第二章 指针进阶与引用

主讲教师：同济大学电子与信息工程学院 陈宇飞

目录

- 多维数组与指针
- 函数与指针
- 指针数组
- 指向指针的指针
- 其它指针
- 数组的引用
- 总结与应用

```
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;
}
```

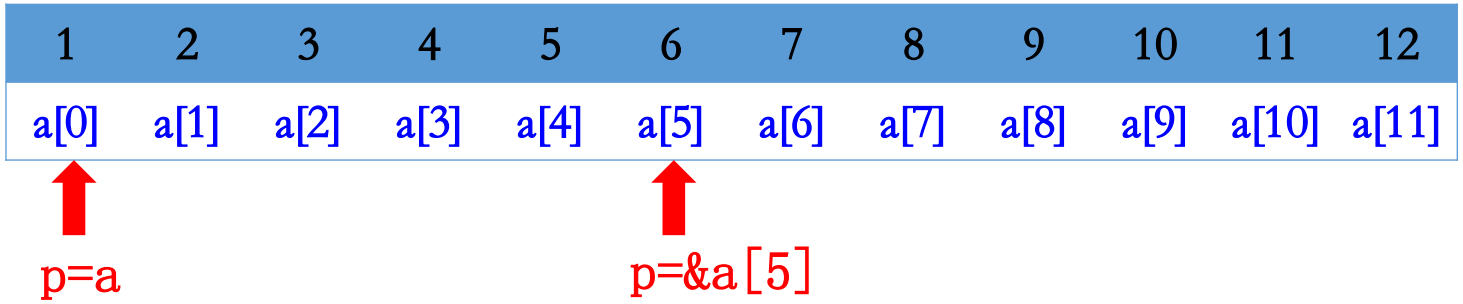
```
int fun() { ...; }
int main ()
{
   int (*p[10])();
   p[0] = fun;
   return 0;
}
```

