

# 实践7 小结

**指针，指针，还是指针！！！！**

- 任何对堆(heap)地址的访问(读、写、释放)，都以**地址有效**为基本前提。程序没有对地址有效性的判断——编译正确，运行也正确——只是你的测试范围过小。
- 栈(stack)地址的错误，会产生逻辑错误(有越界写的时候也可能产生程序崩溃现象)。而堆(heap)内地址非法访问(读、写、释放)，程序一定会异常终止。
- 通常我们使用栈内**指针变量**存储堆(heap)地址。防止**指针变量**在运行过程中被错误覆盖(如数组越界等)，是C/C++程序员的基本素养。

**那些看起来是冗余的代码，是必不可少的！！！！**

## 实践7 小结

- 指针变量，一段内存区间的首地址，这段区间可能在**栈**内(变量)，可能在**堆**内(动态申请)
  - delete操作——指针指向**堆**内动态申请空间的地址，动态申请的空间，**需要且仅能释放一次**。
  - **读**操作——指针指向**有效**地址 “**先判断后使用**”
  - **写**操作——指针指向**有效**地址 “**先判断后使用**”写入的数据量应少于有效区间可容纳的数量。

## 实践7 小结

- 向函数内传地址时，尽可能使用引用，不是必需不使用指针：

- **&**在声明时和使用时的区别：不易产生致命错误，编译器能检出大部分的误用。
- **\***在声明时和使用时的区别：

```
char *pA=NULL;  if(pA) cout<<pA;
```

```
pA=NULL;  //指针变量pA存储的地址值清零。
```


```
*pA=NULL; //指向地址的内容被置零。
```

```
//例如当pA=&a时，a被置零。
```

# 浅复制

默认复制构造  
默认=  
完成成员变量  
的简单复制,如

```
class A
{
    double *pd;
    public:
    A(const A &oa){pd=oa.pd;}
};
```



Tips: 当仅声明一个复制构造函数时, 默认无参构造函数也被屏蔽。上例类A无法构造对象。

## 深复制典型错误：

```
class A{
    double *pD;
public:
    A(double *p=NULL)
    {
        if(p)
        {
            if(pD=new double)
                *pD=*p; //写入指向空间的内容
                //经典错误： pD=p 编译器忠实执行程序员的指令。
            }
            else pD=NULL; //必不可少，否则无参构造时pD指向无效地址。
        }
        ~A(){delete pD;}
    };
};
```

测试： `void main() { A a1; return; }`

## 深复制典型错误：

```
class A { .....//接前页
    A(const A &oa)
    {
        if(oa.pD)
        {
            if(pD=new double)
                *pD=*oa.pD; //写入指向空间的内容
                //经典错误： pD=oa.pD 依旧是浅复制
        }
        else pD=NULL; //必不可少，否则pD指向未初始化的无效地址。
    }
};
```

测试：            `void main() { A a, b(a); return; }`  
步步为营！

```
MyString MyString::operator+ (const MyString &s2) const{
```

```
    MyString temp;    //无参构造
```

```
    // 无字符时设置last=0, str=NULL
```

```
    temp.last=last+s2.last;
```

```
    temp.str=new char[temp.last+1]; ←
```

```
    for(int i=0; i<last; i++) {
```

```
        temp.str[i]=str[i]; }
```

//数组访问法

```
    for(int j=0; j<s2.last; j++) {
```

```
        *(temp.str+last+j)=*(s2.str+j); } //指针移位法
```

```
    temp.str[temp.last]='\0'; //不可漏
```

```
    return temp; ←
```

```
}
```

测试:

S1+S2;

S3=S1+S2;

/\*\*/

申请的内存将在析构中释放  
若用 strlen(str)+strlen(s2.str)+1  
需要“先判断, 后使用”  
故用last域更简便

/\*\*/

/\*\*返回时涉及:

1.深复制构造函数是否定义完备

2.析构函数是否完备

3.深赋值=重载是否完备(返回主调函数后赋值)

\*\*\*\*\*/

```
istream& operator>> (istream& is, MyString& ms)
```

```
{
```

```
    delete [] ms.str;
```

```
    char c[100];
```

```
    is.getline(c,100); //可接收空格
```

```
    if(ms.str=new char[strlen(c)+1])
```

```
        strcpy(ms.str,c);
```

```
    else {
```

```
        ms.str=NULL;
```

```
        ms.last=0; //配合+的实现
```

```
    }
```

```
    return in;
```

```
}
```

