## 第十一章 选择适当的操作

1. 位逻辑运算符

位逻辑运算符&、|、~、>>、<<作用域整形对象，即char、short、int、long、long long及其对应的unsigned版本，以及bool、wchar\_t、char16\_16、char32\_t等类型。一个普通的enum（而非enum class）可被隐式地转换成整数类型，从而作为位逻辑运算符的运算对象。

# 2017/11/07 Tue

1. 未限定列表（p249）

当我们明确知道所用类型时，可以使用未限定列表。它只能被用作一条表达式，并且仅限于以下场景：

* 函数实参
* 返回值
* 赋值运算符（=、+=、\*=等）的右侧运算对象
* 下标

例如：

int f(double d, Maxtrix& M)

{

int v{7};//初始化器（直接初始化）

int v2 = {7};//初始化器（拷贝初始化）

int v3 = m[{2,3}];//假设m接受一个值对作为其下标

v = {8};//赋值运算符的右侧运算对象

v += {88};//赋值运算符的右侧运算对象

{v} = 9;//错误：不能作为赋值运算符的左侧运算对象

v = 7 +{10};//错误：不能作为非赋值运算符的运算对象

f({10.0});//函数实参

return {11};//返回值

}

我们之所以不允许未定义限定列表出现在赋值运算符的左侧，主要是因为C++语法允许{出现在该位置表示复合语句（块）。

标准库类型initializer\_list<T>用于处理长度可变的{}列表。我们常把他用于用户自定义容器的初始化器列表，但是除此之外也可以直接使用它。例如：

int high\_value(initializer\_list<int> val)

{

int high = numeric\_traits<int>lowest();

if(val.size() == 0) return high;

for(auto i : val)

if (x>high) high = x;

return high;

}

只有当{}列表的所有元素类型相同时，我们才能推断该列表的类型。例如：

auto x0 = {};//错误（缺少元素类型）

auto x0 = {1};//initializer\_list<int>

auto x0 = {1,2};// initializer\_list<int>

auto x0 = {1,2,3};// initializer\_list<int>

auto x0 = {1,2.0};//错误:元素类型不完全相同

可惜，我们无法通过推断未定义列表的类型使其作为普通模板的实参。

template<typename T>

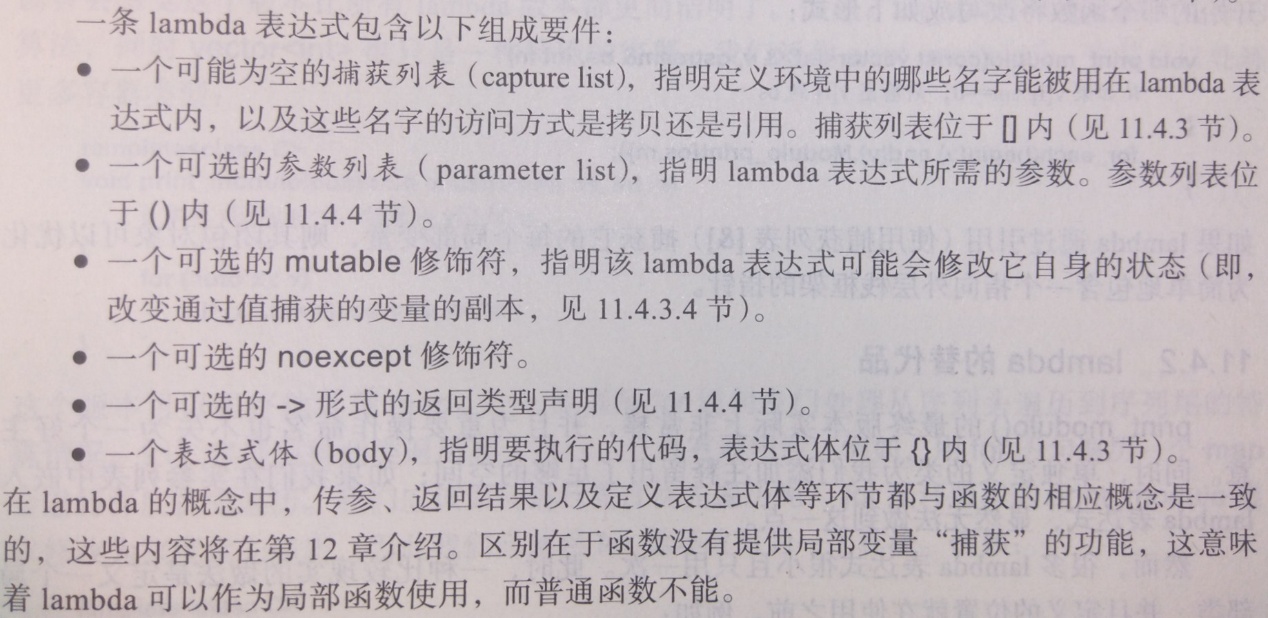
void f(T);

f({});//错误：初始化器的类型未知

f({1});//错误：未限定的列表与“普通的T”不匹配

1. lambda表达式（p251）

lambda表达式（lambda expression）有时也称为lambda函数（lambda function），或者直接简称为lambda（严格来说或者并不正确，但符合口语习惯）。它是定义和使用匿名函数对象的一种简便方式。人们习惯的方式是先定义一个含有operator()的命名类，随后再创建该类的一个对象并通过该对象调用函数；相比之下，lambda表达式就像文字速记法一样简单易行。尤其是当我们想把操作当成实参传给算法时，这种便捷性明显的尤其重要。



# 2017/11/08 Wed

1. 如果我们确实需要一个名字，那就命名lambda：（p254）

void print\_modulo(cosnt vector<int>& v, ostream& os, int m)

{

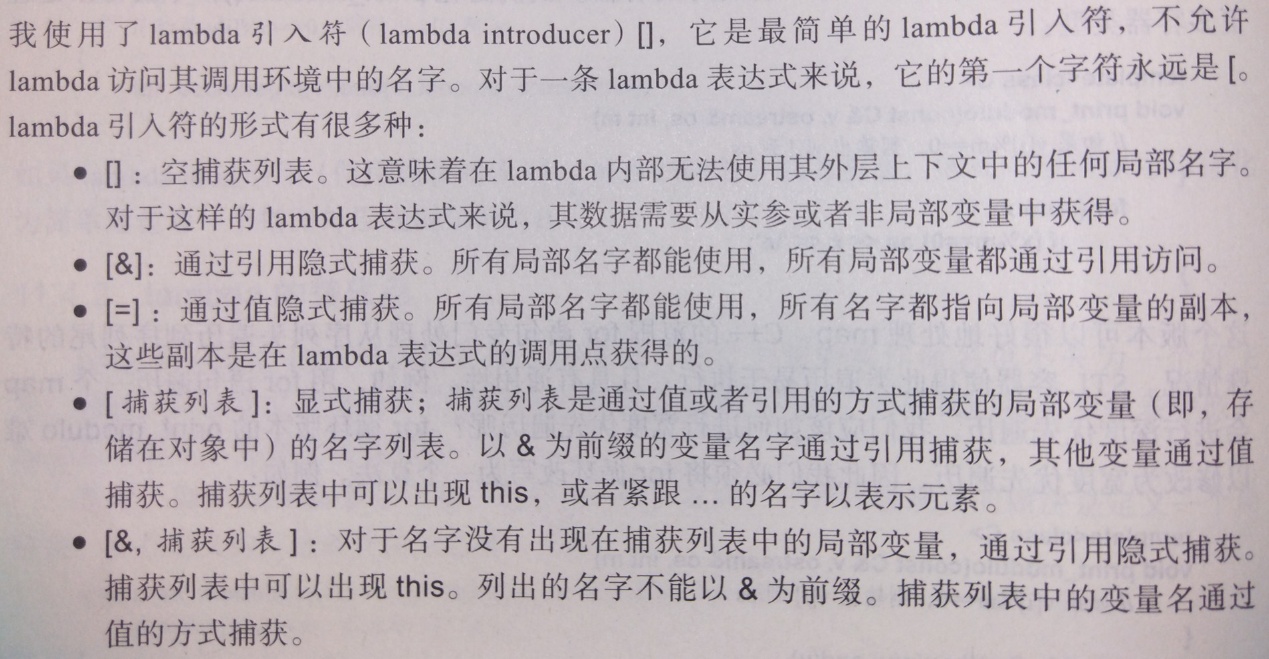
auto Modulo\_print = [&os, m](int x){ if(x%m==0) os<<x<<’\n’; }

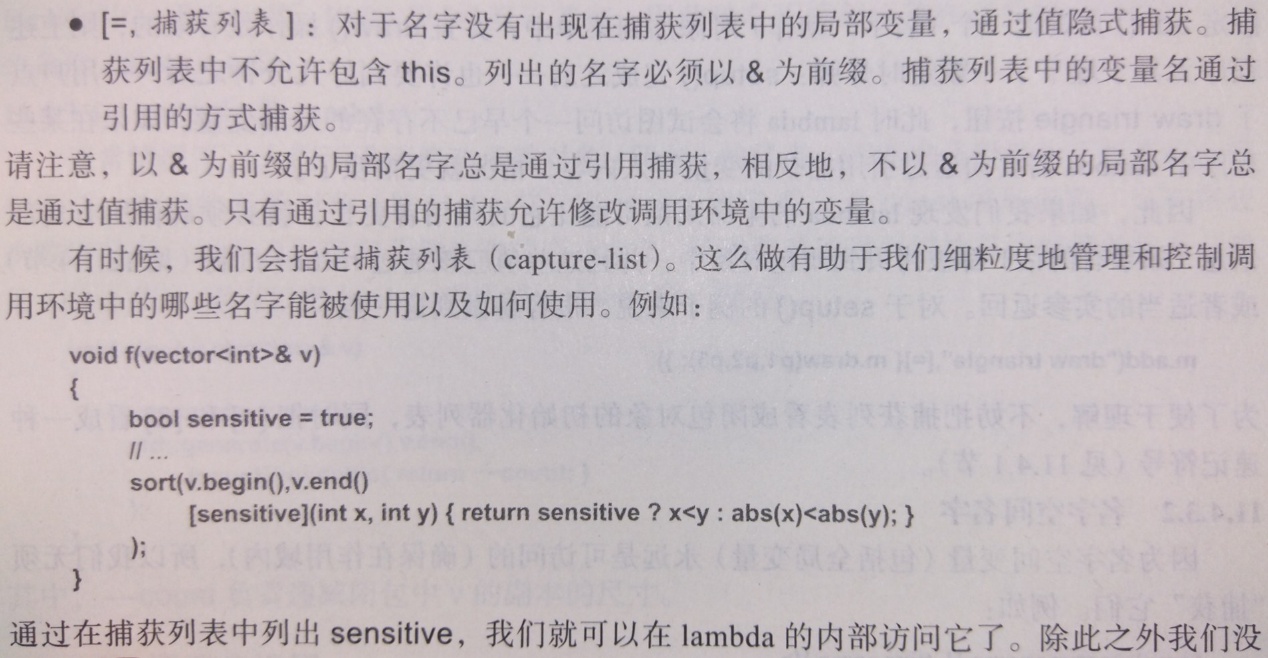
for\_each(begin(v), end(v), Modulo\_print);

