第④章

新手易学，老兵易用

在 C++ 变得原来越强大的同时，一些程序员也在抱怨使用 C++ 进行编码过于复杂。由于 STL 与语言核心部分的关联越来越紧密，而 STL 往往需要涉及解释一些模板的知识，很多新手也会因此对使用 C++ 进行编程产生排斥。C++11 的设计者对此作了改进。本章中，我们可以看到 auto 类型推导、基于范围的 for 循环等非常易用的特性。这些特性都非常具有亲和力，而用于进行代码编写也能有效地提升代码效率，必然会被 C++11 代码大量使用。

4.1 右尖括号 > 的改进

☞ **类别：所有人**

在 C++98 中，有一条需要程序员规避的规则：如果在实例化模板的时候出现了连续的两个右尖括号 >，那么它们之间需要一个空格来进行分隔，以避免发生编译时的错误。我们可以看代码清单 4-1 所示的例子。

代码清单 4-1

```
template <int i> class X{};
template <class T> class Y{};

Y<X<1> > x1;     // 编译成功
Y<X<2>> x2;      // 编译失败

// 编译选项 :g++ -c 4-1-1.cpp
```

在代码清单 4-1 中，我们定义了两个模板类型 X 和 Y，并且使用模板定义分别声明了以 X<1> 为参数的 Y<X<1> > 类型变量 x1，以及以 X<2> 为参数的 Y<X<2> > 类型变量 x2。不过 x2 的定义编译器却不能正确解析。在 x2 的定义中，编译器会把 >> 优先解析为右移符号。使用 gcc 编译的时候，我们会得到如下错误提示：

```
4-1-1.cpp:5:9: error: 'x2' was not declared in this scope
 Y<X<2>> x2;      // 编译失败
         ^
4-1-1.cpp:5:9: error: template argument 1 is invalid
4-1-1.cpp:5:3: error: template argument 1 is invalid
 Y<X<2>> x2;      // 编译失败
```

事实上，除去嵌套的模板标识，在使用形如 static_cast、dynamic_cast、reinterpret_cast，或者 const_cast 表达式进行转换的时候，我们也常会遇到相同的情况。

```
const vector<int> v = static_cast<vector<int>>(v);      // 无法通过编译
```

C++98 同样会将 >> 优先解析为右移。C++11 中，这种限制被取消了。事实上，C++11 标准要求编译器智能地去判断在哪些情况下 >> 不是右移符号。使用 C++11 标准，代码清单 4-1 所示代码则会成功地通过编译。

不过这些“智能”的判断也会带来一些与 C++98 的有趣的不兼容性。比如用户只是想让 >> 在模板的实例化中表示的是真正的右移，但是 C++11 会把它解析为模板参数界定符。比如代码清单 4-2 所示的代码。

代码清单 4-2

```
template <int i> class X {};
X<1 >> 5> x ;
```

如果使用 C++98 标准进行编译的话，这个例子会编译通过，因为编译器认为 X<1 >> 5> x ; 中的双尖括号是一个位移操作，那么最终可以得到一个形如 X<0> x 的模板实例。而如果使用 C++11 标准进行编译，那么程序员会得到一个编译错误的警告，因为编译器优先将双尖括号中的第一个 > 与 X 之后的 < 进行了配对。

虽然很少有人在模板实例化时同时进行位移操作，但是从语法上来说，C++98 和 C++11 确实在这一点上不兼容。要避免这样的不兼容性也很简单，使用圆括号将“1 >> 5”括起来，保证右移操作优先，就不会出现类似问题了。

4.2 auto 类型推导

☞ **类别：所有人**

4.2.1 静态类型、动态类型与类型推导

在编程语言的分类中，C/C++ 常被冠以“静态类型”的称号。而有的编程语言则号称是“动态类型”的，比如 Python。通常情况下，“静”和“动”的区别非常直观。我们可以看看下面这段简单的 Python 代码：

```
name = 'world\n'
print 'hello, ' % name
```

这段代码是 Python 中的一个 helloworld 的实现。这里的代码使用了一个变量 name，用于存储 world 这个字符串。接下来代码又使用 print 将 'hello, ' 字符串及 name 变量一起打印出来。请读者忽略其他的细节，只注意一下变量 name 的使用方式。事实上，变量 name 在使用

前没有进行过任何声明，而当程序员想使用时，可以拿来就用。

这种变量的使用方式显得非常随性，而在 C/C++ 程序员的眼中，每个变量使用前必须定义几乎是天经地义的事，这样通常被视为编程语言中的“静态类型”的体现。而对于如 Python、Perl、JavaScript 等语言中变量不需要声明，而几乎“拿来就用”的变量使用方式，则被视为是编程语言中“动态类型”的体现。不过从技术上严格地讲，静态类型和动态类型的主要区别在于对变量进行类型检查的时间点。对于所谓的静态类型，类型检查主要发生在编译阶段；而对于动态类型，类型检查主要发生在运行阶段。形如 Python 等语言中变量“拿来就用”的特性，则需要归功于一个技术，即类型推导。

事实上，类型推导也可以用于静态类型的语言中。比如在上面的 Python 代码中，如果按照 C/C++ 程序员的思考方式，world\n 表达式应该返回一个临时的字符串，所以即使 name 没有进行声明，我们也能轻松地推导出 name 的类型应该是一个字符串类型。在 C++11 中，这个想法得到了实现。C++11 中类型推导的实现的方式之一就是重定义了 auto 关键字。另外一个现实是 decltype，关于 decltype 的实现，我们将在后面详细描述。

我们可以使用 C++11 的方式书写一下刚才的 Python 代码，如代码清单 4-3 所示。

代码清单 4-3

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

int main() {
    auto name = "world.\n";
    cout << "hello, " << name;
}

// 编译选项 :g++ -std=c++11 4-2-1.cpp
```

这里我们使用了 auto 关键字来要求编译器对变量 name 的类型进行自动推导。这里编译器根据它的初始化表达式的类型，推导出 name 的类型为 char*。这样一来就达到了跟刚才的 Python 代码差不多的效果。

事实上，auto 关键字在早期的 C/C++ 标准中有着完全不同的含义。声明时使用 auto 修饰的变量，按照早期 C/C++ 标准的解释，是具有自动存储期的局部变量。不过现实情况是该关键字几乎无人使用，因为一般函数内没有声明为 static 的变量总是具有自动存储期的局部变量。因此在 C++11 中，标准委员会决定赋予 auto 全新的含义，即 auto 不再是一个存储类型指示符（storage-class-specifier，如 static、extern、thread_local 等都是存储类型指示符），而是作为一个新的类型指示符（type-specifier，如 int、float 等都是类型指示符）来指示编译器，auto 声明变量的类型必须由编译器在编译时期推导而得。

我们可以通过代码清单 4-4 所示的例子来了解一下 auto 类型推导的基本用法。

代码清单 4-4

```
int main() {
    double foo();
    auto x = 1;        // x的类型为int
    auto y = foo();   // y的类型为double
    struct m { int i; }str;
    auto str1 = str;     // str1的类型是struct m

    auto z;       // 无法推导，无法通过编译
    z = x;
}

// 编译选项:g++ -std=c++11 4-2-2.cpp
```

在代码清单 4-4 中，变量 x 被初始化为 1，因为字面常量 1 的类型为 const int，所以编译器推导出 x 的类型应该为 int（这里 const 类型限制符被去掉了，后面会解释）。同理在变量 y 的定义中，auto 类型的 y 被推导为 double 类型；而在 auto str1 的定义中，其类型被推导为 struct m。

值得注意的是变量 z，这里我们使用 auto 关键字来“声明”z，但不立即对其进行定义，此时编译器则会报错。这跟通过其他关键字（除去引用类型的关键字）先声明后定义变量的使用规则是不同的。auto 声明的变量必须被初始化，以使编译器能够从其初始化表达式中推导出其类型。从这个意义上来讲，auto 并非一种“类型”声明，而是一个类型声明时的“占位符”，编译器在编译时期会将 auto 替代为变量实际的类型。

4.2.2 auto 的优势

直观地，auto 推导的一个最大优势就是在拥有初始化表达式的复杂类型变量声明时简化代码。由于 C++ 的发展，声明变量类型也变得越来越复杂，很多时候，名字空间、模板成为了类型的一部分，导致程序员在使用库的时候如履薄冰。我们可以看看代码清单 4-5 所示的例子。

代码清单 4-5

