第⑤章

提高类型安全

相比于 C 语言，C++ 则更为强调类型，其目的是为了在构建复杂的软件系统时，能够尽可能地在编译时期找到错误并提醒程序员。虽然 C++98 对于类型系统的构建已经近乎完美，却还是有枚举这样的漏网之鱼。所以 C++11 对其进行了增强。另外一方面，指针使用的安全则一直是 C++ 的重要议题。几乎所有的 C++ 的书籍都少不了指针方面的探讨。而 C++11 则再次为指针安全使用作出了努力。在本章，我们会看到 C++11 的做法。

5.1 强类型枚举

☞ **类别：部分人**

5.1.1 枚举：分门别类与数值的名字

枚举类型是 C 及 C++ 中一个基本的内置类型，不过也是一个有点“奇怪”的类型。从枚举的本意上来讲，就是要定义一个类别，并穷举同一类别下的个体以供代码中使用。由于枚举来源于 C，所以出于设计上的简单的目的，枚举值常常是对应到整型数值的一些名字。比如：

```
enum Gender { Male, Female };
```

定义了 Gender（性别）枚举类型，其中包含两种枚举值 Male 及 Famale。编译器会默认为 Male 赋值 0，为 Famale 赋值 1。这是 C 对名称的简单包装，即将名称对应到数值。

而枚举类型也可以是匿名的，匿名的枚举会有意想不到的用处。比如当程序中需要“数值的名字”的时候，我们常常可以使用以下 3 种方式来实现。

第一种方式是宏，比如：

```
#define Male 0
#define Female 1
```

宏的弱点在于其定义的只是预处理阶段的名字，如果代码中有 Male 或者 Female 的字符串，无论在什么位置一律将被替换。这在有的时候会干扰到正常的代码，因此很多时候为了避免这种情况，程序员会让宏全部以大写字母来命名，以区别于正常的代码。

而第二种方式——匿名的 enum 的状况会好些。

```
enum { Male, Female };
```

这里的匿名枚举中的 Male 和 Female 都是编译时期的名字，会得到编译器的检查。相比于宏的实现，匿名枚举不会有干扰正常代码的尴尬。

不过在 C++ 中，更受推荐是第三种方式——静态常量。如：

```
const static int Male = 0;
const static int Female = 1;
```

Male 和 Female 的名字同样得到编译时期检查。由于是静态常量，其名字作用域也被很好地局限于文件内。不过相比于 enum，静态常量不仅仅是一个编译时期的名字，编译器还可能会为 Male 和 Female 在目标代码中产生实际的数据，这会增加一点存储空间。相比而言，匿名的枚举似乎更为好用。

不过事实上，这 3 种“数值的名字”的实现方式孰优孰劣，程序员们各执一词。不过枚举类型的使用的独特性则是无需质疑的。

注意 历史上，枚举还有一个被称为“enum hack”的独特应用，在上面的静态常量的例子中，如果 static 的 Male 和 Female 声明在 class 中，在一些较早的编译器上不能为其就地赋值（赋值需要在 class 外），因此有人也采用了 enum 的方式在 class 中来代替常量声明。这就是“enum hack”。

虽然 enum 确实有些“奇怪”的用途，不过作为“枚举类型”本身而言，enum 并非完美，具体见下节。

5.1.2 有缺陷的枚举类型

C/C++ 的 enum 有个很“奇怪”的设定，就是具名（有名字）的 enum 类型的名字，以及 enum 的成员的名字都是全局可见的。这与 C++ 中具名的 namespace、class/struct 及 union 必须通过“名字 :: 成员名”的方式访问相比是格格不入的（namespace 等被称为强作用域类型，而 enum 则是非强作用域类型）。一不小心，程序员就容易遇到问题。比如下面两个枚举：

```
enum Type { General, Light, Medium, Heavy };
enum Category { General, Pistol, MachineGun, Cannon };
```

Category 中的 General 和 Type 中的 General 都是全局的名字，因此编译会报错。

而在下面的代码清单 5-1 中，则是一个通过 namespace 分割了全局空间，但 namespace 中的成员依然会被 enum 成员污染的例子。

代码清单 5-1

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

namespace T{
    enum Type { General, Light, Medium, Heavy };
}

namespace {
    enum Category { General = 1, Pistol, MachineGun, Cannon };
}

int main() {
    T::Type t = T::Light;
    if (t == General)    // 忘记使用 namespace
        cout << "General Weapon" << endl;
    return 0;
}

// 编译选项 :g++ 5-1-1.cpp
```

可以看到，Category 在一个匿名 namespace 中，所以所有枚举成员名都默认进入全局名字空间。一旦程序员在检查 t 的值的时候忘记使用了 namespace T，就会导致错误的结果（事实上，有的编译器会在这里做出一些警告，但并不会阻止编译，而有的编译器则不会警告）。

另外，由于 C 中枚举被设计为常量数值的“别名”的本性，所以枚举的成员总是可以被隐式地转换为整型。很多时候，这也是不安全的。我们可以看看代码清单 5-2 所示的这个恼人的例子。

代码清单 5-2

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

enum Type { General, Light, Medium, Heavy };
//enum Category { General, Pistol, MachineGun, Cannon }; // 无法编译通过，重复定义了 General
enum Category { Pistol, MachineGun, Cannon };

struct Killer {
    Killer(Type t, Category c) : type(t), category(c){}
    Type type;
    Category category;
};

int main() {
    Killer cool(General, MachineGun);
    // ...
```

```
    // ...其他很多代码...
    // ...
    if (cool.type >= Pistol)
        cout << "It is not a pistol" << endl;
    // ...
    cout << is_pod<Type>::value << endl;          // 1
    cout << is_pod<Category>::value << endl;      // 1

    return 0;
}

// 编译选项 :g++ -std=c++11 5-1-2.cpp
```

