C++中函数调用时的三种参数传递方式详解_这里记录着我一点一滴的进步-CSDN博客_c++参数传递

None Sat Nov, 21 01:04

原文地址：http://blog.csdn.net/cocohufei/article/details/6143476；

http://blog.chinaunix.net/uid-21411227-id-1826834.html

在C++中，参数传递的方式是“实虚结合”。

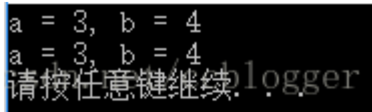
- 按值传递(pass by value)
- 地址传递(pass by pointer)
- 引用传递(pass by reference)

按值传递的过程为：首先计算出实参表达式的值，接着给对应的形参变量分配一个存储空间，该空间的大小等于该形参类型的，然后把以求出的实参表达式的值一一存入到形参变量分配的存储空间中，成为形参变量的初值，供被调用函数执行时使用。这种传递是把实参表达式的值传送给对应的形参变量，故称这种传递方式为“按值传递”。

使用这种方式，调用函数本身不对实参进行操作，也就是说，即使形参的值在函数中发生了变化，实参的值也完全不会受到影响，仍为调用前的值。

```
/*
  pass By value
*/
#include <iostream>
using namespace std;
void swap(int,int);
int main()
{
    int a = 3, b = 4;
    cout << 'a = ' << a << ', b = '
    << b << endl;
    swap(a,b);
    cout << 'a = ' << a << ', b = '
    << b << endl;
    return 0;
}
void swap(int x, int y)
{
     int t = x;
     x = y;
     y = t;
}
```

输出：



如果在函数定义时将形参说明成指针，对这样的函数进行调用时就需要指定地址值形式的实参。这时的参数传递方式就是地址传递方式。

地址传递与按值传递的不同在于，它把实参的存储地址传送给对应的形参，从而使得形参指针和实参指针指向同一个地址。因此，被调用函数中对形参指针所指向的地址中内容的任何改变都会影响到实参。

```
#include <iostream>

using namespace std;

void swap(int*, int*);
int main(){
        int a = 3, b = 4;
        cout << 'a=' << a << ', b=' << b << endl;
        swap(&a, &b);
        cout << 'a=' << a << ', b=' << b << endl;
        system('pause');
        return 0;
}

void swap(int *x, int *y){
        int t = *x;
        *x = *y;
        *y = t;
}
```

输出：

按值传递方式容易理解，但形参值的改变不能对实参产生影响。

地址传递方式虽然可以使得形参的改变对相应的实参有效，但如果在函数中反复利用指针进行间接访问，会使程序容易产生错误且难以阅读。

如果以引用为参数，则既可以使得对形参的任何操作都能改变相应的数据，又使得函数调用显得方便、自然。引用传递方式是在函数定义时在形参前面加上引用运算符“&”。

```
#include <iostream>

using namespace std;

void swap(int&, int&);
int main(){
        int a = 3, b = 4;
        cout << 'a=' << a << ', b=' << b << endl;
        swap(a, b);
        cout << 'a=' << a << ', b=' << b << endl;
        system('pause');
        return 0;
}

void swap(int &x, int &y){
        int t = x;
        x = y;
        y = t;
}
```

输出:

一、 函数参数传递机制的基本理论

函数参数传递机制问题在本质上是调用函数（过程）和被调用函数（过程）在调用发生时进行通信的方法问题。基本的参数传递机制有两种：值传递和引用传递。以下讨论称调用其他函数的函数为主调函数，被调用的函数为被调函数。

值传递（passl-by-value）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，**即在堆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。

引用传递(pass-by-reference)过程中，被调函数的形式参数虽然也作为局部变量在堆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，即通过堆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。

二、 C语言中的函数参数传递机制

在C语言中，值传递是唯一可用的参数传递机制。但是据笔者所知，由于受指针变量作为函数参数的影响，有许多朋友还认为这种情况是引用传递。这是错误的。请看下面的代码：

```
#include <iostream>

using namespace std;

int swap(int *x, int *y)
{
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
        return temp;
}

void main()
{


        int a = 1, b = 2;
        int *p1 = &a;
        int *p2 = &b;
        swap(p1, p2);
}
```

函数swap以两个指针变量作为参数，当main()调用swap时，是以值传递的方式将指针变量p1、p2的值（也就是变量a、b的地址）放在了swap在堆栈中为形式参数x、y开辟的内存单元中。