```
#include <string>
#include <vector>

void loopover(std::vector<std::string> & vs) {
    std::vector<std::string>::iterator i = vs.begin(); // 想要使用iterator，往往需要
                                                       书写大量代码

    for (; i < vs.end(); i++) {
        // 一些代码
    }
```

```
}

// 编译选项 :g++ -c 4-2-3.cpp
```

代码清单 4-5 中，我们在不使用 using namespace std 的情况下（事实上，很多专家的建议就是如此）想对一个 vector 数组进行循环。可以看到，当我们想定义一个迭代器 i 的时候，我们必须写出 std::vector<std::string>::iterator 这样长的类型声明。即使是一位擅长克服各种困难的 C++ 老手，也不会对如此冗长的代码视而不见。而使用 auto 的话，代码会的可读性可以成倍增长，如代码清单 4-6 所示。

代码清单 4-6

```
#include <string>
#include <vector>

void loopover(std::vector<std::string> & vs) {

    for (auto i = vs.begin(); i < vs.end(); i++) {
        // 一些代码
    }
}

// 编译选项 :g++ -c -std=c++11 4-2-4.cpp
```

如我们所见，使用了 auto，程序员甚至可以将 i 的声明放入 for 循环中，i 的类型将由表达式 vs.begin() 推导出。事实上，形如代码清单 4-5 的复杂声明在使用 STL 的代码中处处可见，在 C++11 中，由于 auto 的存在，使用 STL 将会变得更加容易，写出的代码也会更加清晰可读。

auto 的第二个优势则在于可以免除程序员在一些类型声明时的麻烦，或者避免一些在类型声明时的错误。事实上，在 C/C++ 中，存在着很多隐式或者用户自定义的类型转换规则（比如整型与字符型进行加法运算后，表达式返回的是整型，这是一条隐式规则）。这些规则并非很容易记忆，尤其是在用户自定义了很多操作符之后。而这个时候，auto 就有用武之地了。我们可以看看代码清单 4-7 所示的例子。

代码清单 4-7

```
class PI {
public:
    double operator* (float v) {
        return (double)val * v;     // 这里精度被扩展了
    }
    const float val = 3.1415927f;
};
```

```
int main() {
    float radius = 1.7e10;
    PI pi;
    auto circumference = 2 * (pi * radius);
}

// 编译选项 :g++ -std=c++11 4-2-5.cpp
```

代码清单 4-7 中，定义了 float 型的变量 radius（半径）以及一个自定义类型 PI 变量 pi（π 值），在计算圆周长的时候，使用了 auto 类型来定义变量 circumference。这里，PI 在与 float 类型数据相乘时，其返回值为 double。而 PI 的定义可能是在其他的地方（头文件里），main 函数的程序员可能不知道 PI 的作者为了避免数据上溢或者精度降低而返回了 double 类型的浮点数。因此 main 函数程序员如果使用 float 类型声明 circumference，就可能享受不了 PI 作者细心设计带来的好处。反之，将 circumference 声明为 auto，则毫无问题，因为编译器已经自动地做了最好的选择。

值得指出的是，auto 并不能解决所有的精度问题，我们可以看看代码清单 4-8 所示的例子。

代码清单 4-8

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

int main() {
    unsigned int a = 4294967295;     // 最大的 unsigned int 值
    unsigned int b = 1;
    auto c = a + b;          // c 的类型依然是 unsigned int

    cout << "a = " << a << endl;     // a = 4294967295
    cout << "b = " << b << endl;     // b = 1
    cout << "a + b = " << c << endl;// a + b = 0
    return 0;
}

// 编译选项 :g++ -std=c++11 4-2-6.cpp
```

在代码清单 4-8 中，程序员可能指望通过声明变量 c 为 auto 就能解决 a + b 溢出的问题。而实际上由于 a+b 返回的依然是 unsigned int 的值，故而 c 的类型依然被推导为 unsigned int，auto 并不能帮上忙。这跟一些动态类型语言中数据会自动进行扩展的特性还是不一样的。

auto 的第三个优点就是其“自适应”性能够在一定程度上支持泛型的编程。

我们再回到代码清单 4-7 的例子，这里假设 PI 的作者改动了 PI 的定义，比如将 operator* 返回值变为 long double，此时，main 函数并不需要修改，因为 auto 会“自适应”

新的类型。

同理，对于不同的平台上的代码维护，auto 也会带来一些“泛型”的好处。这里我们以 strlen 函数为例，在 32 位的编译环境下，strlen 返回的为一个 4 字节的整型，而在 64 位的编译环境下，strlen 会返回一个 8 字节的整型。虽然系统库 <cstring> 为其提供了 size_t 类型来支持多平台间的代码共享支持，但是使用 auto 关键字我们同样可以达到代码跨平台的效果。

```
auto var = strlen("hello world!").
```

由于 size_t 的适用范围往往局限于 <cstring> 中定义的函数，auto 的适用范围明显更为广泛。

当 auto 应用于模板的定义中，其“自适应”性会得到更加充分的体现。我们可以看看代码清单 4-9 所示的例子。

代码清单 4-9

```
template<typename T1, typename T2>
double Sum(T1 & t1, T2 & t2) {
    auto s = t1 + t2;    // s 的类型会在模板实例化时被推导出来
    return s;
}

int main() {
    int a = 3;
    long b = 5;
    float c = 1.0f, d = 2.3f;

    auto e = Sum<int ,long>(a, b);        // s 的类型被推导为 long
    auto f = Sum<float,float>(c, d);      // s 的类型被推导为 float
}

// 编译选项 :g++ -std=c++11 4-2-7.cpp
```

在代码清单 4-9 中，Sum 模板函数接受两个参数。由于类型 T1、T2 要在模板实例化时才能确定，所以在 Sum 中将变量 s 的类型声明为 auto 的。在函数 main 中我们将模板实例化时，Sum<int, long> 中的 s 变量会被推导为 long 类型，而 Sum<float, float> 中的 s 变量则会被推导为 float。可以看到，auto 与模板一起使用时，其“自适应”特性能够加强 C++ 中“泛型”的能力。不过在这个例子中，由于总是返回 double 类型的数据，所以 Sum 模板函数的适用范围还是受到了一定的限制，在 4.4 节中，我们可以看到怎么使用追踪返回类型的函数声明来完全释放 Sum 泛型的能量。

另外，应用 auto 还会在一些情况下取得意想不到的好效果。我们可以看看代码清单 4-10 所示的例子[⊖]。

⊖ 本例素材来源于 GNU C 的手册 http://gcc.gnu.org/onlinedocs/gcc/Typeof.html。

代码清单 4-10

```
#define Max1(a, b) ((a) > (b)) ? (a) : (b)
#define Max2(a, b) ({ \
        auto _a = (a); \
        auto _b = (b); \
        (_a > _b) ? _a: _b; })

int main() {
    int m1 = Max1(1*2*3*4, 5+6+7+8);
    int m2 = Max2(1*2*3*4, 5+6+7+8);
}

// 编译选项 :g++ -std=c++11 4-2-8.cpp
```

在代码清单 4-10 中，我们定义了两种类型的宏 Max1 和 Max2。两者作用相同，都是求 a 和 b 中较大者并返回。前者采用传统的三元运算符表达式，这可能会带来一定的性能问题。因为 a 或者 b 在三元运算符中都出现了两次，那么无论是取 a 还是取 b，其中之一都会被运算两次。而在 Max2 中，我们将 a 和 b 都先算出来，再使用三元运算符进行比较，就不会存在这样的问题了。

