第八章 程序设计复合类型-指针

模块8.2：指针与数组

主讲教师：同济大学计算机科学与技术学院 陈宇飞
同济大学计算机科学与技术学院 龚晓亮

目录

目录

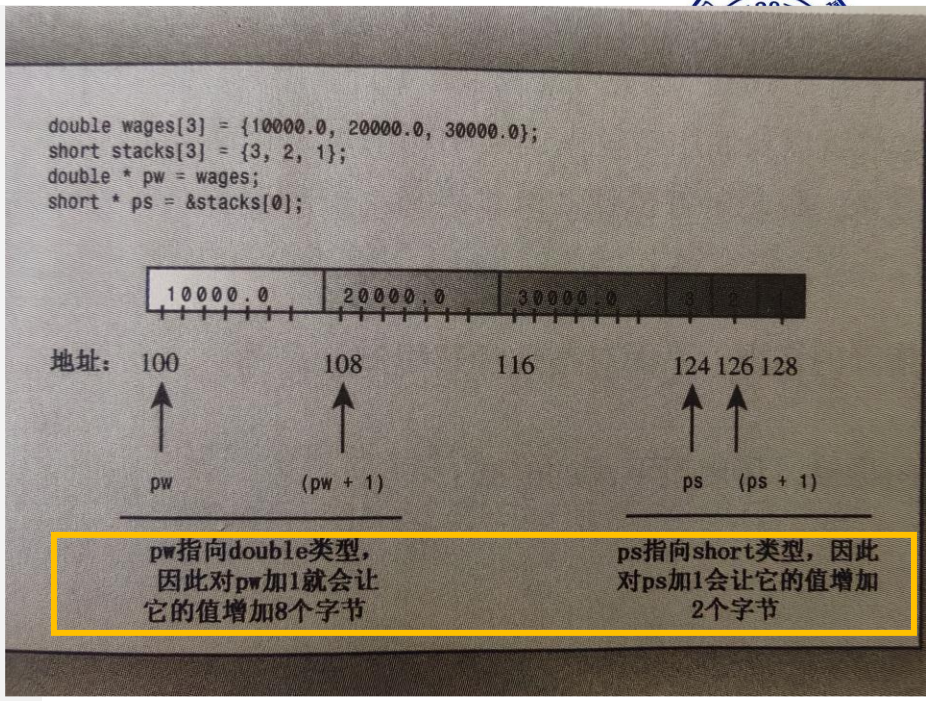
- 一维数组与指针

part1

```
// addpntrs.cpp -- pointer addition
#include <iostream>
int main()
{
    using namespace std;
    double wages[3] = { 10000.0, 20000.0, 30000.0 };
    short stacks[3] = { 3, 2, 1 };

    // Here are two ways to get the address of an array
    double* pw = wages;     // name of an array = address
    short* ps = &stacks[0]; // or use address operator
    // with array element
    cout << "pw = " << pw << ", *pw = " << *pw << endl;
    pw = pw + 1;
    cout << "add 1 to the pw pointer:\n";
    cout << "pw = " << pw << ", *pw = " << *pw << "\n\n";

    cout << "ps = " << ps << ", *ps = " << *ps << endl;
    ps = ps + 1;
    cout << "add 1 to the ps pointer:\n";
    cout << "ps = " << ps << ", *ps = " << *ps << "\n\n";
```

part2

```
    cout << "access two elements with array notation\n";
    cout << "stacks[0] = " << stacks[0]
         << ", stacks[1] = " << stacks[1] << endl;
    cout << "access two elements with pointer notation\n";
    cout << "*stacks = " << *stacks
         << ", *(stacks + 1) =  " << *(stacks + 1) << endl;

    cout << sizeof(wages) << " = size of wages array\n";
    cout << sizeof(pw) << " = size of pw pointer\n";

    return 0;
}
```



注意：将指针变量加1后，其增加的值等于指向类型占用的字节数

```
double wages[3] = { 10000.0, 20000.0, 30000.0 };
short stacks[3] = { 3, 2, 1 };
```

```
ps = 006FFC1C, *ps = 3
add 1 to the ps pointer:
ps = 006FFC1E, *ps = 2

access two elements with array notation
stacks[0] = 3, stacks[1] = 2
access two elements with pointer notation
*stacks = 3, *(stacks + 1) =  2
24 = size of wages array
4 = size of pw pointer
```

1.1 一维数组与指针

✓ 一维数组与指针的相同点

❖在多数表达式中指针名和数组名都表示地址，可以使用数组方括号表示法，也可以使用解除引用运算符（*）

```
arrayname[i] <----------> * (arrayname +i)
stacks[1]    <----------> * (stacks+1)
```

```
pintername[i] <----------> * (pointername +i)
pw[1]         <----------> * (pw+1)
```

1.1 一维数组与指针

✓ 一维数组与指针的区别

❖可以修改指针的值,而数组名是常量

```
pointername = pointername + 1 ; // vaild
arrayname = arrayname +1; // not allowed
```

❖对数组应用sizeof运算符得到的是数组的长度，而对指针应用sizeof得到的是指针的长度，即使指针指向的是一个数组。这种情况下，C++不会将数组名解释为地址

```
24 = size of wages arrary    // displaying sizeof wages
4 = size of pw pointer       // displaying sizeof pw
```

1.1 一维数组与指针

✓ 数组的地址

❖数组名被解释为其第一个元素的地址，而对数组名应用地址运算符时，得到的时整个数组的地址

```
short tell[10]; // tell an array of 20 bytes
cout << tell << endl; // displays &tell[0]
cout << &tell << endl; // displays address of whole array
```

❖从数字上说，这两个地址相同；但从概念上来说，&tell[0]是一个2字节内存块的地址，而&tell时一个20字节内存的地址

1.1 一维数组与指针

✓ 数组的地址

```
short tell[10];
short (*pas)[10] = &tell; // pas points to array of 10 shorts
short *pas[10]; // pas 是一个short指针数组，它包含10个元素
```

一层层看！！！

```
short (*pas)[10]     //本质是指针，指向一个长度为10的short数组
short *pas[10]       //本质是数组，包含10个元素，每个元素都是short*
```

目录

- 二维数组的地址
  - 二维数组的基本概念
  - 二维数组的地址（行地址/元素地址）

2.1 二维数组的基本概念

- 一维数组与指针

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

1	2	3	4	5	6	7	8	9	10	11	12
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]

- 从一维数组到二维数组

```
int a[3][4]={略}
```

	0列	1列	2列	3列
0行	1 a[0][0]	2 a[0][1]	3 a[0][2]	4 a[0][3]
1行	5 a[1][0]	6 a[1][1]	7 a[1][2]	8 a[1][3]
2行	9 a[2][0]	10 a[2][1]	11 a[2][2]	12 a[2][3]

2.1 二维数组的基本概念

思考： 如何正确的理解二维数组？

- 二维数组在内存中存放时先行后列，占用一块连续的空间

元素	地址	值
a[0][0]	2000 2003	1
a[0][1]	2004 2007	2
a[0][2]	2008 2011	3
a[0][3]	2012 2015	4
a[1][0]	2016 2019	5
a[1][1]	2020 2023	6
a[1][2]	2024 2027	7
a[1][3]	2028 2031	8
a[2][0]	2032 2035	9
a[2][1]	2036 2039	10
a[2][2]	2040 2043	11
a[2][3]	2044 2047	12

