第⑦章

为改变思考方式而改变

如我们提到过的，C++ 是一门成熟的语言，语言的核心部分的改变通常都遵从着一贯的设计思想。不过这并不意味着 C++ 会墨守成规，在 C++11 中，我们还是会看到一些新元素。这些新鲜出炉的元素可能会带来一些习惯上的改变，不过权衡之下，可能这样的改变是值得的。比如 lambda 就是一个典型的例子。

7.1 指针空值——nullptr

☞ 类别：所有人

7.1.1 指针空值：从 0 到 NULL，再到 nullptr

在良好的 C++ 编程习惯中，声明一个变量的同时，总是需要记得在合适的代码位置将其初始化。对于指针类型的变量，这一点尤其应当注意。未初始化的悬挂指针通常会是一些难于调试的用户程序的错误根源。

典型的初始化指针是将其指向一个“空”的位置，比如 0。由于大多数计算机系统不允许用户程序写地址为 0 的内存空间，倘若程序无意中对该指针所指地址赋值，通常在运行时就会导致程序退出。虽然程序退出并非什么好事，但这样一来错误也容易被程序员找到。因此在大多数的代码中，我们常常能看见指针初始化的语法如下：

```
int * my_ptr = 0;
```

或者是使用 NULL：

```
int * my_ptr = NULL;
```

一般情况下，NULL 是一个宏定义。在传统的 C 头文件（stddef.h) 里我们可以找到如下代码：

```
#undef NULL
#if defined(__cplusplus)
#define NULL 0
#else
#define NULL ((void *)0)
#endif
```

可以看到，NULL 可能被定义为字面常量 0，或者是定义为无类型指针（void *）常量。不过无论采用什么样的定义，我们在使用空值的指针时，都不可避免地会遇到一些麻烦。让我们先看一个关于函数重载的例子。这个例子我们引用自 C++11 标准关于 nullptr 的提案，并进行了少许修改，具体如代码清单 7-1 所示。

代码清单 7-1

```
#include <stdio.h>

void f(char* c) {
    printf("invoke f(char*)\n");
}

void f(int i) {
    printf("invoke f(int)\n");
}

int main() {
    f(0);
    f(NULL); // 注意：如用 gcc 编译，NULL 转化为内部标识 __null，该语句会编译失败
    f((char*)0);
}

// 编译选项 :xlC -+ 7-1-1.cpp
```

在代码清单 7-1 所示的例子当中，用户重载了 f 函数，并且试图使用 f(NULL) 来调用指针的版本。不过很可惜，当使用 XLC 编译器编译以上语句并运行时，会得到以下结果：

```
invoke f(int)
invoke f(int)
invoke f(char*)
```

在这里，XLC 编译器采用了 stddef.h 头文件中 NULL 的定义，即将 NULL 定义为 0。因此使用 NULL 做参数调用和使用字面量 0 做参数调用版本的结果完全相同，都是调用到了 f(int) 这个版本。这实际与程序员编写代码的意图相悖。

引起该问题的元凶是字面常量 0 的二义性，在 C++98 标准中，字面常量 0 的类型既可以是一个整型，也可以是一个无类型指针（void* ）。如果程序员想在代码清单 7-1 中调用 f(char*) 版本的话，则必须像随后的代码一样，对字面常量 0 进行强制类型转换（(void*)0 ）并调用，否则编译器总是会优先把 0 看作是一个整型常量。

虽然这个问题可以通过修改代码来解决，但为了避免用户使用上的错误，有的编译器做了比较激进的改进。典型的如 g++ 编译器，它直接将 NULL 转换为编译器内部标识（ __null ），并在编译时期做了一些分析，一旦遇到二义性就停止编译并向用户报告错误。虽然这在一定程度上缓解了二义性带来的麻烦，但由于标准并没有认定 NULL 为一个编译时期的标识，所

以也会带来代码移植性的限制。

注意 关于 nullptr 和 void* 的翻译，void* 习惯被翻作无类型指针，我们这里把 nullptr 翻作指针空值。

在 C++11 新标准中，出于兼容性的考虑，字面常量 0 的二义性并没有被消除。但标准还是为二义性给出了新的答案，就是 nullptr。在 C++11 标准中，nullptr 是一个所谓“指针空值类型”的常量。指针空值类型被命名为 nullptr_t，事实上，我们可以在支持 nullptr 的头文件（cstddef）中找出如下定义：

```
typedef decltype(nullptr) nullptr_t;
```

可以看到，nullptr_t 的定义方式非常有趣，与传统的先定义类型，再通过类型声明值的做法完全相反（充分利用了 decltype 的功能）。我们发现，在现有编译器情况下，使用 nullptr_t 的时候必须 #include<cstddef>（#include 有些头文件也会间接 #include<cstddef>，比如 <iostream>），而 nullptr 则不用。这大概就是由于 nullptr 是关键字，而 nullptr_t 是通过推导而来的缘故。

而相比于 gcc 等编译器将 NULL 预处理为编译器内部标识 __null，nullptr 拥有更大的优势。简单而言，由于 nullptr 是有类型的，且仅可以被隐式转化为指针类型，那么对于代码 7-1 的例子，nullptr 做参数则可以成功调用 f(char*) 版本的函数，而不是像 gcc 对 NULL 的处理一样，仅仅给出一个出错提示，好让程序员去修改代码。

我们来看看代码清单 7-2 所示的例子。

代码清单 7-2

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

void f(char *p) {
    cout << "invoke f(char*)" << endl;
}

void f(int) {
    cout << "invoke f(int)" << endl;
}

int main()
{
    f(nullptr);     // 调用 f(char*) 版本
    f(0);           // 调用 f(int) 版本
    return 0;
}

// 编译选项 :g++ 7-1-2.cpp -std=c++11
```

可以看到，在改为使用 nullptr 之后，用户能够准确表达自己的意图，也不会再出现在 XLC 编译器上调用了 f(int) 版本而在 gcc 上却在编译时期给出了错误提示的不兼容问题。因此，通常情况下，在书写 C++11 代码想使用 NULL 的时候，将 NULL 替换成为 nullptr 我们就能获得更加健壮的代码。

7.1.2 nullptr 和 nullptr_t

C++11 标准不仅定义了指针空值常量 nullptr，也定义了其指针空值类型 nullptr_t，也就表示了指针空值类型并非仅有 nullptr 一个实例。通常情况下，也可以通过 nullptr_t 来声明一个指针空值类型的变量（即使看起来用途不大）。

除去 nullptr 及 nullptr_t 以外，C++ 中还存在各种内置类型。C++11 标准严格规定了数据间的关系。大体上常见的规则简单地列在了下面：

- ❑所有定义为 nullptr_t 类型的数据都是等价的，行为也是完全一致。
- ❑nullptr_t 类型数据可以隐式转换成任意一个指针类型。
- ❑nullptr_t 类型数据不能转换为非指针类型，即使使用 reinterpret_cast<nullptr_t>() 的方式也是不可以的。
- ❑nullptr_t 类型数据不适用于算术运算表达式。
- ❑nullptr_t 类型数据可以用于关系运算表达式，但仅能与 nullptr_t 类型数据或者指针类型数据进行比较，当且仅当关系运算符为 ==、<=、>= 等时返回 true。

我们可以看看代码清单 7-3 所示的例子，这个例子集合了大多数我们需要的场景。

代码清单 7-3

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

int main()
{
    // nullptr 可以隐式转换为 char*
    char * cp = nullptr;

    // 不可转换为整型，而任何类型也不能转换为 nullptr_t，
    // 以下代码不能通过编译
    // int n1 = nullptr;
    // int n2 = reinterpret_cast<int>(nullptr);

    // nullptr 与 nullptr_t 类型变量可以作比较，
    // 当使用 ==、<=、>= 符号比较时返回 true
    nullptr_t nptr;
    if (nptr == nullptr)
        cout << "nullptr_t nptr == nullptr" << endl;
```

```
    else
        cout << "nullptr_t nptr != nullptr" << endl;

    if (nptr < nullptr)
        cout << "nullptr_t nptr < nullptr" << endl;
    else
        cout << "nullptr_t nptr !< nullptr" << endl;

    // 不能转换为整型或 bool 类型，以下代码不能通过编译
    // if (0 == nullptr);
    // if (nullptr);

    // 不可以进行算术运算，以下代码不能通过编译
    // nullptr += 1;
    // nullprt * 5;

    // 以下操作均可以正常进行
    sizeof(nullptr);
    typeid(nullptr);
    throw(nullptr);

    return 0;
}

// 编译选项 :g++ 7-1-3.cpp -std=c++11
```

编译运行代码清单 7-3，我们可以得到以下结果：

```
nullptr_t nptr == nullptr
nullptr_t nptr !< nullptr
terminate called after throwing an instance of 'decltype(nullptr)'
Aborted
```

读者可以对应之前的规则试着分析一下上面的代码为什么有的不能够通过编译，以及为什么会产生上述的运行结果。

注意 如果读者的编译器能够编译 if(nullptr) 或者 if(nullptr == 0) 这样的语句，可能是因为编译器版本还不够新。老的 nullptr 定义中允许 nullptr 向 bool 的隐式转换，这带来了一些问题，而 C++11 标准中已经不允许这么做了。

此外，虽然 nullptr_t 看起来像是个指针类型，用起来更是，但在把 nullptr_t 应用于模板中时候，我们会发现模板却只能把它作为一个普通的类型来进行推导（并不会将其视为 T* 指针）。代码清单 7-4 所示的这个例子来源于 C++11 标准提案。

