§13. 动态内存申请 – 顺序表适应不同的数据类型

1. 线性表的顺序表示的基本概念

用一组地址连续的存储单元依次存储线性表的数据元素，借助元素在存储器中的相对位置来表示元素间的逻辑关系

★ 假设线性表的每个元素需占用L个存储单元，并以所占的第一个单元的存储地址作为数据元素的起始存储位置，则线性表中第i+1个数据元素的存储位置Loc(a_{i+1})和第i个数据元素的存储位置Loc(a_i)之间满足下列关系：

$$Loc(a_{i+1}) = Loc(a_i) + L$$

- 形式化定义中线性表从1..n，C/C++中数组从0..n-1

★ 线性表的第i个元素a_i的存储位置和a_1的关系为：

$$Loc(a_i) = Loc(a_1) + (i-1)*L$$

★ a_1(表头元素)通常称作线性表的起始位置或基地址

★ 每个元素的存储位置和起始位置相差一个和数据元素在线性表中的位序成正比的常数（即L）

★ 只要确定了线性表的起始位置，即可随机存取表中任一元素

- 顺序表：顺序存储，随机存取
- 链表　：随机存储，顺序存取

★ C/C++语言中数组具备顺序存储的特点，但数组大小必须固定，因此不直接使用数组，而是用动态申请空间的方法模拟数组，方便线性表的扩大

§13. 动态内存申请 – 顺序表适应不同的数据类型

1. 线性表的顺序表示的基本概念

★ C/C++语言中数组具备顺序存储的特点，但数组大小必须固定，因此不直接使用数组，而是用动态申请空间的方法模拟数组，方便线性表的扩大

假设数据元素为int型，则：

（1）采用数组形式：

```
#define MAX_NUM 100
int a[MAX_NUM];
```

可用a[i]形式访问，当线性表中元素满100后，无法再增加，如果初始值设置很大，则会造成巨大的浪费

（2）采用C动态申请空间模拟数组形式：

```
#define MAX_NUM 100
int *a;
a = (int *)malloc(MAX_NUM * sizeof(int));
```

也可用a[i]形式访问，当线性表中元素满100后，可以再增加，方法：

```
a = (int *)realloc(a, (MAX_NUM+10)*sizeof(int));
```

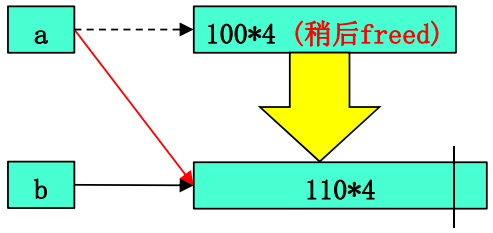
（3）采用C++的动态申请空间模拟数组形式：

```
#define MAX_NUM 100
int *a;
a = new int[MAX_NUM];
```

也可用a[i]形式访问，当线性表中元素满100后，可以再增加，方法：

```
int *b = new int[MAX_NUM+10]; //申请新空间
for(i=0; i<MAX_NUM; i++)  //原空间内容=>新空间
    b[i] = a[i];
delete a;  //释放原空间
a = b;     //原指针指向新空间
```

思考：如果新空间小于原空间，应如何？



§13. 动态内存申请 – 顺序表适应不[illegible]类型

2. 线性表顺序表示的基本操作的实现

2.1. C语言版

★ 线性表的数据类型

线性表允许存放任何类型的数据，不失一般性，讨论以下四种类型(六种形式)：

int ：整形数据

double ：浮点型数据

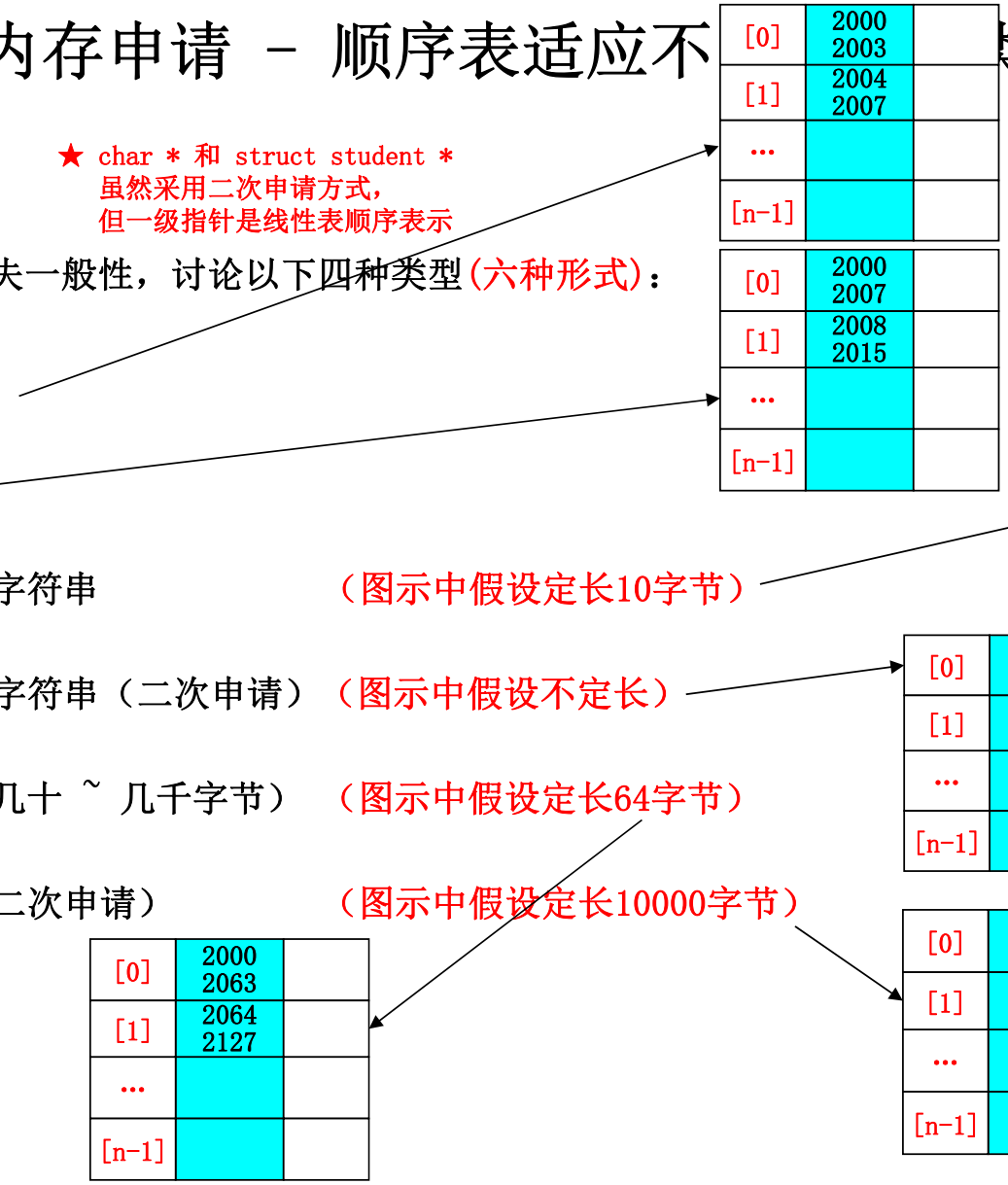
char [] ：定长或不定长字符串

char * ：定长或不定长字符串（二次申请）

struct student ：复杂结构体（几十 ~ 几千字节）

struct student *：复杂结构体（二次申请）

★ char * 和 struct student *
虽然采用二次申请方式，
但一级指针是线性表顺序表示



§13.动态内存申请 - 顺序表适应不同的数据类型

2.线性表顺序表示的基本操作的实现

2.1.C语言版

★ 程序的组成

◆ linear_list_sq.h	：头文件
◆ linear_list_sq.c	：具体实现
◆ linear_list_sq_main.c	：使用（测试）示例

说明：
- 从思维上把这个程序理解为两个人/小组完成，其中头文件(.h)和(_sq.c)看做一个人/小组的工作(基本功能的实现)，而将测试程序(_main.c)看做另一个人/小组在使用(用他人提供的基本操作函数来实现自己的应用目标)，两方层次不同(底层向上层提供支持)，适合团队合作和分工
- 假设为一个大型程序中的一个子集
- 程序实现后要进行详尽的测试，通过测试并稳定后，尽量不要修改(设计时尽量考虑得完全一些)
- 测试程序可以修改/调整，只要符合使用要求即可

§13.动态内存申请 – 顺序表适应不同的数据类型

2.线性表顺序表示的基本操作的实现

2.1.C语言版

★ 程序的组成

◆ linear_list_sq.h	：头文件
◆ linear_list_sq.c	：具体实现
◆ linear_list_sq_main.c	：使用（测试）示例

★ 算法与程序的区别

- 算法采用抽象数据接口，抽象数据操作
  => 程序必须有明确的数据定义以及数据操作方法
- 算法的返回值，错误处理都可以抽象
  => 程序必须明确且类型匹配
- 算法可以不定义主程序及配套函数
  => 程序必须补充完整
- 算法可以不定义临时变量
  => 程序必须补充完整

★ 与书上算法的区别

- C语言无引用，需要用指针代替
- 临时变量算法中无定义，程序要补齐
- 某些形式化定义和实际表示之间有区别

§13.动态内存申请 - 顺序表适应不同的数据类型

2.线性表顺序表示的基本操作的实现

2.1.C语言版

★ linear_list_sq.h 中各定义项的解析

/* linear_list_sq.h 的组成 */

```
#define TRUE          1
#define FALSE         0
#define OK            1
#define ERROR         0
#define INFEASIBLE   -1
#define OVERFLOW     -2

typedef int Status;
```

=> 先去看typedef部分的课件!!!

P.10 预定义常量和类型

(1) 预定义常量和类型：

```
// 函数结果状态代码
#define  TRUE         1
#define  FALSE        0
#define  OK           1
#define  ERROR        0
#define  INFEASIBLE   -1
#define  OVERFLOW     -2
// Status 是函数的类型,其值是函数结果状态代码
typedef  int  Status;
```

类型是int

```
/* linear_list_sq.h 的组成 */
#define LIST_INIT_SIZE     100 //初始大小为100(可按需修改)
#define LISTINCREMENT      10  //空间分配增量(可按需修改)

typedef struct {
    int *elem;              //存放动态申请空间的首地址
    int length;             //记录当前长度
    int listsize;           //当前分配的元素的个数
} sqlist;
```

```
/* 相当于两步
   1、先定义结构体类型
   2、用typedef声明为新类型 */
struct _sqlist_ {
    int *elem;
    int length;
    int listsize;
};

typedef struct _sqlist_ sqlist;
```

/* linear_list_sq.h 的组成 */

```
#define LIST_INIT_SIZE     100
#define LISTINCREMENT      10
typedef struct {
    int *elem;
    int length;
    int listsize;
} sqlist;
```

```
Status   InitList(sqlist *L);
Status   DestroyList(sqlist *L);
Status   ClearList(sqlist *L);
Status   ListEmpty(sqlist L);
int      ListLength(sqlist L);
Status   GetElem(sqlist L, int i, int *e);
int      LocateElem(sqlist L, int e, Status (*compare)(int e1, int e2));
Status   PriorElem(sqlist L, int cur_e, int *pre_e);
Status   NextElem(sqlist L, int cur_e, int *next_e);
Status   ListInsert(sqlist *L, int i, int e);
Status   ListDelete(sqlist *L, int i, int *e);
Status   ListTraverse(sqlist L, Status (*visit)(int e));
```

