Item42: 考虑使用emplacement代替insertion

如果你拥有一个容器，例如`std::string`，那么当你通过插入函数（例如`insert, push_front, push_back`，或者对于`std::forward_list`， `insert_after`）添加新元素时，你传入的元素类型应该是`std::string`。毕竟，这就是容器里的内容。

逻辑上看来如此，但是并非总是如此。考虑如下代码：

```
std::vector<std::string> vs; // container of std::string
vs.push_back("xyzzy"); // add string literal
```

这里，容量里内容是`std::string`，但是你试图通过`push_back`加入字符串字面量，即引号内的字符序列。字符转字面量并不是`std::string`，这意味着你传递给`push_back`的参数并不是容器里的内容类型。

`std::vector`的`push_back`被按左值和右值分别重载：

```
template<class T, class Allocator = allocator<T>>
class vector {
public:
  ...
  void push_back(const &T x); // insert lvalue
  void push_back(T&& x); // insert rvalue
  ...
};
```

在`vs.push_back("xyzzy")`这个调用中，编译器看到参数类型（const char[6]）和`push_back`采用的参数类型（`std::string`的引用）之间不匹配。它们通过从字符串字面量创建一个`std::string`类型的临时变量来消除不匹配，然后传递临时变量给`push_back`。换句话说，编译器处理的这个调用应该像这样：

```
vs.push_back(std::string("xyzzy")); // create temp std::string and pass it to
push_back
```

代码编译并运行，皆大欢喜。除了对于性能执着的人意识到了这份代码不如预期的执行效率高。

为了创建`std::string`类型的临时变量，调用了`std::string`的构造器，但是这份代码并不仅调用了一次构造器，调用了两次，而且还调用了析构器。这发生在`push_back`运行时：

1. 一个`std::string`的临时对象从字面量"xyzzy"被创建。这个对象没有名字，我们可以称为*temp*，*temp*通过`std::string`构造器生成，因为是临时变量，所以*temp*是右值。
2. *temp*被传递给`push_back`的右值x重载函数。在`std::vector`的内存中一个x的副本被创建。这次构造器是第二次调用，在`std::vector`内部重新创建一个对象。（将x副本复制到`std::vector`内部的构造器是移动构造器，因为x传入的是右值，有关将右值引用强制转换为右值的信息，请参见Item25）。
3. 在`push_back`返回之后，*temp*被销毁，调用了一次`std::string`的析构器。

性能执着者（译者注：直译性能怪人）不禁注意到是否存在一种方法可以获取字符串字面量并将其直接传入到步骤2中的`std::string`内部构造，可以避免临时对象*temp*的创建与销毁。这样的效率最好，性能执着者也不会有什么意见了。

因为你是一个C++开发者，所以你会有高于平均水平的要求。如果你不是C++开发者，你可能也会同意这个观点（如果你根本不考虑性能，为什么你没在用python？）。所以让我来告诉你如何使得`push_back`达到最高的效率。就是不使用`push_back`，你需要的是`emplace_back`。

`emplace_back`就是像我们想要的那样做的：直接把传递的参数（无论是不是`std::string`）直接传递到`std::vector`内部的构造器。没有临时变量会生成：

```
vs.emplace_back("xyzzy"); // construct std::string inside vs directly from
"xyzzy"
```

`emplace_back`使用完美转发，因此只要你没有遇到完美转发的限制（参见Item30），就可以传递任何参数以及组合到`emplace_back`。比如，如果你在vs传递一个字符和一个数量给`std::string`构造器创建`std::string`，代码如下：

```
vs.emplace_back(50, 'x'); // insert std::string consisting of 50 'x' characters
```

`emplace_back`可以用于每个支持`push_back`的容器。类似的，每个支持`push_front`的标准容器支持`emplace_front`。每个支持`insert`（除了`std::forward_list`和`std::array`）的标准容器支持`emplace`。关联容器提供`emplace_hint`来补充带有“hint”迭代器的插入函数，`std::forward_list`有`emplace_after`来匹配`insert_after`。

