中国科学技术大学

指针的高级应用

吴锋

Email: wufeng02@ustc.edu.cn

本章内容概述

- 指针基本概念回顾
- 指向数组的指针与指针数组
- 二级与多级指针
- 带指针的函数与函数指针
- 高级内存管理技术
  - 动态内存分配
  - 内存池管理
- 应用实例：链表
- 内存调试与优化

本章内容概述

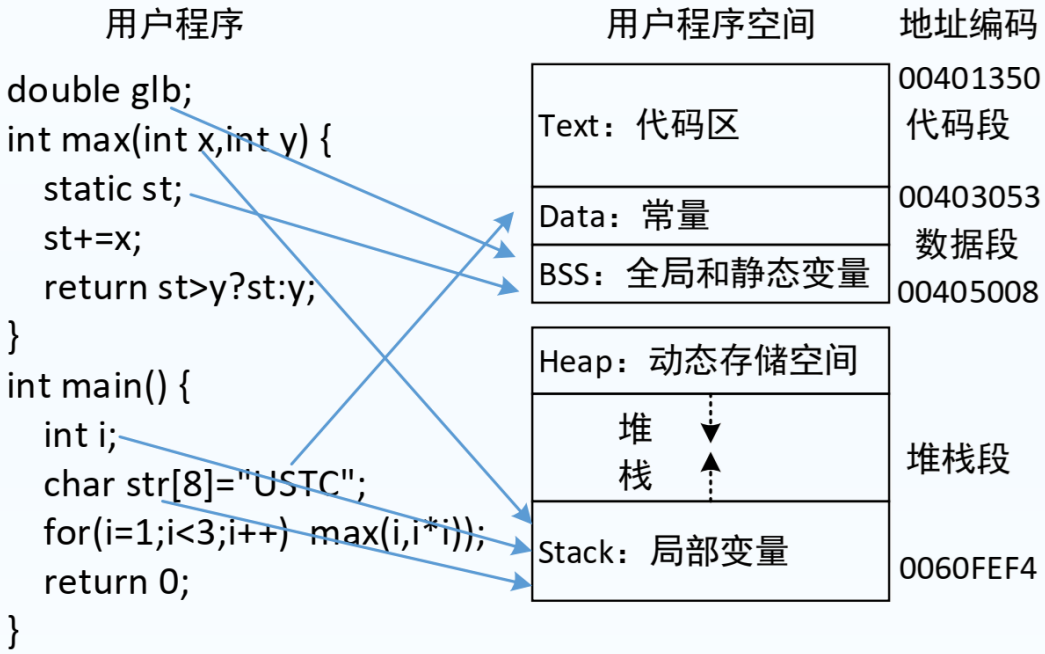
- 指针基本概念回顾
- 指向数组的指针与指针数组
- 二级与多级指针
- 带指针的函数与函数指针
- 高级内存管理技术
  - 动态内存分配
  - 内存池管理
- 应用实例：链表
- 内存调试与优化

变量名到地址的映射

- 简单说：程序运行过程中，CPU根据变量名与内存地址的映射关系，在该变量所在的地址进行数据操作
- 实际上，从变量名到地址的映射
  - 程序编译后，变量名映射为逻辑地址
  - 程序加载到内存中时，逻辑地址映射为物理内存地址
  - 也有操作系统将逻辑地址映射为逻辑内存地址，再由操作系统维护逻辑内存地址到物理内存地址的映射
- 程序运行过程中，CPU通过地址对内存单元进行数据操作

程序空间与数据存储

- 程序空间
  - 操作系统为程序分配内存空间用于加载指令和数据
- 程序空间的分段
  - 代码段、堆栈段、数据段



指针变量的声明

- 指向复杂数据类型的指针
  - `int (*ptr)[5]; //ptr是指向由5个整型元素构成的数组的指针变量`
  - `int *parr[5]; //parr是一个5个整形指针元素构成的指针数组`
  - `int **ip; //ip是一个“指向整型量的指针变量”的指针变量`
  - `int *fip(); //fip是返回整型指针的函数（函数声明）`
  - `int (*pti)(); //pti是指向一个返回整数的函数的指针变量`
  - `int *(*pfpi)();//pfpi是一个指向函数的指针变量，该函数返回整数指针`
- 指针变量也占据内存空间（在32/64位系统上分别占4/8字节）
- 调试指针时可使用 %p 打印指针所指向的地址

```
int num = 10;
int *p = &num;
printf("Address of num: %p\n", (void *)p);
// 输出地址如: 0x7fffa1c9818c
```