代码清单 7-4

```
#include <iostream>
```

```
using namespace std;

template<typename T> void g(T* t) {}
template<typename T> void h(T t) {}

int main()
{
   g(nullptr);              // 编译失败，nullptr 的类型是 nullptr_t，而不是指针
   g((float*) nullptr);     // 推导出 T = float

   h(0);                    // 推导出 T = int
   h(nullptr);              // 推导出 T = nullptr_t
   h((float*)nullptr);      // 推导出 T = float*
}

// 编译选项 :g++ 7-1-4.cpp -std=c++11
```

代码清单 7-4 中，g(nullptr) 并不会被编译器“智能”地推导成某种基本类型的指针（或者 void* 指针），因此要让编译器成功推导出 nullptr 的类型，必须做显式的类型转换。

7.1.3 一些关于 nullptr 规则的讨论

nullptr 这个名字看起来是比较古怪的。在 C++98 标准的时候，字符串 NULL 实际上已经得到了广泛的应用。而在 C++11 标准中，委员会采取了另起炉灶的方式，硬生生地添加了 nullptr 这个关键字，这让很多人表示不能理解。但另起炉灶而不是重用 NULL 的原因却是非常明显的。因为 NULL 已经是一个用途广泛的宏，且这个宏被不同的编译器实现为不同的解释，重用 NULL 会使得很多已有的 C++ 程序不能通过 C++11 编译器的编译。因此为了保证最大的兼容性，委员会采用了新的名称 nullptr，以避免和现有标识符的冲突。

此外在 C++11 标准中，nullptr 类型数据所占用的内存空间大小跟 void* 相同的，即：

```
sizeof(nullptr_t) == sizeof(void*)
```

关于这一点也可能引起疑惑，即是否 nullptr 就是 (void*)0 的一个别名。不过答案却是否定的，尽管两者看起来很相似，都可以被转换为任何类型的指针，但两者在语法层面有着不同的内涵。nullptr 是一个编译时期的常量，它的名字是一个编译时期的关键字，能够为编译器所识别。而 (void*)0 只是一个强制转换表达式，其返回的也是一个 void* 指针类型。

而且最为重要的是，在 C++ 语言中，nullptr 到任何指针的转换是隐式的，而 (void*)0 则必须经过类型转换后才能使用。我们可以看看代码清单 7-5 所示的例子。

代码清单 7-5

```
int foo()
{
    int* px = (void*)0;   // 编译错误，不能隐式地将无类型指针转换为 int* 类型的指针
```

```
    int* py = nullptr;
}
```

编译选项：g++ -std=c++11 7-1-5.cpp

可以看到，(void*)0 在使用上并不如 nullptr 方便。在 nullptr 出现之后，程序员大可以忘记 (void*)0，因为 nullptr 已经足够用了，而且也很好用。

注意 C 语言标准中的 void* 指针是可以隐式转换为任意指针的，这一点跟 C++ 是不同的。

此外，我们还注意到 C++11 标准有一条有趣的规定，nullptr_t 对象的地址可以被用户使用（虽然看起来好像没什么实用价值）。但这条规则有一点例外，就是虽然 nullptr 也是一个 nullptr_t 的对象，C++11 标准却规定用户不能获得 nullptr 的地址。其原因主要是因为 nullptr 被定义为一个右值常量，取其地址并没有意义。

不过 C++11 标准并没有禁止声明一个 nullptr 的右值引用，并打印其地址，因此我们在一些编译器上也做了个有趣的实验，对 nullptr 感兴趣的用户可以试运行一下代码清单 7-6 所示的这段的代码。

代码清单 7-6

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

int main(){
    nullptr_t my_null;
    printf("%x\n", &my_null);

    // printf("%x", &nullptr);       // 根据 C++11 的标准设定，本句无法编译通过
    printf("%d\n", my_null == nullptr);

    const nullptr_t && default_nullptr = nullptr;      // default_nullptr 是
    nullptr 的一个右值引用
    printf("%x\n", &default_nullptr);
}

// 编译选项 :g++ -std=c++11 7-1-6.cpp
```

编译运行代码清单 7-6，我们的实验机上的结果看起来是这样：

```
7498fca8
1
7498fcb0
```

当然，运行结果跟所用编译器以及平台都有关系。不过，对于普通用户而言，需要记得的仅仅是，不要对 nullptr 做取地址操作即可。

7.2 默认函数的控制

☞ **类别：类作者**

7.2.1 类与默认函数

在 C++ 中声明自定义的类，编译器会默认帮助程序员生成一些他们未自定义的成员函数。这样的函数版本被称为“默认函数”。这包括了以下一些自定义类型的成员函数：

- ❑构造函数
- ❑拷贝构造函数
- ❑拷贝赋值函数（operator =）
- ❑移动构造函数
- ❑移动拷贝函数
- ❑析构函数

此外，C++ 编译器还会为以下这些自定义类型提供全局默认操作符函数：

- ❑operator,
- ❑operator &
- ❑operator &&
- ❑operator *
- ❑operator ->
- ❑operator ->*
- ❑operator new
- ❑operator delete

在 C++ 语言规则中，一旦程序员实现了这些函数的自定义版本，则编译器不会再为该类自动生成默认版本。有时这样的规则会被程序员忘记，最常见的是声明了带参数的构造版本，则必须声明不带参数的版本以完成无参的变量初始化。不过通过编译器的提示，这样的问题通常会得到更正。但更为严重的问题是，一旦声明了自定义版本的构造函数，则有可能导致我们定义的类型不再是 POD 的。我们可以看看代码清单 7-7 所示的例子。

代码清单 7-7

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

```
class TwoCstor {
public:
    // 提供了带参数版本的构造函数，则必须自行提供
    // 不带参数版本，且 TwoCstor 不再是 POD 类型
    TwoCstor() {};
    TwoCstor(int i): data(i) {}

private:
    int data;
};

int main(){
    cout << is_pod<TwoCstor>::value << endl;    // 0
}

// 编译选项 :g++ -std=c++11 7-2-1.cpp
```

代码清单 7-7 所示的例子中，程序员虽然提供了 TwoCstor() 构造函数，它与默认的构造函数接口和使用方式也完全一致，不过按照 3.6 节我们对“平凡的构造函数”的定义，该构造函数却不是平凡的，因此 TwoCstor 也就不再是 POD 的了。使用 is_pod 模板类查看 TwoCstor，也会发现程序输出为 0。对于形如 TwoCstor 这样只是想增加一些构造方式的简单类型而言，变为非 POD 类型带来一系列负面影响有时是程序员所不希望的（读者可以回顾一下 3.6 节，很多时候，这意味着编译器失去了优化这样简单的数据类型的可能）。因此客观上我们需要一些方式来使得这样的简单类型“恢复”POD 的特质。

而在 C++11 中，标准是通过提供了新的机制来控制默认版本函数的生成来完成这个目标的。这个新机制重用了 default 关键字。程序员可以在默认函数定义或者声明时加上“= default”，从而显式地指示编译器生成该函数的默认版本。而如果指定产生默认版本后，程序员不再也不应该实现一份同名的函数。具体如代码清单 7-8 所示。

代码清单 7-8

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

class TwoCstor {
public:
    // 提供了带参数版本的构造函数，再指示编译器
    // 提供默认版本，则本自定义类型依然是 POD 类型
    TwoCstor() = default;
    TwoCstor(int i): data(i) {}

private:
    int data;
```

```
};

int main(){
    cout << is_pod<TwoCstor>::value << endl; // 1
}

// 编译选项 :g++ 7-2-2.cpp -std=c++11
```

编译运行代码清单 7-8，会得到结果 1。TwoCstor 还是一个 POD 的类型。

另一方面，程序员在一些情况下则希望能够限制一些默认函数的生成。最典型地，类的编写者有时需要禁止使用者使用拷贝构造函数，在 C++98 标准中，我们的做法是将拷贝构造函数声明为 private 的成员，并且不提供函数实现。这样一来，一旦有人试图（或者无意识）使用拷贝构造函数，编译器就会报错。

我们来看看代码清单 7-9 所示的例子。

代码清单 7-9

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

class NoCopyCstor {
public:
    NoCopyCstor() = default;

private:
    // 将拷贝构造函数声明为 private 成员并不提供实现
    // 可以有效阻止用户错用拷贝构造函数
    NoCopyCstor(const NoCopyCstor &);
};

int main(){
    NoCopyCstor a;
    NoCopyCstor b(a);    // 无法通过编译
}

// 编译选项 :g++ 7-2-3.cpp -std=c++11
```

代码清单 7-9 中，NoCopyCstor b(a) 试图调用 private 的拷贝构造函数，该句编译不会通过。不过这样的做法也会对友元类或函数使用造成麻烦。友元类很可能需要拷贝构造函数，而简单声明 private 的拷贝构造函数不实现的话，会导致编译的失败。为了避免这种情况，我们还必须提供拷贝构造函数的实现版本，并将其声明为 private 成员，才能达到需要的效果。

在 C++11 中，标准则给出了更为简单的方法，即在函数的定义或者声明加上 “=

delete”。“= delete”会指示编译器不生成函数的缺省版本。我们可以看代码清单 7-10 所示的例子。

代码清单 7-10

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

class NoCopyCstor {
public:
    NoCopyCstor() = default;

    // 使用“= delete”同样可以有效阻止用户
    // 错用拷贝构造函数
    NoCopyCstor(const NoCopyCstor &) = delete;
};

int main(){
    NoCopyCstor a;
    NoCopyCstor b(a);    // 无法通过编译
}

// 编译选项 :g++ 7-2-4.cpp -std=c++11
```

