step further

专题

起步：

认知与体验（硬件、软件、程序与C语言）

进阶：

判断与推理（流程控制方法、语句）

抽象与联系（模块设计方法、函数）

表达与转换（基本操作、数据类型）

提高：

构造与访问（数组、**指针-1**、**字符串**、结构体、**指针-2**、**链表**）

归纳与推广（程序设计的本质）

指针及其运用

- 指针的基本概念
- 用指针操纵数组
- **用指针在函数间传递数据**
  - 指针类型参数
  - const的作用
  - 引用类型参数
  - 指针类型返回值
- **用指针访问动态变量**
  - 通用指针与void类型
  - 动态变量的创建、访问和撤销
  - 内存泄漏与悬浮指针
- 用指针操纵函数*

通用指针与void类型

- void：空类型的关键字
- 空类型的值集为空，在计算机中不占空间，一般不能参与基本操作

- 通常用来描述不返回数据的函数的返回值，以及不需要参数的函数的形参

- 还可以作为指针类型的基类型，形成通用指针类型（void *）。

- 通用指针类型变量不指向具体的数据，不能用来访问数据，但任何指针类型都可以隐式转换为通用指针类型，而且在将通用指针类型转换回原来的指针类型时不会丢失信息。

- 通用指针类型经常作为函数的形参和返回值的类型，以提高所定义的函数（比如创建动态变量的库函数）的通用性。

内存分配方式

- **系统为程序中的数据分配内存空间一般有两种方式**
  - 静态：程序开始执行前在静态数据区分配空间
    - 先由编译器对其定义行进行特殊处理（规划所需内存，分析、处理其初始化值），程序执行时，由执行环境在静态数据区为之分配空间，并写入初始化值，若未初始化则写入初值0，此后程序可获取或修改其值，直到整个程序执行结束才收回其空间。
  - 动态：程序执行时在栈区或堆区分配空间
    - 在栈区：定义行有对应的目标代码，通过代码的执行在栈区获得空间，若已初始化则获得初始化值，若未初始化则其初值是内存里原有的值，此后程序可以访问其内存空间，获取或修改其值，一旦复合语句执行结束即收回其内存空间
    - 在堆区：由程序员编写的相关代码（调用malloc库函数）申请内存空间，通过代码的执行在堆区获得空间，由于没有初始化，其初值为内存里原有的值，此后程序可以访问其内存空间，进行赋值操作，以及获取或修改其值，最后通过程序员编写的相关代码（调用free库函数）释放其内存空间。如果程序员忘记编写释放其内存空间的代码，则要等整个程序执行结束时才收回其内存空间

内存分配方式

```
void MyMax(int n1, int n2, int n3);
int max = 0;                          //全局变量
int main( )
{
    int n1, n2, n3;                   //局部变量
    scanf("%d%d%d", &n1, &n2, &n3);
    while(n1 != n2 && n2 != n3)
    {
        MyMax(n1, n2, n3);
        printf("%d \n", max);
        scanf("%d%d%d", &n1, &n2, &n3);
    }
    return 0;
}
```

静态内存分配方式

动态内存分配方式（栈区）

```
void MyMax(int n1, int n2, int n3)
{
        static int count = 0;   //静态变量
        if(n1 >= n2)
                max = n1;
        else
                max = n2;
        if(max < n3)
                max = n3;
        ++count;
}
```

重点

动态内存分配方式（堆区）

- **动态变量的创建、访问和撤销**

动态变量的创建

- 动态变量是指系统根据程序执行过程中的动态需求，临时在零星的空闲内存中寻找合适的若干单元来存储的一类数据。
- 动态变量没有定义过程，由malloc库函数创建。

```
#include <cstdlib>
#include <stdlib.h>

#include <malloc.h>
```

动态变量的创建

- malloc库函数的原型是

```
void *malloc(unsigned int size);
```

  - 该函数在stdlib中声明，其功能是在程序的堆区分配size个单元，并返回该内存空间的首地址。调用时一般需要用操作符sizeof计算需要分配的单元个数作为实参，并对返回值进行强制类型转换，以便存储某具体类型的数据。
  - 比如，

```
(int *)malloc(sizeof(int));
        // 创建一个int型动态变量，可存储1个int型数据
(double *)malloc(sizeof(double) * n);
        // 创建一个double型动态数组，含n个元素
```

动态变量的访问

- 创建的动态变量没有变量名，需要通过指针变量来访问。

动态变量的访问

```
void *malloc(unsigned int size);
```

(1) 一般动态变量的访问

```
int *pd;
pd = (int *)malloc(sizeof(int));
        // 接下来可用 *pd 表示和访问该动态变量
```

```
void *malloc(unsigned int size);
```