前提是指针的规范使用。  
无内容则置零，  
有内容则指向有效地址。

依旧有 is>>c

依旧缺对last的操作



## 关于[]的重载

```
string s1("abc"); //测试string类的[]运算符
cout<<s1[2]<<endl;
s1[2]='d'; cout<<s1<<endl; //可修改
s1[3]='d'; cout<<s1<<endl; //将产生越界读
s1[5]='d'; cout<<s1<<endl; //将产生越界写
/*****
```

证明string类没有扩容，也没作保护措施，仅单纯返回`str[i]`，下标大于`last`时将产生非法地址访问。你定义的MyString可以做得比标准string类更好。

```
*****/
```

```
student::student(const char *i,string n, MyString a, const double *s, int sn)
{
    name=n;           //完备定义的string类对象成员
    address=a;        //完备定义的MyString类对象成员

    /**静态数组**/
    if(i) strcpy(id,i); //前提是i内存储的字符个数少于9个

    /**动态数组**/
    score=NULL;  scoreNum=0;
    if(s && sn)
    {
        if(score=new double[sn])
        {
            scoreNum=sn;
            for(int i=0; i<sn; i++)
                score[i]=s[i];
        }
    }
}
```

```

istream& operator>> (istream& is, student& s)
{
    is >> s.name;           //完备定义的string类对象成员
    is >> s.address;        //完备定义的MyString类对象成员

    /**静态数组**/
    is >> s.id; //前提是标准输入支持字符数组整串输入
    //若是int或double数组，则与score类似逐个元素输入

    /**动态数组**/
    delete [] s.score;      ← 前提是指针的规范使用。
    is >> s.scoreNum;        无内容则置零，
                            有内容则指向有效地址。
    if(s.score=new double[s.scoreNum]){
        for(int i=0; i<s.scoreNum; i++)
            is >> s.score[i];
    }
    return is;
}

```

输入，对客户的要求！！  
啊！！

# 实践7 小结

- 对堆(heap)地址的访问，地址有效的含义：
  - 释放——地址有效！
  - 读操作——地址有效！
  - 写操作——地址有效！ 空间足够！！！！例如：拷贝，=，+，+=，>> 等
- 栈(stack)空间的分配释放由系统管理，堆(heap)内申请的空间，不释放则长期有效——C程序员的职责。
  - new delete 必须成对使用！ 逻辑上的成对，运行中的成对。
  - 物理上 new、delete可分布在不同模块，并且通常不是一对一的。

在0:1的程序世界里，坚定一个信念：  
错误是确定一定以及肯定可以排除掉的；

# 实践7 小结

- 函数返回引用/指针的本质都是返回地址

(作用域和生命期有效的地址)

- 定义全局变量，返回该全局变量的地址。

(语法正确，地址有效，破坏封装性)

- 函数内定义局部静态变量，返回该局部变量的地址。

(语法正确，地址有效，但逻辑(作用域)模糊，无内存反复消耗)

- 函数内动态申请堆空间的地址。如: localtime()函数

(语法正确，地址有效，逻辑上同局部静态变量，但内存消耗反复，内存泄漏)

- 将函数调用处有效的地址(栈或堆)通过形参(引用或指针)传入函数。如: <<和>>的重载，例4.5， localtime\_s()函数

- 函数返回引用/指针在形式上的一点区别：引用返回必须显式地返回，指针不必需显式地返回。

# 实践8：链表及应用

- 要求

- 理解线性表的**链表**的构成。
- 掌握链表的算法编程与应用，包括**构造**链表、**查找**结点、**插入**结点、**删除**结点和对链表**排序**等基本操作。

- 编程：

- 1. 习题7.3-6，改造**完善单链表类模板**。
- 2. 习题7.7，有序双向链表合并。

【具体要求和实现提示见cpp，求稳固基础掌握的完成习题7.3、7.4，学有余力求提高熟练使用指针的同学**继续**完成7.5、7.6、7.7】

# 实践8

- 本次作业所有改造都是针对链表类模板的。
- 在 `class List` 中添加成员函数;