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

2.1 多维数组与指针

- 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;
```



  - 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 多维数组与指针

- 基本概念

一维数组的理解方法(下标法、指针法)

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

a ：数组名/数组的首元素地址　　a <=> &a[0]　地址

&a[i]：第i个元素的地址（下标法）

a+i：第i个元素的地址（指针法）　　&a[i] <=> a+i　地址

a[i]：第i个元素的值（下标法）

*(a+i)：第i个元素的值（指针法）　　a[i] <=> *(a+i) 值

a	2000	1	a[0]
	2004	2	a[1]
	2008	3	a[2]
	2012	4	a[3]
	2016	5	a[4]
	2020	6	a[5]
	2024	7	a[6]
	2028	8	a[7]
	2032	9	a[8]
	2036	10	a[9]
	2040	11	a[10]
	2044	12	a[11]

第0个元素的特殊表示：

值：a[0] <=> *(a+0) <=> *a　　地址：&a[0] <=> a+0 <=> a

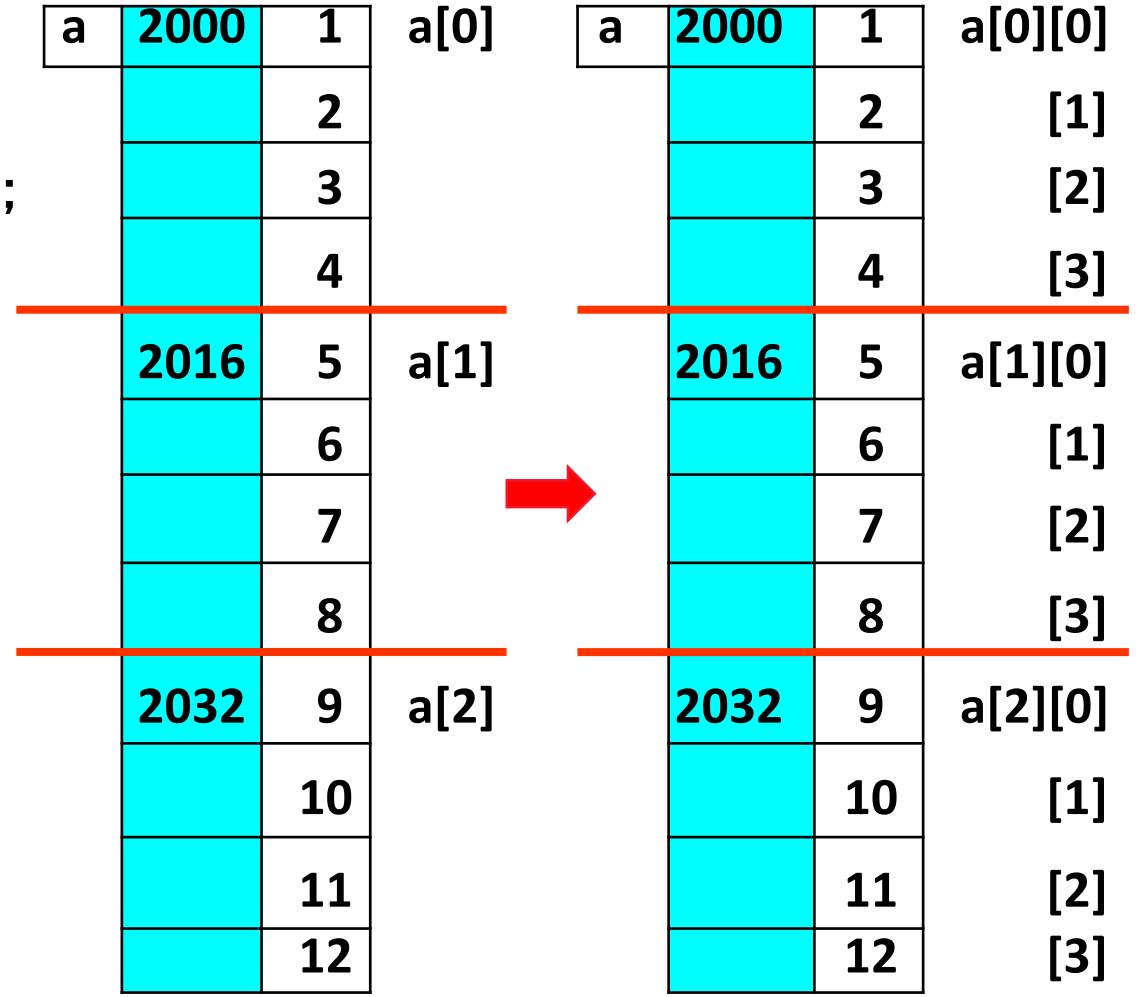
2.1 多维数组与指针

- 二维数组的地址

二维数组：

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

1 2 3 4
5 6 7 8
9 10 11 12



2.1 多维数组与指针

- 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，占用一块连续的存储空间

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 多维数组与指针

- 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，占用一块连续的存储空间
- 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 多维数组与指针

- 基本概念

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

2.1 多维数组与指针

- 基本概念
  - 行地址：

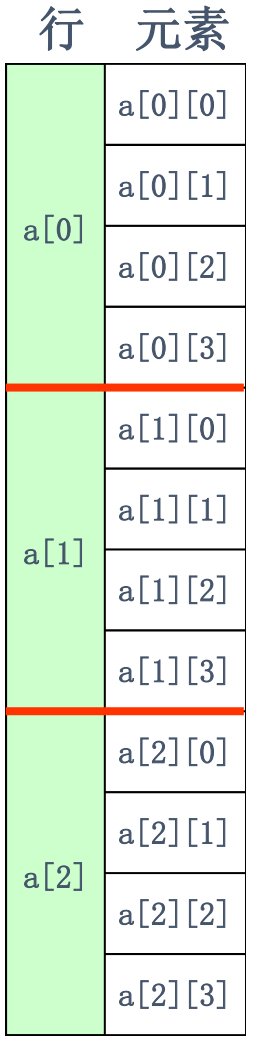
a：① 二维数组的数组名，即a

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



2.1 多维数组与指针

- 基本概念
  - 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

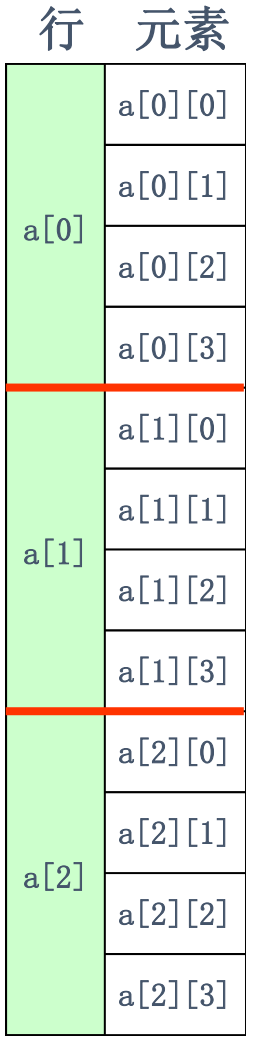
    ③ 4元素一维数组的首元素的地址

    *(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



2.1 多维数组与指针

- 基本概念
  - 数组元素值和地址的表示形式：

    三种等价：

    a[i][j] <=> *(a[i]+j) <=> *(*(a+i)+j)　　值

    &a[i][j] <=> a[i]+j <=> *(a+i)+j　　元素地址

2.1 多维数组与指针

- 基本概念

*行地址=> 首元素地址

&首元素地址 => 行地址

a : 地址(二维数组/第0行)

&a[i] : 地址(第i行)

行地址

a+i : 地址(第i行)

a[i] : 地址(第i行0列)

*(a+i) : 地址(第i行0列)

&a[i][j] : 地址(第i行j列)

元素地址

a[i]+j : 地址(第i行j列)

*(a+i)+j : 地址(第i行j列)

a[i][j] : 值(第i行j列)

*(a[i]+j) : 值(第i行j列)

((a+i)+j): 值(第i行j列)

值

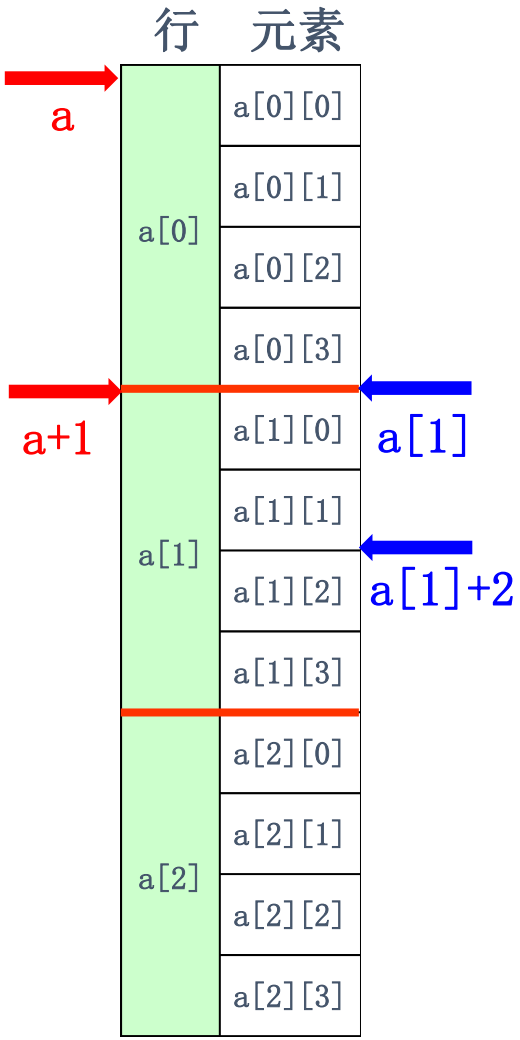
2.1 多维数组与指针

- 基本概念
  - 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)



2.1 多维数组与指针

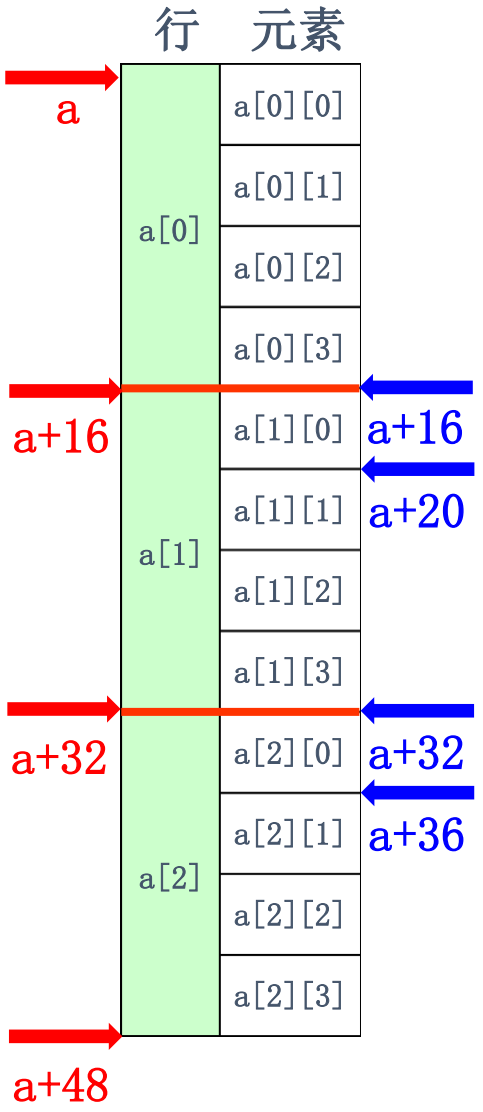
例：观察程序运行结果，体会行地址和元素地址的不同

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

行地址

元素地址

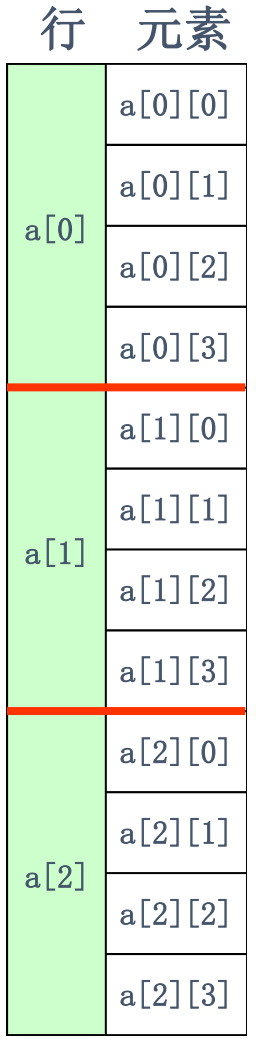
行地址



2.1 多维数组与指针

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];
  行 cout << sizeof(a)   << endl;          //48 数组大小
  地 cout << sizeof(a+1)   << endl;        //4  即&a[1],是地址(指针)
  址 cout << sizeof(*(a+1))   << endl;     //16 指针基类型是int[4]
  元 cout << sizeof(*(a+1))   << endl;     //16 a[1]是数组(4元素)
  素 cout << sizeof(**(a+1))   << endl;    //4  数组元素是int
  地 cout << sizeof(a[2])   << endl;       //16 a[2]是数组(4元素)
  址 cout << sizeof(*(a[2]))   << endl;    //4  数组元素是int
  行 cout << sizeof(&a[2])   << endl;      //4  数组a[2]的地址(指针)
  地 cout << sizeof(*(&a[2])) << endl;     //16 指针基类型是int[4]
  址
}
```



