



东南大学吴健雄学院
CHIEN-SHIUNG WU COLLEGE OF SEU

C++程序设计

——动态内存分配



内容

指针与自由存储区的使用

链表与单链表模板类

栈与队列及其模板类

指针与自由存储区的使用

堆区、栈区空间的分配与释放

栈区 —— 系统自动分配与撤销

堆区 —— 编程者显式分配与撤销

动态空间操作需借助指针

new运算申请空间: `int* p = new int;` //申请一个整数单元

delete运算撤销空间: `delete p;` //撤销整数单元, 不是撤销指针单元





再对比几例

<code>double *pd = new double(5.6);</code>	<code>//申请实数空间并初始化为5.6</code>
<code>cout << *pd << endl;</code>	<code>//动态数据只能用指针访问</code>
<code>delete pd;</code>	<code>//撤销实数空间</code>
<code>class Cgoods { /*成员略*/ };</code>	
<code>Cgoods *pCg = new Cgoods(...);</code>	<code>//调用构造函数创建对象</code>
<code>pCg -> print();</code>	
<code>delete pCg;</code>	<code>//撤销动态对象空间</code>



```
double *parr = new double [20];           //申请一个数组空间
for (int i = 0; i < n; i++)  cin >> parr[i];  //读入元素值
delete [ ]parr;      //撤销数组[]必不可少，否则只撤销第一个元素
```

//可否申请长度可变的数组？如下：

```
int n;  cin >> n;           //一个变量
double *parr = new double [n];      //可以吗?
```

√

动态变量的生命期谁决定？ ——程序员用new和delete决定

几个重要问题

1. 动态内存分配失败

申请动态空间正常时返回对象地址；异常怎么办

```
Cgoods *pCg = new Cgoods[1000000](...);  
if ( !pCg )  
{   cout << "内存分配失败！" ;   exit(1); }
```

2. 内存泄漏 (memory leak)

若动态空间的指针丢失，则空间无法撤销，称内存泄漏。空间无法撤销直至系统重启。

```
int *p = new int (3);  
p = new int (5);      //前一个动态空间泄漏
```

新标准采用异常处理机制：分配失败抛出异常

```
try {  
    Cgoods *pCg = new goods[1000000](...);  
}  
catch (const bad_alloc& e) {  
    cerr << "内存分配失败：" << e.what()  
    << endl;  
}
```

如果不处理异常，则由terminate()终止程序。



3. 内存重复释放 —— 会引发内存错误

```
int *p1 = new int (3)
```

```
int *p2 = p1;    //指向同一单元
```

```
delete p1;       //p1仍指向该单元，但单元被标记撤销
```

```
delete p2;       //重复释放，导致出错
```

4. 关于动态数组

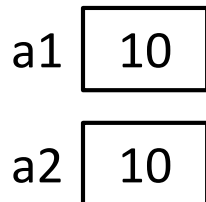
数组长度可动态确定，但不能初始化数据。

含指针成员对象的深拷贝和浅拷贝

回顾命题：封装类时一般不需要定义拷贝构造函数和析构函数，也不重载=运算符，除非对象中含有指针成员。为什么？对比：

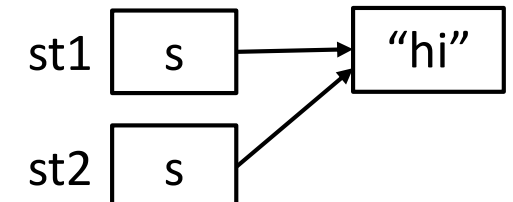
```
class A
{   int x;
public:
    A(int x0 = 0) : x(x0) {}
};

void main()
{   A a1(10), a2(a1); }
```



```
class mystring
{   char *s;
public:
    mystring(char *p = 0) : s(p) {}
};

void main()
{   mystring st1("hi"), st2(st1); }
```



