《现代C++编程指南》：理解统一初始化

花括号初始化是C++11中一种统一的数据初始化方法。因此，它也被称为统一初始化。可以说，这是C++11中开发者应该理解和使用的最重要的特性之一。它消除了之前在初始化基本类型、聚合类型和非聚合类型以及数组和标准容器之间的区别。

准备工作

要继续本教程，你需要熟悉直接初始化（使用一组显式的构造函数参数来初始化对象）和复制初始化（使用另一个对象来初始化对象）。以下是这两种初始化类型的简单示例：

```
std::string s1("test"); // 直接初始化
std::string s2 = "test"; // 复制初始化
```

牢记这些内容，让我们探索如何进行统一初始化。

如何做...

要统一初始化对象，无论其类型如何，都使用花括号初始化形式 `{}`，这可用于直接初始化和复制初始化。当与花括号初始化一起使用时，这些分别称为直接列表初始化和复制列表初始化：

```
T object {other}; // 直接列表初始化
T object = {other}; // 复制列表初始化
```

统一初始化的示例如下：

- 标准容器：

  ```
  std::vector<int> v { 1, 2, 3 };
  std::map<int, std::string> m { {1, "one"}, { 2, "two" }};
  ```

- 动态分配的数组：

  ```
  int* arr2 = new int[3]{ 1, 2, 3 };
  ```

- 数组：

```
int arr1[3] { 1, 2, 3 };
```

- 内置类型：

```
int i { 42 };
double d { 1.2 };
```

- 用户自定义类型：

```
class foo
{
    int a_;
    double b_;
public:
    foo() :a_(0), b_(0) {}
    foo(int a, double b = 0.0) :a_(a), b_(b) {}
};
foo f1{};
foo f2{ 42, 1.2 };
foo f3{ 42 };
```

- 用户自定义POD类型：

```
struct bar { int a_; double b_;};
bar b{ 42, 1.2 };
```

工作原理

在C++11之前，对象的初始化根据其类型需要不同的方法：

- 基本类型可以使用赋值进行初始化：

```
int a = 42;
double b = 1.2;
```

- 如果类对象有一个转换构造函数（在C++11之前，带有一个参数的构造函数称为转换构造函数），也可以使用单个值的赋值进行初始化：

```
class foo
{
   int a_;
public:
   foo(int a) :a_(a) {}
};
foo f1 = 42;
```

- 非聚合类在有参数时可以使用括号（函数形式）进行初始化，而在执行默认初始化（调用默认构造函数）时则不能使用任何括号。在下面的示例中，`foo`
是在“如何做...”部分定义的结构体：

```
foo f1; // 默认初始化
foo f2(42, 1.2);
foo f3(42);
foo f4(); // 函数声明
```

- 聚合类型和POD类型可以使用花括号初始化。

```
bar b = {42, 1.2};
int a[] = {1, 2, 3, 4, 5};
```

POD类型是一种既是平凡的（具有编译器提供或显式默认的特殊成员，并且占用连续的内存区域），又具有标准布局的类型（一个不包含与C语言不兼容的语言特性，如虚函数，并且所有成员具有相同访问控制的类）。POD类型的概念在C++20中被弃用，取而代之的是平凡类型和标准布局类型。

除了不同的数据初始化方法外，还存在一些限制。例如，初始化标准容器（除了复制构造）的唯一方法是先声明一个对象，然后向其中插入元素；`std::vector`是一个例外，因为它可以从可以使用聚合初始化提前初始化的数组中赋值。然而，动态分配的聚合类型不能直接初始化。

“如何做...”部分中的所有示例都使用直接初始化，但也可以使用花括号初始化进行复制初始化。这两种形式，直接初始化和复制初始化，在大多数情况下可能是等效的，但复制初始化的限制更多，因为它在其隐式转换序列中不考虑显式构造函数，必须直接从初始化器产生对象，而直接初始化期望从初始化器到构造函数参数的隐式转换。

动态分配的数组只能使用直接初始化进行初始化。在前面的示例中，`foo`是唯一一个既有默认构造函数又有带参数构造函数的类。要使用默认构造函数进行默认初始化，我们需要使用空的花括号，即`{}`。要使用带参数的构造函数，我们需要在花括号`{}`中为所有参数提供值。与非聚合类型不同，非聚合类型的默认初始化意味着调用默认构造函数，而对于聚合类型，默认初始化意味着用零进行初始化。

前面也展示了标准容器（如 `vector` 和 `map` ）的初始化是可能的，因为所有标准容器在C++11中都有一个额外的构造函数，该构造函数接受一个类型为 `std::initializer_list<T>` 的参数。这基本上是一个轻量级的代理，用于类型为 `T const` 的元素数组。这些构造函数然后从初始化器列表中的值初始化内部数据。