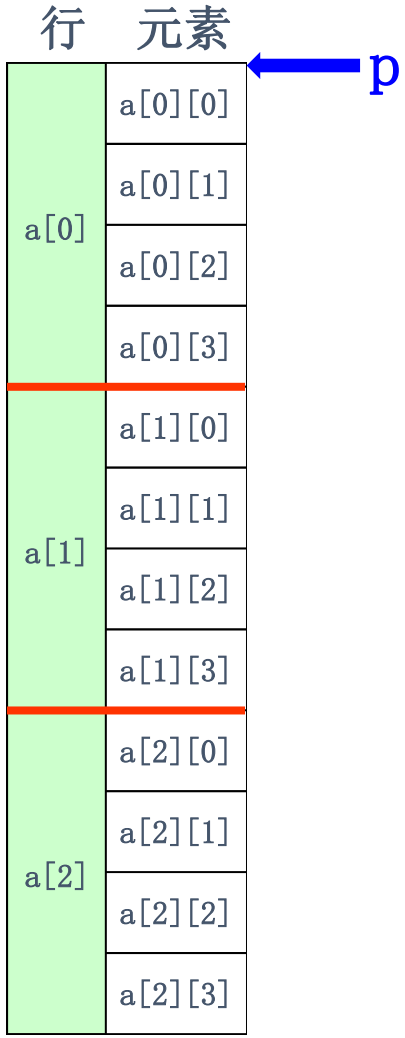
2.1 多维数组与指针

- 二维数组的指针
  - 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]

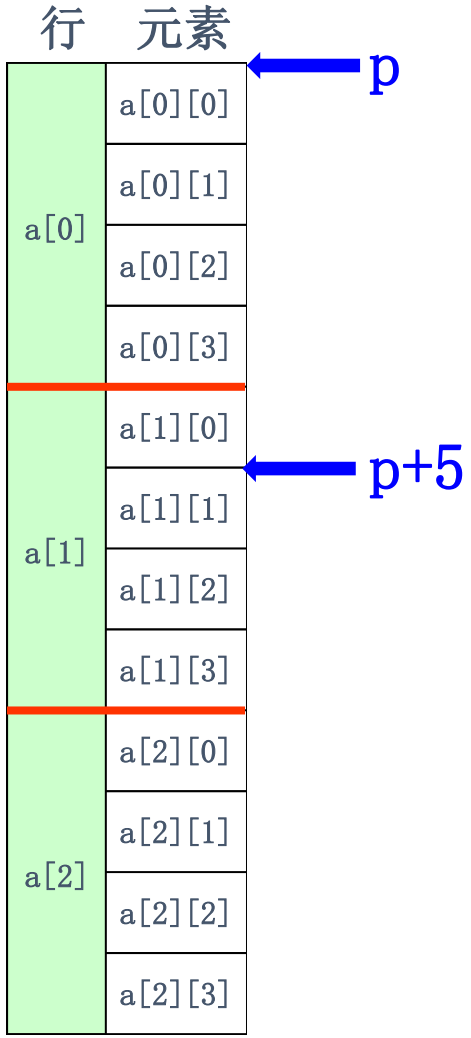
编译错误，因为a/&a[0]代表的是行地址



2.1 多维数组与指针

- 二维数组的指针
  - 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
cout << sizeof(a)  << endl;    //48    数组大小
cout << sizeof(p) << endl;     //4     因为指针
cout << sizeof(*p) << endl;    //4     因为int
```



2.1 多维数组与指针

例：打印二维数组的值