2.1 二维数组的基本概念

思考： 如何正确的理解二维数组？

- 二维数组在内存中存放时先行后列，占用一块连续的空间
- 二维数组 int a[3][4]，理解为一维数组，有3(行)个元素，每个元素又是一维数组，有4(列)个元素

  a是二维数组名

  a[0],a[1],a[2]是一维数组名

行	元素	地址	值
a[0]	a[0][0]	2000 2003	1
	a[0][1]	2004 2007	2
	a[0][2]	2008 2011	3
	a[0][3]	2012 2015	4
a[1]	a[1][0]	2016 2019	5
	a[1][1]	2020 2023	6
	a[1][2]	2024 2027	7
	a[1][3]	2028 2031	8
a[2]	a[2][0]	2032 2035	9
	a[2][1]	2036 2039	10
	a[2][2]	2040 2043	11
	a[2][3]	2044 2047	12

2.1 二维数组的基本概念

思考：如何正确的理解二维数组？

➢ 二维数组在内存中存放时先行后列，占用一块连续的空间

➢ 二维数组 int ... ）

个元素，每个元 ... 元素

a是二维数组名

a[0],a[1],a[2]是一维数组名

问题：
(1)如何区分二维数组的行地址和元素地址？
(2)如何使用指针正确的访问数组元素？

行	元素	地址	值
a[0]	a[0][0]	2000 2003	1
	a[0][1]	2004 2007	2
	a[0][2]	2008 2011	3
	a[0][3]	2012 2015	4
a[1]	a[1][0]	2016 2019	5
	a[1][1]	2020 2023	6
	a[1][2]	2024 2027	7
	a[1][3]	2028 2031	8
a[2]	a[2][0]	2032 2035	9
	a[2][1]	2036 2039	10
	a[2][2]	2040 2043	11
	a[2][3]	2044 2047	12

2.2 二维数组的地址

- 行地址：

a ： ① 二维数组的数组名，即a

② 3元素一维数组的数组名，即a

③ 3元素一维数组的首元素地址，即&a[0]

&a[i]：3元素一维数组的第i个元素的地址

a+i：同上

行	元素
a[0]	a[0][0]
	a[0][1]
	a[0][2]
	a[0][3]
a[1]	a[1][0]
	a[1][1]
	a[1][2]
	a[1][3]
a[2]	a[2][0]
	a[2][1]
	a[2][2]
	a[2][3]

2.2 二维数组的地址

- 元素地址：

  a[i]：① 3元素一维数组的第i个元素的值

  ② 4元素一维数组的数组名

  ③ 4元素一维数组的首元素的地址

  *(a+i)：同上

  a[i]+j ：第i行第j列元素的地址

  *(a+i)+j ：同上

  &a[i][j] ：同上

行	元素
a[0]	a[0][0]
	a[0][1]
	a[0][2]
	a[0][3]
a[1]	a[1][0]
	a[1][1]
	a[1][2]
	a[1][3]
a[2]	a[2][0]
	a[2][1]
	a[2][2]
	a[2][3]

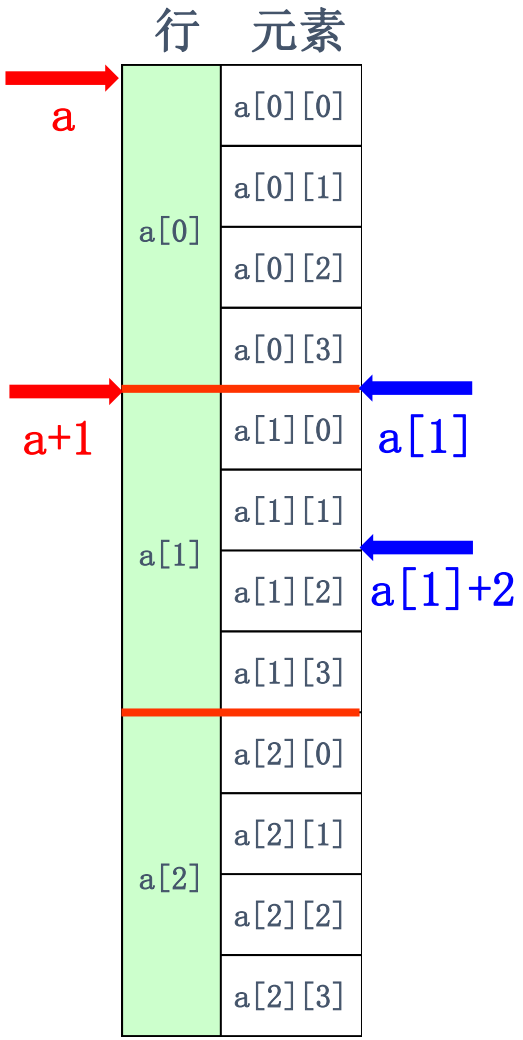
2.2 二维数组的地址

- 地址增量的变化规律：

对二维数组a[m][n]：

a+i 实际 a+i*n*sizeof(基类型)

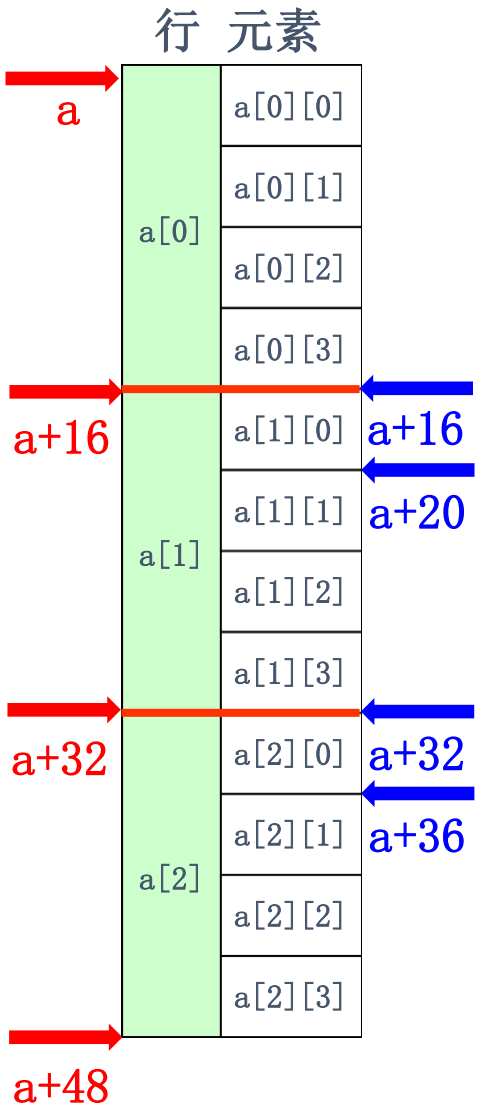
a[i]+j 实际 a+(i*n+j)*sizeof(基类型)