指针相关运算符

- 取地址运算符&，例： scanf("%d", &a);
- 间接访问运算符*，例：*p=*p+10;
- 赋值运算符=，例：int a,*p; p=&a;
- 与整数加、减运算 +，-
  - 例：int a[10], *p=a; p+n指向a[i+n]，p-n指向a[i-n]
  - 例：若指针q指向数组a的第j个元素a[j]，则 p-q的值即为i-j
- 自增、自减运算++，--
  - 例：p++的值为a[i]的地址，p自增后指向a[i+1]
  - 例：p--的值为a[i]的地址，p自减后指向a[i-1]
  - 类似的，可以得到 ++p，--p 的值和指向

指针相关运算符（续）

- 关系运算>，<，==，>=，<=，!=
  - 同类型指针之间可比较是否==或!=，判断二者是否指向同一个数据
  - 指向同一个数组中的元素的指针可以进行>、>=、<、<=的关系比较，判断这两个指针所指元素在数组中的前后次序关系
  - 任何类型的指针都可以与0或NULL比较是否==或!=，判断指针值是否为“空”，既是否为空指针
- 下标运算[ ]
  - 等价性：a[i] ⇔ *(a+i)，*(p+i) ⇔ p[i]
  - 若指针p指向数组a的起始元素a[0]，a[i]就可以表示为*(p+i)
  - 由于p和a的值相等，p[i]和*(a+i)也可以表示a[i]

特殊指针

- **野指针**是一个不应存在的无效指针，如指针变量未赋值或指针运算超出范围（数组越限），对该指针操作的后果不确定
- **空指针NULL**是C指针类型中的一个特殊值，表示指针变量的值为空，即不指向任何内存单元，常用于初始化指针变量

  如：`int *p=NULL;  //NULL的值为0，但不是地址0H`

- **空类型指针**是用void声明的一种通用指针变量（万能指针）

  声明形式：`void *变量名;`

  - 其它任意类型的指针都可以直接赋值给void指针：`void *p = &n;`
  - void指针赋值给其它类型指针时，必须进行强制类型转换
  - void指针不能直接取值或运算，一般应将void指针强制转换成特定类型的指针后再操作：`int *p_int = (int *)p; // 转换回int*`

指针的安全使用原则

- 指针必须初始化：int *p = NULL;
- 使用前检查有效性：if (p != NULL) *p = 10;
- 释放后置为 NULL：free(p); p = NULL;
- 避免类型不匹配：int *p_int = p_char; // 尽量避免
- 谨慎使用类型转换：int *p_int = (int *)p_void; // 谨慎使用
- 擅用const 修饰符修饰指针：

```
// 指针指向的内容不可修改
const int *p = NULL; int num = 10;

p = &num; // 允许修改指针所指向的变量
// *p = 20;// 错误：不可修改指向的值
```

指向常量的指针

```
// 指针本身不可修改
int *const p = &num; // 初始化指针

*p = 20; // 允许修改指针指向变量的值
// p = &other;// 错误：不可修改指针指向
```

指针常量

本章内容概述

- 指针基本概念回顾
- 指向数组的指针与指针数组
- 二级与多级指针
- 带指针的函数与函数指针
- 高级内存管理技术
  - 动态内存分配
  - 内存池管理
- 应用实例：链表
- 内存调试与优化

指向数组的指针

- 声明变量时的数组名是一个符号地址常量，可以理解为指针常量，指向数组的首元素地址，是右值，不能对数组名赋值
- 除了数组名的值不能改变外，数组名和指针在很多地方可以通用，例如 `int a[5], *iptr = a;`
  - `iptr = a;` 等价于 `iptr = &a[0];` 即取元素地址
  - `a[3]` ⇔ `*(a+3)` ⇔ `*(iptr+3)` ⇔ `iptr[3]`
  - `*(iptr+3)` 的运算过程：取出指针变量 `iptr` 的值，加上 `3×sizeof(int)`，得到新地址 `&a[3]`，再取值得到内容 `a[3]`
  - `a[3]`的运算过程：取出数组首地址，加上 `3×sizeof(int)`，得到新地址 `&a[3]`，再取值得到内容 `a[3]`

