(87条消息) 简单聊聊C/C++中的左值和右值_AlbertS Home of Technology-CSDN博客

历史渊源

这个世界一直是在变化的，可能之前你一直引以为豪的经验大楼，转眼之间就会倾塌。关于左值和右值的历史，普遍的观点是最初来源于 `C` 语言，后来被引入到了 `C++`，但是关于左值和右值的含义和实现却在一直改变和完善，对于它的历史讲解发现一篇总结的比较好的文章 《C/C++ 左值和右值, L-value和R-value》。

这是2012年的一篇文章，文中给出了历史说明依据，最后还举了一些例子来说明 `C` 和 `C++` 关于左值实现的不同，但是实际操作后你会发现，时间的车轮早已向前行进了一大截，文中提到的那些不同，在最新的 `gcc` 和 `g++` 编译器上早已变得相同，文中提到的反例现在看来几乎没有意义了。

简单梳理下，左值的定义最早出现在 《The C Programming Language》一书中，指的是引用一个对象，放在赋值表达式 `=` 左边的值。

后来在新的 `C` 语言标准中提到左值是赋值表达式 `=` 左边的值或者需要被改变的值，而等号的右边的值被称为右值。左值更好的表达为可以定位的值，而右值是一种表达数据的值，基于这个表述 `L-value` 可以理解为 `locator value`，代表可寻址，而 `R-value` 可以理解为 `read value`，代表可读取。

不过以上的新解，完全是人们为了理解左值、右值赋予的新含义，从历史发展来看，一开始左值和右值完全就是通过等号的左边和右边来命名的，只不过随着标准的完善和语言的发展、更替，虽然两个名字保留了下来，但是它们的含义却在逐步发生改变，与最初诞生时的 `=` 左右两边的值这个含义相比，已经相差很多了。

认识左值和右值

关于左值右值有几条规则和特点，先列举在这里，后面可以跟随例子慢慢体会：

1. 左值和右值都是指的表达式，比如 `int a = 1` 中的 `a` 是左值，`++a` 是左值，`func()` 也可能是左值，而 `a+1` 是右值，`110` 也是一个右值。
2. 左值可以放在 `=` 的左边，右值只能放在 `=` 的右边，这其中隐含的意思就是左值也能放在 `=` 的右边，但是右值不能放在 `=` 的左边。
3. 左值可以取地址，代表着内存中某个位置，可以存储数据，右值仅仅是一个值，不能取地址，或者它看起来是一个变量，但它是临时的无法取地址，例如一个函数的非引用的值返回。

以上规则从定义来看一点也不严谨，比如一个常量定义是可以赋值，后面就不行了，它也可以取地址，但是不能赋值的它到底是左值还是右值，这点其实不用纠结，心里知道这个情况就可以了。

再比如一个普通变量，它原本是一个左值，当用它给其他变量赋值的时候，它又化身为一个右值，这时它也可以取地址，好像与上面的说法相违背了，但是仔细想想真的是这样吗？它只是临时化身为右值，其实是一个左值，所以才可以取地址的。

其实你如果不做学术研究、不斤斤计较，那么完全可以把能够赋值的表达式作为左值，然后把左值以外的表达式看成右值，如果你不熟悉解左值和右值可能根本不会影响你平时的工作和学习，但是了解它有助于我们深入理解一些内置运算符和程序执行过程，以及在出现编译错误的时候及时定位问题。

具体的示例

最简单的赋值语句

```
1 | int age = 18;
```

这个赋值语句很简单，`=` 作为分界线，左边的 `age` 是左值，可以被赋值，可以取地址，它其实就是一个表达式，代表一个可以存储整数的内存地址；右边的 `18` 也是一个表达式，明显只能作为右值，不能取地址。

```
1 | 18 = age;
```

这个语句在编译时会提示下面的错误：

```
1 | error: lvalue required as left operand of assignment
```