在上面代码清单 5-2 的例子中，类型 Killer 同时拥有 Type 和 Category 两种命名类似的枚举类型成员。在一定时候，程序员想查看这位“冷酷”（cool）的“杀手”（Killer）是属于什么 Category 的。但很明显，程序员错用了成员 type。这是由于枚举类型数值在进行数值比较运算时，首先被隐式地提升为 int 类型数据，然后自由地进行比较运算。当然，程序员的本意并非如此（事实上，我们实验机上的编译器会给出警告说不同枚举类型枚举成员间进行了比较。但程序还是编译通过了，因为标准并不阻止这一点）。

为了解决这一问题，目前程序员一般会对枚举类型进行封装。可以看看代码清单 5-2 改良后的版本，如代码清单 5-3 所示。

代码清单 5-3

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

class Type {
public:
    enum type { general, light, medium, heavy };
    type val;
public:
    Type(type t): val(t){}
    bool operator >= (const Type & t) { return val >= t.val; }
    static const Type General, Light, Medium, Heavy;
};

const Type Type::General(Type::general);
const Type Type::Light(Type::light);
const Type Type::Medium(Type::medium);
const Type Type::Heavy(Type::heavy);

class Category {
public:
    enum category { pistol, machineGun, cannon };
    category val;
```

```
public:
    Category(category c): val(c) {}
    bool operator >= (const Category & c) { return val >= c.val; }
    static const Category Pistol, MachineGun, Cannon;
};

const Category Category::Pistol(Category::pistol);
const Category Category::MachineGun(Category::machineGun);
const Category Category::Cannon(Category::cannon);

struct Killer {
    Killer(Type t, Category c) : type(t), category(c){}
    Type type;
    Category category;
};

int main() {
    // 使用类型包装后的 enum
    Killer notCool(Type::General, Category::MachineGun);
    // ...
    // ... 其他很多代码 ...
    // ...
    if (notCool.type >= Type::General)    // 可以通过编译
        cout << "It is not general" << endl;
    if (notCool.type >= Category::Pistol)    // 该句无法编译通过
        cout << "It is not a pistol" << endl;
// ...
    cout << is_pod<Type>::value << endl;          // 0
    cout << is_pod<Category>::value << endl;      // 0
    return 0;
}

// 编译选项 :g++ -std=c++11 5-1-3.cpp
```

封装的代码长得让人眼花缭乱，不过简单地说，封装即是使得枚举成员成为 class 的静态成员。由于 class 中的数据不会被默认转换为整型数据（除非定义相关操作符函数），所以可以避免被隐式转换。而且我们也可以看到，通过封装，枚举的成员也不再会污染全局名字空间了，使用时还必须带上 class 的名字，这样一来，之前枚举的一些小毛病都能够得到克服。

不过这种解决方案并非完美，至少可能有三个缺点：

- 显然，一般程序员不会为了简单的 enum 声明做这么复杂的封装。
- 由于封装且采用了静态成员，原本属于 POD 的 enum 被封装成为非 POD 的了（is_pod 均返回为 0，请对照代码清单 5-2 所示的情况），这会导致一系列的损失（参见 3.6 节）。
- 大多数系统的 ABI 规定，传递参数的时候如果参数是个结构体，就不能使用寄存器来传参（只能放在堆栈上），而相对地，整型可以通过寄存器中传递。所以一旦将 class

封装版本的枚举作为函数参数传递，就可能带来一定的性能损失。

无论上述哪一条，对于封装方案来说都是极为不利的。

此外，枚举类型所占用的空间大小也是一个“不确定量”。标准规定，C++ 枚举所基于的“基础类型”是由编译器来具体指定实现的，这会导致枚举类型成员的基本类型的不确定性问题（尤其是符号性）。我们可以看看代码清单 5-4 所示的这个例子。

代码清单 5-4

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

enum C { C1 = 1, C2 = 2};
enum D { D1 = 1, D2 = 2, Dbig = 0xFFFFFFF0U };
enum E { E1 = 1, E2 = 2, Ebig = 0xFFFFFFFFLL};
int main() {
    cout << sizeof(C1) << endl;    // 4

    cout << Dbig << endl;    // 编译器输出不同，g++: 4294967280
    cout << sizeof(D1) << endl;      // 4
    cout << sizeof(Dbig) << endl;    // 4

    cout << Ebig << endl;    // 68719476735
    cout << sizeof(E1) << endl; // 8
    return 0;
}

// 编译选项 :g++ 5-1-4.cpp
```

在代码清单 5-4 所示的例子当中，我们可以看到，编译器会根据数据类型的不同对 enum 应用不同的数据长度。在我们对 g++ 的测试中，普通的枚举使用了 4 字节的内存，而当需要的时候，会拓展为 8 字节。此外，对于不同的编译器，上例中 Dbig 的输出结果将会不同：使用 Visual C++ 编译程序的输出结果为 –16，而使用 g++ 来编译输出为 4294967280。这是由于 Visual C++ 总是使用无符号类型作为枚举的底层实现，而 g++ 会根据枚举的类型进行变动造成的。