代码清单 7-10 即是一个使用“= delete”删除拷贝构造函数的缺省版本的实例。值得注意的是，一旦缺省版本被删除了，重载该函数也是非法的。

7.2.2 “= default”与“= deleted”

在上面一节中，我们基本已经看到了 C++11 中“= default”和“= delete”的使用方法，事实上，C++11 标准称“= default”修饰的函数为显式缺省（explicit defaulted）函数，而称“= delete”修饰的函数为删除（deleted）函数。为了方便称呼，本书将删除函数称为显式删除函数。在下面的描述中，我们会沿用这些术语。

C++11 引入显式缺省和显式删除是为了增强对类默认函数的控制，让程序员能够更加精细地控制默认版本的函数。不过这并不是它们的唯一功能，而且使用上，也不仅仅局限在类的定义内。事实上，显式缺省不仅可以用于在类的定义中修饰成员函数，也可以在类定义之外修饰成员函数。代码清单 7-11 所示便是一个例子。

代码清单 7-11

```
class DefaultedOptr{
public:
    // 使用“= default”来产生缺省版本
    DefaultedOptr() = default;
```

```
    // 这里没使用“= default”
    DefaultedOptr & operator = (const DefaultedOptr & );
};

// 在类定义外用“= default”来指明使用缺省版本
inline DefaultedOptr &
DefaultedOptr::operator =( const DefaultedOptr & ) = default;

// 编译选项 :g++ -std=c++11 -c 7-2-5.cpp
```

在本例中，类 DefaultedOptr 的操作符 operator= 被声明在了类的定义外，并且被设定为缺省版本。这在 C++11 规则中也是被允许的。在类定义外显式指定缺省版本所带来的好处是，程序员可以对一个 class 定义提供多个实现版本。假设我们有下面的几个文件：

```
type.h : struct type { type(); };
type1.cc : type::type() = default;
type2.cc : type::type() { /*do some thing */ };
```

那么程序员就可以选择地编译 type1.cc 或者 type2.cc，从而轻易地在提供缺省函数的版本和使用自定义版本的函数间进行切换。这对于一些代码的调试是很有帮助的。

此外，除去我们在上节提到了多个可以由编译器默认提供的缺省函数，显式缺省还可以修饰一些其他函数，比如“operator ==”。C++11 标准并不要求编译器为这些函数提供缺省的实现，但如果将其声明为显式缺省的话，则编译器会按照某些“标准行为”为其生成所需要的版本。

关于显式删除，正如我们在上一节中看到，显式删除可以避免用户使用一些不应该使用的类的成员函数。不过显式删除也并非局限于成员函数，使用显式删除还可以避免编译器做一些不必要的隐式数据类型转换。我们来看看代码清单 7-12 所示的例子。

代码清单 7-12

```
class ConvType {
public:
    ConvType(int i) {};
    ConvType(char c) = delete;  // 删除 char 版本
};

void Func(ConvType ct) {}

int main() {
    Func(3);
    Func('a');  // 无法通过编译

    ConvType ci(3);
    ConvType cc('a');   // 无法通过编译
```

```
}

// 编译选项 :g++ -std=c++11 7-2-6.cpp
```

代码清单 7-12 中，我们显式删除了 ConvType(char) 版本的构造函数。则在调用 Func('a') 及构造变量 cc 的时候，编译器会给出错误提示并停止编译。这是因为编译器发现从 char 构造 ConvType 的方式是不被允许的。不过如果读者将 ConvType(char c) = delete; 这一句注释掉，代码清单 7-12 就可以通过编译了。这种情况下，编译器会隐式地将 a 转换为整型，并调用整型版本的构造函数。这样一来，我们就可以对一些危险的、不应该发生的隐式类型转换进行适当的控制。

不过我们还需要注意一下 explicit 关键字在这里可能产生的影响。让我们稍稍改变一下代码清单 7-12 中的代码，一些意想不到的结果就可能发生，如代码清单 7-13 所示。

代码清单 7-13

```
class ConvType {
public:
    ConvType(int i) {};
    explicit ConvType(char c) = delete;   // 删除 explicit 的 char 构造函数
};

void Func(ConvType ct) {}

int main() {
    Func(3);
    Func('a');        // 可以通过编译

    ConvType ci(3);
    ConvType cc('a');    // 无法通过编译
}

// 编译选项 :g++ -std=c++11 7-2-7.cpp
```

代码清单 7-13 中，语句 explicit ConvType(char) = delete 将从 char explicit 构造 ConvType 的方式显式删除了，这导致 cc 变量的构造不成功，因为其是显式构造的。不过在函数 Func 的调用中，编译器会尝试隐式地将 c 转换成 int，从而 Func('a') 的调用会导致一次 ConvType(int) 构造，因而能够通过编译。这样一来，explicit 带来了令人尴尬的效果，即没有彻底地禁止类型转换的发生。如果程序员发生了这样的错误，也可能会比较难找到原因。

事实上，在 C++11 提案中，提案的作者并不建议用户将 explicit 关键字和显式删除合用。因为两者的合用只会引起一些混乱性，并无什么好处。因此，程序员在使用显式删除时候，应该总是避免 explicit 关键字修饰的函数，反之亦然。

还有一点必须指出，对于使用显式删除来禁止编译器做一些不必要的类型转换上，我们

并不局限于缺省版本的类成员函数或者全局函数上，对于一些普通的函数，我们依然可以通过显式删除来禁止类型转换，如代码清单 7-14 所示。

代码清单 7-14

```
void Func(int i){};
void Func(char c) = delete;  // 显式删除 char 版本

int main(){
    Func(3);
    Func('c');  // 本句无法通过编译
    return 1;
}

// 编译选项 :g++ -std=c++11 7-2-8.cpp
```

代码清单 7-14 所示的例子中，我们显式删除了 Func 的 char 版本，这就会导致 Func('c') 调用的编译失败。

显式删除还有一些有趣的使用方式。比如程序员使用显式删除来删除自定义类型的 operator new 操作符的话，就可以做到避免在堆上分配该 class 的对象。代码清单 7-15 就是这样一个例子。

代码清单 7-15

```
#include <cstddef>

class NoHeapAlloc{
public:
    void * operator new(std::size_t) = delete;
};

int main(){
    NoHeapAlloc nha;
    NoHeapAlloc * pnha = new NoHeapAlloc;    // 编译失败
    return 1;
}

// 编译选项 :g++ -std=c++11 7-2-9.cpp
```

而在一些情况下，比如在代码清单 7-16 中，我们需要对象在指定内存位置进行内存分配，并且不需要析构函数来完成一些对象级别的清理。这个时候，我们可以通过显式删除析构函数来限制自定义类型在栈上或者静态的构造。

代码清单 7-16

```
#include <cstddef>
```

```
#include <new>

extern void* p;

class NoStackAlloc{
public:
    ~NoStackAlloc() = delete;
};

int main(){
    NoStackAlloc nsa;    // 无法通过编译
    new (p) NoStackAlloc(); // placement new, 假设 p 无需调用析构函数
    return 1;
}

// 编译选项 :g++ 7-2-10.cpp -std=c++11 -c
```

由于 placement new 构造的对象，编译器不会为其调用析构函数，因此析构函数被删除的类能够正常地构造。事实上，读者可以推而广之，将显式删除析构函数用于构建单件模式（Singleton），不过本书就不再展开讲了。

7.3 lambda 函数

☞ 类别：所有人

7.3.1 lambda 的一些历史

lambda（λ）在希腊字母表中位于第 11 位。同时，由于希腊数字是基于希腊字母的，所以 λ 在希腊数字中也表示了值 30。在数理逻辑或计算机科学领域中，lambda 则是被用来表示一种匿名函数，这种匿名函数代表了一种所谓的 λ 演算（lambda calculus）。

λ 演算是计算机语言领域的老古董，或者更确切地讲，λ 演算应该算做编程语言理论的研究成果，它的出现指引了实际编程语言的诞生。20 世纪 30 年代，阿隆佐·邱奇[⊖]（Alonzo Church）引入了这套表示计算（computation）的形式系统。1958 年，当时身在 MIT 的约翰·麦肯锡（John McCarthy）创造出了基于 λ 演算的 LISP 语言（但他并没有实现 LISP。第一个 Lisp 语言的实现是 Steve Russell 在 IBM 704 机器上完成的）。LISP 语言历史远早于 C 语言，可以算作第二古老的高级编程语言（第一个成功的高级编程语言是 IBM 的 FORTRAN），也是在学术界产生的第一个成功的编程语言，其被广泛应用于人工智能的研究领域。相比于基于 lambda 的 LISP 的成功，C++ 则显得非常年轻。直到 30 年后，Bjarne Stroustrup 才在贝尔实验室里开始设计并实现 C++。

⊖ 阿隆佐·邱奇是图灵的老师。

而从软件开发的角度看，以 lambda 概念为基础的“函数式编程”（Functional Programming）是与命令式编程（Imperative Programming）、面向对象编程（Object-orientated Programming）等并列的一种编程范型（Programming Paradigm）。现在的高级语言也越来越多地引入了多范型支持，很多近年流行的语言都提供了 lambda 的支持，比如 C#、PHP、JavaScript 等。而现在 C++11 也开始支持 lambda，并可能在标准演进过程中不停地进行修正。这样一来，从最早基于命令式编程范型的语言 C，到加入了面向对象编程范型血统的 C++，再到逐渐融入函数式编程范型的 lambda 的新语言规范 C++11，C/C++ 的发展也在融入多范型支持的潮流中。

