东南大學 SCHOOL OF INTEGRATED CIRCUITS, SEU

集成电路学院

计算机科学基础I

—— 指针II

东南大学 集成电路学院 朱彬武

E-mail: bwzhu@seu.edu.cn

回顾——指针和地址


```
void swap(int *pA, int *pB);

int main(){
    int a = 3;
    int b = 4;
    swap(&a, &b);
    cout << a << " " << b;  //4 3
}

void swap(int *pA, int *pB) {
    int temp = *pA;
    *pA = *pB;
    *pB = temp;
    return;
}
```

- 如何获得变量的地址?
- 如何保存变量的地址?
- 如何通过变量地址访问变量?

禁止出现野指针，如果不知道初始化值，先初始化成空指针，操作系统规定0（NULL）地址不能被写

回顾——数组名：特殊的指针


```
int arr[5] = {1, 2, 3, 4, 5};
int *p = arr;

cout << *p << endl;      // 等价arr[0]
cout << p[0] << endl;    // 等价arr[0]
cout << p[1] << endl;    // 等价arr[1]

cout << *arr << endl;    //等价arr[0]
```

- 下标运算符[]可以对数组名做，也可以对指针做：`p[0] == arr[0]`
- *运算符可以对指针做，也可以对数组名做：`*p == *arr`

回顾——insert函数

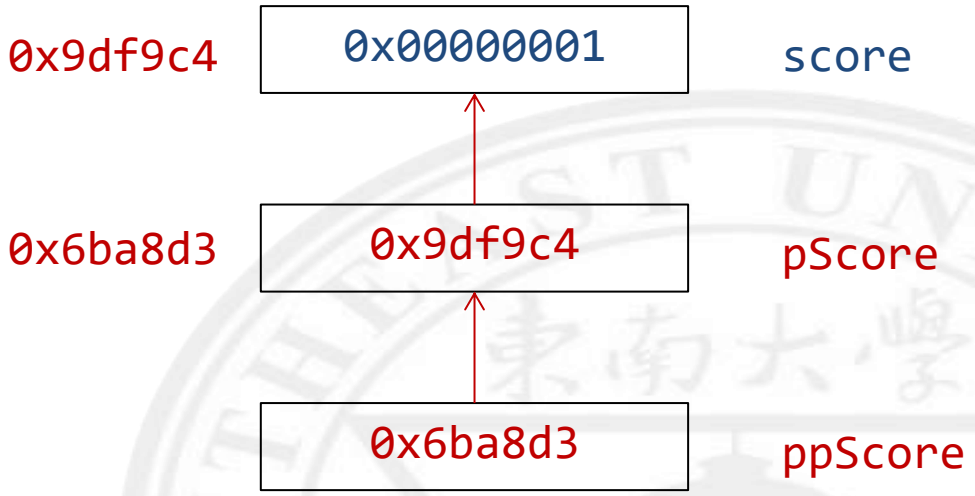

```
void insert(int a[], int len, int pos, int value)
{
    // 元素后移
    for (int i = len; i > pos; i--)
    {
        a[i] = a[i - 1];
    }

    // 插入新元素
    a[pos] = value;
}

int main(){
    int a[10] = {1, 2, 3, 4, 5};
    int value = 99, pos = 2;
    insert(a, 5, pos, value);

    cout << "in main function" << endl;
    for(int i = 0; i < 6; i++){
        cout << a[i] << " ";
    }
}
```

由于传入的是数组地址，因此函数中通过该地址对数组元素进行的修改，会直接作用在主函数中的原数组上。

二级指针

- 存放“指针变量地址”的变量，这类变量称为二级指针：<类型> **<指针名>

```
int score = 1;
int *pScore = &score; //一级指针
int **ppScore = &pScore; //二级指针

cout << *pScore;
cout << *ppScore;
cout << **ppScore;
```