5.1.3 强类型枚举以及 C++11 对原有枚举类型的扩展

非强类型作用域，允许隐式转换为整型，占用存储空间及符号性不确定，都是枚举类的缺点。针对这些缺点，新标准 C++11 引入了一种新的枚举类型，即“枚举类”，又称“强类型枚举”(strong-typed enum)。

声明强类型枚举非常简单，只需要在 enum 后加上关键字 class。比如：

```
enum class Type { General, Light, Medium, Heavy };
```

就声明了一个强类型的枚举 Type。强类型枚举具有以下几点优势：

- 强作用域，强类型枚举成员的名称不会被输出到其父作用域空间。
- 转换限制，强类型枚举成员的值不可以与整型隐式地相互转换。
- 可以指定底层类型。强类型枚举默认的底层类型为 int, 但也可以显式地指定底层类型，具体方法为在枚举名称后面加上“：type”，其中 type 可以是除 wchar_t 以外的任何整型。比如：

```
enum class Type: char { General, Light, Medium, Heavy };
```

就指定了 Type 是基于 char 类型的强类型枚举。

我们可以看看具体的例子，如代码清单 5-5 所示。

代码清单 5-5

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

enum class Type { General, Light, Medium, Heavy };
enum class Category { General = 1, Pistol, MachineGun, Cannon };

int main() {
    Type t = Type::Light;
    t = General;                        // 编译失败，必须使用强类型名称
    if (t == Category::General)         // 编译失败，必须使用 Type 中的 General
        cout << "General Weapon" << endl;
    if (t > Type::General)              // 通过编译
        cout << "Not General Weapon" << endl;
    if (t > 0)                          // 编译失败，无法转换为 int 类型
        cout << "Not General Weapon" << endl;
    if ((int)t > 0)                     // 通过编译
        cout << "Not General Weapon" << endl;
    cout << is_pod<Type>::value << endl;          // 1
    cout << is_pod<Category>::value << endl;      // 1
    return 0;
}

// 编译选项 :g++ -std=c++11 5-1-5.cpp
```

在代码清单 5-5 中，我们定义了两个强类型枚举 Type 和 Category，它们都包含一个称为 General 的成员。由于强类型枚举成员的名字不会输出到父作用域，因此不会有编译问题。也由于不输出成员名字，所以我们在使用该类型成员的时候必须加上其所属的枚举类型的名字。此外，可以看到，枚举成员间仍然可以进行数值式的比较，但不能够隐式地转为 int 型。事实上，如果要将强类型枚举转化为其他类型，必须进行显式转换。

事实上，强类型制止 enum 成员和 int 之间的转换，使得枚举更加符合“枚举”的本来意

义，即对同类进行列举的一个集合，而定义其与数值间的关联则使之能够默认拥有一种对成员排列的机制。而制止成员名字输出则进一步避免了名字空间冲突的问题。这两点跟之前我们使用 class 对枚举进行封装并无二致。不过新的强类型枚举没有任何 class 封装枚举的缺点。我们可以看到，Type 和 Category 都是 POD 类型，不会像 class 封装版本一样被编译器视为结构体，书写也很简便。在拥有类型安全和强作用域两重优点的情况下，几乎没有任何额外的开销。

此外，由于可以指定底层基于的基本类型，我们可以避免编译器不同而带来的不可移植性。此外，设置较小的基本类型也可以节省内存空间，如代码清单 5-6 所示。

代码清单 5-6

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

enum class C : char { C1 = 1, C2 = 2};
enum class D : unsigned int { D1 = 1, D2 = 2, Dbig = 0xFFFFFFF0U };

int main() {
    cout << sizeof(C::C1) << endl;    // 1

    cout << (unsigned int)D::Dbig << endl;    // 编译器输出一致,4294967280
    cout << sizeof(D::D1) << endl;      // 4
    cout << sizeof(D::Dbig) << endl;    // 4
    return 0;
}
// 编译选项 :g++ -std=c++11 5-1-6.cpp
```

在代码清单 5-6 中，我们为强类型枚举 C 指定底层基本类型为 char，因为我们只有 C1、C2 两个值较小的成员，一个 char 足以保存所有的枚举成员。而对于强类型枚举 D，我们指定基本类型为 unsigned int，则所有编译器都会使用无符号的 unsigned int 来保存该枚举。故各个编译器都能保证一致的输出。

相比于原来的枚举，强类型枚举更像是一个属于 C++ 的枚举。但为了配合新的枚举类型，C++11 还对原有枚举类型进行了扩展。

首先是底层的基本类型方面。在新标准 C++11 中，原有枚举类型的底层类型在默认情况下，仍然由编译器来具体指定实现。但也可以跟强类型枚举类一样，显式地由程序员来指定。其指定的方式跟强类型枚举一样，都是枚举名称后面加上“：type”，其中 type 可以是除 wchar_t 以外的任何整型。比如：

```
enum Type: char { General, Light, Medium, Heavy };
```

在 C++11 中也是一个合法的 enum 声明。

第二个扩展则是作用域的。在 C++11 中，枚举成员的名字除了会自动输出到父作用域，

也可以在枚举类型定义的作用域内有效。比如：

```
enum Type { General, Light, Medium, Heavy };
Type t1 = General;
Type t2 = Type::General;
```

General 和 Type::General 两行都是合法的使用形式。

这两个扩展都保留了向后兼容性，也方便了程序员在代码中同时操作两种枚举类型。

此外，我们在声明强类型枚举的时候，也可以使用关键字 enum struct。事实上 enum struct 和 enum class 在语法上没有任何区别（enum class 的成员没有公有私有之分，也不会使用模板来支持泛化的声明）。