```
int main()
{  int a[3][4]={...}, *p;
   for(p=a[0];p<a[0]+12;p++)
       cout << *p << ' ';
   cout << endl;
   return 0;
}
```

```
int main()
{  int a[3][4]={...};
   int i, j, *p = a[0];
   for(i=0; i<3; i++)
       for(j=0; j<4;j++)
           cout << *p++ << ' ';
       return 0;
}
```

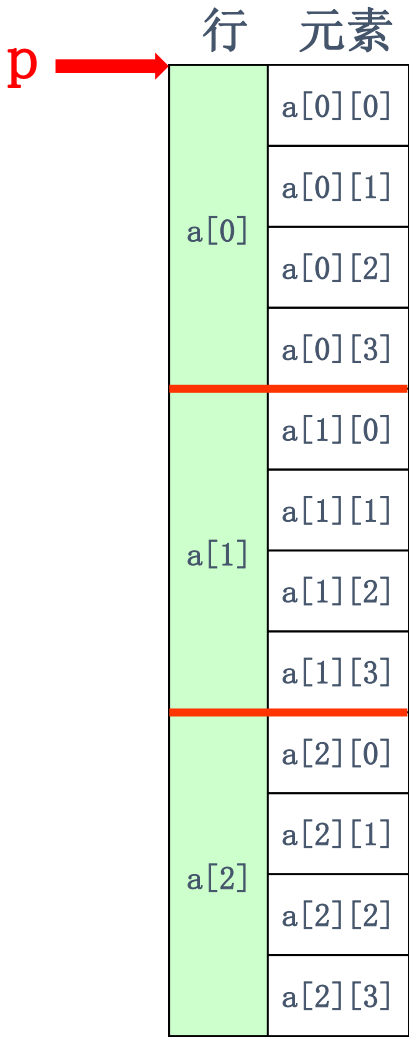
```
int main()
{  int a[3][4]={...};
   int i, j, *p=&a[0][0];
   for(i=0; i<12; i++)
       cout << *p++ << ' ';
   return 0;
}
```

```
int main()
{  int a[3][4]={...}
   int i, j, *p=&a[0][0];
   for(; p-a[0]<12;)
       cout << *p++ << ' ';
   return 0;
}
```

2.1 多维数组与指针

- 二维数组的指针
  - 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];          } 编译错误
    p=*a;
    p=a;                 } 编译正确
    p=&a[0];
}
```



2.1 多维数组与指针

- 二维数组的指针
  - 通过指针取任意元素a[i][j]的值：

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

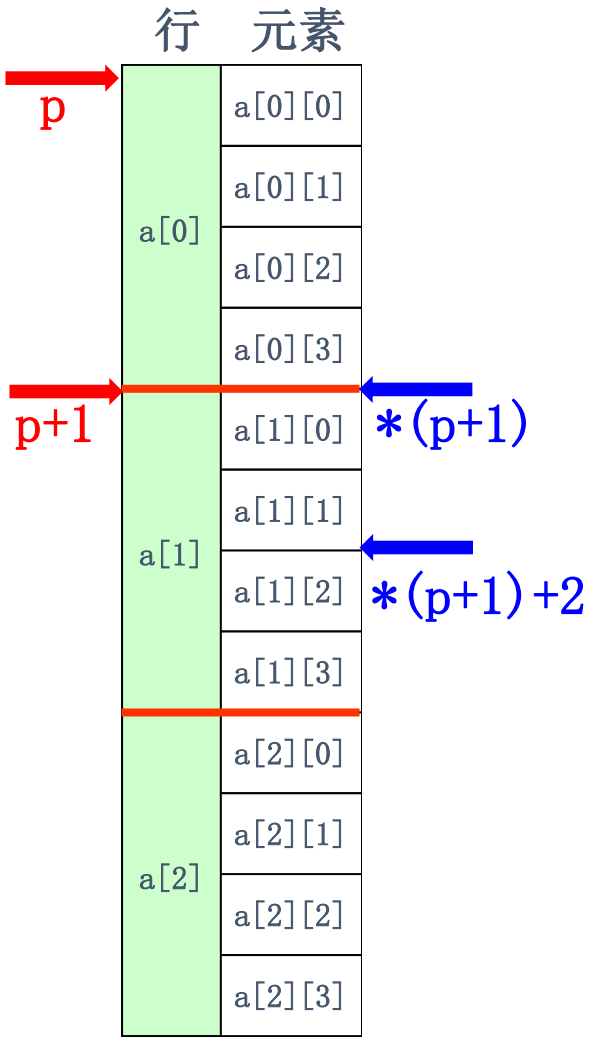
p+i，指向第i行的“int[4]”型元素，即&a[i]

*(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)



