梅河口市第五中学信息学竞赛培训材料

第 7 次笔记

对应章节：Chapter 6

说明：我们根据需要发布参考笔记，这部分内容作为对教材的说明解释或者额外补充，教材正确合理的部分不作赘述。对于竞赛，语言深度不要求很深，那些不必掌握的知识，笔记中会有注明。

---

1. 综述

本章结束后，语言的特性就介绍完全了，更深入的知识已经没有必要，这几个章节的内容熟练掌握即可。结构体给出了组织数据的新方法，如果我们分析的对象是“学生”，他有学号、姓名、性别、专业、年级、导师等等信息，为这些信息分别建立数组是不利于管理的，例如，将学生按学号排序，如果信息分散，排序的操作将变得非常棘手，但当我们把所有信息用一个包裹（结构体）包起来，它就变成了一种“类型”，对数据的管理就会更方便。（事实上，C++的类就是对结构体的扩充）

自引用结构从刚才的基础上进一步增加了灵活性，在教材的基础上，笔记会给出经典的单向、双向链表的写法供同学们参考。相比于数组，基于指针的动态的链表让查找、删除、追加等操作变得非常简单，而且动态的内存分配不需要“整块”的内存，使内存利用效率更高。不过也会有内存泄漏的危险。联合的思想仍然是“同一块内存的不同解析”，笔记中也会讲解。

不必阅读的部分：6.9 节第 131 页“尽管这些方法”之后的部分

2. 结构体的类型定义书写格式

首先，不论怎么写，不要漏分号。形如这样的书写格式：

```
struct
{
    int n;
} myt1, myt2;
```

称作**匿名结构体**，它只能在分号前定义变量，变量类型事实上是 struct <unnamed>。形如这种：

```
struct MyStruct
{
    int n;
} myt1;
```

则结构体有了名称 MyStruct，分号前的 myt1 是定义的该结构体的变量，当还需要定义变量时，可以：

```
struct MyStruct myt2;
```

此时必须加关键字 struct 以让编译器明白 MyStruct 是一种类型名，教材出现 typedef 之前也是这么做的。为了方便起见，我们引入了类型定义也就是 typedef：

```
typedef struct
{
    int n;
} MyStruct;
```

将之视为 typedef…MyStruct;这样的格式，可见，typedef 将 Mystruct 定义为…所示的匿名结构体。从而在定义变量时我们可以不再写 struct 关键字，直接写：

```
MyStruct *pmyt;
```

但是 typedef 有一个规定，那就是“…”中的内容必须完全定义，不能与 typedef 的结果也就是新的关键字有递归关系，例如下面这种书写方式中有错误：

（内部资料）

```
typedef struct CNode
{
    struct CNode *pNext;
    TNode *pPrev;
} TNode;
```

如果编译器要知道 TNode 作为类型说明符的含义，就要完全定义 struct CNode，但这个结构体里用到了 TNode，这就好比“什么是 TNode？——参见 TNode”，编译器会疯掉的。因此我倾向这种写法：

```
typedef struct CNode
{
    struct CNode *pNext;
} CNode;
```

既做到了令 CNode 为类型名又避免了递归定义，事实上，struct CNode 与 CNode 在程序中是两种类型名，尽管 CNode 是相同的，却有着不同的上下文，换言之，把前两个 CNode 改成别的名字比如 XNode 也不影响最终的使用，因为 typedef 的作用是让 CNode 与 struct XNode 在编译器眼中等价。

3. 二分查找函数

教材第 120 页写了一个 binsearch 函数，它依靠指针实现，返回值也是指针，**这个函数很重要，一定要会写**。标准库<stdlib.h>中也提供了一个通用的版本 bsearch，它原型如下：

```
void *bsearch(void *Base, size_t NumOfElements, size_t SizeOfElements,
int (*PtFuncCompare)(const void *, const void *));
```

是不是感觉似曾相识？它的参数和 qsort()函数是完全一样的，尤其是比较函数的指针必须一致。需要注意的是，**搜索的数组必须预先排好序**，且排序规则和函数指针指向的比较函数规则相同，如果查找成功则返回数组中匹配元素的地址，反之返回 NULL，对于有多于一个元素匹配成功的情况，返回哪一个是未定义的。**虽说通用，也有限制条件，看清楚自己的需求再决定是否调用**。