这里我们可以得到以下几点：

1. 进程的堆栈存储区是主调函数和被调函数进行通信的主要区域。

2. C语言中参数是从右向左进栈的。

3. 被调函数使用的堆栈区域结构为：

   局部变量（如temp）

   返回地址

   函数参数

   低地址

   高地址

4． 由主调函数在调用后清理堆栈。

5． 函数的返回值一般是放在寄存器中的。

这里尚需补充说明几点：一是参数进栈的方式。对于内部类型，由于编译器知道各类型变量使用的内存大小故直接使用push指令；对于自定义的类型（如structure），采用从源地址向目的（堆栈区）地址进行字节传送的方式入栈。**二是函数返回值为什么一般放在寄存器中**，这主要是为了支持中断；如果放在堆栈中有可能因为中断而被覆盖。**三是函数的返回值如果很大**，则从堆栈向存放返回值的地址单元（由主调函数在调用前将此地址压栈提供给被调函数）进行字节传送，以达到返回的目的。对于第二和第三点，《Thinking in C++》一书在第10章有比较好的阐述。四是一个显而易见的结论，如果在被调函数中返回局部变量的地址是毫无意义的；因为局部变量存于堆栈中，调用结束后堆栈将被清理，这些地址就变得无效了。

三、 C++语言中的函数参数传递机制

众所周知，在c＋＋中调用函数时有三种参数传递方式:

（1）传值调用;

（2）传址调用（传指针）；

（3）引用传递;

实际上，还有一种参数传递方式，就是全局变量传递方式。这里的“全局”变量并不见得就是真正的全局的，所有代码都可以直接访问的，只要这个变量的作用域足够这两个函数访问就可以了，比如一个类中的两个成员函数可以使用一个成员变量实现参数传递，或者使用static关键字定义，或者使用namespace进行限制等，而这里的成员变量在这种意义上就可以称作是“全局”变量（暂时还没有其它比“全局”更好的词来描述）。当然，可以使用一个类外的真正的全局变量来实现参数传递，但有时并没有必要，从工程上讲，作用域越小越好。这种方式有什么优点呢?

效率高!

的确，这种效率是所有参数传递方式中效率最高的，比前面三种方式都要高，无论在什么情况下。但这种方式有一个致命的弱点，那就是对多线程的支持不好，如果两个进程同时调用同一个函数，而通过全局变量进行传递参数，该函数就不能够总是得到想要的结果。

下面再分别讨论上面三种函数传递方式。

1. 从功能上。按值传递在传递的时候，实参被复制了一份，然后在函数体内使用，**函数体内修改参数变量时修改的是实参的一份拷贝，**而实参本身是没有改变的，所以如果想在调用的函数中修改实参的值，使用值传递是不能达到目的的，这时只能使用引用或指针传递。例如，要实现两个数值交换。

```
void swap(int a  int b)

void main(){

    int a=1  b=2

    swap(a b)

}
```

这样，在main()函数中的a b值实际上并没有交换，如果想要交换只能使用指针传递或引用传递，如：

```
void swap(int *a  int *b)
```

或

```
void swap(int &a  int &b)
```

2.从传递效率上。这里所说传递效率，是说调用被调函数的代码将实参传递到被调函数体内的过程，正如上面代码中，这个过程就是函数main()中的a b传递到函数swap()中的过程。这个效率不能一概而论。对于内建的int char   short long float等4字节或以下的数据类型而言，实际上传递时也只需要传递1－4个字节，而使用指针传递时在32位cpu中传递的是32位的指针，4个字节，都是一条指令，这种情况下值传递和指针传递的效率是一样的，而传递double long long等8字节的数据时，在32位cpu中，其传值效率比传递指针要慢，因为8个字节需要2次取完。而在64位的cpu上，传值和传址的效率是一样的。再说引用传递，这个要看编译器具体实现，引用传递最显然的实现方式是使用指针，这种情况下与指针的效率是一样的，而有些情况下编译器是可以优化的，采用直接寻址的方式，这种情况下，效率比传值调用和传址调用都要快，与上面说的采用全局变量方式传递的效率相当。

再说自定义的数据类型，class struct定义的数据类型。这些数据类型在进行传值调用时生成临时对象会执行构造函数，而且当临时对象销毁时会执行析构函数，如果构造函数和析构函数执行的任务比较多，或者传递的对象尺寸比较大，那么传值调用的消耗就比较大。这种情况下，采用传址调用和采用传引用调用的效率大多数下相当，正如上面所说，某些情况下引用传递可能被优化，总体效率稍高于传址调用。

3. 从执行效率上讲。这里所说的执行效率，是指在被调用的函数体内执行时的效率。因为传值调用时，当值被传到函数体内，临时对象生成以后，所有的执行任务都是通过直接寻址的方式执行的，而指针和大多数情况下的引用则是以间接寻址的方式执行的，所以实际的执行效率会比传值调用要低。如果函数体内对参数传过来的变量进行操作比较频繁，执行总次数又多的情况下，传址调用和大多数情况下的引用参数传递会造成比较明显的执行效率损失。

综合2、3两种情况，具体的执行效率要结合实际情况，通过比较传递过程的资源消耗和执行函数体消耗之和来选择哪种情况比较合适。**而就引用传递和指针传递的效率上比，引用传递的效率始终不低于指针传递，所以从这种意义上讲，在c＋＋中进行参数传递时优先使用引用传递而不是指针。**