2.1 多维数组与指针

- 二维数组的指针
  - 指向一维数组（行）的指针变量：

```
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]

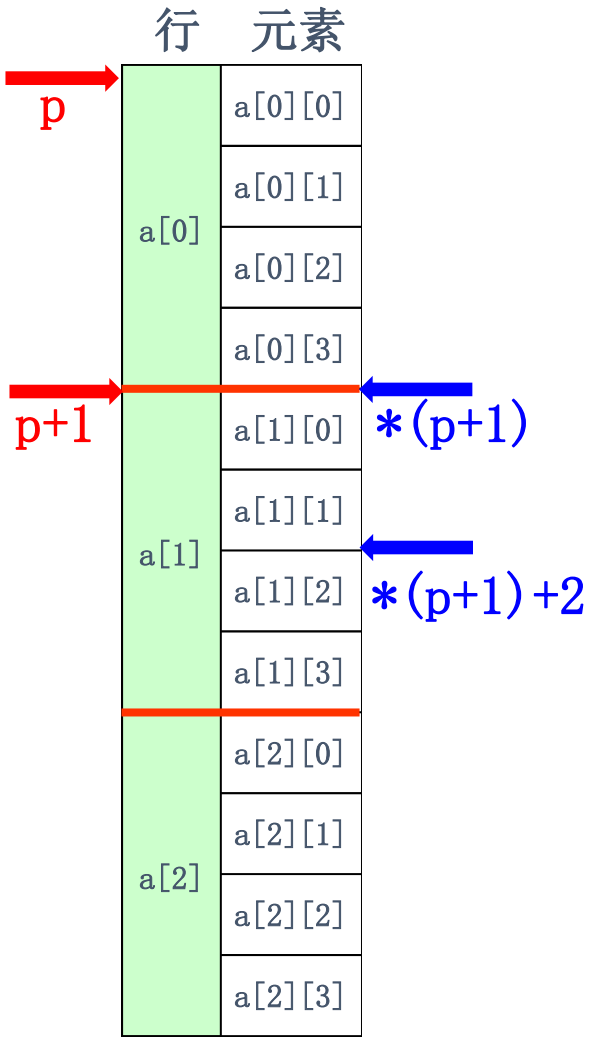
*(p+i)，即a[i]

逐行查找-->逐元素查找

*(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]

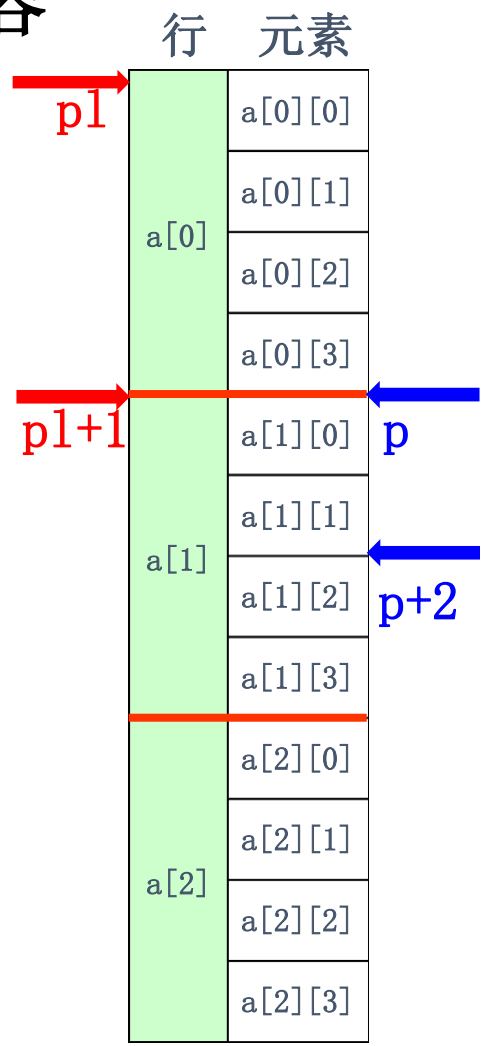
((p+i)+j)



2.1 多维数组与指针

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
   }
}
```



2.1 多维数组与指针

- 用指向二维数组元素的指针做函数参数
  - 形参是对应类型的简单指针变量

```
#include <iostream>
using namespace std;

void fun(int *data)
{
    if (*data%2==0)
        cout << *data << endl;
}
```

```
int main()
{
    int a[3][4]={...}, *p;
    for(p=a[0]; p<a[0]+12; p++)
        fun(p);
    cout << endl;
}
```

实参是指向二维数组元素的指针变量
形参是对应类型的简单指针变量

2.1 多维数组与指针

- 用指向二维数组的指针做函数参数
  - 形参的本质是指针变量

```
#include <iostream>
using namespace std;
void f1(int x1[][4]) //形参数组不指定行大小
{   cout << "x1_size=" << sizeof(x1) << endl;
}
void f2(int x2[3][4]) //形参数组行大小与实参相同
{   cout << "x2_size=" << sizeof(x2) << endl;
}
void f3(int x3[123][4]) //形参数组行大小与实参不同
{   cout << "x3_size=" << sizeof(x3) << endl;
}
```

```
int main()
{  int a[3][4];
   cout << "a_size=" << sizeof(a);
   cout << endl;
   f1(a);
   f2(a);
   f3(a);
}
```

```
a_size=48
x1_size=4  因为int*
x2_size=4  因为int*
x3_size=4  因为int*
```

2.1 多维数组与指针

- 用指向二维数组的指针做函数参数
  - 形参的本质是指针变量

```
#include <iostream>
using namespace std;
void f1(int x1[][4]) //形参数组不指定行大小
{   cout << "x1_size=" << sizeof(*x1) << endl;
}
void f2(int x2[3][4]) //形参数组行大小与实参相同
{   cout << "x2_size=" << sizeof(*x2) << endl;
}
void f3(int x3[123][4]) //形参数组行大小与实参不同
{   cout << "x3_size=" << sizeof(*x3) << endl;
}
```

```
int main()
{  int a[3][4];
   cout << "a_size=" << sizeof(a);
   cout << endl;
   f1(a);
   f2(a);
   f3(a);
}
```

```
a_size=48
x1_size=16  因为int[4]
x2_size=16  因为int[4]
x3_size=16  因为int[4]
```

2.1 多维数组与指针

- 用指向二维数组的指针做函数参数
  - 形参的本质是指针变量

```
#include <iostream>
using namespace std;
void f1(int x1[][4]) //形参数组不指定行大小
{   cout << "x1_size=" << sizeof(**x1) << endl;
}
void f2(int x2[3][4]) //形参数组行大小与实参相同
{   cout << "x2_size=" << sizeof(**x2) << endl;
}
void f3(int x3[123][4]) //形参数组行大小与实参不同
{   cout << "x3_size=" << sizeof(**x3) << endl;
}
```

```
int main()
{  int a[3][4];
   cout << "a_size=" << sizeof(a);
   cout << endl;
   f1(a);
   f2(a);
   f3(a);
}
```

```
a_size=48
x1_size=4  因为int
x2_size=4  因为int
x3_size=4  因为int
```

