条款7:创建对象时使用()和{}的区别

在C++11中，你可以有多种语法选择用以对象的初始化，这样的语法显得混乱不堪并让人无所适从，括号，等号，大括号均可以用来进行初始化：

```
int x(0);       //使用()进行初始化
int y = 0;      //使用=进行初始化
int z{0};       //使用{}进行初始化
```

在很多情况下，可以同时使用等号和中括号

```
int z = {0};    //使用{}和=进行初始化
```

对于这一条，我通常的会忽略“等于-大括号”这种语法，因为C++通常认为它只有大括号。认为这种语法"混乱不堪"的辩护者指出使用等号用于初始化会误导C++的初学者认为赋值动作已经发生了，而实际上并没有。对于内建类型例如int，这种区别只是理论上的，但是对于用户自定义类型，初始化和赋值的区别很重要，因为两者调用了不同的函数：

```
Widget w1;      //调用了构造函数
Widget w2 = w1; //不是赋值语句，调用了拷贝构造函数
w1 = w2;//赋值语句，调用了=操作符
```

尽管拥有多种初始化的语法，C++98在很多情况下无法实现想要的初始化。例如，很难直接判断当一个STL容器包含一些值集合(例如,1,3,5)的时候是否应该创建。为了解决多种初始化语法冲突的问题，并且它们也无法涵盖所有的初始化场景，C++11引入了统一的初始化:使用单一的初始化语法，理论上可以在任何地方表达所有的初始化。它基于大括号，这也是我倾向于大括号初始化的原因。"统一的初始化"是一个概念，"大括号初始化"是一个语法实现。大括号初始化使你可以实现以前不能表达的含义。使用大括号，指定容器初始值变得简单:

```
std::vector<int> v{1,3,5};//v的初始值是1,3,5
```

大括号也可以用来为非静态数据成员指定初始的默认值。C++11中的一项新语法是支持大括号像"="一样初始化，但是括号不行：

```
class Widget{
   ...
   private:
        int x{0};       //正确，x默认值为0
        int y = 0;      //正确
        int z(0);       //错误
}
```

另一方法，非拷贝对象(std:atomics-参考第40条)可以使用大括号和括号初始化，但是"="不行:

```
std::atomic<int> ai1{0};//正确
std::atomic<int> ai2(0);//正确
std::atomic<int> ai3 = 0;       //错误
```

因此很容易理解括号初始化被称为标准。在三种为C++初始化设计的表达式中，只有括号可以被用在任何地方。括号初始化的一个新的特性是它禁止在基本类型中使用隐式的数值转换。如果一个括号中的表达式和初始化对象的类型不一致，代码将无法编译：

```
double x,y,z;
...
int sum1{x + y + z};    //错误，双精度类型数值相加无法表达为整形
```

使用括号和等号进行初始化时不会进行数值转换，因为这样会减少很多冗余的代码：

int sum2(x + y + z); //正确，表达式的值为转换成整形 int sum3 = x + y + z; //如上

括号初始化另外一个值得一提的特点是它摆脱了C++的令人头疼的歧义原则。C++原则的一个副作用是任何的可以用来申明的必须被解释成唯一的，经常困扰开发者的一个问题是他们想要

Widget w1(10); //调用Widget的构造函数并传递参数10

但是如果你想使用类似语法调用一个无参的构造函数，实际上你申明了一个函数，而不是对象：

Widget w2(); //非常的歧义，定义了一个返回Widget对象的w2函数

函数申明不可能使用大括号传递参数列表，所有使用大括号默认构造函数不会带来这个问题：

Widget w3{}; //调用Widget无参构造函数

使用大括号进行初始化还有很多值得一提的地方。它的语法可以用于广泛的上下文语境之中，它可以防止隐式值转换，而且不会出现C++的二义性。这真实一举三得的好事。那么为什么i
大括号的初始化的缺点是它带来的一些令人意外的行为。这些行为来自使用大括号初始化 `std::initializer_lists`和重载构造函数的异常纠结的关系中。它们之间的相互关系使得f
在构造函数调用中，只要不包含`std::initializer_list`参数列表，大括号和括号意义是一样：

class Widget{ public: Widget(int i,bool b); //构造函数没有声明为std::initializer_list的参数 Widget(int i,double d); }； Widget w1(10,true); //调用第一个构造函数 Widget w2(10,true); //同样调用第一个构造函数 Widget w3(10,50); //调用第二个构造函数 Widget w4(10,50); //同样调用第二个构造函数

