程序设计分组训练 实验 6 实验报告

专　　业：计算机科学与技术

班　　级：_____

学生姓名：_____

学　　号：_____

北京交通大学计算机与信息技术学院

2021 年 12 月 20 日

目录

1.相关知识准备函数名： system

1.1 qsort 函数

qsort()函数是 C 库中实现的快速排序算法，包含在 stdlib.h 头文件中，其时间复杂度为 O(nlogn)。函数原型如下：

void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));

此函数需要四个参数。

第一个参数是需要排序的数组的基地址，因为是 void * 类型，所以此函数可以给任何类型的数组进行排序；第二个参数是待排序的数量(size_t 是一种特别的数据类型，可以近似理解为 int 型)；第三个是单个数组元素的大小，即字节数，例如 int 型就是 4 或者 sizeof(int) (sizeof 的返回值类型就是 sizeof)，char 型就是 1 或者 sizeof(char)。因为为了适用于各种数据结构，第一个参数将指向数组的指针强转成了 void * 类型，也即此时函数并不知道将要进行排序的数组内存储的是什么元素，因此我们需要显式地告诉它单个元素所占的长度

1.2 冒泡排序

冒泡排序（Bubble Sort）也是一种简单直观的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢"浮"到数列的顶端。

2.实验操作

2.1 思路

新加入了数据排序功能，则按数据结构不同编写相应的排序函数，最后修改一下菜单，加入新的排序功能选项即可。

2.2 Qsort 的自定义比较函数

```
01:二维数组的比较函数
   01:int cmparray(const void *a, const void *b) {
   02:    int* x = *(int **) a;
   03:    int* y = *(int **) b;
   04:    return (x+2 – y+2);
   05:}
02:
03:
04:
05:结构体数组的比较函数
   06:int cmpstructarray(const void *a, const void *b){
   07:    DATAITEM * x= (DATAITEM *)a;
   08:    DATAITEM * y= (DATAITEM *)b;
   09:    return (x->item3 - y->item3);
   10:}
06:
07:结构体指针数组的比较函数
   11:int cmppionterarray(const void*a, const void *b){
   12:    int x = ((DATAITEM *)a)->item3;
   13:    int y = ((DATAITEM*) b)->item3;
   14:    return  (x - y);
   15:}
08:
```

2.3 对链表的冒泡排序

09:思路：
10:冒泡排序属于交换排序的一种，需要频繁交换，链表需要知道前驱和后继，十分复杂，故采用转换数据结构的方法。
11:先转换成数组形式，再进行排序，最后转换回链表。
12:

```
01:void bubblesortlink(Link * head){
02:    int a[head->totaldatanum][3];
03:    Node *tem = head->next;
04:    int i=0;
05:    while(!tem){
06:        a[i][0] = tem->data[0];
07:        a[i][1] = tem->data[1];
08:        a[i][2] = tem->data[2];
09:        i++;
10:        tem = tem->next;
11:    }
12:
13:    for(i = 0; i<head->totaldatanum-1; ++i){
14:        for(int j=0; j <head->totaldatanum-i-1;++j){
15:            if(a[j][2] > a[j+1][2]){
16:                for(int k = 0 ;k<3; ++k){
17:                    int t = a[j][k];
18:                    a[j][k] = a[j+1][k];
19:                    a[j+1][k] = t;
20:                }
21:            }
22:        }
23:    }
24:    i = 0;
25:    tem = head->next;
26:    while(!tem){
27:        tem->data[0] = a[i][0];
28:        tem->data[1] = a[i][1];
29:        tem->data[2] = a[i][2];
30:        i++;
31:        tem = tem->next;
32:    }
33:}
```

2.4 排序前排序后输出到控制台

二维数组

```
01:void show_array(int (*a)[3], int num) {
02:    for (int i = 0; i < num; ++i) {
03:        printf("%d, %d, %d\n", a[i][0], a[i][1], a[i][2]);
04:    }
05:}
```

结构体数组

```
06:void show_structarray(DATAITEM *data, int num) {
07:    for(int i= 0; i<num; ++i){
08:        printf("%d, %d, %d\n", data[i].item1, data[i].item2,
09:data[i].item3);
10:    }
11:}
12:
```