使得emplacement函数功能优于insertion函数的原因是它们灵活的接口。insertion函数接受对象来插入，而emplacement函数接受构造器接受的参数插入。这种差异允许emplacement函数避免临时对象的创建和销毁。

因为可以传递容器内类型给emplacement函数（该参数使函数执行复制或者移动构造器），所以即使insertion函数不会构造临时对象，也可以使用emplacement函数。在这种情况下，insertion和emplacement函数做的是同一件事，比如：

```
std::string queenOfDisco("Donna Summer");
```

下面的调用都是可行的，效率也一样：

```
vs.push_back(queenOfDisco); // copy-construct queenOfDisco
vs.emplace_back(queenOfDisco); // ditto
```

因此，emplacement函数可以完成insertion函数的所有功能。并且有时效率更高，至上在理论上，不会更低效。那为什么不在所有场合使用它们?

因为，就像说的那样，理论上，在理论和实际上没有什么区别，但是实际，区别还是有的。在当前标准库的实现下，有些场景，就像预期的那样，emplacement执行性能优于insertion，但是，有些场景反而insertion更快。这种场景不容易描述，因为依赖于传递的参数类型、容器类型、emplacement或insertion的容器位置、容器类型构造器的异常安全性和对于禁止重复值的容器（即`std::set,std::map,std::unorder_set,set::unorder_map`）要添加的值是否已经在容器中。因此，大致的调用建议是：通过benchmakr测试来确定emplacment和insertion哪种更快。

当然这个结论不是很令人满意，所以还有一种启发式的方法来帮助你确定是否应该使用emplacement。如果下列条件都能满足，emplacement会优于insertion：

- **值是通过构造器添加到容器，而不是直接赋值**。例子就像本Item刚开始的那样（添加"xyzzy"到`std::string`的`std::vector`中）。新值必须通过`std::string`的构造器添加到`std::vector`。如果我们回看这个例子，新值放到已经存在对象的位置，那情况就完全不一样了。考虑下：

```
std::vector<std::string> vs; // as before
... // add elements to vs
vs.emplace(vs.begin(), "xyzzy"); // add "xyzzy" to beginning of vs
```

对于这份代码，没有实现会在已经存在对象的位置`vs[0]`构造添加的`std::string`。而是，通过移动赋值的方式添加到需要的位置。但是移动赋值需要一个源对象，所以这意味着一个临时对象要被创建，而emplacement优于insertion的原因就是没有临时对象的创建和销毁，所以当通过赋值操作添加元素时，emplacement的优势消失殆尽。

而且，向容器添加元素是通过构造还是赋值通常取决于实现者。但是，启发式仍然是有帮助的。基于节点的容器实际上总是使用构造器添加新元素，大多数标准库容器都是基于节点的。例外的容器只有`std::vector, std::deque, std::string`（`std::array`也不是基于节点的，但是它不支持emplacement和insertion）。在不是基于节点的容器中，你可以依靠`emplace_back`来使用构造向容器添加元素，对于`std::deque`，`emplace_front`也是一样的。

- **传递的参数类型与容器的初始化类型不同**。再次强调，emplacement优于insertion通常基于以下事实：当传递的参数不是容器保存的类型时，接口不需要创建和销毁临时对象。当将类型为T的对象添加到container时，没有理由期望emplacement比insertion运行的更快，因为不需要创建临时对象来满足insertion接口。
- **容器不拒绝重复项作为新值**。这意味着容器要么允许添加重复值，要么你添加的元素都是不重复的。这样要求的原因是为了判断一个元素是否已经存在于容器中，emplacement实现通常会创建一个具有新值的节点，以便可以将该节点的值与现有容器中节点的值进行比较。如果要添加的值不在容器中，则链接该节点。然后，如果值已经存在，emplacement创建的节点就会被销毁，意味着构造和析构时浪费的开销。这样的创建就不会在insertion函数中出现。

本Item开始的例子中下面的调用满足上面的条件。所以调用比`push_back`运行更快。

```
vs.emplace_back("xyzzy"); // construct new value at end of container; don't pass
the type in container; don't use container rejecting duplicates
vs.emplace_back(50, 'x'); // ditto
```