在传统的 C++98 标准中，由于 a 和 b 的类型无法获得，所以我们无法定义 Max2 这样高性能的宏。而新的标准中的 auto 则提供了这种可行性。

4.2.3 auto 的使用细则

虽然我们在 4.2.1 节及 4.2.2 节中看到了很多关于 auto 的使用，不过 auto 使用上还有很多语法相关的细节。如果读者在使用 auto 的时候遇到一些不理解的状况，不妨回头来查阅一下这些规则。

首先我们可以看看 auto 类型指示符与指针和引用之间的关系。在 C++11 中，auto 可以与指针和引用结合起来使用，使用的效果基本上会符合 C/C++ 程序员的想象。我们可以看看代码清单 4-11 所示的例子。

代码清单 4-11

```
int x;
int * y = &x;
double foo();
int & bar();

auto * a = &x;       // int*
auto & b = x;        // int&
auto c = y;          // int*
auto * d = y;        // int*
auto * e = &foo();   // 编译失败，指针不能指向一个临时变量
```

```
auto & f = foo();    // 编译失败，nonconst 的左值引用不能和一个临时变量绑定
auto g = bar();      // int
auto & h = bar();    // int&

// 编译选项 :g++ -std=c++11 4-2-9.cpp
```

本例中，变量 a、c、d 的类型都是指针类型，且都指向变量 x。实际上对于 a、c、d 三个变量而言，声明其为 auto * 或 auto 并没有区别。而如果要使得 auto 声明的变量是另一个变量的引用，则必须使用 auto &，如同本例中的变量 b 和 h 一样。

其次，auto 与 volatile 和 const 之间也存在着一些相互的联系。volatile 和 const 代表了变量的两种不同的属性：易失的和常量的。在 C++ 标准中，它们常常被一起叫作 cv 限制符（cv-qualifier）。鉴于 cv 限制符的特殊性，C++11 标准规定 auto 可以与 cv 限制符一起使用，不过声明为 auto 的变量并不能从其初始化表达式中“带走”cv 限制符。我们可以看看代码清单 4-12 所示的例子。

代码清单 4-12

```
double foo();
float * bar();

const auto a = foo();          // a: const double
const auto & b = foo();        // b: const double&
volatile auto * c = bar();     // c: volatile float*

auto d = a;                    // d: double
auto & e = a;                  // e: const double &
auto f = c;                    // f: float *
volatile auto & g = c;         // g: volatile float * &

// 编译选项 :g++ -std=c++11 4-2-10.cpp
```

在代码清单 4-12 中，我们可以通过非 cv 限制的类型初始化一个 cv 限制的类型，如变量 a、b、c 所示。不过通过 auto 声明的变量 d、f 却无法带走 a 和 f 的常量性或者易失性。这里的例外还是引用，可以看出，声明为引用的变量 e、g 都保持了其引用的对象相同的属性（事实上，指针也是一样的）。

此外，跟其他的变量指示符一样，同一个赋值语句中，auto 可以用来声明多个变量的类型，不过这些变量的类型必须相同。如果这些变量的类型不相同，编译器则会报错。事实上，用 auto 来声明多个变量类型时，只有第一个变量用于 auto 的类型推导，然后推导出来的数据类型被作用于其他的变量。所以不允许这些变量的类型不相同，如代码清单 4-13 所示。

代码清单 4-13

```
auto x = 1, y = 2;          // x 和 y 的类型均为 int

// m 是一个指向 const int 类型变量的指针 , n 是一个 int 类型的变量
const auto* m = &x, n = 1;

auto i = 1, j = 3.14f;  // 编译失败

auto o = 1, &p = o, *q = &p;      // 从左向右推导

// 编译选项 :g++ -std=c++11 4-2-11.cpp -c
```

在代码清单 4-13 中，我们使用 auto 声明了两个类型相同变量 x 和 y，并用逗号进行分隔，这可以通过编译。而在声明变量 i 和 j 的时候，按照我们所说的第一变量用于推导类型的规则，那么由于 x 所推导出的类型是 int，那么对于变量 j 而言，其声明就变成了 int j = 3.14f，这无疑会导致精度的损失。而对于变量 m 和 n，就变得非常有趣，这里似乎是 auto 被替换成了 int，所以 m 是一个 int * 指针类型，而 n 只是一个 int 类型。同样的情况也发生在变量 o、p、q 上，这里 o 是一个类型为 int 的变量，p 是 o 的引用，而 q 是 p 的指针。auto 的类型推导按照从左往右，且类似于字面替换的方式进行。事实上，标准里称 auto 是一个将要推导出的类型的“占位符”（placeholder）。这样的规则无疑是直观而让人略感意外的。当然，为了不必要的繁琐记忆，程序员可以选择每一个 auto 变量的声明写成一行（有些观点也认为这是好的编程规范）。

此外，只要能够进行推导的地方，C++11 都为 auto 指定了详细的规则，保证编译器能够正确地推导出变量的类型。包括 C++11 新引入的初始化列表，以及 new，都可以使用 auto 关键字，如代码清单 4-14 所示。

代码清单 4-14

```
#include <initializer_list>

auto x = 1;
auto x1(1);

auto y {1};       // 使用初始化列表的 auto
auto z = new auto(1);    // 可以用于 new

// 编译选项 :g++ -std=c++11 -c 4-2-12.cpp
```

代码清单 4-14 中，auto 变量 y 的初始化使用了初始化列表，编译器可以保证 y 的类型推导为 int。而 z 指针所指向的堆变量在分配时依然选择让编译器对类型进行推导，同样的，编译器也能够保证这种方式下类型推导的正确性。

不过 auto 也不是万能的，受制于语法的二义性，或者是实现的困难性，auto 往往也会有使用上的限制。这些例外的情况都写在了代码清单 4-15 所示的例子当中。

代码清单 4-15

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

void fun(auto x =1){}  // 1: auto 函数参数，无法通过编译

struct str{
    auto var = 10;    // 2: auto 非静态成员变量，无法通过编译
};

int main() {
    char x[3];
    auto y = x;
    auto z[3] = x; // 3: auto 数组，无法通过编译

    // 4: auto 模板参数（实例化时），无法通过编译
    vector<auto> v = {1};
}

// 编译选项 :g++ -std=c++11 4-2-13.cpp
```

我们分别来看看代码清单 4-15 中的 4 种不能推导的情况。

1）对于函数 fun 来说，auto 不能是其形参类型。可能读者感觉对于 fun 来说，由于其有默认参数，所以应该推导 fun 形参 x 的类型为 int 型。但事实却无法符合大家的想象。因为 auto 是不能做形参的类型的。如果程序员需要泛型的参数，还是需要求助于模板。

2）对于结构体来说，非静态成员变量的类型不能是 auto 的。同样的，由于 var 定义了初始值，读者可能认为 auto 可以推导 str 成员 var 的类型为 int 的。但编译器阻止 auto 对结构体中的非静态成员进行推导，即使成员拥有初始值。

3）声明 auto 数组。我们可以看到，main 中的 x 是一个数组，y 的类型是可以推导的，而声明 auto z[3] 这样的数组同样会被编译器禁止。

4）在实例化模板的时候使用 auto 作为模板参数，如 main 中我们声明的 vector<auto> v。虽然读者可能认为这里一眼而知是 int 类型，但编译器却阻止了编译。

以上 4 种情况的特点基本相似，人为地观察很容易能够推导出 auto 所在位置应有的类型，但现有的 C++11 的标准还没有支持这样的使用方式。如果程序员遇到 auto 不够聪明的情况，不妨回头看看是否违背了以上一些规则。

此外，程序员还应该注意，由于为了避免和 C++98 中 auto 的含义发生混淆，C++11 只保

留 auto 作为类型指示符的用法，以下的语句在 C++98 和 C 语言中都是合法的，但在 C++11 中，编译器则会报错。