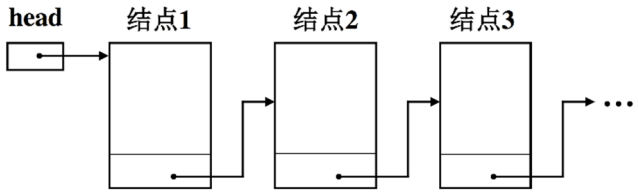
4. 自引用结构

指针的神奇之处就是它能指向任何东西（因此也可以说 C 语言中数据类型被扩充到了无穷多种），当指针指向“同类”时，就形成了自引用结构。事实上，**结构体如果看成包裹的话，指针也就是把这个包裹“串”起来的线，而链表，也就是靠指针串起来包裹形成的链而已**。

相比于数组，链表操作有两大优势，一是空间利用率高（数组必须连续，多维数组也必须连续，因此我们建议采用指针数组），二是操作方便，尤其是插入、删除等操作（数组大小固定，改变长度则需整体拷贝，除非开一个足够大的数组），劣势是链表（C 阶段）需要自己写，故有必要掌握这项技能。

现在我们介绍几种经典模式：单向链表、环形链表、基于链表的二叉树。

a) **单向链表**



该小节对应：教材 6.6 小节的内容。

单向链表就像这样→_→

教材 6.6 小节更高效，为了理解它，我们讲解基础的链表实现方式。考虑这样一个问题：

编写一个程序，该程序接受用户输入，每一行有三个数据，分别是学号（int 型）、姓名（最长 20 字符）和期末成绩（int 型），姓名为英文字符且没有空白字符，如果该行为#号则终止输入，并输出按学生的期末成绩升序排序和降序排序的结果。

首先是要存储的数据的定义（代码顺序不是这样，首先和接下来只表示我讲解的顺序）：

```
typedef struct
{
    int m_id;
    char m_name[MAX_NAME_LEN];
    int m_score;
} Content;
```

接下来，单向链表的定义：

```
typedef struct CNode
{
    Content m_ctt;
    struct CNode *next;
} CNode;
```

我们当然可以在 struct CNode 中写 id、name、score 这几个变量，将它们定义成结构体是为了让演示代码中变与不变的部分分离开，这样只需要做很少修改就可以适用于其他情况。自己写的时候按自己的风格处理，解决问题即可。

接下来是头文件、定义和全局变量

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_NAME_LEN    21

typedef int (*pCmpFunc)(CNode *, CNode *);

static CNode theHead;
static CNode * const pHead = &theHead;
```

链表各项操作要保证在没有结点、只有一个结点和有多个结点这几种情况下都能顺利进行，为了简化链表操作我们约定：**头结点用来保护表头，头结点不存放数据**，这只是约定，也就是说头结点存放数据也是可以的，但你要修改演示代码。另外，**当 next 为 NULL 时表明链表结束**。pHead 指针是指针常量，这样我们在任何操作中都能轻松地访问头结点而且不会丢失或破坏该数据。寻找链表尾结点的遍历操作为：

```
CNode *tail;
for (tail = pHead; tail->next != NULL; tail = tail->next)
    ;
return tail;
```

这是遍历单向链表的基本方式，我们用[ ]运算符访问数组元素是因为它们在内存中连续，但链表在内存中不连续因此要靠 next 指针进行移动。

接下来开始写函数，首先是与特定题目相关的读取数据函数：

```
void FillData(CNode *toFill)
{
    scanf("%d%s%d", &toFill->m_ctt.m_id, toFill->m_ctt.m_name,
        &toFill->m_ctt.m_score);
}
```

它的功能是向结点中填充数据，可以修改它使链表适用于其他题目。

和链表打印函数：

```
void PrintList()
{
    CNode *curr;
    for (curr = pHead->next; curr != NULL; curr = curr->next)
        printf("%d\t%s\t%d\n", curr->m_ctt.m_id,
            curr->m_ctt.m_name, curr->m_ctt.m_score);
}
```

接下来是创建整个链表的函数：

```
int Create()
{
    char mark;
    CNode *curr = NULL;
    while ((mark = getchar()) != '#')
    {
```

```
        if (mark == '\n' || mark == '\t' || mark == ' ')
            continue;
        ungetc(mark, stdin);
        curr->next = (CNode *)malloc(sizeof(CNode));
        curr = curr->next;
        if (curr == NULL)
            return 0;
        FillData(curr);
    }
    curr->next = NULL;
    return 1;
}
```

其中 getchar 与 ungetc 的作用是提前读一个字符进来判断是不是#号，是的话输入结束，不是的话把预读的符号退回去。第一个 if 语句是为了跳过多余的空白字符。返回值用于判断是否创建成功。当需要适应其他情境时，只修改 FillData()函数就可以了。

我们说过，动态分配的内存一定要主动释放，因此链表在结束使命后必须由我们销毁。

下面是释放链表所占空间的函数：