ADT List {
  数据对象：D = { a_i | a_i ∈ ElemSet, i = 1,2,…,n, n≥0 }
  数据关系：R1 = { <a_{i-1},a_i> | a_{i-1},a_i ∈ D, i = 2,…,n }
  基本操作：
    InitList(&L)
      操作结果：构造一个空的线性表 L。
    DestroyList(&L)
      初始条件：线性表 L 已存在。
      操作结果：销毁线性表 L。
    ClearList(&L)
      初始条件：线性表 L 已存在。
      操作结果：将 L 重置为空表。
    ListEmpty(L)
      初始条件：线性表 L 已存在。
      操作结果：若 L 为空表，则返回 TRUE，否则返回 FALSE。
    ListLength(L)
      初始条件：线性表 L 已存在。
      操作结果：返回 L 中数据元素个数。
    GetElem(L, i, &e)
      初始条件：线性表 L 已存在，1≤i≤ListLength(L)。
      操作结果：用 e 返回 L 中第 i 个数据元素的值。
    LocateElem(L, e, compare())
      初始条件：线性表 L 已存在，compare()是数据元素判定函数。
      操作结果：返回 L 中第 1 个与 e 满足关系 compare()的数据元素的位序。若这样的数据元素
                不存在，则返回值为 0。

Status InitList(sqlist *L);

算法转程序时：
★ 引用都表示为指针(C无引用)
★ 每个形参都要有类型定义

★ P.19-20 抽象数据类型定义转换为实际的C语言定义的函数原型说明
- 引用都表示为指针
- 每个形参都要有类型定义，其中compare和visit区别较大

问：当类型是double时，
需要做什么改变？

```
/* linear_list_sq.h 的组成 */
#define LIST_INIT_SIZE     100 //初始大小为100(可按需修改)
#define LISTINCREMENT      10  //空间分配增量(可按需修改)

typedef struct {
    double *elem;          //存放动态申请空间(当数组用)的首地址
    int length;            //记录当前长度
    int listsize;          //当前分配的元素的个数
} sqlist;

Status   InitList(sqlist *L);
Status   DestroyList(sqlist *L);
Status   ClearList(sqlist *L);
Status   ListEmpty(sqlist L);
int      ListLength(sqlist L);
Status   GetElem(sqlist L, int i, double *e);
int      LocateElem(sqlist L, double e, Status (*compare)(double e1, double e2));
Status   PriorElem(sqlist L, double cur_e, double *pre_e);
Status   NextElem(sqlist L, double cur_e, double *next_e);
Status   ListInsert(sqlist *L, int i, double e);
Status   ListDelete(sqlist *L, int i, double *e);
Status   ListTraverse(sqlist L, Status (*visit)(double e));
```

为什么不用换？

问：当类型是char[]时，
需要做什么改变？

```
/* linear_list_sq.h 的组成 */
#define LIST_INIT_SIZE     100 //初始大小为100(可按需修改)
#define LISTINCREMENT      10  //空间分配增量(可按需修改)

typedef struct {
    char (*elem)[10];      //存放动态申请空间(当数组用)的首地址
    int length;            //记录当前长度
    int listsize;          //当前分配的元素的个数
} sqlist;

Status   InitList(sqlist *L);
Status   DestroyList(sqlist *L);
Status   ClearList(sqlist *L);
Status   ListEmpty(sqlist L);
int      ListLength(sqlist L);
Status   GetElem(sqlist L, int i, char *e);
int      LocateElem(sqlist L, char *e, Status (*compare)(char *e1, char *e2));
Status   PriorElem(sqlist L, char *cur_e, char *pre_e);
Status   NextElem(sqlist L, char *cur_e, char *next_e);
Status   ListInsert(sqlist *L, int i, char *e);
Status   ListDelete(sqlist *L, int i, char *e);
Status   ListTraverse(sqlist L, Status (*visit)(char *e));
```

```
double cur_e, double *pre_e);
main:
double d1=5.2, d2;
PriorElem(L,  d, &d2);

main:
char s1[10], s2[10];
PriorElem(L, s1,  s2);

问题1：调用方式不一致，是否正确？
```

问：当类型是char[]时，
需要做什么改变？

```
/* linear_list_sq.h 的组成 */
#define LIST_INIT_SIZE     100 //初始大小为100(可按需修改)
#define LISTINCREMENT      10  //空间分配增量(可按需修改)

typedef struct {
    char (*elem)[10];      //存放动态申请空间(当数组用)的首地址
    int length;            //记录当前长度
    int listsize;          //当前分配的元素的个数
} sqlist;

Status   InitList(sqlist *L);
Status   DestroyList(sqlist *L);
Status   ClearList(sqlist *L);
Status   ListEmpty(sqlist L);
int      ListLength(sqlist L);
Status   GetElem(sqlist L, int i, char (*e)[10]);
int      LocateElem(sqlist L, char *e, Status (*compare)(char *e1, char *e2));
Status   PriorElem(sqlist L, char *cur_e, char (*pre_e)[10]);
Status   NextElem(sqlist L, char *cur_e, char (*next_e)[10]);
Status   ListInsert(sqlist *L, int i, char *e);
Status   ListDelete(sqlist *L, int i, char (*e)[10]);
Status   ListTraverse(sqlist L, Status (*visit)(char *e));
```

```
double cur_e, double *pre_e);
main:
double d1=5.2, d2;
PriorElem(L,  d, &d2);

main:
char s1[10], s2[10];
PriorElem(L, s1, &s2);

问题2：若要求保持一致，如何做？
```

问：当类型是char *时，
需要做什么改变？

```
/* linear_list_sq.h 的组成 */
#define LIST_INIT_SIZE     100 //初始大小为100(可按需修改)
#define LISTINCREMENT      10  //空间分配增量(可按需修改)

typedef struct {
    char **elem;           //存放动态申请空间(当数组用)的首地址
    int length;            //记录当前长度
    int listsize;          //当前分配的元素的个数
} sqlist;

Status   InitList(sqlist *L);
Status   DestroyList(sqlist *L);
Status   ClearList(sqlist *L);
Status   ListEmpty(sqlist L);
int      ListLength(sqlist L);
Status   GetElem(sqlist L, int i, char **e);
int      LocateElem(sqlist L, char *e, Status (*compare)(char *e1, char *e2));
Status   PriorElem(sqlist L, char *cur_e, char **pre_e);
Status   NextElem(sqlist L, char *cur_e, char **next_e);
Status   ListInsert(sqlist *L, int i, char *e);
Status   ListDelete(sqlist *L, int i, char **e);
Status   ListTraverse(sqlist L, Status (*visit)(char *e));
```

```
/* linear_list_sq.h 的组成 */
#define LIST_INIT_SIZE     100 //初始大小为100(可按需修改)
#define LISTINCREMENT      10  //空间分配增量(可按需修改)
struct student {
    ...
};
typedef struct {
    struct student *elem;  //存放动态申请空间(当数组用)的首地址
    int length;            //记录当前长度
    int listsize;          //当前分配的元素的个数
} sqlist;
Status   InitList(sqlist *L);
Status   DestroyList(sqlist *L);
Status   ClearList(sqlist *L);
Status   ListEmpty(sqlist L);
int      ListLength(sqlist L);
Status   GetElem(sqlist L, int i, struct student *e);
int      LocateElem(sqlist L, struct student e, Status (*compare)( struct student e1, struct student e2));
Status   PriorElem(sqlist L, struct student cur_e, struct student *pre_e);
Status   NextElem(sqlist L, struct student cur_e, struct student *next_e);
Status   ListInsert(sqlist *L, int i, struct student e);
Status   ListDelete(sqlist *L, int i, struct student *e);
Status   ListTraverse(sqlist L, Status (*visit)(struct student e));
```

问：当类型是struct student时，需要做什么改变？

注：纯C编译器，struct student e 不能写成 student e

问：当类型是struct student *时，需要做什么改变？

注：纯C编译器，struct student e 不能写成 student e

```
/* linear_list_sq.h 的组成 */
#define LIST_INIT_SIZE     100 //初始大小为100(可按需修改)
#define LISTINCREMENT      10  //空间分配增量(可按需修改)
struct student {
    ...
};
typedef struct {
    struct student **elem;//存放动态申请空间(当数组用)的首地址
    int length;           //记录当前长度
    int listsize;         //当前分配的元素的个数
} sqlist;
Status   InitList(sqlist *L);
Status   DestroyList(sqlist *L);
Status   ClearList(sqlist *L);
Status   ListEmpty(sqlist L);
int      ListLength(sqlist L);
Status   GetElem(sqlist L, int i, struct student **e);
int      LocateElem(sqlist L, struct student *e, Status (*compare)( struct student *e1, struct student *e2));
Status   PriorElem(sqlist L, struct student *cur_e, struct student **pre_e);
Status   NextElem(sqlist L, struct student *cur_e, struct student **next_e);
Status   ListInsert(sqlist *L, int i, struct student *e);
Status   ListDelete(sqlist *L, int i, struct student **e);
Status   ListTraverse(sqlist L, Status (*visit)(struct student *e));
```

```
/* linear_list_sq.h 的组成 */
#define LIST_INIT_SIZE     100 //初始大小为100(可按需修改)
#define LISTINCREMENT      10  //空间分配增量(可按需修改)


typedef struct {
    int *elem;     //存放动态申请空间(当数组用)的首地址
    int length;    //记录当前长度
    int listsize; //当前分配的元素的个数
} sqlist;
```

问：当类型是
int
double
char[]
char *
struct student
struct student *
时，能否使改动尽可能少？

答：在数据结构中一般不讨论具体类型，引入一个通用类型（Elemtype）来表示元素的类型即可

```
// - - - - - 线性表的动态分配顺序存储结构 - - - - -
#define  LIST_INIT_SIZE     100   // 线性表存储空间的初始分配量
#define  LISTINCREMENT      10    // 线性表存储空间的分配增量
typedef  struct {
    ElemType   * elem;          // 存储空间基址
    int        length;          // 当前长度
    int        listsize;        // 当前分配的存储容量(以 sizeof(ElemType)为单位)
}SqList;
```

P. 22

```
/* linear_list_sq.h 的组成 */
#define LIST_INIT_SIZE     100 //初始大小为100(可按需修改)
#define LISTINCREMENT      10  //空间分配增量(可按需修改)

typedef int ElemType;          //算法到程序的补充

typedef struct {
    ElemType *elem;            //存放动态申请空间的首地址
    int length;                //记录当前长度
    int listsize; //当前分配的元素的个数
} sqlist;

Status   InitList(sqlist *L);
Status   DestroyList(sqlist *L);
Status   ClearList(sqlist *L);
Status   ListEmpty(sqlist L);
int      ListLength(sqlist L);
Status   GetElem(sqlist L, int i, ElemType *e);
int      LocateElem(sqlist L, ElemType e, Status (*compare)(ElemType e1, ElemType e2));
Status   PriorElem(sqlist L, ElemType cur_e, ElemType *pre_e);
Status   NextElem(sqlist L, ElemType cur_e, ElemType *next_e);
Status   ListInsert(sqlist *L, int i, ElemType e);
Status   ListDelete(sqlist *L, int i, ElemType *e);
Status   ListTraverse(sqlist L, Status (*visit)(ElemType e));
```

问：当类型不同时，能否使改动尽可能少？

答：在数据结构中一般不讨论具体类型，引入一个通用类型（Elemtype）来表示元素的类型即可

问：算法转为程序时，如何对应实际类型？

答：用typedef声明新类型的方法来实现实际类型和通用类型间的映射

```
/* linear_list_sq.h 的组成 */

  struct student {
      int   num;       //设学号为主关键字
      char  name[10];
      char  sex;
      float score;
      char  addr[30];
  };  //算上填充，共52字节

  //typedef int ElemType;
  typedef double ElemType;
  //typedef char ElemType[10];
  //typedef char* ElemType;
  //typedef struct student ElemType;
  //typedef struct student* ElemType;

  typedef struct {
      ElemType *elem;
      int length;
      int listsize;
  } sqlist;
```

