

85 lvalues and rvalues in C++

本节开始将进入一些更高级的C++特性，包括move语义，但要理解它还需要很多前置知识，比如左值、右值。可以先不要着急给左值、右值下定义，慢慢理解即可。

1. 什么是左值、右值？

先从左值和右值的简单定义开始：

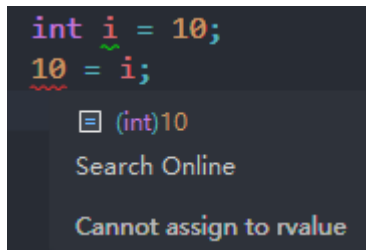
```
int i = 10;
```

这个表达式有两部分：左边和右边。

这也是一个考虑左、右值是什么的好方法，即左值绝大多数时候在等号左边，右值则绝大多数时候在右边。

但这不是一个完整的事实，不能总是这样理解，不过在这个例子中是对的。

我们有一个变量 `i`，这是一个在内存中有位置的实际变量，然后我们有一个值、一个数字字面量 `10`，它没有存储空间，没有位置，直到我们把它赋值给了左值 `i`。但是我们不能给一个右值赋值，比如我们不能说 `10 = i`，这会很奇怪。



```
int i = 10;
10 = i;
```

(int)10
Search Online
Cannot assign to rvalue

因为 `10` 没有位置，我们不能在 `10` 中存储数据。而 `i` 是一个左值，显然我们可以用另一个变量 `a` 让它等于 `i`。

```
int i = 10;
int a = i;
```

这个例子中，我们设置了一个左值等于一个同样是左值的值，这就是为什么说“等于号右边就是右值”这样的说法是不对的。

而右值不只能是这样字面量，还可以是函数的结果：

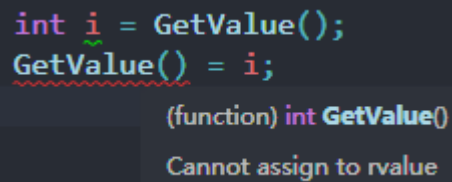
C++

```
int GetValue()
{
    return 10;
}

int main()
{
    int i = GetValue();
    int a = i;
}
```

我们调用这个函数，它返回一个临时值，因为这个返回的int没有位置，没有存储空间，只是一个值 10，所以也是右值。我们取这个右值、临时值，然后把它存储到左值 i 中。

因为GetValue()是一个右值，所以给它赋值是行不通的：



```
int i = GetValue();
GetValue() = i;
      (function) int GetValue()
      Cannot assign to rvalue
```

2. 左值引用

但如果函数返回的是一个左值，事情就变得有趣起来了，我们可以通过返回 **int&**（**左值引用**）来实现：

C++

```
// 我们需要给我们的值提供某种存储空间，比如使用一个静态int，然后返回它
int& GetValue()
{
    static int value = 10;
    return value;
}

int main()
{
    int i = GetValue();
    GetValue() = 5;    // 可以编译
}
```