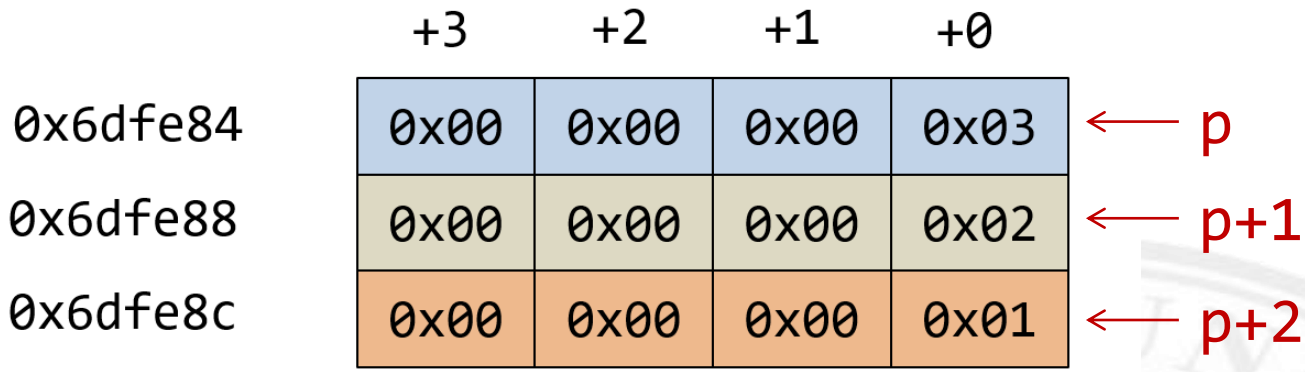


- 访问二级指针指向的变量：使用两次解引用运算符

指针运算

指针 + 1

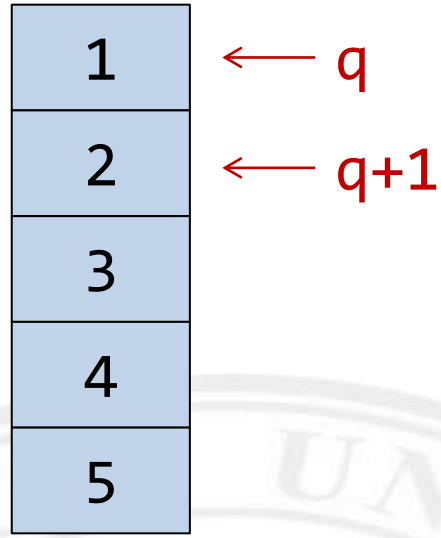

```
int i = 1, *p = &i;
cout << p << endl;
cout << p + 1 << endl;

short j = 2, *q = &j;
cout << q << endl;
cout << q + 1 << endl;
```



- 指针 + 1，并不是简单地把地址加 1，而是地址增加 sizeof(指针所指向的类型) 个字节

数组名 + 1


```
int a[5] = {1,2,3,4,5};
int *q = a;
cout << *(q + 1) << endl;
cout << *(a + 2) << endl;
```



- 解引用运算符是单目运算符，优先级高于算术运算符
- 如果指针不是指向一片连续分配的空间（如数组），则这种运算没有意义且很危险

指针的更多运算

- 给指针加、减一个整数（+, +=, -, -=）
- 两个指针相减：&a[3] - &a[0] ==> 3
- 递增递减（++/--）：*的优先级虽然高，但是没有++高
  *p++ ==> *(p++)
- <, <=, ==, >, >=, !=都可以对指针做，来比较它们在内存中的地址

```
int n = sizeof(a) / sizeof(a[0]);
for (int* p = a; p < a + n; ++p) {
    cout << *p << endl;
}
```

```
int a[5] = {1,2,3,4,5};
int *q = a;
cout << *q++ << endl; // 1
cout << *q++ << endl; // 2
cout << *q++ << endl; // 3
```