结构体指针数组

```
13:void show_pointerarry(DATAITEM **data, int num) {
14:    for (int i = 0; i < num; ++i) {
15:        printf("%d, %d, %d\n", data[i]->item1, data[i]->item2,
16:data[i]->item3);
17:    }
18:}
```

链表

```
19:void show_link(Link *head) {
20:    Node *tem = head->next;
21:    while(!tem){
22:        printf("%d, %d, %d\n", tem->data[0], tem->data[1], tem-
   >data[2]);
23:        tem = tem->next;
24:    }
25:}
```

2.5 输出演示

```
please enter the recordd count
4
before sort
41, 18467, 6334
26500, 19169, 15724
11478, 29358, 26962
24464, 5705, 28145

after sort
18467, 41, 6334
26500, 19169, 15724
11478, 29358, 26962
24464, 5705, 28145
```

图 2.5-1 二维数组

```
please enter the recordd count
5
before sort
41, 18467, 6334
26500, 19169, 15724
11478, 29358, 26962
24464, 5705, 28145
23281, 16827, 9961

after sort
41, 18467, 6334
23281, 16827, 9961
26500, 19169, 15724
11478, 29358, 26962
24464, 5705, 28145

进程已结束，退出代码为 0
```

图 2.5-2 结构体数组

```
please enter the recordd count
5
before sort
41, 18467, 6334
26500, 19169, 15724
11478, 29358, 26962
24464, 5705, 28145
23281, 16827, 9961

after sort
41, 18467, 6334
23281, 16827, 9961
26500, 19169, 15724
11478, 29358, 26962
24464, 5705, 28145

进程已结束，退出代码为 0
```

图 2.5-2 结构体数组

```
please enter the record count
5
before sort
41, 18467, 6334
26500, 19169, 15724
11478, 29358, 26962
24464, 5705, 28145
23281, 16827, 9961
491, 2995, 11942

after sort
24464, 5705, 28145
41, 18467, 6334
26500, 19169, 15724
23281, 16827, 9961
11478, 29358, 26962
491, 2995, 11942
```

图 2.5-3 结构体指针数组

```
please enter the record count
8
before sort
41, 18467, 6334
26500, 19169, 15724
11478, 29358, 26962
24464, 5705, 28145
23281, 16827, 9961
491, 2995, 11942
4827, 5436, 32391
14604, 3902, 153

after sort
14604, 3902, 153
41, 18467, 6334
23281, 16827, 9961
491, 2995, 11942
26500, 19169, 15724
11478, 29358, 26962
24464, 5705, 28145
4827, 5436, 32391

进程已结束，退出代码为 0
```