指向数组的指针（例一）

- 例：指针与数组的效率比较

```
#include<stdio.h>
void main() {
     int a[5] = {1, 2, 3, 4, 5};
     int i, *p;
     for (i=0;i<5;i++) printf("%6d", a[i]);   //下标法
     printf("\n");
     for (i=0;i<5;i++) printf("%6d", *(a+i));   //数组名法
     putchar('\n');
     for (p=a;p<a+5;p++) printf("%6d", *p);   //指针变量法
     printf("\n");
}  // 3条printf运行结果一样
```

效率相同

效率略高

- ☑ 若是按递增或递减顺序访问数组，用指针，效率高
- ☑ 若不确定多维数组的高维大小，用指针，灵活
- ☑ 若随机访问数组，用下标，简洁明了

指向数组的指针（例二）

- 例：猜意图，找错误

```
#include<stdio.h>
void main() {
    int i, a[10], *p;
    p=a;
    for(i=0; i<6; i++) scanf("%d", p++);
    … …
    for(i=0; i<6; i++) printf("%6d\t", *p++);
    printf("\n");
}
```

☑ 指针未复位

☑ 数组越界

指向数组的指针（例三）

- 例：利用一维指针访问二维数据，并输出

```
#include <stdio.h>
void main() {
     int a[3][4], i, j, *p, n = 0;
     for (i = 0; i < 3; i++)
          for (j = 0; j < 4; j++)
               *(*(a+i)+j)=i*4+j;
     for (p = &a[0][0]; p <= &a[2][3]; p++) {
          printf("%6d", *p);
          if (++n%4 == 0) printf("\n");
     }
}
```

程序运行结果

```
0     1     2     3
4     5     6     7
8     9    10    11
```

指向数组的指针（例四）

- 例：要求不使用下标运算，仅通过指针移动来实现字符串的原地逆序

```
void reverse_string(char *str) {
    // 1. 安全检查：如果字符串为空或长度为0，直接返回
    if (str == NULL || *str == '\0') {return;}
    char *head = str, *tail = str;   // 指向字符串首字符
    // 2. 将 tail 移动到字符串的最后一个有效字符上（'\0'的前一个）
    while (*(tail + 1) != '\0') { tail++;}
    // 3. 交换 head 和 tail 指向的内容，直到它们相遇
    while (head < tail) {
        // 经典的交换逻辑
        char temp = *head;*head = *tail;*tail = temp;
        // 4. 移动指针
        head++; // 头指针向后移
        tail--; // 尾指针向前移
    }
}
```

☑ 效率：避免每次循环都去计算 str[i]（即基地址 + 偏移量），直接自增指针更快
☑ 通用性：这种“头尾指针”的思想是 所有双指针算法的基础

指向二维数组的指针

- 定义形式：类型名 (*变量名)[数组大小]
  - 如：int (*pa)[4]，pa 是指向“由4个整型量构成的数组”的指针变量
    - 若有 int a[2][4]，则可以 pa = a;
  - 如：char (*next)[16]，next 是指向“由16个字符构成的数组”的指针变量
    - 若有 char n[5][16]，则可以 next = n;

指向二维数组的指针（例）

- 例：被调函数的形参分别用指向基本类型的指针变量和指向数组的指针变量实现二维数组传递

```
void print_1(int *p, int n) {
     … *(p+i) …
}

void print_2(int (*p)[4], int m, int n) {
     … *(*(p+i)+j) …
}

void main() {
     int a[3][4];
     int (*pa)[4];
     … pa=a; print_1(*pa++, 4) …
     … print_2(a, 3, 4) …
}
```