使用 `std::initializer_list` 进行初始化的工作方式如下：

- 编译器解析初始化列表中元素的类型（所有元素必须具有相同的类型）。
- 编译器创建一个包含初始化器列表中元素的数组。
- 编译器创建一个 `std::initializer_list<T>` 对象来包装之前创建的数组。
- 将 `std::initializer_list<T>` 对象作为参数传递给构造函数。

当使用花括号初始化时，初始化器列表总是优先于其他构造函数。如果类存在这样的构造函数，则在执行花括号初始化时会调用它：

```
class foo
{
   int a_;
   int b_;
public:
   foo() :a_(0), b_(0) {}
   foo(int a, int b = 0) :a_(a), b_(b) {}
   foo(std::initializer_list<int> l) {}
};
foo f{ 1, 2 }; // 调用带有 std::initializer_list<int> 的构造函数
```

这条优先规则适用于任何函数，而不仅仅是构造函数。在以下示例中，存在同一个函数的两个重载版本。使用初始化器列表调用该函数将解析为调用带有 `std::initializer_list` 的重载版本：

```
void func(int const a, int const b, int const c)
{
   std::cout<< a << b<< c << '₩n';
}
void func(std::initializer_list<int> const list)
{
   for (auto const & e : list)
      std::cout<< e << '₩n';
}
func({ 1,2,3 }); // 调用第二个重载版本
```

然而，这可能会导致错误。例如，对于 `std::vector` 类型，其构造函数中有一个带有一个参数的构造函数，表示要分配的初始元素数量，另一个构造函数带有 `std::initializer_list` 作为参数。如果意图是创建一个具有预分配大小的向量，使用花括号初始化将不起作用，因为带有 `std::initializer_list` 的构造函数将是最佳的重载版本：

```
std::vector<int> v {5};
```

前面的代码没有创建一个包含五个元素的向量，而是创建了一个包含一个值为5的元素的向量。要实际创建一个包含五个元素的向量，必须使用括号形式的初始化：

```
std::vector<int> v (5);
```

另一个需要注意的事项是，花括号初始化不允许窄化转换。根据C++标准（参见标准的第8.5.4段），窄化转换是一种隐式转换：

- 从浮点类型到整数类型。
- 从 `long double` 到 `double` 或 `float`，或从 `double` 到 `float`，除非源是常量表达式并且转换后的实际值在可以表示的值范围内（即使不能精确表示）。
- 从整数类型或无作用域枚举类型到浮点类型，除非源是常量表达式并且转换后的实际值适合目标类型并且在转换回原始类型时产生原始值。
- 从整数类型或无作用域枚举类型到不能表示原始类型所有值的整数类型，除非源是常量表达式并且转换后的实际值适合目标类型并且在转换回原始类型时产生原始值。

以下声明会触发编译器错误，因为它们需要窄化转换：

```
int i{ 1.2 }; // 错误
double d = 47 / 13;
float f1{ d }; // 错误
```

要修复此错误，必须进行显式转换：

```
int i{ static_cast<int>(1.2) };
double d = 47 / 13;
float f1{ static_cast<float>(d) };
```

花括号初始化列表不是一个表达式，也没有类型。因此，`decltype`
不能用于花括号初始化列表，模板类型推导也不能推导出与花括号初始化列表匹配的类型。

让我们考虑另一个示例：

float f2{47/13}; // 正确，f2=3

前面的声明是正确的，因为存在从 `int` 到 `float` 的隐式转换。表达式 `47/13`
首先被评估为整数值 `3`，然后将其赋值给类型为 `float` 的变量 `f2`。

更多内容

以下示例展示了几个直接列表初始化和复制列表初始化的示例。在C++11中，所有这些表达式的推导类型都是 `std::initializer_list<int>`：

auto a = {42}; // std::initializer_list<int>
auto b {42}; // std::initializer_list<int>
auto c = {4, 2}; // std::initializer_list<int>
auto d {4, 2}; // std::initializer_list<int>

C++17更改了列表初始化的规则，区分了直接列表初始化和复制列表初始化。新的类型推导规则如下：

- 对于复制列表初始化，如果列表中的所有元素类型相同，`auto` 推导将推导为 `std::initializer_list<T>`，否则推导无效。
- 对于直接列表初始化，如果列表只有一个元素，`auto` 推导将推导为 `T`，如果有多个元素，则推导无效。

根据这些新规则，前面的示例将发生变化（注释中提到了推导类型）：

auto a = {42}; // std::initializer_list<int>
auto b {42}; // int
auto c = {4, 2}; // std::initializer_list<int>
auto d {4, 2}; // 错误，元素太多

在这种情况下，`a` 和 `c` 被推导为 `std::initializer_list<int>`，`b` 被推导为 `int`，而 `d`

使用直接初始化并且在花括号初始化列表中有多个值，触发了编译器错误。