缺省拷贝：导致对象不独立
——浅拷贝

对象含有指针成员时，缺省拷贝导致对象不独立，称为**浅拷贝**。

对象不独立有什么问题？

```
class mystring
{   char *s;
public:
    mystring(char * = 0);
    ~mystring();
};

void main()
{   mystring st1("hi"), s2(s1); }
```

```
mystring :: mystring(char * p)
{   s = new char[100]; //开辟自己的地盘
    strcpy (s, p);
}

mystring :: ~mystring()
{   if (s)
    {   delete []s;   s = 0;   }
}
```

含指针成员往往涉及动态内存，不析构则泄漏，析构则重复释放。

深拷贝 —— 让含有指针成员的对象相互独立

关键点：避免对象的指针直接赋值

- 1、为指针成员申请独立空间；
- 2、实现数据拷贝

显式定义拷贝构造函数内容

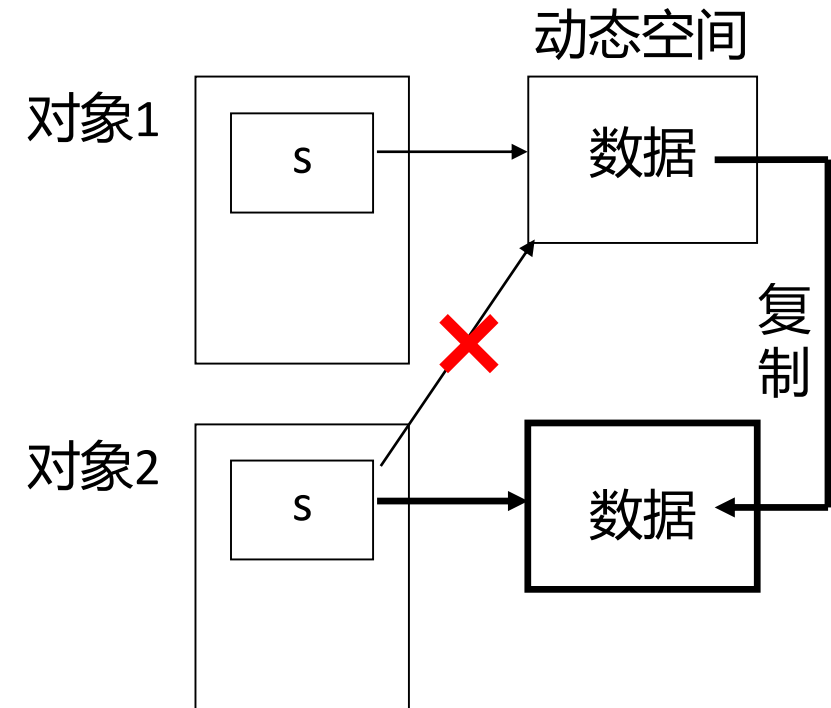
——2步实现**深拷贝**

```
mystring :: mystring( mystring &st)
```

```
{    //指针成员申请空间
```

```
    //拷贝值
```

```
}
```



【例7.4】定义学生类——能够实现深拷贝

```
class Student
{   char *pName;           //其他成员略
public:
    student(char *pname = 0); //参数构造
    student(student &s);      //拷贝构造
    ~student();               //析构
    student & operator= (student &s); //赋值运算符重载
    // 其他成员略
};
```

```
int main()
{   Student s1("范英明" ),
    s2("沈俊" ), s3(s1);
    Student s4 = s2; //拷贝
    s3 = s2;         //赋值
    return 0;
}
```



Student :: Student(char *pname) //参数构造

```
{    if(pname)
        if(pName = new char[100])    //pName = pname可以, 但不好
            strcpy(pName, pname);
        else pName = 0;
}
```

Student :: ~Student() //析构

```
{    if(pName)
        {    delete []pName;
            pName = 0;
        }
}
```



Student :: Student(student &st) //拷贝构造

```
{ if(st.pName)
{ if(pName = new char[strlen(st.pName)+1])
    strcpy(pName, st.pName);
}
else pName = 0;
}
```