问：当类型是
int
double
char[]
char *
struct student
struct student *
时，需要做什么改变？

答：实际使用时，6选1即可(只能打开其中一项，否则错)，函数声明部分不同类型完全一致

```
Status InitList(sqlist *L);
Status DestroyList(sqlist *L);
Status ClearList(sqlist *L);
Status ListEmpty(sqlist L);
int    ListLength(sqlist L);
Status GetElem(sqlist L, int i, ElemType *e);
int    LocateElem(sqlist L, ElemType e, Status (*compare)(ElemType e1, ElemType e2));
Status PriorElem(sqlist L, ElemType cur_e, ElemType *pre_e);
Status NextElem(sqlist L, ElemType cur_e, ElemType *next_e);
Status ListInsert(sqlist *L, int i, ElemType e);
Status ListDelete(sqlist *L, int i, ElemType *e);
Status ListTraverse(sqlist L, Status (*visit)(ElemType e));
```

不同类型一致，无任何变化

§13.动态内存申请 - 顺序表适应不同的数据类型

2.线性表顺序表示的基本操作的实现

2.1.C语言版

★ linear_list_sq.h 中各定义项的解析

★ linear_list_sq.c 中各函数的具体实现

ElemType => int

```
/* linear_list_sq.c 的实现 */
#include <stdio.h>
#include <stdlib.h>                    //malloc/realloc函数
#include <unistd.h>                    //exit函数
#include "linear_list_sq.h"            //形式定义

/* 初始化线性表 */
Status InitList(sqlist *L)
{
    L->elem = (ElemType *)malloc(LIST_INIT_SIZE * sizeof(ElemType));
    if (L->elem == NULL)
        exit(OVERFLOW);
    L->length = 0;
    L->listsize = LIST_INIT_SIZE;

    return OK;
}
```

★ main函数中
声明为 sqlist L;
调用为 InitList(&L);

★ 形参为指针，因为函数中要改变并返回
★ 书上为引用，因此 L.elem形式应变为L->elem形式

ElemType => int

```
/* linear_list_sq.c 的实现 */
#include <stdio.h>
#include <stdlib.h>                    //malloc/realloc函数
#include <unistd.h>                    //exit函数
#include "linear_list_sq.h"            //形式定义

/* 初始化线性表 */
Status InitList(sqlist L)
{
    L.elem = (ElemType *)malloc(LIST_INIT_SIZE * sizeof(ElemType));
    if (L.elem == NULL)
        exit(OVERFLOW);
    L.length = 0;
    L.listsize = LIST_INIT_SIZE;

    return OK;
}
```

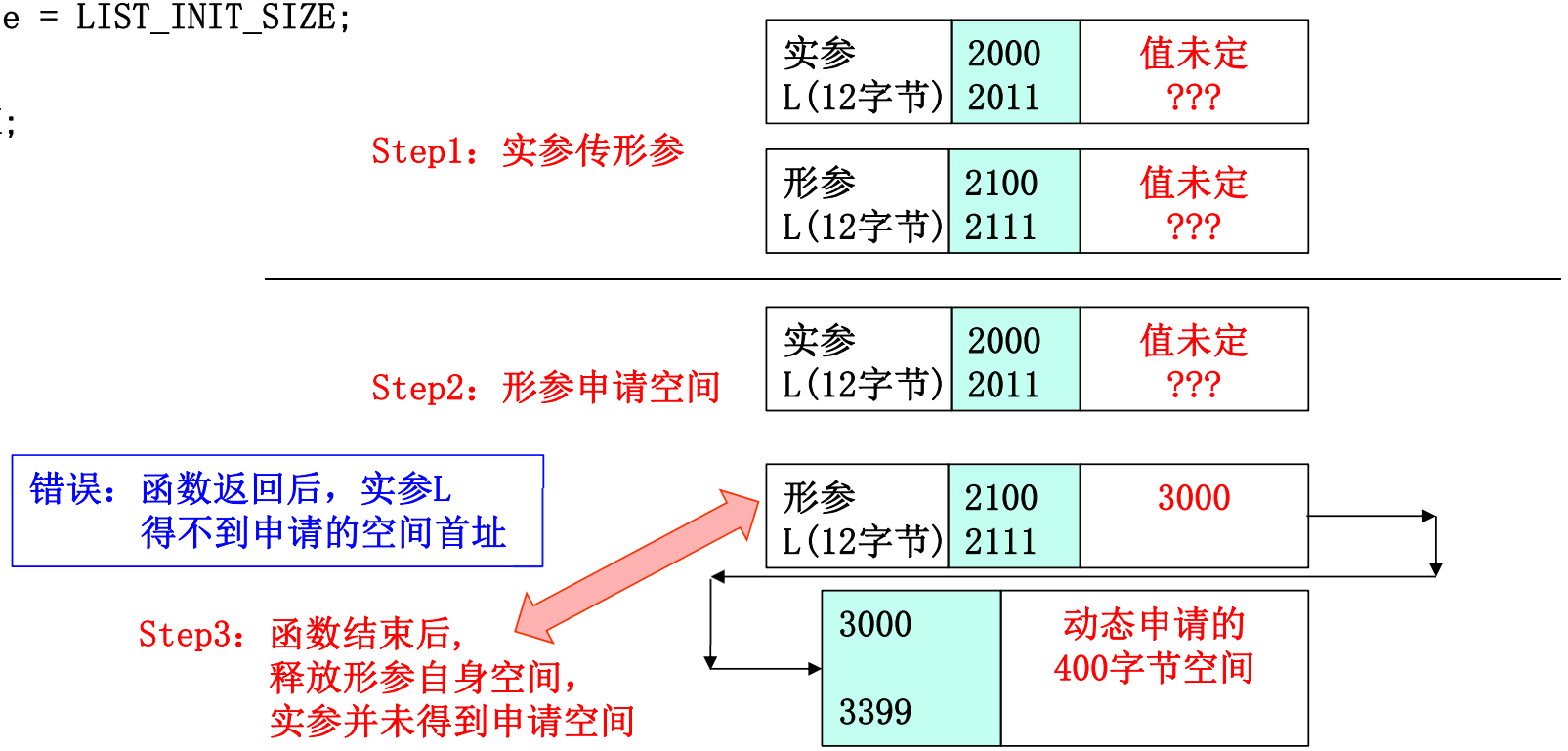
★ main函数中
声明为 sqlist L;
调用为 InitList(L);

★ 形参为sqlist结构体变量，函数实现中
L-> 均改为 L. 是否正确？为什么？

```
/* linear_list_sq.c 的实现 */
/* 初始化线性表 */
Status InitList(sqlist L)
{
    L.elem = (ElemType *)malloc(LIST_INIT_SIZE * sizeof(ElemType));
    if (L.elem == NULL)
        exit(OVERFLOW);
    L.length = 0;
    L.listsize = LIST_INIT_SIZE;

    return OK;
}
```

错误分析

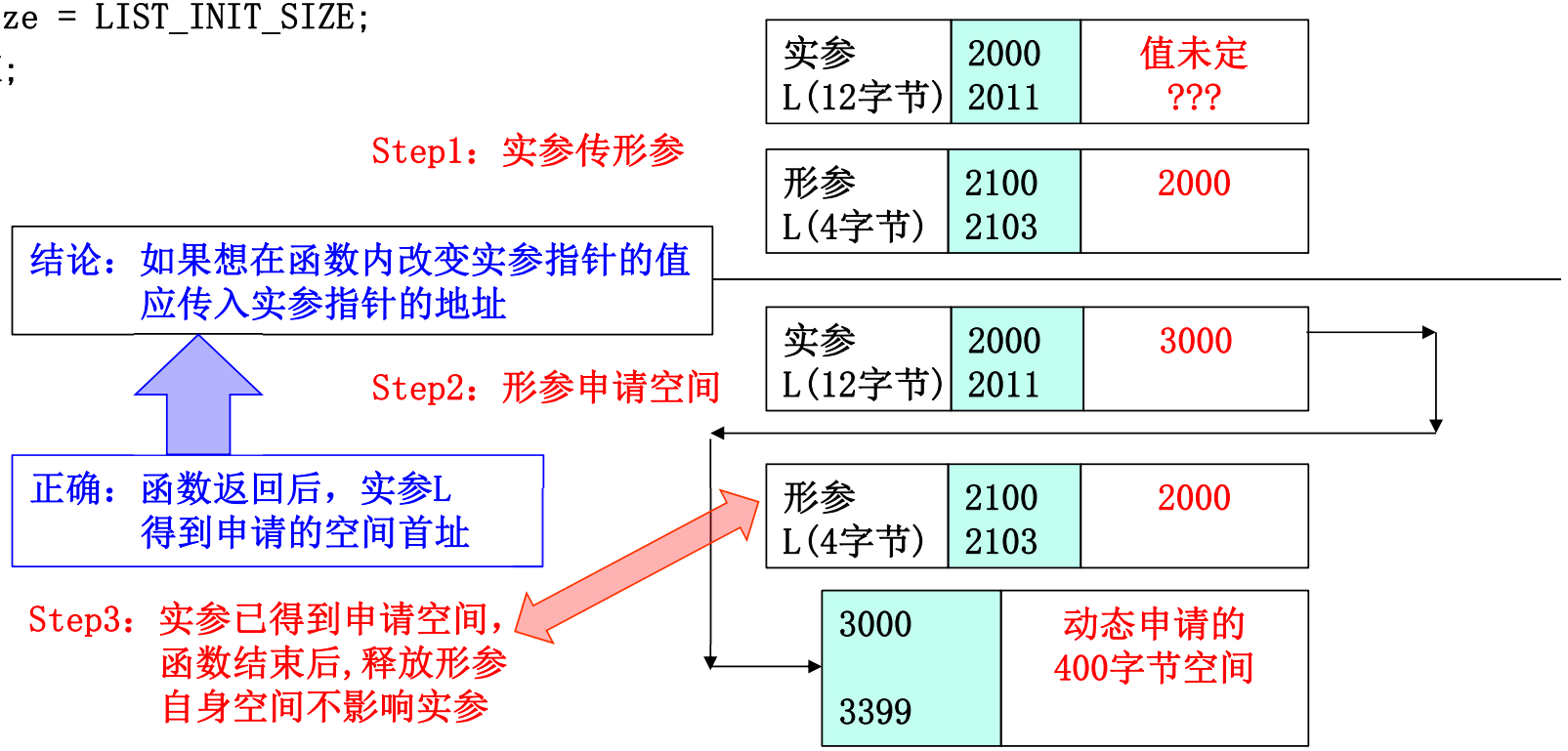
★ main函数中
声明为 sqlist L;
调用为 InitList(L);



```
/* linear_list_sq.c 的实现 */
/* 初始化线性表 */
Status InitList(sqlist *L)
{
    L->elem = (ElemType *)malloc(LIST_INIT_SIZE * sizeof(ElemType));
    if (L->elem == NULL)
        exit(OVERFLOW);
    L->length = 0;
    L->listsize = LIST_INIT_SIZE;
    return OK;
}
```

正确分析

★ main函数中
声明为 sqlist L;
调用为 InitList(&L);



```
/* linear_list_sq.c 的实现 */
#include <stdio.h>
#include <stdlib.h>                    //malloc/realloc函数
#include <unistd.h>                    //exit函数
#include "linear_list_sq.h"            //形式定义

/* 初始化线性表 */
Status InitList(sqlist *L)
{
    L->elem = (ElemType *)malloc(LIST_INIT_SIZE * sizeof(ElemType));
    if (L->elem == NULL)
        exit(OVERFLOW);
    L->length = 0;
    L->listsize = LIST_INIT_SIZE;

    return OK;
}
```

★ main函数中
声明为 sqlist L;
调用为 InitList(&L);

★ 形参为指针，因为函数中要改变并返回
★ 书上为引用，因此 L.elem形式应变为L->elem形式

问：当类型是
double
char[]
char *
struct student
struct student *
时，需要做什么改变？

答：不需要任何变化!!!