```
void Destroy()
{
    CNode *prev, *curr = pHead->next;
    while (curr != NULL)
    {
        prev = curr;
        curr = curr->next;
        free(prev);
    }
    pHead->next = NULL;
}
```

每步释放操作进行后，prev 指向的内存中的内容会被销毁，我们也就丢失了 prev 的 next 指针，所以需要用 curr 把这个指针保存下来，从而使链表中所有结点被回收。

请注意：全局的变量 theHead 不是分配在堆上的，它在程序结束后将自动释放，free(pHead);将产生严重错误，因为 free 只有回收堆上由 malloc 申请的内存块的权限。

接下来我们写一个函数计算结点的总个数：

```
int GetLength()
{
    int count;
    CNode *curr = pHead->next;
    for (count = 0; curr != NULL; curr = curr->next, count++)
        ;
    return count;
}
```

现在我们开始考虑排序，由于链表在内存中是不连续的，我们只能比较相邻两个结点（结点和结点的后一个结点）中数据的关系，因此选择冒泡排序。另外，交换两个结点的指针和数据都可以，但是改变指针域的代价更高，因此我们采用交换数据的方式进行交换操作。

先写一个与特定问题相关的覆盖函数：

```
void NodeCover(CNode *dst, const CNode *src)
{
    dst->m_ctt.m_id = src->m_ctt.m_id;
    strcpy(dst->m_ctt.m_name, src->m_ctt.m_name);
    dst->m_ctt.m_score = src->m_ctt.m_score;
}
```

这个函数其实是一种赋值运算，当遇到其他问题时，需要修改它。

接下来是两个节点数据交换函数：

```
void NodeSwap(CNode *node1, CNode *node2)
{
    CNode *temp = (CNode *)malloc(sizeof(CNode));
    NodeCover(temp, node1);
    NodeCover(node1, node2);
    NodeCover(node2, temp);
    free(temp);
}
```

有了这些准备，我们就可以进行冒泡排序了，思路和第四次笔记完全相同，只是情境变了而已。

冒泡排序函数：

```
void Sort(pCmpFunc cmp)
{
    int i, j, len = GetLength();
    CNode *ptr;
    for (i = 1; i < len; i++)
    {
        ptr = pHead->next;
        for (j = 0; j < len - i; j++)
        {
            if ((*cmp)(ptr, ptr->next) < 0)
                NodeSwap(ptr, ptr->next);
            ptr = ptr->next;
        }
    }
}
```

当数组变成链表，i 和 j 也就没有了下标的意义，只起计数作用，此时我曾经提过的问题就很关键了，对于冒泡排序，重要的是内外层循环的次数，否则会出现排序不完全的现象。

pCmpFunc 是函数指针，它让我们得以针对结点中不同的排序规则进行排序，该函数只要满足参数列表接受 CNode *指针，当序列符合目标顺序时返回正值，不符合返回负值，相等返回零即可。

接下来定义按成绩排序的函数：

```
int SortByScore(CNode *n1, CNode *n2)
{
    return n2->m_ctt.m_score - n1->m_ctt.m_score;
}
```

我们需要的准备工作都做完了，main()函数就变得很简单——

程序入口（main 函数）：

```
int main()
{
    theHead.next = NULL;
    Create();
    Sort(SortByScore);
    PrintList();
    Destroy();
    return 0;
}
```

该函数中第一句话将全局的头结点的 next 指针初始化为 NULL，这句话是有必要的（为防指针没有自动初始化），NULL 将标记 theHead 这个链表现在为空，没有结点。**输入空、一个结点、多个结点进行测试，程序顺利运行且结果正确表示代码无误**。

要按成绩的降序输出结果，与其再排序一次，不如写一个链表反转函数，把链表头尾对调。对于反转函数，如果我们交换结点的数据，那似乎不太明智，效率太低，因此我们选择更改指针域，指针操作的顺序很关键，搞不好就会让链表“断”掉，**请同学们先根据这个小节开始的图示，思考并画出你认为的指针操作顺序，再看下面的内容**。

链表反转函数：