Student& student :: operator=(student &st) //重载=运算符

{ if(pName) delete []pName; //先清空原对象

深复制代码

return *this; //不能返回局部对象
}



再看自定义字符串类和线性表类模板

```
class mystring
{   char * s;           //支持动态创建
public:
    //成员略
    //实现操作的安全性（针对指针）
};

class someclass
{   mystring st; //不再有指针
public:
    //不必再考虑深拷贝问题    封装的意义所在
};
```



重新封装线性表类模板

```
template <typename T, int size>
class Seqlist1
{   T slist[size];
    int last;
public:
    Seqlist1();
    Seqlist1& operator=(Seqlist1&);
    bool insert(T, int);
    //其他成员略
};
```

```
template <typename T>
class Seqlist2
{   T *slist;           //数组首地址
    int last;           // 最后一个元素下标
    int maxsize;        //数组长度
public:
    seqlist2( int n = 10); //构造缺省长度10
    void insert(T, int);   //下标处插入
    Seqlist2& operator=(Seqlist2&); //赋值
    void enlarge(int n = 10); //扩容
    ~Seqlist2();           //析构
    //其他成员略
};
```



```
template<typename T>
```

```
Seqlist2<T> :: Seqlist2 (int n)
```

//带参构造

```
{    last = -1;
```

//红色为核心功能

```
    if(n > 0 )
```

```
    {    maxsize = n;
```

```
        slist = new T [n];
```

//申请n个元素数组空间

```
        if(!slist) cout << "fail 0" << endl;
```

```
    }
```

```
    else {    maxsize = 0;  slist = 0;  }
```

```
}
```

```
template<typename T>
```

//析构函数释放动态空间

```
Seqlist2<T> :: ~Seqlist2()
```

```
{    if (slist)
```

```
    {    delete []slist;  slist = 0;  }
```

```
}
```




```
template<typename T>
Seqlist2& Seqlist2<T> :: operator=(Seqlist2& sl)
{   if (slist) {   delete []slist;   slist = 0;   }
    //算法同拷贝构造, 略
    return *this;
}

template<typename T>
void Seqlist2<T> :: insert(T &t, int pos) //元素t插入数组下标pos处
{   if (isfull()) enlarge(20); //如果数组已满, 扩容20个元素
    for (int i = last; i >= pos; i--)
        slist[i + 1] = slist[i] //移动元素, 空出位置
    slist[i] = t; //插入元素t
    last ++;
}
```



```
template<typename T>
void Seqlist2<T> :: enlarge ( int n)
{   T* p = slist;
    slist = new T [maxsize + n];
    if(!slist) cout<<"fail 1"<<endl;
    else
    {   for (int i = 0; i <= last; i++)
        slist[i] = p[i];
        maxsize = maxsize + n;
        if (p) delete []p;
    }
}
```

//数组扩容

//记录原数组地址

//申请新的数组空间

//原数组数据移入新空间

//释放原数组空间

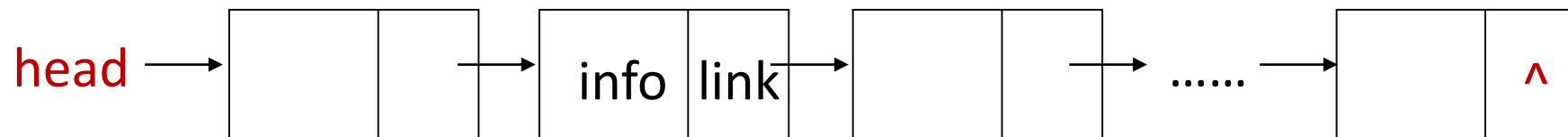


Q&A

链表基本操作与单链表模板类

线性表的两种物理结构：顺序表、链表

单（向）链表的表达（Single Linked List）



结点（Node）——数据单元：数据、指针（什么指针）

```
typedef int T;  
struct Node  
{ T info;  
  Node *link;  
};
```

有关结点访问的表达式和语句：

```
Node *p = head, *q;  
p->info, p->link  
p = p->link; q = p->link;
```

