将扩展整型的规则预先定义好。这样一来，就保证了各个编译器扩展内置类型遵守统一的规则。此外，C++11 还将做法不一的静态断言做成了编译器级别的支持，以方便程序员使用。而通过抛弃 throw() 异常描述符和新增可以推导是否抛出异常的 noexcept 异常描述符，C++11 又对标准库大量代码做了改进。

在类方面，C++11 先是对非静态成员的初始化做了改进，同时允许 sizeof 直接作用于类的成员，再者 C++11 对 friend 的声明予以了一定扩展，以方便通过模板的方式指定某个类是

否是其他类或者函数的友元。而 final 和 override 两个关键字的引入，则又为对象编程增加了实用的功能。而在模板方面，C++11 则把默认模板参数的概念延伸到了模板函数上。而且局部类型和匿名类型也可以用做模板的实参。这两个约束的解除，使得模板的使用中需要记忆的规则又少了一些。而外部模板声明的引入，C++11 又为很看重编译性能的用户提供了一些优化编译时间和内存消耗的方法。

在读者读完并理解了这些特性之后，会发现它们几乎像是一台轰鸣作响的机器上的螺丝钉、润滑油、电线丝。C++ 标准委员会则通过这些小修小补，让 C++11 已有的特性看起来更加成熟，更加完美。在这一章里，虽然有的特性会带来一些“小欣喜”，但我们还看不到脱胎换骨、让人眼前一亮的新特性。不过这些零散的特性又确实非常重要，是 C++ 发展中必要的“维护”过程的必然结果。

不过如同我们讲到的，C++11 其实已经看起来像一门新的语言了。在接下来的几章中，我们会看到更多更“闪亮”的新特性。如果读者已经等不及了，那么请现在就翻开下一页。