- 习题7.3, 与已有成员函数的异同

`bool DeleteNode(T data);` // (重载) 删除指定值结点。

`Node<T>* GetNodeK(int K) const;` // 取第K个元素, 重载find也可, 返回类型声明为T也可。

- 习题7.4, 二者的异同

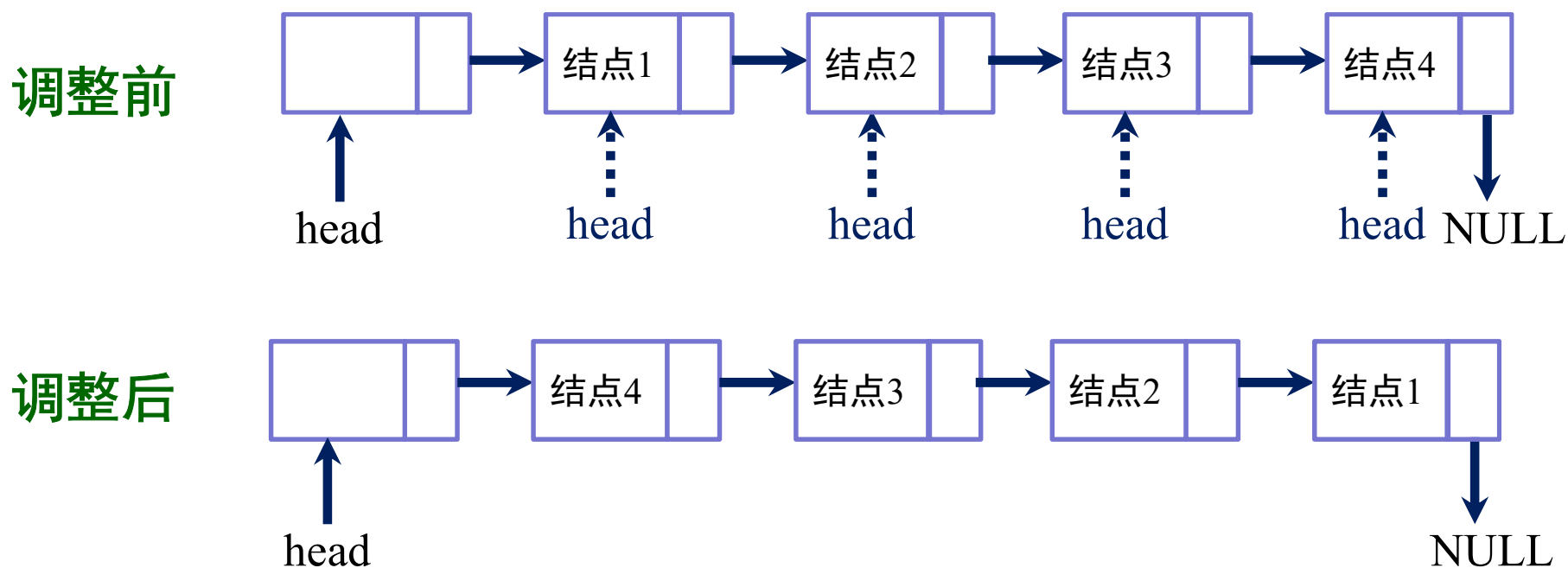
`List(const List<T> &);` // 复制构造, 创建一个新链表

`List<T> & operator=(const List<T> &);` // 赋值, 修改一个已有链表

# 实践8

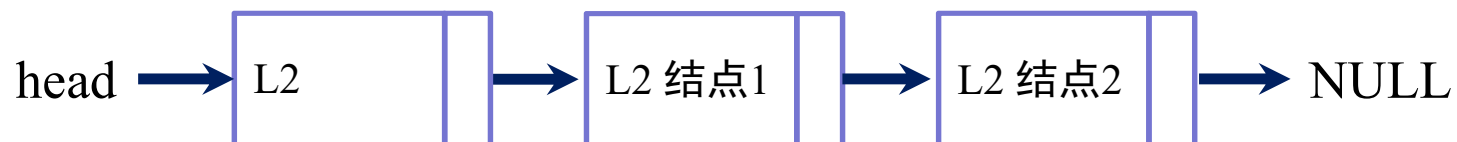
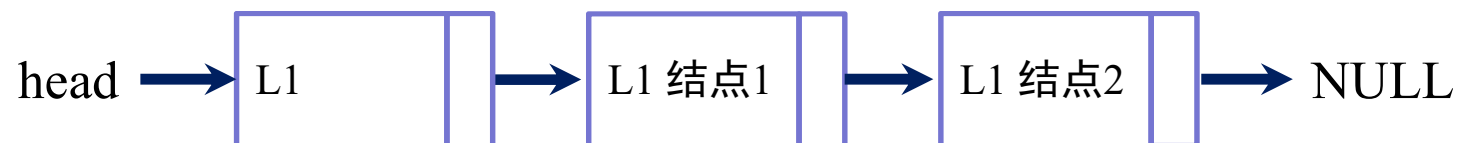
- 在 `class List` 中添加成员函数；
  - 习题7.5，难点在不改变所有结点的存储，仅通过**指针域连接**的改变。