```
auto int i = 1;
```

总的来说，auto 在 C++11 中是相当关键的特性之一。我们之后还会在很多地方看到 auto，比如 4.4 节中的追踪返回类型的函数声明，以及 7.3 节中 lambda 与 auto 的配合使用等。（事实上，第 3 章中我们也使用过）。不过，如我们提到的，auto 只是 C++11 中类型推导体现的一部分。其余的，则会在 decltype 中得到体现。

4.3 decltype

☞ **类别：库作者**

4.3.1 typeid 与 decltype

我们在 4.2 节中曾经提到过静态类型和动态类型的区别。不过与 C 完全不支持动态类型不同的是，C++ 在 C++98 标准中就部分支持动态类型了。如读者能够想象的，C++98 对动态类型支持就是 C++ 中的运行时类型识别（RTTI）。

RTTI 的机制是为每个类型产生一个 type_info 类型的数据，程序员可以在程序中使用 typeid 随时查询一个变量的类型，typeid 就会返回变量相应的 type_info 数据。而 type_info 的 name 成员函数可以返回类型的名字。而在 C++11 中，又增加了 hash_code 这个成员函数，返回该类型唯一的哈希值，以供程序员对变量的类型随时进行比较。我们可以看看代码清单 4-16 所示的例子。

代码清单 4-16

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

class White{};
class Black{};

int main() {
    White a;
    Black b;

    cout << typeid(a).name() << endl;    // 5White
    cout << typeid(b).name() << endl;    // 5Black

    White c;

    bool a_b_sametype = (typeid(a).hash_code() == typeid(b).hash_code());
```

```
    bool a_c_sametype = (typeid(a).hash_code() == typeid(c).hash_code());

    cout << "Same type? " << endl;
    cout << "A and B? " << (int)a_b_sametype << endl;    // 0
    cout << "A and C? " << (int)a_c_sametype << endl;    // 1
}

// 编译选项 :g++ -std=c++11 4-3-1.cpp
```

这里我们定义了两个不同的类型 White 和 Black，以及其类型的变量 a 和 b。此外我们使用 typeid 返回类型的 type_info，并分别应用 name 打印类型的名字（5 这样的前缀是 g++ 这类编译器输出的名字，其他编译器可能会打印出其他的名字，这个标准并没有明确规定），应用 hash_code 进行类型的比较。在 RTTI 的支持下，程序员可以在一定程度上了解程序中类型的信息（这里可以注意一下，相比于 4.1.2 节中的 is_same 模板函数的成员类型 value 在编译时得到信息，hash_code 是运行时得到的信息）。

除了 typeid 外，RTTI 还包括了 C++ 中的 dynamic_cast 等特性。不过不得不提的是，由于 RTTI 会带来一些运行时的开销，所以一些编译器会让用户选择性地关闭该特性（比如 XL C/C++ 编译器的 -qnortti，GCC 的选项 -fno-rttion，或者微软编译器选项 /GR-）。而且很多时候，运行时才确定出类型对于程序员来说为时过晚，程序员更多需要的是在编译时期确定出类型（标准库中非常常见）。而通常程序员是要使用这样的类型而不是识别该类型，因此 RTTI 无法满足需求。

事实上，在 C++ 的发展中，类型推导是随着模板和泛型编程的广泛使用而引入的。在非泛型的编程中，我们不用对类型进行推导，因为任何表达式中变量的类型都是明确的，而运算、函数调用等也都有明确的返回类型。然而在泛型的编程中，类型成了未知数。我们可以回顾一下 4.2 节中代码清单 4-9 所示的例子，其中的模板函数 Sum 的参数的 t1 和 t2 类型都是不确定的，因此 t1 + t2 这个表达式将返回的类型也就不可由 Sum 的编写者确定。无疑，这样的状况会限制模板的使用范围和编写方式。而最好的解决办法就是让编译器辅助地进行类型推导。

在 decltype 产生之前，很多编译器的厂商都开发了自己的 C++ 语言扩展用于类型推导。比如 GCC 的 typeof 操作符就是其中的一种。C++11 则将这些类型推导手段进行了细致的考量，最终标准化为 auto 以及 decltype。与 auto 类似地，decltype 也能进行类型推导，不过两者的使用方式却有一定的区别。我们可以看代码清单 4-17 所示的这个简单的例子。

代码清单 4-17

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

```
int main() {
    int i;
    decltype(i) j = 0;
    cout << typeid(j).name() << endl;    // 打印出 "i", g++ 表示 int

    float a;
    double b;
    decltype(a + b) c;
    cout << typeid(c).name() << endl;    // 打印出 "d", g++ 表示 double
}

// 编译选项 :g++ -std=c++11 4-3-2.cpp
```

在代码清单 4-17 中，我们看到变量 j 的类型由 decltype(i) 进行声明，表示 j 的类型跟 i 相同（或者准确地说，跟 i 这个表达式返回的类型相同）。而 c 的类型则跟 (a + b) 这个表达式返回的类型相同。而由于 a + b 加法表达式返回的类型为 double（a 会被扩展为 double 类型与 b 相加），所以 c 的类型被 decltype 推导为 double。

从这个例子中可以看到，decltype 的类型推导并不是像 auto 一样是从变量声明的初始化表达式获得变量的类型，decltype 总是以一个普通的表达式为参数，返回该表达式的类型。而与 auto 相同的是，作为一个类型指示符，decltype 可以将获得的类型来定义另外一个变量。与 auto 相同，decltype 类型推导也是在编译时进行的。

4.3.2 decltype 的应用

在 C++11 中，使用 decltype 推导类型是非常常见的事情。比较典型的就是 decltype 与 typdef/using 的合用。在 C++11 的头文件中，我们常能看以下这样的代码：

```
using size_t = decltype(sizeof(0));
using ptrdiff_t = decltype((int*)0 - (int*)0);
using nullptr_t = decltype(nullptr);
```

这里 size_t 以及 ptrdiff_t 还有 nullptr_t（参见 7.1 节）都是由 decltype 推导出的类型。这种定义方式非常有意思。在一些常量、基本类型、运算符、操作符等都已经被定义好的情况下，类型可以按照规则被推导出。而使用 using，就可以为这些类型取名。这就颠覆了之前类型拓展需要将扩展类型“映射”到基本类型的常规做法。

除此之外，decltype 在某些场景下，可以极大地增加代码的可读性。比如代码清单 4-18 所示的例子。

代码清单 4-18

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

```
int main() {
    vector<int> vec;
    typedef decltype(vec.begin()) vectype;

    for (vectype i = vec.begin(); i < vec.end(); i++) {
        // 做一些事情
    }
    for (decltype(vec)::iterator i = vec.begin(); i < vec.end(); i++) {
        // 做一些事情
    }
}

// 编译选项 :g++ -std=c++11 4-3-3.cpp
```

在代码清单 4-18 中，我们定义了 vector 的 iterator 的类型。这个类型还可以在 main 函数中重用。当我们遇到一些具有复杂类型的变量或表达式时，就可以利用 decltype 和 typedef/using 的组合来将其转化为一个简单的表达式，这样在以后的代码写作中可以提高可读性和可维护性。此外我们可以看到 decltype(vec)::iterator 这样的灵活用法，这看起来跟 auto 非常类似，也类似于是一种“占位符”式的替代。

在 C++ 中，我们有时会遇到匿名的类型，而拥有了 decltype 这个利器之后，重用匿名类型也并非难事。我们可以看看代码清单 4-19 所示的例子。

代码清单 4-19

```
enum class{K1, K2, K3}anon_e;    // 匿名的强类型枚举

union {
    decltype(anon_e) key;
    char* name;
}anon_u;     // 匿名的 union 联合体

struct {
    int d;
    decltype(anon_u) id;
}anon_s[100];    // 匿名的 struct 数组

int main() {
    decltype(anon_s) as;
    as[0].id.key = decltype(anon_e)::K1;    // 引用匿名强类型枚举中的值
}

// 编译选项 :g++ -std=c++11 4-3-4.cpp
```