(2) 动态数组的访问

```
int n;
scanf("%d", &n);   //假设输入的 n 为5
double *pda;
pda = (double *)malloc(sizeof(double) * n);
```

//接下来，可用*(pda+3)或pda[3]表示和访问该动态数组中的第4个元素

```
pda[0] ~ pda[n-1]
*pda; ++pda, *pda; ……; ++pda, *pda;
*pda ~ *(pda + n-1)
```

动态变量的撤销

- C语言中的动态变量（包括动态数组）在所属的函数执行完后不会自动消亡，需要由free库函数在程序中显式地撤销。
- free库函数的原型是

```
void free(void *p);
```

  - 该函数在stdlib中声明，其功能是释放由malloc函数分配的p所指向的内存空间。
  - 比如，

```
free(pd);          //  撤销pd指向的动态变量
free(pda);         //  撤销pda指向的动态数组
```

```
int *pda = (int *)malloc(sizeof(int) * n * 2);//看成一维数组
```

二维动态数组

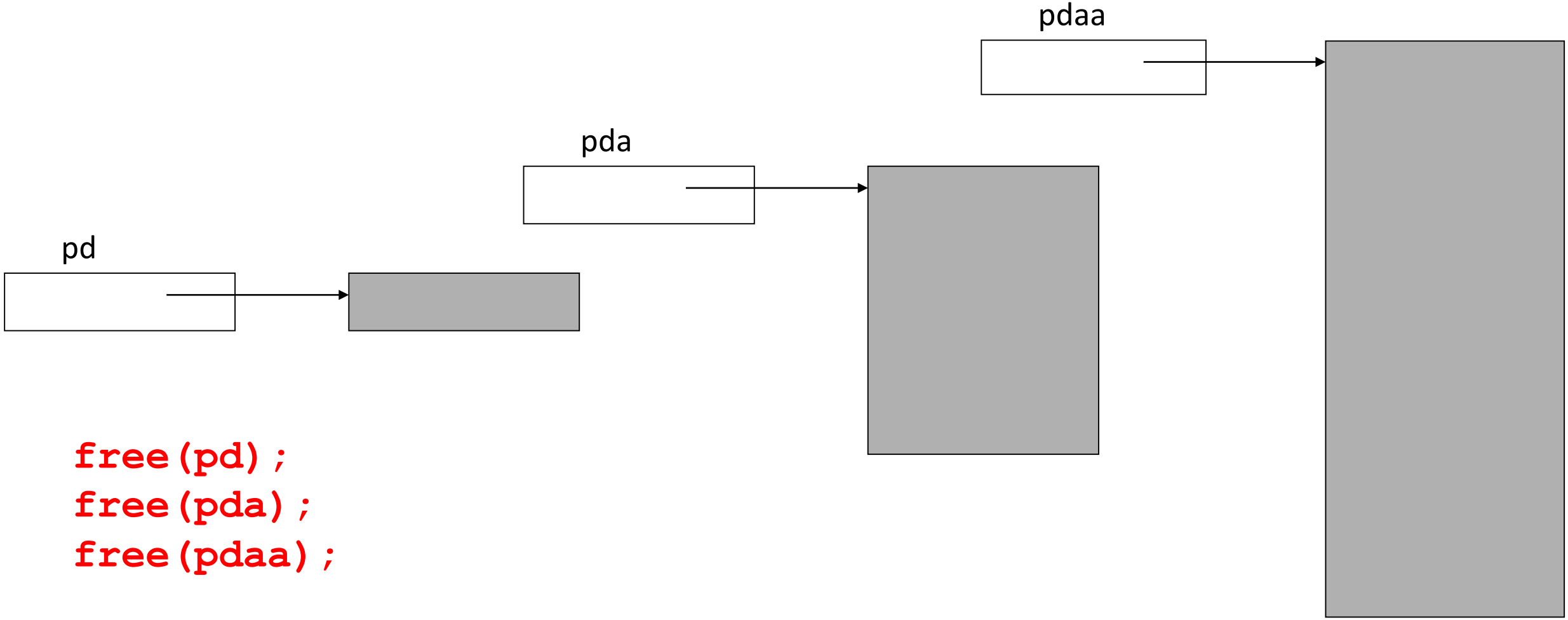
```
int (*pdaa)[2] = (int (*)[2])malloc(sizeof(int) * n * 2);
```

```
typedef int A[2];
A *pdaa;          //pdaa可以指向列数为2的int型二维数组中的一行
pdaa = (A *)malloc(sizeof(int) * n * 2);
                  //返回值强制转换成 A *（即int(*)[2]）类型
```

```
int **pdaa = (int **)malloc(sizeof(int) * n * 2);
                  //创建一个int型n行2列二维动态数组（C语言是弱类型）
```

```
free(pdaa);       //  撤销pdaa指向的二维动态数组
```