这种情况下，函数返回的是一个左值引用，所以给它赋值的表达式就没问题了，这就是左值引用。

补充一点，我们有这样一个带参数的函数：

C++

```
void SetValue(int value) {}
```

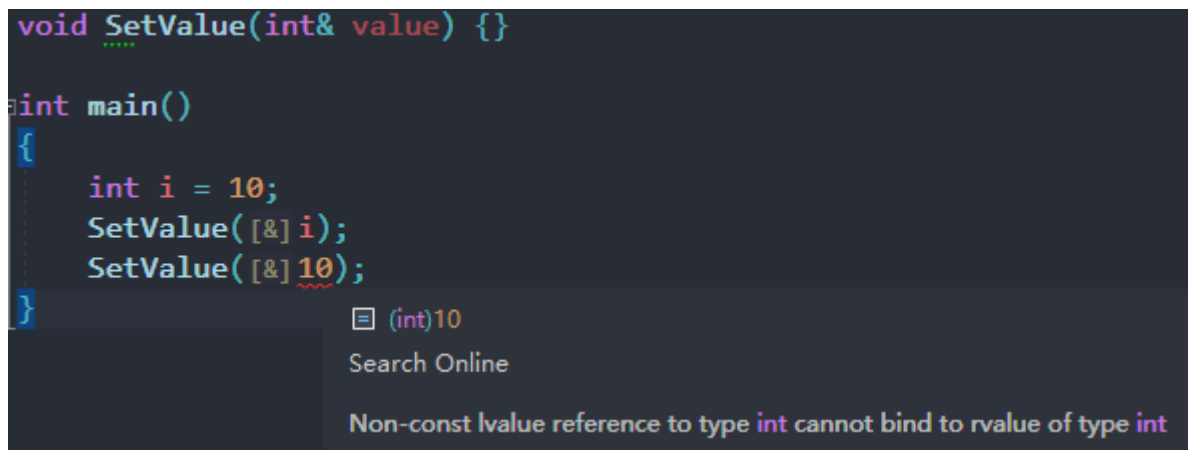
我们有多种方式可以调用它，可以用左值或右值来调用：

C++

```
int main()
{
    int i = 10;
    SetValue(i); // 参数是左值
    SetValue(10); // 参数是临时变量右值
}
```

这种情况下，当函数被调用时，这个右值会用来创建一个左值。

我们可以很容易得看出来哪个是临时的，哪个不是，因为有一个规则是：你不能将右值赋给左值引用，所以左值引用的只能是左值。在函数参数中加上一个 `&`，取一个左值引用，这里就马上得到了一个错误：



```
void SetValue(int& value) {}

int main()
{
    int i = 10;
    SetValue(&i);
    SetValue(&10);
}
```

(int)10
Search Online
Non-const lvalue reference to type 'int' cannot bind to rvalue of type 'int'

非const引用的初始值必须是左值。

实际上这里有一个特殊的规则，虽然我们不能用左值来引用右值，也就是说 `int& a = 10;` 是不行的，但如果是 `const` 加上左值引用就可以：

C++

```
const int& a = 10;
```

这只是一个特殊的规则，一种变通方法。

实际情况是，编译器可能会用你的存储创建一个临时变量，然后把它赋值给那个引用：

C++

```
int temp = 10;
const int& a = temp;
```

所以实际上，它不可避免地创建了一个左值，但同时支持了左值和右值。

因为如果把函数参数这里改成 `const int&`，就可以看到它现在同时可以接受两种类型数据了：

```

void SetValue(const int& value) {}

int main()
{
    int i = 10;
    SetValue(i);
    SetValue(10);
}

```

看另一个例子，请找出你认为的左值和右值都是什么：

```

int main()
{
    std::string firstName = "Yan";
    std::string lastName = "Chernikov";

    std::string fullName = firstName + lastName;
    std::cout << fullName << "\n";
}

```

揭晓答案：这个例子中左边的都是左值，右边的都是右值。

需要理解的应该是：`firstName + lastName` 组成了一个临时字符串，然后把它赋值给了一个左值。

```

// 设置一个左值引用打印名字的函数

void PrintName(std::string& name)
{
    std::cout << name << std::endl;
}

```

```

std::string fullName = firstName + lastName;

PrintName([&] fullName);
PrintName([&] firstName + lastName);

```

(local variable) `std::string firstName`
 Non-const lvalue reference to type `std::string` cannot bind to rvalue of type `std::string`

无法用临时变量作为参数

这就是为什么你会在C++中看到很多常量引用（如 `const int&`），因为它们能兼容临时的右值和实际存在的左值变量：

```

4 void PrintName(const std::string& name)
5 {
6     std::cout << name << std::endl;
7 }
8
9 int main()
10 {
11     std::string firstName = "Yan";
12     std::string lastName = "Chernikov";
13
14     std::string fullName = firstName + lastName;
15
16     PrintName(fullName);
17     PrintName(firstName + lastName);
18 }

```

那有没有办法写一个只接受临时对象的函数呢？

3. 右值引用

当然可以，这就是用到**右值引用**的时候了。

右值引用和左值引用很像，只不过要用到两个**&**：

```

void PrintName(const std::string&& name)
{
    std::cout << name << std::endl;
}

int main()
{
    std::string firstName = "Yan";
    std::string lastName = "Chernikov";

    std::string fullName = firstName + lastName;

    PrintName(fullName);
    PrintName(firstName) (local variable) std::string fullName
}

```

Rvalue reference to type `const std::string` cannot bind to lvalue of type `std::string`

可以发现报错位置交换了，现在不能使用左值作为参数，但是可以传递右值。

因为有了右值引用，我们现在有了一种检测临时值，并对它们做一些特殊事情的方法：

```
void PrintName(const std::string& name)
{
    std::cout << "[lvalue]" << name << std::endl;
}
```

```
void PrintName(std::string&& name)
{
    std::cout << "[rvalue]" << name << std::endl;
}
```

/* 这里定义了两个`PrintName`函数版本。其中一个接受`const std::string&`参数（常量左值引用），另一个接受`std::string&&`参数（右值引用）。