在决定是否使用emplacement函数时，需要注意另外两个问题。**首先**是资源管理。假定你有一个`std::shared_ptr<Widget>s`的容器，

```
std::list<std::shared_ptr<Widget>> ptrs;
```

然后你想添加一个通过自定义deleted释放的`std::shared_ptr`（参见Item 19）。Item 21说明你应该使用`std::make_shared`来创建`std::shared_ptr`，但是它也承认有时你无法做到这一点。比如当你要指定一个自定义deleter时。这时，你必须直接创建一个原始指针，然后通过`std::shared_ptr`来管理。

如果自定义deleter是这个函数，

```
void killWidget(Widget* pWidget);
```

使用insertion函数的代码如下:

```
ptrs.push_back(std::shared_ptr<Widget>(new Widget, killWidget));
```

也可以像这样

```
ptrs.push_back({new Widget, killWidget});
```

不管哪种写法，在调用`push_back`中会生成一个临时`std::shared_ptr`对象。`push_back`的参数是`std::shared_ptr`的引用，因此必须有一个`std::shared_ptr`。

`std::shared_ptr`的临时对象创建应该可以避免，但是在这个场景下，临时对象值得被创建。考虑如下可能的时间序列：

1. 在上述的调用中，一个`std::shared_ptr<Widget>`的临时对象被创建来持有`new Widget`对象。称这个对象为*temp*。
2. `push_back`接受*temp*的引用。在节点的分配一个副本来复制*temp*的过程中，OOM异常被抛出
3. 随着异常从`push_back`的传播，*temp*被销毁。作为唯一管理Widget的弱指针`std::shared_ptr`对象，会自动销毁`Widget`，在这里就是调用`killWidget`。

这样的话，即使发生了异常，没有资源泄露：在调用`push_back`中通过`new Widget`创建的`Widget`在`std::shared_ptr`管理下自动销毁。生命周期良好。

考虑使用`emplace_back`代替`push_back`

```
ptrs.emplace_back(new Widget, killWidget);
```

1. 通过`new Widget`的原始指针完美转发给`emplace_back`的内部构造器。如果分配失败，还是抛出OOM异常
2. 当异常从`emplace_back`传播，原始指针是仅有的访问途径，但是因为异常丢失了，这就发生了资源泄露

在这个场景中，生命周期不良好，这个失误不能赖`std::shared_ptr`。`std::unique_ptr`使用自定义deleter也会有同样的问题。根本上讲，像`std::shared_ptr`和`std::unique_ptr`这样的资源管理类的有效性取决于资源被**立即**传递给资源管理对象的构造函数。实际上，这就是`std::make_shared`和`std::make_unique`这样的函数如此重要的原因。

在对存储资源管理类的容器调用insertion函数时（比如`std::list<std::shared_ptr<Widget>>`），函数的参数类型通常确保在资源的获取和管理资源对象的创建之间没有其他操作。在emplacement函数中，完美转发推迟了资源管理对象的创建，直到可以在容器的内存中构造它们为止，这给异常导致资源泄露提供了可能。所有的标准库容器都容易受到这个问题的影响。在使用资源管理对象的容器时，比如注意确保使用emplacement函数不会为提高效率带来降低异常安全性的后果。

坦白说，无论如何，你不应该将`new Widget`传递给`emplace_back`或者`push_back`或者大多数这种函数，因为，就像Item 21中解释的那样，这可能导致我们刚刚讨论的异常安全性问题。使用独立语句将从`new Widget`获取指针然后传递给资源管理类，然后传递这个对象的右值引用给你想传递`new Widget`的函数（Item 21 有这个观点的详细讨论）。代码应该如下：

```
std::shared_ptr<Widget> spw(new Widget, killWidget); // create Widget and have
spw manage it
ptrs.push_back(std::move(spw)); // add spw as rvalue
```

emplace_back的版本如下：

```
std::shared_ptr<Widget> spw(new Widget, killWidget); // create Widget and have
spw manage it
ptrs.emplace_back(std::move(spw));
```