但是，如果有一个或多个构造函数的参数类型是`std::initializer_list`，使用大括号初始化语法会优先调用使用了参数类型`std::initializer_list`的构造函数。更明确的一点

```
class Widget{
public:
   Widget(int i,bool b);//和上面一样
   Widget(int i,double d);       //和上面一样
   Widget(std::initializer_list<long double> il); //新加的构造函数
   ...
};
```

w2和w4将会使新的构造函数创建，即使std::initializer_list参数的构造函数看起来比非std::initializer_list构造函数更难匹配,如下：

```
Widget w1(10,true);             //使用括号构造函数
Widget w2{10,true};             //使用大括号构造函数,调用std::initializer_list参数，10和true被转换成long dobule型
Widget w3(10,5,0);              //使用括号构造函数
Widget w4{10,5.0};              //使用大括号，调用std::initializer_list参数，10和5.0被转换为long double
```

即使通常复制和移动的构造也会被认为使用std::initializer_list构造函数:

```
class Widget{
public:
    Widget(int i,bool b);                               //同上
    Widget(int i,double d);                             //同上
```

```
   Widget(std::initializer_list<long double> il);       //同上

   operator float() const;                              //转换成float型
   ...
};
Widget w5(w4);                                          //使用括号，调用拷贝构造函数
Widget w6(w4);                                          //使用大括号，调用std::initializer_list参数类型构造函数,w4被转换成float，然后再转换成lon
Widget w7(std::move(w4));                               //使用括号，调用move构造函数
Widget w8{std::move(w4)};                               //使用大括号，调用std::initializer_list构造函数，和w6原因一样
```

编译决定采用std::initializer_lists**参数的构造函数意愿强烈，及时这样的调用是不通过的。例如**

```
class Widget{
public:
   Widget(int i,bool b);                //如上
   Widget(int i,double d);                          //如上

   Widget(std::initializer_list<bool> il);          //元素类型是bool

   ...
};
Widget w{10,5.0};                                   //错误，要求类型收窄的转换
```

在这里，编译器会忽略前两个构造函数(第二个还是参数完全匹配的)而是试图调用std::initializer_list**参数的构造函数。调用这个构造函数需要将**int(10)**和**double(5.0)**转换成**bool**型。两个转换均会出现类型收窄(**bool**型不能代表**int**和**double**类型)，收窄的类型转换在大括号内初始化是被禁止的，所以这个调用非法的，代码编译不通过。只有当无法使用大括号初始化的参数转换成**std::initializer_list**的时候，编译器才会回来调用正常的构造函数，例如，如果我们将**std::initializer_list<bool>**构造函数替换成**std::initializer_list<std::string>**，那么非**std::initializer_list**参数的构造函数称为候选，因为没有办法将**int**和**bool**型转换成**std::strings:

```
class Widget{
public:
    Widget(int i,bool b);                           //同上
   Widget(int i,double d);                          //同上

   //std::initalizer_list元素类型是std::string
    Widget(std::initializer_list<std::string> il);//没有隐式转换
   ...
    };

    Widget w1(10,true);                             //使用括号初始化，调用第一个构造函数
   Widget w2{10,true};                              //使用大括号初始化，调用第一个构造函数
   Widget w3(10,5.0);                               //使用括号初始化，调用第二个构造函数
   Widget w4{10,5.0};                               //使用大括号初始化，调用第二个构造函数
```

我们现在已经接近完成探索大括号初始化和重载构造函数，但是有一个有趣的情形值得一提。假设你使用一个空的大括号构造对象，对象同时支持std::initializer_list**作为参数的构造函数。空的大括号参数指什么呢？如果表示空的参数，那么你将调用默认构造函数，如果表示空的**std::initializer_list**，那么调用无实际传入参数**std::initializer_list**构造函数。规则是调用默认构造函数。空的大括号意味着无参数，并不是空的**std::initializer_list**：**

```
class Widget{
public:
    Widget();                                       //默认构造函数

   Widget(std::initializer_list<init> il);          //std::initializer_list构造函数
   ...
};
Widget w1;                                          //调用默认构造函数
Widget w2{};                                        //调用默认构造函数
Widget w3{};                                        //令人恼火的解析，声明了一个函数!
```

如果你想使用空的initializer_list**参数来调用**std::initializer_list**参数的构造函数，你可以使用空的大括号作为参数--把空的大括号放在小括号之中来标定你传递的内容**:

```
Widget w4({});                                      //使用空列表作为参数调用std::initializer_list型的构造函数
Widget w5({});                                      //如上
```