当调用`PrintName`时，编译器会尝试选择最合适重载版本。它的选择是基于引用折叠规则和左值/右值属性来进行的。

1. `PrintName(fullName);`:

- 在这里，`fullName`是一个左值，因为它有一个名称，并且它可以被寻址。
- 选择`const std::string&`版本是因为左值更倾向于绑定到左值引用（包括常量左值引用）而不是右值引用。因此，`[lvalue]`会被输出。

2. `PrintName(firstName + lastName);`:

- `firstName + lastName`返回的是一个临时的`std::string`对象，这是一个右值，因为它不具有持久的名称，且不能被寻址。
- 对于临时对象（也即右值），它们更倾向于绑定到右值引用。所以，这里选择了`std::string&&`版本的重载。因此，`[rvalue]`会被输出。

*/

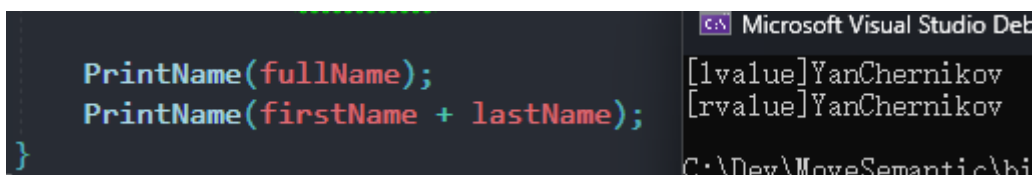
```
int main()
{
    std::string firstName = "Yan";
    std::string lastName = "Chernikov";

    std::string fullName = firstName + lastName;

    PrintName(fullName);
    PrintName(firstName + lastName);
}
```

/* 尽管`const std::string&`版本的函数可以接受左值和右值，但如果存在专门为右值设计的版本（也即`std::string&&`版本），当传递右值时，编译器会优先选择这个版本。

这种设计的主要目的是允许开发者提供特定于右值的优化，如移动语义，从而提高代码的性能。*/



```
PrintName(fullName);
PrintName(firstName + lastName);
}
```

Microsoft Visual Studio Deb
[lvalue]YanChernikov
[rvalue]YanChernikov
C:\Dev\MoveSemantic\bi

这里主要的优势在于优化，如果我们知道传入的是一个临时对象的话，那么我们就不需要考虑它们是否活着、是否完整、是否拷贝，我们可以简单地偷它的资源，给到特定的对象，或者在其它地方使用它们，因为我们知道它是暂时的。不会存在很长时间，只会在这个PrintName中使用。

而如果用左值引用这样的函数，你就不能从name字符串中窃取任何东西，因为它可能再很多函数中使用。

小补充

在这里，“偷”用于描述在移动语义中将资源从一个对象转移到另一个对象的过程。这并不是字面意义上的偷窃，而是一个比喻，用来形象地解释这一过程。

来详细解释一下：

1. **传统的复制操作**：当你创建一个对象的复制时，所有的资源（例如内存）都会被复制到新的对象。这可能是一个昂贵的操作，特别是如果对象持有大量资源时。
2. **移动语义和资源的“偷取”**：移动语义允许你不是复制资源，而是将它们从一个对象移动到另一个对象。在这个过程中，原始对象通常会被置于一个有效但未定义的状态（例如，其指针成员可能被设置为 `nullptr`）。由于资源已经被“偷走”（即转移到新对象），所以原始对象不再拥有它们。

例如，考虑以下代码段：

```
std::string str1 = "Hello, world!";  
std::string str2 = std::move(str1);
```

C++

在上述代码中，`str2`的构造不会复制`str1`的字符串内容。相反，它将“偷走”（即移动）`str1`的内部指针，现在该指针指向的内存属于`str2`。`str1`被置于一个有效但未定义的状态，通常不应再使用。

通过避免不必要的复制，移动语义可以显著提高性能。这在处理大对象和容器时尤为重要，例如，当你在处理大型数组或矩阵时，或者在构造和返回临时对象时。

4. 总结

总之记住：

- 左值是某种**存储支持的变量**，右值是**临时值**；
- 左值引用之接受左值，除非用 `const`，右值引用只接受右值

本节主要是帮助对此话题感到困惑的人，用于 *clear the air*（消除分歧）关于左值和右值是什么。随着后面深入了解了移动语义，就会清楚地认识到它为什么有用了。如果你在处理一个右值引用，你能从那个临时值中偷取资源（因为它是临时的），这对优化会有很大帮助。