有一点比较有趣的是匿名的 enum class。由于 enum class 是强类型作用域的，故匿名的 enum class 很可能什么都做不了，如代码清单 5-7 所示。

代码清单 5-7

```
enum class { General, Light, Medium, Heavy } weapon;

int main() {
    weapon = General;    // 无法编译通过
    bool b = (weapon == weapon::General);    // 无法编译通过
    return 0;
}

// 编译选项 :g++ -std=c++11 5-1-7.cpp
```

代码清单 5-7 中我们声明了一个匿名的 enum class 实例 weapon，却无法对其设置值或者比较其值（这和匿名 struct 是不一样的）。事实上，使用 enum class 的时候，应该总是为 enum class 提供一个名字（我们实验机上的 clang 编译器以及 XLC 编译器甚至会因为用户使用匿名的强类型枚举而阻止编译）。联系到我们在 5.1.1 中提到的让匿名 enum 成为“数值的名字”，匿名的 enum class 则完全做不到。所以在实际使用中必须注意（当然，程序员还是可以通过 decltype 来获得匿名强类型枚举的类型并且进行使用，即使这样做没什么太大的意义，请参见 4.3 节）。

5.2 堆内存管理：智能指针与垃圾回收

☞ 类别：类作者、库作者

5.2.1 显式内存管理

程序员在处理现实生活中的 C/C++ 程序的时候，常会遇到诸如程序运行时突然退出，或

占用的内存越来越多，最后不得不定期重启的一些典型症状。这些问题的源头可以追溯到 C/C++ 中的显式堆内存管理上。通常情况下，这些症状都是由于程序没有正确处理堆内存的分配与释放造成的，从语言层面来讲，我们可以将其归纳为以下一些问题。

- 野指针：一些内存单元已被释放，之前指向它的指针却还在被使用。这些内存有可能被运行时系统重新分配给程序使用，从而导致了无法预测的错误。
- 重复释放：程序试图去释放已经被释放过的内存单元，或者释放已经被重新分配过的内存单元，就会导致重复释放错误。通常重复释放内存会导致 C/C++ 运行时系统打印出大量错误及诊断信息。
- 内存泄漏：不再需要使用的内存单元如果没有被释放就会导致内存泄漏。如果程序不断地重复进行这类操作，将会导致内存占用剧增。

虽然显式的管理内存在性能上有一定的优势，但也被广泛地认为是容易出错的。随着多线程程序的出现和广泛使用，内存管理不佳的情况还可能会变得更加严重。因此，很多程序员也认为编程语言应该提供更好的机制，让程序员摆脱内存管理的细节。在 C++ 中，一个这样的机制就是标准库中的智能指针。在 C++11 新标准中，智能指针被进行了改进，以更加适应实际的应用需求。而进一步地，标准库还提供了所谓“最小垃圾回收”的支持。

5.2.2 C++11 的智能指针

在 C++98 中，智能指针通过一个模板类型“auto_ptr”来实现。auto_ptr 以对象的方式管理堆分配的内存，并在适当的时间（比如析构），释放所获得的堆内存。这种堆内存管理的方式只需要程序员将 new 操作返回的指针作为 auto_ptr 的初始值即可，程序员不用再显式地调用 delete。比如：

```
auto_ptr(new int);
```

这在一定程度上避免了堆内存忘记释放而造成的问题。不过 auto_ptr 有一些缺点（拷贝时返回一个左值，不能调用 delete [] 等），所以在 C++11 标准中被废弃了。C++11 标准中改用 unique_ptr、shared_ptr 及 weak_ptr 等智能指针来自动回收堆分配的对象。

这里我们可以看一个 C++11 中使用新的智能指针的简单例子，如代码清单 5-8 所示。

代码清单 5-8

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

int main() {
    unique_ptr<int> up1(new int(11));    // 无法复制的 unique_ptr
    unique_ptr<int> up2 = up1;           // 不能通过编译
```

```
    cout << *up1 << endl;    // 11

    unique_ptr<int> up3 = move(up1);     // 现在 p3 是数据唯一的 unique_ptr 智能指针

    cout << *up3 << endl;    // 11
    cout << *up1 << endl;    // 运行时错误
    up3.reset();             // 显式释放内存
    up1.reset();             // 不会导致运行时错误
    cout << *up3 << endl;    // 运行时错误

    shared_ptr<int> sp1(new int(22));
    shared_ptr<int> sp2 = sp1;