2.2 二维数组的地址

例：观察程序运行结果，体会行地址和元素地址的不同

```
#include <iostream>
using namespace std;
int main()
{       int a[3][4];
        cout << a << endl;                  //地址a
行地址  cout << (a+1) << endl;              //地址a+16
        cout << (a+1)+1 << endl;            //地址a+32
        cout << *(a+1) << endl;             //地址a+16
元素    cout << *(a+1)+1 << endl;           //地址a+20
地址    cout << a[2] << endl;               //地址a+32
        cout << a[2]+1 << endl;             //地址a+36
行      cout << &a[2] << endl;              //地址a+32
地址    cout << &a[2]+1 << endl;            //地址a+48(已超范围)
        return 0;
}
```



目录

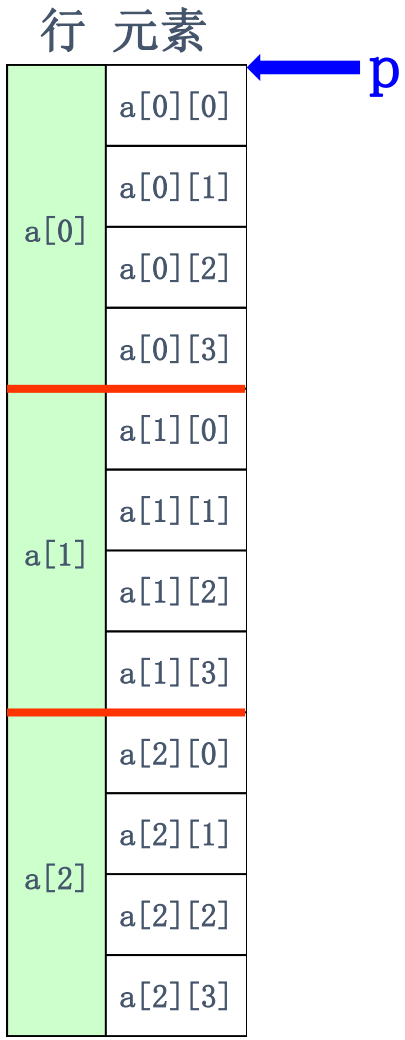
3.1 二维数组与指针

➢ 指向二维数组元素的指针变量：

```
#include <iostream>
using namespace std;
int main()
{   int a[3][4], *p;
    p=a[0];
    p=&a[0][0];
    p=*a;
    p=a;
    p=&a[0];
}
```

编译正确，p指向a[0][0]

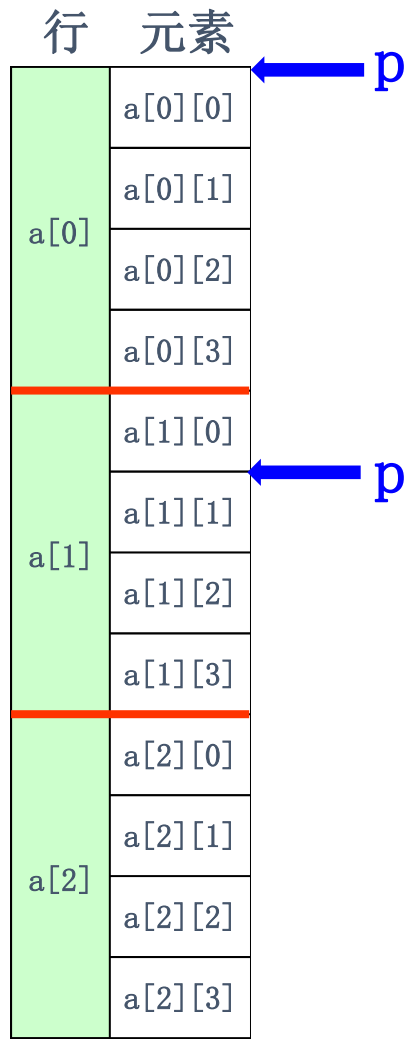
编译错误，因为a/&a[0]代表的是行地址



3.1 二维数组与指针

- 指向二维数组元素的指针变量：

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

int *p = a[0];

cout <<    p   << endl;      //元素a[0][0]地址

cout <<   p+5  << endl;      //元素a[1][1]地址

cout << *(p+5) << endl;      //a[1][1]的值
```



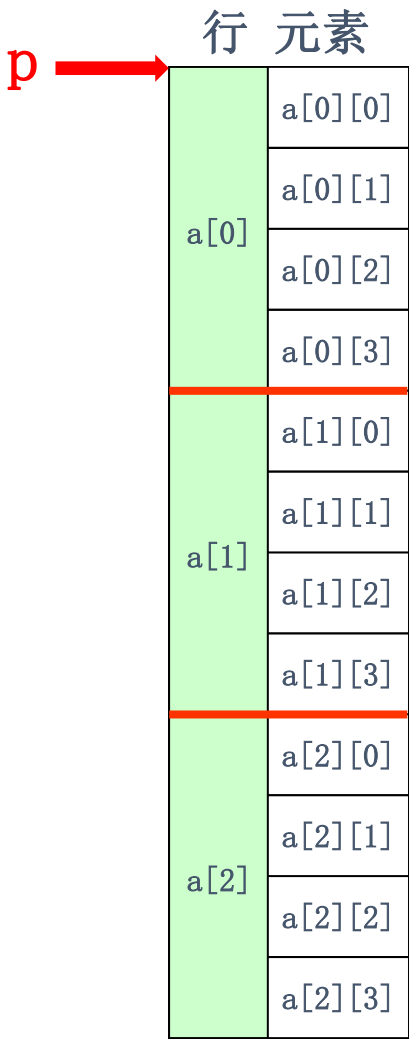
3.1 二维数组与指针

➢ 指向一维数组（行）的指针变量：

```
#include <iostream>
using namespace std;
int main()
{   int a[3][4], (*p)[4];
    p=a[0];
    p=&a[0][0];
    p=*a;
    p=a;
    p=&a[0];
}
```

编译错误

编译正确



3.1 二维数组与指针

➢ 通过指针取任意元素a[i][j]的值：

