|  |  |  |
| --- | --- | --- |
|  | 不修改 | 修改 |
| 内置数据类型 | int var  const int& var(不建议) | int\* var  int &var |
| 类 | const Cat &var  Cat var(不建议) | Cat &var  Cat \*var（不建议） |
| 结构体 | Node var  const Node& var | Node& var  Node\* var |

T var

var如果是内置数据类型，为形参分配存储单元(变量)，实参的值赋给该变量。在函数体内，该变量的值可以随意修改。实参变量完全不受影响，不会被修改。

var如果是对象，为对象分配存储单元(变量)，实参可以是对象，也可以是对象别名。在函数体内，对象可以调用非const成员函数，修改自己数据成员。

T&var

var如果是内置数据类型，形参可以是变量，也可以是变量的引用。不会为形参分配存储单元，相当于直接var这个实参，这个存储单元（变量）在函数体内使用，会修改实参变量的值。

var如果是对象，相当于在函数体内，直接拿着var，拿着实参变量，可以利用 . 任意调用对象的成员函数。

const T var

const int var = 100; int b = var;//OK const &b = var;//error

如果是内置数据类型，为形参分配存储单元(变量)，实参的值赋给该变量。在函数体内，拿着var，就相当于拿着一个固封的存储单元（变量）使用。实参变量完全不受影响，不会被修改。

如果是对象，为形参分配存储单元(变量)，实参的值赋给该变量。在函数体内，拿着var，就像拿着全部数据成员被固封的变量，不得调用非const成员函数。

const T&var

如果是内置数据类型，相当于拿着固封的实参，在函数体中使用。不会改变实参变量的值。

如果是对象，相当于拿着固封的实参对象，无法在函数体中调用非const成员函数。

**一、引用简介**

　　引用就是某一变量（目标）的一个别名，对引用的操作与对变量直接操作完全一样。

　　引用的声明方法：类型标识符 &引用名=目标变量名；

　　【例1】：int a; int &ra=a; //定义引用ra,它是变量a的引用，即别名

　　说明：

　　（1）&在此不是求地址运算，而是起标识作用。

　　（2）类型标识符是指目标变量的类型。

　　（3）声明引用时，必须同时对其进行初始化。

　　（4）引用声明完毕后，相当于目标变量名有两个名称，即该目标原名称和引用名，且不能再把该引用名作为其他变量名的别名。

　　 ra=1; 等价于 a=1;

　　（5）声明一个引用，不是新定义了一个变量，它只表示该引用名是目标变量名的一个别名，它本身不是一种数据类型，因此引用本身不占存储单元，系统也不给引用分配存储单元。故：对引用求地址，就是对目标变量求地址。&ra与&a相等。

　　（6）不能建立数组的引用。因为数组是一个由若干个元素所组成的集合，所以无法建立一个数组的别名。

**二、引用应用**

　　1、引用作为参数

　　引用的一个重要作用就是作为函数的参数。以前的C语言中函数参数传递是值传递，如果有大块数据作为参数传递的时候，采用的方案往往是指针，因为 这样可以避免将整块数据全部压栈，可以提高程序的效率。但是现在（C++中）又增加了一种同样有效率的选择（在某些特殊情况下又是必须的选择），就是引 用。

　　【例2】：

void swap(int &p1, int &p2) //此处函数的形参p1, p2都是引用  
{ int p; p=p1; p1=p2; p2=p; }

　　为在程序中调用该函数，则相应的主调函数的调用点处，直接以变量作为实参进行调用即可，而不需要实参变量有任何的特殊要求。如：对应上面定义的swap函数，相应的主调函数可写为：

main( )  
{  
　int a,b;  
　cin>>a>>b; //输入a,b两变量的值  
　swap(a,b); //直接以变量a和b作为实参调用swap函数  
　cout<<a<< ' ' <<b; //输出结果  
}

　　上述程序运行时，如果输入数据10 20并回车后，则输出结果为20 10。

　　由【例2】可看出：

　　（1）传递引用给函数与传递指针的效果是一样的。这时，被调函数的形参就成为原来主调函数中的实参变量或对象的一个别名来使用，所以在被调函数中对形参变量的操作就是对其相应的目标对象（在主调函数中）的操作。