```
void Reverse()
{
    CNode *curr = pHead->next, *fowd;
    if (curr == NULL)
        return;
    pHead->next = NULL;
    while (curr != NULL)
    {
        fowd = curr->next;
        curr->next = pHead->next;
        pHead->next = curr;
        curr = fowd;
    }
}
```

我们一共需要操作几个指针？（3 个）；每次循环要有几步操作？（4 步）。pHead->next 也就是 theHead.next 这个指针是链表的头指针，由于我们要反转，这个指针将时刻变化最终变化到原链表表尾，它在变化的时候是 curr 指针的前一个结点，也就是说 curr->next 要指向原链表的上一个（上次循环的头结点），此时为了让链表不断掉就需要提前保存 curr->next 为 fowd，并在下次循环时让 curr 变成 fowd，照此进行。

再在 main()函数中加入 Reverse();进行测试，发现输出结果升序变成了降序，表明我们成功了。

至此原问题解决，我们继续讲解几个新的操作让链表更完全。我们知道链表的优势是“动态”，那么动态增加和删除结点的操作是必须具备的。

动态创建结点（在 loc 指向的结点之后）的函数：

```
CNode *Reserve(CNode *loc)
{
    CNode *temp;
    temp = loc->next;
    loc->next = (CNode *)malloc(sizeof(CNode));
    if (loc->next == NULL)
    {
        loc->next = temp;
        return NULL;
    }
    loc->next->next = temp;
    return loc->next;
}
```

函数名是 reserve（预订）而不是 reverse（反转）哈，这时你也可以感受到头结点不保存数据的巨大优势，如果没有这一约定，Reserve()函数将需要额外处理预订第一个结点的情况。这个函数使一些数组中不方便进行的操作变成现实，例如，我们可以修改 Create()函数，当读取学生信息时，边读取边排序（插入法），也就是说，按成绩找到新加入的学生在链表中应有的位置，然后直接用 Reserve()函数分配空间再将新学生塞进来，那么这道题我们连排序都没有用到直接就解决了。

请自行修改 Create()函数实现上面这段话，并仿照上面的函数，自己写一个名为 Push_Back()的函数，该函数无参数，返回值为指向新创建结点的指针，功能是在链表末尾创建一个新结点。

除了动态创建还有动态删除，动态删除操作需要保证不论删除头、尾还是中间都不会使链表断掉，同时该函数应该有一定的健壮性（防止试图删除 theHead 的操作引起程序崩溃）。

动态删除（第 n 个）结点的函数：

```
void Delete(int n)
{
    CNode *temp = pHead, *toDel;
    if (n <= 0)
        return;
    for (; n > 1 && temp != NULL; temp = temp->next, n--)
```