```
int a[3][4]={1,…,12}, (*p)[4] ;
p = a;
```

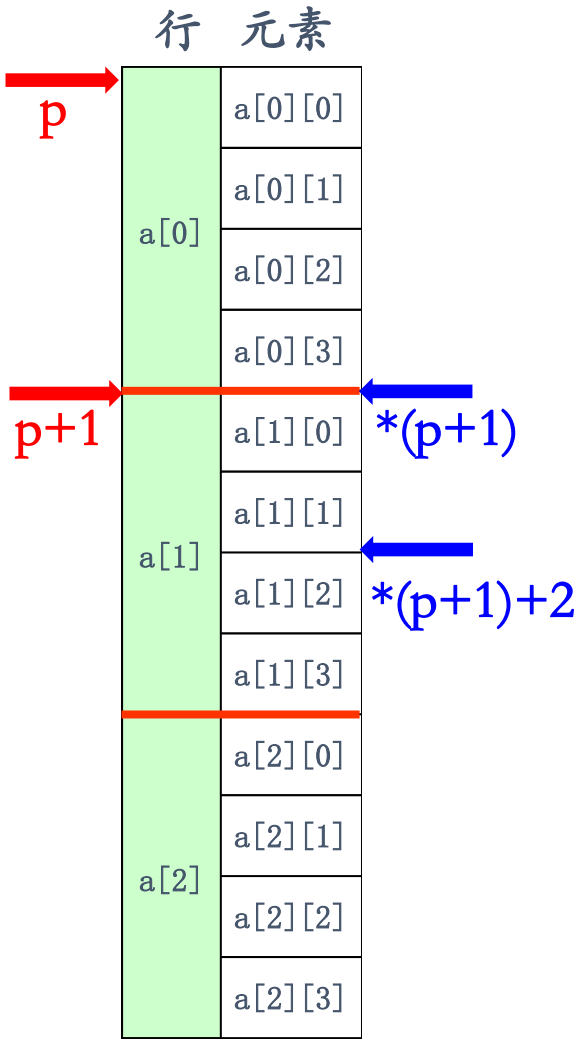
p+i，指向第i行的“int[4]”型元素，即&a[i]

*(p+i)，即a[i]

*(p+i)+j，指向第i行第j列的int型元素

((p+i)+j)，取出第i行第j列的内容，即a[i][j]

((p+i)+j)



3.1 二维数组与指针

- 通过指针取任意元素a[i][j]的值：

```
int a[3][4]={1,…,12}, (*p)[4] ;
p = a;
```

p+i，指向第i行的“int[4]”型元素，即&a[i]

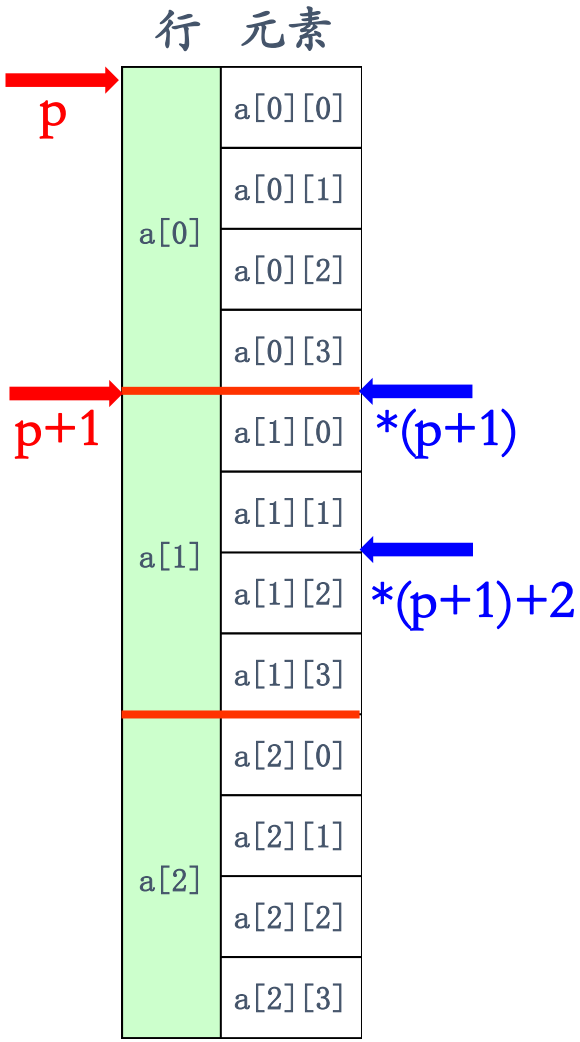
*(p+i)，即a[i]

*(p+i)+j，指向第i行第j列的int型元素

((p+i)+j)，取出第i行第j列的元素，即a[i][j]

逐行查找-->逐元素查找

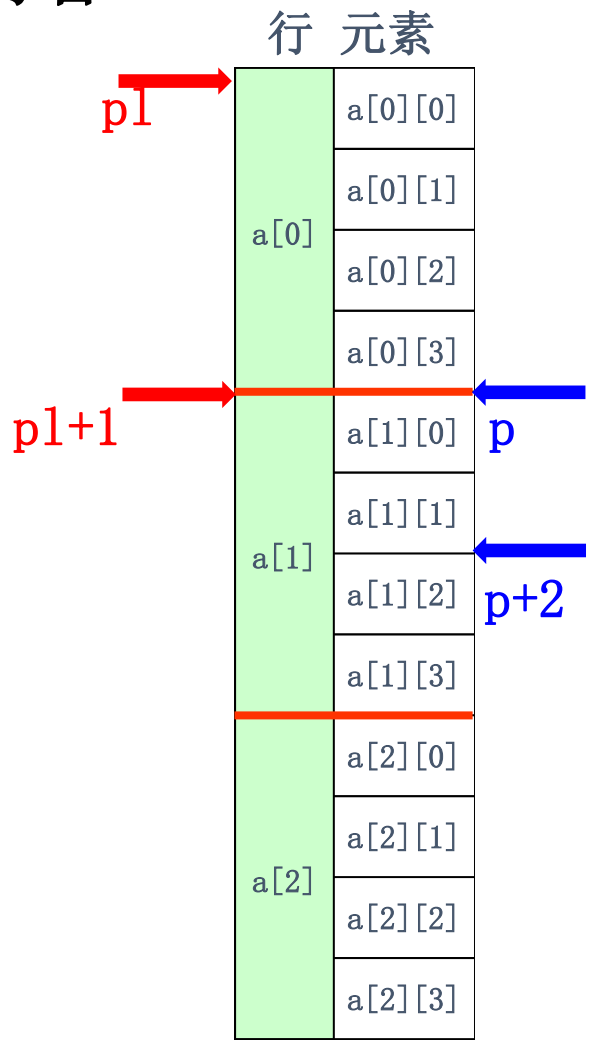
((p+i)+j)



3.1 二维数组与指针

例：使用行指针和元素指针输出二维数组的内容

```
#include <iostream>
using namespace std;
int main()
{
   int a[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
   int (*pl)[4], *p;
   for (pl=a; pl<a+3; pl++) //行指针
   {
       for (p=*pl; p<*pl+4; p++)    //元素指针
           cout << *p << ' ';
       cout << endl;                //每行一个回车
   }
}
```



指针指向二维数组（行）：

int (*p)[3]=a (或&a[0]);

```
#include<iostream>
using namespace std;
int main()
{
    int a[2][3] = { {1,2,3},{4,5,6} };
    int i, j;
    int(*p)[3];  //一定要加上括号，因为[]的优先级高于*
    p = a;  //相当于p=&a[0]，也就是指向第一个数组a[0]的首地址
    cout << p << endl;
    cout << p + 1 << endl;   //数组a[1]的首地址
    for (i = 0; i < 2; i++)
        for (j = 0; j < 3; j++)
            cout << p[i][j] << "##" << &p[i][j] << endl;
            //输出数组元素以及数组元素所属地址 ， &p[i][j]可以替换为&a[i][j]或(*(p+i)+j)
    return 0;
}
```