2.1 多维数组与指针

- 用指向二维数组的指针做函数参数
  - 形参是指向m个元素组成的一维数组的指针变量

```
void output(int (*p)[4])
{
    int i,j;
    for(i=0; i<3; i++)
        for(j=0; j<4; j++)
            cout<< *(*(p+i)+j)    << " ";
    cout << endl;  //*(p[i]+j)
}                  //p[i][j]
                   //二维数组值的三种形式
```

```
int main()
{
    int a[3][4]={...};
    output(a);
    return 0;
}
```

实参是二维数组名
形参是指向m个元素
的一维数组的指针变量

2.1 多维数组与指针

- 用指向二维数组的指针做函数参数
  - 形参是相应类型的二维数组(行的大小可省略，本质上仍然是指向m个元素组成的一维数组的指针变量)

```
void output(int (*p)[4])
{
    int i,j;
    for(i=0; i<3; i++)
        for(j=0; j<4; j++)
            cout<< *(*(p+i)+j) << " ";
    cout << endl;   //*(p[i]+j)
}                   //p[i][j]
                    //二维数组值的三种形式
```

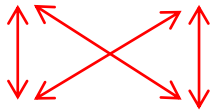
```
void output(int p[][4]) //int p[3][4]
{                       //int p[123][4]
    int i,j;            //本质都是行指针变量
    for(i=0; i<3; i++)
        for(j=0; j<4; j++)
            cout<< *(*(p+i)+j) << " ";
    cout << endl;
}
```

2.1 多维数组与指针

- 二维数组做函数参数的实参/形参的四种组合

```
//形参是二维数组名
void fun(int p[][4])
{
    ...
}
```

```
//形参是指向m个元素组成的一维数组的指针变量
void fun(int (*p)[4])
{
  ...
}
```



```
//实参是二维数组名
int main()
{   int a[3][4]={...};
    fun(a);
}
```

```
//实参是指向m个元素组成的一维数组的指针变量
int main()
{  int a[3][4]={...};
   int (*p)[4];
   p=a;
   fun(p);
 }
```

2.1 多维数组与指针

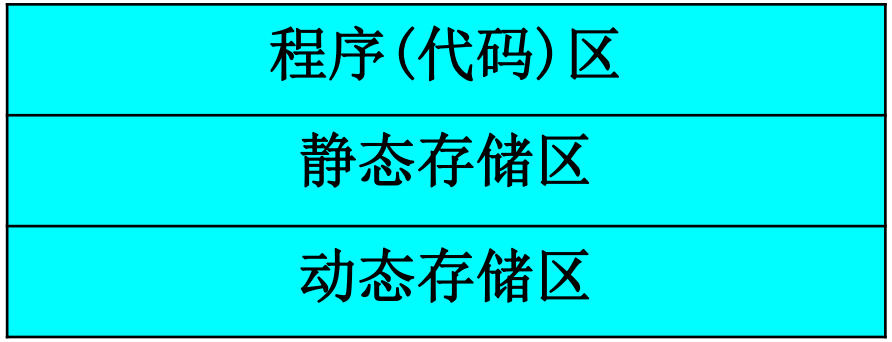
- 指向一维数组的指针变量
  - 对一维数组而言，数组的指针和数组元素的指针，其实都是指向数组元素的指针变量(特指0/任意i)，因此本质相同(基类型相同)
  - 数组名代表数组首地址，指针是地址，但本质不同(sizeof(数组名)/sizeof(指针)大小不同)

```
int a[10] = {0,1,2,3,4,5,6,7,8,9};
cout << a << endl;                    数组a的地址                          地址a
cout << &a[0] << endl;                a[0]元素的地址                       地址a
cout << sizeof(*a) << endl;           地址a的基类型        a[0]的类型      4
cout << sizeof(*(&a[0])) << endl;     地址&a[0]的基类型    a[0]的类型      4
cout << sizeof(a) << endl;            数组a的大小                          40
cout << sizeof(&a[0]) << endl;        地址&a[0]的大小                      4
```

目录

2.2.1 指向函数的指针变量

- 函数的地址



程序(代码)区：存放程序的执行代码

由若干函数的代码组成，每个函数占据一段连续内存空间

每个函数的内存空间的起始地址，称为函数的地址(指针)

函数名代表函数的首地址

2.2.1 指向函数的指针变量

- 用函数指针变量调用函数
  - 指向函数的指针变量的定义：

    数据类型 (*指针变量名)(形参表)

    int (*p)(int,int); （一层层看）

    是指针变量

    指针变量指向函数,形参为两个int数据类型，int是函数的返回类型
  - 使用：

    赋初值：指针变量名 = 函数名 不要参数表

    调用：指针变量名(函数实参表列)

- 用函数指针变量调用函数

```
//谭书P.171 例6.11
#include <iostream>
using namespace std;
int max(int x, int y)
{
   return (x>y?x:y);
}
int main()
{
   int a,b,m;
   cin >> a >> b;
   m=max(a,b);
   cout << "max=" << m << endl;
   return 0;
}
```

```
//谭书P.172 例6.11
#include <iostream>
using namespace std;
int max(int x, int y)
{
   return (x>y?x:y);
}
int main()
{
    int a,b,m;
    int (*p)(int,int);//指向函数的指针变量
    p = max;          //赋初值，不带参数
    cin >> a >> b;
    m = p(a,b);       //函数调用，带实参表
    cout << "max=" << m << endl;
    return 0;
}
```

p和*p都是函数的首地址
m=p(a,b);
m=(*p)(a,b);
都正确，但一般不用后者

- 用函数指针变量调用函数

```
#include <iostream>
using namespace std;
int fun()
{
   return 37;
}
int main()
{
   int (*p)();
   p = fun;
   cout << fun() << endl;
   cout << fun << endl;
   cout << *fun << endl;
   cout << p() << endl;
   cout << p << endl;
   cout << *p << endl;
}
```

```
37
00F714D3
00F714D3
37
00F714D3
00F714D3
```