图 2.5-4 链表

2.6 修改程序主菜单

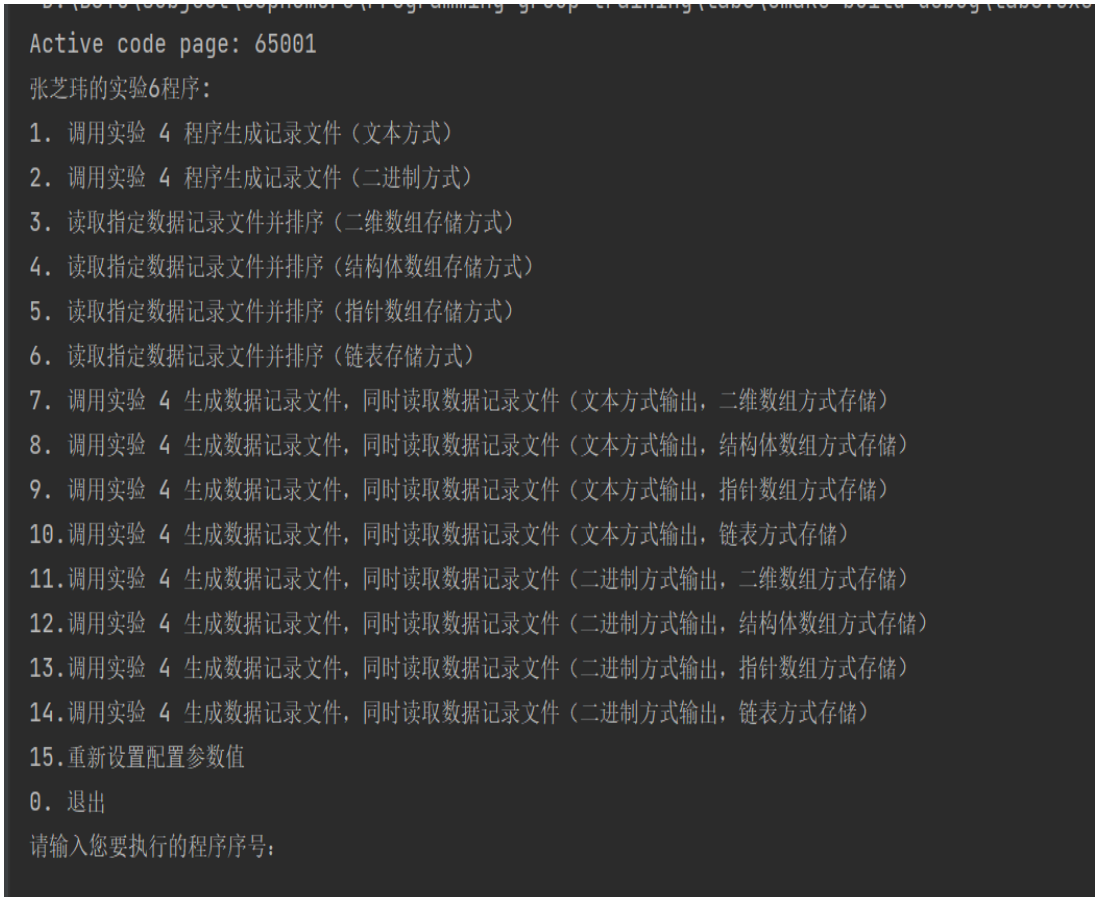

图 2.6

2.7 代码改动

Lab6 在 lab5 的基础上修改而来。

增设 lab6_sort.cpp 和 lab6_sort.h 文件。其中包含了上述的排序函数和显示函数。

然后修改了 loadfile.cpp 文件中的载入函数，在结尾处调用排序函数和显示函数，即排序前显示一次，排序后显示一次。

这里展示了修改后的 loadfile 相关函数，红色部分为修改后的关键函数。

```
01:void loadtoarray(char *path, int filetype) {
02:    FILE * pf;
03:    int record_num;
04:    int **item;
05:    if(filetype==1)
06:    {
07:        pf = fopen(path, "r");
08:        if(!pf)
09:        {
10:            printf("file open error1\n");
11:            exit(1);
12:        }
13:        fscanf(pf, "%d\n", &record_num);
14:
15:        item = (int **) malloc(sizeof (int *)*record_num);
16:
17:        for(int i=0;i<record_num; ++i)
18:        {
19:            item[i] = (int *)malloc(3*sizeof (int ));
20:
21:            //read in format control (cheak in outputfile)
22:            fscanf(pf, "%d, %d, %d,\n", &item[i][0],
&item[i][1], &item[i][2]);
23:        }
24:        fclose(pf);
25:    }
26:    else{
27:        pf = fopen(path, "rb");
28:        if(!pf)
29:        {
30:            printf("file open error1\n");
31:            exit(1);
32:        }
33:
34:        fread(&record_num, 4, 1, pf);
35:        item =(int **) malloc(sizeof(item)*record_num);
36:        for (int i = 0; i < record_num; ++i) {
37:            item[i] = (int *) malloc(sizeof(int)*3);
38:            fread(&item[i], sizeof (int )*3, 1, pf  );
39:        }
40:        fclose(pf);
41:    }
```

```
42:    printf("before sort\n");
43:    showarray(item, record_num);
44:    qsort(&item, record_num, sizeof(int *), cmparray);
45:    printf("\nafter sort\n");
46:    show_array(item, record_num);
47:    showarray(item, record_num);
48:    for (int i = 0; i < record_num; ++i) {
49:        free(item[i]);
50:    }
51:    free(item);
52:}
53:
54:void loadtostructarray(char *path, int filetype) {
55:    FILE * pf;
56:    int record_num;
57:    DATAITEM * item;
58:    if(filetype==1) {
59:        pf = fopen(path, "r");
60:        if (!pf) {
61:            printf("file open error2\n");
62:            exit(1);
63:        }
64:        fscanf(pf, "%d", &record_num);
65:
66:        item = (DATAITEM *) malloc(sizeof(DATAITEM) *
record_num);
67:        for (int i = 0; i < record_num; ++i){
68:
69:            fscanf(pf, "%d, %d, %d,\n", &item[i].item1,
&item[i].item2, &item[i].item3);
70:        }
71:        fclose(pf);
72:    }
73:    else{
74:        pf = fopen(path, "rb");
75:        if(!pf)
76:        {
77:            printf("file open error2\n");
78:            exit(1);
79:        }
80:        fread(&record_num, 4, 1, pf);
81:        item = (DATAITEM*) malloc(sizeof(DATAITEM)*record_num);
82:        for (int i = 0; i <record_num; ++i) {
```

```
 83:           fread(&item[i].item1, 4, 1, pf);
 84:           fread(&item[i].item2, 4, 1, pf);
 85:           fread(&item[i].item3, 4, 1, pf);
 86:       }
 87:       fclose(pf);
 88:   }
 89:   showstructarray(item, record_num);
 90:   printf("before sort\n");
 91:   show_structarray(item, record_num);
 92:
 93:   qsort(item, record_num, sizeof (DATAITEM), cmpstructarray);
 94:   printf("\nafter sort\n");
 95:   show_structarray(item, record_num);
 96:   free(item);
 97:}
 98:
 99:void loadtopointerarray(char *path, int filetype) {
100:   FILE * pf;
101:   int record_num;
102:   DATAITEM **item;
103:   if(filetype==1) {
104:       pf = fopen(path, "r");
105:       if (!pf) {
106:           printf("file open error3\n");