    cout << *sp1 << endl;    // 22
    cout << *sp2 << endl;    // 22

    sp1.reset();
    cout << *sp2 << endl;    // 22
}

// 编译选项 :g++ -std=c++11 5-2-1.cpp
```

在代码清单 5-8 中，使用了两种不同的智能指针 unique_ptr 及 shared_ptr 来自动地释放堆对象的内存。由于每个智能指针都重载了 * 运算符，用户可以使用 *up1 这样的方式来访问所分配的堆内存。而在该指针析构或者调用 reset 成员的时候，智能指针都可能释放其拥有的堆内存。从作用上来讲，unique_ptr 和 shared_ptr 还是和以前的 auto_ptr 保持了一致。

不过从代码清单 5-8 中还是可以看到，unique_ptr 和 shared_ptr 在对所占内存的共享上还是有一定区别的。

直观地看来，unique_ptr 形如其名地，与所指对象的内存绑定紧密，不能与其他 unique_ptr 类型的指针对象共享所指对象的内存。比如，本例中的 unique_ptr<int> up2 = up1; 不能通过编译，是因为每个 unique_ptr 都是唯一地“拥有”所指向的对象内存，由于 up1 唯一地占有了 new 分配的堆内存，所以 up2 无法共享其“所有权”。事实上，这种“所有权”仅能够通过标准库的 move 函数来转移。我们可以看到代码中 up3 的定义，unique_ptr<int> up3 = move(up1); 一旦“所有权”转移成功了，原来的 unique_ptr 指针就失去了对象内存的所有权。此时再使用已经“失势”的 unique_ptr，就会导致运行时的错误。本例中的后段使用 *up1 就是很好的例子。

而从实现上讲，unique_ptr 则是一个删除了拷贝构造函数、保留了移动构造函数的指针封装类型（我们在 7.2 节中可以看到如何删除一个类的拷贝构造函数）。程序员仅可以使用右值对 unique_ptr 对象进行构造，而且一旦构造成功，右值对象中的指针即被“窃取”，因此该右值对象即刻失去了对指针的“所有权”。

而 shared_ptr 同样形如其名，允许多个该智能指针共享地“拥有”同一堆分配对象的内存。与 unique_ptr 不同的是，由于在实现上采用了引用计数，所以一旦一个 shared_ptr 指针放弃了“所有权”（失效），其他的 shared_ptr 对对象内存的引用并不会受到影响。代码清单 5-8 中，智能指针 sp2 就很好地说明了这种状况。虽然 sp1 调用了 reset 成员函数，但由于 sp1 和 sp2 共享了 new 分配的堆内存，所以 sp1 调用 reset 成员函数只会导致引用计数的降低，而不会导致堆内存的释放。只有在引用计数归零的时候，share_ptr 才会真正释放所占有的堆内存的空间。

在 C++11 标准中，除了 unique_ptr 和 shared_ptr，智能指针还包括了 weak_ptr 这个类模板。weak_ptr 的使用更为复杂一点，它可以指向 shared_ptr 指针指向的对象内存，却并不拥有该内存。而使用 weak_ptr 成员 lock，则可返回其指向内存的一个 shared_ptr 对象，且在所指对象内存已经无效时，返回指针空值（nullptr，请参见 7.1 节）。这在验证 share_ptr 智能指针的有效性上会很有作用，如代码清单 5-9 所示。

代码清单 5-9

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

void Check(weak_ptr<int> & wp) {
    shared_ptr<int> sp = wp.lock(); // 转换为 shared_ptr<int>
    if (sp != nullptr)
        cout << "still " << *sp << endl;
    else
        cout << "pointer is invalid." << endl;
}

int main() {
    shared_ptr<int> sp1(new int(22));
    shared_ptr<int> sp2 = sp1;
    weak_ptr<int> wp = sp1; // 指向 shared_ptr<int> 所指对象

    cout << *sp1 << endl;   // 22
    cout << *sp2 << endl;   // 22
    Check(wp);              // still 22

    sp1.reset();
    cout << *sp2 << endl;   // 22
    Check(wp);              // still 22

    sp2.reset();
    Check(wp);              // pointer is invalid
}

// 编译选项 :g++ -std=c++11 5-2-2.cpp
```

在代码清单 5-9 中，我们定义了一个共享对象内存的两个 shared_ptr——sp1 及 sp2。而 weak_ptr wp 同样指向该对象内存。可以看到，在 sp1 及 sp2 都有效的时候，我们调用 wp 的 lock 函数，将返回一个有效的 shared_ptr 对象供使用，于是 Check 函数会输出以下内容：

```
still 22
```

此后我们分别调用了 sp1 及 sp2 的 reset 函数，这会导致对唯一的堆内存对象的引用计数降至 0。而一旦引用计数归 0，shared_ptr<int> 就会释放堆内存空间，使之失效。此时我们再调用 weak_ptr 的 lock 函数时，则返回一个指针空值 nullptr。这时 Check 函数则会打印出：

```
pointer is invalid
```

这样的语句了。在整个过程当中，只有 shared_ptr 参与了引用计数，而 weak_ptr 没有影响其指向的内存的引用计数。因此可以验证 shared_ptr 指针的有效性。

简单情况下，程序员用 unique_ptr 代替以前使用 auto_ptr 的代码就可以使用 C++11 中的智能指针。而 shared_ptr 及 weak_ptr 则可用在用户需要引用计数的地方。事实上，关于智能指针的历史、使用及各种讨论还有很多，不过本书的重点不在于标准库，因此这里就不一一展开了。

总地来说，虽然智能指针能帮助用户进行有效的堆内存管理，但是它还是需要程序员显式地声明智能指针，而完全不需要考虑回收指针类型的内存管理方案可能会更讨人喜欢。当然，这种方案早已有了，就是垃圾回收机制。

5.2.3 垃圾回收的分类

如果追根溯源的话，显式内存管理的替代方案很早就有了。因为早在 1959 年前后，约翰·麦肯锡（John McCarthy）就为 Lisp 语言发明了所谓“垃圾回收”的方法。这里，我们把之前使用过，现在不再使用或者没有任何指针再指向的内存空间就称为“垃圾”。而将这些“垃圾”收集起来以便再次利用的机制，就被称为“垃圾回收”（Garbage Collection）。在编程语言的发展过程中，垃圾回收的堆内存管理也得到了很大的发展。如今，垃圾回收机制已经大行其道，在大多数编程语言中，我们都可以看到对垃圾回收特性的支持，如表 5-1 所示。