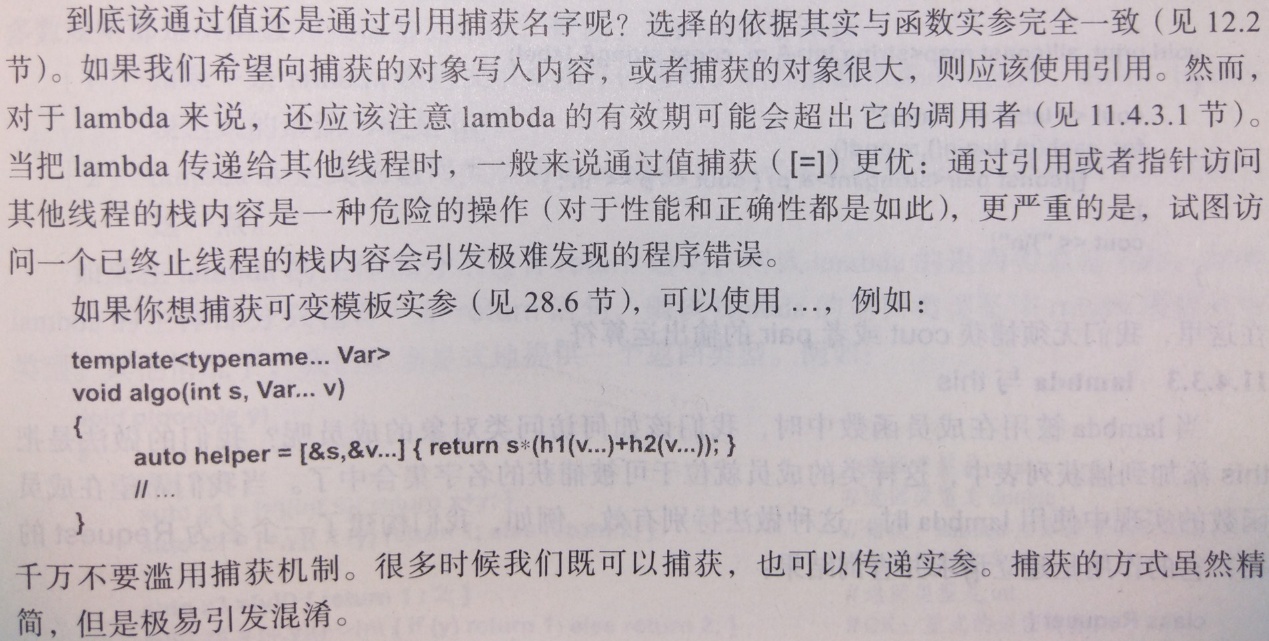
}

通常情况下，命名lambda是一种有效的手段。这么做可以让我们把注意力集中在如何设计操作本身上，同时，代码布局更加简单，也能使用递归了。

1. 捕获







1. lambda与生命周期（p255）

lambda的生命周期可能比它的调用者更长。当我们把lambda传递给另一个线程或者被调用者吧lambda存在别处以供后续使用时，这种情况就会发生。例如：

void setup(menu& m)

{

//…

Point p1,p2,p3;

//…计算p1,p2和p3的位置…

m.add(“draw triangle”, [&]{ m.draw(p1,p2,p3); });//可能发生程序错误

//…

}

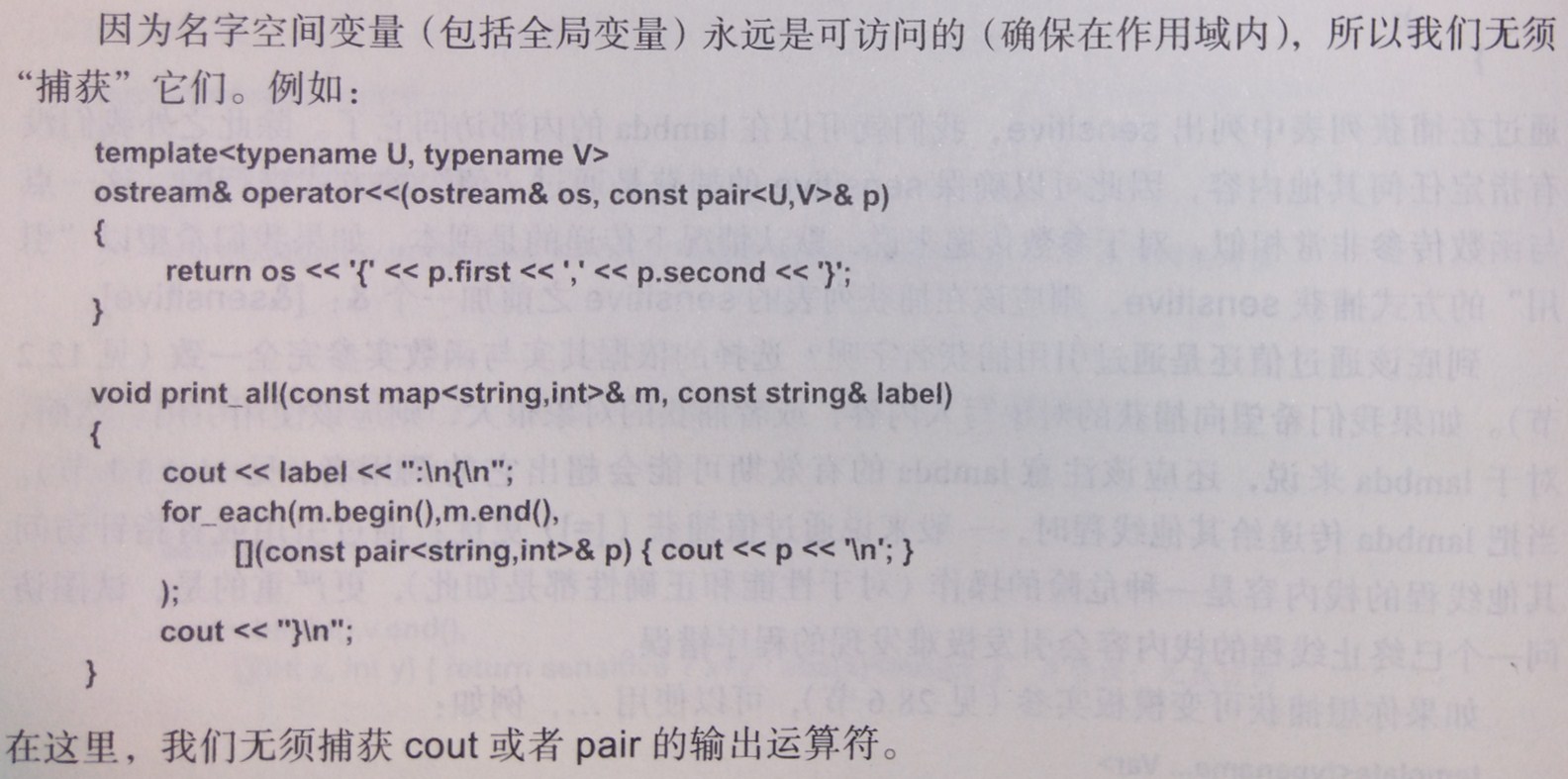
假设add()负责把一个（名字、动作）对添加到菜单中，并且draw()操作是有效的，则上述程序无异于买下了一颗定时炸弹:setup()完成之后—也许要到好几分钟后—用户点击了draw triangle按钮，此时lambda将会视图访问一个早已不存在的局部变量。如果在某些程序中lambda需要向通过引用捕获的变量写入内容，情况就更糟糕了。

因此，如果我们发现lambda的生命周期可能比它的调用者更长，就必须确保所有的局部信息（如果有的话）都被拷贝到闭包对象中，并且这些值应该通过return机制或者适当的实参返回。对于setup()的例子来说，很容易做到这一点：

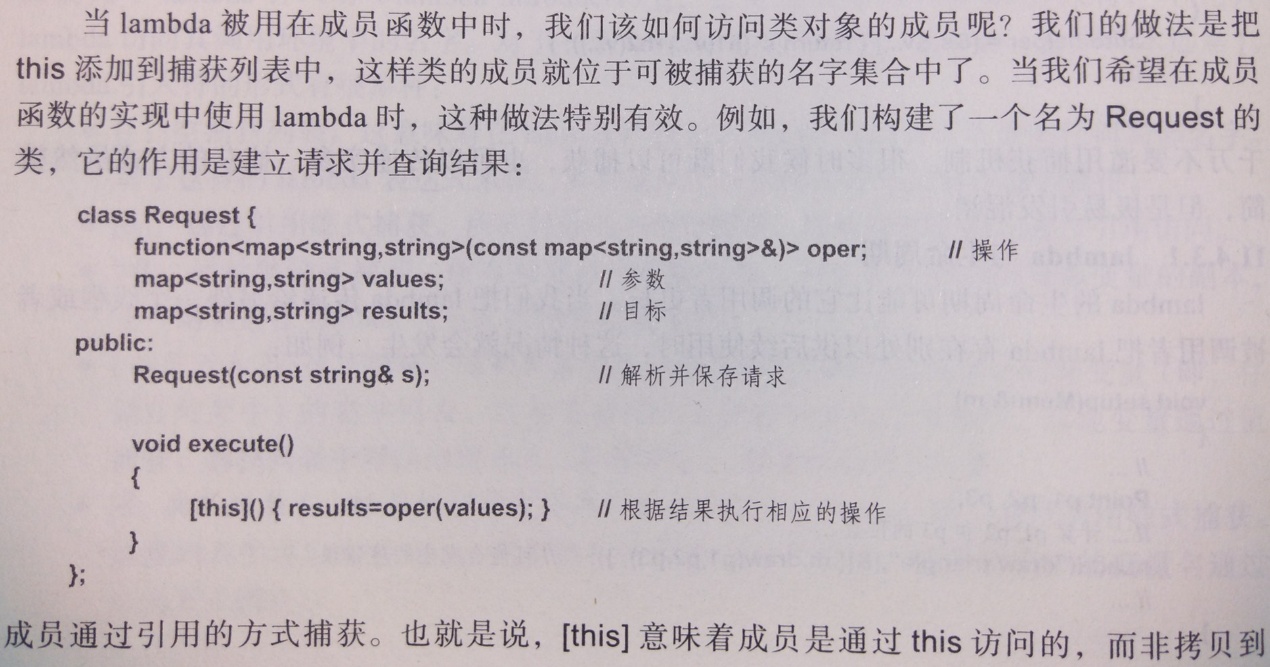
m.add(“draw triangle”, [=]{ m.draw(p1,p2,p3); });

为了便于理解，不放吧捕获列表看成闭包对象的初始化器列表，同时把[=]和[&]看成一种速记符号。

1. 名空间名字（p256）



1. lambda与this（p256）



lambda中。不幸的是，[this]和[=]不兼容，因此稍有不慎就可能在多线程中产生竞争的条件。

1. mutable的lambda

通常情况下，人们不希望修改函数对象（闭包）的状态，因此默认设置为不可修改。换句换说，生成的函数对象的operator()()是一个const成员函数。只有在极少数情况下，如果我们确实希望修改状态（注意，不是修改通过引用捕获的变量的状态），则可以把lambda声明称mutable的。例如：

void algo(vector<int>& v)

{

int count = v.size();

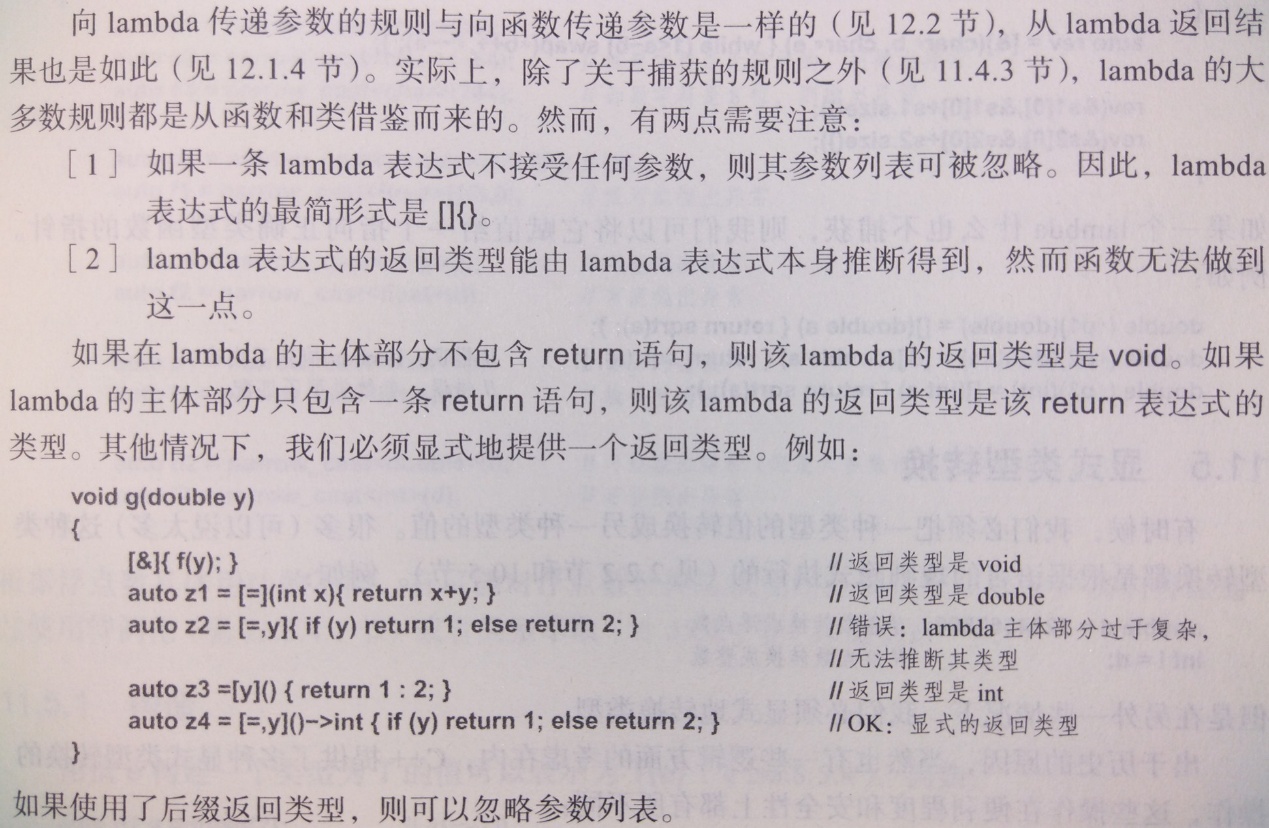
std::generate(v.begin, v.end,

[count]mutable{ return –count; }

);