107:           exit(1);
108:       }
109:       fscanf(pf, "%d", &record_num);
110:       item = (DATAITEM **) malloc(sizeof(DATAITEM *) *
 record_num);
111:       for (int i = 0; i < record_num; ++i) {
112:           item[i] = (DATAITEM *) malloc(sizeof(DATAITEM));
113:           fscanf(pf, "%d %d %d,\n", &item[i]->item1, &item[i]-
 >item2, &item[i]->item3);
114:       }
115:       fclose(pf);
116:   }
117:   else{
118:       pf = fopen(path, "rb");
119:       if(!pf)
120:       {
121:           printf("file open error3\n");
122:           exit(1);
123:       }
```

```
124:        fread(&record_num, 4, 1, pf);
125:        item = (DATAITEM**)malloc(sizeof
 (DATAITEM*)*record_num);
126:        for (int i = 0; i < record_num; ++i) {
127:            item[i] = (DATAITEM *) malloc(sizeof(DATAITEM));
128:            fread(&item[i]->item1, 4, 1, pf);
129:            fread(&item[i]->item2, 4, 1, pf);
130:            fread(&item[i]->item3, 4, 1, pf);
131:        }
132:        fclose(pf);
133:    }
134:    showpointerarray(item, record_num);
135:    printf("before sort\n");
136:    show_pointerarry(item, record_num) ;
137:    qsort(item, record_num, sizeof (DATAITEM*),
 cmppionterarray);
138:    printf("\nafter sort\n");
139:    show_pointerarry(item, record_num);
140:    for (int i = 0; i < record_num; ++i) {
141:        free(item[i]);
142:    }
143:    free(item);
144:}
145:
146:void loadtolink(char *path, int filetype) {
147:    FILE *pf;
148:    int record_num;
149:    Link *head = (Link *) malloc(sizeof(Link));
150:
151:
152:    if (filetype == 1) {
153:        pf = fopen(path, "r");
154:        if (!pf) {
155:            printf("file open error4\n");
156:            exit(1);
157:        }
158:        fscanf(pf, "%d", &record_num);
159:        head->totaldatanum = record_num;
160:        Node *pre = (Node *) malloc(sizeof(Node));
161:        fscanf(pf, "%d, %d, %d,\n", &pre->data[0], &pre-
 >data[1], &pre->data[2]);
162:        head->next = pre;
163:
```

```
164:        for (int i = 0; i < record_num; ++i) {
165:            Node *p = (Node *) malloc(sizeof(Node));
166:            fscanf(pf, "%d, %d, %d,\n", &p->data[0], &p-
 >data[1], &p->data[2]);
167:            pre->next = p;
168:            pre = p;
169:        }
170:        pre->next = NULL;
171:        fclose(pf);
172:    }
173:    else{
174:        pf = fopen(path, "rb");
175:        if(!pf)
176:        {
177:            printf("file open error4\n");
178:            exit(1);
179:        }
180:        fread(&record_num, 4, 1, pf);
181:
182:        Link * head = (Link*) malloc(sizeof(Link));
183:        head->totaldatanum = record_num;
184:        Node * pre = (Node*) malloc(sizeof (Node));
185:        fread(&pre->data[0], 4, 1, pf);
186:        fread(&pre->data[1], 4, 1, pf);
187:        fread(&pre->data[2], 4, 1, pf);
188:        head->next = pre;
189:
190:        for (int i = 0; i < record_num; ++i) {
191:            Node * p = (Node*) malloc(sizeof(Node));
192:            fread(&p->data[0], 4, 1, pf);
193:            fread(&p->data[1], 4, 1, pf);
194:            fread(&p->data[2], 4, 1, pf);
195:            pre->next = p;
196:            pre = p;
197:        }
198:        pre->next = NULL;
199:        fclose(pf);
200:        showlink(head->next);
201:        printf("before sort\n");
202:        show_link(head);
203:        printf("\nafter sort\n");
204:        bubblesortlink(head);
205:        show_link(head);
```

```
206:    }
207:}
```

