C++ 值传递、指针传递、引用传递详解 - Geek_Ling - 博客园

https://www.cnblogs.com/yanlingyin/archive/2011/12/07/2278961.html

None Sat Nov, 21 01:08

最近写了几篇深层次讨论数组和指针的文章，其中提到了“C语言中，所有非数组的形式参数传递均以值传递形式”

而关于值传递，指针传递，引用传递这几个方面还会存在误区， 所有我觉的有必要在这里也说明一下~

下文会通过例子详细说明哦

值传递：

形参是实参的拷贝，改变形参的值并不会影响外部实参的值。从被调用函数的角度来说，值传递是单向的（实参->形参），参数的值只能传入，

不能传出。当函数内部需要修改参数，并且不希望这个改变影响调用者时，采用值传递。

指针传递：

形参为指向实参地址的指针，当对形参的指向操作时，就相当于对实参本身进行的操作

引用传递：

形参相当于是实参的“别名”，对形参的操作其实就是对实参的操作，在引用传递过程中，被调函数的形式参数虽然也作为局部变量在栈

中开辟了内存空间，但是这时存放的是由主调函数放进来的实参变量的地址。被调函数对形参的任何操作都被处理成间接寻址，即通过

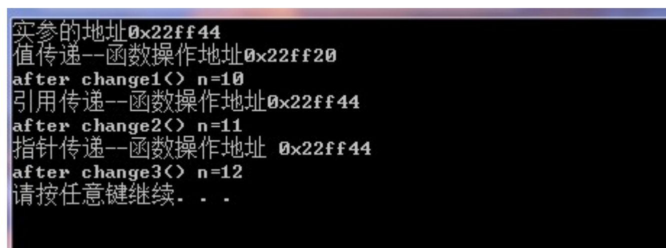
栈中存放的地址访问主调函数中的实参变量。正因为如此，被调函数对形参做的任何操作都影响了主调函数中的实参变量。

理论性的就不多说，

下面的代码对此作出了细致解释（从实参，形参在内存中存放地址的角度 说明了问题的本质，容易理解 ）

```
1 #include<iostream>
2 using namespace std;
3 //值传递
4  void change1(int n){
5     cout<<'值传递--函数操作地址'<<&n<<endl;         //显示的是拷贝的地址而不是源地址
6     n++;
7 }
8
9 //引用传递
10 void change2(int & n){
11     cout<<'引用传递--函数操作地址'<<&n<<endl;
12     n++;
13 }
14  //指针传递
15 void change3(int *n){
16      cout<<'指针传递--函数操作地址 '<<n<<endl;
17     *n=*n+1;
18  }
19 int     main(){
20     int n=10;
21     cout<<'实参的地址'<<&n<<endl;
22     change1(n);
23     cout<<'after change1() n='<<n<<endl;
24     change2(n);
25     cout<<'after change2() n='<<n<<endl;
26     change3(&n);
27     cout<<'after change3() n='<<n<<endl;
28     return true;
29 }
```

运行结果如下，（不同的机器可能会有所差别）



可以看出，实参的地址为0x22ff44

采用值传递的时候，函数操作的地址是0x22ff20并不是实参本身，所以对它进行操作并不能改变实参的值

再看引用传递，操作地址就是实参地址，只是相当于实参的一个别名，对它的操作就是对实参的操作

接下来是指针传递，也可发现操作地址是实参地址

那么，引用传递和指针传递有什么区别吗?