二维数组名

二维数组

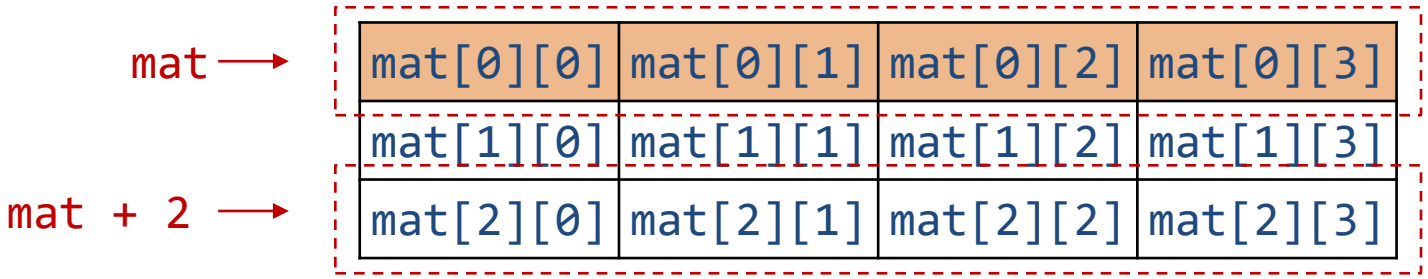
- 二维数组可以理解为每一个数组元素都是一个一维数组
- 数组中的每个变量在内存里按行连续存放

```
int mat[3][4] = {{1,2,3,4},
                 {5,6,7,8},
                 {9,10,11,12}};
```

mat[0][0]
mat[0][1]
……
mat[1][0]
mat[1][1]
……
mat[2][2]
mat[2][3]

二维数组名

- 数组名存的是首元素的地址，二维数组的首元素是什么？
- int mat[3][4]，mat存的是具有4个int的一维数组的地址。
  所以mat的类型是 int (*)[4]

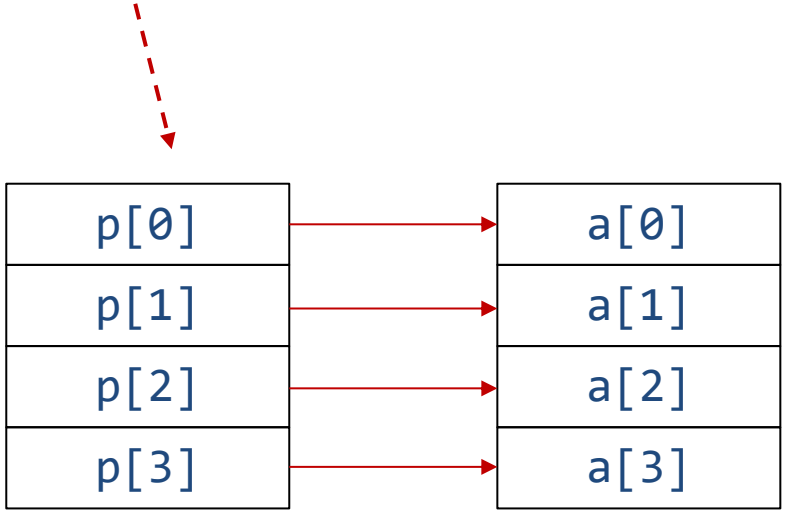



```
int mat[3][4] = {0};

int (*p)[4] = mat; ✅

cout << mat << endl; //0x9df9c4

cout << mat + 1 << endl; //0x9df9d4
```

数组指针和指针数组

注意区分int (*)[4]和int *[4]的区别：

- 前者表示一个数组指针，指向一个包含4个int元素的数组；
- 后者表示指针数组，存了四个int *类型的指针；




```
int mat[3][4] = {0};
int (*p)[4] = mat;  ✅
int *p[4] = mat;  ❌ error
```

数组名不同于二级指针

- 不要把mat理解成二级指针int **

  mat == &mat[0], mat[0] == &mat[0][0]

- 换个角度，如果mat是一个int **, mat + 1 就是 mat + sizeof(int *)

  这样就丢失了列维度的信息，二维数组名必须保存“每一行有多长”。

二维数组名的运算

对于二维数组mat

- `mat + i ==> &mat[i]`：返回结果是第i个一维数组的地址
- `*(mat + i) ==> mat[i]`：第i个一维数组，返回结果是第i个一维数组首元素的地址
- `*(*(mat + i) + j) ==> mat[i][j]`：第i个一维数组的第j个元素

运算实例


```
int a[2][3] = { {1, 2, 3}, {4, 5, 6} };
int (*p)[3] = a;
cout << a[0][1] << endl;
cout << p[0][1] << endl;
cout << *(*a + 1) << endl;
cout << *(*p + 1) << endl;
```

- 下标运算符[]可以对数组名做，也可以对指针做：`p[0][1] == a[0][1]`
- 解引用运算符可以对指针做，也可以对数组名做：`*(*p + 1) == *(*a + 1)`