2.8 模块化程序设计思想

模块化程序设计是指在进行程序设计时将一个大程序按照功能划 分为若干小程序模块，每个小程序模块完成一个确定的功能，并在这些模块之间建立必要的联系，通过模块的互相协作完成整个功能的程序设计方法。对于程序的二次开发，以下因素会对程序二次开发的效率产生影响：原有程序的健壮性、原有程序的模块化程度、二次程序与原有程序的功能重叠比例、二次程序相对原有程序的特殊功能的代码实现难易程度。

这个思想体现在计算机软件编程中直接体现就是函数的使用。函数的使用不仅节省了空间和编写的时间，实现了代码复用，而且方便了阅读，人们在阅读和使用时大部分时间只需要关注函数的功能和参数。但是在这次实验中，也发现因为函数接口和内部实现的问题造成了一些编写时间的耗费，例如，`qsort` 函数的传参数要求和 `lab5` 中编写调用 `qsort` 函数的函数内部参数还是有一点区别的，这里造成了一点小困扰（编写 `lab5` 时并没有后序二次开发的考量）。

由此我们可以看出，虽然我们现在拥有许多现成的、功能强大的库包，但是仍然需要了解一些 `stl` 函数中的具体实现过程；并且，统一形式的编写标准十分必要。

3.心得体会

3.1 自身收获

随着学习的深入，实验的要求也越发繁杂，但就算法要求本身而言其实并不是很难，主要就是各个功能之间的配合，也就是接口比较复杂，并且最后做菜单

的时候篇幅也比较长。这里我对工程任务的分解的重要性和接口的一般性的理解更加深入。首先是通读实验要求，明确任务，做出程序框图进行指引，最重要的是分解任务，以不同的函数去完成各个小任务，这样一次只关注一个小部分，完成了整体到局部的对立统一。还有必要性注释的书写，每隔几行或重要部位，写上几句注释，不管什么时候来看，都知道当时为什么这样写， 也便于后期的维护和优化。

这样的学习对之后的工程性代码的完成也有很大的助力。

3.2 小组互评

这次临近期中，小组成员其他杂事都比较多，导致实验进度缓慢。由此，我们应借助小组合作机制建立相互督促机制。

张开元同学的报告最后还是完成的比较好的，他的代码清晰，构成逻辑严密。因为是相互交流比较多，所以不借助注释也可以很快看明白。这恰恰说明小组合作的重要性。

王麒越同学的报告完成的很好，只是在变量命名上还不能做到见名知义，还需要一起多努力。