

# 实验四 C++对C的扩展（动态存储与链表）

## 一、 问题描述

### 1. 实验目的：

掌握“C++对C扩展”中涉及的若干基本概念和特性，并能够应用于程序编写。

掌握验证性实验的基本方法和过程(认知、实验、总结)。

### 2. 实验内容：

分别编写一段测试代码来回答任务书中的相关问题（每一个问题，用一个工程文件，同时需要记录相应的调试过程），具体问题请参考“实验任务说明04.doc”；

调试的过程；（动态调试的相关截图，比如 设置断点、查看当前变量值等）；

编译出来的可执行程序单独放在一个目录下（bin/exe/debug目录下，同时附上输入数据说明和输出结果）

### 3. 具体内容：

- 1、巩固数组，特别是巩固把数组作为存储空间的算法；
- 2、动态存储与动态数组
- 3、提升链表

## 二、 实验过程

### 1. 名词解释

#### 1、动态存储与静态存储

答：

① 动态存储方式是指在程序运行期间根据需要进行动态的分配存储空间的方式。动态存储变量在程序执行过程中，使用它时才分配存储单元，使用完毕立即释放。典型的例子是函数的形式参数，在函数定义时并不给形参分配存储单元，只是在函数被调用时，才予以分配，调用函数完毕立即释放。如果一个函数被多次调用，则反复地分配、释放形参变量的存储单元。

② 静态存储方式是指在**程序编译期间**分配固定的存储空间的方式。该存储方式通常是在变量定义时就分定存储单元并一直保持不变，直至整个程序结束。全局变量，静态变量等就属于此类存储方式。

## 2、动态数组

答：

动态数组是相对于静态数组而言。静态数组的长度是预先定义好的，在整个程序中，一旦给定大小后就无法改变。而动态数组则不然，它可以随程序需要而重新指定大小。动态数组的内存空间是从堆（**heap**）上分配（即动态分配）的。是通过执行代码而为其分配存储空间。当程序执行到这些语句时，才为其分配。程序员自己负责释放内存。

## 3、单链表、循环链表、双向链表

答：

（1）单链表包含两个域，一个信息域和一个指针域。这个链接指向表中的下一个节点，而最后一个节点则指向一个空值**NULL**。单向链表只可向一个方向遍历。查找一个节点的时候需要从第一个节点开始每次访问下一个节点，一直访问到需要的位置。也可以提前把一个节点的位置另外保存起来，然后直接访问。

（2）循环链表首节点和末节点被连接在一起。循环链表可以被视为“无头无尾”。循环链表中第一个节点之前就是最后一个节点，反之亦然。循环链表的无边界使得在这样的链表上设计算法会比普通链表更加容易。

（3）双向链表中不仅有指向后一个节点的指针，还有指向前一个节点的指针。第一个节点的“前连接”指向**NULL**，最后一个节点的“后连接”指向**NULL**。这样可以从任何一个节点访问前一个节点，也可以访问后一个节点，以至整个链表。

## 4、头结点、首结点、头指针

答：

（1）头结点：它是在首元结点之前附设的一个节点，其指针域指向首元结点。头结点的数据域可以不存储任何信息，也可以存储与数据元素类型的其他附加信息，例如，当数据元素为整数型时，头结点的数据域中可存放该线性表的长度。

（2）首结点：链表中存储第一个数据元素的结点。

（3）头指针：它是指向链表中的第一个结点的指针。若链表设有头结点，则头指针所指结点为线性表的头结点；若链表不设头结点，则头指针所指结点为该线性表的首元结点。

## 5、内存泄漏、悬浮指针

答：

(1) 内存泄露：操作堆内存时，如果分配了内存，就有责任回收它，否则这块内存就无法重新使用，称为内存泄漏。

(2) 当所指向的对象被释放或者收回，但是对该指针没有作任何的修改，以至于该指针仍旧指向已经回收的内存地址，此情况下该指针称为悬浮指针。若操作系统将这部分已经释放的内存重新分配给另外一个进程，而原来的程序重新引用现在的悬浮指针，则将产生无法预料的后果。

## 2. 填空题

动态存储与静态存储

在存储数据时，有顺序存储（如数组等）和链式存储（如链表），请回答：

1、当线性表的元素总数基本稳定，且很少进行插入和删除操作，但要求以最快的速度存取线性表中的元素时，应采用（**顺序**）存储结构。

2、在一个长度为 $n$ 的顺序表中第 $i$ 个元素（ $1 \leq i \leq n$ ）之前插入一个元素时，需向后移动（ **$n-i+1$** ）个元素。

