CHAPTER 3 Moving to Modern C++

说起知名的特性，C++11/14有一大堆可以吹的东西，auto，智能指针，移动语意，lambda，并发——每个都是如此的重要，这章将覆盖这些内容。
精通这些特性是必要的，但是成为高效率的现代C++程序员也要求一系列小步骤。
从C++98移步C++11/14遇到的每个细节问题都会在本章得到答复。
应该在创建对象时用{}而不是()吗？为什么alias声明比typedef好？constexpr和const有什么不同？常量成员函数和线程安全有什么关系？这个列表越列越多。
这章将会逐个回答这些问题。

Item 7:Distinguish between () and {} when creating objects

条款七:区别使用()和{}创建对象

从不同的角度看，C++11初始化对象的语法选择既丰富得让人尴尬又混乱得让人糊涂。一般来说，初始化值要用()或者{}括起来或者放到"="的右边：

```
int x(0);               //使用小括号初始化

int y = 0;              //使用"="初始化

int z{0};               //使用花括号初始化
```

在很多情况下，你可以使用"="和花括号的组合：

```
int z = {0};            //使用"="和花括号
```

在这个条款的剩下部分，我通常会忽略"="和花括号组合初始化的语法，因为C++通常把它视作和只有花括号一样。
"混乱得令人糊涂"指出在初始化中使用"="可能会误导C++新手，使他们以为这里是赋值运算符。
对于像int这样的内置类型，研究两者区别是没有多大意义的，但是对于用户定义的类型而言，区别赋值运算符和初始化就非常重要了，因为这可能包含不同的函数调用：

```
Widget w1;              //调用默认构造函数

Widget w2 = w1;         //不是赋值运算符，调用拷贝构造函数

w1 = w2;                //是一个赋值运算符，调用operator=函数
```

甚至对于一些初始化语法，在一些情况下C++98没有办法去表达初始化。举个例子，要想直接表示一些存放一个特殊值的STL容器是不可能的（比如Item1,3,5）

C++11使用统一初始化(uniform initialization)来整合这些混乱且繁多的初始化语法，所谓统一初始化是指使用单一初始化语法在任何地方*[0]*表达任何东西。
它基于花括号，出于这个原因我更喜欢称之为括号初始化*[1]*。统一初始化是一个概念上的东西，而括号初始化是一个具体语法构型。
括号初始化让你可以表达以前表达不出的东西。使用花括号，指定一个容器的元素变得很容易：

```
std::vector<int> v{1,3,5};      //v包含1,3,5
```

括号初始化也能被用于为非静态数据成员指定默认初始值。C++11允许"="初始化也拥有这种能力：

```
class Widget{
    ...
private:
    int x{0};        //没问题，x初始值为0
    int y = 0;       //同上
    int z(0);        //错误!
}
```

另一方面，不可拷贝的对象可以使用花括号初始化或者小括号初始化，但是不能使用"="初始化：

```
std::vector<int> ai1{0};    //没问题，x初始值为0
std::atomic<int> ai2(0);    //没问题
std::atomic<int> ai3 = 0;   //错误!
```

因此我们很容易理解为什么括号初始化又叫统一初始化，在C++中这三种方式都被指派为初始化表达式，但是只有括号任何地方都能被使用。

括号表达式有一个异常的特性，它不允许内置类型隐式的变窄转换（narrowing conversion）。如果一个使用了括号初始化的表达式的值无法用于初始化某个类型的对象，代码就不会通过编译：

```
double x,y,z;

int sum1{x+y+z};            //错误！三个double的和不能用来初始化int类型的变量
```

使用小括号和"="的初始化不检查是否转换为变窄转换，因为由于历史遗留问题它们必须要兼容老旧代码

```
int sum2(x + y +z);         //可以（表达式的值被截为int）

int sum3 = x + y + z;       //同上
```

另一个值得注意的特性是括号表达式对于C++最令人头疼的解析问题*[2]*有天生的免疫性。
C++规定任何能被决议为一个声明的东西必须被决议为声明。这个规则的副作用是让很多程序员备受折磨：当他们想创建一个使用默认构造函数构造的对象，却不小心变成了函数声明。
问题的根源是如果你想使用一个实参调用一个构造函数，你可以这样做：

```
Widget w1(10);              //使用实参10调用Widget的一个构造函数
```

但是如果你尝试使用一个没有参数的构造函数构造对象，它就会变成函数声明：

```
Widget w2();                //最令人头疼的解析！声明一个函数w2，返回Widget
```

由于函数声明中形参列表不能使用花括号，所以使用花括号初始化表明你想调用默认构造函数构造对象就没有问题：

```
Widget w3{};                    //调用没有参数的构造函数构造对象
```

关于括号初始化还有很多要说的。它的语法能用过各种不同的上下文，它防止了隐式的变窄转换，而且对于C++最令人头疼的解析也天生免疫。
既然好到这个程度那为什么这个条款不叫“Prefer braced initialization syntax”呢?