　　（2）使用引用传递函数的参数，在内存中并没有产生实参的副本，它是直接对实参操作；而使用一般变量传递函数的参数，当发生函数调用时，需要给 形参分配存储单元，形参变量是实参变量的副本；如果传递的是对象，还将调用拷贝构造函数。因此，当参数传递的数据较大时，用引用比用一般变量传递参数的效 率和所占空间都好。

　　（3）使用指针作为函数的参数虽然也能达到与使用引用的效果，但是，在被调函数中同样要给形参分配存储单元，且需要重复使用"\*指针变量名"的 形式进行运算，这很容易产生错误且程序的阅读性较差；另一方面，在主调函数的调用点处，必须用变量的地址作为实参。而引用更容易使用，更清晰。

　　如果既要利用引用提高程序的效率，又要保护传递给函数的数据不在函数中被改变，就应使用常引用。

　　2、常引用

　　常引用声明方式：const 类型标识符 &引用名=目标变量名；

　　用这种方式声明的引用，不能通过引用对目标变量的值进行修改,从而使引用的目标成为const，达到了引用的安全性。

　　【例3】：

int a ;  
const int &ra=a;  
ra=1; //错误  
a=1; //正确

　　这不光是让代码更健壮，也有些其它方面的需要。

　　【例4】：假设有如下函数声明：

string foo( );  
void bar(string & s);

　　那么下面的表达式将是非法的：

bar(foo( ));  
bar("hello world");

　　原因在于foo( )和"hello world"串都会产生一个临时对象，而在C++中，这些临时对象都是const类型的。因此上面的表达式就是试图将一个const类型的对象转换为非const类型，这是非法的。

　　引用型参数应该在能被定义为const的情况下，尽量定义为const 。

　　3、引用作为返回值

　　要以引用返回函数值，则函数定义时要按以下格式：

类型标识符 &函数名（形参列表及类型说明）  
{函数体}

　　说明：

　　（1）以引用返回函数值，定义函数时需要在函数名前加&

　　（2）用引用返回一个函数值的最大好处是，在内存中不产生被返回值的副本。

　　【例5】以下程序中定义了一个普通的函数fn1（它用返回值的方法返回函数值），另外一个函数fn2，它以引用的方法返回函数值。

#include <iostream.h>  
float temp; //定义全局变量temp  
float fn1(float r); //声明函数fn1  
float &fn2(float r); //声明函数fn2  
float fn1(float r) //定义函数fn1，它以返回值的方法返回函数值  
{  
　temp=(float)(r\*r\*3.14);  
　return temp;  
}  
float &fn2(float r) //定义函数fn2，它以引用方式返回函数值  
{  
　temp=(float)(r\*r\*3.14);  
　return temp;  
}  
void main() //主函数  
{  
　float a=fn1(10.0); //第1种情况，系统生成要返回值的副本（即临时变量）  
　float &b=fn1(10.0); //第2种情况，可能会出错（不同 C++系统有不同规定）  
　//不能从被调函数中返回一个临时变量或局部变量的引用  
　float c=fn2(10.0); //第3种情况，系统不生成返回值的副本  
　//可以从被调函数中返回一个全局变量的引用  
　float &d=fn2(10.0); //第4种情况，系统不生成返回值的副本  
　//可以从被调函数中返回一个全局变量的引用  
　cout<<a<<c<<d;  
}

　　引用作为返回值，必须遵守以下规则：

　　（1）不能返回局部变量的引用。这条可以参照Effective C++[1]的Item 31。主要原因是局部变量会在函数返回后被销毁，因此被返回的引用就成为了"无所指"的引用，程序会进入未知状态。

　　（2）不能返回函数内部new分配的内存的引用。这条可以参照Effective C++[1]的Item 31。虽然不存在局部变量的被动销毁问题，可对于这种情况（返回函数内部new分配内存的引用），又面临其它尴尬局面。例如，被函数返回的引用只是作为一 个临时变量出现，而没有被赋予一个实际的变量，那么这个引用所指向的空间（由new分配）就无法释放，造成memory leak。