表 5-1 各种编程语言对垃圾回收的支持情况

编程语言	对垃圾回收的支持情况
C++	部分
Java	支持
Python	支持
C	不支持
C#	支持

（续）

编程语言	对垃圾回收的支持情况
Ruby	支持
PHP	支持
Perl	支持
Hashkell	支持
Pascal	不支持

垃圾回收的方式虽然很多，但主要可以分为两大类：

1. 基于引用计数（reference counting garbage collector）的垃圾回收器

简单地说，引用计数主要是使用系统记录对象被引用（引用、指针）的次数。当对象被引用的次数变为0时，该对象即可被视作“垃圾”而回收。使用引用计数做垃圾回收的算法的一个优点是实现很简单，与其他垃圾回收算法相比，该方法不会造成程序暂停，因为计数的增减与对象的使用是紧密结合的。此外，引用计数也不会对系统的缓存或者交换空间造成冲击，因此被认为“副作用”较小。但是这种方法比较难处理“环形引用”问题，此外由于计数带来的额外开销也并不小，所以在实用上也有一定的限制。

2. 基于跟踪处理（tracing garbage collector）的垃圾回收器

相比于引用计数，跟踪处理的垃圾回收机制被更为广泛地应用。其基本方法是产生跟踪对象的关系图，然后进行垃圾回收。使用跟踪方式的垃圾回收算法主要有以下几种：

（1）标记－清除（Mark-Sweep）

顾名思义，这个算法可以分为两个过程。首先该算法将程序中正在使用的对象视为“根对象”，从根对象开始查找它们所引用的堆空间，并在这些堆空间上做标记。当标记结束后，所有被标记的对象就是可达对象（Reachable Object）或活对象（Live Object），而没有被标记的对象就被认为是垃圾，在第二步的清扫（Sweep）阶段会被回收掉。

这种方法的特点是活的对象不会被移动，但是其存在会出现大量的内存碎片的问题。

（2）标记－整理（Mark-Compact）

这个算法标记的方法和标记-清除方法一样，但是标记完之后，不再遍历所有对象清扫垃圾了，而是将活的对象向“左”靠齐，这就解决了内存碎片的问题。

标记－整理的方法有个特点就是移动活的对象，因此相应的，程序中所有对堆内存的引用都必须更新。

（3）标记－拷贝（Mark-Copy）

这种算法将堆空间分为两个部分：From 和 To。刚开始系统只从 From 的堆空间里面分配

内存，当 From 分配满的时候系统就开始垃圾回收：从 From 堆空间找出所有活的对象，拷贝到 To 的堆空间里。这样一来，From 的堆空间里面就全剩下垃圾了。而对象被拷贝到 To 里之后，在 To 里是紧凑排列的。接下来是需要将 From 和 To 交换一下角色，接着从新的 From 里面开始分配。

标记 - 拷贝算法的一个问题是堆的利用率只有一半，而且也需要移动活的对象。此外，从某种意义上讲，这种算法其实是标记 - 整理算法的另一种实现而已。

虽然历来 C++ 都没有公开地支持过垃圾回收机制，但垃圾回收并非某些语言的专利。事实上，C++11 标准也开始对垃圾回收做了一定的支持，虽然支持的程度还非常有限，但我们已经看到了 C++ 语言变得更为强大的端倪。

5.2.4 C++ 与垃圾回收

如我们提到的，在 C++11 中，智能指针等可以支持引用计数。不过由于引用计数并不能有效解决形如“环形引用”等问题，其使用会受到一些限制。而且基于一些其他的原因，比如因多线程程序等而引入的内存管理上的困难，程序员可能也会需要垃圾回收。

一些第三方的 C/C++ 库已经支持标记 - 清除方法的垃圾回收，比如一个比较著名的 C/C++ 垃圾回收库——Boehm [⊖]。该垃圾回收器需要程序员使用库中的堆内存分配函数显式地替代 malloc，继而将堆内存的管理交给垃圾回收器来完成垃圾回收。不过由于 C/C++ 中指针类型的使用非常灵活，这样的库在实际使用中会有一些限制，可移植性也不好。

为了解决垃圾回收中的安全性和可移植性问题，在 2007 年，惠普的 Hans-J. Boehm（Boehm 的作者）和赛门铁克的 Mike Spertus 共同向 C++ 委员会递交了一个关于 C++ 中垃圾回收的提案。该提案通过添加 gc_forbidden、gc_relaxed、gc_required、gc_safe、gc_strict 等关键字来支持 C++ 语言中的垃圾回收。该提案甚至可以让程序员显式地要求垃圾回收。刚开始这得到了大多数委员的支持，后来却在标准的初稿中删除了，原因是该特性过于复杂，并且还存在一些问题（比如与显式调用析构函数的现有的库的兼容问题等）。所以，Boehm 和 Spertus 对初稿进行了简化，仅仅保留了支持垃圾回收的最基本的部分，即通过对语言的约束，来保证安全的垃圾回收。这也是我们现在看到的 C++11 标准中的“最小垃圾回收支持”的历史来由。

而要保证安全的垃圾回收，首先必须知道 C/C++ 语言中什么样的行为可能导致垃圾回收中出现“不安全”的状况。简单地说，不安全源自于 C/C++ 语言对指针的“放纵”，即允许过分灵活的使用。我们可以看代码清单 5-10 所示的例子。

