Item 26: Avoid overloading on universal references

Item 26: 避免在通用引用上重载

假定你需要写一个函数，它使用name这样一个参数，打印当前日期和具体时间到日志中，然后将name加入到一个全局数据结构中。你可能写出来这样的代码：

```
std::multiset<std::string> names; // global data structure
void logAndAdd(const std::string& name)
{
  auto now = std::chrono::system_lock::now(); // get current time
  log(now, "logAndAdd"); // make log entry
  names.emplace(name); // add name to global data structure; see Item 42 for info
on emplace
}
```

这份代码没有问题，但是同样的也没有效率。考虑这三个调用：

```
std::string petName("Darla");
logAndAdd(petName); // pass lvalue std::string
logAndAdd(std::string("Persephone")); // pass rvalue std::string
logAndAdd("Patty Dog"); // pass string literal
```

在第一个调用中，`logAndAdd`使用变量作为参数。在`logAndAdd`中`name`最终也是通过`emplace`传递给`names`。因为`name`是左值，会拷贝到`names`中。没有方法避免拷贝，因为是左值传递的。

在第三个调用中，参数`name`绑定一个右值，但是这次是通过"Patty Dog"隐式创建的临时`std::string`变量。在第二个调用总，`name`被拷贝到`names`，但是这里，传递的是一个字符串字面量。直接将字符串字面量传递给`emplace`，不会创建`std::string`的临时变量，而是直接在`std::multiset`中通过字面量构建`std::string`。在第三个调用中，我们会消耗`std::string`的拷贝开销，但是连移动开销都不想有，更别说拷贝的。

我们可以通过使用通用引用（参见Item 24）重写第二个和第三个调用来使效率提升，按照Item 25的说法，`std::forward`转发引用到`emplace`。代码如下：

```
template<typename T>
void logAndAdd(T&& name)
{
  auto now = std::chrono::system_lock::now();
  log(now, "logAndAdd");
  names.emplace(std::forward<T>(name));
}
std::string petName("Darla"); // as before
logAndAdd(petName); // as before , copy
logAndAdd(std::string("Persephone")); // move rvalue instead of copying it
logAndAdd("Patty Dog"); // create std::string in multiset instead of copying a
temporary std::string
```

非常好，效率优化了！

在故事的最后，我们可以骄傲的交付这个代码，但是我没有告诉你client不总是有访问`logAndAdd`要求的`names`的权限。有些clients只有`names`的下标。为了支持这种client，`logAndAdd`需要重载为：

```
std::string nameFromIdx(int idx); // return name corresponding to idx
void logAndAdd(int idx)
{
  auto now = std::chrono::system_lock::now();
  log(now, "logAndAdd");
  names.emplace(nameFromIdx(idx));
}
```

之后的两个调用按照预期工作：

```
std::string petName("Darla");
logAndAdd(petName);
logAndAdd(std::string("Persephone"));
logAndAdd("Patty Dog"); // these calls all invoke the T&& overload

logAndAdd(22); // calls int overload
```

事实上，这只能基本按照预期工作，假定一个client将`short`类型当做下标传递给`logAndAdd`：

```
short nameIdx;
...
logAndAdd(nameIdx); // error!
```

之后一行的error说明并不清楚，下面让我来说明发生了什么。

有两个重载的`logAndAdd`。一个使用通用应用推导出T的类型是`short`，因此可以精确匹配。对于`int`参数类型的重载`logAndAdd`也可以`short`类型提升后匹配成功。根据正常的重载解决规则，精确匹配优先于类型提升的匹配，所以被调用的是通用引用的重载。

在通用引用中的实现中，将`short`类型`emplace`到`std::string`的容器中，发生了错误。所有这一切的原因就是对于`short`类型通用引用重载优先于`int`类型的重载。

使用通用引用类型的函数在C++中是贪婪函数。他们机会可以精确匹配任何类型的参数（极少不适用的类型在Item 30中介绍）。这也是组合重载和通用引用使用是糟糕主意的原因：通用引用的实现会匹配比开发者预期要多得多的参数类型。

一个更容易调入这种陷阱的例子是完美转发构造函数。简单对`logAndAdd`例子进行改造就可以说明这个问题。将使用`std::string`类型改为自定义`Person`类型即可：

```
class Person
{
public:
  template<typename T>
  explicit Person(T&& n) :name(std::forward<T>(n)) {} // perfect forwarding ctor;
initializes data member
  explicit Person(int idx): name(nameFromIdx(idx)) {}
  ...
private:
  std::string name;
};
```