```
00000004563CF798
00000004563CF7A4
1##00000004563CF798
2##00000004563CF79C
3##00000004563CF7A0
4##00000004563CF7A4
5##00000004563CF7A8
6##00000004563CF7AC
```

指针指向二维数组（元素）：

int*p=a[0]或&a[0][0]或*a;

```
#include<iostream>
using namespace std;
int main()
{
    int a[2][3] = { {1,2,3},{4,5,6} };
    int i, j;
    int* p = &a[0][0]; //数组元素首地址p相当于&a[0][0]
    cout << p << endl;
    cout << p + 1 << endl;   //p+1相当于&a[0][1]
    for (i = 0; i < 2; i++)
        for (j = 0; j < 3; j++)
            cout << *(p++) << "##" << p << endl;
    return 0;
}
```

```
000000D1BAEFFBD8
000000D1BAEFFBDC
1##000000D1BAEFFBDC
2##000000D1BAEFFBE0
3##000000D1BAEFFBE4
4##000000D1BAEFFBE8
5##000000D1BAEFFBEC
6##000000D1BAEFFBF0
```

目录

- 字符串与指针

4.1 字符串与指针

✓在cout和多数C++表达式中，char数组名、char指针以及用引号括起的字符串常量都被解释为字符串第一个字符的地址

```
char flower[10] = "rose";
cout << flower << "s are red\n";
```

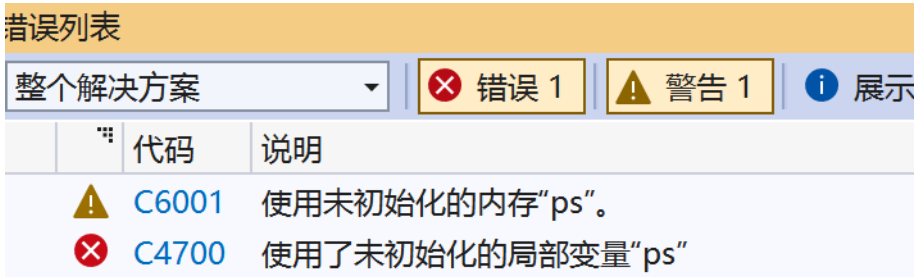
❖如果给cout提供一个字符的地址，则它将从该字符开始打印，直到遇到空字符为止

例1：演示如何使用不同形式的字符串

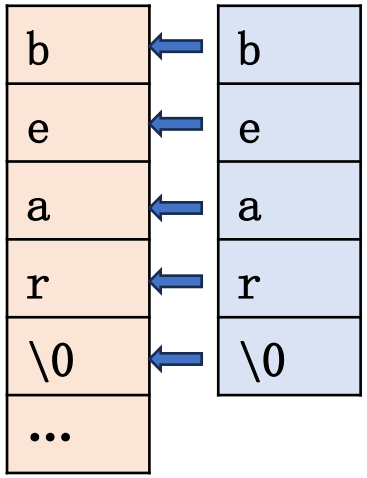
part1

```
// ptrstr.cpp -- using pointers to strings
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>                  // declare strlen(), strcpy()
int main()
{
    using namespace std;
    char animal[20] = "bear";   // animal holds bear
    const char* bird = "wren";  // bird holds address of string
    char* ps;                   // uninitialized

    cout << animal << " and ";  // display bear
    cout << bird << "\n";       // display wren
读  // cout << ps << "\n";       //may display garbage, may cause a crash

    cout << "Enter a kind of animal: ";
    cin >> animal;              // ok if input < 20 chars
写  // cin >> ps; Too horrible a blunder to try; ps doesn't
    //            point to allocated space
```

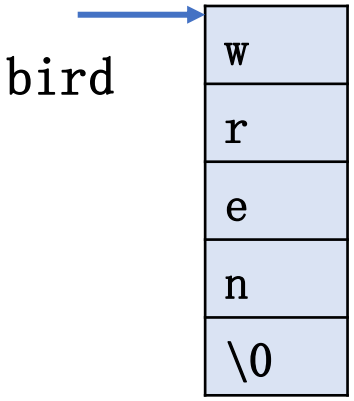
```
bear and wren
Enter a kind of animal: fox
```



变量
animal数组
后续可以修改



常量空间



常量空间

```
char animal[20] = "bear";   // animal holds bear
const char* bird = "wren";  // bird holds address of string
char* ps;                   // uninitialized

cout << animal << " and ";  // display bear
cout << bird << "\n";       // display wren
// cout << ps << "\n";      //may display garbage, may cause a crash

cout << "Enter a kind of animal: ";
cin >> animal;             // ok if input < 20 chars
// cin >> ps; Too horrible a blunder to try; ps doesn't point to allocated space
```

例1：演示如何使用不同形式的字符串

part2

```
ps = animal;                    // set ps to point to string
cout << ps << "!\n";            // ok, same as using animal
cout << "Before using strcpy():\n";
cout << animal << " at " << (int*)animal << endl;
cout << ps << " at " << (int*)ps << endl;

ps = new char[strlen(animal) + 1];  // get new storage
strcpy(ps, animal);             // copy string to new storage
cout << "After using strcpy():\n";
cout << animal << " at " << (int*)animal << endl;
cout << ps << " at " << (int*)ps << endl;
delete[] ps;

return 0;
}
```

```
bear and wren
Enter a kind of animal: fox
fox!
Before using strcpy():
fox at 000000EFAEBBF818
fox at 000000EFAEBBF818
After using strcpy():
fox at 000000EFAEBBF818
fox at 0000025097537OB0
```

- 将字符串animal的首地址赋给指针ps
- (int*)ps显示该字符串的地址

例1：演示如何使用不同形式的字符串

part2

```
    ps = animal;                    // set ps to point to string
    cout << ps << "!\n";            // ok, same as using animal
    cout << "Before using strcpy():\n";
    cout << animal << " at " << (int*)animal << endl;
    cout << ps << " at " << (int*)ps << endl;
```

```
    ps = new char[strlen(animal) + 1];  // get new storage
    strcpy(ps, animal);                 // copy string to new storage
    cout << "After using strcpy():\n";
    cout << animal << " at " << (int*)animal << endl;
    cout << ps << " at " << (int*)ps << endl;
    delete[] ps;
```

```
    return 0;
}
```

```
bear and wren
Enter a kind of animal: fox
fox!
Before using strcpy():
fox at 000000EFAEBBF818
fox at 000000EFAEBBF818
After using strcpy():
fox at 000000EFAEBBF818
fox at 0000025097537OB0
```

- 使用new来分配内存来存储字符串，根据字符串的长度来指定所需空间（了解即可）
- 使用完毕后，delete来释放空间

例1：演示如何使用不同形式的字符串

part2

```
ps = animal;                    // set ps to point to string
cout << ps << "!\n";            // ok, same as using animal
cout << "Before using strcpy():\n";
cout << animal << " at " << (int*)animal << endl;
cout << ps << " at " << (int*)ps << endl;
```

```
ps = new char[strlen(animal) + 1];  // get new storage
strcpy(ps, animal);                 // copy string to new storage
cout << "After using strcpy():\n";
cout << animal << " at " << (int*)animal << endl;
cout << ps << " at " << (int*)ps << endl;
delete[] ps;
```

