链滴

给 C++ 增加“取引用”运算符

作者：localvar
原文链接：https://ld246.com/article/1521519405371
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

一篇胡思乱想之作。

1. 程序实践的需要

首先问一个问题：在C++语言中，“int”和“int&”是相同的数据类型吗？我想大家肯定回答：“然不是，它们连兼容类型（如int和short的关系）都不是。”甚至有些人会想：这家伙脑袋是不是不常，怎么问这么弱的问题？

好，既然不是相同的类型，那以它们为区别的函数重载就应该是合法的、没有二义性的，但我相信你试也知道下面的代码是编译不过去的，因为它有二义性。

```
void foo( int a );
void foo( int& a );
......
int m = 10;
foo( m );    // 错误，二义性
```

而二义性的原因是程序员无法明确说明他想按引用传参还是按值传参，但这种情况很罕见，因为恐怕有几个人会这样定义重载函数，所以问题不大。不过下面的情况就不一样了：

```
//以下代码是甲写的
void foo( int a );
//以下代码是乙写的
int m = 10;
foo( m );
```

乙的代码调用甲的函数，一切良好，编译、链接、测试全部通过。但突然有一天，甲觉得让foo的参是“int&”会更好，所以他改了自己的程序，乙拿到甲的新版代码，发现编译链接毫无问题，就放心发布了，与此同时，一个严重的bug也潜伏下来了。但这关C++什么事吗？甲应该告诉乙他对代码的改，乙也应该对自己的产品进行完整的测试呀？是的，这两点没错，但我觉得C++也应该对此负责，为它应该帮助乙避免这种错误，而且它在另外一种类似的情形下也确实这么做了，看下面的例子（参Effective C++，Item 26编写）：

```
//以下代码是甲写的
class Base1
{
public:
    void doIt();
};

class Base2
{
private:
    void doIt();
};
//以下代码是乙写的
// Devried没有定义自己的doIt
class Devried : public Base1, public Base2
{
};
......
Devried d;
d.doIt();            //①
```

大家说①处的代码调的是哪一个doIt？当然是Base1的那个，因为Base2的那个是私有的，在这里无访问；那①处的代码有二义性吗？当然没有，编译器可以很容易得知道其目的；那它能编译过去吗？然不能，因为编译器不允许“因为改变类成员的保护级别而改变程序的含义”。在这里，如果编译器它通过了，那么当有一天甲把Base1的改为private，Base2的改成public，编译器应该也让它通过。这时，乙的程序的含义就被偷偷修改了，与前面的例子很相似吧？但因为编译器有自己的那条原则，以①处的代码编译不了，乙必须明确的用

```
d.Base1::doIt();
```

才行。这样，如果甲再做修改，这一行就会出错，乙就能及时跟进了。

其实，我认为这两个问题都属于“潜在的二义性”问题，照理说编译器应该一视同仁，给予相同的处。可它为什么对第二种大嚷大叫，却对第一种不闻不问呢？原因还是在于，第一种情况下，C++没有乙“明确指出传值还是传引用”的权力。

2. 不完备的C++类型系统

C++本身的类型系统庞大而复杂，而且还给了程序员创建与内置类型具有相同行为模式的自定义类型能力，但它在引用类型上却有一些缺陷。从汇编的角度，我想大家都知道，引用和指针几乎是一模一的。但指针有一个取指针（取地址）运算符“&”和一个取值运算符“*”，用来在指针和它所指的对之间作相互转换，而引用却没有，两个都没有。C++对此的解决办法是让引用类型必须在定义的时候初值，并且终生不能改变其所引用的对象。我们初学C++时就知道引用类型有此限制，但恐怕很少有想过这是为什么。好了，看下面的代码：

```
//假设sometype和整数是兼容类型
sometype m = 10, n = 11;
sometype& r = m;
r = n;                    //①
```