无论哪种方式，都会产生spw的创建和销毁成本。给出选择emplacement函数优于insertion函数的动机是避免临时对象的开销，但是对于swp的概念来讲，当根据正确的方式确保获取资源和连接到资源管理对象上之间无其他操作，添加资源管理类型对象到容器中，emplacement函数不太可能胜过insertion函数。

emplacement函数的**第二个**值得注意的方面是它们与显式构造函数的交互。对于C++11正则表达式的支持，假设你创建了一个正则表达式的容器：

```
std::vector<std::regex> regexes;
```

由于你同事的打扰，你写出了如下看似毫无意义的代码：

```
regexes.emplace_back(nullptr); // add nullptr to container of regexes?
```

你没有注意到错误，编译器也没有提示你，所以你浪费了大量时间来调试。突然，你发现你插入了空指针到正则表达式的容器中。但是这怎么可能？指针不是正则表达式，如果你试图下面这样写

```
std::regex r = nullptr; // error! won't compile
```

编译器就会报错。有趣的是，如果你调用`push_back`而不是`emplace_back`，编译器就会报错

```
regexes.push_back(nullptr); // error! won't compile
```

当前你遇到的奇怪行为由于可能用字符串构造`std::regex`的对象，这就意味着下面代码合法：

```
std::regex upperCaseWorld("[A-Z]+");
```

通过字符串创建`std::regex`要求相对较长的运行时开销，所以为了最小程度减少无意中产生此类开销的可能性，采用`const char*`指针的`std::regex`构造函数是显式的。这就是为什么下面代码无法编译的原因：

```
std::regex r = nullptr; // error! won't compile
regexes.push_back(nullptr); // error
```

在上面的代码中，我们要求从指针到`std::regex`的隐式转换，但是显式构造的要求拒绝了此类转换。

但是在`emplace_back`的调用中，我们没有声明传递一个`std::regex`对象。代替的是，我们传递了一个`std::regex`构造器参数。那不是隐式转换，而是显式的：

```
std::regex r(nullptr); // compiles
```

如果简洁的注释“compiles”表明缺乏直观理解，好的，因为这个代码可以编译，但是行为不确定。使用`const char*`指针的`std::regex`构造器要求字符串是一个有效的正则表达式，nullptr不是有效的。如果你写出并编译了这样的代码，最好的希望就是运行时crash掉。如果你不幸运，就会花费大量的时间调试。

先把`push_back, emplace_back`放在一边，注意到相似的初始化语句导致了多么不一样的结果：

```
std::regex r1 = nullptr; // error ! won't compile
std::regex r2(nullptr); // compiles
```

在标准的官方术语中，用于初始化r1的语法是所谓的复制初始化。相反，用于初始化r2的语法是（也被称为braces）被称为直接初始化。复制初始化不是显式调用构造器的，直接初始化是。这就是r2可以编译的原因。

然后回到`push_back`和 `emplace_back`，更一般来说，insertion函数对比emplacment函数。emplacement函数使用直接初始化，这意味着使用显式构造器。insertion函数使用复制初始化。因此：

```
regexes.emplace_back(nullptr); // compiles. Direct init permits use of explicit
std::regex ctor taking a pointer
regexes.push_back(nullptr); // error! copy init forbids use of that ctor
```

要汲取的是，当你使用emplacement函数时，请特别小心确保传递了正确的参数，因为即使是显式构造函数，编译器可以尝试解释你的代码称为有效的（译者注：这里意思是即使你写的代码逻辑上不对，显式构造器时编译器可能能解释通过即编译成功）

需要记住的事

- 原则上，emplacement函数有时会比insertion函数高效，并且不会更差
- 实际上，当执行如下操作时，emplacement函数更快
    1. 值被构造到容器中，而不是直接赋值
    2. 传入的类型与容器类型不一致
    3. 容器不拒绝已经存在的重复值
- emplacement函数可能执行insertion函数拒绝的显示构造