指针和引用

指针和引用的作用

教材在讲解指针概念时，一般会使用`swap`函数作为例子。我不太喜欢这个例子，因为在现实的编程环境中，类似`swap`的情况是很少的。这个例子不仅需要花一点时间去解释，而且回避了指针和引用的真正作用。我在上课时更喜欢用两个数相除的例子。

假设写一个函数实现`a/b`，一般的写法如下：

```
float div(float a,float b)
{
    return a/b;
}
```

这个写法显然是错误的。为什么呢？因为没有考虑`b=0`的情况。如果`b=0`，应该通过某个返回值告知除0错误。那么怎么做呢？很显然，上面这个函数满足不了这个要求，而且下面的函数可以：

```
bool div(float a,float b,float* c)
{
    if(b==0)
        return false;
    *c=a/b;
    return true;
}
```

或者

```
bool div(float a,float b,float& c)
{
    if(b==0)
        return false;
    c=a/b;
    return true;
}
```

因此我们可以总结出指针和引用的第一个用途：`当函数需要用参数返回结果时，需要使用指针`。

指针和引用还有一个用途，先看看下面的代码：

```
#include<iostream>
#include<memory>
#include<vector>
using namespace std;
void doNothing(vector<int> b)
{
}
int main()
{
        vector<int> a(10000000000);
        doNothing(a);
        return 0;
}
```

当上述程序执行到`vector<int> a(10000000000)`时，内存消耗约5.48GB。

进程名称	内存	压缩后的内存	线程	端口	PID	用户
a.out	5.48 GB	3.35 GB	1	12	23636	liuji
Xcode	2.26 GB	2.07 GB	15	597	6775	liuji

执行到`doNothing(a)`时，内存消耗约为10.61GB。

进程名称	内存	压缩后的内存	线程	端口	PID	用户
a.out	10.61 GB	8.44 GB	1	12	23636	liuji
Xcode	2.26 GB	2.07 GB	13	593	6775	liuji

可见`doNothing`并非什么都不做。它做的事情是将实参`a`拷贝到形参`b`，因此内存增加了一倍。如果只是为了传递参数，这种拷贝是毫无价值的，不仅会增加内存的使用，而且会降低运行速度。解决此问题的方法就是使用指针和引用。

```
#include<iostream>
#include<memory>
#include<vector>
using namespace std;
void doNothing(vector<int>* b)
{
}
int main()
{
        vector<int> a(10000000000);
        doNothing(a);
        return 0;
}
```

或者

```
#include<iostream>
#include<memory>
#include<vector>
using namespace std;
void doNothing(vector<int>& b)
{
}
int main()
{
        vector<int> a(10000000000);
        doNothing(a);
        return 0;
}
```

因此指针和引用的第二个用途就是：`大数据的参数传递`。

总结一下指针和引用的用途：

1. 参数返回结果
2. 参数传递

Java引用与C++引用的区别

Java的引用是`命名上的错误`，尽管叫做`引用`，但是它的实质是C++中的`指针`，只是不允许做`++`、`--`和`*`。我们来看看以下的C++代码（此代码只是为了示例，写法可不能学）：

```
void func(Student&s)
{
    Student *ps=new Student();
    s=*ps;
}
```

这段代码用`s`返回了指向局部对象`*ps`的引用。能够用Java来实现吗？有人认为可以这么写：

```
void func(Student s) //Java的引用
{
    s=new Student();
}
```

但这段代码达不到C++代码的效果，因为它和下面的C++代码等价：

```
void func(Student* s) //C++的指针
{
    s=new Student();
}
```

不过C++的指针可以这么写：

```
void func(Student* s) //C++的指针
{
    *s=Student();
}
```

这样可以用指针返回局部对象的`内容`。同样的代码Java可以吗？不行，真的不行，因为Java没有`*`运算。Java要实现同样的效果必须这么写：

```
void func(Student s) //Java的指针
{
    Student a=Student();
    //从a向s拷贝内容
    s.setName(a.getName());
    s.setAge(a.getAge());
}
```

记住Java的引用是`没有++、--和*运算符的指针`。

const和final的区别

C++中用const修饰指针，Java中用final修饰引用，二者的效果是完全不同的。先看C++的代码：

```
int a=0;
int b=1;
const int *p=&a;
p=&b; //ok!
*p=10; //error!
```

`const int *p`表示`*p`的内容是不能更改的，但是`p`是可以更改的。

再来看看Java的代码：

```
final int[] p=new int[10];
p[0]=1; //ok!
p=new int[10]; //error
```

在Java中`p所指对象`的内容是可以更改的（相当于C++的`*p`可更改），而`p`的值是不能更改的。换句话说，恰好和C++的含义相反。如果在C++中声明为`int * const p`，则效果与Java相同，即：

```
int a=0;
int b=1;
int *const p=&a;
*p=10; //ok!
p=&b; //error!
```