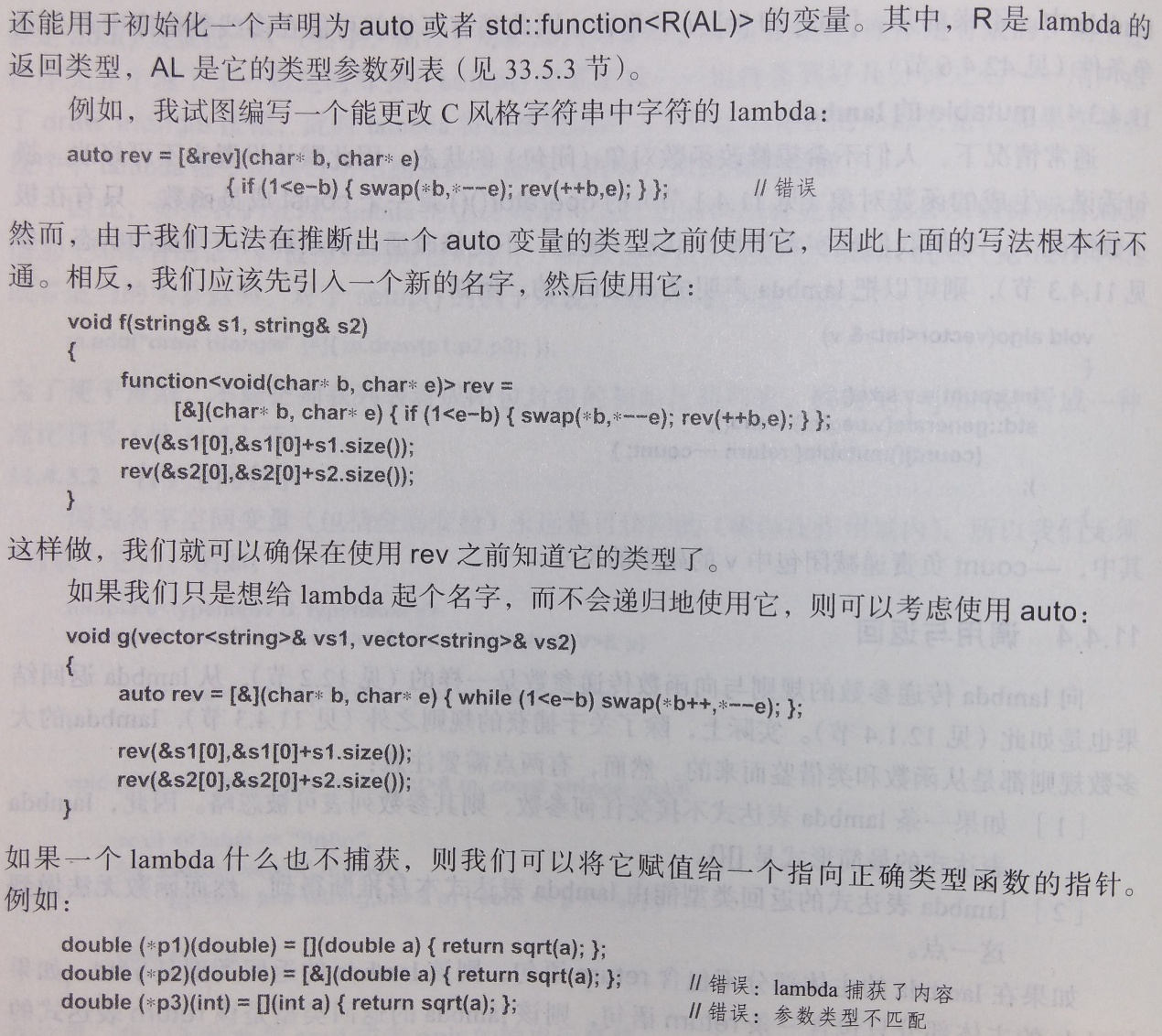
}

1. 调用与返回（p257）



# 2017/11/09 Thu

1. lambda的类型



1. 显示类型转换（p258）

C++提供了多种显示类型转换的操作，这些操作在便利程度和安全性上有所不同：

* 构造，使用{}符号提供新值类型安全的构造
* 命名的转换，提供不同等级的类型转换：
  + const\_cast，对某些声明为const的对象获得写入的权利
  + static\_cast，反转一个定义良好的隐式类型转换
  + reinterpret\_cast，改变位模式的含义
  + dynamaic\_cast，动态地检查类型层次关系
* C风格的转换，提供命名的类型转换或者其组合
* 函数化符号，提供c风格转换的另一种形式

除了{}构造符号之外，我对上面这些转换没有好感。当然，至少dynamic\_cast执行了运行时检查。对于发生在两中标量数字类型转换，我更倾向于使用一个自制的显示类型转换函数narrow\_cast；此时，值可能会被窄化：

template<class Target, class Source>

Target narrow\_cast(Source v)

{

auto r = static\_cast<Target>(v);//把值转化成目标类型

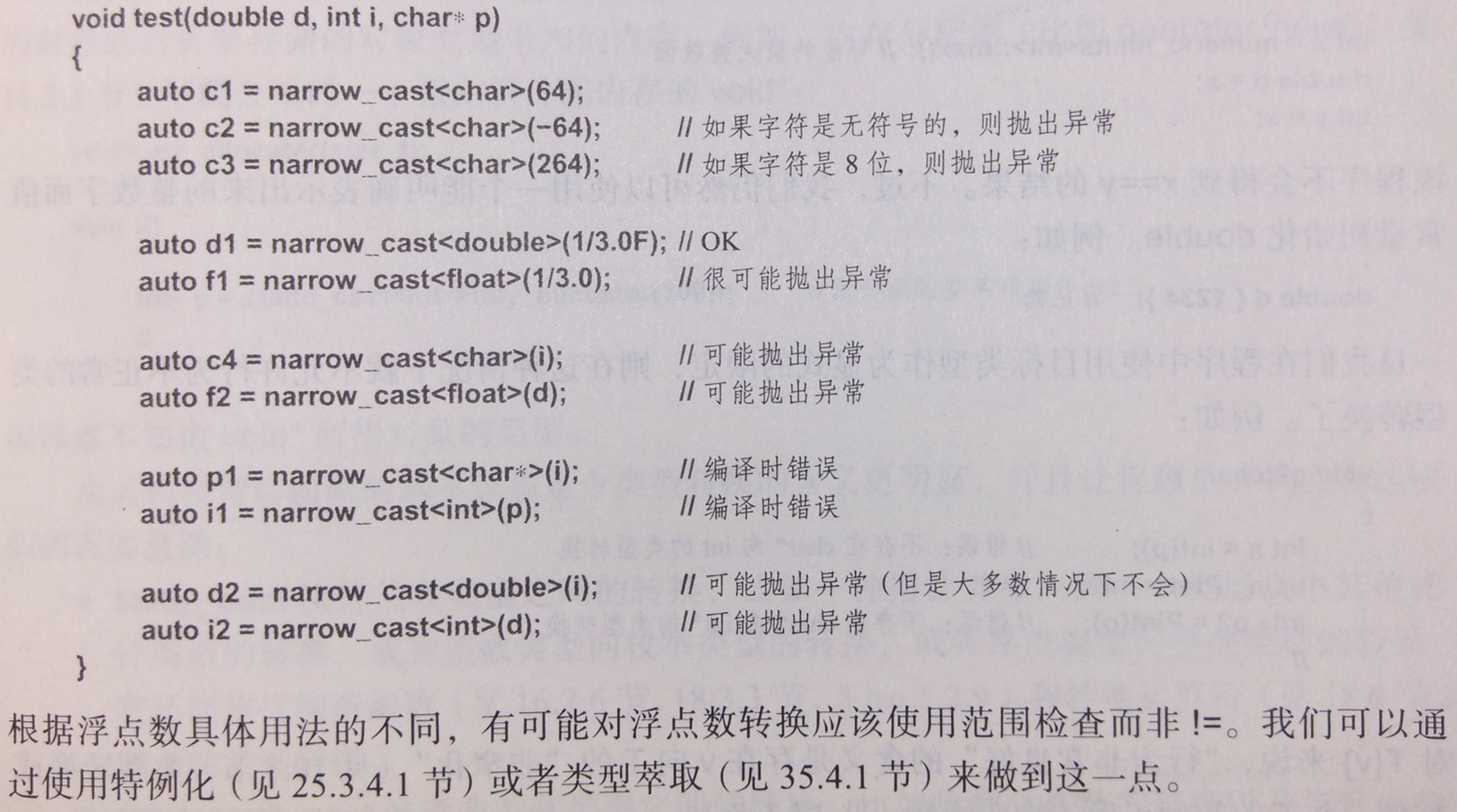
if(static\_cast<Scource>(r) != v)

throw runtime\_error(“narrow\_cast<>() failed”);

return r;

}

也就是说，如果把某个值转化成目标类型而在这个过程中发生了窄化运算，则把结果转换原类型，并且恢复原值。



1. 构造（p259）

用值e构件一个类型为T的值可以表示为T{e}，例如：

auto d1 = double{2};//d1 == 2.0

double d2 {double{2}/4}; //d2 == 0.5

符号T{V}有一个好处，就是它只执行“行为良好的”类型转换。例如：

void f(int);

void f(double);

void g(int i, double d)