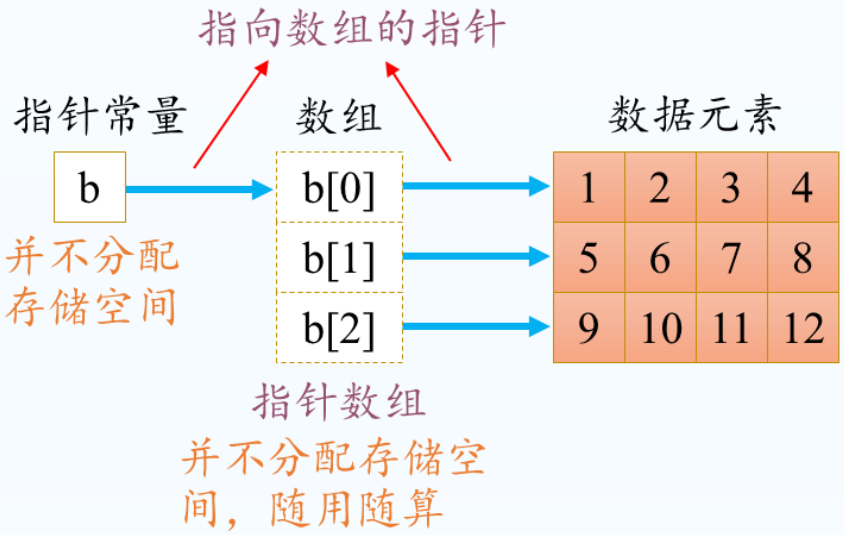
指向二维数组的指针

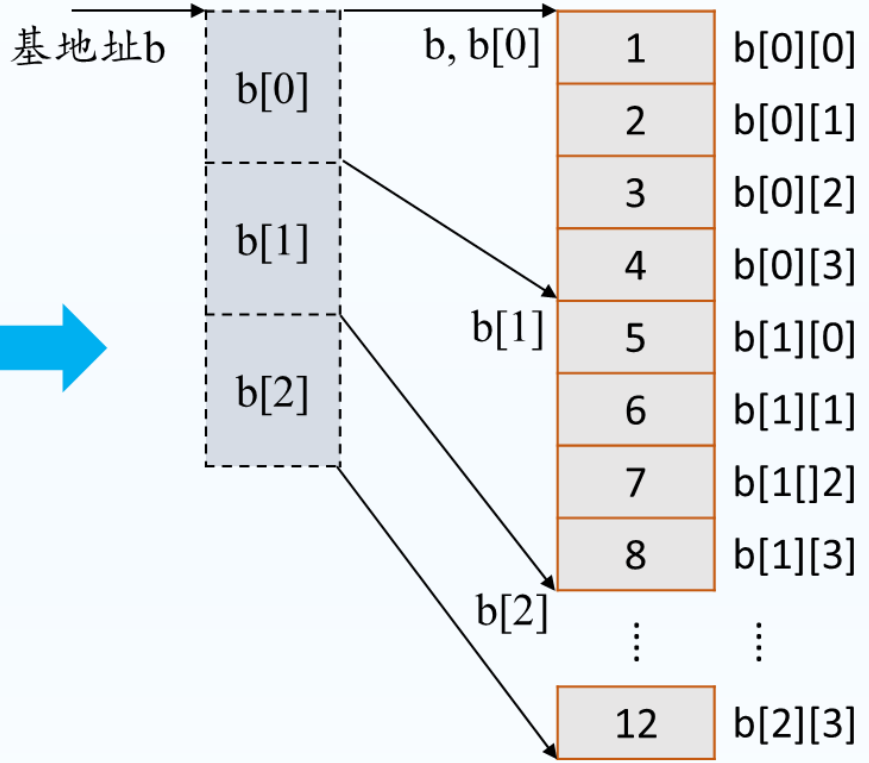
- 数组取值时，每处理一个[]或 *，相当于一次指针取值运算
  - 如 int a[3][4];
  - a、a+i，指向“由4个整型数构成的数组”的指针
  - a[i]、a[i]+j ，指向整型数的指针，即 *(a+i)、*(a+i)+j
  - a[i][j]，整型数，即 *(*(a+i)+j) 或 *(*a+i*4+j)
- 数组取地址时，&表示指向该类型元素的指针
  - 如 int a[3][4];
  - &a[i][j]，指向整型数的指针，值为 *(a+i)+j 或 a[i]+j
  - &a[i]，指向“由4个整型数构成的数组”的指针，值为 a+i
  - &a，指向二维数组的指针，值为 a
    - 注意 &a 与 &a[0] 的差异
    - a[0] = * (a + 0)，二者值相等且类型一样，可以有 &(a+0) 吗?

```
printf("%p", &(a+0));
// error: lvalue required as
unary ‘&’ operand
```

指向多维数组的指针

- 多维数组虽然也是连续存储在内存中的元素集合，但语法上（编程使用时）是元素为数组的数组
- 例如：int b[3][4]={{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};