引用的规则：
（1）引用被创建的同时必须被初始化（指针则可以在任何时候被初始化）。

（2）不能有NULL引用，引用必须与合法的存储单元关联（指针则可以是NULL）。
（3）一旦引用被初始化，就不能改变引用的关系（指针则可以随时改变所指的对象）。

指针传递的实质：

指针传递参数本质上是值传递的方式，它所传递的是一个地址值。值传递过程中，被调函数的形式参数作为被调函数的局部变量处理，

即在栈中开辟了内存空间以存放由主调函数放进来的实参的值，从而成为了实参的一个副本。值传递的特点是被调函数对形式参数的

任何操作都是作为局部变量进行，不会影响主调函数的实参变量的值。（这里是在说实参指针本身的地址值不会变）如果理解不了大可跳过这段

指针传递和引用传递一般适用于:

函数内部修改参数并且希望改动影响调用者。对比指针/引用传递可以将改变由形参"传给"实参（实际上就是直接在实参的内存上修改，

不像值传递将实参的值拷贝到另外的内存地址中才修改）。

另外一种用法是：当一个函数实际需要返回多个值，而只能显式返回一个值时，可以将另外需要返回的变量以指针/引用传递

给函数，这样在函数内部修改并且返回后，调用者可以拿到被修改过后的变量，也相当于一个隐式的返回值传递吧。

以下是我觉得关于指针和引用写得很不错的文章，大家可参照看一下，原文出处地址: http://xinklabi.iteye.com/blog/653643

从概念上讲。指针从本质上讲就是存放变量地址的一个变量，在逻辑上是独立的，它可以被改变，包括其所指向的地址的改变和其指向的地址中所存放的数据的改变。

而引用是一个别名，它在逻辑上不是独立的，它的存在具有依附性，所以引用必须在一开始就被初始化，而且其引用的对象在其整个生命周期中是不能被改变的（自始至终只能依附于同一个变量）。

在C++中，指针和引用经常用于函数的参数传递，然而，指针传递参数和引用传递参数是有本质上的不同的：

指针传递参数本质上是值传递的方式，它所传递的是一个地址值。值传递过程中，被调函数的形式参数作为被调函数的局部变量处理，即在栈中开辟了内存空间以存放由主调函数放进来的实参的值，从而成为了实参的一个副本。值传递的特点是被调函数对形式参数的任何操作都是作为局部变量进行，不会影响主调函数的实参变量的值。（这里是在说实参指针本身的地址值不会变）

而在引用传递过程中，被调函数的形式参数虽然也作为局部变量在栈中开辟了内存空间，但是这时存放的是由主调函数放进来的实参变量的地址。被调函数对形参的任何操作都被处理成间接寻址，即通过栈中存放的地址访问主调函数中的实参变量。正因为如此，被调函数对形参做的任何操作都影响了主调函数中的实参变量。

引用传递和指针传递是不同的，虽然它们都是在被调函数栈空间上的一个局部变量，但是任何对于引用参数的处理都会通过一个间接寻址的方式操作到主调函数中的相关变量。而对于指针传递的参数，如果改变被调函数中的指针地址，它将影响不到主调函数的相关变量。如果想通过指针参数传递来改变主调函数中的相关变量，那就得使用指向指针的指针，或者指针引用。

为了进一步加深大家对指针和引用的区别，下面我从编译的角度来阐述它们之间的区别：

程序在编译时分别将指针和引用添加到符号表上，符号表上记录的是变量名及变量所对应地址。指针变量在符号表上对应的地址值为指针变量的地址值，而引用在符号表上对应的地址值为引用对象的地址值。符号表生成后就不会再改，因此指针可以改变其指向的对象（指针变量中的值可以改），而引用对象则不能修改。

最后，总结一下指针和引用的相同点和不同点：

★相同点：

●都是地址的概念；

指针指向一块内存，它的内容是所指内存的地址；而引用则是某块内存的别名。

★不同点：

●指针是一个实体，而引用仅是个别名；

- 引用只能在定义时被初始化一次，之后不可变；指针可变；引用“从一而终”，指针可以“见异思迁”；
- 引用没有const，指针有const，const的指针不可变；（具体指没有int& const a这种形式，而const int& a是有　的，　前者指引用本身即别名不可以改变，这是当然的，所以不需要这种形式，后者指引用所指的值不可以改变）
- 引用不能为空，指针可以为空；
- “sizeof 引用”得到的是所指向的变量(对象)的大小，而“sizeof 指针”得到的是指针本身的大小；
- 指针和引用的自增(++)运算意义不一样；
- 引用是类型安全的，而指针不是(引用比指针多了类型检查)