如果没有那个限制，语句①就有二义性了。程序员是想把n的值赋给r（也就是m）呢，还是想让r成为的引用呢？编译器那个愁啊，最后终于决定：引用类型必须在定义时赋初值[注1]，这里的语义只能是n的值赋给r。不过这一限制使引用看起来像一个“劣等公民”，让整个C++类型系统显得稍微有些不谐。

3. 增加一个“取引用”运算符又如何？

在第一节的讨论中，如果有一个“按引用传参”运算符[注2]，C++只需规定：对引用类型的参数必使用按引用传参运算符，就不会有问题了（为了后文讨论方便，我们假设这个运算符用“@”表示）因为，一旦甲把foo的参数改成了“int&”，而乙在调用时没有用“foo(@m)”，这一行就是个编错误，乙就必须进行修改了。

不过，我们似乎没有理由让它只用于函数调用，它也应该能用于程序的任何位置，就像取指针运算符样。所以，它应该叫“取引用”运算符，而不是“按引用传参”运算符。更进一步，为了类型系统的备，我们应该也有一个用于引用的“取值”运算符，为避免混淆，就叫它“解引用”运算符吧，其符定位“$”。

啊哈！大功告成！现在我宣布，一门全新的语言—C+++—诞生了。让我们看看这门语言激动人心的特性吧。首先，第一节中的问题彻底成了历史，因为我们有了强大的“@”运算符。其次，引用类型义时赋初值的限制没有了，因为语义可以通过“@”作明确的表达，如：

```
int m = 10, n = 11;
int& r;
r = @m;       //让r成为m的引用
```

```
r = n;      //把n的值赋给r
r = @n;     //让r成为n的引用
```

不过这些都是“@”的用处，“$”怎么用呢？没问题，接着看下面的例子：

```
int m = 10;
int& r1 = @m;
int n = $r1;   //①
int$ r2 = @n;
r2 = $r1;    //② 把r1引用的值赋给r2
r2 = r1;     //让r2也去引用m
```

怎么样，还不错吧？不过语句①显得有点画蛇添足了，因为这里有没有“”，语义都一样，只能是把r引用的值赋给n。好，既然如此，那就再对我们的新语言进行一个小修改，规定“”只用于语句②的况，在①中不能用，也就是说由引用类型向被引用类型赋值不需要“$”。

好像很完美了，但C++允许通过重载“[]”运算符来定义程序员自己的数组类，我们的C+++当然也支持。而且，为了支持对数组元素的赋值，“[]”的返回值肯定应该是引用类型，所以我们可以写下的代码：

```
//array是程序员自定义的整形数组的实例
array[0] = 10;
int n = array[1];      //①
int& r = @n;
array[3] = r;          //②
```

语句②好像有些不大对，array[3]是“int&”类型，所以这一句的含义是让“array的‘[]’运算符的回值”去引用“r的引用对象”，而不是我们所要达到的把“r引用的值”赋给array[3]。当然，出现一问题的原因是我们没有按C+++的语法规则去写代码，只要把它改成

```
array[3] = $r;
```

就可以了。

可细想一下就又不对了，如果array是语言的内置数组类型，那么array[3]就是“int”类型，我们刚刚定，这种情况不用“”。而自定义类型的行为模式应该和内置类型保持一致，所以……，我们还是把才那条规定废了吧，毕竟多写一个“”不会带来什么问题，C+++是新生事物，在一些地方出现反复所难免，大家说是吧？

但编译器马上叫起来：“语句①有错误，缺少‘’运算符”。我们的编译器真不错，尽职尽责，它要说我还真就忘了，按我们最新的语法规则，这里确实应该有一个“”。但话还得说回来，如果array内置数组类型，这个“”是不应该加的，自定义类型的行为又何内置类型不一样了，这可怎么办呢？不咱就再加条规定，让内置类型在这种情况下也加个“”？好，说干就干，现在那段代码变成了：

```
//array是程序员自定义的整形数组的实例
array[0] = 10;
int n = $array[1];      //①
int& r = @n;
array[3] = $r;          //②
```