单链表基本操作

遍历（输出、查找）

节点插入（p节点后、p节点前）

删除节点（p的后节点、p节点）

链表建立（正向、倒向）

```
void print(node* head);
```

```
node* search(node* head, T key);
```

```
void insertAfter(node* p, T key);
```

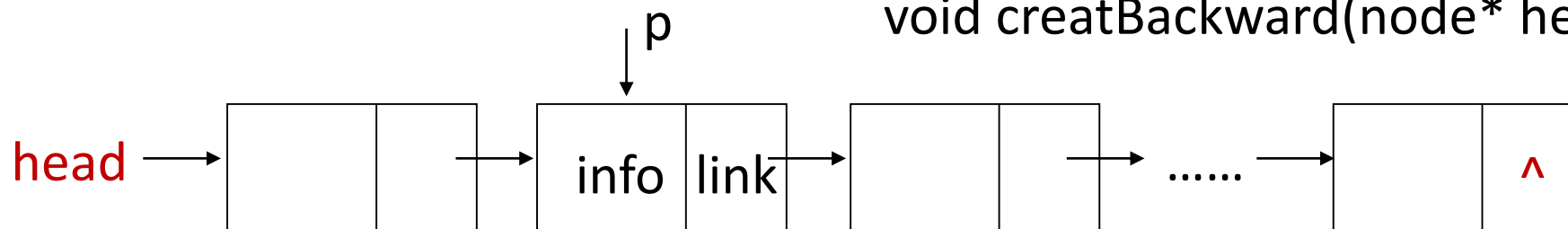
```
void insertFront(node* head, node* p, T key);
```

```
void deleteAfter(node* p);
```

```
void delete(node* head, node* p);
```

```
void creatForward(node* head);
```

```
void creatBackward(node* head);
```



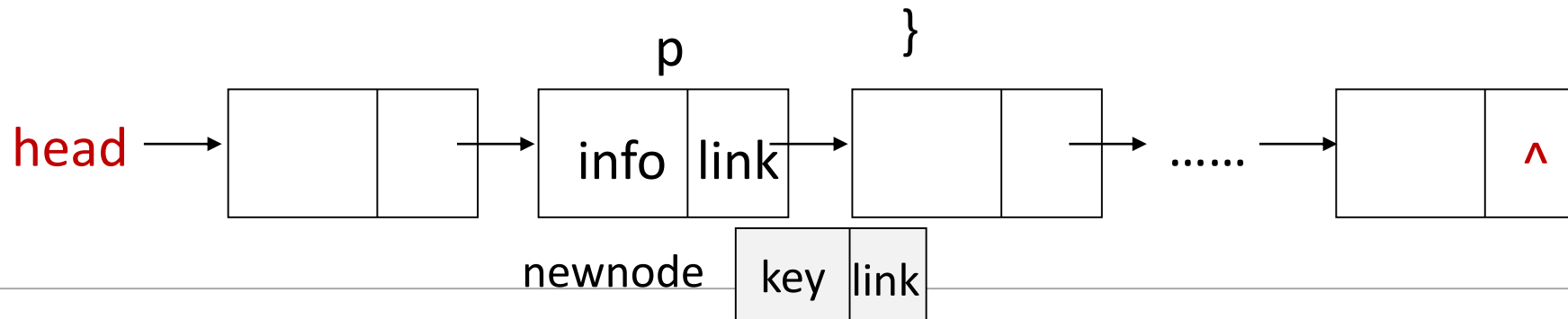
```
node* search(node* head, T key)
{
    node* p = head;
    while(p != 0 && p->info != key)
        p = p->link;
    return p;
}
```





```
void insertAfter(node* p, T key)
{
    node* newnode;
    newnode = new node(key);
    newnode->link = p->link;
    p->link = newnode;
}
```