函数名带()，表示调用函数

函数名不带()，表示地址

函数指针

加*不加* 的结果相同

2.2.1 指向函数的指针变量

- 用函数指针变量调用函数
  - 指向函数的指针的形参表声明时，与被调用函数的形参表类型、顺序、数量一致,是否带形参变量名，形参变量名称是否一致不作要求

```
int max(int x, int y) { ... }
int main()
{
    int (*p)(int, int);     //不带形参变量名
    int (*p)(int x,int y); //形参变量名相同
    int (*p)(int p,int q); //形参变量名不同
    p=max;
}
```

2.2.1 指向函数的指针变量

- 用函数指针变量调用函数
  - 指向函数的指针变量进行指针运算是无意义的

    p+n ： 编译出错

    p++ ： 编译出错

    p<q ： 编译出错/不出错但无意义

    *p ： 编译不错但无意义

```
#include <iostream>
using namespace std;
int max(int x, int y)
{   return (x>y?x:y);
}
int main()
{
    int (*p)(int, int);
    p=max;
    p++;          //编译报错
    p=p-2;        //编译报错
    return 0;
}
```

- 指向函数的指针变量进行指针运算是无意义的

```
#include <iostream>
using namespace std;
int max(int x, int y)
{   return (x>y?x:y);
}
int min(int x, int y)
{   return (x<y?x:y);
}
int main()
{   int (*p)(int, int);
    int (*q)(int, int);
    p=max;
    q=min;
    cout << (p<q) << endl;
     //输出0/1,无意义
    return 0;
}
```

```
#include <iostream>
using namespace std;
int max(int x, int y)
{    return (x>y?x:y);
}
int fun(int x)
{    return x;
}
int main()
{    int (*p)(int, int);
     int (*q)(int);
     p=max;
     q=fun;
     cout << (p<q) << endl; //编译出错
     return 0;
}
```

2.2.1 指向函数的指针变量

- 指向函数的指针做函数参数

例：编写一个函数fun，每次调用它的时候实现不同的功能。第一次调用时求出a与b的和，第二次求出a与b的差，第三次求出a与b的积。

```
void add(int x, int y)
{  cout << x+y << endl;
}
void sub(int x, int y)
{  cout << x-y << endl;
}
void multi(int x, int y)
{  cout << x*y << endl;
}
```

```
void fun( void (*f)(int, int) )
{    int a=10,b=15;
     f(a, b);
}
int main()
{    fun(add);        //25
     fun(sub);        //-5
     fun(multi);      //150
     return 0;
}
```

2.2.1 指向函数的指针变量

- 指向函数的指针做函数参数
  - 适用于在函数中每次调用不同的函数，增加了函数使用的灵活性
  - 被调用的函数必须有相同的返回类型和形参表列
  - C++可通过重载函数、多态性与虚函数等方法解决同样的问题，因此C++中这种方法不常用（纯C使用）

//atexit示例演示：int atexit(void (*)(void)); 了解即可

2.2.1 指向函数的指针变量

- 指向类成员函数的指针变量

```
//指向全局函数的指针
#include <iostream>
using namespace std;
void fun()
{
    cout << "fun()" <<endl;
}
int main()
{
    void (*p)();
    p=fun;  //赋值，正确
    p();    //调用，正确
}
```

(1) 返回类型匹配
(2) 形参表匹配

```
#include <iostream>
using namespace std;
class Time {
  private:
    int hour;
  public:
    Time() {
      hour=0;
    }
    void display() {
      cout << hour << endl;
    }
};
```

(1) 返回类型匹配　(3) 类匹配
(2) 形参表匹配

```
//指向类成员函数的指针
int main()
{
    Time t1;
    void (*p)();
    p=t1.display;//赋值,错误
    p();     //调用,错误
}
```

2.2.1 指向函数的指针变量

- 指向类成员函数的指针变量

```
//指向全局函数的指针
#include <iostream>
using namespace std;
void fun()
{
    cout << "fun()" <<endl;
}
int main()
{
    void (*p)();
    p=fun;  //赋值，正确
    p();    //调用，正确
}
```

(1) 返回类型匹配
(2) 形参表匹配

```
#include <iostream>
using namespace std;
class Time {
  private:
    int hour;
  public:
    Time() {
      hour=0;
    }
    void display() {
      cout << hour << endl;
    }
};
```

```
//指向类成员函数的指针
int main()
{
 Time t1;
 void (Time::*p)();
 p=&Time::display;//赋值,正确
 (t1.*p)();       //调用,正确
}
```

(1) 返回类型匹配　(3) 类匹配
(2) 形参表匹配　类成员函数必须是public

2.2.1 指向函数的指针变量

- 指向类成员函数的指针变量

  - 定义：成员函数返回类型 (类::*指针变量名)(形参表)

  - 赋值：指针变量名 = &类::成员函数名

```
Time t1, t2;
void (Time::*p)();
p=&Time::display;
(t1.*p)() ⇔  t1.display()
(t2.*p)() ⇔  t2.display()
(t1.p)();  //错误，t1无p成员
```

//使用：(对象名.*指针变量名)(实参表)

2.2.2 返回值为指针的函数

- 定义

  返回基类型 *函数名（形参表）

  ```
  int *fun(int x)
  float *function(char ch)
  ```

- 区别

```
int *fun(int x); //fun是函数名(函数形参为一个int型，返回类型 int *)
int (*fun)(int x); //fun是指针变量名，
                   指向函数(函数形参为一个int型，返回类型 int)
```