⊖ http://www.hpl.hp.com/personal/Hans_Boehm/gc/

代码清单 5-10

```
int main(){
    int* p = new int;
    p += 10;     // 移动指针，可能导致垃圾回收器
    p -= 10;     // 回收原来指向的内存
    *p = 10;     // 再次使用原本相同的指针则可能无效
}

// 编译选项 :g++ 5-2-3.cpp
```

在代码清单 5-10 中，我们对指针 p 做了自加和自减操作。这在 C/C++ 中被认为是合理的，因为指针有所指向的类型，自加或者自减能够使程序员轻松地找到“下一个”同样的对象（实际是一个迭代器的概念）。不过对于垃圾回收来说，一旦 p 指向了别的地址，则可认为 p 曾指向的内存不再使用。垃圾回收器可以据此对其进行回收。这对之后 p 的使用（*p = 10）带来的后果将是灾难性的。

我们再来看一个例子，如代码清单 5-11 所示。

代码清单 5-11

```
int main() {
    int *p = new int;
    int *q = (int*)(reinterpret_cast<long long>(p) ^ 2012);    // q 隐藏了 p
    // 做一些其他工作，垃圾回收器可能已经回收了 p 指向对象
    q = (int*)(reinterpret_cast<long long>(q) ^ 2012); // 这里的 q == p
    *q = 10;
}

// 编译选项 :g++ 5-2-4.cpp
```

在代码清单 5-11 中，我们用指针 q 隐藏了指针 p。而之后又用可逆的异或运算将 p “恢复”了出来。在 main 函数中，p 实际所指向的内存都是有效的，但由于该指针被隐藏了，垃圾回收器可以早早地将 p 指向的对象回收掉。同样，语句 *q = 10 的后果也是灾难性的。

指针的灵活使用可能是 C/C++ 中的一大优势，而对于垃圾回收来说，却会带来很大的困扰。被隐藏的指针会导致编译器在分析指针的可达性（生命周期）时出错。而即使编译器开发出了隐藏指针分析的手段，其带来的编译开销也不会让程序员对编译时间的显著增长视而不见。历史上，解决这种问题的方法通常是新接口。C++11 和垃圾回收的解决方案也不例外，就是让程序员利用这样的接口来提示编译器代码中存在指针不安全的区域。

5.2.5 C++11 与最小垃圾回收支持

C++11 新标准为了做到最小的垃圾回收支持，首先对“安全”的指针进行了定义，或者

使用 C++11 中的术语说，安全派生（safely derived）的指针。安全派生的指针是指向由 new 分配的对象或其子对象的指针。安全派生指针的操作包括：

- ❑在解引用基础上的引用，比如：&*p。
- ❑定义明确的指针操作，比如：p + 1。
- ❑定义明确的指针转换，比如：static_cast <void *> (p)。
- ❑指针和整型之间的 reinterpret_case，比如：reinterpret_cast <intptr_t>(p)。

注意 intptr_t 是 C++11 中一个可选择实现的类型，其长度等于平台上指针的长度（通过 decltype 声明）。

我们可以回头看看代码清单 5-11。reinterpret_cast<long long>(p) 是合法的安全派生操作，而转换后的指针再进行异或操作：reinterpret_cast<long long>(p) ^ 2012 之后，指针就不再是安全派生的了，这是因为异或操作（^）不是一个安全派生操作。同理，reinterpret_cast<long long> (q) ^ 2012 也不是安全派生指针。因此，根据定义，在使用内存回收器的情况下，*q = 10 的行为是不确定的，如果程序在此处发生错误也是合理的。

在 C++11 的规则中，最小垃圾回收支持是基于安全派生指针这个概念的。程序员可以通过 get_pointer_safety 函数查询来确认编译器是否支持这个特性。get_pointer_safety 的原型如下：

```
pointer_safety get_pointer_safety() noexcept
```

其返回一个 pointer_safety 类型的值。如果该值为 pointer_safety::strict，则表明编译器支持最小垃圾回收及安全派生指针等相关概念，如果该值为 pointer_safety::relax 或是 pointer_safety::preferred，则表明编译器并不支持，基本上跟没有垃圾回收的 C 和 C++98 一样。不过按照一些解释，pointer_safety::preferred 和 pointer_safety::relax 也略有不同，前者垃圾回收器可能被用作一些辅助功能，如内存泄露检测或检测对象是否被一个错误的指针解引用（事实上，在本书编写时，几乎没有编译器实现了最小垃圾回收支持，甚至连 get_pointer_safety 这个函数接口都还没实现）。

此外，如果程序员代码中出现了指针不安全使用的状况，C++11 允许程序员通过一些 API 来通知垃圾回收器不得回收该内存。C++11 的最小垃圾回收支持使用了垃圾回收的术语，即需声明该内存为“可到达”的。

```
void declare_reachable( void* p );
template <class T> T *undeclare_reachable(T *p) noexcept;
```

declare_reachable() 显式地通知垃圾回收器某一个对象应被认为可达的，即使它的所有指针都对回收器不可见。undeclare_reachable() 则可以取消这种可达声明。针对代码清单 5-11，我们对隐藏的指针做一些声明，如代码清单 5-12 所示。