这里我们使用了 3 种不同的匿名类型：匿名的强类型枚举 anon_e（请参见 5.1 节）、匿名的联合体 anon_u，以及匿名的结构体数组 anon_s。可以看到，只要通过匿名类型的变量名 anon_e、anon_u，以及 anon_s，decltype 可以推导其类型并且进行重用。这些都是以前 C++

代码所做不到的。事实上，在一些C代码中，匿名的结构体和联合体并不少见。不过匿名一般都有匿名理由，一般程序员都不希望匿名后的类型被重用。这里的decltype只是提供了一种语法上的可能。

进一步地，有了decltype，我们可以适当扩大模板泛型的能力。还是以代码清单4-9为例，如果我们稍微改变一下函数模板的接口，该模板将适用于更大的范围。我们来看看代码清单4-20中经过改进的例子。

代码清单4-20

```
// s的类型被声明为decltype(t1 + t2)
template<typename T1, typename T2>
void Sum(T1 & t1, T2 & t2, decltype(t1 + t2) & s) {
    s = t1 + t2;
}

int main() {
    int a = 3;
    long b = 5;
    float c = 1.0f, d = 2.3f;

    long e;
    float f;
    Sum(a, b, e);    // s的类型被推导为long
    Sum(c, d, f);    // s的类型被推导为float
}

// 编译选项 :g++ -std=c++11 4-3-5.cpp
```

相比于代码清单4-9的例子，代码清单4-20的Sum函数模板增加了类型为decltype(t1+t2)的s作为参数，而函数本身不返回任何值。这样一来，Sum的适用范围增加，因为其返回的类型不再是代码清单4-9中单一的double类型，而是根据t1 + t2推导而来的类型。不过这里还是有一定的限制，我们可以看到返回值的类型必须一开始就被指定，程序员必须清楚Sum运算的结果使用什么样的类型来存储是合适的，这在一些泛型编程中依然不能满足要求。解决的方法是结合decltype与auto关键字，使用追踪返回类型的函数定义来使得编译器对函数返回值进行推导。我们会在4.4节中看到具体的细节（事实上，decltype一个最大的用途就是用在追踪返回类型的函数中）。

在代码清单4-20中模板定义虽然存在一些限制，但也基本是可以广泛使用的。但是不得不提的是，某些情况下，模板库的使用人员可能认为一些自然而简单的数据结构，比如数组，也是可以被模板类所包括的。不过很明显，如果t1和t2是两个数组，t1 + t2不会是合法的表达式。为了避免不必要的误解，模板库的开发人员应该为这些特殊的情况提供其他的版本，如代码清单4-21所示。

代码清单 4-21

```
template<typename T1, typename T2>
void Sum(T1 & t1, T2 & t2, decltype(t1 + t2) & s) {
    s = t1 + t2;
}

void Sum(int a[], int b[], int c[]){
    // 数组版本
}

int main() {
    int a[5], b[5], c[5];
    Sum(a, b, c);    // 选择数组版本

    int d, e, f;
    Sum(d, e, f);    // 选择模板的实例化版本
}

// 编译选项 :g++ -std=c++11 4-3-6.cpp
```

在代码清单 4-21 中，由于声明了数组版本 Sum，编译器在编译 Sum(a, b, c) 的时候，会优先选择数组版本，而编译 Sum(d, e, f) 的时候，依然会对应到模板的实例化版本。这就能够保证 Sum 模板函数最大的可用性（不过这里的数组版本似乎做不了什么事情，因为数组长度丢失了）。

我们在实例化一些模板的时候，decltype 也可以起到一些作用，我们可以看看代码清单 4-22 所示的例子。

代码清单 4-22

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

int hash(char*);

map<char*, decltype(hash)> dict_key;     // 无法通过编译
map<char*, decltype(hash(nullptr))> dict_key1;

// 编译选项 :g++ -c -std=c++11 4-3-7.cpp
```

在代码清单 4-22 中，我们实例化了标准库中的 map 模板。因为该 map 是为了存储字符串以及与其对应哈希值的，因此我们可以通过 decltype(hash(nullptr)) 来确定哈希值的类型。这样的定义非常直观，但是程序员必须要注意的是，decltype 只能接受表达式做参数，像函数名做参数的表达式 decltype(hash) 是无法通过编译的。

事实上，decltype 在 C++11 的标准库中也有一些应用，一些标准库的实现也会依赖于

decltype 的类型推导。一个典型的例子是基于 decltype 的模板类 result_of，其作用是推导函数的返回类型。我们可以看一下应用的实例，如代码清单 4-23 所示。

代码清单 4-23

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

typedef double (*func)();

int main() {
    result_of<func()>::type f;    // 由 func() 推导其结果类型
}

// 编译选项 :g++ -std=c++11 4-3-8.cpp
```

这里 f 的类型最终被推导为 double，而 result_of 并没有真正调用 func() 这个函数，这一切都是因为底层的实现使用了 decltype。result_of 的一个可能的实现方式如下：

```
template<class>
struct result_of;

template<class F, class... ArgTypes>
struct result_of<F(ArgTypes...)>
{
    typedef decltype(
                std::declval<F>()(std::declval<ArgTypes>()...)
                ) type;
};
```

请读者忽略 declval [⊖]，这里标准库将 decltype 作用于函数调用上（使用了变长函数模板），并将函数调用表达式返回的类型 typedef 为一个名为 type 的类型。这样一来，代码清单 4-23 中的 result_of<func()>::type 就会被 decltype 推导为 double。

4.3.3 decltype 推导四规则

作为 auto 的伙伴，decltype 在 C++11 中也非常重要。不过跟 auto 一样，由于应用广泛，所以使用 decltype 也有很多的细则条款需要注意。很多时候，用户会发现 decltype 的行为并不如预期，那么下面的规则可能会更好地解释这些“不如预期”的编译器行为。

大多数时候，decltype 的使用看起来非常平易近人，但是有时我们也会落入一些令人疑惑的陷阱。最典型的就是代码清单 4-24 所示的这个例子。

⊖ 实际是 STL 中的一种语法技巧，更多的内容可以查阅一些在线文档，如 http://en.cppreference.com/w/cpp/utility/declval。

代码清单 4-24

```
int i;
decltype(i) a;        // a: int
decltype((i)) b;      // b: int &, 无法编译通过

// 编译选项 :g++ -std=c++11 4-3-9.cpp
```

我们在编译代码清单 4-24 的时候，会惊奇地发现，decltype((i)) b; 这样的语句编译不过。编译器会提示 b 是一个引用，但没有被赋初值。而 decltype(i) a; 这一句却能通过编译，因为其类型被如预期地推导为 int。

这种问题显得非常诡异，单单多了一对圆括号，decltype 所推导出的类型居然发生了变化。事实上，C++11 中 decltype 推导返回类型的规则比我们想象的复杂。具体地，当程序员用 decltype(e) 来获取类型时，编译器将依序判断以下四规则：

1）如果 e 是一个没有带括号的标记符表达式（id-expression）或者类成员访问表达式，那么 decltype(e) 就是 e 所命名的实体的类型。此外，如果 e 是一个被重载的函数，则会导致编译时错误。

2）否则，假设 e 的类型是 T，如果 e 是一个将亡值 (xvalue)，那么 decltype(e) 为 T&&。

3）否则，假设 e 的类型是 T，如果 e 是一个左值，则 decltype(e) 为 T&。

4）否则，假设 e 的类型是 T，则 decltype(e) 为 T。

这里我们要解释一下标记符表达式（id-expression）。基本上，所有除去关键字、字面量等编译器需要使用的标记之外的程序员自定义的标记（token）都可以是标记符（identifier）。而单个标记符对应的表达式就是标记符表达式。比如程序员定义了：

```
int arr[4];
```