一、引用的概念
引用引入了对象的一个同义词。定义引用的表示方法与定义指针相似，只是用&代替了*。
例如：Point pt1(10,10);
Point &pt2=pt1; 定义了pt2为pt1的引用。通过这样的定义，pt1和pt2表示同一对象。
需要特别强调的是引用并不产生对象的副本，仅仅是对象的同义词。因此，当下面的语句执行后：
pt1.offset（2，2）；
pt1和pt2都具有（12，12）的值。
引用必须在定义时马上被初始化，因为它必须是某个东西的同义词。你不能先定义一个引用后才
初始化它。例如下面语句是非法的：
Point &pt3；
pt3=pt1；
那么既然引用只是某个东西的同义词，它有什么用途呢？
下面讨论引用的两个主要用途：作为函数参数以及从函数中返回左值。

二、引用参数

1、传递可变参数
传统的c中，函数在调用时参数是通过值来传递的，这就是说函数的参数不具备返回值的能力。
所以在传统的c中，如果需要函数的参数具有返回值的能力，往往是通过指针来实现的。比如，实现
两整数变量值交换的c程序如下：

```
void swapint(int *a,int *b)
{
int temp;
temp=*a;
a=*b;
*b=temp;
}
```

使用引用机制后，以上程序的c++版本为：

```
void swapint(int &a,int &b)
{
int temp;
temp=a;
a=b;
b=temp;
}
```

调用该函数的c++方法为：swapint（x,y); c++自动把x,y的地址作为参数传递给swapint函数。

2、给函数传递大型对象
当大型对象被传递给函数时，使用引用参数可使参数传递效率得到提高，因为引用并不产生对象的
副本，也就是参数传递时，对象无须复制。下面的例子定义了一个有限整数集合的类：

```
const maxCard=100;
Class Set
{
int elems[maxCard]; // 集和中的元素，maxCard 表示集合中元素个数的最大值。
int card; // 集合中元素的个数。
public:
Set () {card=0;} //构造函数
friend Set operator * (Set ,Set ) ; //重载运算符号*，用于计算集合的交集 用对象作为传值参数
// friend Set operator * (Set & ,Set & ) 重载运算符号*，用于计算集合的交集 用对象的引用作为传值参数
```

```
...
}
```
先考虑集合交集的实现
```
Set operator *( Set Set1,Set Set2)
{
Set res;
for(int i=0;i<Set1.card;++i)
for(int j=0;j>Set2.card;++j)
if(Set1.elems[i]==Set2.elems[j])
{
res.elems[res.card++]=Set1.elems[i];
break;
}
return res;
}
```
由于重载运算符不能对指针单独操作，我们必须把运算数声明为 Set 类型而不是 Set * 。
每次使用*做交集运算时，整个集合都被复制，这样效率很低。我们可以用引用来避免这种情况。
```
Set operator *( Set &Set1,Set &Set2)
{ Set res;
for(int i=0;i<Set1.card;++i)
for(int j=0;j>Set2.card;++j)
if(Set1.elems[i]==Set2.elems[j])
{
res.elems[res.card++]=Set1.elems[i];
break;
}
return res;
}
```

三、引用返回值

如果一个函数返回了引用，那么该函数的调用也可以被赋值。这里有一函数，它拥有两个引用参数并返回一个双精度数的引用：
```
double &max(double &d1,double &d2)
{
return d1>d2?d1:d2;
}
```
由于max()函数返回一个对双精度数的引用，那么我们就可以用max() 来对其中较大的双精度数加1：
```
max(x,y)+=1.0;
```

如有转载请注明出处：http://www.cnblogs.com/yanlingyin/

一条鱼@博客园

2011-12-7