{

f(i);//调用f(int)

f(double{i});//错误：{}拒绝执行整数想浮点数的类型转换

f(d);//调用f(double)

f(int{d});//错误：{}拒绝截断的行为

f(static\_cast<int>(d));//调用f(int)，传入的是一个截断的值

f(round(d));//调用f(double)，传入的是一个四舍五入的值

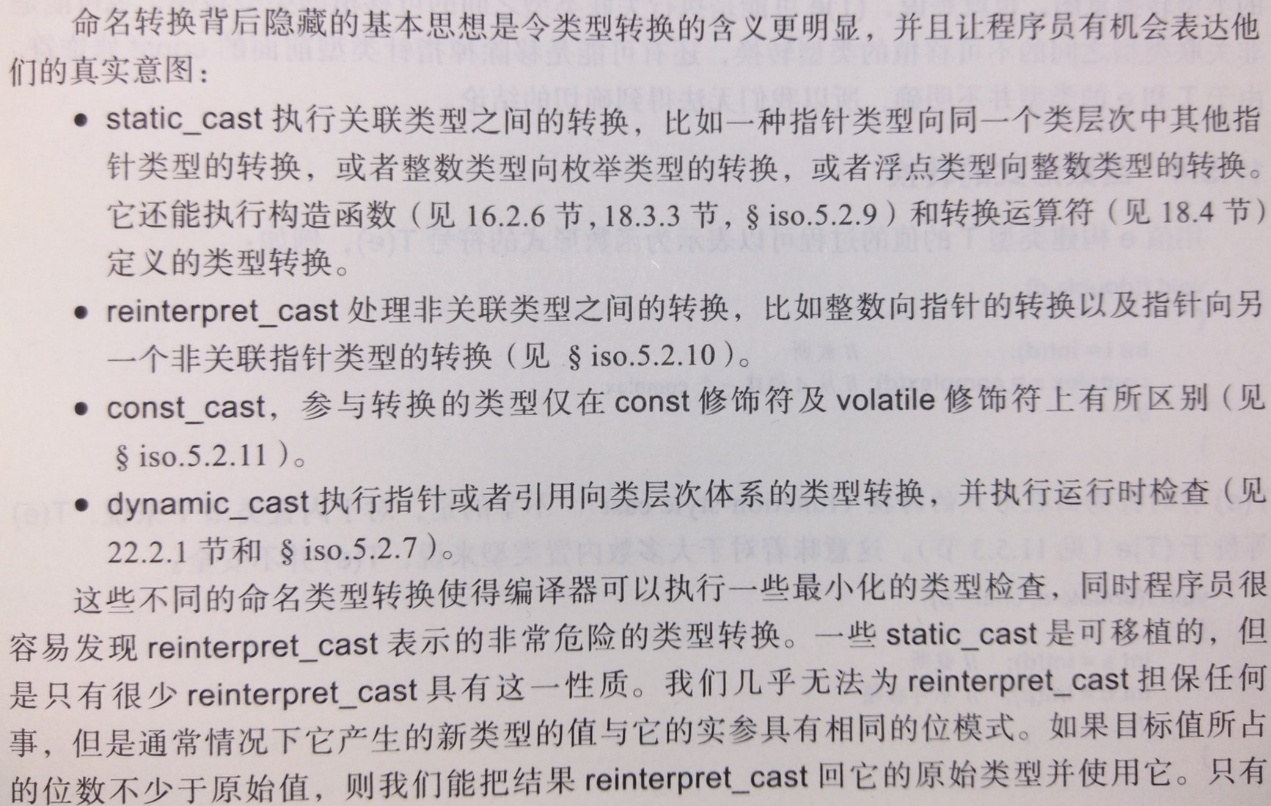
f(static\_cast<int>(lround(d)));//调用f(int)，传入的是一个四舍五入的值

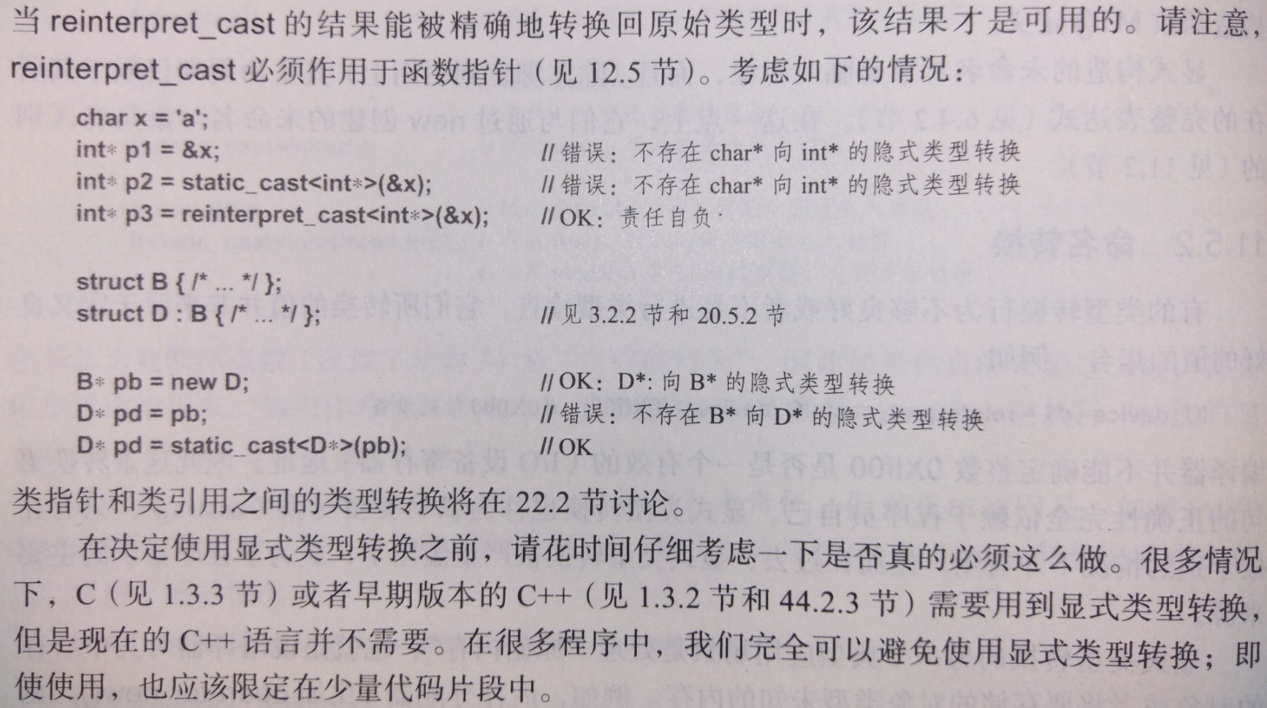
//如果round(d)溢出int的范围，他仍会被截断

}



1. 命名转换





1. 函数形式的转换（p262）

用值e构件类型T的值的过程可以表示为函数形式的符号T(e),例如：

void f(double d)

{

int i = int(d);//截断

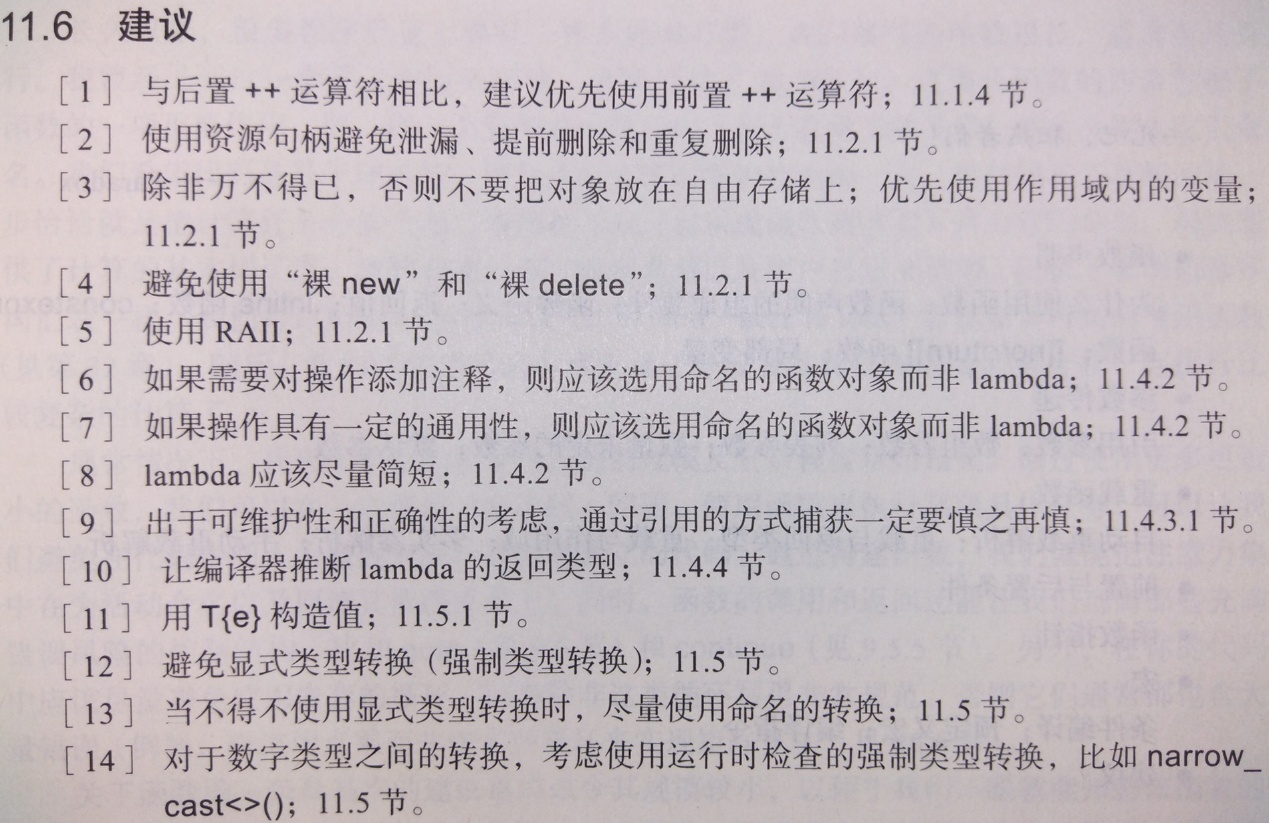
complex z = complex(d);//从d构建一个complex

//…

}

T(e)有时称为函数形式的转换（function-style cast）。不幸的是，对于内置类型T来说，T(e)等价于(T)e。这意味着对于大多数内置类型来说。T(e)并不安全。

1. 建议（p263）



# 2017/11/10 FRI

## 第十二章 函数

1. constexpr，表示当给定常量表达式作为实参时，应该可以在编译时对函数求值。

noexcept，表示该函数不允许抛出异常。

[[noreturn]]，表示该函数不会用常规的调用/返回机制返回结果。

成员函数好能被限定为：

* virtual，表示该函数可以被派生类覆盖
* override，表示该函数必须覆盖基类中的一个虚函数
* final，表示该函数不会被派生类覆盖
* static，表示该函数不与某一特定的对象关联
* const，表示该函数不能修改其对象的内容。

如果你想让代码的读者头疼，不放把函数声明成下面的形式：

struct S

{

[[noreturn]] virtual inline auto f( const unsigned long int \* const )-> void const no except;

};

1. 函数定义（p266）

函数定义是一种特殊的函数声明，他给出了函数体的内容。

函数的定义以及声明必须对应同一类型。不过，为了与C语言兼容，我们会自动忽略参数类型的顶层const。例如，下面两条声明语句对应的是同一个函数：

void f(int); //类型是void(int)

void f(const int);//类型是void(int)

函数f()可以定义成：

void f(int x) { /\* 允许在此处修改x \*/}

或者定义成：

void f(const int x) { /\* 不允许在此处修改x \*/ }

函数参数名字不属于函数类型的一部分，不同的声明语句中参数的名字不许保持一致。例如：

int& max(int& a, int& b, int& c);

int& max(int& x1, int& x2, int& x3)

{

return (x1>x2)？((x1>x3)?x1:x3): ((x2>x3)?x2:x3);

}

一般来说，未命名的参数有助于简化代码并提升代码的可扩展性。此时，尽管某些参数未被使用，但是为其预留位置可以确保函数的调用者不会受到未来函数变化的影响。

void search(table\* t, const char\* key, const char\*)