那么 arr 是一个标记符表达式，而 arr[3] + 0, arr[3] 等，则都不是标记符表达式。

我们再回到代码清单 4-24，并结合 decltype 类型推导的规则，就可以知道，decltype(i) a; 使用了推导规则 1——因为 i 是一个标记符表达式，所以类型被推导为 int。而 decltype((i)) b; 中，由于 (i) 不是一个标记符表达式，但却是一个左值表达式（可以有具名的地址），因此，按照 decltype 推导规则 3，其类型应该是一个 int 的引用。

上面的规则看起来非常复杂，但事实上，在实际应用中，decltype 类型推导规则中最容易引起迷惑的只有规则 1 和规则 3。我们可以通过代码清单 4-25 所示的这个例子再加深一下理解。

代码清单 4-25

```
int i = 4;
```

```
int arr[5] = {0};
int *ptr = arr;

struct S { double d; } s;

void Overloaded(int);
void Overloaded(char);        // 重载的函数

int && RvalRef();

const bool Func(int);

// 规则 1：单个标记符表达式以及访问类成员，推导为本类型
decltype(arr) var1;              // int[5]，标记符表达式
decltype(ptr) var2;              // int*，标记符表达式
decltype(s.d) var4;              // double，成员访问表达式
decltype(Overloaded) var5;       // 无法通过编译，是个重载的函数

// 规则 2：将亡值，推导为类型的右值引用
decltype(RvalRef()) var6 = 1;   // int&&

// 规则 3：左值，推导为类型的引用
decltype(true ? i : i) var7 = i;     // int&，三元运算符，这里返回一个 i 的左值
decltype((i)) var8 = i;              // int&，带圆括号的左值
decltype(++i) var9 = i;              // int&，++i 返回 i 的左值
decltype(arr[3]) var10 = i;          // int& [] 操作返回左值
decltype(*ptr)  var11 = i;           // int& * 操作返回左值
decltype("lval") var12 = "lval";     // const char(&)[9]，字符串字面常量为左值

// 规则 4：以上都不是，推导为本类型
decltype(1) var13;                 // int，除字符串外字面常量为右值
decltype(i++) var14;               // int，i++ 返回右值
decltype((Func(1))) var15;         // const bool，圆括号可以忽略

// 编译选项 :g++ -std=c++11 -c 4-3-10.cpp
```

代码清单 4-25 中我们将四种规则的例子都列了出来。可以看到，规则 1 不但适用于基本数据类型，还适用于指针、数组、结构体，甚至函数类型的推导，事实上，规则 1 在 decltyp 类型推导中运用的最为广泛。而规则 2 则比较简单，基本上符合程序员的想象。

规则 3 其实是一个左值规则。decltype 的参数不是标志表达式或者类成员访问表达式，且参数都为左值，推导出的类型均为左值引用。规则 4 则是适用于以上都不适用者。我们这里看到了 ++i 和 i++ 在左右值上的区别，以及字符串字面常量 lval、非字符串字面常量 1 在左右值间的区别。

看过这么多规则，读者可能觉得过于复杂，但事实上，如同我们之前提到的，引起麻烦的只是规则 3 带来的左值引用的推导。一个简单的能够让编译器提示的方法是，如果使用

decltype 定义变量，那么先声明这个变量，再在其他语句里对其进行初始化。这样一来，由于左值引用总是需要初始化的，编译器会报错提示。另外一些时候，C++11 标准库中添加的模板类 is_lvalue_reference，可以帮助程序员进行一些推导结果的识别。我们看看代码清单 4-26 所示的例子。

代码清单 4-26

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

int i = 4;
int arr[5] = {0};
int *ptr = arr;

int && RvalRef();

int main(){
    cout << is_rvalue_reference<decltype(RvalRef())>::value << endl;     // 1

    cout << is_lvalue_reference<decltype(true ? i : i)>::value << endl; // 1
    cout << is_lvalue_reference<decltype((i))>::value << endl;          // 1
    cout << is_lvalue_reference<decltype(++i)>::value << endl;          // 1
    cout << is_lvalue_reference<decltype(arr[3])>::value << endl;       // 1
    cout << is_lvalue_reference<decltype(*ptr)>::value << endl;         // 1
    cout << is_lvalue_reference<decltype("lval")>::value << endl;       // 1

    cout << is_lvalue_reference<decltype(i++)>::value << endl;          // 0
    cout << is_rvalue_reference<decltype(i++)>::value << endl;          // 0
}

// 编译选项 :g++ -std=c++11 4-3-11.cpp
```

代码清单 4-26 中，我们使用了模板类 is_lvalue_reference 的成员 value 来查看 decltype 的效果（1 表示是左值引用，0 则反之）。如我们所见，代码清单 4-26 中凡是符合规则 3 的，都会被推导为左值引用。如果程序员在程序的书写中不是非常确定 decltype 是否将类型推导为左值引用，也可以通过这样的小实验来辅助确定。这里我们还使用了模板函数 is_rvalue_reference，同样，程序员也可以通过它来确定 decltype 是否推导出了右值引用。

4.3.4 cv 限制符的继承与冗余的符号

与 auto 类型推导时不能“带走”cv 限制符不同，decltype 是能够“带走”表达式的 cv 限制符的。不过，如果对象的定义中有 const 或 volatile 限制符，使用 decltype 进行推导时，其成员不会继承 const 或 volatile 限制符。我们可以看看如代码清单 4-27 所示的例子。

代码清单 4-27

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

const int ic = 0;
volatile int iv;

struct S { int i; };

const S a = {0};
volatile S b;
volatile S* p = &b;

int main() {
    cout << is_const<decltype(ic)>::value << endl;       // 1
    cout << is_volatile<decltype(iv)>::value << endl;    // 1

    cout << is_const<decltype(a)>::value << endl;        // 1
    cout << is_volatile<decltype(b)>::value << endl;     // 1

    cout << is_const<decltype(a.i)>::value << endl;      // 0, 成员不是const
    cout << is_volatile<decltype(p->i)>::value << endl; // 0, 成员不是volatile
}

// 编译选项:g++ -std=c++11 4-3-12.cpp
```

代码清单 4-27 的例子中，我们使用了 C++ 库提供的 is_const 和 is_volatile 来查看类型是否是常量或者易失的。可以看到，结构体变量 a、b 和结构体指针 p 的 cv 限制符并没有出现在其成员的 decltype 类型推导结果中。

而与 auto 相同的，decltype 从表达式推导出类型后，进行类型定义时，也会允许一些冗余的符号。比如 cv 限制符以及引用符号 &，通常情况下，如果推导出的类型已经有了这些属性，冗余的符号则会被忽略，如代码清单 4-28 所示。

代码清单 4-28

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

int i = 1;
int & j = i;
int * p = &i;
const int k = 1;

int main() {
    decltype(i) & var1 = i;
```

```
    decltype(j) & var2 = i;      // 冗余的 &，被忽略

    cout << is_lvalue_reference<decltype(var1)>::value << endl;// 1，是左值引用

    cout << is_rvalue_reference<decltype(var2)>::value << endl;// 0，不是右值引用
    cout << is_lvalue_reference<decltype(var2)>::value << endl;// 1，是左值引用

    decltype(p)* var3 = &i;      // 无法通过编译
    decltype(p)* var3 = &p;      // var3 的类型是 int**

    auto* v3 = p;                // v3 的类型是 int*
    v3 = &i;

