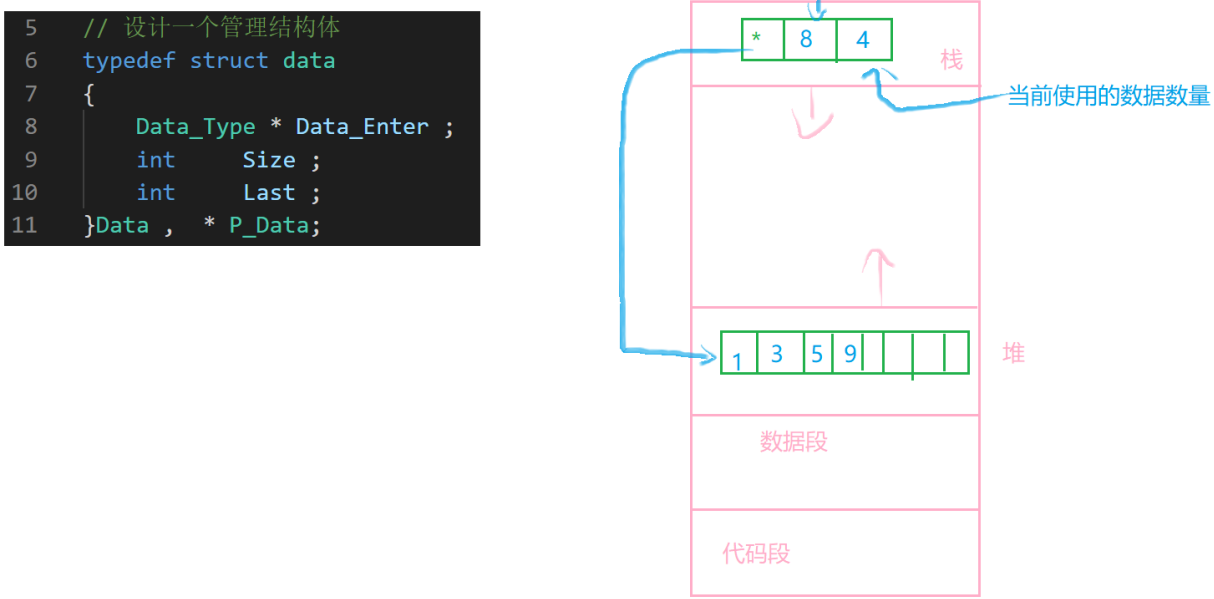
概念：

有一组有序的数据，被存放到一个连续的内存当中去。

设计一个管理结构体（用来说明：数据的入口，总数据的量，当前数据的量）：

```c
// 设计一个管理结构体
typedef struct data
{
    Data_Type * Data_Enter ;    // 堆空间的入口地址
    int     Size ;              // 数据总量（可以存放的数据量） 数组最大的下标
    int     Last ;              // 当前的使用量       数组当前使用的下标
}Data ,  * P_Data;
```



初始化：

1. 建立堆空间， 来初始化 Data_Type * Data_Enter ;
2. 确定堆空间的大小， 初始化 int Size ;
3. 初始化 当前使用量为 0 int Last ;

```c
Data * init_data()
{
    // 在堆中申请一个管理结构体的内存空间
    P_Data ptr = calloc(1,sizeof(Data));

    // 申请堆空间来作为顺序表的存储空间
    ptr->Data_Enter = calloc( DATA_SIZE ,sizeof(Data_Type));

```

```
    ptr->Size = DATA_SIZE ; // 设置顺序表的大小为 10
    ptr->Last = 0 ;  // 设置当前使用量为 0

    return ptr ; // 返回管理结构体指针
}
```

插入数据：

需要传递的参数：

1. 管理结构体的地址
2. 新的数据内容
3. 主要判断表是否满

```
//普通插入（无序）
int ins_data( Data* Ctrl , Data_Type new_data )
{
    //  判断管理结构体指针是否为空
    //  当前管理结构体中现有数据是否已满
    if(Ctrl == NULL || Ctrl->Last == Ctrl->Size )
    {
        printf("当前内存已满！！！\n");
        return Ctrl->Last; // 下标为 size
    }

    // 把数据存放到顺序表的内存中
    Ctrl->Data_Enter[Ctrl->Last]  = new_data ;
    // *(Ctrl->Data_Enter+Ctrl->Last) = new_data;

    // 让顺序表的当前下标往后移动
    Ctrl->Last ++ ;

    return Ctrl->Last ; // 为具体的下标值
}

// 顺序插入
int ins_data( Data* Ctrl , Data_Type new_data )
{
    //  判断管理结构体指针是否为空
```

```
    //  当前管理结构体中现有数据是否已满
    if(Ctrl == NULL || Ctrl->Last == Ctrl->Size )
    {
        printf("当前内存已满！！！\n");
        return 0 ;
    }

    int tmp = 0 ;
    if(!(Ctrl->Last == 0))  // 判断是否给空表
    {
        for (int i = 0; i < Ctrl->Last ; i++)// 遍历整个表
        {
            if(new_data <= Ctrl->Data_Enter[i]) // 比较数据是否与新数据相同或大于
            {
                tmp = i ;
                for (int j = Ctrl->Last ; j > i   ; j--)
                {
                    Ctrl->Data_Enter[j] =
                            Ctrl->Data_Enter[j-1] ;
                }
                // 把数据存放到顺序表的内存中
                Ctrl->Data_Enter[ tmp ]  = new_data ;
                // 让顺序表的当前下标往后移动
                Ctrl->Last ++ ;
                return Ctrl->Last ;
            }

        }

    }

    // 把数据存放到顺序表的内存中
    Ctrl->Data_Enter[ Ctrl->Last ]  = new_data ;
    // 让顺序表的当前下标往后移动
    Ctrl->Last ++ ;


    return Ctrl->Last ; // 为具体的下标值
}
```