3、顺序存储结构是通过（**物理上相邻**）表示元素之间的关系的；链式存储结构是通过（**指针**）表示元素之间的关系的。

## 3. 选择题

1、new是C++语言提供的用于动态数据生成的（**B**），返回值为（**D**）。

A 函数    B 运算符    C void \*指针    D 与new后面的类型一致的类型

2、链表不具有的特点是（**B**）。

A 插入、删除不需要移动元素    B 可随机访问任一元素  
C 不必事先估计存储空间    D 所需空间与线性长度成正比

3、在单链表指针为 $p$ 的结点之后插入指针为 $s$ 的结点，正确的操作是（**B**）。

A  $s \rightarrow next = s; s \rightarrow next = p \rightarrow next;$     B  $s \rightarrow next = p \rightarrow next; p \rightarrow next = s;$   
C  $p \rightarrow next = s; p \rightarrow next = s \rightarrow next;$     D  $p \rightarrow next = s \rightarrow next; p \rightarrow next = s;$

4、对于一个头指针为head的带头结点的单链表，判定该表为空表的条件是（**B**）。

A  $head == NULL$     B  $head \rightarrow next == NULL$     C  $head \rightarrow next == head$   
D  $head != NULL$

#### 4. 简答题

##### 1、比较静态数组和动态数组之间的异同点。

答：

（1）静态数组的大小是在编译期间就确定，并且分配的，其内存使用结束后由计算机自动释放，效率高；动态数组是在程序运行时，由程序员根据实际需要，从堆内存中动态申请的，使用结束后由程序员进行释放，效率低。

（2）对静态数组进行sizeof运算时，结果是整个数组的大小，而对动态数组进行sizeof运算时，结果为常数4。因为此时数组名是一个数组指针，即一个地址，占用4个字节的内存。

（3）在一个函数内声明的静态数组不可能通过函数返回，因为生存期的问题，函数调用完其内部变量占用的内存就被释放了。如果想通过函数返回一个数组，可以在函数中用new动态创建该数组，然后返回其首地址。静态数组是在栈中申请的，而函数中的局部变量也是在栈中的，而new动态数组是在堆中的分配的，所以函数返回后，栈中的申请的内存被自动释放，而堆中申请的内存如果没有delete就不会自动释放。

##### 2、在比较顺序存储（如数组等）和链式存储（如链表）特性的基础上，重点分析他们的应用情景（适合解决什么问题）。

答：

（1）顺序表的存储空间是静态分配的，在程序执行之前必须明确规定它的存储规模，设定过大会造成存储空间的浪费，过小造成溢出。因此，当对线性表的长度或存储规模难以估计时，不宜采用顺序表。链表的动态分配则可以克服这个缺点。链表不需要预留存储空间，也不需要知道表长如何变化，只要内存空间尚有空闲，就可以再程序运行时随时地动态分配空间，不需要时还可以动态回收。因此，当线性表的长度变化较大或者难以估计其存储规模时，宜采用动态链表作为存储结构。

（2）在链表中，除数据域外需要在每个节点上附加指针。如果节点的数据占据的空间小，则链表的结构性开销就占去了整个存储空间的大部分。当顺序表被填满时，则没有结构开销。在这种情况下，顺序表的空间效率更高。由于设置指针域额外地开销了一定的存储空间，从存储密度的角度来讲，链表的存储密度小于1。因此，当线性表的长度变化不大而且事先容易确定其大小时，为节省存储空间，则采用顺序表作为存储结构比较适宜。

（3）在顺序表中做插入，删除时平均移动表中一半的元素，当数据元素的信息量较大而且表比较长时，在链表中作插入、删除，虽然要找插入位置，但操作是比较快捷，从这个角度考虑显然后者优于前者。通常“较稳定”的线性表，即主要操作是查找操作的线性表，适于选择顺序存储；而频繁做插入删除运算的（即动态性比较强）的线性表适宜选择链式存储。

3、问题域中有一数据序列 ( $a_1, a_2, \dots, a_n$ )，用顺序存储表示时， $a_i$ 和 $a_{i+1}$ 的物理位置相邻吗？用链表表示时呢？

答：

顺序映射时， $a_i$ 与 $a_{i+1}$ 的物理位置相邻；链表表示时 $a_i$ 与 $a_{i+1}$ 的物理位置不要求相邻。

## 5. 程序阅读题

### 1、阅读程序，并输出结果

```
#include <iostream>
using namespace std;

int main() {
    using namespace std;
    //动态分配大小为3的数组
    double* p3 = new double[3];
    //给该数组赋值
    p3[0] = 0.2;    p3[1] = 0.5;    p3[2] = 0.8;
    cout << "p3[1] is" << p3[1] << ".\n";
    p3 = p3 + 1;
    cout << "Now p3[0] is" << p3[0] << " and ";
    cout << " p3[1] is" << p3[1] << ".\n";
    p3 = p3 - 1;
    delete[] p3;
    return 0;
}
```

#### (1) 实验结果

p3[1] is 0.5.