指向多维数组的指针（例）

- 例：多维数组的指针进行赋值时，应特别注意类型的匹配
  - pa 将以自己的类型（指向包含两个整型数的数组的指针）去理解 b 和 c 数组中的数据

```
int a[2][2] = {{1,2},{3,4}};
int b[3][3] = {2,3,4,5,6,7,8,9,10};
int c[4] = {12,13,14,15}
int (*pa)[2];

pa = a;
pa = b; //警告但通过编译
pa = c; //警告但通过编译
```

指向字符串常量的指针

- 可以定义一个字符指针变量，让其指向一个字符串常量
  - 例如：`const char *msg = “Hello”; // 推荐写法`

    `msg[0] = ‘h’; // 错误：试图修改字符串常量`
  - 实际上就是把字符串常量在内存中的首地址赋给该指针
  - 其地位等价于指向一维字符数组第一个元素的指针
  - 注意这个指针指向常量区，指针值可以改，指针指向的内容不可以改
- 字符数组与字符串常量的差别
  - 字符数组存放在程序活动的栈中，可以修改
  - 字符串常数存放在常量区，不可以修改

指向字符串常量的指针

- 当字符指针变量指向一个字符串（不需要是首字符）后，可以用数组下标的形式来引用该字符串中的字符

```
const char *pstr = "Hello, world!";
char str[40];
int i;
for(i = 0; *(pstr+i) != '\0'; i++) {
    str[i] = pstr[i];
}
str[i] = '\0'; //此句不能少
//注意，*(pstr+i)等价于pstr[i]，即访问字符串中第i个字符
```

指向字符串常量的指针

- 可以用字符串常量对字符指针变量和字符数组初始化
- 可以用字符串常量对字符指针变量赋值
- 不可以用字符串常量对数组名赋值（试图修改数组名常量）

```
char *pstr1 = "Hello", str1[40] = "Okay";
char *pstr2, str2[40];
pstr2 = "world";
str2 = "world";  //错误!
```

- 只能逐个元素地对数组赋值（比如for循环等方法）

指针数组

- 可以定义由指针作为元素的数组，具体的声明形式：

  类型说明符 *数组名[数组长度]

  例：
```
int *p[3]; char *name[]=...;
    // 注意 int (*p)[3] 与 int *p[3] 的区别
```

- 指针数组中的元素都是指向同一类型的指针，常见的应用：
  - 若有大量数据（例如学生信息）需要按某个元素排序时（例如按姓名拼音顺序），使用指针数组排序，就可以避免大量数据的搬移
  - 当需要对多个已存在的对象进行批量处理时，也可以使用指针数据
    - 例：游戏程序中各自独立行动的人物（不使用多线程），每人一个指针组成数组，就可以在一个循环中顺序处理所有人物
    - 例：多线程程序使用一个指针数组/链表来顺序处理各个进程

指针数组

- 指针数组常用于处理一组字符串，比使用二维数组更方便，也节省内存空间，在排序等操作中效率更高
  - 例：`char *name[]={“allen”, “word”, “jones”, “martin”};`
  - 字符串常量 “allen” 等又称为字符串字面量，它们被存储于常量区
  - 在代码中，字符串字面量可以等同于一个 `char *` 指针
    - 甚至允许 “allen”[3] = ‘e’ （虽然编译可以通过，但是执行会发生错误，即常量区不允许改写）
  - 在这里，相当于在初始化器中列举了一系列字符指针，它们分别指向常数区的字符串

指针数组（例）

- 例：使用指针数组存储指向字符串的指针的优点
  - 允许不等长的字符串以相对规整的方式组织在一起，注意它们不一定存储在一起，便于调整各字符串在数组内的位置，修改指针即可

```
#include<string.h>
void main() {
  int n = 4;
  char *name[] = { "allen", "word", "jones", "martin" };
  for (int i = 0; i < n - 1; i++) {
    for (int j = i + 1; j < n; j++)
      if (strcmp(name[k], name[j]) > 0) {
        char *t = name[i];
        name[i] = name[k];
        name[k] = t;
      }
    }
}
```

本章内容概述

- 指针基本概念回顾
- 指向数组的指针与指针数组
- 二级与多级指针
- 带指针的函数与函数指针
- 高级内存管理技术
  - 动态内存分配
  - 内存池管理
- 应用实例：链表
- 内存调试与优化