代码清单 5-12

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

int main() {
    int *p = new int;
    declare_reachable(p);    // 在 p 被隐藏之前声明为可达的
    int *q = (int*)((long long)p ^ 2012);
    // 解除可达声明
    q = undeclare_reachable<int>((int*)((long long)q ^ 2012));
    *q = 10;
}
```

代码清单 5-12 可能是一个能够运行的例子。这里，我们在 p 指针被不安全派生（隐藏）之前使用 declare_reachable 声明其是可达的。这样一来，它会被垃圾回收器忽略而不会被回收。而在我们通过可逆的异或运算使得 q 指针指向 p 所指对象时，我们则使用了 undeclare_reachable 来取消可达声明。注意 undeclare_reachable 不是通知垃圾回收器 p 所指对象已经可以回收。实际上，declare_reachable 和 undeclare_reachable 只是确立了一个代码范围，即在两者之间的代码运行中，p 所指对象不会被垃圾回收器所回收。

这里可能有的读者会注意到一个细节，declare_reachable 只需要传入一个简单的 void* 指针，但 undeclare_reachable 却被设计为一个函数模板。这是一个极不对称的设计。但事实上 undeclare_reachable 使用模板的主要目的是为了返回合适类型以供程序使用。而垃圾回收器本来就知道指针所指向的内存的大小，因此 declare_reachable 传入 void* 指针就已经足够了。

有的时候程序员会选择在一大片连续的堆内存上进行指针式操作，为了让垃圾回收器不关心该区域，也可以使用 declare_no_pointers 及 undeclare_no_pointers 函数来告诉垃圾回收器该内存区域不存在有效的指针。

```
void declare_no_pointers(char *p, size_t n) noexcept;
void undeclare_no_pointers(char *p, size_t n) noexcept;
```

其使用方式与 declare_reachable 及 undeclare_reachable 类似，不过指定的是从 p 开始的连续 n 的内存。

5.2.6 垃圾回收的兼容性

尽管在设计 C++11 标准时想尽可能保证向后兼容，但是对于垃圾回收来说，破坏向后兼容是不可避免的。通常情况下，如果我们想要程序使用垃圾回收，或者可靠的内存泄漏检测，我们就必须做出必要的假设来保证垃圾回收器能工作。而为此，我们必须限制指针的使用或者使用 declare_reachable/undeclare_reachable、declare_no_pointer/undeclare_no_pointer 来让一些不安全的指针使用免于垃圾回收器的检查。因此想让老的代码毫不费力地使用垃圾回

收，现实情况下对大多数代码还是不可能的。

此外，C++11 标准中对指针的垃圾回收支持仅限于系统提供的 new 操作符分配的内存，而 malloc 分配的内存则会被认为总是可达的，即无论何时垃圾回收器都不予回收。因此使用 malloc 等的较老代码的堆内存还是必须由程序员自己控制。

而更为现实的状况是在本书写作时，垃圾回收的特性还没有得到任何编译器支持，即使是所谓的“最小垃圾回收”。标准的发展以及垃圾回收在 C/C++ 中的实现可能还需要一定的时间。不过有了最小支持，用户可能在新代码中会注意指针的使用，并对形如指针隐藏的状况使用合适的函数来对被隐藏指针的堆对象进行保护。按照 C++ 的设计，显式地 delete 使用与垃圾回收并不会形成冲突。如果程序员选择这么做的话，就应该能够保证最大的代码向前兼容性。在未来某个时刻 C++ 垃圾回收支持完成的时候，代码可以直接享受其带来的益处。

5.3 本章小结

C++ 是一种静态类型的语言，在 C++ 的发展中，一系列的改进使得 C++ 的类型机制几乎严丝合缝，滴水不漏。而在 C++11 中，最后的漏网之鱼——枚举，终于也以新增的强类型枚举的方式进行了规范。虽然为了兼容性，旧的枚举语义没有任何改变，但新的强类型枚举可以避免形如名字冲突、隐式向整型转换等诸多问题，所以在使用上更加安全。事实上，现有的标准库中的枚举就已经大量采用了强类型枚举来规避编程中的风险。

而堆分配变量的自动释放从来都是编程者津津乐道的问题。理想情况下，程序员应该总是在栈上分配变量，这样变量能够有效地自动释放。不过现实情况下，很多时候程序员在编程时并不能预期所需内存的使用期，所以程序员还离不开 malloc/free 或者 new/delete 的堆式内存分配。在 C++98 中，用户可以使用智能指针 auto_ptr 来自动释放内存。C++11 中则进一步改进了 auto_ptr——程序员可以使用行为更加良好的 unique_ptr 和 shared_ptr/weak_ptr 智能指针来进行自动的堆内存释放。虽然 unique/shared/weak 在概念上比起简单的 auto 变得复杂了，但是行为上却更加安全（返回右值、调用 delete[] 等）。程序员应该在 C++11 中全面使用新的智能指针代替 auto_ptr。

而随着编程模型的复杂化，C++ 语言也在考虑引入全面的垃圾回收。不过在 C++11 中，全面的垃圾回收这个心愿尚未达成。实际在 C++11 中的是“最小化”的垃圾回收支持，即定义了什么样的指针对于垃圾回收是安全的，什么样的动作会对垃圾回收造成影响。一旦有了会对垃圾回收造成影响的操作，程序员则可以调用 API 来通知垃圾回收器代码对于垃圾回收是否可达。事实上，现有情况下，我们并看不到“最小垃圾回收支持”的实质应用，大多数编译器也还没有实现相关的内容，但有了最小的支持，程序员和编译器及库的制作者间的接口也就清晰了。不排除以后有编译器或者库制作者设计出能够实现垃圾回收的、能够编译满足最小垃圾回收支持的 C++ 代码的编译器或库产品。不过，能否实现还需要假以时日。