4. 从类型安全上讲。值传递与引用传递在参数传递过程中都执行强类型检查，而指针传递的类型检查较弱，特别地，如果参数被声明为 void ，那么它基本上没有类型检查，只要是指针，编译器就认为是合法的，所以这给bug的产生制造了机会，使程序的健壮性稍差，如果没有必要，就使用值传递和引用传递，最好不用指针传递，更好地利用编译器的类型检查，使得我们有更少的出错机会，以增加代码的健壮性。

这里有个特殊情况，就是对于多态的情况，如果形参是父类，而实参是子类，在进行值传递的时候，临时对象构造时只会构造父类的部分，是一个纯粹的父类对象，而不会构造子类的任何特有的部分，因为办有虚的析构函数，而没有虚的构造函数，这一点是要注意的。如果想在被调函数中通过调用虚函数获得一些子类特有的行为，这是不能实现的。

5. 从参数检查上讲。一个健壮的函数，总会对传递来的参数进行参数检查，保证输入数据的合法性，以防止对数据的破坏并且更好地控制程序按期望的方向运行，在这种情况下使用值传递比使用指针传递要安全得多，因为你不可能传一个不存在的值给值参数或引用参数，而使用指针就可能，很可能传来的是一个非法的地址（没有初始化，指向已经delete掉的对象的指针等）。所以使用值传递和引用传递会使你的代码更健壮，具体是使用引用还是使用，**最简单的一个原则就是看传递的是不是内建的数据类型，对内建的数据类型优先使用值传递，而对于自定义的数据类型，特别是传递较大的对象，那么请使用引用传递。**

6. 从灵活性上。无疑，指针是最灵活的，因为指针除了可以像值传递和引用传递那样传递一个特定类型的对象外，还可以传递空指针，不传递任何对象。指针的这种优点使它大有用武之地，比如标准库里的time( )函数，你可以传递一个指针给它，把时间值填到指定的地址，你也可以传递一个空指针而只要返回值。

以上讨论了四种参数传递方式的优缺点，下面再讨论一下在参数传递过程中一些共同的有用的技术。

1. const关键字。当你的参数是作为输入参数时，你总不希望你的输入参数被修改，否则有可能产生逻辑错误，这时可以在声明函数时在参数前加上const关键字，防止在实现时意外修改函数输入，对于使用你的代码的程序员也可以告诉他们这个参数是输入，而不加const关键字的参数也可能是输出。例如strlen，你可以这样声明

int strlen(char str)

功能上肯定没有什么问题，但是你想告诉使用该函数的人，参数str是一个输入参数，它指向的数据是不能被修改的，这也是他们期望的，总不会有人希望在请人给他数钱的时候，里面有张100的变成10块的了，或者真钞变成假钞了，他们希望有一个保证，说该函数不会破坏你的任何数据，声明按如下方式便可让他们放心：

int strlen(const char str)

可不可以给str本身也加一个限制呢，如果把地址改了数得的结果不就错了吗？总得给人点儿自由吧，只要它帮你数钱就行了，何必介意他怎么数呢？只要不破坏你的钱就ok了，如果给str一个限制，就会出现问题了，按照上面的声明，可以这样实现：

```
int strlen(const char str){
        int cnt;
        if(!str)
                return 0;
        cnt = 0;
        while((str++)){
                ++cnt;
        }
        return cnt;

}
```

可是，如果你硬要把声明改成

int strlen(const char const str)

上面的函数肯定就运行不了了，只能改用其它的实现方式，但这个不是太有必要。只要我们保护好我们的钱就行了，如果它数不对，下次我次不让它数，再换个人就是了。

对于成员函数，如果我们要显示给客户代码说某个成员函数不会修改该对象的值，只会读取某些内容，也可以在该函数声明中加一个const.

class　person

{......

public:

unsigned char age(void) const // 看到const就放心了，这个函数肯定不会修改m_age

private:

unsigned char m_age // 我认为这个类型已经足够长了，如果觉得不改可以改为unsigned long

}

2. 默认值。个人认为给参数添加一个默认值是一个很方便的特性，非常好用，这样你就可以定义一个具有好几个参数的函数，然后给那些不常用的参数一些默认值，客户代码如果认为那些默认值正是他们想要的，调用函数时只需要填一些必要的实参就行了，非常方便，这样就省去了重载好几个函数的麻烦。可是我不明白c#为什么把这个特性给去掉了，可能是为了安全，这样就要求每次调用函数时都要显示地给函数赋实参。所以要注意，这可是个双刃剑，如果想用使刀的招跟对手武斗，很可能伤到自己。

3.参数顺序。当同个函数名有不同参数时，如果有相同的参数尽量要把参数放在同一位置上，以方便客户端代码。

c++ 中经常使用的是常量引用，如将swap2改为：

Swap2(const int& x; const int& y)

这时将不能在函数中修改引用地址所指向的内容，具体来说，x和y将不能出现在'='的左边。