通过 ** 可以跨越两个层级直接操作原始数据

二级指针

- 二级指针也称为：指向指针变量的指针变量
- 数据类型：必须与一级指针指向的数据类型一致
- 初始化要求：必须指向已存在的一级指针的地址

- 定义的一般形式：

  类型名 **指针变量名;

```
int num = 10;
int *p1 = &num; // 一级指针p1指向num
int **p2 = &p1; // 二级指针p2指向一级指针p1
// *p2：第一次获取p2指向的一级指针p1的值（即num的地址）
// **p2：第二次通过p1的值（地址）获取num的实际值10
printf("%d\n", **p2); // 输出10
```

二级指针（例一）

- 例：动态创建一个3行4列的二维数组（矩阵）

```
// 创建3行4列的二维数组（矩阵）
int rows = 3, cols = 4;
// 分配行指针数组
int **matrix = malloc(rows * sizeof(int *));
for (int i = 0; i < rows; i++) {
      // 分配每行的列内存
      matrix[i] = malloc(cols * sizeof(int));
}

// 初始化数组
for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
            matrix[i][j] = i * cols + j + 1;
      }
}
```

二级指针（例一续）

- 例：动态创建一个3行4列的二维数组（矩阵）

```
// 访问元素
printf("%d\n", matrix[1][2]); // 输出7
for (int i = 0; i < rows; i++) {
  for (int j = 0; j < cols; j++) {
    printf("%d\n", matrix[i][j]); // 遍历元素
  }
}

// 释放内存
for (int i = 0; i < rows; i++) {
  free(matrix[i]); // 先释放列内存
}
free(matrix); // 再释放行内存
matrix = NULL; // 置为空指针
```

二级指针（例二）

- 例： 5个学生，每人所修课程门数不同（成绩存放在数组中，以-1表示结束），编写程序输出他们的各项成绩
  - 每人的成绩各用一个一维数组保存
  - 指针数组 int *gread[]，每个元素指向一个学生的数组
  - 指向指针的指针int **p=gread，用于指向gread的元素

```
int stu1[]={...}, stu2[]={...}, stu3[]={...},
     stu4[]={...}, stu5[]={...};
int *gread[]={stu1, stu2, stu3, stu4, stu5};
int **p=gread, i;
```

二级指针（例二续）

- 例： 5个学生，课程数不同，编写程序输出他们的各项成绩
  - **p 取出一个成绩；(*p)++ 修改了 gread 数组中某一个元素的值，以指向该学生的下一项成绩；p++ 使得p指向gread中的下一个元素

```
for (i=1; i<=5; i++) {
    printf("student %d gread:",i);
    while (**p>0) {
        printf("%4d", **p);
        (*p)++;
    }
    p++;
    pirntf("\n");
}
```

```
int stu1[]={...}, stu2[]={...}, stu3[]={...},
    stu4[]={...}, stu5[]={...};
int *gread[]={stu1, stu2, stu3, stu4, stu5};
int **p=gread, i;
```

二级指针的安全使用原则

- 初始化规则：二级指针必须指向有效的一级指针地址
  - 例：
```
int **p2; **p2 = 10;
    // 错误：p2未初始化，指向不确定地址
```
- 双重检查：解引用前检查两级指针有效性
  - 例：`if (p2 != NULL && *p2 != NULL) { **p2 = 10;  }`
- 内存释放顺序：先释放底层内存，再释放上层内存
  - 例：
```
free(matrix); // 使用后仅释放行指针
    // 错误：列内存未释放，导致内存泄漏
```
- 类型一致性：确保二级指针类型与一级指针类型完全匹配、
  - 例：
```
int *p1; char **p2 = (char **)&p1;
    // 错误：类型不匹配，可能导致访问错误
```

多级指针

- 理论上：C 语言支持无限级的指针（ANSI C 标准要求实现支持至少12级指针），如 `int ***p3; int ****p4; … ; int *****p5; …`
- 现实情况：在实际工程中，三级指针已经非常少见超过三级的指针往往意味着程序设计过于复杂，建议重新审视架构
- 三级指针是 指向二级指针的指针，它存储的是二级指针的内存地址。在处理三维数组、链表的链表等复杂数据结构时具有不可替代的作用。

```
int num = 10;
int *p1 = &num;                 // 一级指针p1指向num
int **p2 = &p1;                 // 二级指针p2指向p1
int ***p3 = &p2;                // 三级指针p3指向p2
// *p3: 获取 p3 指向的二级指针 p2 的值（即 p1 的地址）
// **p3: 通过 p2 的值（地址）获取一级指针 p1 的值（即 num 的地址）
// ***p3: 通过 p1 的值（地址）获取 num 的实际值
printf("%d\n", ***p3);    // 输出10
```