7.3.2 C++11 中的 lambda 函数

lambda 的历时悠久，不过具体到 C++11 中，lambda 函数却显得与之前 C++ 规范下的代码在风格上有较大的区别。我们可以通过一个例子先来观察一下，如代码清单 7-17 所示。

代码清单 7-17

```
int main() {
    int girls = 3, boys = 4;
    auto totalChild = [](int x, int y)->int{ return x + y; };
    return totalChild(girls, boys);
}

编译选项：g++ -std=c++11 7-3-1.cpp
```

在代码清单 7-17 所示的例子当中，我们定义了一个 lambda 函数。该函数接受两个参数 (int x, int y)，并且返回其和。直观地看，lambda 函数跟普通函数相比不需要定义函数名，取而代之的多了一对方括号（[]）。此外，lambda 函数还采用了追踪返回类型的方式声明其返回值。其余方面看起来则跟普通函数定义一样。

而通常情况下，lambda 函数的语法定义如下：

```
[capture](parameters) mutable ->return-type{statement}
```

其中，

- [capture]：捕捉列表。捕捉列表总是出现在 lambda 函数的开始处。事实上，[] 是 lambda 引出符。编译器根据该引出符判断接下来的代码是否是 lambda 函数。捕捉列表能够捕捉上下文中的变量以供 lambda 函数使用。具体的方法在下文中会再描述。
- (parameters)：参数列表。与普通函数的参数列表一致。如果不需要参数传递，则可以连同括号 () 一起省略。
- mutable：mutable 修饰符。默认情况下，lambda 函数总是一个 const 函数，mutable 可以取消其常量性。在使用该修饰符时，参数列表不可省略（即使参数为空）。
- ->return-type：返回类型。用追踪返回类型形式声明函数的返回类型。出于方便，不

需要返回值的时候也可以连同符号 -> 一起省略。此外，在返回类型明确的情况下，也可以省略该部分，让编译器对返回类型进行推导。

- {statement}：函数体。内容与普通函数一样，不过除了可以使用参数之外，还可以使用所有捕获的变量。

在 lambda 函数的定义中，参数列表和返还类型都是可选的部分，而捕捉列表和函数体都可能为空。那么在极端情况下，C++11 中最为简略的 lambda 函数只需要声明为

```
[]{};
```

就可以了。不过理所应当地，该 lambda 函数不能做任何事情。

代码清单 7-18 中列出了各种各样的 lambda 函数。

代码清单 7-18

```
int main(){
    []{};                    // 最简 lambda 函数
    int a = 3;
    int b = 4;
    [=] { return a + b;};          // 省略了参数列表与返回类型，返回类型由编译器推断为 int
    auto fun1 = [&](int c) { b = a + c; };       // 省略了返回类型，无返回值
    auto fun2 = [=, &b](int c)->int { return b += a + c; };// 各部分都很完整的 lambda 函数
}

// 编译选项 :g++ -std=c++11 7-3-2.cpp
```

在代码清单 7-18 中，我们看到了各种各样的捕捉列表的使用。直观地讲，lambda 函数与普通函数可见的最大区别之一，就是 lambda 函数可以通过捕捉列表访问一些上下文中的数据。具体地，捕捉列表描述了上下文中哪些的数据可以被 lambda 使用，以及使用方式（以值传递的方式或引用传递的方式）。在代码清单 7-17 的例子中，我们是使用参数的方式传递变量，现在让我们使用捕捉列表来改写这个例子，如代码清单 7-19 所示。

代码清单 7-19

```
int main() {
    int boys = 4, int girls = 3;
    auto totalChild = [girls, &boys]()->int{ return girls + boys; };
    return totalChild();
}

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

代码清单 7-19 中，我们使用了捕捉列表捕捉上下文中的变量 girls、boys。与代码清单 7-17 相比，函数的原型发生了变化，即 totalChild 不再需要传递参数。这个改变看起来平淡无奇，不过读者在阅读 7.3.3 节之后可以知道，此时 girls 和 boys 可以视为 lambda 函数的一

种初始状态，lambda 函数的运算则是基于初始状态进行的运算。这与函数简单基于参数的运算是有所不同的。

语法上，捕捉列表由多个捕捉项组成，并以逗号分割。捕捉列表有如下几种形式：

- ❑[var] 表示值传递方式捕捉变量 var。
- ❑[=] 表示值传递方式捕捉所有父作用域的变量（包括 this）。
- ❑[&var] 表示引用传递捕捉变量 var。
- ❑[&] 表示引用传递捕捉所有父作用域的变量（包括 this）。
- ❑[this] 表示值传递方式捕捉当前的 this 指针。

注意 父作用域：enclosing scope，这里指的是包含 lambda 函数的语句块，在代码清单 7-19 中，即 main 函数的作用域。

通过一些组合，捕捉列表可以表示更复杂的意思。比如：

- ❑[=, &a, &b] 表示以引用传递的方式捕捉变量 a 和 b，值传递方式捕捉其他所有变量。
- ❑[&, a, this] 表示以值传递的方式捕捉变量 a 和 this，引用传递方式捕捉其他所有变量。

不过值得注意的是，捕捉列表不允许变量重复传递。下面一些例子就是典型的重复，会导致编译时期的错误。

- ❑[=, a] 这里 = 已经以值传递方式捕捉了所有变量，捕捉 a 重复。
- ❑[&, &this] 这里 & 已经以引用传递方式捕捉了所有变量，再捕捉 this 也是一种重复。

利用以上的规则，对于代码清单 7-19 的 lambda 函数，我们可以通过 [=] 来声明捕捉列表，进而对 totalChild 书写上的进一步简化，如代码清单 7-20 所示。

代码清单 7-20

```
int main() {
    int boys = 4, girls = 3;
    auto totalChild = [=]()->int{ return girls + boys; };// 捕捉所有父作用域的变量
    return totalChild();
}

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

通过捕捉列表 [=]，lambda 函数的父作用域中所有自动变量都被 lambda 依照传值的方式捕捉了。

必须指出的是，依照现行 C++11 标准，在块作用域（block scope，可以简单理解为在 {} 之内的任何代码都是块作用域的）以外的 lambda 函数捕捉列表必须为空。因此这样的 lambda 函数除去语法上的不同以外，跟普通函数区别不大。而在块作用域中的 lambda 函数

仅能捕捉父作用域中的自动变量，捕捉任何非此作用域或者是非自动变量（如静态变量等）都会导致编译器报错。

7.3.3 lambda 与仿函数

好的编程语言一般都有好的库支持，C++ 也不例外。C++ 语言在标准程序库 STL 中向用户提供了一些基本的数据结构及一些基本的算法等。在 C++11 之前，我们在使用 STL 算法时，通常会使用到一种特别的对象，一般来说，我们称之为函数对象，或者仿函数（functor）。仿函数简单地说，就是重定义了成员函数 operator () 的一种自定义类型对象。这样的对象有个特点，就是其使用在代码层面感觉跟函数的使用并无二样，但究其本质却并非函数。我们可以看一个仿函数的例子，如代码清单 7-21 所示。

代码清单 7-21

```
class _functor {
public:
    int operator()(int x, int y) { return x + y; }
};

int main(){
    int girls = 3, boys = 4;
    _functor totalChild;
    return totalChild(5, 6);
}

// 编译选项 :g++ 7-3-5.cpp -std=c++11
```

这个例子中，class _functor 的 operator() 被重载，因此，在调用该函数的时候，我们看到跟函数调用一样的形式，只不过这里的 totalChild 不是函数名称，而是对象名称。

注意 相比于函数，仿函数可以拥有初始状态，一般通过 class 定义私有成员，并在声明对象的时候对其进行初始化。私有成员的状态就成了仿函数的初始状态。而由于声明一个仿函数对象可以拥有多个不同初始状态的实例，因此可以借由仿函数产生多个功能类似却不同的仿函数实例（这里是一个多状态的仿函数的实例）。

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

class Tax {
private:
    float rate;
    int base;
public:
    Tax(float r, int b): rate(r), base(b){}
```

```
    float operator() (float money) { return (money - base) * rate; }
};

int main() {
    Tax high(0.40, 30000);
    Tax middle(0.25, 20000);
    cout << "tax over 3w: " << high(37500) << endl;
    cout << "tax over 2w: " << middle(27500) << endl;
    return 0;
}
```

这里通过带状态的仿函数，可以设定两种不同的税率的计算。

而仔细观察的话，除去自定义类型 _functor 的声明及其对象的定义，可以发现代码清单 7-21 跟代码清单 7-17 中 lambda 函数的定义看起非常类似。这是否说明仿函数跟 lambda 在实现之间存在着一种默契呢？我们可以再来看一个例子，如代码清单 7-22 所示。

代码清单 7-22