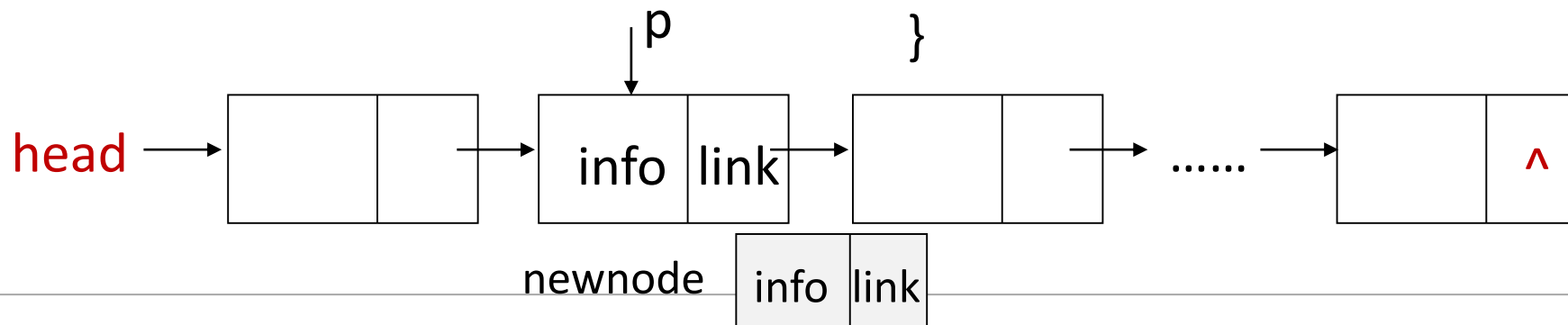
```
void insertFront(node* head, node* p, T key)
{
    if(p == head)           //key做成头节点
    else
    {
        node* t1 = head, *t2 = t1->link;
        while(t2 != 0)     //找p的前节点t1
            if (t2 == p) break;
            else
            {
                t1 = t1->link; t2 = t1->link;
            }
        insertAfter(t1, key);
    }
}
```



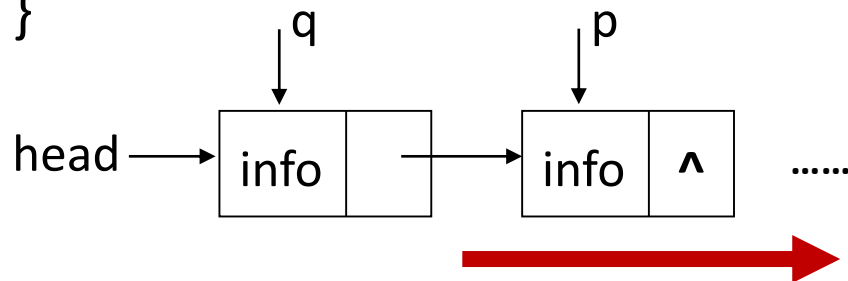


```
void deletetAfter(node* p)
{ if (p->link != 0)
  { node* q = p->link;
    p->link = q->link;
    delete q;
  }
}
```

```
void delete(node* head, node* p)
{ if(p == head)           //删除头节点
  else
  { node* t1 = head, t2 = t1->link;
    while(t2 != 0)        //找p的前节点t1
      if (t2 == p) break;
      else
      { t1 = t1->link; t2 =t1->link; }
    deleteAfter(t1);
  }
}
```




```
void creatForward(node* head)
{
    node* p, *head = 0, *q;
    while( /*循环条件*/ )
    {
        p = new node; //创建节点
        cin >> p->info; //放数据
        p->link = 0;
        if(head == 0) head = p;
        else q->link = p;
        q = p;
    }
}
```



```
void creatBackward(node* head)
{
    node* p, *head = 0, *q;
    while( /*循环条件*/ )
    {
        p = new node; //创建节点
        cin >> p->info; //放数据
        if(head == 0) p->link = 0;
        else p->link = head;
        head = p;
    }
}
```