ElemType => int

```
/* linear_list_sq.c 的实现 */

/* 销毁线性表 */
Status DestroyList(sqlist *L)
{
    /* 未执行 InitList，直接执行本函数，
       则可能出错，因为指针初值未定 */
    if (L->elem)
        free(L->elem);
    L->length   = 0; //可以不要
    L->listsize = 0; //可以不要

    return OK;
}
```

问：当类型是
- double
- char[]
- char *
- struct student
- struct student *

时，需要做什么改变？

/* linear_list_sq.c 的实现 */

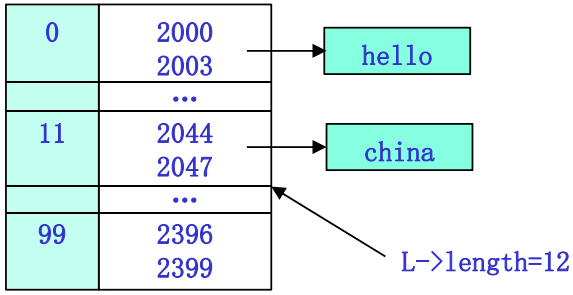
```
/* 销毁线性表 */
Status DestroyList(sqlist *L)
{



    /* 未执行 InitList，直接执行本函数，
       则可能出错，因为指针初值未定 */
    if (L->elem)
        free(L->elem);
    L->length   = 0; //可以不要
    L->listsize = 0; //可以不要

    return OK;
}
```

```
/* 销毁线性表 */
Status DestroyList(sqlist *L)
{
    int i;
    /* 首先释放二次申请空间 */
    for(i=0; i<L->length; i++)
        free(L->elem[i]);

    /* 未执行 InitList，直接执行本函数，
       则可能出错，因为指针初值未定 */
    if (L->elem)
        free(L->elem);
    L->length   = 0; //可以不要
    L->listsize = 0; //可以不要

    return OK;
}
```

答：当类型是
double
char[]
struct student时，
不需要任何变化!!!

问：当类型是
double
char[]
char *
struct student
struct student *
时，需要做什么改变？

答：当类型是
char *
struct student *时，
要首先释放二次申请空间



/* linear_list_sq.c 的实现 */

```
/* 销毁线性表 */
Status DestroyList(sqlist *L)
{



    /* 未执行 InitList，直接执行本函数，
       则可能出错，因为指针初值未定 */
    if (L->elem)
        free(L->elem);
    L->length   = 0; //可以不要
    L->listsize = 0; //可以不要

    return OK;
}
```

```
/* 销毁线性表 */
Status DestroyList(sqlist *L)
{
    int i;
    /* 首先释放二次申请空间 */
    for(i=0; i<L->length; i++)
        free(L->elem[i]);

    /* 未执行 InitList，直接执行本函数，
       则可能出错，因为指针初值未定 */
    if (L->elem)
        free(L->elem);
    L->length   = 0; //可以不要
    L->listsize = 0; //可以不要

    return OK;
}
```

类型为 char * 和 struct student *时，此处打开，其它类型时注释掉

问：如何处理具体类型不同时的代码差异？
答：按需注释/非注释

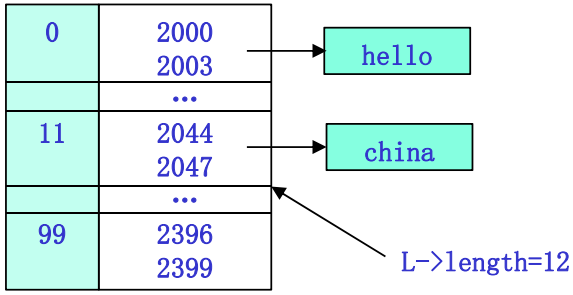
续问：有没有更好的方法？
续答：编译预处理 - 条件编译

请先去查看"条件编译"部分的课件

答：当类型是 double、char[]、struct student时，不需要任何变化!!!

问：当类型是 double、char[]、char *、struct student、struct student *时，需要做什么改变？

答：当类型是 char *、struct student *时，要首先释放二次申请空间



§13.动态内存申请 - 顺序表适应不同的数据类型

2.线性表顺序表示的基本操作的实现

2.1.C语言版

★ linear_list_sq.h 中各定义项的解析

★ linear_list_sq.c 中各函数的具体实现

(前面全部废弃，用条件编译的方法完整实现六合一程序)

=> 先去看条件编译部分课件!!!

六合一

```
/* linear_list_sq.h 的组成 */
//#define ELEMTYPE_IS_INT           //不定义也行
//#define ELEMTYPE_IS_DOUBLE
//#define ELEMTYPE_IS_CHAR_ARRAY
//#define ELEMTYPE_IS_CHAR_P
//#define ELEMTYPE_IS_STRUCT_STUDENT
//#define ELEMTYPE_IS_STRUCT_STUDENT_P

/* P.10 的预定义常量和类型 */
#define TRUE              1
#define FALSE             0
#define OK                1
#define ERROR             0
#define INFEASIBLE       -1
#define LOVERFLOW        -2  //因为<math.h>中已有 OVERFLOW ，因此换一下

typedef int Status;
```

定义6个宏定义，
目前全部disable，
使用时按需enable即可
每次只能enable一个!!!

六合一

```
/* linear_list_sq.h 的组成 */
#define LIST_INIT_SIZE  100  //初始大小定义为100（可按需修改）
#define LISTINCREMENT   10   //若空间不够，每次增长10（可按需修改）

#ifdef ELEMTYPE_IS_DOUBLE
    typedef double ElemType;
#elif defined (ELEMTYPE_IS_CHAR_ARRAY)
    typedef char ElemType[10];
#elif defined (ELEMTYPE_IS_CHAR_P)
    typedef char* ElemType;
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    typedef struct student {
            int   num;
            char  name[10];
            char  sex;
            float score;
            char  addr[30];
    } ElemType;
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    typedef struct student {
            int   num;
            char  name[10];
            char  sex;
            float score;
            char  addr[30];
    } ET, *ElemType;  //此处为什么多一个ET类型的声明？后面讲
#else       //缺省当做int处理
    typedef int ElemType;
#endif
```

根据不同的宏定义决定ElemType的实际类型

六合一

```
/* linear_list_sq.h 的组成 */
#define LIST_INIT_SIZE   100   //初始大小定义为100（可按需修改）
#define LISTINCREMENT    10    //若空间不够，每次增长10（可按需修改）

#ifdef ELEMTYPE_IS_DOUBLE
    ......
#endif

typedef struct {
    ElemType *elem;      //存放动态申请空间的首地址
    int length;          //记录当前长度
    int listsize;        //当前分配的元素的个数
} sqlist;

Status      InitList(sqlist *L);
Status      DestroyList(sqlist *L);
Status      ClearList(sqlist *L);
Status      ListEmpty(sqlist L);
int         ListLength(sqlist L);
Status      GetElem(sqlist L, int i, ElemType *e);
int         LocateElem(sqlist L, ElemType e, Status (*compare)(ElemType e1, ElemType e2));
Status      PriorElem(sqlist L, ElemType cur_e, ElemType *pre_e);
Status      NextElem(sqlist L, ElemType cur_e, ElemType *next_e);
Status      ListInsert(sqlist *L, int i, ElemType e);
Status      ListDelete(sqlist *L, int i, ElemType *e);
Status      ListTraverse(sqlist L, Status (*visit)(ElemType e));
```

不同类型一致

不同类型一致

★ 引用都表示为指针

★ 每个形参都要有类型定义，其中compare和visit区别较大

六合一

```
/* linear_list_sq.c 的实现 */
#include <stdio.h>
#include <stdlib.h>          //malloc/realloc函数
#include <unistd.h>          //exit函数
#include <math.h>            //fabs函数
#include <string.h>          //strcpy/strcmp等函数
#include "linear_list_sq.h"  //形式定义

/* 初始化线性表 */
Status InitList(sqlist *L)
{
    L->elem = (ElemType *)malloc(LIST_INIT_SIZE * sizeof(ElemType));
    if (L->elem == NULL)
        exit(LOVERFLOW);
    L->length = 0;
    L->listsize = LIST_INIT_SIZE;

    return OK;
}
```

把各种数据类型需要的库函数一起包含进来

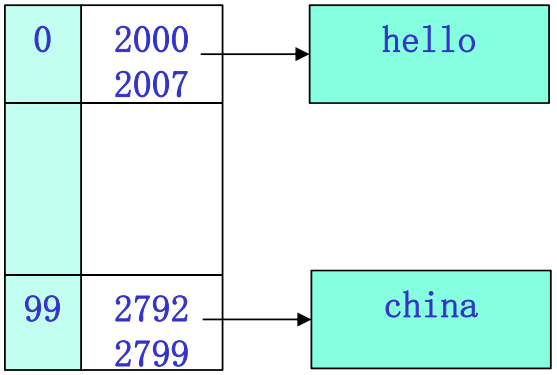
所有数据类型的处理方法都相同无变化

六合一

```
/* linear_list_sq.c 的实现 */

/* 销毁线性表 */
Status DestroyList(sqlist *L)
{
   /* 两种指针类型需要释放二级空间 */
#if defined (ELEMTYPE_IS_CHAR_P) || defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    int i;

    /* 首先释放二级空间 */
    for(i=0; i<L->length; i++)
        free(L->elem[i]);
#endif

    /* 若未执行 InitList，直接执行本函数，则可能出错 */
    if (L->elem)
        free(L->elem);
    L->length   = 0; //可不要
    L->listsize = 0; //可不要

    return OK;
}
```

两种数据类型的
特殊处理方法
其它四种无

六种数据类型的
公共部分处理部分



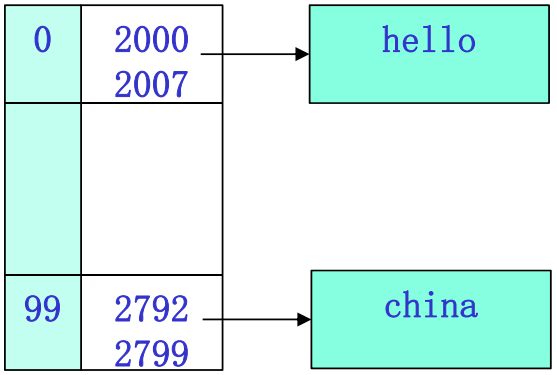
/* linear_list_sq.c 的实现 */

六合一

```
/* 清除线性表（已初始化，不释放空间，只清除内容） */
Status ClearList(sqlist *L)
{
   /* 两种指针类型需要释放二级空间 */
#if defined (ELEMTYPE_IS_CHAR_P) || defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    int i;

    /* 首先释放二级空间 */
    for(i=0; i<L->length; i++)
        free(L->elem[i]);
#endif

    L->length = 0;

    return OK;
}
```

两种数据类型的
特殊处理方法
其它四种无

六种数据类型的
公共部分处理部分



/* linear_list_sq.c 的实现 */

六合一


```
/* 判断是否为空表 */
Status ListEmpty(sqlist L)
{
    if (L.length == 0)
        return TRUE;
    else
        return FALSE;
}
```

所有数据类型的
处理方法都相同
无变化

/* linear_list_sq.c 的实现 */

六合一

```
/* 求表的长度 */
int ListLength(sqlist L)
{
    return L.length;
}
```

所有数据类型的处理方法都相同无变化

/* linear_list_sq.c 的实现 */

六合一

```
/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType *e)
{
    /* i值合理范围[1..length]  */
    if (i<1 || i>L.length)
        return ERROR;

    e <= L.elem[i-1];

    return OK;
}
```