```
class AirportPrice{
private:
    float _dutyfreerate;
public:
    AirportPrice(float rate): _dutyfreerate(rate){}
    float operator()(float price) {
        return price * (1 - _dutyfreerate/100);
    }
};

int main(){
    float tax_rate = 5.5f;
    AirportPrice Changi(tax_rate);

    auto Changi2 =
        [tax_rate](float price)->float{ return price * (1 - tax_rate/100); };

    float purchased = Changi(3699);

    float purchased2 = Changi2(2899);
}

// 编译选项 :g++ 7-3-6.cpp -std=c++11
```

代码清单 7-22 是一个机场返税的例子。该例中，分别使用了仿函数和 lambda 两种方式来完成扣税后的产品价格计算。在这里我们看到，lambda 函数捕捉了 tax_rate 变量，而仿函数则以 tax_rate 初始化类。其他的，如在参数传递上，两者保持一致。可以看到，除去在语法层面上的不同，lambda 和仿函数却有着相同的内涵——都可以捕捉一些变量作为初始状态，并接受参数进行运算。

而事实上，仿函数是编译器实现 lambda 的一种方式。在现阶段，通常编译器都会把 lambda 函数转化为成为一个仿函数对象。因此，在 C++11 中，lambda 可以视为仿函数的一种等价形式了，或者更动听地说，lambda 是仿函数的“语法甜点”。

我们可以通过图 7-1 展现代码清单 7-22 中的 lambda 函数和仿函数是如何等价的。

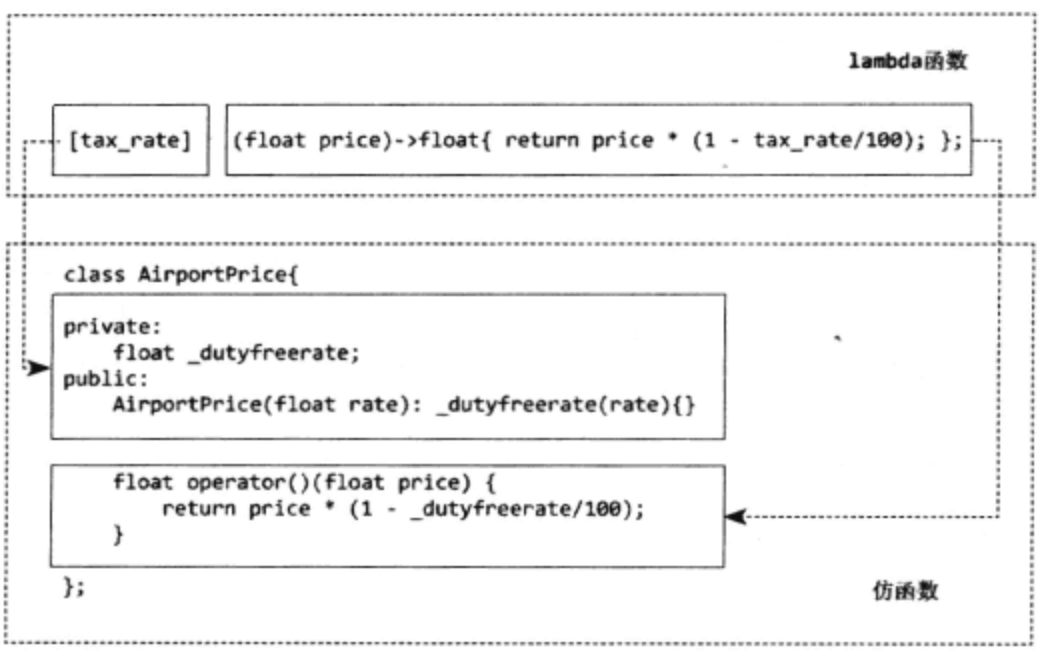

图 7-1 lambda 函数及与其等价的仿函数

注意 有的时候，我们在编译时发现 lambda 函数出现了错误，编译器会提示一些构造函数等相关信息。这显然是由于 lambda 的这种实现方式造成的。理解了这种实现，用户也就能够正确理解错误信息的由来。

如前面提到的，仿函数被广泛地用于 STL 中，同样的，在 C++11 中，lambda 也在标准库中被广泛地使用。由于其书写简单，通常可以就地定义，因此用户常可以使用 lambda 代替仿函数来书写代码，我们可以在 7.3.4 节中看到相关的应用方式，在 7.3.6 中了解 lambda 何时可以取代仿函数。

7.3.4 lambda 的基础使用

依据 lambda 的语法，编写 lambda 函数是非常容易的。不过用得上 lambda 函数的地方比较特殊。最为简单的应用下，我们会利用 lambda 函数来封装一些代码逻辑，使其不仅具有函数的包装性，也具有就地可见的自说明性。让我们首先来看一个例子，如代码清单 7-23 所示。

代码清单 7-23

```
extern int z;
```

```
extern float c;
void Calc(int& , int, float &, float);

void TestCalc() {
    int x, y = 3;
    float a, b = 4.0;

    int success = 0;

    auto validate = [&]() -> bool
    {
        if ((x == y + z) && (a == b + c))
            return 1;
        else
            return 0;
    };

    Calc(x, y, a, b);
    success += validate();

    y = 1024;
    b = 1e13;

    Calc(x, y, a, b);
    success += validate();
}
```

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

在代码清单 7-23 所示的例子中，用户试图用自己写的函数 TestCalc 进行测试。这里使用了一个 auto 关键字推导出了 validate 变量的类型为匿名 lambda 函数。可以看到，我们使用 lambda 函数直接访问了 TestCal 中的局部的变量来完成这个工作。

在没有 lambda 函数之前，通常需要在 TestCalc 外声明同样一个函数，并且把 TestCalc 中的变量当作参数进行传递。出于函数作用域及运行效率考虑，这样声明的函数通常还需要加上关键字 static 和 inline。相比于一个传统意义上的函数定义，lambda 函数在这里更加直观，使用起来也非常简便，代码可读性很好，效果上，lambda 函数则等同于一个“局部函数”。

注意 局部函数（local function，即在函数作用域中定义的函数），也称为内嵌函数（nested function）。局部函数通常仅属于其父作用域，能够访问父作用域的变量，且在其父作用域中使用。C/C++ 语言标准中不允许局部函数存在（不过一些其他语言是允许的，比如 FORTRAN），C++11 标准却用比较优雅的方式打破了这个规则。因为事实上，lambda 可以像局部函数一样使用。

必须指出的是，相比于在函数外定义的 static inline 函数，或者是自定义的宏，本例中 lambda 函数并没有实际运行时的性能优势（但也不会差，lambda 函数在 C++11 标准中默认是内联的）。同局部函数一样，lambda 函数在代码的作用域上仅属于其父作用域，不过直观地看，lambda 函数代码的可读性可能更好，尤其对于小的函数而言。

对于运算比较复杂的函数（比如实现一些很复杂的算法），通常函数中会有大量的局部状态（变量），这个时候如果程序员只是需要一些“局部”的功能——比如打印一些内部状态，或者做一些固定的操作，这些功能往往不能与其他任何的代码共享，却要在一个函数中多次重用。那么使用 lambda 的捕捉列表功能则相较于独立的全局静态函数或私有成员函数方便很多。设计一个仅使用捕捉列表 lambda 函数不会像设计传统函数一样要关心大量细节：需要多少参数、哪些参数需要按值传递、哪些参数需要按引用或者指针传递，通通不需要程序员来考虑。而且父函数结束后，该 lambda 函数也就不再可用了，不会污染任何名字空间（当然，事实上如我们所讲的，lambda 本身就是匿名的函数）。因此，对于复杂代码的快速开发而言，lambda 的出现意义重大。事实上，这些也是局部函数的好处。在 C++11 之前，程序员只能通过编写类来模拟局部函数，实现复杂而且常常会存在着各种各样的问题，lambda 的出现使得这样的做法统统成为了历史。

我们再来看一个常量性的例子。在编写程序的时候，程序员通常会发现自己需要一些“常量”，不过这些常量的值却由自己初始化状态决定的。我们来看看代码清单 7-24 所示的例子。

代码清单 7-24

```
int Prioritize(int );

int AllWorks(int times){
    int i;

    int x;
    try {
        for (i = 0; i < times; i++)
            x += Prioritize(i);
    }
    catch(...) {
        x = 0;
    }

    const int y = [=]{
        int i, val;
        try {
            for (i = 0; i < times; i++)
                val += Prioritize(i);
        }
        catch(...) {
```

```
            val = 0;
        }
        return val;
    }();
}

// 编译选项 :g++ -std=c++11 7-3-8.cpp -c
```

在代码清单 7-24 所示的例子中，我们对 x 和 y 的初始化实际是完全一致的。可以看到，x（或 y）的初始化需要循环调用函数 Prioritize，并且在 Prioritize 抛出异常的时候对 x（或 y）赋默认值 0。

在不使用函数的情况下，由于初始化要在运行时修改 x 的值，因此，虽然 x 在初始化之后对于程序而言是个常量，却不能被声明为 const。而在定义 y 的时候，由于我们就地定义 lambda 函数并且调用，y 仅需使用其返回值，于是常量性得到了保证。

读者可能觉得也可以定义一个普通函数来完成 y 的常量定义，但对于代码清单 7-24 中的代码量较少，自说明意义较强的初始化而言，无疑采用 lambda 函数初始化的时候，代码的可读性会更好。而且这么写，我们也就不需要为这段代码逻辑取个函数名（通常 init 这样的名字是正确的，但是却很难解释清楚代码做了什么）。这是 lambda 函数的一个优势所在。

7.3.5 关于 lambda 的一些问题及有趣的实验

使用 lambda 函数的时候，捕捉列表不同会导致不同的结果。具体地讲，按值方式传递捕捉列表和按引用方式传递捕捉列表效果是不一样的。对于按值方式传递的捕捉列表，其传递的值在 lambda 函数定义的时候就已经决定了。而按引用传递的捕捉列表变量，其传递的值则等于 lambda 函数调用时的值。我们可以看一下代码清单 7-25 所示的例子。

代码清单 7-25

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

int main() {
    int j = 12;
    auto by_val_lambda = [=] { return j + 1;};
    auto by_ref_lambda = [&] { return j + 1;};
    cout << "by_val_lambda: " << by_val_lambda() << endl;
    cout << "by_ref_lambda: " << by_ref_lambda() << endl;

    j++;
    cout << "by_val_lambda: " << by_val_lambda() << endl;
    cout << "by_ref_lambda: " << by_ref_lambda() << endl;
}
```