```
return 0;
}
```

```
bear and wren
Enter a kind of animal: fox
fox!
Before using strcpy():
fox at 000000EFAEBBF818
fox at 000000EFAEBBF818
After using strcpy():
fox at 000000EFAEBBF818
fox at 0000025097537OB0
```

- 需要使用strcpy将animal数组中的字符串复制到新分配空间，并需要特别注意长度不要越界：

strcpy(ps, animal);//需确定有足够的空间来存储副本

❖ 关于strcpy的引申思考：

需确定有足够的空间来存储副本

```
char food[20] = "carrots"; //initialization
strcpy(food,"flan"); //ok
strcpy(food,"a picnic basket filled with many goodies"); //cause problem
```

为避免这种问题，可使用strncpy：

```
strncpy(food,"a picnic basket filled with many goodies", 19);
food[19] = '\0';
```

例2：字符数组、字符串指针数组与string对象数组

part1

```
// nested.cpp -- nested loops and 2-D array
#include <iostream>
const int Cities = 5;
const int Years = 4;
int main()
{   using namespace std;
    const char* cities[Cities] =   // array of pointers
    // char cities[Cities][25] =
    // const string cities[Cities]=
   {                                // to 5 strings
        "Gribble City",
        "Gribbletown",
        "New Gribble",
        "San Gribble",
        "Gribble Vista"
    };
    int maxtemps[Years][Cities] =   // 2-D array
        {96, 100, 87, 101, 105},    // values for maxtemps[0]
        {96, 98, 91, 107, 104},    // values for maxtemps[1]
        {97, 101, 93, 108, 107}, // values for maxtemps[2]
        {98, 103, 95, 109, 108}    // values for maxtemps[3]
    };
```

例2：字符数组、字符串指针数组与string对象数组

part2

```
    cout << "Maximum temperatures for 2008 - 2011\n\n";
    for (int city = 0; city < Cities; ++city)
    {
        cout << cities[city] << ":\t";
        for (int year = 0; year < Years; ++year)
            cout << maxtemps[year][city] << "\t";
        cout << endl;
    }

    return 0;
}
```

```
Maximum temperatures for 2008 - 2011

Gribble City:   96      96      97      98
Gribbletown:    100     98      101     103
New Gribble:    87      91      93      95
San Gribble:    101     107     108     109
Gribble Vista:  105     104     107     108
```

```
const char* cities[Cities]    // 字符串指针数组
char cities[Cities][25]       // 二维字符数组
const string cities[Cities]   // string对象数组
```

4.1 字符串与指针

```
(1) const char* cities[Cities]    // 字符串指针数组
(2) char cities[Cities][25]       // 二维字符数组
(3) const string cities[Cities]   // string对象数组
```

- ❖从存储空间的角度看，使用指针数组更为经济
- ❖如果要修改其中任何一个字符串，则二维数组是更好的选择
- ❖如果希望字符串可修改，则省略限定符cosnt
- ❖这三种方式使用相同的初始化列表，显示字符串的for循环代码也一样
- ❖在希望字符串时可以修改的情况下，string类自动调整大小的特性，使这种方法比使用二维数组更为方便

例3：字符串指针数组

```
 // more_and.cpp -- using the logical AND operator
#include <iostream>
const char* qualify[4] =        // an array of pointers*/
{   "10,000-meter race.\n",     // to strings
    "mud tug-of-war.\n",
    "masters canoe jousting.\n",
    "pie-throwing festival.\n"
};
int main()
{   using namespace std;
    int age;
    cout << "Enter your age in years: ";
    cin >> age;
    int index;
    if (age > 17 && age < 35)
        index = 0;
    else if (age >= 35 && age < 50)
        index = 1;
    else if (age >= 50 && age < 65)
        index = 2;
    else
        index = 3;
    cout << "You qualify for the " << qualify[index];

    return 0;
}
```

- char指针数组可以表示一系列字符串，只要将每个字符串的地址赋给各个数组元素即可
- 之后程序可以将cout、strlen()或strcmp()用于qualify[i]，就像用于其他字符串指针
- 使用const限定符，可以避免无意间修改这些字符串

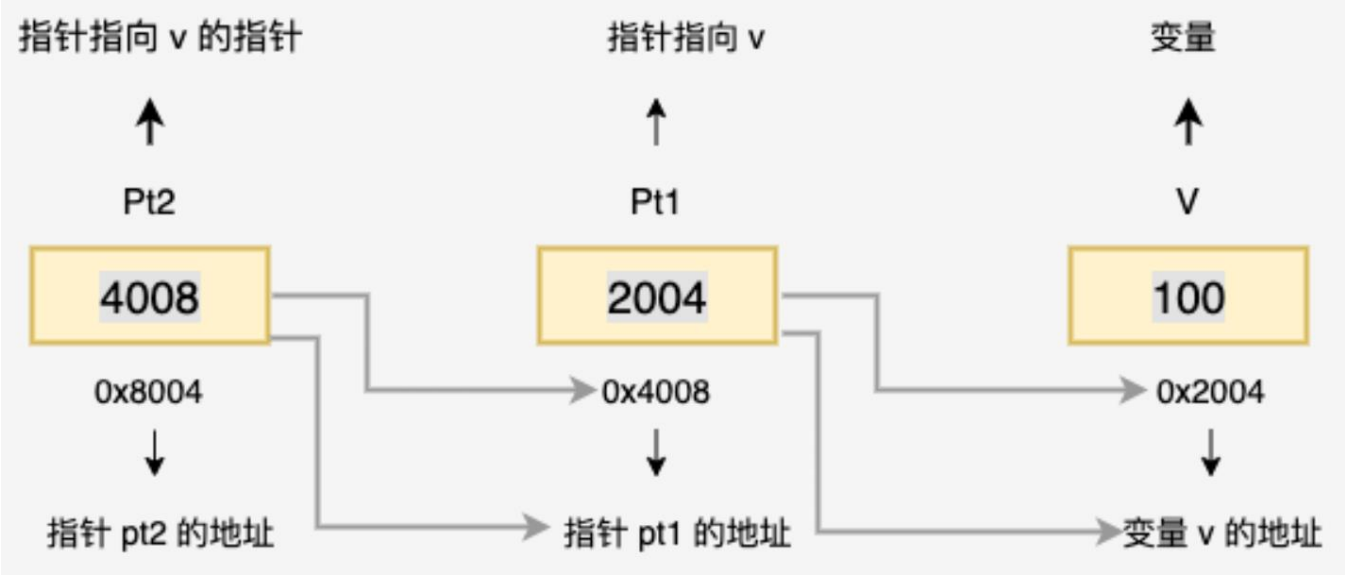
目录

- 指针进阶
  - 指向指针的指针（二级指针）
  - 二维数组与二级指针
  - 指针小结

5.1 指向指针的指针

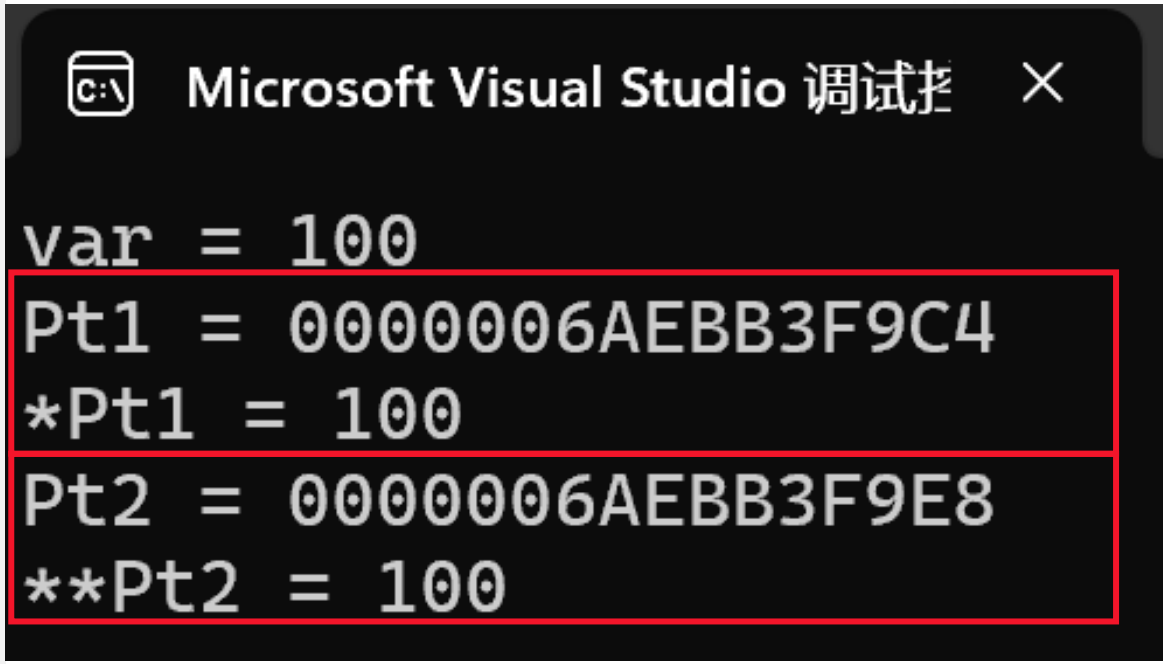
✓指向指针的指针是一种多级间接寻址的形式，或者说是一个指针链。通常，一个指针包含一个变量的地址。当定义一个指向指针的指针时，第一个指针包含了第二个指针的地址，第二个指针指向包含实际值的位置。