具体讨论不同数据类型的
不同处理方法

/* linear_list_sq.c 的实现 */

六合一： ElemType是int/double

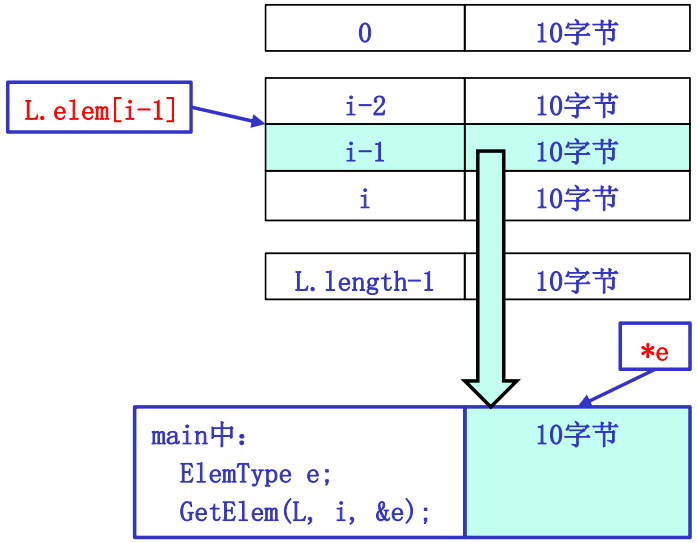
```
/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType *e)
{
    /* i值合理范围[1..length]  */
    if (i<1 || i>L.length)
        return ERROR;

    *e = L.elem[i-1];       //下标从0开始，第i个实际在elem[i-1]中
    return OK;
}
```

/* linear_list_sq.c 的实现 */

六合一： ElemType是char[]

```
/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType *e)
{
    /* i值合理范围[1..length]  */
    if (i<1 || i>L.length)
        return ERROR;

    strcpy(*e, L.elem[i-1]);
    return OK;
}
```

因为：typedef char Elemtype[10];
所以：Elemtype e => char e[10];
Elemtype *e => char (*e)[10];
e是指向有10个字符组成的一维字符数组的指针
*e指向一维字符数组的首字符的指针



加深理解

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

void f(char x[10])
{
    printf("%d %d\n", sizeof(x), sizeof(*x));
    strcpy(x, "hello");
}
int main()
{
    char s[10];
    f(s);
    printf("%s\n", s);

    return 0;
}
```

char x[10]
char *x
char x[任意数字]

4 1

s 2000 2009 10个字符

x 2100 2103 2000

s是2000，表示2000里存放一个字符
&s还是2000，表示2000里存放一个大小为10的一维数组

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

void f(char (*x)[10])
{
    printf("%d %d\n", sizeof(x), sizeof(*x));
    strcpy(*x, "hello");
}
int main()
{
    char s[10];
    f(&s);
    printf("%s\n", s);

    return 0;
}
```

4 10

&s 2000 2009 1个10字节的一维数组

x 2100 2103 2000

/* linear_list_sq.c 的实现 */

六合一： ElemType是char[]

```
/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType *e)
{
    /* i值合理范围[1..length]  */
    if (i<1 || i>L.length)
        return ERROR;

    strcpy(*e, L.elem[i-1]);
    return OK;
}
```

因为：typedef char Elemtype[10];
所以：Elemtype e => char e[10];
Elemtype *e => char (*e)[10];
e是指向有10个字符组成的一维字符数组的指针
*e指向一维字符数组的首字符的指针

main中：

```
ElemType e;
GetElem(L, i, &e);
```

```
/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType e)
{
    /* i值合理范围[1..length]  */
    if (i<1 || i>L.length)
        return ERROR;

    strcpy(e, L.elem[i-1]);
    return OK;
}
```

main中：

```
ElemType e;
GetElem(L, i, e);
```

如果形参写成ElemType e，也是可以的

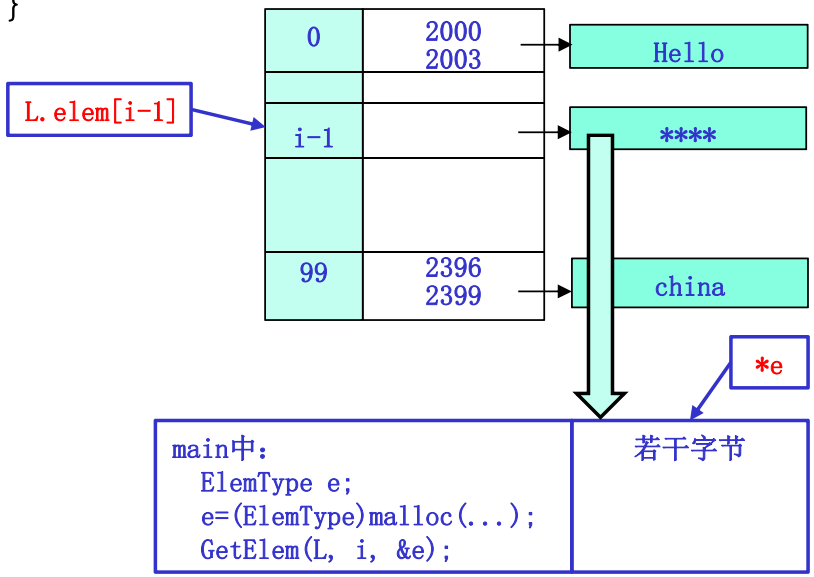
但为了保持不同类型下GetElem的声明统一性，仍使用ElemType *e

/* linear_list_sq.c 的实现 */

六合一： ElemType是char *

```
/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType *e)
{
    /* i值合理范围[1..length]  */
    if (i<1 || i>L.length)
        return ERROR;

    strcpy(*e, L.elem[i-1]);
    return OK;
}
```

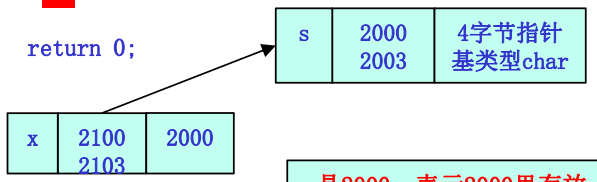
因为：typedef char* Elemtype;
所以：Elemtype e => char *e;
Elemtype *e => char **e;
e是基类型为char *的指针
*e是基类型为char的指针



加深理解

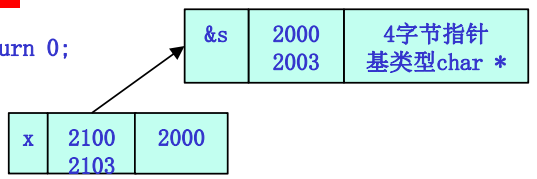
```
#include <stdio.h>
#include <string.h>

void f(char *x)
{
    printf("%d %d\n", sizeof(x), sizeof(*x));
    return;
}

int main()
{
    char *s;
    f(s);

    return 0;
}
```

4 1




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

void f(char **x)
{
    printf("%d %d\n", sizeof(x), sizeof(*x));
    return;
}

int main()
{
    char *s;
    f(&s);

    return 0;
}
```

4 4



s是2000，表示2000里存放一个指针
&s还是2000，表示2000里存放一个指针的指针

/* linear_list_sq.c 的实现 */

六合一： ElemType是char *

```
/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType *e)
{
    /* i值合理范围[1..length]  */
    if (i<1 || i>L.length)
        return ERROR;

    strcpy(*e, L.elem[i-1]);
    return OK;
}
```

因为：typedef char* Elemtype;
所以：Elemtype e => char *e;
Elemtype *e => char **e;
e是基类型为char *的指针
*e是基类型为char的指针

```
main中：
  ElemType e;
  e=(ElemType)malloc(...);
  GetElem(L, i, &e);
```

```
/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType e)
{
    /* i值合理范围[1..length]  */
    if (i<1 || i>L.length)
        return ERROR;

    strcpy(e, L.elem[i-1]);
    return OK;
}
```

```
main中：
  ElemType e;
  e=(ElemType)malloc(...);
  GetElem(L, i, e);
```

如果形参写成ElemType e，也是可以的

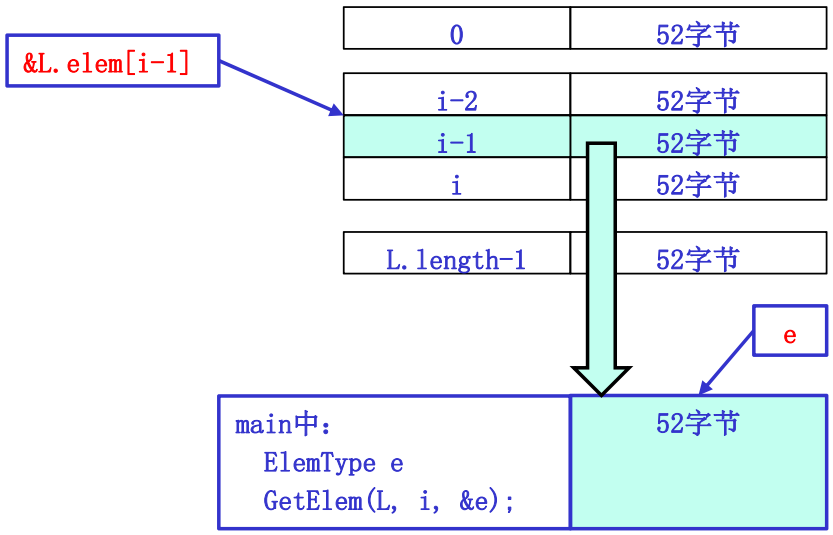
但为了保持不同类型下GetElem的声明统一性，仍使用ElemType *e

```
/* linear_list_sq.c 的实现 */

/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType *e)
{
    /* i值合理范围[1..length]  */
    if (i<1 || i>L.length)
        return ERROR;

    memcpy(e, &(L.elem[i-1]), sizeof(ElemType));
    return OK;
}
```

六合一： ElemType是struct student
假设占用52字节

main中：
```
ElemType e;
GetElem(L, i, &e);
```

memcpy函数的使用：
void *memcpy(void *dest, const void *src, int n);
将从源地址开始的n个字节复制到目标地址中
★ 整体内存拷贝，不论中间是否有尾零
★ 内存理解同char型数组但无法保证尾零，因此不能用strcpy



/* linear_list_sq.c 的实现 */

六合一： ElemType是struct student *

```
/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType *e)
{
    /* i值合理范围[1..length]  */
    if (i<1 || i>L.length)
        return ERROR;

    memcpy(*e, L.elem[i-1], sizeof(ET));
    return OK;
}
```

ElemType是struct student时：

```
memcpy(e, &(L.elem[i-1]), sizeof(ElemType));
```

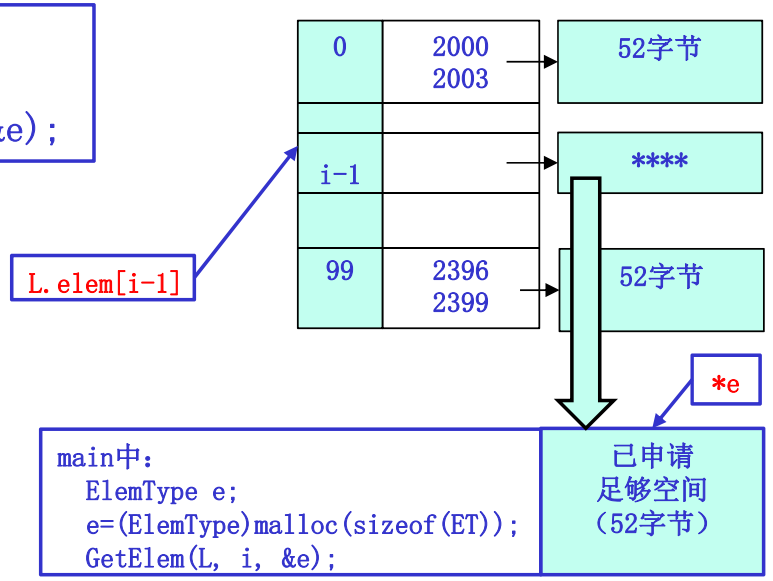
```
typedef struct student {
    int   num;
    char  name[10];
    char  sex;
    float score;
    char  addr[30];
} ET, *ElemType; //此处为什么多个ET类型的声明？后面讲
```

main中：

```
ElemType e;
e=(ElemType)malloc(sizeof(ET));
GetElem(L, i, &e);
```

main中：

```
ElemType e;
GetElem(L, i, &e);
```



六合一

```
/* linear_list_sq.c 的实现 */
/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType *e)
{
    if (i<1 || i>L.length)
        return ERROR;

