2、右值（r-值 or r-value）引用

Chapter	M
Name	xingzp

在第9章中，我们介绍了值类别的概念(9.2—值类别(左值和右值))，这是表达式的一个属性，帮助确定表达式是解析为值、函数还是对象。我们还介绍了l值和r值，以便讨论l值引用。

如果你对l值和r值还不清楚，现在是复习这个话题的好时机，因为我们将在本章中详细讨论它们。

左值(l-值 or l-value)引用回顾

在c++ 11之前，c++中只存在一种类型的引用，因此它被称为“引用”。然而，在c++ 11中，它被称为l-value引用。l值引用只能用可修改的l值初始化。

l-值引用	可以用来初始化	可以修改
可修改的l-值	Yes	Yes
不可修改的l-值	No	No
r-值	No	No

对const对象的l-值引用可以用可修改、不可修改的l-值和r-值进行初始化。但是，这些值不能被修改。

对const对象的l-值引用	可以用来初始化	可以修改
可修改的l-值	Yes	No
不可修改的l-值	Yes	No
r-值	Yes	No

对const对象的l-值引用特别有用，因为它们允许我们向函数传递任何类型的实参(L-value或r-value)，而不需要复制实参。

右值引用

c++ 11增加了一种新的引用类型，称为r-value引用。r值引用是被设计为用r值(仅)初始化的引用。l值引用是用一个&号创建的，r值引用是用双&号创建的:

```
int x{ 5 };
int &lref{ x }; // l-value reference initialized with l-value x
int &&rref{ 5 }; // r-value reference initialized with r-value 5
```

r值引用不能用l值初始化。

r-值引用	可以用来初始化	可以修改
可修改的l-值	No	No
不可修改的l-值	No	No
r-值	Yes	Yes

对const对象的r-值引用	可以用来初始化	可以修改
可修改的l-值	No	No
不可修改的l-值	No	No
r-值	Yes	Yes

r值引用有两个有用的属性。首先，r值引用将初始化它们的对象的生命周期延长到r值引用的生命周期(对const对象的l值引用也可以做到这一点)。其次，非const r值引用允许您修改r值!

让我们来看一些例子:

```
#include <iostream>

class Fraction
{
private:
  int m_numerator;
  int m_denominator;

public:
  Fraction(int numerator = 0, int denominator = 1) :
    m_numerator{ numerator }, m_denominator{ denominator }
  {
  }

  friend std::ostream& operator<<(std::ostream& out, const Fraction &f1)
  {
    out << f1.m_numerator << '/' << f1.m_denominator;
    return out;
  }
};

int main()
{
  auto &&rref{ Fraction{ 3, 5 } }; // r-value reference to temporary Fraction

  // f1 of operator<< binds to the temporary, no copies are created.
  std::cout << rref << '\n';

  return 0;
} // rref (and the temporary Fraction) goes out of scope here
```

上述程序打印：

```
3/5
```

作为匿名对象，Fraction(3,5)通常会在定义它的表达式的末尾超出作用域。然而，由于我们用它初始化一个r值引用，它的持续时间被延长到块的结束。然后我们可以使用r-value引用来打印分数的值。

现在让我们看一个不那么直观的例子:

```
#include <iostream>

int main()
{
    int &&rref{ 5 }; // because we're initializing an r-value reference with a litera
l, a temporary with value 5 is created here
    rref = 10;
    std::cout << rref << '\n';
```

```
    return 0;
}
```

上述程序打印：

```
10
```

虽然用字面值初始化一个r值引用然后能够更改该值看起来有些奇怪，但当用字面值初始化一个r值引用时，会从字面值构造一个临时对象，这样引用引用的是一个临时对象，而不是一个字面值。

r值引用并不经常以上述两种方式使用。

---

r值引用作为函数形参

r值引用通常用作函数形参。**当您希望l值和r值参数具有不同的行为时，这对于函数重载最有用**。

```
#include <iostream>

void fun(const int &lref) // l-value arguments will select this function
{
  std::cout << "l-value reference to const\n";
}

void fun(int &&rref) // r-value arguments will select this function
{
  std::cout << "r-value reference\n";
}

int main()
{
  int x{ 5 };
  fun(x); // l-value argument calls l-value version of function
  fun(5); // r-value argument calls r-value version of function

  return 0;
}
```

上述程序打印：

```
l-value reference to const
r-value reference
```

如您所见，当传递一个l-值时，重载函数解析为具有l-值引用的版本。当传递一个r-值时，重载函数解析到具有r-值引用的版本(这被认为是比l-值引用const更好的匹配)。

我们将在下一课中更详细地讨论这个问题。不用说，这是move语义的重要组成部分。

一个有趣的注意事项:

```
int &&ref{ 5 };
fun(ref);
```

实际上调用了函数的l值版本!尽管变量ref为r值类型引用一个整数，但它本身实际上是一个l值(所有命名变量都是如此)。混淆源于在两个不同的上下文中使用术语r-value。可以这样想:命名对象是l值。匿名对象是r值。命名对象或匿名对象的类型与它是l值还是r值无关。

返回一个r值引用

您几乎不应该返回r值的引用，原因与您几乎不应该返回l值的引用相同。在大多数情况下，当被引用的对象在函数结束时超出作用域时，您将最终返回一个悬空的引用。