多级指针（例一）

- 例：动态创建2层3行4列的三维数组（张量）

```
// 分配三维数组（张量）内存
void allocate_3d_array(int ****arr, int x, int y, int z, int v) {
    *arr = (int ***)malloc(x * sizeof(int **)); // 分配层指针数组
    for (int i = 0; i < x; i++) {
        (*arr)[i] = (int **)malloc(y * sizeof(int *)); //分配行指针数组
        for (int j = 0; j < y; j++) {
            (*arr)[i][j] = (int *)malloc(z * sizeof(int));//分配列指针数组
        }
    }
    for (int i = 0; i < layers; i++) { // 初始化数组
        for (int j = 0; j < rows; j++) {
            for (int k = 0; k < cols; k++) {
                tensor[i][j][k] = v;
            }
        }
    }
}
```

多级指针（例一续）

- 例：动态创建2层3行4列的三维数组（张量）

```
// 释放三维数组（张量）内存
void free_3d_array(int ****arr, int x, int y) {
    for (int i = 0; i < x; i++) {
        for (int j = 0; j < y; j++) {
            free((*arr)[i][j]); // 释放列指针数组
        }
        free((*arr)[i]); // 释放行指针数组
    }
    free(*arr); // 释放层指针数组
    *arr = NULL; // 置为空指针
}
int main() {
    int ***tensor;
    allocate_3d_array(&tensor, 2, 3, 4, 0);
    // 使用tensor...
    free_3d_array(&tensor, 2, 3);
    return 0;
}
```

多级指针（例二）

- 例：创建复杂的数据结构

```
typedef struct {
    int id;
    char *name;
} Person;

typedef struct {
    Person **people;  // 二级指针，指向Person数组
    int count;
} Team;

typedef struct {
    Team ***teams;    // 三级指针，指向多个Team数组
    int team_count;
} Organization;
```

多级指针（例二续）

- 例：创建复杂的数据结构

```
// 初始化Organization
Organization init_organization(int num_teams, int num_people) {
    Organization org;
    org.team_count = num_teams;
    org.teams = malloc(org.team_count * sizeof(Team **));
    for (int i = 0; i < org.team_count; i++) {
        org.teams[i] = malloc(num_teams * sizeof(Team *));
        for (int j = 0; j < num_teams; j++) {
            org.teams[i][j] = malloc(sizeof(Team));
            org.teams[i][j]->people =
                malloc(num_people * sizeof(Person *));
            org.teams[i][j]->count = num_people;
        }
    }
    return org;
}
```

多级指针的安全使用原则

- 初始化规则
  - N级指针必须指向有效的N-1级指针地址
- N重检查：
  - 使用前检查N级指针、N-1级指针、…、一级指针的有效性
- 内存释放顺序
  - 从最内层到最外层逐层释放内存
- 类型一致性：
  - 确保N级指针类型与指向的N-1级指针类型完全匹配
- 避免过度嵌套
  - 超过三级的指针嵌套会显著降低代码可读性
- 访问校验
  - 使用宏或函数进行安全访问校验

本章内容概述

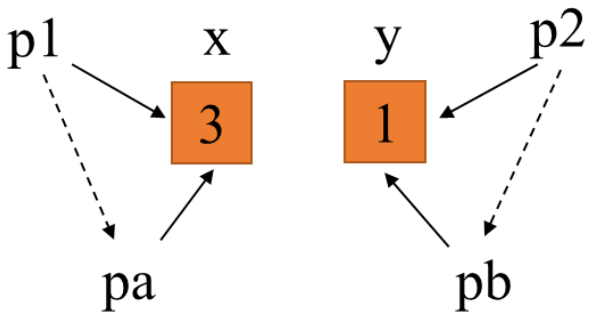
- 指针基本概念回顾
- 指向数组的指针与指针数组
- 二级与多级指针
- 带指针的函数与函数指针
- 高级内存管理技术
  - 动态内存分配
  - 内存池管理
- 应用实例：链表
- 内存调试与优化