编译选项：g++ -std=c++11 7-3-9.cpp

完成编译运行后，我们可以看到运行结果如下：

```
by_val_lambda: 13
by_ref_lambda: 13
by_val_lambda: 13
by_ref_lambda: 14
```

第一次调用 by_val_lambda 和 by_ref_lambda 时，其运算结果并没有不同。两者均计算的是 12+1=13。但在第二次调用 by_val_lambda 的时候，其计算的是 12+1=13，相对地，第二次调用 by_ref_lambda 时计算的是 13+1=14。这个结果的原因是由于在 by_val_lambda 中，j 被视为了一个常量，一旦初始化后不会再改变（可以认为之后只是一个跟父作用域中 j 同名的常量）而在 by_ref_lambda 中，j 仍在使用父作用域中的值。

因此简单地总结的话，在使用 lambda 函数的时候，如果需要捕捉的值成为 lambda 函数的常量，我们通常会使用按值传递的方式捕捉；反之，需要捕捉的值成为 lambda 函数运行时的变量（类似于参数的效果），则应该采用按引用方式进行捕捉。

此外，关于 lambda 函数的类型以及该类型跟函数指针之间的关系，读者可能存在一些困惑。回顾之前的例子，大多数情况下把匿名的 lambda 函数赋值给了一个 auto 类型的变量，这是一种声明和使用 lambda 函数的方法。结合之前关于 auto 的知识，有人会猜测 totalChild 是一种函数指针类型的变量，而在阅读过 7.3.2 节关于 lambda 与仿函数之间关系之后，大多数读者会更倾向于认为 lambda 是一种自定义类型。而事实上，lambda 的类型并非简单的函数指针类型或者自定义类型。

从 C++11 标准的定义上可以发现，lambda 的类型被定义为“闭包”（closure）的类[⊖]，而每个 lambda 表达式则会产生一个闭包类型的临时对象（右值）。因此，严格地讲，lambda 函数并非函数指针。不过 C++11 标准却允许 lambda 表达是向函数指针的转换，但前提是 lambda 函数没有捕捉任何变量，且函数指针所示的函数原型，必须跟 lambda 函数有着相同的调用方式。我们可以通过代码清单 7-26 所示的这个例子来说明。

代码清单 7-26

```
int main() {
    int girls = 3, boys = 4;
    auto totalChild = [](int x, int y)->int{ return x + y; };
    typedef int (*allChild)(int x, int y);
    typedef int (*oneChild)(int x);

    allChild p;
```

⊖ C++11 标准定义，closure 类型被定义为特有的（unique）、匿名且非联合体（unnamed nonunion）的 class 类型。

```
    p = totalChild;

    oneChild q;
    q = totalChild;      // 编译失败，参数必须一致

    decltype(totalChild) allPeople = totalChild;    // 需通过 decltype 获得 lambda 的类型
    decltype(totalChild) totalPeople = p;       // 编译失败，指针无法转换为 lambda
    return 0;
}

// 编译选项 :g++ -std=c++11 7-3-10.cpp
```

在代码清单 7-26 所示的例子中，我们可以把没有捕捉列表的 totalChild 转化为接受参数类型相同的 allChild 类型的函数指针。不过，转化为参数类型不一致的 oneChild 类型则会失败。此外，将函数指针转化为 lambda 也是不成功的（虽然似乎 C++11 标准并没有明确禁止这一点）。

值得注意的是，程序员也可以通过 decltype 的方式来获得 lambda 函数的类型。其方式如同代码清单 7-26 中声明 allPeople 一样，虽然使用 decltype 来获得 lambda 函数类型的做法不是很常见，但在实例化一些模板的时候使用该方法会较为有用。

除此之外，还有一个问题是关于 lambda 函数的常量性及 mutable 关键字的。我们来看看代码清单 7-27 所示的这个例子[⊖]。

代码清单 7-27

```
int main(){
    int val;
    // 编译失败，在 const 的 lambda 中修改常量
    auto const_val_lambda = [=]() { val = 3;};

    // 非 const 的 lambda，可以修改常量数据
    auto mutable_val_lambda = [=]() mutable { val = 3;};

    // 依然是 const 的 lambda，不过没有改动引用本身
    auto const_ref_lambda = [&] { val = 3;};

    // 依然是 const 的 lambda，通过参数传递 val
    auto const_param_lambda = [&](int v) { v = 3;};
    const_param_lambda(val);

    return 0;
}

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

⊖ 该例子的问题实际上来源自网络上的一次讨论：http://stackoverflow.com/questions/5501959/why-does-c0xs-lambda-require-mutable-keyword-for-capture-by-value-by-defau。

在代码清单 7-27 所示的例子中，我们定义了 4 种不同的 lambda 函数，这 4 种 lambda 函数本身的行为都是一致的，即修改父作用域中传递而来的 val 参数的值。不过对于 const_val_lambda 函数而言，编译器认为这是一个错误。

```
7-3-10.cpp: In lambda function:
7-3-10.cpp:4:43: error: assignment of read-only variable 'val'
```

而对于声明了 mutable 属性的 mutable_val_lambda 函数，以及通过引用传递变量 val 的 const_ref_lambda 函数，甚至是通过参数来传递变量 val 的 const_param_lambda，编译器均不会报错。

如我们之前的定义中提到一样，C++11 中，默认情况下 lambda 函数是一个 const 函数。按照规则，一个 const 的成员函数是不能在函数体中改变非静态成员变量的值的。但这里明显编译器对不同传参或捕捉列表的 lambda 函数执行了不同的规则有着不同的见解。其究竟是基于什么样的规则而做出了这样的决定呢?

初看这个问题比较让人困惑，但事实上这跟 lambda 函数的特别的常量性相关。这里我们还是需要使用 7.3.3 中的知识将 lambda 函数转化为一个完整的仿函数，需要注意的是，lambda 函数的函数体部分，被转化为仿函数之后会成为一个 class 的常量成员函数。整个 const_val_lambda 看起来会是代码清单 7-28 所示代码的样子。

代码清单 7-28

```
class const_val_lambda{
public:
    const_val_lambda(int v): val(v){}

public:
    void operator()() const { val = 3; }/* 注意：常量成员函数 */

private:
    int val;
};

// 编译选项 :g++ -std=c++11 7-3-12.cpp -c
```

对于常量成员函数，其常量的规则跟普通的常量函数是不同的。具体而言，对于常量成员函数，不能在函数体内改变 class 中任何成员变量。因此，如果将代码清单 7-28 中的仿函数替代代码清单 7-27 中的 lambda 函数，编译报错则显得理所应当。

现在问题就比较清楚了。lambda 的捕捉列表中的变量都会成为等价仿函数的成员变量（如 const_val_lambda 中的成员 val），而常量成员函数（如 operator()）中改变其值是不允许的，因而按值捕捉的变量在没有声明为 mutable 的 lambda 函数中，其值一旦被修改就会导致编译器报错。

而使用引用的方式传递的变量在常量成员函数中值被更改则不会导致错误。关于这一点在很多 C++ 书籍中已经有过讨论。简单地说，由于函数 const_ref_lambda 不会改变引用本身，而只会改变引用的值，因此编译器将编译通过。至于按传递参数的 const_param_lambda，就更加不会引起编译器的“抱怨”了。

准确地讲，现有 C++11 标准中的 lambda 等价的是有常量 operator() 的仿函数。因此在使用捕捉列表的时候必须注意，按值传递方式捕捉的变量是 lambda 函数中不可更改的常量（如同我们之前在按值和按引用方式捕捉的讨论中提到的一样，不过现在我们已经在语言层面看到了限制）。标准这么设计可能是源自早期 STL 算法一些设计上的缺陷（对仿函数没有做限制，从而导致一些设计不算特别良好的算法出错，可以参考 Scott Mayer 的 Effective STL item 39，或者 Nicolai M.Josuttis 的 The C++ Standard Library - A Tutorial and Reference 关于仿函数的部分）。而更一般地讲，这样的设计有其合理性，改变从上下文中拷贝而来的临时变量通常不具有任何意义。绝大多数时候，临时变量只是用于 lambda 函数的输入，如果需要输出结果到上下文，我们可以使用引用，或者通过让 lambda 函数返回值来实现。

此外，lambda 函数的 mutable 修饰符可以消除其常量性，不过这实际上只是提供了一种语法上的可能性，现实中应该没有多少需要使用 mutable 的 lambda 函数的地方。大多数时候，我们使用默认版本的（非 mutable）的 lambda 函数也就足够了。

注意 关于按值传递捕捉的变量不能被修改这一点，有人认为这算是“闭包”类型的名称的体现，即在复制了上下文中变量之后关闭了变量与上下文中变量的联系，变量只与 lambda 函数运算本身有关，不会影响 lambda 函数（闭包）之外的任何内容。

7.3.6 lambda 与 STL

如 7.3.3 节中讲到的，lambda 对 C++11 最大的贡献，或者说是改变，应该在 STL 库中。而更具体地说，我们会发现使用 STL 的算法更加容易了，也更加容易学习了。

首先我们来看一个最为常见的 STL 算法 for_each。简单地说，for_each 算法的原型如下：

```
UnaryProc for_each(InputIterator beg, InputIterator end, UnaryProc op)
```

让我们忽略一些细节，大概讲，for_each 算法需要一个标记开始的 iterator，一个标记结束的 iterator，以及一个接受单个参数的“函数”（即一个函数指针、仿函数或者 lambda 函数）。

for_each 的一个示意实现如下：