    const decltype(k) var4 = 1; // 冗余的 const，被忽略
}

// 编译选项 :g++ -std=c++11 4-3-13.cpp
```

在代码清单 4-28 中，我们定义了类型为 decltype(i) & 的变量 var1，以及类型为 decltype(j) & 的变量 var2。由于 i 的类型为 int，所以这里的引用符号保证 var1 成为一个 int& 引用类型。而由于 j 本来就是一个 int & 的引用类型，所以 decltype 之后的 & 成为了冗余符号，会被编译器忽略，因此 j 的类型依然是 int &。

这里特别要注意的是 decltype(p)* 的情况。可以看到，在定义 var3 变量的时候，由于 p 的类型是 int*，因此 var3 被定义为了 int** 类型。这跟 auto 声明中，* 也可以是冗余的不同。在 decltype 后的 * 号，并不会被编译器忽略。

此外我们也可以看到，var4 中 const 可以被冗余的声明，但会被编译器忽略，同样的情况也会发生在 volatile 限制符上。

总的说来，decltype 算得上是 C++11 中类型推导使用方式上最灵活的一种。虽然看起来它的推导规则比较复杂，有的时候跟 auto 推导结果还略有不同，但大多数时候，我们发现使用 decltype 还是自然而亲切的。一些细则的区别，读者可以在使用时遇到问题再返回查验。而下面的追踪返回类型的函数定义，则将融合 auto、decltype，将 C++11 中的泛型能力提升到更高的水平。

4.4 追踪返回类型

☞ **类别：库作者**

4.4.1 追踪返回类型的引入

如我们在 4.2 节与 4.3 节中反复提到的，追踪返回类型配合 auto 与 decltype 会真正释放 C++11 中泛型编程的能力。

在C++98中，如果一个函数模板的返回类型依赖于实际的入口参数类型，那么该返回类型在模板实例化之前可能都无法确定，这样的话我们在定义该函数模板时就会遇到麻烦。可以回想一下代码清单4-9的例子，由于Sum模板函数的两个参数t1与t2的类型没有确定，所以我们只能简单地设置结果s为double类型并返回。这就限制了Sum的使用范围（大概只能用于数值不算太大的算术运算）。而在代码清单4-20中，我们改进了Sum模板函数，通过增加decltype(t1+t2)的参数的方式来返回泛型的值。这样做虽然扩大了Sum的适用范围，但改变了Sum的使用方式，在一些情况下，也是不可以接受的。而且由于程序员必须预先知道返回的类型，其使用上的灵活性也就打了一些折扣。

那么，最为直观的解决方式就是对返回类型进行类型推导。而最为直观的书写方式如下所示：

```
template<typename T1, typename T2>
decltype(t1 + t2) Sum(T1 & t1, T2 & t2) {
    return t1 + t2;
}
```

这样的写法虽然看似不错，不过对编译器来说有些小问题。编译器在推导decltype(t1 + t2)时的，表达式中的t1和t2都未声明（虽然它们近在咫尺，编译器却只会从左往右地读入符号）。按照C/C++编译器的规则，变量使用前必须已经声明，因此，为了解决这个问题，C++11引入新语法——追踪返回类型，来声明和定义这样的函数。

```
template<typename T1, typename T2>
auto Sum(T1 & t1, T2 & t2) -> decltype(t1 + t2){
    return t1 + t2;
}
```

如上面的写法所示，我们把函数的返回值移至参数声明之后，复合符号-> decltype(t1 + t2)被称为追踪返回类型。而原本函数返回值的位置由auto关键字占据。这样，我们就可以让编译器来推导Sum函数模板的返回类型了。而auto占位符和->return_type也就是构成追踪返回类型函数的两个基本元素。

4.4.2 使用追踪返回类型的函数

追踪返回类型的函数和普通函数的声明最大的区别在于返回类型的后置。在一般情况下，普通函数的声明方式会明显简单于最终返回类型。比如：

```
int func(char* a, int b);
```

这样的书写会比下面的书写少上不少。

```
auto func(char*a, int b) -> int;
```

不过有的时候，追踪返回类型声明的函数也会带给大家一些意外，比如代码清单4-29所

示的这个例子。

代码清单 4-29

```
class OuterType{
    struct InnerType { int i; };

    InnerType GetInner();
    InnerType it;
};

// 可以不写 OuterType::InnerType
auto OuterType::GetInner() -> InnerType {
    return it;
}

// 编译选项 :g++ -std=c++11 4-4-1.cpp
```

在代码清单 4-29 中，可以看到我们使用最终返回类型的时候，InnerType 不必写明其作用域。这对于讨厌写很长作用域的程序员来说，也算得上是一个好消息。

如我们刚才提到的，返回类型后置，使模板中的一些类型推导就成为了可能。我们可以看看代码清单 4-30 所示的使用追踪返回类型的例子。

代码清单 4-30

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

template<typename T1, typename T2>
auto Sum(const T1 & t1, const T2 & t2) -> decltype(t1 + t2){
    return t1 + t2;
}

template <typename T1, typename T2>
auto Mul(const T1 & t1, const T2 & t2) -> decltype(t1 * t2){
    return t1 * t2;
}

int main() {
    auto a = 3;
    auto b = 4L;
    auto pi = 3.14;

    auto c = Mul(Sum(a, b), pi);
    cout << c << endl;  // 21.98
}

// 编译选项 :g++ -std=c++11 4-4-2.cpp
```

在代码清单 4-30 的例子中，我们定义了两个模板函数 Sum 和 Mul，它们的参数的类型和返回值都在实例化时决定。而由于 main 函数中还使用了 auto，整个例子中没有看到一个“具体”的类型声明。事实上，这段代码尤其是主函数，看起来有点像是一个动态类型语言的代码，而不像是一个有着严格静态类型的 C++ 的代码。当然，这一切都要归功于类型推导帮助下的泛型编程。程序员在编写代码时无需关心任何时段的类型选择，编译器会合理地进行推导，而简单程序的书写也由此得到了极大的简化。

除了解决以上所描述的问题，追踪返回类型的另一个优势是简化函数的定义，提高代码的可读性。这种情况常见于函数指针中。我们可以看一下代码清单 4-31 所示的例子。

代码清单 4-31

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

// 有的时候，你会发现这是面试题
int (*(*pf())())() {
    return nullptr;
}

// auto (*)() -> int(*) () 一个返回函数指针的函数（假设为 a 函数）
// auto pf1() -> auto (*)() -> int (*)() 一个返回 a 函数的指针的函数
auto pf1() -> auto (*)() -> int (*)() {
    return nullptr;
}

int main() {
    cout << is_same<decltype(pf), decltype(pf1)>::value << endl;    // 1
}

// 编译选项 :g++ -std=c++11 4-4-3.cpp
```

在代码清单 4-31 中，定义了两个类型完全一样的函数 pf 和 pf1。其返回的都是一个函数指针。而该函数指针又指向一个返回函数指针的函数。这一点通过 is_same 的成员 value 已经能够确定了（参见 4.1.1）。而仔细看一看函数类型的声明，可以发现老式的声明法可读性非常差。而追踪返回类型只需要依照从右向左的方式，就可以将嵌套的声明解析出来。这大大提高了嵌套函数这类代码的可读性。

除此之外，追踪返回类型也被广泛地应用在转发函数中，如代码清单 4-32 所示。

代码清单 4-32

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

```
double foo(int a) {
    return (double)a + 0.1;
}

int foo(double b) {
    return (int)b;
}

template <class T>
auto Forward(T t) -> decltype(foo(t)){
    return foo(t);
}

int main(){
    cout << Forward(2) << endl;      // 2.1
    cout << Forward(0.5) << endl;    // 0
}

// 编译选项 :g++ -std=c++11 4-4-4.cpp
```

代码清单 4-32 中，我们可以看到，由于使用了追踪返回类型，可以实现参数和返回类型不同时的转发。

追踪返回类型还可以用在函数指针中，其声明方式与追踪返回类型的函数比起来，并没有太大的区别。比如：