带指针参数的函数

- 当指针作为函数的参数时，所传的是值，其内容是地址（也称传地址调用）
  - 被调函数通过指针访问函数外部的数据——具有数据双向传递的功效

```
#include<stdio.h>
void swap2(int *pa, int *pb) {
     int t;
     t=*pa; *pa=*pb; *pb=t;
}

void main() {
     int x=1, y=3, *p1=&x, *p2=&y;
     swap2(p1, p2);
     printf("%d,%d,%d,%d\n", x, y, *p1, *p2);
}
```



带指针参数的函数（例一）

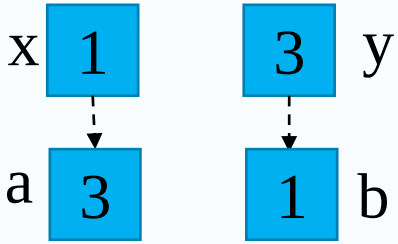
- 例：三种变量交换写法的区别

```
void swap1(int a, int b) {
    int t;
    t=a; a=b; b=t;
}
```

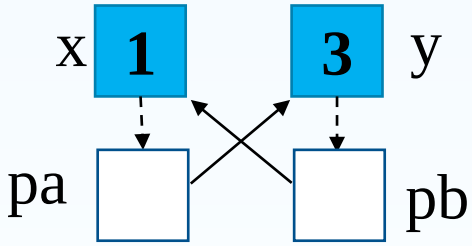
```
void swap2(int *pa, int *pb) {
    int t;
    t=*pa; *pa=*pb; *pb=t;
}
```

```
void swap3(int *pa,int *pb) {
    int *pt;
    pt=pa; pa=pb; pb=pt;
}
```

只交换形参a,b的值，未修改main函数中的变量值



交换了形参pa,pb指针的值，未修改pa,pb所指变量的值



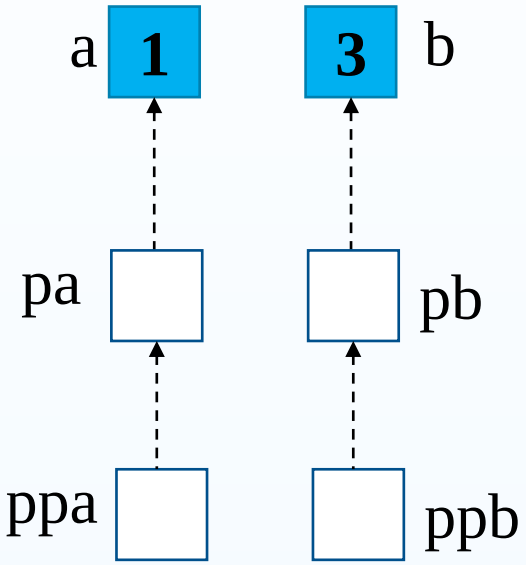
带指针参数的函数（例一续）

- 例：三种变量交换写法的区别

```
void swap1(int **ppa, int **ppb) {
     int t;
     t=**ppa; **ppa=**ppb; *ppb=t;
}
```

```
void swap2(int **ppa, int **ppb) {
     int *pt;
     pt=*ppa; *ppa=*ppb; *ppb=pt;
}
```

```
void swap3(int **ppa,int **ppb) {
     int **ppt;
     ppt=ppa; ppa=ppb; ppb=ppt;
}
```




```
int a = 1, b = 3;
int *pa = &a
int *pb = &b;
int **ppa = &pa;
int **ppb = &pb;
```

带指针参数的函数（例二）

- 例：借助void实现编写通用的数据交换函数（泛型编程）

```
void genswap(void *a, void *b, int size) {
    char t, *ac, *bc;
    int i;
    ac = (char *)a; bc = (char *)b;
    for (i = 0; i < size; i++, ac++, bc++) {
        t = *ac;
        *ac = *bc;
        *bc = t;
    }
}
```

```
int a, b; genswap(&a, &b, sizeof(a));
double c, d; genswap(&c, &d, sizeof(c));
int e[10], f[10]; genswap(e, f, sizeof(e));
struct Student g, h; genswap(&g, &h, sizeof(g));
```