显示数据：

注意判断表是否为空

```
int displayer_data(Data * Ctrl)
{
    //  判断管理结构体指针是否为空
    //  当前管理结构体中现有数据是否已满
    if(Ctrl == NULL || Ctrl->Last == 0 )
    {
        printf("顺序表为空！！！\n");
        return Ctrl->Last; // 下标为 size
    }

    for (int i = 0; i < Ctrl->Last ; i++)
    {
        printf("data:%d\n" , *(Ctrl->Data_Enter+i));
    }

    return Ctrl->Last ;
}
```

删除数据：

先找到数据，然后把该数据后面所有的数据往前移动一下（覆盖）

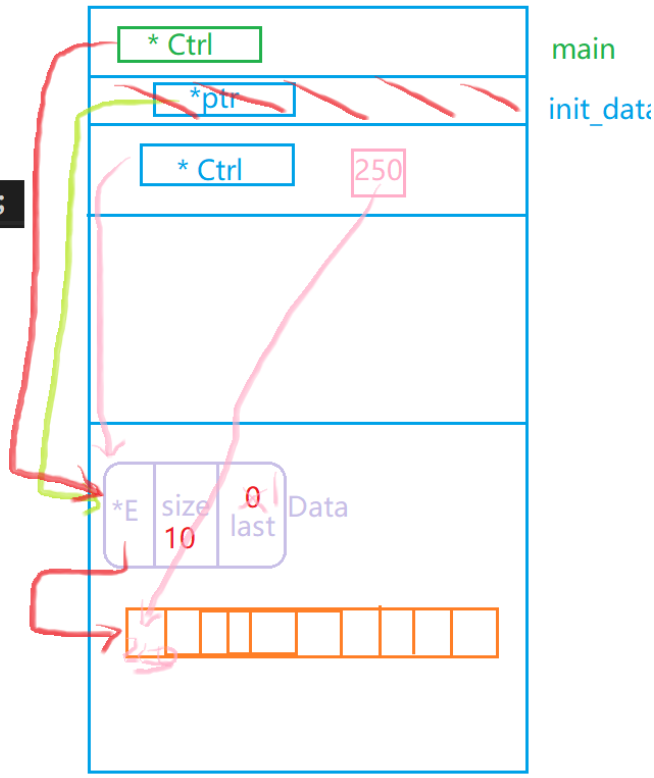
```

int del_data( Data * Ctrl , int del )
{
    //  判断管理结构体指针是否为空
    //  当前管理结构体中现有数据是否已满
    if(Ctrl == NULL || Ctrl->Last == 0 )
    {
        printf("顺序表为空！！！\n");
        return 0 ; // 下标为 size
    }

    for (int i = 0; i < Ctrl->Last ; i++)// 遍历整个表
```

```
    {
        if(del == Ctrl->Data_Enter[i]) // 比较数据是否与新数据相同或大于
        {
            for (size_t j = i ; j < Ctrl->Last ; j++)
            {
                *(Ctrl->Data_Enter+j) = *(Ctrl->Data_Enter+j+1) ;
            }
            Ctrl->Last -- ;
        }
    }
    return 1 ;
}
```

```
// 初始化
P_Data Ctrl = init_data( );
```

```
P_Data ptr = calloc(1,sizeof(Data));
```

```
// 插入数据
ins_data(Ctrl , new_data);
```



顺序表优缺点：

优点：

- 数据所存储的物理位置能够体现数据与数据的逻辑关系
- 存储的形式是按顺序存储的， 因此存储的密度非常高，存储在一片连续的内存中
- 可以单独访问任何一个数据

缺点：

- 在实现插入/删除操作的时候需要保持数据之间的逻辑关系和物理位置的关系需要大面积的移动数据，操作起来很费力
- 如果数据量较多，则需要一大片连续的内存
- 如果数据量较多，对某一个数据进行删除/插入则需要大量的时间来进行移动其它的数据

练习：

0. 先看懂刚才写的代码
1. 尝试自己写一个顺序表的初始化+添加+显示
2. 尝试实现插入数据的操作（使得顺序表有序）

作业：

创建一个顺序表，并实现一下功能：

1. 从键盘获取用户输入的整型数据，存储到顺序表中。
2. 在存入新数据时实现排序，把数据按从小到大的顺序存储到表中
3. 当用户输入负数时，则把该数据所对应的绝对值进行删除
4. 每次增加/删除数据操作成功后把顺序表打印到屏幕中