Now p3[0] is 0.5 and p3[1] is 0.8 .

#### (2) 上机验证



#### (3) 分析过程

首先使用new分配了大小为3的数组，并给数组赋值。p3=p3+1，这时p3

的地址移动一位，p3[0]由原来的p3[0]（值为0.2）变为原来的p3[1]（值为0.5）。p3[1]由原来的p3[1]（值为0.5）变为原来的p3[2]（值为0.8）。

(4) 断点调试

① p3=p3+1前

p3	0x006fb940 {0.20000000000000001}
p3[0]	0.20000000000000001
p3[1]	0.5000000000000000
p3[2]	0.8000000000000004
添加要监视的项	

② p3=p3+1后

p3	0x006fb948 {0.5000000000000000}
p3[0]	0.5000000000000000
p3[1]	0.8000000000000004
p3[2]	-1.4568160835476641e+144

## 6. 程序题

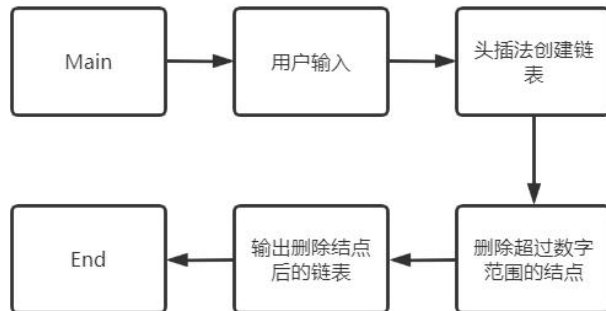
(一) 链表应用题

1、已知线性表中的元素以值递增有序，并以单链表作存储结构。试写一程序，删除有序表中所有其值大于mink 且小于maxk的数据元素。

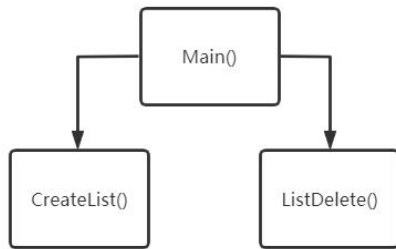
(0) 设计思路：

先遍历找到这些节点(只需要找到从首元结点到第一个 data 域 大于等于 maxk 的节点就行)，再找到第一个 data 域大于等于 mink 的节点，并让该节点指向其的 next->next 同时释放该节点的 next 的内存空间。

(1) 主程序模块：



(2) 函数调用关系:



(3) 具体设计:

### 1、结构体定义

```
//定义结构体
typedef struct LNode
{
    int data;
    struct LNode* next;
}LNode, * LinkList;
```

### 2、建立单链表函数

```
//建立单链表
void CreateList(LinkList& L, int n)
{
    LinkList s, p;
    p = L = new LNode;
    cout << "请输入结点的数据:" << endl;
    for (int i = 0; i < n; i++){
        s = new LNode;
        cin >> s->data; //读入数据
        p->next = s;
        p = s;
    }
    p->next = NULL;
}
```

### 3、删除节点函数

```

//删除超过范围的数字
void ListDelete(LinkList& L, int mink, int maxk){
    LNode* p, * q;
    p = new LNode;
    q = new LNode;
    q = L->next;
    p = L;
    while (q && q->data <= mink) {
        p = q;
        q = q->next;
    }
    while (q && q->data < maxk) {
        p->next = q->next;
        delete(q);
        q = p->next;
    }
    p = L->next;
    cout << "删除结点后链表为:";
    while (p) {
        cout << p->data << ' ';
        p = p->next;
    }
}

```

#### 4、main函数

```

int main() {
    LinkList L;
    int n, mink, maxk;
    cout << "请输入结点数量:";
    cin >> n;
    CreateList(L, n);
    cout << "请输入mink和maxk:";
    int k;
    cin >> mink >> maxk;
    ListDelete(L, mink, maxk);
    return 0;
}