- **注意，撤销的是动态变量所在的堆区内存空间，即图中的阴影区域，而不是指向它的指针变量。**




```
free(pd);
free(pda);
free(pdaa);
```

C：创建与撤销动态变量

重点

```
#include <stdlib.h>

int *pd = (int *)malloc(sizeof(int));
double *pda = (double *)malloc(sizeof(double) * n);
int (*pdaa)[10] = (int (*)[10])malloc(sizeof(int) * n * 10);

free(pd);
free(pda);
free(pdaa);
```

C++：创建与撤销动态变量

重点

```
int *pd = new int;
double *pda = new double[n];
int (*pdaa)[10] = new int[n][10];

delete pd;
delete []pda;
delete []pdaa;
```

eg.

```
int *pd = new int;
*pd = 3;
cout << endl << *pd << endl;
```

```
delete pd;
```

eg.

```
int *pda = new int[n];
for(int i=0; i < n; ++i, ++pda)
      cin >> *pda;

pda -= n;
for(int i=0; i < n; ++i, ++pda)
      cout << *pda << ", ";
```

```
for(int i=0; i < n; ++i)
      cin >> pda[i];


for(int i=0; i < n; ++i)
      cout << pda[i] << ", ";
```

```
delete []pda;
```

eg.

```
const int N = 10;
int m;
cin >> m;
int (*pdaa)[N] = new int[m][N];
for(int i=0; i < m; ++i)
      for(int j=0; j < N; ++j)
            cin >> *(*(pdaa+i) + j);

for(int i=0; i < m; ++i)
{     for(int j=0; j < N; ++j)
            cout << *(*(pdaa+i) + j) << "   ";
      cout << endl;
}
```

```
for(int i=0; i < m; ++i)
      for(int j=0; j < N; ++j)
            cin >> pdaa[i][j];


for(int i=0; i < m; ++i)
{     for(int j=0; j < N; ++j)
            cout << pdaa[i][j] << "
      cout << endl;
}
```

```
delete []pdaa;
```

内存泄露*

难点

- 对于动态变量，如果没有显式地撤销，那么当指向它的指针变量的生存期结束后（比如其所属函数执行完毕但整个程序尚未执行完毕时），或者指向它的指针变量指向了别处，则该动态变量仍然存在，但却无法访问，从而造成内存空间的浪费，这一现象称作“内存泄露”，即上图中阴影区域的内存空间“泄露”了。
  - 比如，

```
int *pda;
int m, n;
scanf("%d", &n);  //假设输入的 n 为5
pda = (int *)malloc(sizeof(int) * n);  //pda指向动态数组
pda = &m;  //     pda指向m，上面的动态数组造成内存泄露
```

悬浮指针*

难点

- 动态变量在用free库函数撤销后，指向它的指针变量则指向一个无效空间，这时该指针变量变为“悬浮指针”（dangling pointer）。即前图中阴影区域的内存空间释放后，指针变量p则变为“悬浮指针”
  - 比如，

```
int *pda;
int n;
scanf("%d", &n);  //假设输入的 n 为5
pda = (int *)malloc(sizeof(int) * n); // pda指向动态数组
free(pda);
// pda变为“悬浮指针”，不能通过pda访问数据，比如不可*pda = 0
```