```
        ;
    if (temp == NULL)
        return;
    toDel = temp->next;
    temp->next = toDel->next;
    free(toDel);
}
```

该函数对于合法输入、非法输入都作出了应有的响应，这使它具有**较强的健壮性**。参数由指针变为整型是想演示给大家这两种参数都是可以的只是遍历上有一点点差别。

请自行修改 Delete()函数使它接受 CNode *类型的参数并实现相同的功能，并仿照该函数，自己写一个名为 Pop_Back()的函数，该函数无参数返回 void，功能是销毁链表的最后一个结点。

***小结：**

在我学习链表的时候，老师课件上链表的指针名称是 p、q、r 等，于是我眼前的示例代码是：

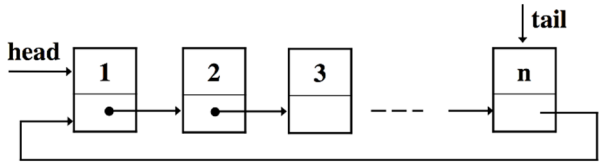
```
while (q != NULL)
{
    p = q->next;
    q->next = r;
    r = q;
    q = p;
}
```

并且由于头结点存了数据，动态操作时往往需要考虑无结点、单结点、多结点三种情况再分别处理。直到有一天我将指针名变成了 **curr（current，当前）、prev（previous，上一个）、fowd（forward，下一个）**，并且从其他书中得到了一条“**头结点不放数据**”的建议，再自己从头到尾尝试一次，才算学会了链表。可能这样命名比 p、q、r 这样写长很多，然而它们能让人更明确每条语句在做什么。示例代码思想上受了 C++的启发，**变与不变分离**开从而让我们只需要修改 Content 结构体的定义、FillData()函数、PrintList()函数、NodeCover()函数和 SortByXXX()函数就可以适应其他问题而不是完全重写一遍链表的代码，工作量减小了一大半。反过来，尽管我不建议，但也不反对提高函数耦合性的做法，例如取消函数指针让 Sort()函数只适用于按成绩升序排序（毕竟，题目只问了这种），自己把握程度达到既让自己舒服，又能解决问题即可。

好现在我们说教材 6.6 小节，它比较糟糕的是所有很多动态分配的内存没有相应的释放操作，但高明的地方是，**hash()函数将字符串（无数种可能）散列到 101 种可能内**，通过指针表 hashtab[ ] 建立了 101 个以内的链表，每次建立是从后往前建立所以链表不会断掉。这样的话搜索和插入的效率就会更高，第 125 页下方的图示说明了教材中代码的核心含义。

b) 环形链表

环形链表与单向链表主框架相同，区别在于，**单向链表最后一个数据的 next 指针指向 NULL 表示链表结束，而环形链表最后一个数据的 next 指针指向第一个结点**，从而使链表构成环状（如图），让循环遍历变得更简单。参考代码略，请根据 a）部分的代码自行修改（仍然有头结点不放数据的建议，相当于原来 theHead 拎起一条链，现在 theHead 拎起一个环）。



c) 基于链表的二叉树

这段笔记对应教材 6.5 小节，它的图示如第 122 页所示，目的是让结点和左右子树之间有比较关系，从而**简化搜索的操作**（就像曾经的二分查找一样），这段代码写的非常好（虽然也没有释放内存），我不作改进只作解释。第 123 页中 addtree()函数是递归的，它的递归终点有两个，一个是找到了匹配单词，则单词计数自增，另一个是没有找到单词，则新建一个结点。第 124 页的 treeprint()函数也是递归的，它递归的终点是 **p 指针为 NULL，则函数即刻退出**。那么它的操作就是沿着所有左子树一路向下，最下方的“左子树”是 NULL，则最深层函数返回，执行最深层函数的主调者

的下一条：打印，然后进入右子树，再看有没有左子树，如此重复进行，也就是说它激活了所有的结点的打印操作，这就是为什么第 123 页中 addtree()函数里的 **p->left = p->right = NULL;**这句话不能去掉的原因，我们必须标记好这棵树的终点。

由此，我们可以仿照 treeprint()函数写出释放内存所用的 treedestroy()函数，该函数也可以递归进行，例如：

```
void treedestroy(struct tnode *p)
{
    if (p == NULL)
        return;
    treedestroy(p->left);
    treedestroy(p->right);
    free(p);
    //p = NULL;
}
```

调用这个函数的时候需要传递给它树根的指针，那么它将沿着左子树一路到头，再沿着没有左子树的结点的右子树一路到头，总能**找到没有左右子树的结点，然后将它释放掉**，p = NULL;这步操作可选（如果没有公共子树）。当最深层函数返回时，递归进行的操作就相当于“我的左膀右臂都释放了，了无牵挂，把我也释放了吧！”，可见递归让我们编写代码更加方便。

5. 联合体

联合体和结构体非常相似，匿名、非匿名联合体和联合体类型定义规则都一致。区别是，联合体是不同的变量共享同一块内存，结构体是不同的变量占用相邻的内存并组织起来。我们知道颜色值通常有三个分量：红，绿和蓝，但是函数参数我们希望尽可能简单，这时我们可以用联合体：

```
union UColor
{
    unsigned long m_rgb;
    struct {
        unsigned char m_red;//低 8 位
        unsigned char m_green;
        unsigned char m_blue;
    };
};
```

在使用的时候，可以通过：

```
UColor color;
color.m_rgb = 0;//这句话是否必要？
color.m_red = 0;
color.m_green = 255;
color.m_blue = 127;
```

传参可以传递 color.m_rgb 即可把颜色分量信息都传过去。**请分析此时该变量的值是多少。**

事实上微软体系下颜色分量的表示是用 RGB(r, g, b)宏通过位运算来完成的。

6. 总结提示

针对语言的讲解暂时告一段落（还有一些文件操作等特性，由于接下来用不到，以后再介绍），竞赛中用 C++的部分不是 C++面向对象的部分（例如，竞赛中很少会写到类），因此我坚信良好的 C 语言基础非常重要。我水平有限，如果发现笔记和代码中有疏漏和谬误之处，还请及时与我联系防止资料误导他人，谢谢！

邮箱：jlmhkwang@163.com

部分资料参考了清华大学软件学院刘玉身老师和雍俊海老师的《软件工程》课程。

版权所有，转载请与作者联系。

（所有内容到此结束）