错误提示显示：赋值语句的左边需要一个左值，显然 `18` 不能作为左值，它不代表任何内存地址，不能被改变。

如果程序中的表达式都这么简单就不需要纠结了，接着我们往下看一些复杂点的例子。

自增自减运算

```
++age++;
```

第一眼看到这个表达式，你感觉它会怎样运算，编译一下，你会发现编译失败了，错误如下：

> error: lvalue required as increment operand

加个括号试试：

```
++(age++)
```

编译之后会出现相同的错误：

> error: lvalue required as increment operand

再换一种加括号的方式再编译一次：

```
(++age)++
```

这次成功编译了，并且输出值之后发现 `age` 变量增加了两次。

先不考虑左值右值的问题，我们可以从这个例子中发现自增运算的优先级，后置自增 `age++` 的优先级要高于前置自增 `++age` 的优先级。

现在回过头来看看之前的编译错误，为什么我们加括号改变运算顺序之后就可以正常执行了呢？这其实和自增运算的实现有关。

前置自增

前置自增的一般实现，是直接修改原对象，在原对象上实现自增，然后将原对象以引用方式返回：

```
UPInt& UPInt::operator++()
{
  *this += 1;    // 原对象自增
  return *this;  // 返回原对象
}
```

这里一直操作的是原对象，返回的也是原对象的引用，所以前置自增表达式的结果是左值，它引用的是原对象之前所占用的内存。

后置自增

后置自增的一般实现，是先将原对象的数据存储到临时变量中，接着在原对象上实现自增，然后将临时变量以只读的方式返回：

```
const UPInt UPInt::operator++(int)
{
  UPInt oldValue = *this; // 将原对象赋值给临时变量
  ++(*this);              // 原对象自增
  return oldValue;        // 返回临时变量
}
```

这里返回的是临时变量，在函数返回后就被销毁了，无法对其取地址，所以后置自增表达式的结果是右值，不能对其进行赋值。

所以表达式 `++age++;` 先进行后置自增，然后再进行前置自增就报出编译错误了，因为不能修改右值，也不能对右值进行自增操作。

自增表达式赋值

前面说到前置自增表达式是一个左值，那能不能对其赋值呢？当然可以！试试下面的语句：

```
++age = 20;
```

这条语句是可以正常通过编译的，并且执行之后 `age` 变量的值为 `20`。

函数表达式

函数可以作为左值吗？带着这个疑问我们看一下这个赋值语句：

```
func() = 6;
```

可能有些同学会有疑问，这是正常的语句吗？其实它是可以正常的，只要 `func()` 是一个左值就可以，怎么才能让他成为一个左值呢，想想刚才的前置自增运算可能会给你启发，要想让他成为左值，它必须代表一个内存地址，写成下面这样就可以了。

```
int g;

int& func()
{
    return g;
}

int main()
{
    func() = 100;
}
```

函数 `func()` 返回的是全局变量 `g` 的引用，变量 `g` 是一个可取地址的左值，所以 `func()` 表达式也是一个左值，对其赋值后就改变了全局变量 `g` 的值。

那么我们注意到这里 `func()` 函数返回的是全局变量的引用，如果是局部变量会怎么样呢？

那么我们注意到这里 `func()` 函数返回的是全局变量的引用，如果是局部变量会怎么样呢？

```
int& func()
{
    int i = 101;
    return i;
}

int main()
{
    func() = 100;
}
```

上面的代码编译没有错误，但是会产生一个警告，提示返回了局部变量的引用：

```
warning: reference to local variable ‘i’ returned [-Wreturn-local-addr]
```

运行之后可就惨了，直接显示段错误：

```
Segmentation fault (core dumped)
```

改为局部变量之后，`func()` 函数虽然返回了一个值，但是这个值是一个临时值，函数返回之后该值被销毁，对应的内存空间也不属于它了，所以在最后赋值的时候才会出现段错误，就和我们访问非法内存是产生的错误时一样的。