在`logAndAdd`的例子中，传递一个不是int的整型变量（比如`std::size_t, short, long`等）会调用通用引用的构造函数而不是int的构造函数，这会导致编译错误。这里这个问题甚至更糟糕，因为`Person`中存在的重载比肉眼看到的更多。在Item 17中说明，在适当的条件下，C++会生成拷贝和移动构造函数，即使类包含了模板构造也在合适的条件范围内。如果拷贝和移动构造被生成，Person类看起

来就像这样：

```
class Person
{
public:
  template<typename T>
  explicit Person(T&& n) :name(std::forward<T>(n)) {} // perfect forwarding ctor
  explicit Person(int idx); // int ctor

  Person(const Person& rhs); // copy ctor(complier-generated)
  Person(Person&& rhs); // move ctor (compiler-generated)
  ...
};
```

只有你在花了很多时间在编译器领域时，下面的行为才变得直观（译者注：这里意思就是这种实现会导致不符合人类直觉的结果，下面就解释了这种现象的原因）

```
Person p("Nancy");
auto cloneOfP(p); // create new Person from p; this won't compile!
```

这里我们视图通过一个`Person`实例创建另一个`Person`，显然应该调用拷贝构造即可（p是左值，我们可以思考通过移动操作来消除拷贝的开销）。但是这份代码不是调用拷贝构造，而是调用完美转发构造。然后，该函数将尝试使用Person对象p初始化`Person`的`std::string`的数据成员，编译器就会报错。

“为什么？”你可能会疑问，“为什么拷贝构造会被完美转发构造替代？我们显然想拷贝Person到另一个Person”。确实我们是这样想的，但是编译器严格遵循C++的规则，这里的相关规则就是控制对重载函数调用的解析规则。

编译器的理由如下：`cloneOfP`被`non-const`左值p初始化，这意味着可以实例化模板构造函数为采用`Person`的`non-const`左值。实例化之后，`Person`类看起来是这样的：

```
class Person {
public:
  explicit Person(Person& n)             // instantiated from
    : name(std::forward<Person&>(n)) {}  // perfect-forwarding

// template
  explicit Person(int idx);              // as before
  Person(const Person& rhs); // copy ctor (compiler-generated)
  ...
};
```

在`auto cloneOfP(p);`语句中，p被传递给拷贝构造或者完美转发构造。调用拷贝构造要求在p前加上const的约束，而调用完美转发构造不需要任何条件，所以编译器按照规则：采用最佳匹配，这里调用了完美转发的实例化的构造函数。

如果我们将本例中的传递的参数改为const的，会得到完全不同的结果：

```
const Person cp("Nancy");
auto cloneOfP(cp); // call copy constructor!
```

因为被拷贝的对象是const，是拷贝构造函数的精确匹配。虽然模板参数可以实例化为完全一样的函数签名：

```
class Person {
public:
  explicit Person(const Person& n); // instantiated from template

  Person(const Person& rhs); // copy ctor(compiler-generated)
  ...
};
```

但是无所谓，因为重载规则规定当模板实例化函数和非模板函数（或者称为“正常”函数）匹配优先级相当时，优先使用“正常”函数。拷贝构造函数（正常函数）因此胜过具有相同签名的模板实例化函数。

（如果你想知道为什么编译器在生成一个拷贝构造函数时还会模板实例化一个相同签名的函数，参考Item17）

当继承纳入考虑范围时，完美转发的构造函数与编译器生成的拷贝、移动操作之间的交互会更加复杂。尤其是，派生类的拷贝和移动操作会表现的非常奇怪。来看一下：

```
class SpecialPerson: public Person {
public:
  SpecialPerson(const SpecialPerson& rhs) :Person(rhs)
  {...} // copy ctor; calls base class forwarding ctor!
  SpecialPerson(SpecialPerson&& rhs): Person(std::move(rhs))
  {...} // move ctor; calls base class forwarding ctor!
};
```

如同注释表示的，派生类的拷贝和移动构造函数没有调用基类的拷贝和移动构造函数，而是调用了基类的完美转发构造函数！为了理解原因，要知道派生类使用`SpecialPerson`作为参数传递给其基类，然后通过模板实例化和重载解析规则作用于基类。最终，代码无法编译，因为`std::string`没有`SpecialPerson`的构造函数。

我希望到目前为止，已经说服了你，如果可能的话，避免对通用引用的函数进行重载。但是，如果在通用引用上重载是糟糕的主意，那么如果需要可转发大多数类型的参数，但是对于某些类型又要特殊处理应该怎么办？存在多种办法。实际上，下一个Item，Item27专门来讨论这个问题，敬请阅读。

需要记住的事

- 对通用引用参数的函数进行重载，调用机会会比你期望的多得多
- 完美转发构造函数是糟糕的实现，因为对于`non-const`左值不会调用拷贝构造而是完美转发构造，而且会劫持派生类对于基类的拷贝和移动构造