```

#### (4) 实验结果:

 Microsoft Visual Studio 调试控制台

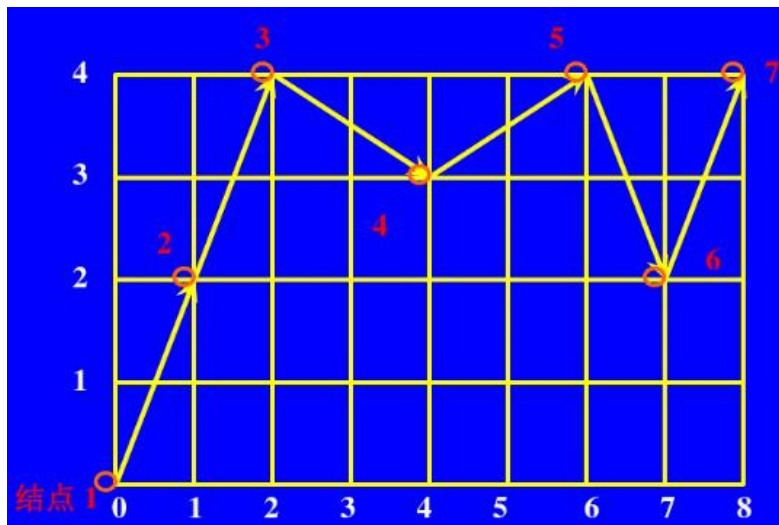
```

请输入结点数量:10
请输入结点的数据:
1 2 3 4 5 6 7 8 9 10
请输入mink和maxk:2 8
1 2 8 9 10

```



2、跳马。依下图将每一步跳马之后的位置 (x,y) 存放到一个“结点”中，再用“链子穿起来”，形成一条链。



(0) 设计思路:

上图有7个结点，为了表示这种既有数据又有指针的情况，使用链表进行表示。马走过的路径先存在数组中，定义位置结点信息如下，其中x代表横坐标，y代表纵坐标。

结点	n1	n2	n3	n4	n5	n6	n7
x	0	1	2	4	6	7	8
y	0	2	4	3	4	2	4

//马的跳步，按顺序排放

```
int a[7][2] = { {0,0}, {1,2}, {2,4}, {4,3}, {6,4}, {7,2}, {8,4} };
```

//位置结点

```
struct jump {
    int x;
    int y;
    struct jump* next;
};
```

(1) 主程序模块:



(2) 实验代码:

```

//main函数
int main() {
    jump* head = nullptr;
    jump* p = nullptr;
    jump* q = nullptr;
    head = new jump; //头结点
    p = q = head;
    head->x = a[0][0], head->y = a[0][1];
    //连接结点
    for (int i=1; i<7; i++) {
        q = new jump;
        q->x = a[i][0], q->y = a[i][1];
        p->next = q;
        q->next = nullptr;
        p = q;
    }
    //遍历链表进行打印
    cout << "马走过的路径如下: " << endl;
    q = head;
    while (q!=nullptr) {
        cout << "(" << q->x << ", " << q->y << ") " << endl;
        q = q->next;
    }
}

```

(3) 实验结果:

```

Microsoft Visual Studio 调试控制台
马走过的路径如下:
(0, 0)
(1, 2)
(2, 4)
(4, 3)
(6, 4)
(7, 2)
(8, 4)

```

(二) 动态数组的应用题

1、用new创建一个数组，从键盘上读取数组长度和数据，分别采用：选择排序、快速排序和冒泡排序，并输出数据。

(0) 设计思路

① 选择排序:

进行n次循环。第一次循环，将第一个数与后面的六个数进行比较，选出最小的放在第一个位置；下一次则是从第二个数开始，与后面的数进行比较，然后将第二小的数放在第二个位置；然后以此类推，直到排完为止。

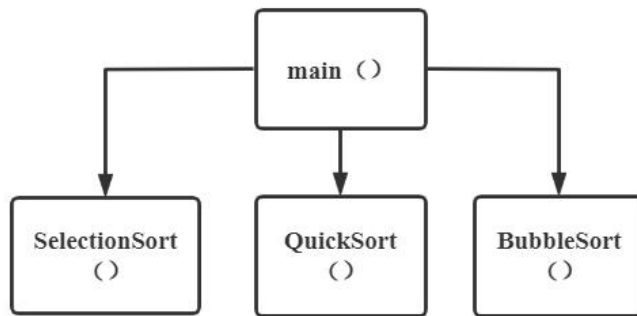
## ② 快速排序

通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据比另一部分的所有数据要小，再按这种方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，使整个数据变成有序序列。

## ③ 冒泡排序

无序表中的所有记录，通过两两比较关键字，得出有序序列。

### (1) 调用关系



### (2) 实验代码

#### ① 选择排序函数

```
//选择排序
void SelectionSort(int n) {
    for (int i = 0; i <= n; i++) {
        for (int j = i + 1; j <= n - 1; j++) {
            if (a[i] > a[j]) {
                int temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

#### ② 快速排序函数