```
int V;
int *Pt1;
int **Pt2;
V = 100;
Pt1 =&V;
Pt2 = &Pt1;
```



例4：指向指针的指针

```c
#include <stdio.h>

int main()
{
    int  V;
    int* Pt1;
    int** Pt2;

    V = 100;
    /* 获取 V 的地址 */
    Pt1 = &V;
    /* 使用运算符 & 获取 Pt1 的地址 */
    Pt2 = &Pt1;

    /* 使用 pptr 获取值 */
    printf("var = %d\n", V);
    printf("Pt1 = %p\n", Pt1);
    printf("*Pt1 = %d\n", *Pt1);
    printf("Pt2 = %p\n", Pt2);
    printf("**Pt2 = %d\n", **Pt2);
    return 0;
}
```



5.2 二维数组与二级指针

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
int ** p;
p = (int**)a;  /* 不做强制类型转换会报错 */
```

- p是一个二级指针，它首先是一个指针，指向一个int*
- a是二维数组名，它首先是一个指针，指向一个含有4个元素的int数组
- 由此可见，a和p的类型并不相同，如果想将a赋值给p，需要强制类型转换

- 首先看一下p的值，p指向a[0][0]，即p的值为a[0][0]的地址
- 再看一下*p的值，p所指向的类型是int*，占4字节，根据前面所讲的解引用操作符的过程：从p指向的地址开始，取连续4个字节的内容。得到的正是a[0][0]的值，即1
- 再看一下**p的值，报错了？因为你访问了地址为1的空间，而这个空间你是没有权限访问的

```
#include<iostream>
using namespace std;
int main()
{
    int a[2][3] = { {1,2,3},{4,5,6} };
    int i, j;
    int** p;
    p = (int**)a;  /* 不做强制类型转换会报错
    cout << **p << endl;
    cout << p << endl;
    cout << p + 1 << endl
    for (i = 0; i < 2; i++
        for (j = 0; j < 3
            cout << *(p++)
    return 0;
}
//输出结果:
```

已引发异常

0x00007FF761962396 处(位于 Project5.exe 中)引发的异常:
0xC0000005: 读取位置 0x0000000200000001 时发生访问冲突。

显示调用堆栈 | 复制详细信息 | 启动 Live Share 会话...

异常设置
- [x] 引发此异常类型时中断
  从以下位置引发时除外:
  - [ ] Project5.exe

打开异常设置 | 编辑条件

5.2 二维数组与二级指针

实参		所匹配的形参	
数组的数组	char c[8][10];	char (*)[10];	数组指针
指针数组	char *c[10];	char **c;	指针的指针
数组指针（行指针）	char (*c)[10];	char(*c)[10];	不改变
指针的指针	char **c;	char **c;	不改变

一、 情况1：实参为二维数组

比如　　int a[3][3];
调用形式　　print(a);

```
//指针形式
void print(int** a);        //ERROR
void print(int* a[3]);      //ERROR 这是一个数组，不能将数组直接传值，因此错误
void print(int (*a)[3]);    //OK   二维数组转数组指针
//纯数组形式
void print(int a[3][3]);    //OK     等同于void print(int(*a)[3]);
void print(int a[4][3]);    //OK     等同于void print(int(*a)[3]);
void print(int a[3][4]);    //ERROR 等同于void print(int(*a)[4]);
void print(int a[][3]);     //OK     等同于void print(int(*a)[3]);
void print(int a[][]);      //ERROR  传参的错误写法;
void print(int a[3][]);     //ERROR  传参的错误写法;
```

二、 情况2：实参为指针数组

```
int* a[3]
int b[3][3] = { 0 };
for(int i = 0; i < 3; ++i){ a[i] = b[i]; }
调用形式    print(a);
```

```
//指针形式
void print(int** a);        //OK
void print(int* a[3]);      //OK
void print(int (*a)[3]);    //ERROR
//纯数组形式
void print(int a[3][3]);    //ERROR    等同于void print(int (*a)[3]);
void print(int a[4][3]);    //ERROR    等同于void print(int(*a)[3]);
void print(int a[3][4]);    //ERROR    等同于void print(int(*a)[4]);
void print(int a[][3]);     //ERROR    等同于void print(int(*a)[3]);
void print(int a[][]);      //ERROR    无法将其转化为void print(int** a);
void print(int a[3][]);     //ERROR    传参的错误写法;
```