所以`Java的final相当于C++的const修饰p`。

当然C++的声明还可以是`const int * const p`这表示`p不能修改，*p也不能修改`。

智能指针

C++11开始智能指针成为了标准库的一部分，只要引入了`memory`头文件，就可以使用`unique_ptr`、`shared_ptr`和`weak_ptr`。`shared_ptr`是智能指针中最常见的，使用它可以为指针加上引用计数，从而解决`new了之后忘记delete`的问题。所以一旦使用`shared_ptr`，就可以取得Java的垃圾回收类似的效果，并且在性能上的代价要低得多。`unique_ptr`和`weak_ptr`用于某些特定场景，这里就只讲解`shared_ptr`。

`shared_ptr`的使用场景如下：

```
while(true)
{
    int *p=new int[10];
}
```

此代码存在内存泄漏，很快内存就会被耗尽。如果使用`shared_ptr`就可以解决这个问题。

```
while(true)
{
    shared_ptr<int> p(new int[10]);
}
```

当然`p`的类型实际上是`shared_ptr<int>`而不是`int*`，故此不能对`p`做`p[0]`操作，或者是`++`、`--`。如果要访问`p`的内容，可以这么做：

```
shared_ptr<int> p(new int[10]);
p.get()[0]=1;
p.get()[2]=2;
cout<<p.get()[0]<<endl;
cout<<p.get()[2]<<endl;
```

只要是使用了`shared_ptr`就不用担心`new int[10]`没有delete啦。更多的情况下，`shared_ptr`是这么用的：

```
struct Point{
    float x;
    float y;
};
shared_ptr<Point> p1(new Point());
p1->x=10;
p1->y=20;
shared_ptr<Point> p2=p1; //增加引用计数,p2和p1指向相同的对象
p2->x=30;
cout<<p1->x<<" "<<p1->y<<endl; //输出30 20
cout<<p2->x<<" "<<p2->y<<endl; //输出30 20
Point p3;
p3.x=50;
p3.y=60;
*p1=p3; //p3拷贝到*p1，p2和p1指向相同对象，故此*p2和*p1相同
cout<<p1->x<<" "<<p1->y<<endl; //输出50 60
cout<<p2->x<<" "<<p2->y<<endl; //输出50 60
shared_ptr<Point> p4(new Point());
p4->x=70;
p4->y=80;
p2=p4; //p2指向p4,p2和p1现在指向不同的对象
cout<<p1->x<<" "<<p1->y<<endl; //输出50 60
cout<<p2->x<<" "<<p2->y<<endl; //输出70 80
```

如果担心内存泄漏，所有涉及指针的地方都可以使用`shared_ptr`。`shared_ptr`导致的性能消耗，多数情况下可以忽略不计。

`shared_ptr与Java的引用的用法基本相同`，但是`Java没有*运算`，因此`要注意*p=和p=的含义是完全不同的`。下面举例，先看`p=`：

```
shared_ptr<Point> p1(new Point());
p1->x=10;
p1->y=20;
shared_ptr<Point> p2=p1; //p2和p1指向相同对象，内容相同
shared_ptr<Point> p3(new Point());
p3->x=70;
p3->y=80;
p2=p3;  //p2和p3指向相同对象，p1和p2内容不同
cout<<p1->x<<" "<<p1->y<<endl; //输出10 20
cout<<p2->x<<" "<<p2->y<<endl; //输出70 80
```

再看`*p=`：

```
shared_ptr<Point> p1(new Point());
p1->x=10;
p1->y=20;
shared_ptr<Point> p2=p1; //p2和p1指向相同对象，内容相同
shared_ptr<Point> p3(new Point());
p3->x=70;
p3->y=80;
*p2=*p3;  //p3的内容拷贝给p2,p1、p2和p3的内容相同
cout<<p1->x<<" "<<p1->y<<endl; //输出70 80
cout<<p2->x<<" "<<p2->y<<endl; //输出70 80
cout<<p3->x<<" "<<p3->y<<endl; //输出70 80
p3->x=100; //p3和p2、p1指向的地址不同，因此修改p3不会反应到p2和p1上
cout<<p1->x<<" "<<p1->y<<endl; //输出70 80
cout<<p2->x<<" "<<p2->y<<endl; //输出70 80
cout<<p3->x<<" "<<p3->y<<endl; //输出70 100
```

这么看就有些复杂啦。

总结一下，`使用shared_ptr只要不用*p=，可以如同Java的引用一样使用`。从这里可以看出，Java的引用设计还是有很多思考的——既兼顾性能也考虑了技术的复杂性。但无论如何`shared_ptr`的引入简化了C++的内存管理，只要在使用上稍加注意（为什么C++的所有技术都有这种附加条件……）。