{

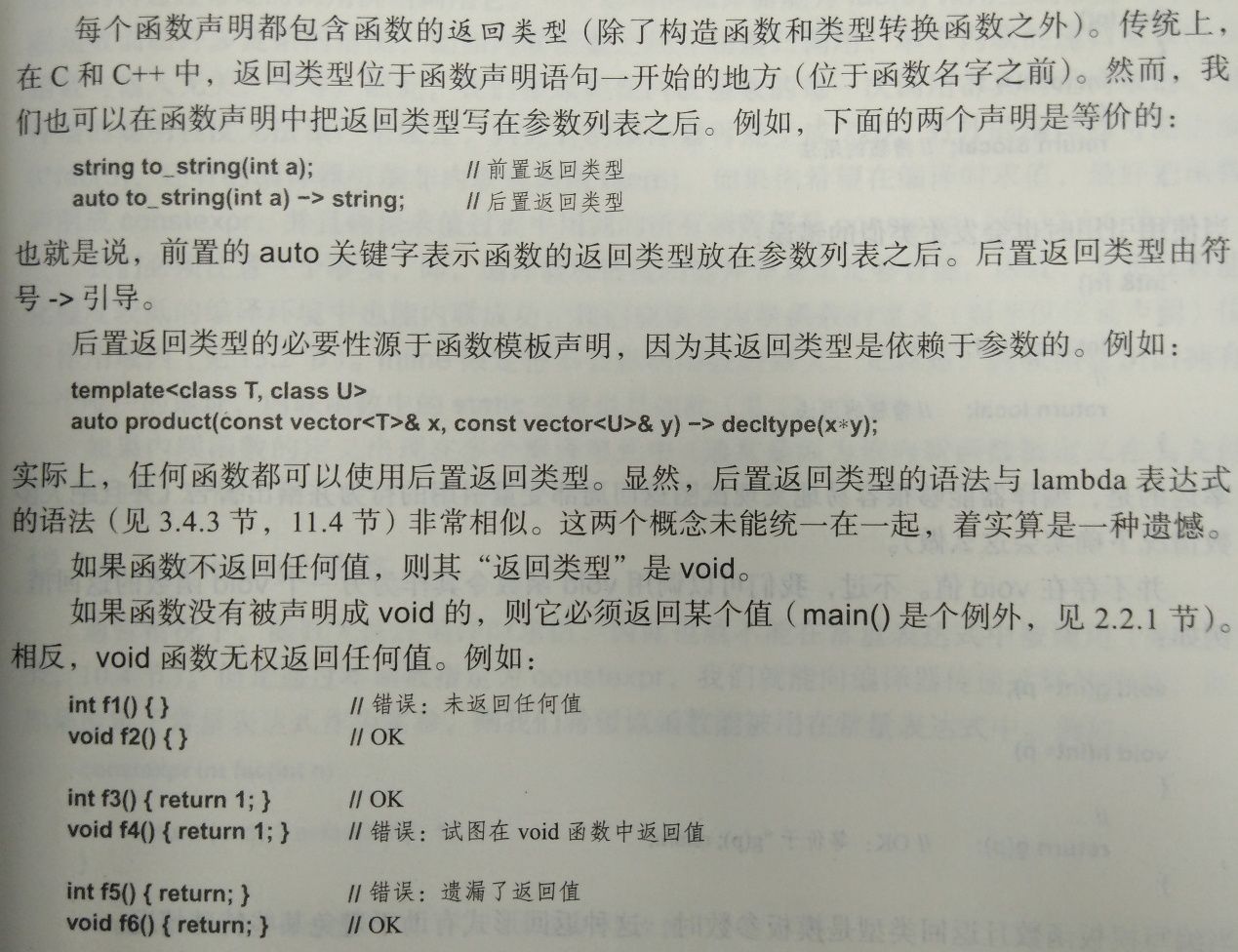
//未用到第三个参数

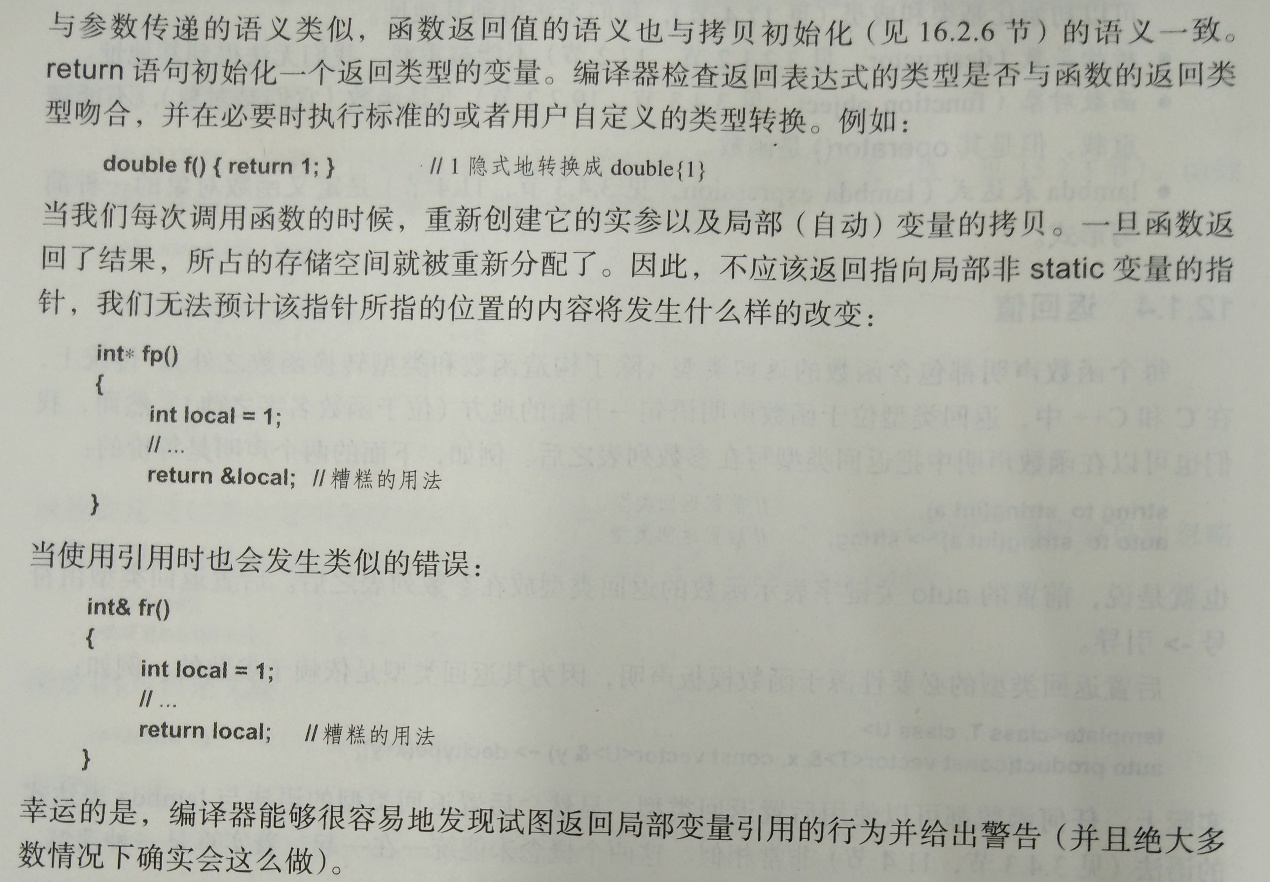
}

除了函数以外，我们还能调用其他一些东西，他们遵循函数的大多数规则，比如参数传递规则：

* 构造函数（constrictor）严格来说不是函数，他没有返回值，可以初始化基类和成员，我们无法得到其地址。
* 析构函数（destructor）不能被重载，我们无法的到其地址。
* 函数对象（funtion object）不是函数（他们是对象），不能被重载，但是其operator()是函数
* lambda表达式（lambda expression）是定义函数对象的一种简写形式。

1. 返回值





在函数中，return语句的形式属于下属5中之一：

* 执行一条return语句。
* “跳转到函数末尾”，也就是说，直接到达函数体的末端。这种情况值允许出现在无返回值的函数以及main()中。此时，跳转到函数末尾意味着函数被成功地执行了。
* 抛出一个未被局部捕获的异常
* 在一个noexcept函数中抛出一个异常并且没有局部捕获，从而造成程序终止。
* 直接或者间接地请求了一个无返回值的系统函数（如exit()）。

我们可以将一个未按常规方式（即，通过return或者“跳转到函数末尾”）返回的函数标记为[[noreturn]]。

# 2017/11/12 Sun

1. constexpr函数（p269）

通常情况下，函数无法在编译时求值，因此也不能在常量表达式中被调用。但是通过将函数指定为constexpr，我们就能向编译器传递这样的信息，即，如果给定了常量表达式作为实参，则我们希望该函数能被用在常量表达式中。例如：

constexpr int fac(int n)

{

return (n-1)?n\*fac(n-1):1;

}

constint f9 = fac(9);//必须在编译时求值

当constexpr出现在函数定义中时，他的含义是“如果给定了常量表达式作为实参，则该函数应该能用在常量表达式中”。而当constexpr出现在対象定义中时，它的含义时“在编译时对初始化器求值”。例如：

void f(int n)

{

int f5 = fac(5);//可能在编译时求值

int fn = fac(n);//在运行时求值（n是变量）

constexpr int f6 = fac(6);//必须在编译时求值

constexpr int fnn = fac(n);//错误：无法确保在编译时求值（n是变量）

char a[fac(4)];//OK，数组的尺寸必须是常量，而fac()恰好是constexpr

char a2[fac(n)];//错误：数组的尺寸必须是常量，而n是一个变量

}

函数必须足够简单才能在编译时求值：constexpr函数必须包含一条独立的return语句，没有循环，也没有局部变量。同时，constexpr函数不能有副作用。也就是说，constexpr函数应该是以个纯函数。例如：

int glob;

constexpr void bad1(int a)//错误：constexpr函数不能是void

{

glob = a;//错误：在constexpr函数中有副作用

}

constexpr int bad2(int a)

{

if(a>=0) return a; else return –a;//错误：在constexpr函数中有if语句

}

constexpr int bad3(int a)

{

sum = 0;//错误：在constexpr函数中有局部变量

for(int i=0; i<a; ++i) sum += fac(i);//错误：constexpr函数中有循环

return sum;

}

与普通的constexpr函数相比，constexpr构造函数的规则有所区别：只允许简单地执行成员函数初始化操作。

constexpr函数允许递归和条件表达式。这意味着如果你确实希望某个函数是constexpr，就一定能做到。随之而来的结果就是，你必须严格遵循constexpr函数使用习惯，将其用于相对简答的任务。一旦有所违背，调用工作就会变得异常困难，并且编译的时间会更长。

1. 数组的sizeof（临时自己添加的知识点）    数组的sizeof值等于数组所占用的内存字节数，如： char a1[] = "abc"; int a2[3]; sizeof( a1 ); // 结果为4，字符 末尾还存在一个NULL终止符 sizeof( a2 ); // 结果为3\*4=12（依赖于int）

    一些朋友刚开始时把sizeof当作了求数组元素的个数，现在，你应该知道这是不对的，那么应该怎么求数组元素的个数呢Easy，通常有下面两种写法：int c1 = sizeof( a1 ) / sizeof( char ); // 总长度/单个元素的长度 int c2 = sizeof( a1 ) / sizeof( a1[0] ); // 总长度/第一个元素的长度

1. constexpr与引用

constexpr函数不允许有副作用，因此我们不能行非局部对象写入内容。反过来说，只要我们不向非局部对象写入内容，就能使用它。

constexpr int ftbl[] = {1,2,3,5,8,13};

constexpr int fib(int n)

{

return ( n<sizeof(ftbl)/sizeof(\*ftbl) ? ftbl[n] : fib(n));

}

constexpr函数可以接受引用实参。尽管它不能通过这些引用写入内容，但是const应用参数同样有用。例如，我们在表批准库中发现

template<>class complex<flot>

{

public:

//…

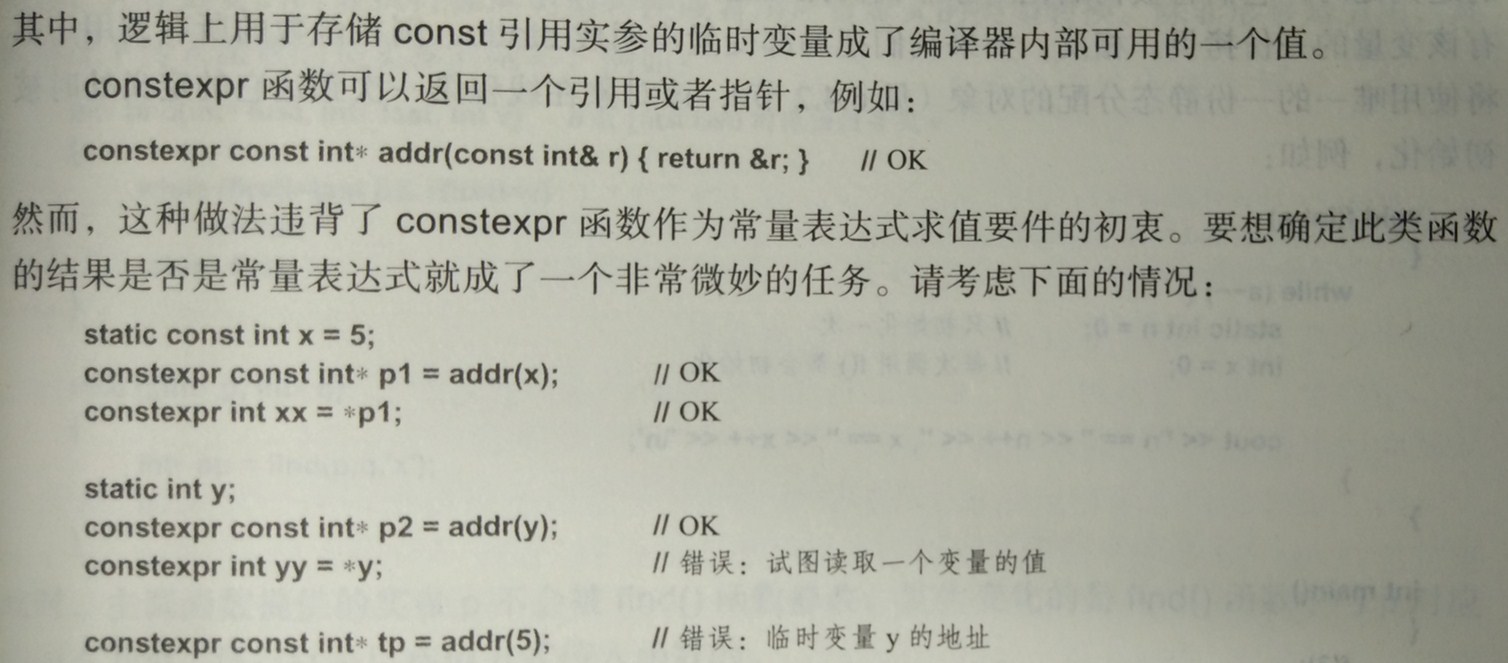
explicit constexpr complex(const complex<double>&);

//…

}

这允许我们编写：

constexpr complex<float> z{2.0};



1. [[noreturn]]函数

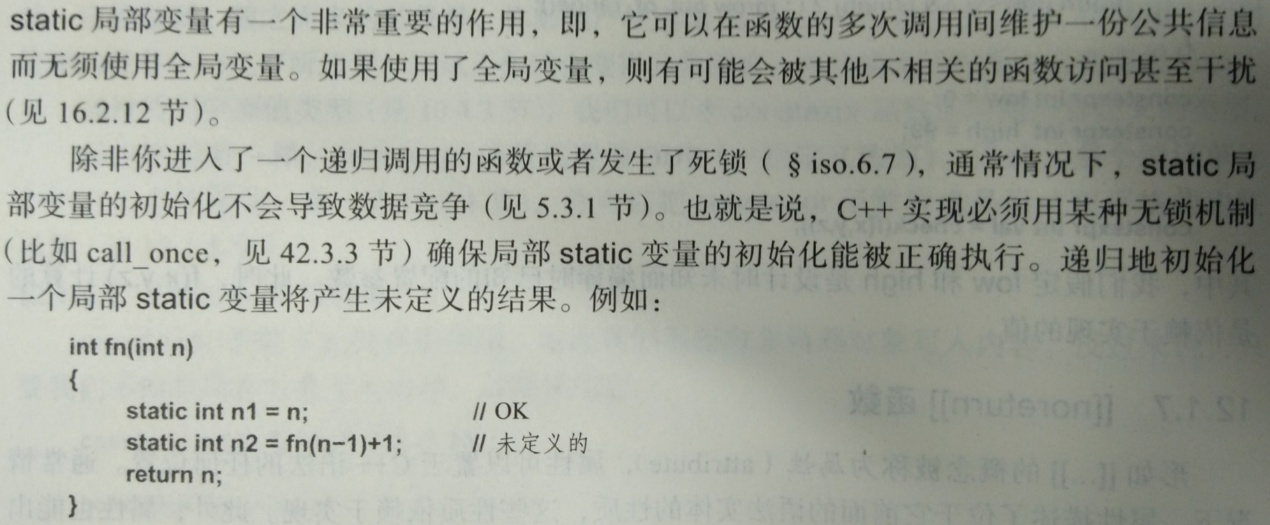
形如[[…]]的概念被称为属性（attribute），属性可以置于c++语法的任何位置。通常情况下，属性描述了位于它前面的语法实体的性质，这些性质赖于实现。此外，属性也能出现在声明语句的开始位置。c++只包含两个标准属性，[[noreturn]]是其中之一，另一个是[[carries\_dependency]]。

把[[noreturn]]放在函数声明语句的开始位置表示我们不希望该函数返回任何结果。例如：

[[noreturn]] void exit(int);//exit永远不会返回任何结果

如果我们预先知道某个函数不会返回结果，对于理解程序和生成代码都很有益处。如果函数被设定为[[noreturn]]，但是在干函数的内部任反悔了某个值，将产生未定义行为。

1. 局部变量



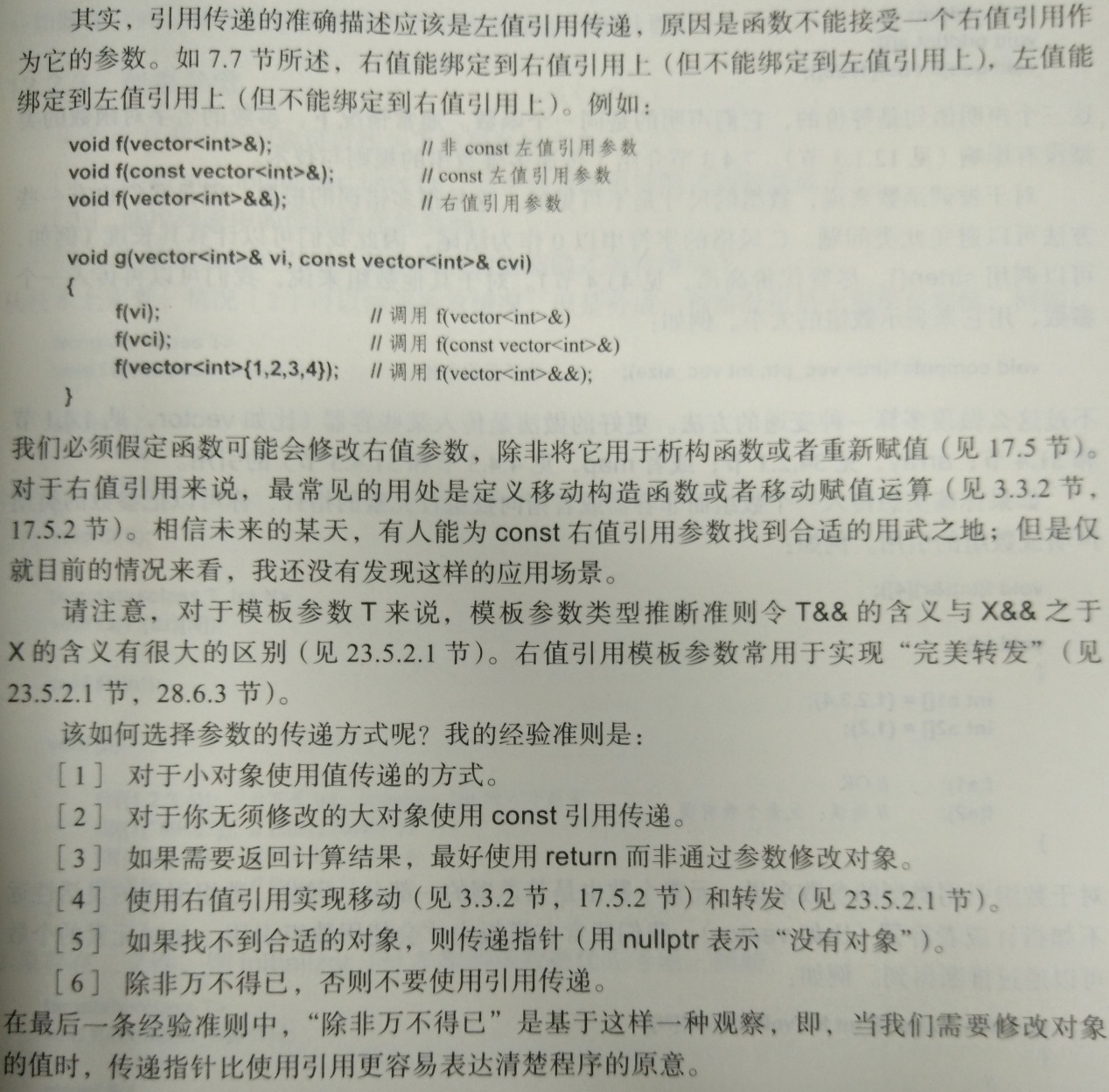
static局部变量有助于避免非局部变量间的顺序依赖

不存在“局部函数”；如果你觉得自己需要一个类似地东西，不妨使用函数对象或者lambda表达式。

1. 参数传递

当程序调用一个函数时（使用后缀()，称为调用运算符call operator或者应用运算符application operator），我们为该函数的形参（formal arguments，即， parameters）申请内存空间，并用实参（actual argument）初始化对应的形参。

1. 引用参数



1. 数组参数（p275）

当数组作为函数的参数时，世纪传入的是该函数首元素的指针。

数组类型的参数与指针类型的参数等价，例如：

void odd(int\* p);

void odd(int a[]);

void odd(int buff[1020]);

这三个声明语句是等价的，它声明的是同一个函数。通常情况下，参数的名字对函数的类型没有影响。

如果你确实想传入一个数组而非容器或者指向数组首元素的指针，你可以把参数的类型声明成数组的引用。例如：

void f(int(&r)[4]);

void g()

{

int a1[] = {1,2,3,4};

int a2[] = {1,2};

f(a1);//OK

f(a2);//错误：元素个数有误

}

对于数组引用类型的参数来说，元素个数也是其类型的一部分。因此数组引用的灵活性远不如指针或者容器（比如vector）。我们通常在模板中才使用数组引用，此时元素的个数可以通过推断得到。例如：

template<class T, int N>

void f(T(&r)[N])

{

//…

}

int a1[10];

double a2[100];

void g()