#if defined (ELEMTYPE_IS_CHAR_ARRAY) || defined (ELEMTYPE_IS_CHAR_P)
    strcpy(*e, L.elem[i-1]);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    memcpy(e, &(L.elem[i-1]), sizeof(ElemType));
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    memcpy(*e, L.elem[i-1], sizeof(ET));
#else    //int和double直接赋值
    *e = L.elem[i-1];
#endif

    return OK;
}
```

不同数据类型的
不同处理方法

/* linear_list_sq.c 的实现 */

六合一

```
/* 查找符合指定条件的元素（返回值等于e的元素在线性表中的位序） */
int LocateElem(sqlist L, ElemType e)
{
    ElemType *p = L.elem;
    int       i = 1;

    while(i<=L.length && (*p 和 e不相等)) { 宏定义方式实现
        i++;
        p++;
        }

    return (i<=L.length) ? i : 0;   //找到返回i，否则返回0
}
```

```
*p != e
fabs(*p, e)<1e-6
strcmp(*p, e)!=0
strcmp(*p, e)!=0
p->num != e.num
(*p)->num != e->num
```

```
main中调用方法：
  ElemType e;
  为e赋值(各类型赋值方法不同)
  LocateElem(L, e);
```

六合一

```
/* linear_list_sq.c 的实现 */

/* 查找符合指定条件的元素 */
int LocateElem(sqlist L, ElemType e, Status (*compare)(ElemType e1, ElemType e2))
{
    ElemType *p = L.elem;
    int       i = 1;

    while(i<=L.length && (*compare)(*p++, e)==FALSE)
        i++;

    return (i<=L.length) ? i : 0;   //找到返回i，否则返回0
}
```

所有数据类型的处理方法都相同无变化

书上程序的思路：
将不同比较方法放在main中，写成形式相同，内容不同的比较函数，传进函数指针

```
/* main中用于比较两个值是否相等的具体函数，与LocateElem中的函数
   指针定义相同，调用时传入 */

Status MyCompare(ElemType e1, ElemType e2)
{
#ifdef ELEMTYPE_IS_DOUBLE
    if (fabs(e1-e2)<1e-6)
#elif defined (ELEMTYPE_IS_CHAR_ARRAY) || defined (ELEMTYPE_IS_CHAR_P)
    if (strcmp(e1, e2)==0)
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    if (e1.num==e2.num)
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    if (e1->num==e2->num)
#else       //缺省当做int处理
    if (e1==e2)
#endif
        return TRUE;
    else
        return FALSE;
}
```

main中调用方法：

```
ElemType e;
为e赋值(各类型赋值方法不同)
LocateElem(L, e, MyCompare);
```

/* linear_list_sq.c 的实现 */

★ GetElem和LocateElem函数不同类型宏定义的位置差异比较

```
/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType *e)
{
    if (i<1 || i>L.length)
        return ERROR;

#if defined (ELEMTYPE_IS_CHAR_ARRAY) || defined (ELEMTYPE_IS_CHAR_P)
    strcpy(*e, L.elem[i-1]);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    memcpy(e, &(L.elem[i-1]), sizeof(ElemType));
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    memcpy(*e, L.elem[i-1], sizeof(ET));
#else       //int和double直接赋值
    *e = L.elem[i-1];
#endif

    return OK;
}
```

```
main中调用方法:
  ElemType e;
  GetElem(L, i, &e);
```

```
/* 查找符合指定条件的元素 */
int LocateElem(sqlist L, ElemType e, Status (*compare)(ElemType e1, ElemType e2))
{
    ElemType *p = L.elem;
    int       i = 1;

    while(i<=L.length && (*compare)(*p++, e)==FALSE)
        i++;

    return (i<=L.length) ? i : 0;   //找到返回i，否则返回0
}
```

```
/* main中用于比较两值是否相等的函数，与LocateElem中的函数指针定义相同，
   调用时传入 */
Status MyCompare(ElemType e1, ElemType e2)
{
#ifdef ELEMTYPE_IS_DOUBLE
    if (fabs(e1-e2)<1e-6)
#elif defined (ELEMTYPE_IS_CHAR_ARRAY) || defined (ELEMTYPE_IS_CHAR_P)
    if (strcmp(e1, e2)==0)
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    if (e1.num==e2.num)
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    if (e1->num==e2->num)
#else       //缺省当做int处理
    if (e1==e2)
#endif
        return TRUE;
    else
        return FALSE;
}
```

```
main中调用方法:
  ElemType e;
  为e赋值(各类型赋值方法不同)
  LocateElem(L, e, MyCompare);
```

问题：GetElem函数，是宏定义放在GetElem函数实现中，而在main中调用时采用统一方法

LocateElem函数，是通过函数指针传递的方式在实现时统一，而在main中使用宏定义区分

问：1、如果想统一放在_sq_main.c中，GetElem如何改？

2、如果想统一放在_sq.c中，LocateElem如何改？

3、到底哪种更好？书上为什么不统一？

/* linear_list_sq.c 的实现 */

★ GetElem函数的不同实现方法对比（左：函数内宏定义，右：main中宏定义）

```
/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType *e)
{
    if (i<1 || i>L.length)
        return ERROR;

#if defined (ELEMTYPE_IS_CHAR_ARRAY) || defined (ELEMTYPE_IS_CHAR_P)
    strcpy(*e, L.elem[i-1]);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    memcpy(e, &(L.elem[i-1]), sizeof(ElemType));
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    memcpy(*e, L.elem[i-1], sizeof(ET));
#else       //int和double直接赋值
    *e = L.elem[i-1];
#endif

    return OK;
}
```

main中调用方法：

```
ElemType e;
GetElem(L, i, &e);
```

```
/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType *e,
                                Status (*assign)(ElemType *dst, ElemType src))
{
    if (i<1 || i>L.length)
            return ERROR;

    (*assign)(e, L.elem[i-1]);
    return OK;
}
```

```
/* main中函数，处理不同数据类型的赋值 */
Status MyAssign(ElemType *dst, ElemType src)
{
#if defined (ELEMTYPE_IS_CHAR_ARRAY)|| defined (ELEMTYPE_IS_CHAR_P)
    strcpy(*dst, src);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    memcpy(dst, &src, sizeof(ElemType));
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    memcpy(*dst, src, sizeof(ET));
#else       //int和double直接赋值
    *dst = src;
#endif

    return OK;
}
```

main中调用方法：

```
ElemType e1;//部分需要申请空间
GetElem(L, i, &e1, MyAssign);
```

问题：GetElem函数，是宏定义放在GetElem函数实现中，而在main中调用时采用统一方法LocateElem函数，是通过函数指针传递的方式在实现时统一，而在main中使用宏定义区分

问：1、如果想统一放在_sq_main.c中，GetElem如何改？

2、如果想统一放在_sq.c中，LocateElem如何改？

3、到底哪种更好？书上为什么不统一？

/* linear_list_sq.c 的实现 */

★ LocateElem函数的不同实现方法对比（左：函数内宏定义，右：main中宏定义）

```c
/* 查找符合指定条件的元素 */
int LocateElem(sqlist L, ElemType e)
{
#ifdef ELEMTYPE_IS_DOUBLE
    while(i<=L.length && fabs(*p-cur_e) >= 1e-6) {
#elif defined (ELEMTYPE_IS_CHAR_ARRAY) || defined (ELEMTYPE_IS_CHAR_P)
    while(i<=L.length && strcmp(*p, cur_e)) {
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    while(i<=L.length && p->num!=cur_e.num) {
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    while(i<=L.length && (*p)->num!=cur_e->num) {
#else       //缺省当做int处理
    while(i<=L.length && *p!=cur_e) {
#endif
        i++;
        }

    return (i<=L.length) ? i : 0; //找到返回i，否则返回0
}
```

```c
main中调用方法：
  ElemType e;
  为e赋值(各类型赋值方法不同)
  LocateElem(L, e);
```

```c
/* 查找符合指定条件的元素 */
int LocateElem(sqlist L, ElemType e, Status (*compare)(ElemType e1, ElemType e2))
{
    ElemType *p = L.elem;
    int       i = 1;

    while(i<=L.length && (*compare)(*p++, e)==FALSE)
        i++;

    return (i<=L.length) ? i : 0;   //找到返回i，否则返回0
}
```

```c
/* main中用于比较两值是否相等的函数，与LocateElem中的函数指针定义相同，
   调用时传入 */
Status MyCompare(ElemType e1, ElemType e2)
{
#ifdef ELEMTYPE_IS_DOUBLE
    if (fabs(e1-e2)<1e-6)
#elif defined (ELEMTYPE_IS_CHAR_ARRAY) || defined (ELEMTYPE_IS_CHAR_P)
    if (strcmp(e1, e2)==0)
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    if (e1.num==e2.num)
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    if (e1->num==e2->num)
#else       //缺省当做int处理
    if (e1==e2)
#endif
        return TRUE;
    else
        return FALSE;
}
```

```c
main中调用方法：
  ElemType e;
  为e赋值(各类型赋值方法不同)
  LocateElem(L, e, MyCompare);
```

问题：GetElem函数，是宏定义放在GetElem函数实现中，而在main中调用时采用统一方法LocateElem函数，是通过函数指针传递的方式在实现时统一，而在main中使用宏定义区分

问：1、如果想统一放在_sq_main.c中，GetElem如何改？

2、如果想统一放在_sq.c中，LocateElem如何改？

3、到底哪种更好？书上为什么不统一？

/* linear_list_sq.c 的实现 */

★ GetElem和LocateElem函数不同类型宏定义的位置差异比较

问题：GetElem函数，是宏定义放在GetElem函数实现中，而在main中调用时采用统一方法

LocateElem函数，是通过函数指针传递的方式在实现时统一，而在main中使用宏定义区分

问：1、如果想统一放在_sq_main.c中，GetElem如何改？

2、如果想统一放在_sq.c中，LocateElem如何改？

3、到底哪种更好？书上为什么不统一？

请透彻理解!!!

答：根据需要

- 判断相等(MyCompare)，因为用户可能会随时改变相等的条件(double精度1e-5/1e-7、student中学号相等/学号姓名相等)，所以应该放_sq_main.c中，由用户决定
- 拷贝内存/赋值(GetElem)等操作，一旦数据类型确定，操作即确定，为了不增加用户负担（写_sq.c的比写_sq_main.c的编程能力的要求更高），应该放在_sq.c中(对用户透明)

六合一

```
/* linear_list_sq.c 的实现 */
/* 查找符合指定条件的元素的前驱元素 */
Status PriorElem(sqlist L, ElemType cur_e, ElemType *pre_e)

{
    ElemType *p = L.elem;
    int       i = 1;

#ifdef ELEMTYPE_IS_DOUBLE
    while(i<=L.length && fabs(*p-cur_e)>=1e-6) {
#elif defined (ELEMTYPE_IS_CHAR_ARRAY) || defined (ELEMTYPE_IS_CHAR_P)
    while(i<=L.length && strcmp(*p, cur_e)) {
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    while(i<=L.length && p->num!=cur_e.num) {
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    while(i<=L.length && (*p)->num!=cur_e->num) {
#else        //缺省当做int处理
    while(i<=L.length && *p!=cur_e) {
#endif
        i++;
        p++;
        }

    if (i==1 || i>L.length) //找到第1个元素或未找到
        return ERROR;

#if defined (ELEMTYPE_IS_CHAR_ARRAY) || defined(ELEMTYPE_IS_CHAR_P)
    strcpy(*pre_e, *--p);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    memcpy(pre_e, --p, sizeof(ElemType));
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    memcpy(*pre_e, *--p, sizeof(ET));
#else        //int和double直接赋值
    *pre_e = *--p;
#endif

    return OK;
}
```

符合书上定义的函数实现，两组条件编译

不同数据类型的不同处理方法

不同数据类型的不同处理方法

请比较左右代码，想想为什么右侧更合理

六合一