Q&A

单链表类模板

先封装节点

```
class Node
{
    int info;
    Node *link;
public:
    Node( );
    Node(int);
    void insertAfter(Node *);
    Node<T>* removeAfter();
};
```

再升级为结点类模板

```
template<typename T> class List;
template<typename T>
class Node
{
    T info;
    Node<T> *link;
public:
    Node(); //生成空结点
    Node(const T& data); //生成数据结点
    void insertAfter(Node<T>* p); //p节点放在本结点后
    Node<T>* removeAfter(); //删除本结点的后结点
    friend class List<T>; //List为友元类，方便访问Node
};
```

在此基础上封装链表类模板



```
template<typename T>
Node<T> :: Node()
{   link = 0;   }

template<typename T>
Node<T> :: Node(const T& data)
{   info = data;   link=0;   }

template<typename T>
void Node<T> :: insertAfter(Node<T>* p)
{   p->link=link;   link=p;   }
```

```
template<typename T>
Node<T>* Node<T> :: removeAfter()
{   if (link == 0)   return 0;   //是孤立结点
    else
    {   Node<T> *t = link ;
        link = t->link;
        return t;   //返回被删结点地址
    }
}
```



定义链表类模板

```
template<typename T>
```

```
class List
```

```
{   Node<T> *head, *tail;
```

//头指针和尾指针

```
public:
```

```
    List();           //构造函数, 生成空链表 (没有有效数据的链表)
```

```
    ~List();          //析构函数
```

```
    int length();      //计算单链表长度
```

```
    void print();      //打印链表
```

```
    void makeEmpty();  //清空链表至空链表
```

```
    Node<T>* find(T &data); //找值为data的结点
```

```
    Node<T>* findFront(Node<T>*p); //找p的前节点, 增加该成员
```

```
    Node<T>* CreatNode(T &data); //创建孤立结点, 去掉
```

```
// to be continued...
```



//go on...

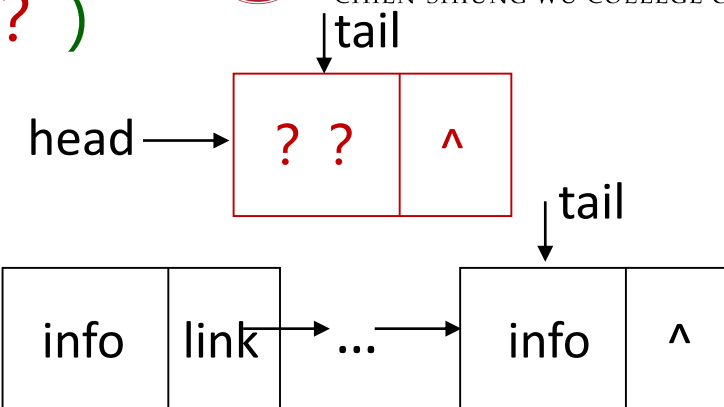
```
void insertFront(Node<T>* p);  
void insertRear(Node<T>* p);  
void insertAsc(Node<T> *p);  
void insertPos(Node<T> *p, Node<T> *pos);  
Node<T>* deleteAfter(Node<T>*p);  
Node <T>* deleteNode(Node<T>* p);  
};
```

//p插在链头，倒生链表
//p插在链尾，正生链表
//p按升序插入
//p插在pos后，增加
//删除p后节点，增加
//删除指定p结点

```
template<typename T>
```

```
List<T> :: List()
{
    head = tail = new Node<T>;
    head->link = 0;
}
```

//建立空链表 (空状态?)
//建虚结点



```
template<typename T>
```

```
List<T> :: ~List()
{
    makeEmpty(); delete head; head = tail = 0;
}
```

虚/空节点 (非有效数据节点)

不带虚结点怎样构造?

```
template<typename T>
```

```
void List<T> :: makeEmpty()
{
    Node<T> * t;
    while (head->link != 0)
    {
        t = head->link; head->link = t->link; delete t;
    }
}
```

//不带虚节点不需要此函数, 直接析构函数
//不带虚结点的情况如何写条件?