{

f(a1);//T是int；N是10

f(a2);//T是double；N是100

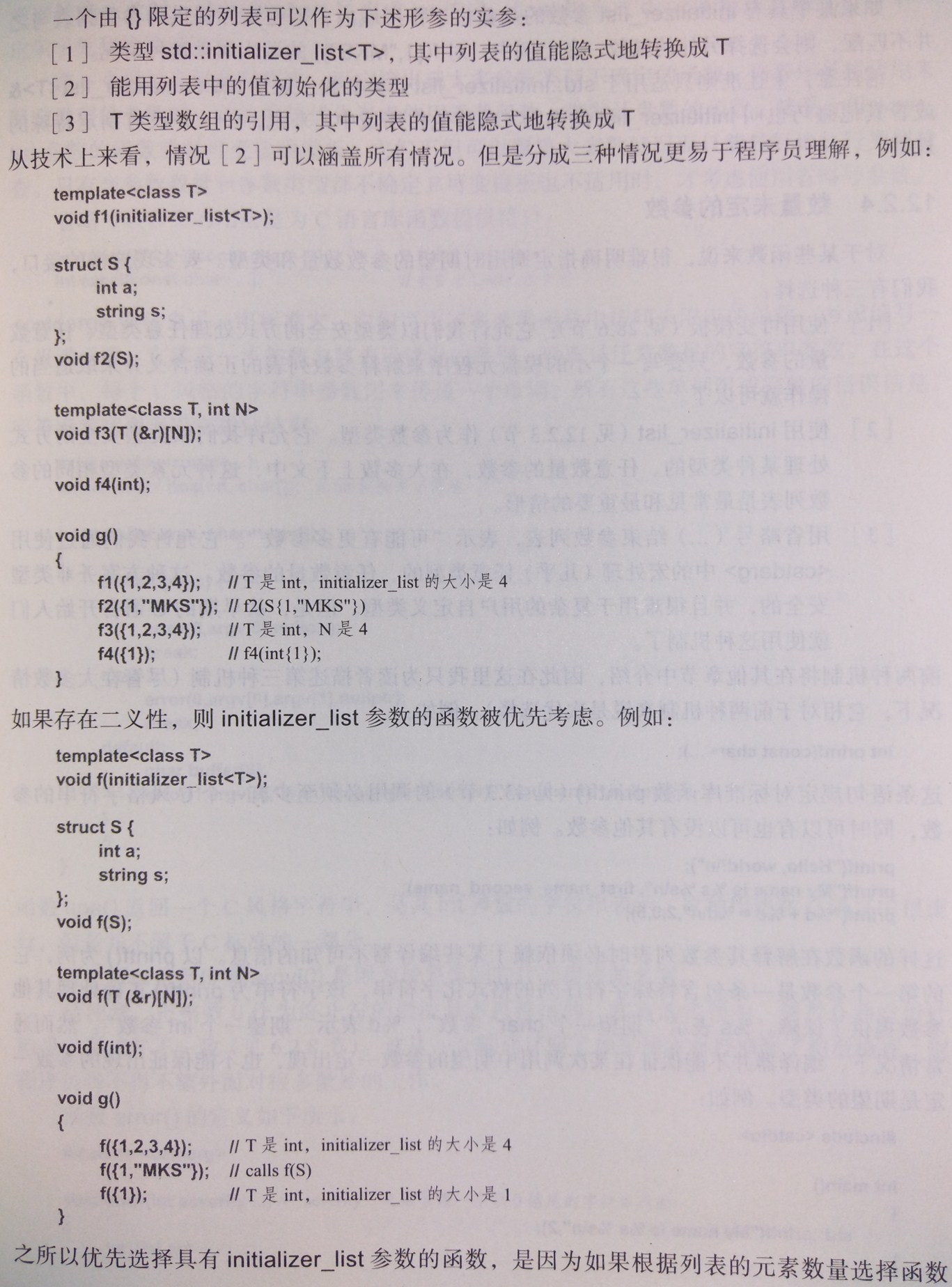
}

这么做的后果是调用f()所用的数组类型有多少个，对应的函数定义就有多少个。

1. typedef用法（自己增加）

typedef int array[10]，这个代表什么意思，这可不是平时那种array[10】是个int了吧，这个代表，array是个重定义的类型，array a； 就代表a是一个含有10个int型元素的数组，这么定义可能觉得简化不了多少，还不如直接int a[10];看着明白呢，好! 继续往下看,typedef array Array[5]; 这个呢，如果我这么定义一个变量 Array arr[2]; 这个又是什么意思呢？这个就是定义个Array类型数组，其中这个“一维”数组有两个元素，但是每个元素又都是Array 类型的，每个Array又是一个5行10列二维数组，所以arr就是一个三维数组，他就等于int arr[2][5][10]; 是不是看的有点蒙了， 其实在C语言中根本不存在多维数组，全部都是一维数组，只是一维数组里面又含一维数组，所以才构成了所谓的“多维”数组， 如果这样理解的话，是不是觉得多维数组也就不那么“神奇”了，变得简单了，没有那么晕了，这样定义的好处就是可以讲一个多维的数组转化成我们比较熟悉的一维数组，这样操作起来就比较容易了，可读性自然也会增强。所以这就是他的“简化”的作用。其实我们还可以在函数指针上体现，想定义个函数指针int (\*p)(); 我们可以定义为typedef int (\*POWER)(); 接下来可以直接定义POWER p1, p2; 他就等价于int (\*p1)(), int (\*p2)(); 这样也非常易懂。

1. 列表参数（p277）



### 数量未定的参数（p278）

对于某些函数来说，很难明确指定调用时期望的参数数量和类型。要实现这样的接口，我们有三种选择：

1. 使用可变模板：它允许我们以类型安全的方式处理任意类型、任意数量的参数，只要写一个小的模板元程序来解释参数列表的正确含义并采取适当的操作就可以了。
2. 使用initializer\_list作为参数类型。它允许我们以类型安全的方式处理某种类型的、任意数量的参数，在大多数上下文中，这种元素类型相同的参数列表是最常见和最重要的情形。
3. 用省略号（…）结束参数列表，表示“可能有更多参数”。它允许我们通过使用<cstdarg>中的红处理（几乎）任意类型的、任意数量的参数。这种方案并非类型安全的，并且很难用于复杂的用户自定义类型。但是，从早期的c语言开始人们就使用这种机制了。

int printf(const char\*…);

这条语句规定对标准库函数pringf()的调用必须至少有一个C风格字符串的参数，同时可以有也可以没有其他参数。例如：

printf(“Hello, world!/n”);

printf(“My name is %s %s \n”, first\_name, second\_name);

printf(“%d +%d = %d\n”, 2,3,5);

在一个设计良好的程序中，不应该出现太多参数类型不确定的函数。当程序员想使用未限定类型的参数时，其实应该优先考虑使用重载函数、带默认参数的函数、接受initializer\_list参数的函数或者可变参数模板，他们不但可以覆盖大多数情况而且能教好地执行类型检查。只有当参数数量和参数类型都不确定且可变模板也不适合时，才考虑使用省略号参数。

省略号最常见的用法是为c语言库函数提供接口：

int fprintf(FILE\*, const char\* …);//来自于<cstdio>

int execl(const char\* …);//来自于UNIX头文件

#include <cstdarg>

void error(int severity …)//“severity”之后紧跟一个以0结尾的字符串列表

{

va\_list ap;

va\_start(ap, severity);//arg启动

for(;;)

{

char\* p = va\_arg(ap, char\*);

if(p == nullptr) break;

cerr<<p<<;

}

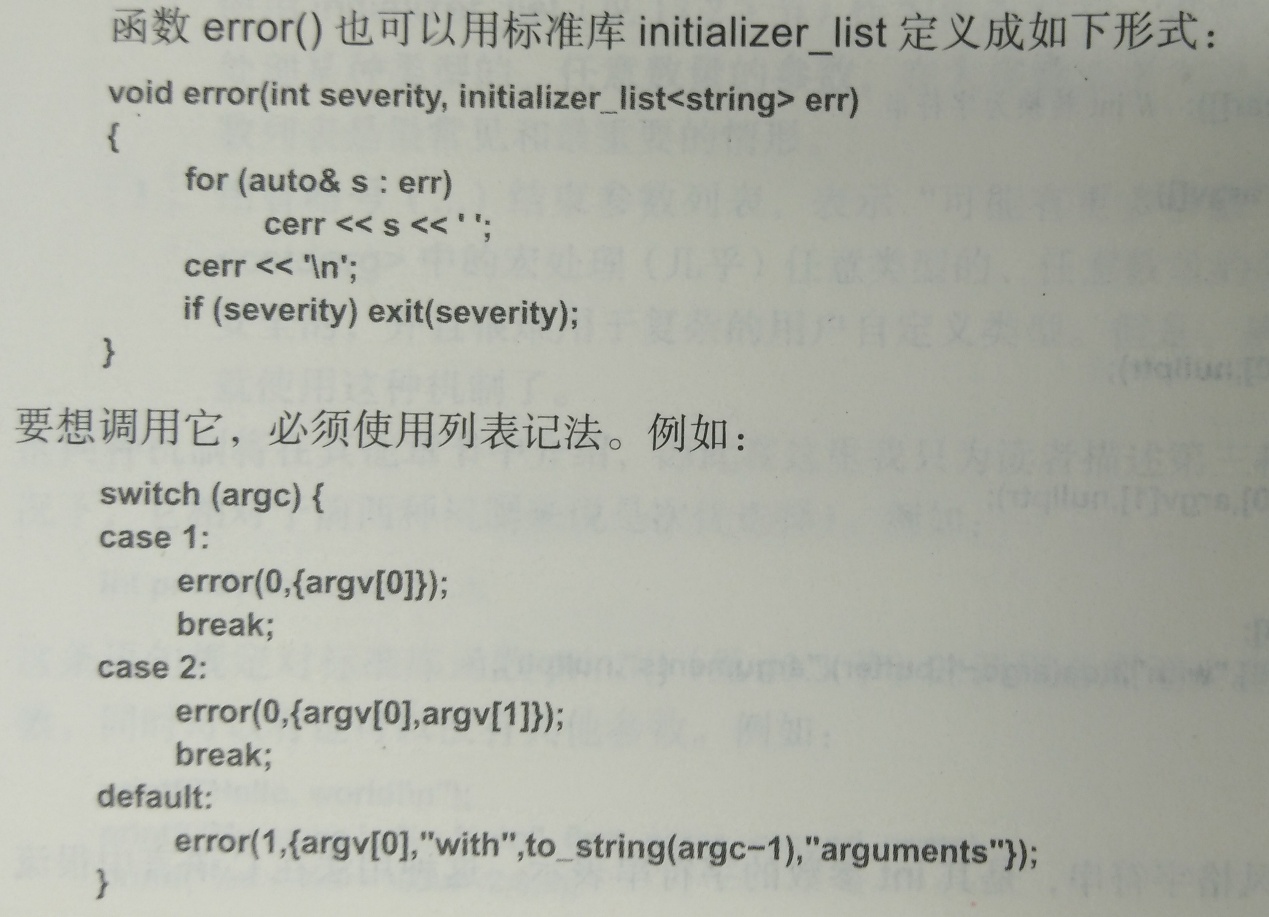
va\_end(ap);//arg结束

cerr<<’\n’;

if(severity) exit(severity);

}

首先，定义va\_list并调用va\_start()初始化它。宏命令va\_start接受va\_list的名字和最后一个正式参数的名字作为它的参数。宏命令va\_arg()用于顺序提取未命名的参数。每次调用它时，程序员必须提供一个类型；va\_arg()假定该类型的一个实参被传入了函数，但是通常它无法确保这一点。如果在函数中使用了va\_start()，则在该函数返回前必须先调用va\_end()。这么做的原因是va\_start()可能会修改栈的内容，从未造成函数无法正常返回；但是va\_end()可以撤销所有此类修改。



标注库负责提供int向string的转换函数to\_string()；

#### 12.3.3 重载与作用域

重载发身在一组重载函数集的成员内部，也就是说，重载函数应该位于同一个作用域内。不同的非命名空间作用域中的函数不会重载。

基类和派生类提供的作用域不同，因此默认情况下基类函数和派生类函数不会发生重载。例如：

struct Base

{

void f(int);

}

struct Derived : Base

{

void f(double);

}

void g(Derived& d)

{

d.f(1); //调用Derived:: f(double)

}

如果我们希望实现跨类作用域或者名字空间作用域的重载，应该使用using声明或者using指示（见14.2.2）。依赖于参数的查找也会导致跨名字空间的重载。（p285）

### 12.5 函数指针

与（数据）对象类似，由函数体生成的代码也置于某块内存区域中，因此他也有自己的地址。既然我们可以让指针指向对象，当然也就可以让指针指向函数。与此同时，出于某些考虑—有的与机器体系结构有关，有的与系统设计有关—我们不允许函数指针修改他所指的代码。程序员只能对函数做两种操作：调用它或者获取它的地址。通过获取函数地址得到的指针能被用来调用该函数。

解引用函数指针时可以用\*，也可以不用；同样，获取函数地址是可以用&也可以不用，例如：

void (\*f1)(string) = &error; //OK,等价于=error

void (\*f2)(string) = error; // OK,等价于=&error

void g()

{

f1(“Vasa”); //OK,等价于(\*f1)(“Vasa”)

(\*f1)(“Mary Rose”); //OK,等价于f1(“Mary Rose”)

}

函数只针对额参数类型声明与函数本身类似。进行指针赋值操作时，要求完整的函数类型必须精确匹配。

C++允许把一个函数指针转换成别的函数指针类型，但之后必须把得到的结果指针转换回它原来的类型，否则就会出现意想不到的情况：

using P1 = int (\*)(int\*);

using P2 = void (\*)(void);

void f(P1 pf)