```
/* linear_list_sq.c 的实现 */
/* 查找符合指定条件的元素的前驱元素 */
Status PriorElem(sqlist L, ElemType cur_e, ElemType *pre_e,
                 Status (*compare)(ElemType e1, ElemType e2))
{
    ElemType *p = L.elem;
    int       i = 1;

    while(i<=L.length && (*compare)(*p, e)==FALSE) {
        i++;
        p++;
        }

   if (i==1 || i>L.length) //找到第1个元素或未找到
       return ERROR;

#if defined (ELEMTYPE_IS_CHAR_ARRAY) || defined(ELEMTYPE_IS_CHAR_P)
    strcpy(*pre_e, *--p);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    memcpy(pre_e, --p, sizeof(ElemType));
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    memcpy(*pre_e, *--p, sizeof(ET));
#else        //int和double直接赋值
    *pre_e = *--p;
#endif

    return OK;
}
```

改变：比较用函数指针，赋值用条件编译

不同数据类型的不同处理方法

六合一

```
/* linear_list_sq.c 的实现 */
/* 查找符合指定条件的元素的后继元素 */
Status NextElem(sqlist L, ElemType cur_e, ElemType *next_e)

{
    ElemType *p = L.elem;
    int       i = 1;
```

符合书上定义的函数实现，两组条件编译

```
#ifdef ELEMTYPE_IS_DOUBLE
    while(i<L.length && fabs(*p-cur_e)>=1e-6) {
#elif defined (ELEMTYPE_IS_CHAR_ARRAY) || defined (ELEMTYPE_IS_CHAR_P)
    while(i<L.length && strcmp(*p, cur_e)) {
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    while(i<L.length && p->num!=cur_e.num) {
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    while(i<L.length && (*p)->num!=cur_e->num) {
#else        //缺省当做int处理
    while(i<L.length && *p!=cur_e) {
#endif
        i++;
        p++;
        }
```

不同数据类型的不同处理方法

```
    if (i>=L.length)       //未找到（最后一个元素未比较）
        return ERROR;
```

```
#if defined (ELEMTYPE_IS_CHAR_ARRAY) || defined (ELEMTYPE_IS_CHAR_P)
    strcpy(*next_e, *++p);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    memcpy(next_e, ++p, sizeof(ElemType));
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    memcpy(*next_e, *++p, sizeof(ET));
#else        //int和double直接赋值
    *next_e = *++p;
#endif

    return OK;
}
```

不同数据类型的不同处理方法

六合一

```
/* linear_list_sq.c 的实现 */
/* 查找符合指定条件的元素的前驱元素 */
Status NextElem(sqlist L, ElemType cur_e, ElemType *next_e,
                Status (*compare)(ElemType e1, ElemType e2))
{
    ElemType *p = L.elem;
    int       i = 1;
```

改变：比较用函数指针，赋值用条件编译

```
    while(i<L.length && (*compare)(*p, e)==FALSE) {
        i++;
        p++;
        }
```

请比较左右代码，想想为什么右侧更合理

```
   if (i>=L.length)       //未找到（最后一个元素未比较）
       return ERROR;
```

```
#if defined (ELEMTYPE_IS_CHAR_ARRAY) || defined (ELEMTYPE_IS_CHAR_P)
    strcpy(*next_e, *++p);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    memcpy(next_e, ++p, sizeof(ElemType));
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    memcpy(*next_e, *++p, sizeof(ET));
#else        //int和double直接赋值
    *next_e = *++p;
#endif

    return OK;
}
```

不同数据类型的不同处理方法

```
/* linear_list_sq.c 的实现 */
/* 在指定位置前插入一个新元素 */
Status ListInsert(sqlist *L, int i, ElemType e)
{
    ElemType *p, *q; //如果是算法，一般可以省略，程序不能

    if (i<1 || i>L->length+1)    //合理范围是 1..length+1
        return ERROR;

    /* 空间已满则扩大空间 */
    if (L->length >= L->listsize) {
        ElemType *newbase;
        newbase = (ElemType *)realloc(L->elem,
                  (L->listsize+LISTINCREMENT)*sizeof(ElemType));
        if (!newbase)
            return OVERFLOW;

        L->elem = newbase;
        L->listsize += LISTINCREMENT;
        }

    q = &(L->elem[i-1]);  //第i个元素，即新的插入位置

    /* 从最后一个开始到第i个元素依次后移一格 */
    for (p=&(L->elem[L->length-1]); p>=q; --p)
#if defined (ELEMTYPE_IS_CHAR_ARRAY)
        strcpy(*(p+1), *p);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
        memcpy(p+1, p, sizeof(ElemType));
#else  //int、double、char指针、struct student指针都是直接赋值
        *(p+1) = *p;
#endif
```

六合一

```
 /* 插入新元素，长度+1 */
#if defined (ELEMTYPE_IS_CHAR_ARRAY)
    strcpy(*q, e);
#elif defined (ELEMTYPE_IS_CHAR_P)
    /* 原L->elem[i-1]的指针已放入[i]中，重新申请空间，插入新元素，
       长度+1 */
    L->elem[i-1] = (ElemType)malloc((strlen(e)+1) * sizeof(char));
    if (L->elem[i-1]==NULL)
        return LOVERFLOW;

    strcpy(*q, e);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    memcpy(q, &e, sizeof(ElemType));
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    L->elem[i-1] = (ElemType)malloc(sizeof(ET));
    if (L->elem[i-1]==NULL)
        return LOVERFLOW;

    memcpy(*q, e, sizeof(ET));
#else    //int和double直接赋值
    *q = e;
#endif

    L->length ++;
    return OK;
}
```

不同数据类型的
不同处理方法

```
/* linear_list_sq.c 的实现 */
/* 在指定位置前插入一个新元素 */
Status ListInsert(sqlist *L, int i, ElemType e)
{
    ElemType *p, *q; //如果是算法，一般可以省略，程序不能

    if (i<1 || i>L->length+1)    //合理范围是 1..length+1
        return ERROR;

    /* 空间已满则扩大空间 */
```

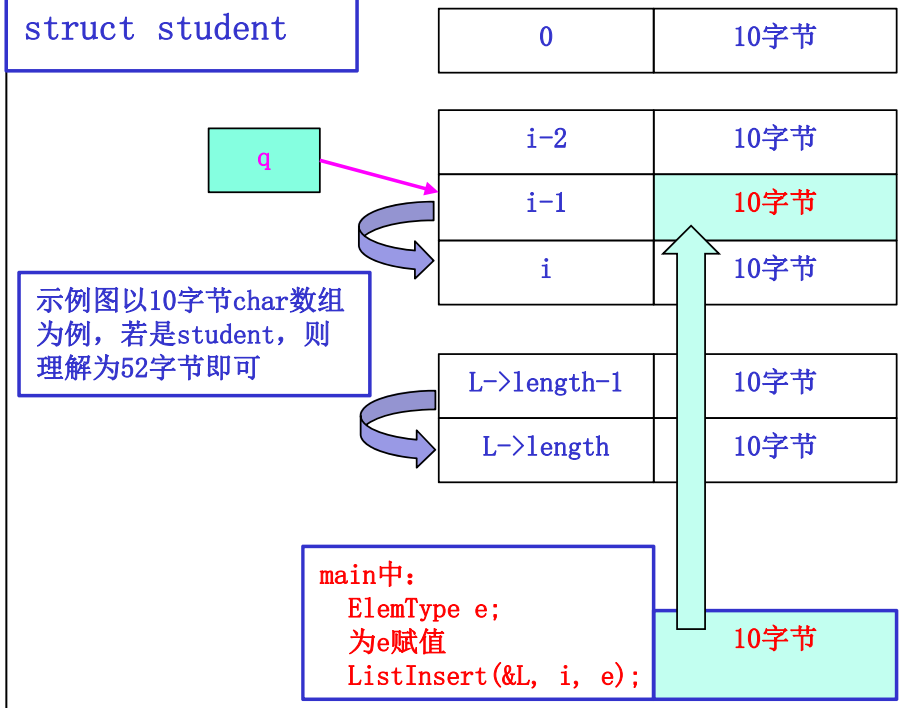
思考：是否可改为如下形式(直接返回L->elem)？两者比较，哪种更好？
提示：若realloc失败，两者区别是什么？从工程的观点去思考问题，回忆课上讲了什么!!!

```
    if (L->length >= L->listsize) {
        L->elem = (ElemType *)realloc(L->elem,
                 (L->listsize+LISTINCREMENT)*sizeof(ElemType));
        if ( !L->elem )
            return OVERFLOW;

        L->listsize += LISTINCREMENT;
    }

    q = &(L->elem[i-1]);  //第i个元素，即新的插入位置

    /* 从最后一个开始到第i个元素依次后移一格 */
    for (p=&(L->elem[L->length-1]); p>=q; --p)
#if defined (ELEMTYPE_IS_CHAR_ARRAY)
        strcpy(*(p+1), *p);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
        memcpy(p+1, p, sizeof(ElemType));
#else  //int、double、char指针、struct student指针都是直接赋值
        *(p+1) = *p;
#endif
    /* 插入新元素，长度+1 */
#if defined (ELEMTYPE_IS_CHAR_ARRAY)
    strcpy(*q, e);
#elif defined (ELEMTYPE_IS_CHAR_P)
    /* 原L->elem[i-1]的指针已放入[i]中，重新申请空间，插入新元素，
       长度+1 */
    L->elem[i-1] = (ElemType)malloc((strlen(e)+1) * sizeof(char));
    if (L->elem[i-1]==NULL)
        return LOVERFLOW;

    strcpy(*q, e);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    memcpy(q, &e, sizeof(ElemType));
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    L->elem[i-1] = (ElemType)malloc(sizeof(ET));
    if (L->elem[i-1]==NULL)
        return LOVERFLOW;

    memcpy(*q, e, sizeof(ET));
#else    //int和double直接赋值
    *q = e;
#endif

    L->length ++;
    return OK;
}
```

六合一

不同数据类型的
不同处理方法

char []
struct student



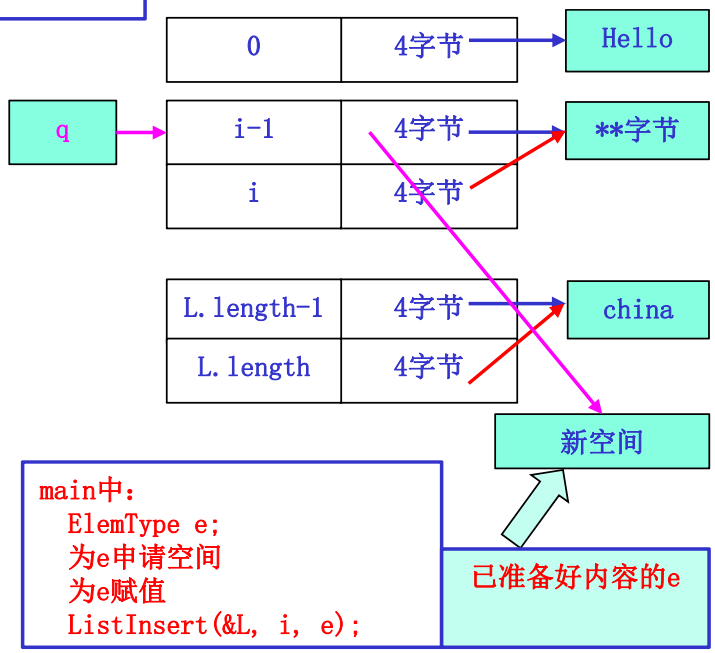

```
    q = &(L->elem[i-1]);  //第i个元素，即新的插入位置

    /* 从最后一个开始到第i个元素依次后移一格 */
    for (p=&(L->elem[L->length-1]); p>=q; --p)
#if defined (ELEMTYPE_IS_CHAR_ARRAY)
      strcpy(*(p+1), *p);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
      memcpy(p+1, p, sizeof(ElemType));
#else  //int、double、char指针、struct student指针都是直接赋值
       *(p+1) = *p;
#endif
```

六合一