`void Reverse();` //通过指针域连接的改变，将链表逆转。

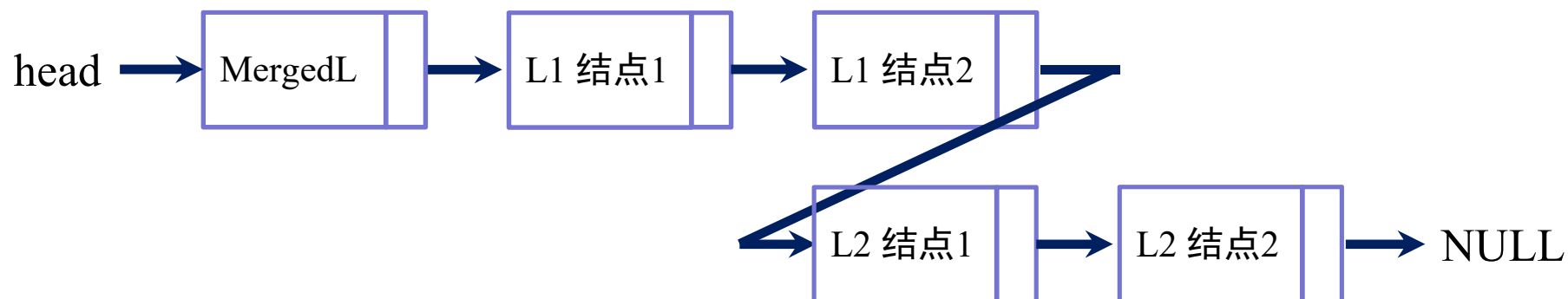




# 实践8



Merge



**注意：**重载`operator+`函数，返回的链表的每一个结点，都应该重新在堆内申请空间；不应该是取L1或L2中的结点地址赋值给objL结点。原理同深浅复制/赋值。

# 实践8

- 在class List中添加成员函数;
  - 习题7.6, 难点在去除数据重复的结点, 两个思路:
    - 1) 有序: 先排序, 然后边归并边去重
    - 2) 无序: 先归并, 后遍历去重

List<T> **operator+**(const List<T> &); //以“深复制构造”或“深赋值=”的重载为前提。

List<T>& **merge**(List<T> &objL, const List<T> &L1, const List<T> &L2);

若用普通函数实现merge函数, 结点类需添加辅助函数:

T GetInfo(return info); Node<T>\* GetLink(){return link;}

- 没有重载从结点(甚至T)构造链表, 友元的意义不大, 可用成员函数实现+重载, 但是会涉及整个链表的拷贝构造。
- 有兴趣有余力的同学, 可自定义从结点类型或T类型构造链表的(类型转换)构造函数, 实现并测试用友元(普通)函数重载的+运算符。

# 实践8

- 在**class DbList**中添加成员函数；
  - 习题7.7，两个有序双向链表的合并，可以定义为普通函数或友元函数。

**DbList<T>& merge(DbList<T> &objL,**  
**const DbList<T> &L1, const DbList<T> &L2);**

注意：这一题不要做成了类似于+=，把L2拼接到此List上。

提醒乙方——请注意甲方的需求。

这个问题在以前的类定义的+重载中也有出现。

若定义普通函数实现，结点类需添加辅助函数：

```
T GetInfo(return info);  
DbListNode<T>* GetlLink(){return llink;}  
DbListNode<T>* GetrLink(){return rlink;}
```