内存泄露与悬浮指针*

难点

```
int *pda;
int m;
pda = (int *)malloc(sizeof(int) * n);
……

pda = &m;
```

pda所指向的动态空间没有释放，但无法访问，泄漏了

pda所指向的动态空间释放了，不知道会分配给谁，
但pda 里存储的还是该动态空间的首地址

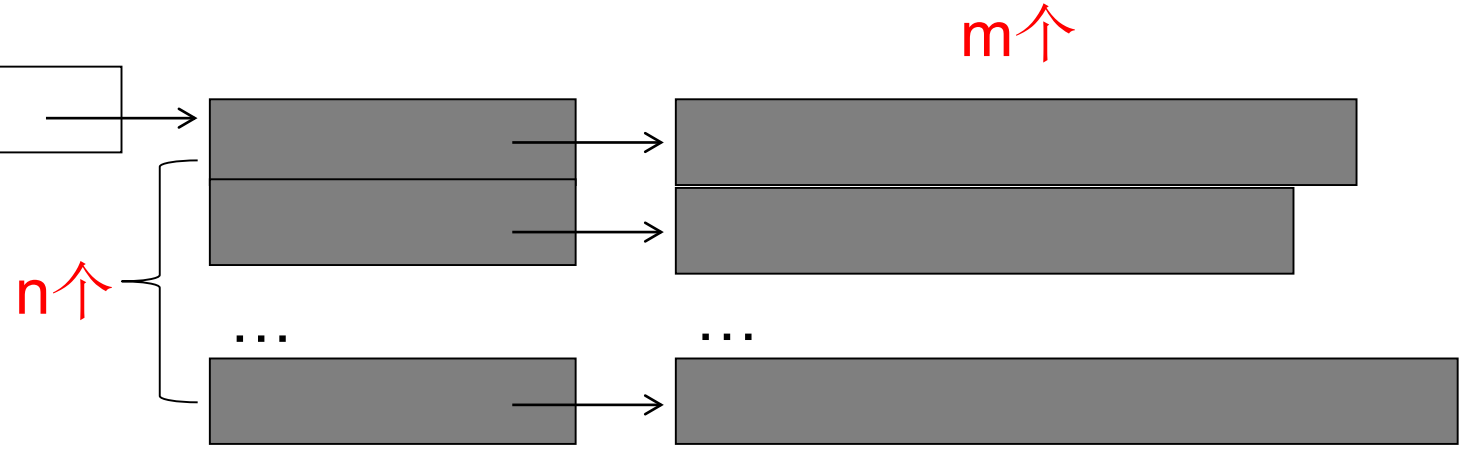
```
int *pda;
pda = (int *)malloc(sizeof(int) * n);
……
free(pda);
```

**

不要求掌握

- 行列都是动态的“二维数组”（第二行第一个元素与第一行最后一个元素未必连续）：

```
int n, m;
cin >> n;
char **array = new char *[n];     //先new一个动态字符指针数组（一维）
for(int i=0; i < n; ++i)
{  cin >> m;
   array[i]=  new char[m];   //再分别new数组的每一个元素指向的数组
}
```




```
for(int i=0; i < n; ++i)
      delete []array[i];
delete []array;
```

- **实际应用**

1）对输入的10个整数进行排序，可以用数组来实现：

```
const int N = 10;
int i, a[N];

for(i = 0; i < N; ++i)
   scanf("%d", &a[i]); //cin >> a[i];

Sort(a, N);
…
```

2）对输入的若干个整数进行排序（先输入整数的个数n，后输入n个整数），可以：

```
int n, i;
scanf("%d", &n); //cin >> n;
int a[n];

for(i = 0; i < n; ++i)
   scanf("%d", &a[i]); //cin >> a[i];

Sort(a, n);
```

有些新标准，
允许数组长度为变量，
此法适用于支持这类新标准的编译器。

…

2）对输入的若干个整数进行排序（先输入整数的个数n，后输入n个整数），可以用动态数组来实现：

老标准和有些新标准，
不允许数组长度为变量，
此法适用于支持这类标准的编译器。

```
int n, i;
int *pda;
scanf("%d", &n); //cin >> n;
pda = (int *)malloc(sizeof(int) * n);    // pda = new int[n];
for(i = 0; i < n; ++i)
  scanf("%d", &pda[i]); //cin >> pda[i];

Sort(pda, n);
…
free(pda);//delete []pda;
```

3）对输入的若干个正整数进行排序（先输入各个正整数，最后输入一个结束标志 -1）

```
?      //用不断调整动态数组的大小来实现

Sort(pda, count);
…      //输出排序后的数组
free(pda); //delete []pda;
```

用不断调整动态数组的大小来实现

```
const int INC = 5;
int max_len = 10, count = 0, m;
int *pda = (int *)malloc(sizeof(int) * max_len);
          //int *pda = new int[max_len];
scanf("%d", &m);
//cin >> m;
while(m != -1)
{
  ……

  pda[count] = m;
  ++count;
  scanf("%d", &m); //cin >> m;
}
```

```
if(count >= max_len)
{  max_len += INC;  //扩容
   int *q = (int *)malloc(sizeof(int) * max_len);
     //int *q = new int[max_len];
   for(int i = 0; i < count; ++i)
     q[i] = pda[i];
   free(pda);          //delete []pda;
   pda = q;
   q = NULL;
}
```

小结

- 要求：
  - 掌握指针的典型用法之二
    - 指针可以用来操作动态数据。
      - ✓动态变量和动态数组需要在程序中创建与撤销，使用过程中要注意避免内存泄露和悬浮指针等问题。
    - 一个程序代码量≈40行
  - 继续保持良好的编程习惯

Thanks!