```
 /* 插入新元素，长度+1 */
#if defined (ELEMTYPE_IS_CHAR_ARRAY)
    strcpy(*q, e);
#elif defined (ELEMTYPE_IS_CHAR_P)
    /* 原L->elem[i-1]的指针已放入[i]中，重新申请空间，插入新元素，
       长度+1 */
    L->elem[i-1] = (ElemType)malloc((strlen(e)+1) * sizeof(char));
    if (L->elem[i-1]==NULL)
        return LOVERFLOW;

    strcpy(*q, e);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    memcpy(q, &e, sizeof(ElemType));
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    L->elem[i-1] = (ElemType)malloc(sizeof(ET));
    if (L->elem[i-1]==NULL)
        return LOVERFLOW;

    memcpy(*q, e, sizeof(ET));
#else    //int和double直接赋值
    *q = e;
#endif

    L->length ++;
    return OK;
}
```

不同数据类型的
不同处理方法

char *
struct student *




```
    q = &(L->elem[i-1]);  //第i个元素，即新的插入位置

    /* 从最后一个开始到第i个元素依次后移一格 */
    for (p=&(L->elem[L->length-1]); p>=q; --p)
#if defined (ELEMTYPE_IS_CHAR_ARRAY)
        strcpy(*(p+1), *p);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
        memcpy(p+1, p, sizeof(ElemType));
#else  //int、double、char指针、struct student指针都是直接赋值
        *(p+1) = *p;
#endif
```

六合一

```
 /* 插入新元素，长度+1 */
#if defined (ELEMTYPE_IS_CHAR_ARRAY)
    strcpy(*q, e);
#elif defined (ELEMTYPE_IS_CHAR_P)
    /* 原L->elem[i-1]的指针已放入[i]中，重新申请空间，插入新元素，
       长度+1 */
    L->elem[i-1] = (ElemType)malloc((strlen(e)+1) * sizeof(char));
    if (L->elem[i-1]==NULL)
        return LOVERFLOW;

    strcpy(*q, e);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    memcpy(q, &e, sizeof(ElemType));
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    L->elem[i-1] = (ElemType)malloc(sizeof(ET));
    if (L->elem[i-1]==NULL)
        return LOVERFLOW;

    memcpy(*q, e, sizeof(ET));
#else    //int和double直接赋值
    *q = e;
#endif

    L->length ++;
    return OK;
}
```

问：对于CHAR_P和STUDENT_P，
如果二次申请不成功，
会有什么后果？如何解决？
提示：想想课上怎么说的？

不同数据类型的
不同处理方法

```
/* linear_list_sq.c 的实现 */
/* 删除指定位置的元素，并将被删除元素的值放入e中返回 */
Status ListDelete(sqlist *L, int i, ElemType *e)
{
    ElemType *p, *q; //如果是算法，一般可以省略，程序不能

    if (i<1 || i>L->length)  //合理范围是 1..length
        return ERROR;

    p = &(L->elem[i-1]);     //指向第i个元素

    //取第i个元素的值放入e中
#if defined (ELEMTYPE_IS_CHAR_ARRAY) ||
                                    defined (ELEMTYPE_IS_CHAR_P)
    strcpy(*e, *p);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    memcpy(e, p, sizeof(ElemType));
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    memcpy(*e, *p, sizeof(ET));
#else    //int和double直接赋值
    *e = *p;
#endif

    q = &(L->elem[L->length-1]);    //指向最后一个元素

    //两种情况需要释放空间，其它4种不需要
#if defined (ELEMTYPE_IS_CHAR_P) ||
                          defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    free(*p); //释放空间
#endif
```

```
 /* 从第i+1到最后，依次前移一格 */
    for (++p; p<=q; ++p) {
#if defined (ELEMTYPE_IS_CHAR_ARRAY)
        strcpy(*(p-1), *p);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
        memcpy((p-1), p, sizeof(ElemType));
#else //int、double、char指针、struct student指针都是直接赋值
        *(p-1) = *p;
#endif
        }

    L->length --;          //长度-1
    return OK;
}
```

六合一

思考：如果若干次ListInsert后，listsize已增大(假设150)，再经过若干次ListDelete后，length变小(假设123)，目前如何处理？应如何更合理？

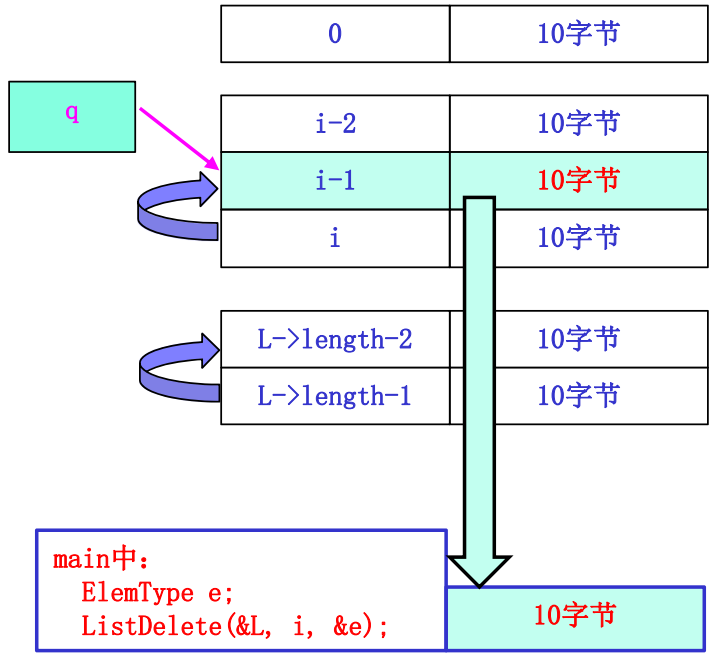
不同数据类型的
不同处理方法

```
/* linear_list_sq.c 的实现 */
/* 删除指定位置的元素，并将被删除元素的值放入e中返回 */
Status ListDelete(sqlist *L, int i, ElemType *e)
{
    ElemType *p, *q; //如果是算法，一般可以省略，程序不能

    if (i<1 || i>L->length)  //合理范围是 1..length
        return ERROR;

    p = &(L->elem[i-1]);     //指向第i个元素

    //取第i个元素的值放入e中
#if defined (ELEMTYPE_IS_CHAR_ARRAY) ||
                                defined (ELEMTYPE_IS_CHAR_P)
    strcpy(*e, *p);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    memcpy(e, p, sizeof(ElemType));
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    memcpy(*e, *p, sizeof(ET));
#else    //int和double直接赋值
    *e = *p;
#endif

    q = &(L->elem[L->length-1]);    //指向最后一个元素

    //两种情况需要释放空间，其它4种不需要
#if defined (ELEMTYPE_IS_CHAR_P) ||
                        defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    free(*p); //释放空间
#endif
```

char []
struct student

```
 /* 从第i+1到最后，依次前移一格 */
    for (++p; p<=q; ++p) {
#if defined (ELEMTYPE_IS_CHAR_ARRAY)
        strcpy(*(p-1), *p);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
        memcpy((p-1), p, sizeof(ElemType));
#else //int、double、char指针、struct student指针都是直接赋值
        *(p-1) = *p;
#endif
        }

    L->length --;        //长度-1
    return OK;
}
```

六合一

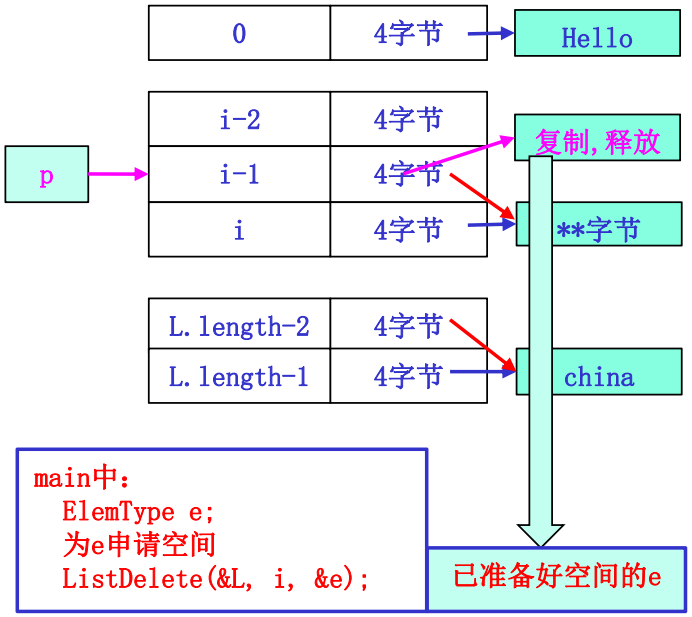


```
/* linear_list_sq.c 的实现 */
/* 删除指定位置的元素，并将被删除元素的值放入e中返回 */
Status ListDelete(sqlist *L, int i, ElemType *e)
{
    ElemType *p, *q; //如果是算法，一般可以省略，程序不能

    if (i<1 || i>L->length)  //合理范围是 1..length
        return ERROR;

    p = &(L->elem[i-1]);     //指向第i个元素

    //取第i个元素的值放入e中
#if defined (ELEMTYPE_IS_CHAR_ARRAY) ||
                                    defined (ELEMTYPE_IS_CHAR_P)
    strcpy(*e, *p);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    memcpy(e, p, sizeof(ElemType));
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    memcpy(*e, *p, sizeof(ET));
#else    //int和double直接赋值
    *e = *p;
#endif

    q = &(L->elem[L->length-1]);    //指向最后一个元素

    //两种情况需要释放空间，其它4种不需要
#if defined (ELEMTYPE_IS_CHAR_P) ||
                          defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    free(*p); //释放空间
#endif
```

char *
struct student *

```
 /* 从第i+1到最后，依次前移一格 */
    for (++p; p<=q; ++p) {
#if defined (ELEMTYPE_IS_CHAR_ARRAY)
        strcpy(*(p-1), *p);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
        memcpy((p-1), p, sizeof(ElemType));
#else //int、double、char指针、struct student指针都是直接赋值
        *(p-1) = *p;
#endif
        }

    L->length --;        //长度-1
    return OK;
}
```

六合一



/* linear_list_sq.c 的实现 */

六合一

```
/* 遍历线性表 */
Status ListTraverse(sqlist L, Status (*visit)(ElemType e))
{   extern int line_count; //main中定义的换行计数器(与算法无关)
    ElemType *p = L.elem;
    int       i = 1;

    line_count = 0;         //计数器恢复初始值(与算法无关)
    while(i<=L.length && (*visit)(*p++)==TRUE)
        i++;
    if (i<=L.length)
         return ERROR;

    printf("\n");  //最后打印一个换行(与算法无关)
    return OK;
}
```

本函数牵涉到不同数据类型的输出格式，条件编译放在main中更合理!!!

所有数据类型的处理方法都相同无变化

```
/* main中用于比较访问线性表某个元素的值的具体函数，与 ListTraverse 中的函数
   指针定义相同，调用时传入 */

Status MyVisit(ElemType e)
{
#ifdef ELEMTYPE_IS_DOUBLE
    printf("%5.1f->", e);
#elif defined (ELEMTYPE_IS_CHAR_ARRAY) || defined (ELEMTYPE_IS_CHAR_P)
    printf("%s->", e);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    printf("%d-%s-%c-%f-%s->", e.num, e.name, e.sex, e.score, e.addr);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    printf("%d-%s-%c-%f-%s->", e->num, e->name, e->sex, e->score, e->addr);
#else       //缺省当做int处理
    printf("%3d->", e);
#endif
    /* 每输出10个，打印一个换行 */
    if ((++line_count)%10 == 0)
            printf("\n");
    return OK;
}
```

```
main中：
  ListTraverse(L, MyVisit);
```

§13.动态内存申请 – 顺序表适应不同的数据类型

2.线性表顺序表示的基本操作的实现

2.1.C语言版

2.2.C++语言版

★ 类模板方式，具体实现见后续课程