```

//快速排序
void QuickSort(int left, int right) {
    int i, j, t, temp;
    if (left > right) return;
    temp = a[left]; //temp中存的就是基准数
    i = left;
    j = right;
    while (i != j) { //顺序很重要，要先从右边开始找
        while (a[j] >= temp && j > i) j--;
        while (a[i] <= temp && j > i) i++; //再找左边的
        if (i < j) { //交换两个数在数组中的位置
            t = a[i];
            a[i] = a[j];
            a[j] = t;
        }
    }
    a[left] = a[i]; //最终将基准数归位
    a[i] = temp;
    QuickSort(left, i - 1); //继续处理左边的，这里是一个递归的过程
    QuickSort(i + 1, right); //继续处理右边的，这里是一个递归的过程
}

```

### ③ 冒泡排序函数

```

//冒泡排序
void BubbleSort(int n) {
    for (int i = 1; i <= n - 1; i++) {
        for (int j = 1; j <= n - i; j++) {
            if (a[j - 1] > a[j]) {
                int temp = a[j - 1];
                a[j - 1] = a[j];
                a[j] = temp;
            }
        }
    }
}

```

### ④ main函数

```

int main() {
    int n;
    cout << "要排序的数的数量: ";
    cin >> n;
    //输入数组
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }
    SelectionSort(n); //选择排序
    QuickSort(0, n - 1); //快速排序
    BubbleSort(n); //冒泡排序
    //输出数组
    cout << endl;
    for (int i = 0; i < n; i++) {
        cout << a[i] << " ";
    }
}

```

2、用动态数组分别存储两个字符串，并将两个字符串连接的结果存储至第一个动态数组中。【重点描述，若动态数组空间申请不够时，如何动态扩展新空间】

(0) 设计思路:

用动态数组mystr1和mystr2分别存储两个字符串，将在堆上分配15个连续内存。当想进行字符串连接，而动态数组空间申请不够时，可以再new一个两倍大的内存，将原来的字符串拷贝到新的空间，再进行字符串拼接。

C++的stl可以帮助我们更简单地实现字符串的连接，使用标准库类型的vector，调用push\_back可将string2数组的内容逐个拼接到string1后，vector的原理其实也是如果往vector中添加元素时，而动态分配的空间不够时，会重新分配一段更大的内存（原来内存大小的2倍或1.5倍），然后将旧内存中的数据拷贝到新内存，最后释放旧内存。

此外，也可以使用标准库类型string，string是c++中表示字符串的容器，其底层的字符串表示方式仍然是以'\0'表示的字符串集合，可以直接用+=的方法实现字符串的连接。

(1) 实验代码:

```

#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main() {
    //使用标准库类型string
    string str1("Hello!"), str2("My friend!");
    str1 += str2;
    cout << str1 << endl; 已用时间 <= 1ms

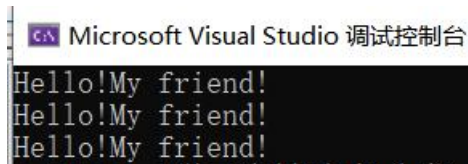
    //使用标准库类型vector
    vector<char> string1{'H','e','l','l','o','!'}; //动态数组
    vector<char> string2{'M','y',' ','f','r','i','e','n','d','!'};
    string1.pop_back();
    for (char i:string2) {
        string1.push_back(i); //字符串连接
    }
    for (char j:string1) {
        cout << j; //输出最终字符
    }

    //自己编写
    char* mystr1 = new char[15]{'H','e','l','l','o','!'}; //在堆上分配15个连续内存
    char* mystr2 = new char[15]{'M','y',' ','f','r','i','e','n','d','!'}; //在堆上分配15个连续内存
    //动态分配内存不够
    char* newstr1 = new char[30]; //在堆上分配30个连续内存
    for (int i = 0; mystr1[i] != 0; i++) {
        newstr1[i] = mystr1[i];
    }
    delete[] mystr1;
    for (int i = 0; mystr2[i] != 0; i++) {
        newstr1[i+6] = mystr2[i];
    }
    delete[] mystr2;
    cout << endl << newstr1;
}

```

## （2）实验结果：

都成功实现了字符串的拼接



```

Microsoft Visual Studio 调试控制台
Hello!My friend!
Hello!My friend!
Hello!My friend!

```

## 三、附录

源程序文件项目清单： 5.1 6.1 6.2 6.3 6.4