　　（3）可以返回类成员的引用，但最好是const。这条原则可以参照Effective C++[1]的Item 30。主要原因是当对象的属性是与某种业务规则（business rule）相关联的时候，其赋值常常与某些其它属性或者对象的状态有关，因此有必要将赋值操作封装在一个业务规则当中。如果其它对象可以获得该属性的非常 量引用（或指针），那么对该属性的单纯赋值就会破坏业务规则的完整性。

　　（4）引用与一些操作符的重载：

　　流操作符<<和>>，这两个操作符常常希望被连续使用，例如：cout << "hello" << endl;　因此这两个操作符的返回值应该是一个仍然支持这两个操作符的流引用。可选的其它方案包括：返回一个流对象和返回一个流对象指针。但是对于返回 一个流对象，程序必须重新（拷贝）构造一个新的流对象，也就是说，连续的两个<<操作符实际上是针对不同对象的！这无法让人接受。对于返回一 个流指针则不能连续使用<<操作符。因此，返回一个流对象引用是惟一选择。这个唯一选择很关键，它说明了引用的重要性以及无可替代性，也许这 就是C++语言中引入引用这个概念的原因吧。 赋值操作符=。这个操作符象流操作符一样，是可以连续使用的，例如：x = j = 10;或者(x=10)=100;赋值操作符的返回值必须是一个左值，以便可以被继续赋值。因此引用成了这个操作符的惟一返回值选择。

　　【例6】 测试用返回引用的函数值作为赋值表达式的左值。

#include <iostream.h>  
int &put(int n);  
int vals[10];  
int error=-1;  
void main()  
{  
put(0)=10; //以put(0)函数值作为左值，等价于vals[0]=10;  
put(9)=20; //以put(9)函数值作为左值，等价于vals[9]=20;  
cout<<vals[0];  
cout<<vals[9];  
}  
int &put(int n)  
{  
if (n>=0 && n<=9 ) return vals[n];  
else { cout<<"subscript error"; return error; }  
}

　　（5）在另外的一些操作符中，却千万不能返回引用：+-\*/ 四则运算符。它们不能返回引用，Effective C++[1]的Item23详细的讨论了这个问题。主要原因是这四个操作符没有side effect，因此，它们必须构造一个对象作为返回值，可选的方案包括：返回一个对象、返回一个局部变量的引用，返回一个new分配的对象的引用、返回一 个静态对象引用。根据前面提到的引用作为返回值的三个规则，第2、3两个方案都被否决了。静态对象的引用又因为((a+b) == (c+d))会永远为true而导致错误。所以可选的只剩下返回一个对象了。

　　4、引用和多态

　　引用是除指针外另一个可以产生多态效果的手段。这意味着，一个基类的引用可以指向它的派生类实例。

　　【例7】：

class 　A;  
class 　B：public A{……};  
B 　b;  
A 　&Ref = b; // 用派生类对象初始化基类对象的引用

　　Ref 只能用来访问派生类对象中从基类继承下来的成员，是基类引用指向派生类。如果A类中定义有虚函数，并且在B类中重写了这个虚函数，就可以通过Ref产生多态效果。

**三、引用总结**

　　（1）在引用的使用中，单纯给某个变量取个别名是毫无意义的，引用的目的主要用于在函数参数传递中，解决大块数据或对象的传递效率和空间不如意的问题。

　　（2）用引用传递函数的参数，能保证参数传递中不产生副本，提高传递的效率，且通过const的使用，保证了引用传递的安全性。

　　（3）引用与指针的区别是，指针通过某个指针变量指向一个对象后，对它所指向的变量间接操作。程序中使用指针，程序的可读性差；而引用本身就是目标变量的别名，对引用的操作就是对目标变量的操作。

　　（4）使用引用的时机。流操作符<<和>>、赋值操作符=的返回值、拷贝构造函数的参数、赋值操作符=的参数、其它情况都推荐使用引用。

const引用只能接收const变量。

非const变量能接收const引用或const变量。