在这一点上，看似神秘的大括号初始化，std::initializer_lists**参数初始化，重载构造函数萦绕在你的脑海中，你或许或好奇有多少的信息量会影响到我们平时日常的编程中。这比你想象的更多，因为其中一个被直接影响的类是**std::vector**。**std::vector**有一个非**std::initializer_list**的构造函数允许你指定容器的大小以及每个元素的初始值，它还拥有一个**std::initializer_list**参数的构造函数允许你指定容器的初始值。如果你创建一个数值类型的(例如**:std::vector)**的容器并且传递两个参数给构造函数，使用大括号和小括号传递参数将会导致非常明显的区别**:

```
std::vector<int> v1(10,20)                          //使用非std::initializer_list参数的构造函数，结果构造了10个元素的std::vector对象，每个对象的值都是
std::vector<int> v2{10,20}                          //使用std::initializer_list参数的构造函数，结果构造了2个元素的std::vector对象，两个元素分别是10和
```

让我们从讨论std::vector**，以及大括号，括号以及重载构造函数的选择细节中脱离出来。有零点需要指出。第一，作为一个类的作者，你需要明白如果你有一系列的重载构造函数，其中包括了一个或多个以**std::initializer_list**作为参数，客户端代码使用大括号初始化将只看到**std::initializer_list**参数重载构造函数。因此，你最好设计你的构造函数是的无论客户端代码使用大括号还是小括号初始化重载构造函数的调用不会受到影响。换句话说，在设计**std::vector**的接口中现在看起来是错误的地方从中学习用来设计自己的类避免同样的错误。一个内在的含义是如果你有一个没有以**std::initializer_list**作为参数的构造函数的类，你添加一个，客户端代码使用大括号初始化时会从以前被解析成调用非**std::initializer_list**的构造函数变成现在一个新的函数。当然，这种情况在你添加新的重载函数时经常发生：函数调用从之前调用老的重载函数变成现在新的函数。**std::initializer_list**参数的构造函数不同的地方在于它不与其他构造函数竞争，它使得其他构造函数变得不再被得到调用。因此添加这样的重载函数需要谨慎考虑。第二个值得学习的地方是作为一个客户端类，你必须谨慎选择大括号或括号创建对象。绝大部门开发者选择使用分隔符作为默认构造函数，除非必要时才会选择其他。使用大括号初始化的开发者被它们广泛的适用性所吸引，它们禁止类型值收缩转换，并且没有**c++**大多数的二义性。这些开发者知道在一些情况下**(**例如，在创建**std::vector**的时候指定大小和初始元素值**)**.要求使用小括号。另一方面，小括号可以被当成默认的参数分隔符。这一点和**C++98**的语言传统一致，避免了**auto**推断**std::initializer_list**的问题，也是吸引人的地方。对象创建的时候调用构造函数不会不经意的被**std::initializer_list**参数型的构造函数做拦截。他们承认有时候只有大括号能做初始化**(**例如，使用指定的值创建容器**)**。关于哪种方式更好没有共识，所以我的建议是选择其中一种方式，并且保持一致。如果你是一个模块作者，使用小括号还是大括号来构造对象带来的不安是令人沮丧的，因为，通常我们并不知道哪一种方式被用到。例如，假设你想使用任意类型任意数量的参数创建一个对象。可变参数类模块是的这变得很简单**:

```
template<typename T,                                //创建对象类型
        typename... Ts>                             //使用的参数类型
void doSomeWork(TS&&... params)
{
   create local T objcet from params...
   ...
}
```

有两种方式可以是的伪代码变成真实的代码(参见条目25关于std::forward**说明)**:

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

考虑如下的调用代码:

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

如果doSomeWork**使用小括号创建**localObject,**结果得到**10**个元素的**std::vector**。如果**doSomeWork**调用大括号，结果得到两个元素的**std::vectore**。哪一种是对的呢？**doSomeWork**的作者不知道，只有调用的才知道。这就是标准库函数**std::make_unique**和**std::make_shared(**参见条目**21)**面临的问题。这些函数在内部使用小括号并在接口文档中注明以解决这个问题。需要注意的地方 *大括号初始化是广泛使用的初始化语法，它禁止了值收窄的类型转换，并且不会出现大多数**C++**的二义性 *在重载构造函数的选择中，大括号初始化会尽量去**std::initializer_list**型参数的的构造函数，即使其他构造函数看起来更匹配 *一个选择大括号还是小括号具有明显区别的例子是构造具有两个参数的数值类型的**std::vector** *在模块中选择大括号还是小括号创建对象很具有挑战性