括号初始化的缺点是有时它有一些令人惊讶的行为。
这些行为使得括号初始化和std::initializer_list和构造函数重载决议本来就不清不楚的暧昧关系进一步混乱。
把它们放到一起会让看起来应该左转的代码右转。
举个例子，Item2解释了当auto声明的变量使用花括号初始化，变量就会被推导为std::initializer_list，尽管使用相同内容的其他初始化方式会产生正常的结果。
所以，你越喜欢用atuo，你就越不能用括号初始化。

在构造函数调用中，只要不包含std::initializer_list参数，那么花括号初始化和小括号初始化都会产生一样的结果：

```
class Widget {
public:
    Widget(int i, bool b);       //未声明默认构造函数
    Widget(int i, double d);     // std::initializer_list参数
    …
};
Widget w1(10, true);           // 调用构造函数
Widget w2{10, true};           // 同上
Widget w3(10, 5.0);            // 调用第二个构造函数
Widget w4{10, 5.0};            // 同上
```

然而，如果有一个或者多个构造函数的参数是std::initializer_list，
使用括号初始化语法绝对比传递一个std::initializer_list实参要好。
而且只要某个调用能使用括号表达式编译器就会使用它。
如果上面的Widget的构造函数有一个**std::initializer_list**实参，就像这样：

```
class Widget {
public:
    Widget(int i, bool b);     // 同上
    Widget(int i, double d);   // 同上
    Widget(std::initializer_list<long double> il);       //新添加的
    …
};
```

w2和**w4**将会使用新添加的构造函数构造，即使另一个非std::initializer_list构造函数对于实参是更好的选择：

```
Widget w1(10, true);      // 使用小括号初始化
                          //调用第一个构造函数

Widget w2{10, true};      // 使用花括号初始化
                          // 调用第二个构造函数
                          // (10 和 true 转化为long double)

Widget w3(10, 5.0);       // 使用小括号初始化
                          // 调用第二个构造函数

Widget w4{10, 5.0};       // 使用花括号初始化
                          // 调用第二个构造函数
                          // (10 和 5.0 转化为long double)
```

甚至普通的构造函数和移动构造函数都会被std::initializer_list构造函数劫持：

```
class Widget {
public:
    Widget(int i, bool b);
    Widget(int i, double d);
    Widget(std::initializer_list<long double> il);
    operator float() const;          // convert to float (译者注：高亮)

};
Widget w5(w4);                  // 使用小括号，调用拷贝构造函数

Widget w6{w4};                  // 使用花括号，调用std::initializer_list构造函数

Widget w7(std::move(w4));       // 使用小括号，调用移动构造函数

Widget w8{std::move(w4)};       // 使用花括号，调用std::initializer_list构造函数
```

编译器热衷于把括号初始化与使std::initializer_list构造函数匹配了，热衷程度甚至超过了最佳匹配。比如：

```
class Widget {
public:
    Widget(int i, bool b);
    Widget(int i, double d);
    Widget(std::initializer_list<bool> il);    // element type is now bool

  …                                    // no implicit conversion funcs
};
Widget w{10, 5.0};      //错误！要求变窄转换
```

这里，编译器会直接忽略前面两个构造函数，然后尝试调用第三个构造函数，也即是std::initializer_list构造函数。
调用这个函数将会把`int(10)`和double(5.0)`转换为bool，由于括号初始化拒绝变窄转换，所以这个调用无效，代码无法通过编译。

只有当没办法把括号初始化中实参的类型转化为std::initializer_list时，编译器才会回到正常的函数决议流程中。
比如我们在构造函数中用`std::initializer_list<std::string`代替`std::initializer_list<bool>`，这时非std::initializer_list构造函数将再次成为函数决议的候选者，因为没有办法把int和bool转换为std::string:

```
class Widget {
public:
    Widget(int i, bool b);
    Widget(int i, double d);
    Widget(std::initializer_list<std::string> il);
     …
};
Widget w1(10, true);      // 使用小括号初始化，调用第一个构造函数
Widget w2{10, true};      // 使用花括号初始化，调用第一个构造函数
Widget w3(10, 5.0);       // 使用小括号初始化，调用第二个构造函数
Widget w4{10, 5.0};       // 使用花括号初始化，调用第二个构造函数
```

代码的行为和我们刚刚的论述如出一辙。这里还有一个有趣的边缘情况*[3]*。
假如你使用的花括号初始化是空集，并且你欲构建的对象有默认构造函数，也有std::initializer_list构造函数。
你的空的花括号意味着什么？如果它们意味着没有实参，就该使用默认构造函数，
但如果它意味着一个空的std::initializer_list，就该调用std::initializer_list构造函数。

最终会调用默认构造函数。空的花括号意味着没有实参，不是一个空的std::initializer_list：

```
class Widget {
public:
    Widget();
    Widget(std::initializer_list<int> il);

    ...
};
Widget w1;              // 调用默认构造函数
Widget w2{};            // 同上
Widget w3();            // 最令人头疼的解析！声明一个函数
```

如果你想调用std::initializer_list构造，你就得创建一个空花括号的实参来表明你想调用一个std::initializer_list构造函数，它的实参是一个空值。

```
Widget w4({});          // 调用std::initializer_list
Widget w5{{}};          // 同上
```