```
for_each(iterator begin, iterator end, Function fn) {
    for (iterator i = begin; i != end; ++i)   fn(*i);
}
```

通过 for_each，我们可以完成各种循环操作，如代码清单 7-29 所示。

代码清单 7-29

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

vector<int> nums;
vector<int> largeNums;

const int ubound = 10;

inline void LargeNumsFunc(int i){
    if (i > ubound)
        largeNums.push_back(i);
}

void Above() {
    // 传统的 for 循环
    for (auto itr = nums.begin(); itr != nums.end(); ++itr) {
        if (*itr >= ubound)
            largeNums.push_back(*itr);
    }

    // 使用函数指针
    for_each(nums.begin(), nums.end(), LargeNumsFunc);

    // 使用 lambda 函数和算法 for_each
    for_each(nums.begin(), nums.end(), [=](int i){
            if (i > ubound)
                largeNums.push_back(i);
            });
}

编译选项：g++ 7-3-13.cpp -c -std=c++11
```

在代码清单 7-29 的例子当中，我们分别用了 3 种方式来遍历一个 vector nums，找出其中大于 ubound 的值，并将其写入另外一个 vector largeNums 中。第一种是传统的 for 循环；第二种，则更泛型地使用了 for_each 算法以及函数指针；第三种同样使用了 for_each，但是第三个参数传入的是 lambda 函数。

首先必须指出的是使用 for_each 的好处。如 Scott Mayer 在 Effective STL（item43）中提到的一样，使用 for_each 算法相较于手写的循环在效率、正确性、可维护性上都具有一定优势。最典型的，程序员不用关心 iterator，或者说循环的细节，只需要设定边界，作用于每个元素的操作，就可以在近似“一条语句”内完成循环，正如函数指针版本和 lambda 版本完成的那样。

那么我们再比较一下函数指针方式以及 lambda 方式。函数指针的方式看似简洁，不过却有很大的缺陷。第一点是函数定义在别的地方，比如很多行以前（后）或者别的文件中，这样的代码阅读起来并不方便。第二点则是出于效率考虑，使用函数指针很可能导致编译器不对其进行 inline 优化（inline 对编译器而言并非强制），在循环次数较多的时候，内联的 lambda 和没有能够内联的函数指针可能存在着巨大的性能差别。因此，相比于函数指针，lambda 拥有无可替代的优势。

此外，函数指针的应用范围相对狭小，尤其是我们需要具备一些运行时才能决定的状态的时候，函数指针就会捉襟见肘了。倘若回到 10 年前（C++98 时代），遇到这种情况的时候，迫切想应用泛型编程的 C++ 程序员或许会毫不犹豫地使用仿函数，不过现在我们则没有必要那么做。

我们稍微修改一下代码清单 7-29 所示的例子，如代码清单 7-30 所示。

代码清单 7-30

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

vector<int> nums;
vector<int> largeNums;

class LNums{
    public:
        LNums(int u): ubound(u){}

        void operator () (int i) const
        {
            if (i > ubound)
                largeNums.push_back(i);
        }
    private:
        int ubound;
};

void Above(int ubound) {
    // 传统的 for 循环
    for (auto itr = nums.begin(); itr != nums.end(); ++itr) {
        if (*itr >= ubound)
            largeNums.push_back(*itr);
    }

    // 使用仿函数
    for_each(nums.begin(), nums.end(), LNums(ubound));

    // 使用 lambda 函数和算法 for_each
```

```
    for_each(nums.begin(), nums.end(), [=](int i){
            if (i > ubound)
                largeNums.push_back(i);
            });

}

// 编译选项:g++ 7-3-14.cpp -std=c++11 -c
```

在代码清单7-30中，为了函数的最大可用性，我们把代码清单7-29中的全局变量ubound变成了代码清单7-30中的Above的参数。这样一来，我们传递给for_each函数的第三个参数（函数指针，仿函数或是lambda）而言就需要知道ubound是多少。由于函数只能通过参数传递这个状态（ubound），那么除非for_each调用函数的方式做出改变（比如增加调用函数的参数），否则编译器不会让其通过编译。因此，这个时候拥有状态的仿函数是更佳的选择。不过比较上面的代码，我们可以直观地看到lambda函数比仿函数书写上的简便性。因此，lambda在这里依然是最佳的选择（如果读者觉得这样还不够过瘾的话，那可以尝试一下C++11新风格的for循环。对于代码清单7-30所示的这个例子，新风格的for会更加简练）。

注意 事实上，STL算法对传入的“函数”的原型有着严格的说明，像for_each就只能传入使用单参数进行调用的“函数”。有的时候用户可以通过STL的一些adapter来改变参数个数，不过必须了解的是，这些adapter其实也是仿函数。

在C++98时代，STL中其实也内置了一些仿函数供程序员使用。代码清单7-31所示的例子中，我们将重点比较一下这些内置仿函数与lambda使用上的特点。

代码清单7-31

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

extern vector<int> nums;

void OneCond(int val){
   // 传统的for循环
   for (auto i = nums.begin(); i != nums.end(); i++)
      if (*i == val) break;

   // 内置的仿函数(template) equal_to, 通过bind2nd使其成为单参数调用的仿函数
   find_if(nums.begin(), nums.end(), bind2nd(equal_to<int>(), val));

   // 使用lambda函数
   find_if(nums.begin(), nums.end(), [=](int i) {
            return i == val;
```

```
            });
}

// 编译选项：g++ -c -std=c++11 7-3-15.cpp
```

在代码清单 7-31 中，我们还是列出了 3 种方式来寻找 vector nums 中第一个值等于 val 的元素。我们可以看一下使用内置仿函数的方式。没有太多接触过 STL 算法的人可能对 bind2nd(equal_to<int>(), val) 这段代码相当迷惑，但简单地说，就是定义了一个功能是比较 i==val 的仿函数，并通过一定方式（bind2nd）使其函数调用接口只需要一个参数即可。反观 lambda 函数，其意义简洁明了，使用者使用的时候，也不需要有太多的背景知识。

但现在为止，我们并不能说 lambda 已经赢过了内置仿函数。而实际上，内置的仿函数应用范围很受限制，我们改动一下代码清单 7-31，使其稍微复杂一点，如代码清单 7-32 所示。

代码清单 7-32

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

extern vector<int> nums;

void TwoCond(int low, int high) {
    // 传统的 for 循环
    for (auto i = nums.begin(); i != nums.end(); i++)
        if (*i >= low && *i < high) break;

    // 利用了 3 个内置的仿函数，以及非标准的 compose2
    find_if(nums.begin(), nums.end(),
            compose2(logical_and<bool>(),
                bind2nd(less<int>(), high),
                bind2nd(greater_equal<int>(), low)));

    // 使用 lambda 函数
    find_if(nums.begin(), nums.end(), [=](int i) {
            return i >= low && i < high;
            });
}

// 编译选项：g++ -c -std=c++11 7-3-16.cpp
```

在代码清单 7-32 中，我们将代码清单 7-31 中的判断条件稍微调整得复杂了一些，即需找到 vector nums 中第一个值介于 [low, high) 间的元素。这里我们看到内置仿函数变得异常复杂，而且程序员不得不接受使用非标准库函数的事实（compose2）。这对于需要移植性的程序来说，是非常难以让人接受的。即使之前曾经很多人对内置仿函数这样的做法点头称赞，

但现实情况下可能人人都必须承认：lambda 版本的实现非常的清晰，而且这一次代码量甚至少于内置仿函数的版本，简直无可挑剔。当然，这还不是全部，我们来看下一个例子，如代码清单 7-33 所示。

代码清单 7-33

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

vector<int> nums;

void Add(const int val){
    auto print = [&]{
        for (auto s: nums){ cout << s << '\t'; }
        cout<< endl;
    };

    // 传统的 for 循环方式
    for (auto i = nums.begin(); i != nums.end(); ++i){
        *i = *i + val;
    }
    print();

    // 试一试 for_each 及内置仿函数
    for_each(nums.begin(), nums.end(), bind2nd(plus<int>(), val));
    print();

    // 实际这里需要使用 STL 的一个变动性算法：transform
    transform(nums.begin(), nums.end(), nums.begin(), bind2nd(plus<int>(), val));
    print();

    // 不过在 lambda 的支持下，我们还是可以只使用 for_each
    for_each(nums.begin(), nums.end(), [=](int &i){
                i += val;
            });
    print();
}


int main(){
    for (int i = 0; i < 10; i++){
        nums.push_back(i);
    }
    Add(10);
    return 1;
}
```

```
编译选项：g++ 7-3-17.cpp -std=c++11
```

在代码清单 7-33 所示的例子中，我们试图改变 vector 中的内容，即将 vector 中所有的元素的数值加 10。这里我们还是使用了传统的 for 方式、内置仿函数的方式，以及 lambda 的方式。此外，为了方便调试，我们使用了一个 lambda 函数来打印局部运行的结果。

在机器上编译运行代码清单 7-33，可以看到如下运行结果：

```
10      11      12      13      14      15      16      17      18      19
10      11      12      13      14      15      16      17      18      19
20      21      22      23      24      25      26      27      28      29
30      31      32      33      34      35      36      37      38      39
```

这里我们注意到，结果的第二行和第一行没有区别。仔细查过资料之后，我们发现，内置的仿函数 plus<int>() 仅仅将加法结果返回，为了将返回结果再应用于 vector nums，通常情况下，我们需要使用 transform 这个算法。如我们第三段代码所示，transform 会遍历 nums，并将结果写入 nums.begin() 出首地址的目标区（第三参数）。