{

P2 pf2 = reinterpret\_cast<P2>(pf);

pf2(); //可能发生严重错误

P1 pf1 = reinterpret\_cast<P1>(pf2); //把pf2“转换回来

int x = 7;

int y = pf1(&x); //OK

}

我们可以将指向noexcept函数的指针声明成noexcept的，例如：

void f(int) noexcept;

void g(int);

void (\*p1)(int) = f; //OK,但是丢失了有用信息

void (\*p2)(int) noexcept = f; //OK,保留了noexcept信息

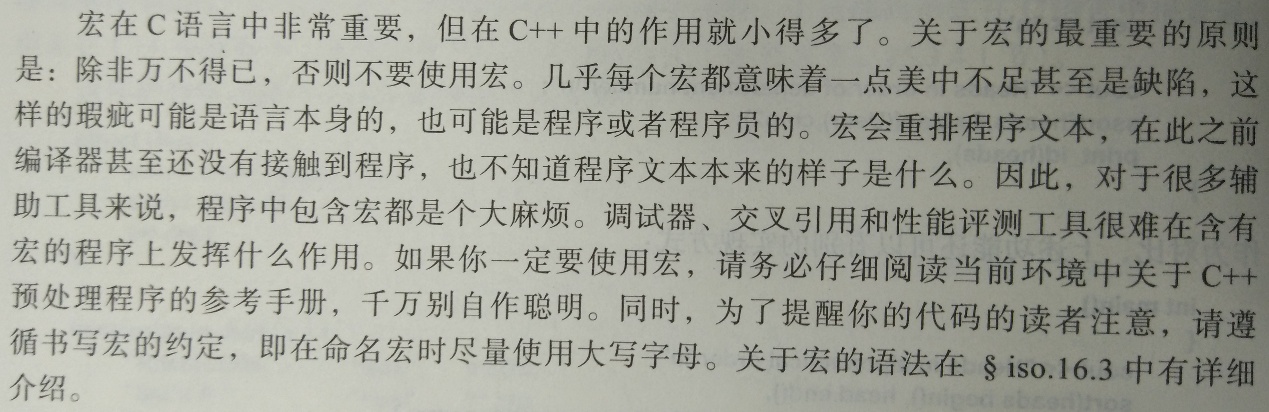
void (\*p3)(int) noexcept = g; //错误，我们并在不知道g会不会抛出异常

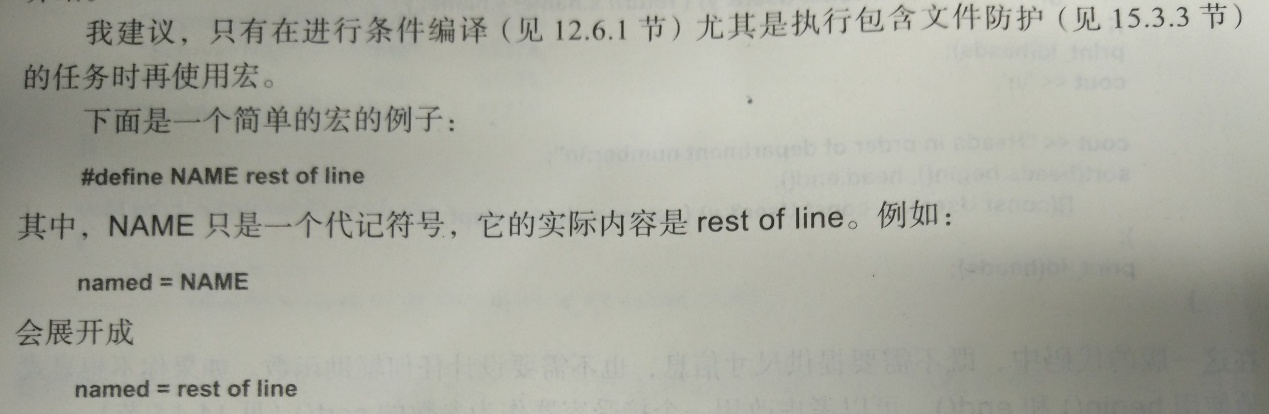
函数指针必须放映函数的链接信息。链接说明和noexcept都不能出现在类型别名中：

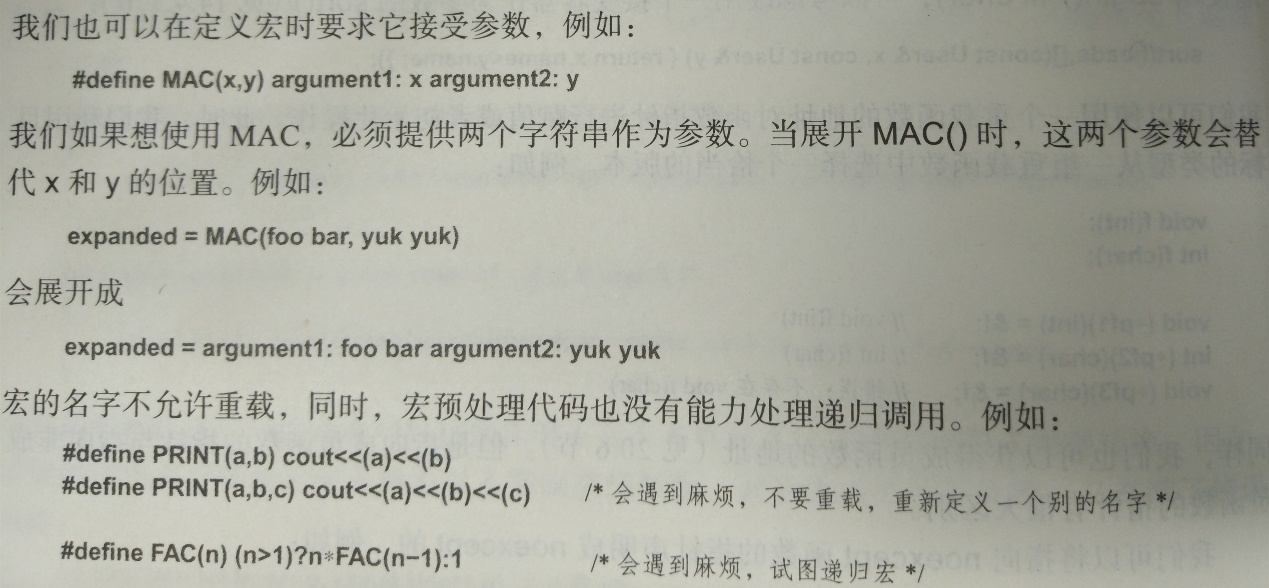
using Pc = extern “C” void (int); //错误，别名中出现了链接说明

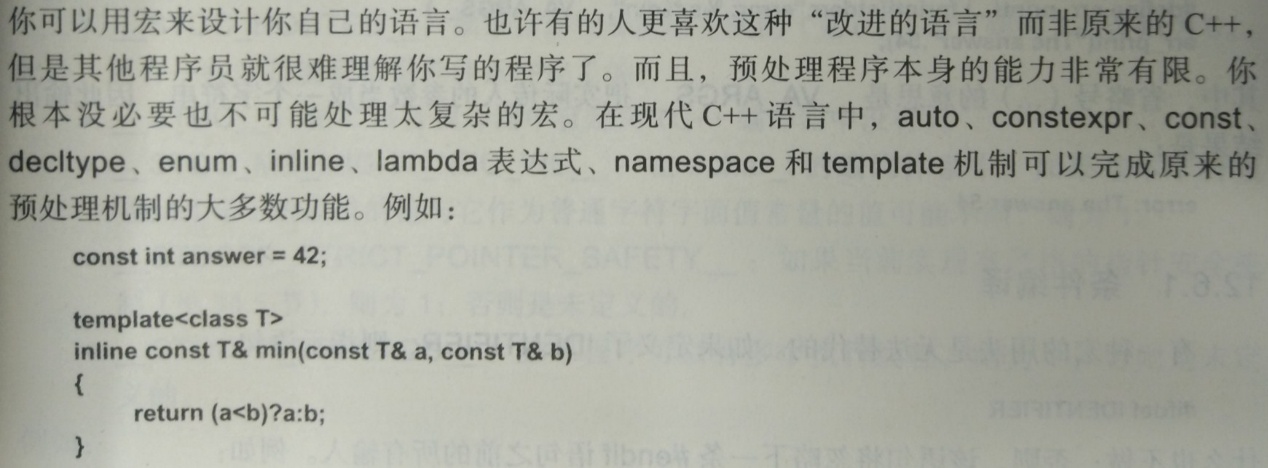
using Pn = void (int) noexcept; //错误，别名中出现了noexcept

### 12.6 宏









1. (p293)在编写宏时，经常需要命名某些东西，我们可以使用##宏运算符把两个字符串拼接成一个。例如：

#define NAME2(a,b) a##b

int NAME2(hack,cah)();

会展开成：

int hackcah();

在置换字符串中，如果参数的名字前面有一个单独的#，表示此处是包含宏参数的字符串。例如：

#define printfx(x) cout<<#x “ = ”<<x<<’\n’;

int a= 7;

string str = “asdf”;

void f()

{

prinfx(a); // cout<< “a” << “ = “ <<a<<’\n’;

prinfx(str); // cout<< “str” << “ = “ <<str<<’\n’;

}

请注意，我们在上面的代码中写成 #x “ = ” 而非 #x<<” = ”,这不是书写错误，而是“比较聪明的代码”。相邻的字符串字面值常量会被连接在一起。

1. 指示语句 #undef X

确保没有任何一个宏定义X——不论在该只是语句之前是否存在名为X的宏。这种机制使得我们可以不必受某种受意料之外的宏的影响。不过，有时候我们很难弄清楚X到底对一段代码应该有什么样的影响。

1. 宏的参数列表（“置换列表”）可以为空：

#define EMPTY() std<<cout<<”empty\n”

EMPTY(); //输出“empty\n”

EMPTY; //错误，缺少宏置换列表

空的参数列表不算是一种有错误风险或者恶意的代码。

1. (p294)宏甚至可以是可变参数的。例如：

#define err\_print(…) fprintf(stderr, “error: %s %d\n”, \_\_VA\_ARGS\_\_)

err\_print(“The answer”, 54);

其中，省略号（…）的意思是\_\_VA\_ARGS\_\_把实际传入的参数当成一个字符串，因此输出结果是：

error:The answer 54

1. 条件编译（p294）

有一种宏的用法是无法替代的。如果定义了IDENTIPIFIER，则只是语句

#ifdef IDENTIFIER

什么也不做；否则，该语句将忽略下一条#endif语句之前的所有输入。例如：

int f( int a

#ifdef arg\_two

, int b

#endif

);

除非我们#define了一个arg\_two的宏，否则将得到

int f( int a

);

一般情况下，#ifdef不会有太大的危害。只要按照规范使用，#ifdef及与之对应的#ifndef都不会带来什么问题。

必须谨慎选择用于控制#ifdef的宏名字，确保他们不会与现有的标识符冲突。例如：

struct Call\_info

{

Node\* arg\_one;

Node\* arg\_two;

//…

}

这段代码看起来没什么问题，但是如果有人写了下面的宏，就会产生混淆：

#define arg\_two x

不幸的是，在很多常见的头文件中包含了大量这种及危险又没什么必要的宏。

1. 预定义宏（p295）

编译器预定义了一些宏（§iso.16.8， §iso.8.4.1）:

* \_\_cplusplus ：在C++编译器中有定义（C语言编译器没有）。在C++11程序中它的值是201103L，在之前的C++标准中该值相应地小一些。
* \_\_DATE\_\_ ：“yyyy:mm:dd”格式的日期。
* \_\_TIME\_\_ ：“hh:mm:ss”格式的时间
* \_\_FILE\_\_ ：当前源文件的名字
* \_\_LINE\_\_ ：当前源文件的代码行数
* \_\_FUNC\_\_ ：是一个由具体实现定义的C风格的字符串，表示当前函数的名字。
* \_\_STDC\_HOSTED\_\_ ：如果当前实现是宿主式的（见6.1.1节）则为1；否则为0。

此外，还有一些宏时实现根据具体条件定义的

* \_\_STDC\_\_ ：在c语言编译器中有定义（C++编译器中没有）。
* \_\_STDC\_MB\_MIGHT\_NEQ\_WC\_\_ ：在wchar\_t的编码体系中，如果基本字符（见6.1节）的成员的值与它作为普通字符字面值常量的值可能不同，则为1。
* \_\_STDCPP\_STRICT\_POINTER\_SAFETY\_\_ ：如果当前实现有严格的指针安全机制，则为1；否则是未定义的。
* \_\_STDCPP\_THREADS\_\_ ：如果程序可以有多个执行线程，则为1；否则是未定义的。

1. 编译指令

具体的C++实现常常提供一些有别于标准甚至标准之外的功能。显然，标准无法规定这样的额外功能应该以何种方式提供，但是标准的句法应该是以预处理指示#pragma作为前缀的一个符号行，例如：

#pragma foo bar 666 foobar

如果可能，尽量避免使用#pragma