重新编译，编译器又叫起来：“语句②有二义性。”什么？二义性？刚才还好好的，加了一条规则怎就有二义性了呢？仔细看一下，它还真没说错。如果array是内置类型，则array[1]是“int”型，所以句①加“”就意味着同类型变量之间赋值要加“”。再来看②，array[3]是“int&”，r也是“int&”所以它的含义又变回了让“array的‘[]’运算符的返回值”去引用“r的引用对象”。幸亏聪明的编器猜出了我们的真实意愿，报了个二义性错误，否则，这样的bug怎么找呀？

看来，语法规则还得改。我又重新检查了前面的一系列问题，发现其根源都在于“引用类型可以相互值”，只要把它禁用，所有问题都将迎刃而解，而且这也让“$”在普通的表达式中失去存在的必要从而使语法更简洁。终于，我们的代码又回到了最初的形式（注意：这里并没有规定引用必须在定义赋初值，也没有规定引用的对象必须终生不变）：

```
//array是程序员自定义的整形数组的实例
array[0] = 10;
int n = array[1];
int& r = @n;
array[3] = r;        //①
```

因为引用之间不能赋值，所以语句①没有任何问题

时间一天天过去，一切都很正常，直到有一天，程序员写出了下面的代码：

```
//array是程序员自定义的整形数组的实例
array[0] = 10;
int& r = @array[1];    //①
int n = array[2];
r = n;                 //②
```

编译器又抱怨了：“语句①有错误，类型不匹配”。说实话，我的心已经经受不了这种打击了，不过是得硬着头皮看看到底出了什么事：又是自定义类型和内置类型行为模式不一致。由于array是程序自己定义的，所以array[1]的类型是“int&”，@array[1]的类型是“int&&”，但表达式的左面却个“int&”，难怪了（注意：这里是引用的初始化，而不是把“int&&”赋值给“int&”，后者是“引用类型向被引用类型赋值”，没有错误）。但如果array是内置的数组类型，该表达式又没有任何题。这可如何是好呢？要不让内置类型在这种情况下也不用加“@”？别急，我得好好分析一下，我不想重蹈覆辙了。

如果内置类型也不用加“@”，那将导致可以通过赋值语句直接让一个“int&”型变量成为一个“in”型变量的引用，可这又会使语句②出现二义性错误，要修正这个错误，就必须规定引用型变量只能定义时赋初值，但这个规定又让“@”像“$”一样在普通表达式中失去存在的价值。

经过了一系列的调整，“@”和“$”的作用就只剩下在函数调用中指定参数类型了，但前面那么多问题，使我对它们这唯一的用途也产生了怀疑。而且，残酷的事实也表明，无情的自定义类型并没有过我最后的这根救命稻草，看下面的代码吧：

```
void foo( int a );
void bar( int& a );
foo( array[0] );        //array是内置数组类型
foo( $array[0] );        //array是自定义数组类型
bar( @array[0] );        //array是内置数组类型
bar( array[0] );        //array是自定义数组类型
```

自定义类型和内置类型的不同行为模式，让我的C+++之梦彻底成了泡影。

4. 结论

真没想到增加一个“取引用”运算符会引出这么多事来，不过这也说明了C++没有引入这个运算符的因。写完这篇文章，突然觉得我们这个世界好像并不喜欢完美的事物，类型系统不完备的C++可以大其道，类型系统完备的C+++却只能是空中楼阁，真是不可思议。

注释

1. 实际上，如果sometype是函数类型的话，即使没有“定义时赋初值”的限制，语句①也没有二义。据我所知，VC6就允许“函数的引用”不在定义时赋初值，但到VC7和VC7.1就不行了。

2. C#就引入了这样一个运算符，也就是关键字“ref”，不过从我个人来讲，更希望它是一个符号而是个单词，另外它的作用和本文所希望的也不是完全一致。