此时，括号初始化的晦涩规则，std::initializer_list和构造函数重载就会一下子涌进你的脑袋，你可能会想研究了半天这些东西在你的日常编程中到底占多大比例。
可能比你想象的要多。因为std::vector也会受到影响。
std::vector有一个非std::initializer_list构造函数允许你去指定容器的初始大小，以及使用一个值填满你的容器。
但它也有一个std::initializer_list构造函数允许你使用花括号里面的值初始化容器。如果你创建一个数值类型的vector，然后你传递两个实参。把这两个实参放到小括号和放到花括号中是不同：

```
std::vector<int> v1(10, 20);    //使用非std::initializer_list
                                //构造函数创建一个包含10个元素的std::vector
                                //所有的元素的值都是20
std::vector<int> v2{10, 20};    //使用std::initializer_list
                                //构造函数创建包含两个元素的std::vector
                                //元素的值为10和20
```

让我们退回之前的讨论。从这个讨论中我有两个重要结论。
第一，作为一个类库作者，你需要意识到如果你的一堆构造函数中重载过一个或者多个std::initializer_list，
用户代码如果使用了括号初始化，可能只会看到你重载的std::initializer_list这一个版本的构造函数。
因此，你最好把你的构造函数设计为不管用户是小括号还是使用花括号进行初始化都不会有什么影响。
换句话说，现在看到std::vector设计的缺点以后你设计的时候避免它。

这里的暗语是如果一个类没有std::initializer_list构造函数，然后你添加一个，
用户代码中如果使用括号初始化可能会发现过去被决议为非std::initializer_list构造函数现在被决议为新的函数。
当然，这种事情也可能发生在你添加一堆重载函数的时候，std::initializer_list重载不会和其他重载函数比较，
它直接盖过了其它重载函数，其它重载函数几乎不会被考虑。所以如果你要使用std::initializer_list构造函数，请三思而后行。

第二个，作为一个类库使用者，你必须认真的在花括号和小括号之间选择一个来创建对象。
大多数开发者都使用其中一种作为默认情况，只有当他们不能使用这种的时候才会考虑另一种。
如果使用默认使用花括号初始化，会得到大范围适用面的好处，它禁止变窄转换，免疫C++最令人头疼的解析。
他们知道在一些情况下（比如给一个容器大小和一个值创建std::vector)要使用小括号。
如果默认使用小括号初始化，它们能和C++98语法保持一致，它避开了auto自动推导std::initializer_list的问题，
也不会不经意间就调用了std::initializer_list构造函数。
他们承认有时候只能使用花括号（比如创建一个包含特殊值的容器）。
关于花括号和小括号的使用没有一个一致的观点，所以我的建议是用一个，并坚持使用。

如果你是一个模板的作者，花括号和小括号创建对象就更麻烦了。
通常不能知晓哪个会被使用。
举个例子，假如你想创建一个接受任意数量的参数，然后用它们创建一个对象。使用可变参数模板(variadic template)可以非常简单的解决：

```
template<typename T,
         typename... Ts>
void doSomeWork(Ts&&... params) {
  create local T object from params...  …
}
```

在现实中我们有两种方式使用这个伪代码（关于**std::forward**请参见Item25）：

```
T localObject(std::forward<Ts>(params)...);     // 使用小括号
T localObject{std::forward<Ts>(params)...};     // 使用花括号
```

考虑这样的调用代码：

```
std::vector<int> v;
…
 doSomeWork<std::vector<int>>(10, 20);
```

如果doSomeWork创建localObject时使用的是小括号，std::vector就会包含10个元素。
如果doSomeWork创建localObject时使用的是花括号，std::vector就会包含2个元素。
哪个是正确的？**doSomeWork**的作者不知道，只有调用者知道。

这正是标准库函数std::make_unique和std::make_shared（参见Item21）面对的问题。
它们的解决方案是使用小括号，并被记录在文档中作为接口的一部分。

记住

- 括号初始化是最广泛使用的初始化语法，它防止变窄转换，并且对于C++最令人头疼的解析有天生的免疫性
- 在构造函数重载决议中，括号初始化尽最大可能与std::initializer_list参数匹配，即便其他构造函数看起来是更好的选择
- 对于数值类型的std::vector来说使用花括号初始化和小括号初始化会造成巨大的不同
- 在模板类选择使用小括号初始化或使用花括号初始化创建对象是一个挑战。

译注

[0] 结合上下文得知这里的"任何地方"指的是初始化表达式存在的地方而不是广义上源代码的各处。

[1] 注意，这里的括号初始化指的是花括号初始化，在没有歧义的情况下下文的括号初始化指的都是用花括号进行初始化；当与小括号初始化同时存在并可能产生歧义时我会直接指出

[2] 所谓最令人头疼的解析即*most vexing parse*，更多信息请参见https://en.wikipedia.org/wiki/Most_vexing_parse

[3] 参见https://en.wikipedia.org/wiki/Edge_case