三、 情况3：实参为数组指针

比如 int (*a)[3];
int b[3][3] = {0};
a = b;
调用形式 print(a);

```
//指针形式
void print(int** a);        //ERROR
void print(int* a[3]);      //ERROR
void print(int (*a)[3]);    //OK
//纯数组形式
void print(int a[3][3]);    //OK    等同于void print(int(*a)[3]);
void print(int a[4][3]);    //OK    等同于void print(int(*a)[3]);
void print(int a[3][4]);    //ERROR 等同于void print(int(*a)[4]);
void print(int a[][3]);     //OK    等同于void print(int(*a)[3]);
void print(int a[][]);      //ERROR 无法将其转化为void print(int**a);
void print(int a[3][]);     //ERROR 传参的错误写法;
```

四、 情况4：实参为指针的指针（二级指针）

```
int** a;
int b[3][3] = { 0 };
int* c[3];
for(int i = 0; i < 3; ++i){ c[i] = b[i]; }
a = c;
调用形式      print(a);
```

```
//指针形式
void print(int** a);        //OK
void print(int* a[3]);      //OK 等价于void print(int** a);
void print(int (*a)[3]);    //ERROR
//纯数组形式
void print(int a[3][3]);    //ERROR    等同于void print(int(*a)[3]);
void print(int a[4][3]);    //ERROR    等同于void print(int(*a)[3]);
void print(int a[3][4]);    //ERROR    等同于void print(int(*a)[4]);
void print(int a[][3]);     //ERROR    等同于void print(int(*a)[3]);
void print(int a[][]);      //ERROR    无法将其转化为void print(int**a);
void print(int a[3][]);     //ERROR    传参的错误写法;
```

5.3 指针小结

1. 声明指针

```
typeName * pointerName;
double * pn;
char *pc;
```

2. 给指针赋值，应将内存地址赋给指针，&运算符获得被命名的内存的地址，new运算符返回未命名的内存的地址

```
double * pn;
char *pc;
double du = 3.2;
pn = &du;
```

3. 对指针解除引用，意味着获得指针所指向的值

```
cout << *pn ;
```

5.3 指针小结

4. 区分指针和指针所指向的值

```
int * pt = new int;  //了解即可
*pt = 5;
```

5. C++将数组名视为数组的第0个元素的地址。将sizeof运算符用于数组名时，返回整个数组的长度（单位为字节）

```
int tacos[10];// now tacos is the same as &tacos[0]
```

5.3 指针小结

6. 指针算术

```
int tacos[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int* pt = tacos;          // pt points to tacos[0], 假设地址3000
pt = pt + 1;              // pt now points to tacos[1], 则地址为3004
int* pe = &tacos[9];  // pe points to tacos[9], 则地址为3036
pe = pe - 1;          // pe now points to tacos[8], 则地址为3032
int diff = pe - pt; // diff is 7, the separation between tacos[1] and tacos[8]
```

5.3 指针小结

7. 数组表示法和指针表示法

❖使用方括号数组表示法等同于对指针解除引用

```
tacos[0] means * tacos // means the value of the first element
tacos[3] means * (tacos + 3) // means the value of the fourth element
```

❖对于指针和数组名，既可以使用指针表示法，也可以使用数组表示法

```
int* pt = new int[10]; // pt points to an array of 10 ints, 了解即可
*pt = 5;               // set element number 0 to 5
pt[0] = 6;             // reset element number 0 to 6
pt[9] = 44;            // set element number 9 to 44
int coats[10];         // an array of 10 ints
*(coats + 4) = 12; // set coats[4] to 3
```

```
int a[10], *p=a;
```

p+1　　：取p所指元素的下一个数组元素的地址 p+sizeof(数组类型)

*(p+1)　：取p所指元素的下一个数组元素的值(p不变)

*p+1　　：取p所指元素的值，值再+1

p++　　：p指向下一个数组元素的地址(p改变)

*(p++)　：取p所指元素的值，p再指向下一个数组元素的地址(p改变)

*p++　　：同上

*++p　　：表示p指向下一个数组元素的地址，再取该元素的值

(*p)++　：取p所指数组元素的值，值再++

```
char *a[3] = {(char *)"china", (char *)"student", (char *)"s"}, **p;
```

```
p=a;
```

p+1 ： a[1]的地址(地址2004)

p++ ： p指向a[1](p的值变为地址2004)

*p ： 取a[0]的值3000(字符串"china"的首地址)

*(p+1) ： 取a[1]的值3100(字符串"student"的首地址)

*p++ ： 取a[0]的值3000，p指向a[1](地址2004)

(*p)++ ： 取a[0]的值3000,再++为3001(字符'h'的地址)

*p+3 ： 取a[0]的值3000,再+3为3003(字符'n'的地址)

*(*p+3) ： 取a[0]的值3000,再+3为3003(字符'n')

p	2100	2000

a	2000	3000
	2004	3100
	2008	3200

字符串常量"china"(无名)

3000	c
3001	h
3002	i
3003	n
3004	a
3005	\0

字符串常量"student"(无名)

3100	s
3101	t
3102	u
3103	d
3104	e
3105	n
3106	t
3107	\0

字符串常量"s"(无名)

3200	s
3201	\0

int *p:	指向整型简单变量/数组元素的指针变量
int *p[n]:	指针数组，数组元素为int *类型
int (*p)[n]:	指向含n个int元素的一维数组的指针变量
int *p():	返回值为int *类型的函数
int (*p)():	指向函数的指针(形参为空，返回int)
int **p:	指向int *类型指针的指针变量
int const *p:	指向常量的指针变量
int *const p:	常指针
const int *const p:	指向常量的常指针
void *p:	基类型为void的指针

- 思考：

1、这4种情况中的p是?(指针/数组/函数)

2、如果是指针，指向什么？

3、如果是数组，数组元素是什么类型？

4、如果是函数，函数的形参及返回类型是什么？

- 方法：一层层看

`int *(*p)():` p是指向函数的指针，被指向的函数没有形参，返回一个int *型指针

`int *(*p)[n]:` p是指针，指向一个n元素数组，每个元素都是指向int的指针

`int (*p[n])():` p是返回值为int，无参数的函数指针数组

`int *(*p[n])():` p是返回值为int *，无参数的函数指针数组

`int *(*p)();` p是指向函数的指针，被指向的函数没有形参，返回一个int *型指针

`int *(*p)[n];` p是指针，指向一个n元素数组，每个元素都是指向int的指针

`int (*p[n])();` p是返回值为int，无参数的函数指针数组

`int *(*p[n])();` p是返回值为int *，无参数的函数指针数组

```
int *fun() { ...; }
int main ()
{
   int *(*p)();
   p = fun;
   return 0;
}
```

```
int main ()
{  int *a[10],*b[3][10];
   int *(*p)[10];
   p = &a;
   p = b;
   return 0;
}
```

```
int fun() { ...; }
int main ()
{
   int (*p[10])();
   p[0] = fun;
   return 0;
}
```

```
int *fun() { ...; }
int main ()
{
   int *(*p[10])();
   p[0] = fun;
   return 0;
}
```

总结

- 一维数组与指针
- 二维数组的地址
  - 二维数组的基本概念
  - 二维数组的地址

    （行地址/元素地址）
- 二维数组与指针
- 字符串与指针
- 指针进阶
  - 指向指针的指针
  - 二维数组与二级指针
  - 指针小结