```
auto (*fp)() -> int;
```

和

```
int (*fp)();
```

的函数指针声明是等价的。同样的情况也适用于函数引用，比如：

```
auto (&fr)() -> int;
```

和

```
int (&fr)();
```

的声明也是等价的。

除了以上所描述的函数模板、普通函数、函数指针、函数引用以外，追踪返回类型还可以用在结构或类的成员函数、类模板的成员函数里，其方法大同小异，这里不一一举例了。另外，没有返回值的函数也可以被声明为追踪返回类型，程序员只需要将返回类型声明为 void 即可。

4.5 基于范围的 for 循环

☞ **类别：所有人**

在 C++98 标准中，如果要遍历一个数组，通常会需要代码清单 4-33 所示的代码。

代码清单 4-33

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

int main() {
    int arr[5] = { 1, 2, 3, 4, 5};
    int * p;
    for (p = arr; p < arr + sizeof(arr)/sizeof(arr[0]); ++p){
        *p *= 2;
    }
    for (p = arr; p < arr + sizeof(arr)/sizeof(arr[0]); ++p){
        cout << *p << '\t';
    }
}

// 编译选项 :g++ 4-5-1.cpp
```

代码清单 4-33 中，我们使用了指针 p 来遍历数组 arr 中的内容，两个循环分别完成了每个元素自乘以 2 和打印工作。而 C++ 的标准模板库中，我们还可以找到形如 for_each 的模板函数。如果我们使用 for_each 来完成代码清单 4-33 中的工作，代码看起来会是代码清单 4-34 所示的样子。

代码清单 4-34

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

int action1(int & e){ e *= 2; }
int action2(int & e){ cout << e << '\t'; }

int main() {
    int arr[5] = { 1, 2, 3, 4, 5};
    for_each(arr, arr + sizeof(arr)/sizeof(arr[0]), action1);
    for_each(arr, arr + sizeof(arr)/sizeof(arr[0]), action2);
}

// 编译选项 :g++ -std=c++11 -c
```

for_each 使用了迭代器的概念，其迭代器就是指针。由于迭代器内含了自增操作的概

念，所以如代码清单 4-33 中的 ++p 操作则可以不写在 for_each 循环中了。不过无论是代码清单 4-33 还是代码清单 4-34，都需要告诉循环体其界限的范围，即 arr 到 arr + sizeof(arr)/sizeof(arr[0]) 之间，才能按元素执行操作。

事实上，循环的“自动范围”这个问题，在很多语言中已经实现了。我们可以看看 bash 中 for 循环的使用方法。

```
for i in '1 2 3 4 5' ;
do
    $i = `expr $i + $i`;
echo $i;
done
```

上面的 bash 完成了与代码清单 4-33 及代码清单 4-34 一样的功能，不过语法上，bash 使用了 for ... in 的方式，因此循环的范围是“自说明”的，是在‘1 2 3 4 5’这样的范围中完成元素操作的。很多时候，对于一个有范围的集合而言，由程序员来说明循环的范围是多余的，也是容易犯错误的。而 C++11 也引入了基于范围的 for 循环，就可以很好地解决了这个问题。

我们可以看一下基于范围的 for 循环改写的例子，如代码清单 4-35 所示。

代码清单 4-35

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

int main() {
    int arr[5] = { 1, 2, 3, 4, 5 };
    for (int & e: arr)
        e *= 2;

    for (int & e: arr)
        cout << e << '\t';
}

// 编译选项 :g++ -std=c++11 4-5-3.cpp
```

代码清单 4-35 就是一个基于范围的 for 循环的实例。for 循环后的括号由冒号“：”分为两部分，第一部分是范围内用于迭代的变量，第二部分则表示将被迭代的范围。在代码清单 4-35 这个具体的例子当中，表示的是在数组 arr 中用迭代器 e 进行遍历。这样一来，遍历数组和 STL 容器就非常容易了。

在代码清单 4-35 中，基于范围的 for 循环中迭代的变量采用了引用的形式，如果迭代变量的值在循环中不会被修改，那我们完全可以不用引用的方式来做迭代变量。比如上例中的第二个基于范围的 for 循环可以被改为：

```
for (int e: arr)
    cout << e << '\t';
```

代码依然可以很好地工作。当然，如果结合之前讲过的 auto 类型指示符，循环会显得更简练。

```
for (auto e: arr)
    cout << e << '\t';
```

基于范围的 for 循环跟普通循环是一样的，可以用 continue 语句来跳过循环的本次迭代，而用 break 语句来跳出整个循环。

值得指出的是，是否能够使用基于范围的 for 循环，必须依赖于一些条件。首先，就是 for 循环迭代的范围是可确定的。对于类来说，如果该类有 begin 和 end 函数，那么 begin 和 end 之间就是 for 循环迭代的范围。对于数组而言，就是数组的第一个和最后一个元素间的范围。其次，基于范围的 for 循环还要求迭代的对象实现 ++ 和 == 等操作符。对于标准库中的容器，如 string、array、vector、deque、list、queue、map、set 等，不会有问题，因为标准库总是保证其容器定义了相关的操作。普通的已知长度的数组也不会有问题。而用户自己写的类，则需要自行提供相关操作。

相反，如果我们数组大小不能确定的话，是不能够使用基于范围的 for 循环的，比如代码清单 4-36 所示的用法，就会导致编译时的错误。

代码清单 4-36

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

int func(int a[]) {

    for (auto e: a)
        cout << e;
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    func(arr);
}

// 编译选项 :g++ -std=c++11 4-5-4.cpp
```

上述代码会报错，因为作为参数传递而来的数组 a 的范围不能确定，因此也就不能使用基于范围循环 for 循环对其进行迭代的操作。

另外一点，习惯了使用迭代器的 C++ 程序员可能需要注意，就是基于范围的循环使用在标准库的容器中时，如果使用 auto 来声明迭代的对象的话，那么这个对象不会是迭代器对

象。代码清单 4-37 所示的这个简单的例子可以说明这一情况。

代码清单 4-37

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

int main() {
    vector<int> v = {1, 2, 3, 4, 5};
    for (auto i = v.begin(); i != v.end(); ++i)
        cout << *i << endl;      // i 是迭代器对象

    for (auto e: v)
        cout << e << endl;       // e 是解引用后的对象
}

// 编译选项 :g++ -std=c++11 4-5-5.cpp
```

读者只需要注意 e 和 *i 的区别就可以了。

4.6 本章小结

在本章里，介绍了 C++11 四个讨人喜欢的新特性，它们的特色非常鲜明，都能够减少代码的书写，或加强代码的可读性。

首先，我们看到 C++11 中解决了双右尖括号的语法问题的小改进。相比于 C++98 中，模板实例化时右尖括号间必须空格的“奇怪”规定，C++11 可以说采取了更加平易近人的态度，使得这一规则不再需要。

其次，我们可以看到 C++11 中关于类型推导的巨大改进。虽然 auto、decltype，以及追踪返回类型的函数声明，它们的由来都可以追溯到 C++ 使用模板进行泛型编程上，但从实际效果上看，由于有了类型推导，整个 C++ 程序的书写的便利性被极大地提高了。相应地，代码可读性也大大改善。可以说，类型推导不仅提高了模板库的泛型能力，导致 C++11 风格下的编程跟以前的 C++98 风格下的编程有了改变，而且在我们的范例中看到了全部倚仗于类型推导，没有一个“明确”类型的 C++ 代码，这对于一个静态类型的、有着长久历史渊源的语言而言，几乎是不可想象的。但是在 C++11 中，这种友好的编程方式已经得到了良好的支持。虽然深入语言细节的时候，我们可能发现一些推导的规则依然复杂，但对于 90% 以上的普通应用，类型推导已经做得足够好用。如果要在 C++11 中挑选最好的新特性的话，类型推导无疑会是非常有力的竞争者。

再者，我们看到了基于范围的 for 循环。结合 auto 关键字，程序员只需要知道“我在迭代地访问每个元素”即可，而再也不必关心范围、如何迭代访问等细节。这比以前标准库的

for_each 做得更加出色。虽然基本上基于范围的 for 循环没有任何灵活性可言，但将常做的事情做得更快更好，也往往是程序员最大的需求。

总的来说，以上新特性对于新手来说，非常易学，对于老兵而言，非常好用。无论对什么水平的编程者来说，总可以从使用这些特性当中获得一些益处。在后面的章节里，我们还会继续看到这些特性的身影（就如在前面的章节里一样）。