二维数组作为函数参数


```
void traverse(int arr[][3], int rows)
{
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }
}
```

参数表里的int arr[][3]，实际上等价于int (*arr)[3]

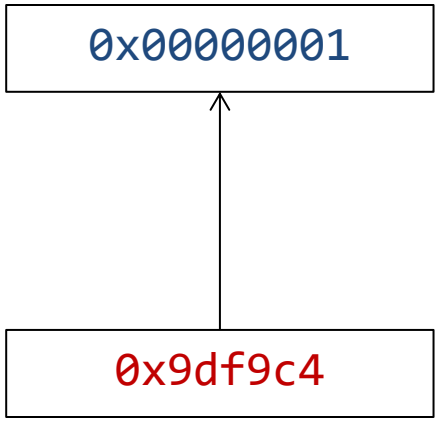
右边这四种函数声明（原型）是等价的，后两种是省略了参数名的版本。

```
void traverse(int arr[][3], int rows);

void traverse(int (*arr)[3], int rows);

void traverse(int [][3], int);

void traverse(int (*)[3], int);
```

指针常量和常量指针

指针和const

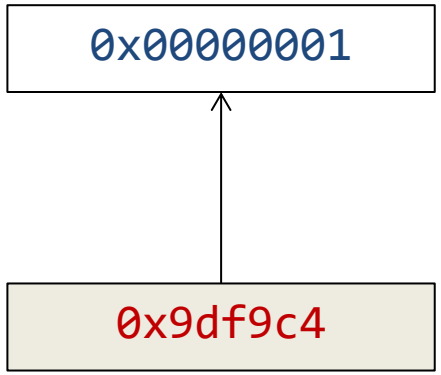
- const是一个修饰符，用来给这个变量加上一个不变的属性。
- const类型的变量必须要初始化，且初始化之后就不能被赋值。



所指可以是const

地址可以是const

指针常量：指针是const



地址是const

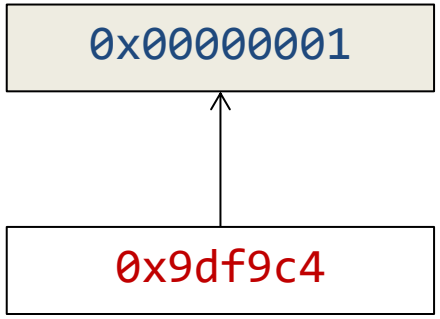
- 指针常量：一旦指针得到某个变量的地址，就不能再指向其他变量
- 语法：<类型> * const <指针名> = 初始化地址

```
int i = 1, j = 2;
int * const pI = &i;  // pI存的地址是const
pI = &j;   ❌ error
*pI = 2;   ✅
```

pI指向的不是const，因此可以被修改

- 数组名是指针常量，不能对数组名赋值

常量指针：所指是const



所指是const
（站在指针的视角看）

- 常量指针：表示不能通过这个指针去修改那个变量（变量是不是const都可以）
- 语法：const <类型> * <指针名>

  <类型> const * <指针名>

```
int i = 1, j = 2;
const int *p = &i;
*p = 2;  ❌ error
i = 2;   ✅
p = &j;  ✅
```

快速辨认


```
int score = 1;
const int *pScore = &score;
int * const pScore = &score;
int const *pScore = &score;
const int * const pScore = &score;

const int grade = 2;
int * pGrade = &grade;
```

- 分辨技巧：判断哪个被赋予了const属性就看const在*的前面还是后面
- 可以把一个非const变量的地址赋给常量指针，反之不行（可以减小权限，但不能放大权限）

常量指针：函数传参更安全

- 只读访问，防止误修改数据
- 在必要的地方加上const可以明确约束，代码看起来更专业

```
void traverse(const int* arr, const int n) {
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}
```

另一种写法：`const int arr[]`

通用指针

通用指针

- 不同类型的指针不应该相互赋值，因为它们指向的数据类型和解释方式不同。
- 但有一类特殊类型的指针：void *，void *是通用指针，不知道指向什么数据类型，仅仅保存一个地址

```
int a = 10;
double *p = &a; ❌ error
void *p = &a;   ✅
int *q = (int *)p;
cout << *q << endl;
```