事实上，在书写 STL 的时候人们总是会告诫新手 for_each 和 transform 之间的区别，因为 for_each 并不像 transform 一样写回结果。这在配合 STL 内置的仿函数的时候就会有些使用上的区别。但在 C++11 的 lambda 来临的时候，这样的困惑就变少了。因为内置的仿函数并非必不可少，lambda 中包含的代码逻辑一目了然，使用 STL 算法的规则也因此变得简单多了。

我们来看最后一个 STL 的例子，如代码清单 7-34 所示。

代码清单 7-34

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

void Stat(vector<int> &v){
    int errors;
    int score;
    auto print = [&]{
        cout << "Errors: " << errors << endl
             << "Score: " << score << endl;
    };
    // 使用 accumulate 算法，需要两次循环
    errors = accumulate(v.begin(), v.end(), 0);
    score = accumulate(v.begin(), v.end(), 100, minus<int>());
    print();

    errors = 0;
    score = 100;
```

```
    // 使用 lambda 则只需要一次循环
    for_each(v.begin(), v.end(), [&](int i){
            errors += i;
            score -= i;
            });
    print();
}

int main(){
    vector<int> v(10);
    generate(v.begin(), v.end(), []{
            return rand() % 10;
            });
    Stat(v);
}
```

编译选项：g++ 7-3-18.cpp -std=c++11

代码清单 7-34 是一个区间统计的例子，通常情况下，我们可以使用 STL 的 accumulate 算法及内置仿函数来完成这个运算。从代码的简洁性上来看，这样做好像也不错。不过实际上我们产生了两次循环，一次计算 errors，一次计算 score。而使用 lambda 和万能的 for_each 的版本的时候，我们可以从源代码层将循环合并起来。事实上，我们可能在实际工作中能够合并的循环更多。如果采用 accumulate 的方式，而编译器不能合理有效地合并循环的话，我们可能就会遭受一定的性能损失（比如 cache 方面）。

可以看到，对于 C++ 的 STL 和泛型编程来讲，lambda 存在的意义重大，尤其是对于 STL 的算法方面，lambda 的出现使得学习使用 STL 算法的代价大大降低，甚至会改变一些使用 STL 算法的思维。当然，在一些情况下，lambda 还能在代码自说明、性能上具有一定的优势。这些都是很多 C++ 程序员，尤其是喜欢泛型编程的 C++ 程序员梦寐以求的。

7.3.7 更多的一些关于 lambda 的讨论

lambda 被纳入 C++ 语言之后，很多人认为 lambda 让 C++11 语言看起来更加复杂。一来 lambda 的语法与之前的 C++ 语法相比起来显得有些独特，二来其基于仿函数的实现，让初学者会感觉一时间找不到它的用途。不过，在考虑过编写仿函数或者使用 STL 内置的仿函数的复杂性之后，大多数人会肯定 lambda 的应用价值。

不过要完全用好 lambda，必须了解一些 lambda 的特质。比如 lambda 和仿函数之间的取舍，如何有效地使用 lambda 的捕捉列表等。

首先必须了解的是，在现有 C++11 中，lambda 并不是仿函数的完全替代者。这一点很大程度上是由 lambda 的捕捉列表的限制造成的。在现行 C++11 标准中，捕捉列表仅能捕捉父作用域的自动变量，而对超出这个范围的变量，是不能被捕捉的。我们可以看看代码清单

7-35 所示的例子。

代码清单 7-35

```
int d = 0;
int TryCapture() {
    auto ill_lambda = [d]{};
}

// 编译选项 :g++ -std=c++11 -c 7-3-19.cpp
```

代码清单 7-35 所示的例子在一些编译器（如 g++）上可以编译通过，但程序员会得到一些编译器的警告，而一些严格遵守 C++11 语言规则的编译器则会直接报错。而如果我们采用仿函数，则不会有这样的限制，如代码清单 7-36 所示。

代码清单 7-36

```
int d = 0;

class healthyFunctor{
public:
    healthyFunctor(int d): data(d){}

    void operator () () const {}

private:
    int data;
};

int TryCapture() {
    healthyFunctor hf(d);
}

// 编译选项 :g++ -std=c++11 -c 7-3-20.cpp
```

在代码清单 7-36 中的 healthyFunctor 说明了两者的不同。更一般地讲，仿函数可以被定义以后在不同的作用域范围内取得初始值。这使得仿函数天生具有了跨作用域共享的特征。

总地来说，lambda 函数被设计的目的，就是要就地书写，就地使用。使用 lambda 的用户，更倾向于在一个屏幕里看到所有的代码，而不是依靠代码浏览工具在文件间找到函数的实现。而在封装的思维层面上，lambda 只是一种局部的封装，以及局部的共享。而需要全局共享的代码逻辑，我们则还是需要用函数（无状态）或者仿函数（有状态）封装起来。

简单地总结一下，使用 lambda 代替仿函数的应该满足如下一些条件：

- 是局限于一个局部作用域中使用的代码逻辑。

❑这些代码逻辑需要被作为参数传递。

此外，关于捕捉列表的使用也存在有很多的讨论。由于 [=], [&] 这些写法实在是太过方便了，有的时候，我们不会仔细思考其带来的影响就开始滥用，这也会造成一些意想不到的问题。

首先，我们来看一下 [=]，除去我们之前提过的，所有捕捉的变量在 lambda 声明一开始就被拷贝，且拷贝的值不可被更改，这两点需要程序员注意之外，还有一点就是拷贝本身。这点跟函数参数按值方式传递是一样的，如果不想带来过大的传递开销的话，可以采用引用传递的方式传递参数。

其次，我们再来看一下 [&]。如我们之前提到过的，通过引用方式传递的对象也会输出到父作用域中。同样的，父作用域对这些对象的操作也会传递到 lambda 函数中。因此，如果我们代码存在异步操作，或者其他可能改变对象的任何操作，我们必须确定其在父作用域及 lambda 函数间的关系，否则也会产生一些错误。

通常情况下，在使用 [=]、[&] 这些默认捕捉列表的时候，我们需要考察其性能、与父作用域如何关联等。捕捉列表是 lambda 最神奇也是最容易犯错的地方，程序员不能一味图方便了事。

7.4 本章小结

本章重点讲解了3个C++11中会给用户带来思维方式上转变的的特性：nullptr、=default/=deleted 以及 lambda 函数。

nullptr 这个新特性主要是用于指针初始化，C++11 标准通过引入新关键字 nullptr 排除字面常量 0 的二义性，使之成为指针初始化的标准方式。nullptr 可以安全地转换成各种指针，这使得 nullptr 使用简便，而 nullptr 不能转换为除指针外的任何类型的特性，又使得使用 nullptr 具有高度的安全性。程序员只需要简单将以前使用 NULL 的地方换成 nullptr 就可以在 C++11 编译器的支持下获得更佳的、更健壮的指针初始化代码。

而 default/deleted 函数则试图在 C++ 缺省函数是否生成上给程序员更多的控制力。通过显式缺省函数，我们可以保证自定义类型符合 POD 的定义，而通过显式删除缺省函数，我们可以禁止 class 的使用者使用不应使用的函数。不过除去 C++ 默认提供的缺省函数外，C++11 标准给予了显式缺省和显式删除更多的能力，比如生成一些并非默认提供的函数的缺省定义，或者禁止一些不必要的隐式类型转换。程序员甚至可以通过删除一些函数来完成编译时期的内存分配的检查。在这样的新特性的支持下，程序员尤其是类的编写者可以更加轻松地保证写出来的类具备所需的特性。

相比于之上两个新特性，lambda 对于 C++ 程序的编写具有更大的冲击力。

早在 C++98 时代，程序员会喜欢使用简单的 STL 部件，如容器等。而使用 STL 的算法（algorithm）的用户相对较少。这一方面固然是由于学习难度较大、简单的算法都可以手工实现造成的，另外一方面的原因则是由于 STL 的算法大多数是依赖于迭代器、仿函数这些较为高级概念造成的。尤其对于仿函数，用户自定义的仿函数需要遵循规则才能被重用。而使用 STL 自带的各种各样的仿函数使用范围和方式过于受限，也使得 STL 良好设计算法的由于“不接地气”而推广受限。

lambda 被设计出来的主要目的之一是简化仿函数的使用，虽然 lambda 的语法看起来不像是“典型的 C++”的，但一旦熟悉了之后，程序员就能准确地完成一个简单的、就地的、带状态的函数定义。虽然 lambda 可以跟仿函数的概念一一对应也实际由仿函数来实现，但理解 lambda 显然比仿函数更加容易。即使没有学习过仿函数的程序员也可以轻松接受 lambda。这样一来，lambda 的出现必然导致 STL 设计、使用方法上的改变，但其最终意义是让更多的程序员无困难地使用 STL 的各种调优后的算法，真正享受 STL 的全部力量。

此外，lambda 作为局部函数也会使得复杂代码的开发加速。通过 lambda 函数，程序员可以轻松地在函数内重用代码，而无需费心设计接口。事实上，曾经出现过一般重构的风潮，在那段时期，C++ 程序员被建议为任何重用的代码创建函数，而 lambda 局部函数的到来则带来了理性思考和折中实现的可能。当然，lambda 函数的出现也导致了函数的作用域在 C++11 中再次被细分，从而也使得 C++ 编程具备了更多的可能。

以上种种或许是 C++ 早就应该做到的，但一直未能实现，现在终于在 C++11 中，lambda 让我们看到了新的光芒。