- return中的返回值必须是指针（地址）

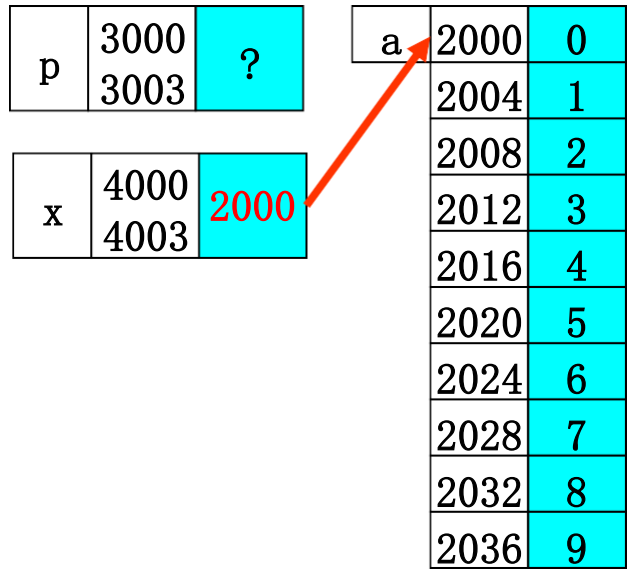
```
#include <iostream>
using namespace std;
int *fun(int *x)
{
   x++;
   return x;
}
int main()
{
   int a[10]={0,1,2,3,4,5,6,7,8,9}, *p;
   p=fun(a);
   cout << "*p=" << *p << endl;
   p=fun(a+5);
   cout << "*p=" << *p << endl;
}
```

*p=1

*p=6

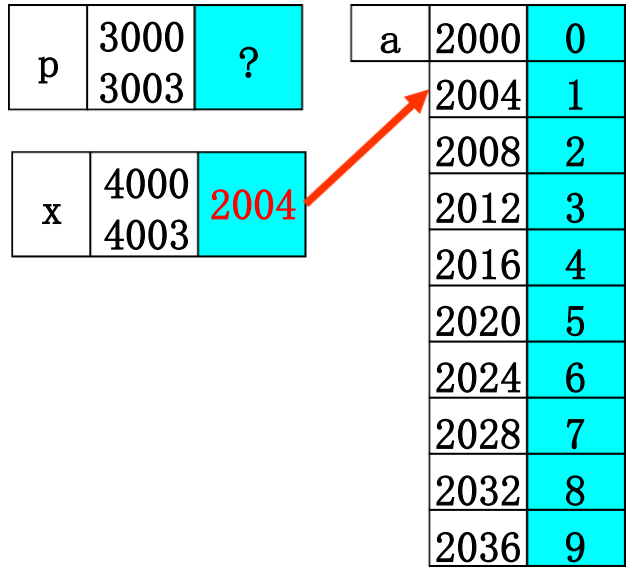
• return中的返回值必须是指针（地址）

```
#include <iostream>
using namespace std;
int *fun(int *x)
{
   x++;
   return x;
}
int main()
{
   int a[10]={0,1,2,3,4,5,6,7,8,9}, *p;
   p=fun(a);
   cout << "*p=" << *p << endl;
   p=fun(a+5);
   cout << "*p=" << *p << endl;
}
```



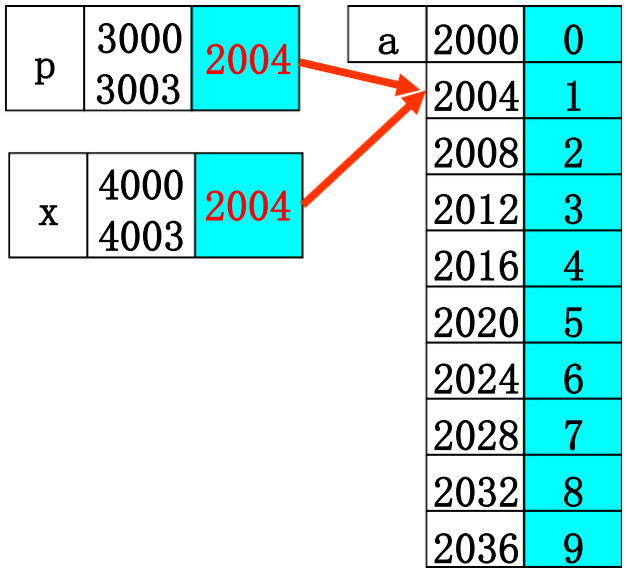
- return中的返回值必须是指针（地址）

```
#include <iostream>
using namespace std;
int *fun(int *x)
{
   x++;
   return x;
}
int main()
{
   int a[10]={0,1,2,3,4,5,6,7,8,9}, *p;
   p=fun(a);
   cout << "*p=" << *p << endl;
   p=fun(a+5);
   cout << "*p=" << *p << endl;
}
```



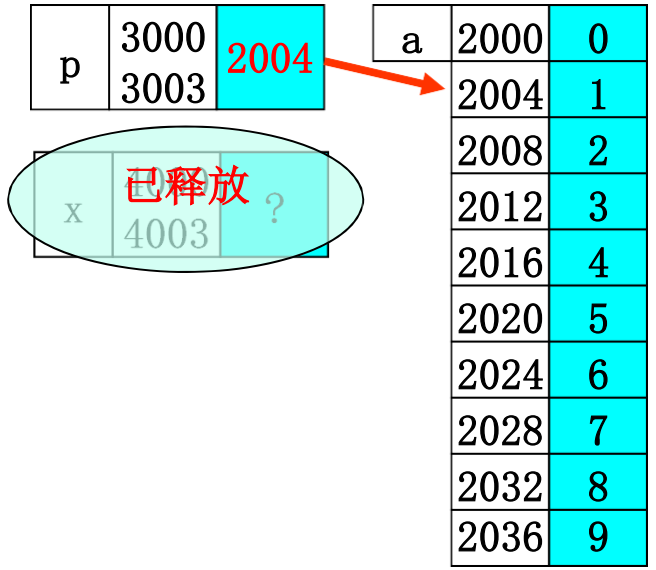
- return中的返回值必须是指针（地址）

```
#include <iostream>
using namespace std;
int *fun(int *x)
{
   x++;
   return x;
}
int main()
{
   int a[10]={0,1,2,3,4,5,6,7,8,9}, *p;
   p=fun(a);
   cout << "*p=" << *p << endl;
   p=fun(a+5);
   cout << "*p=" << *p << endl;
}
```



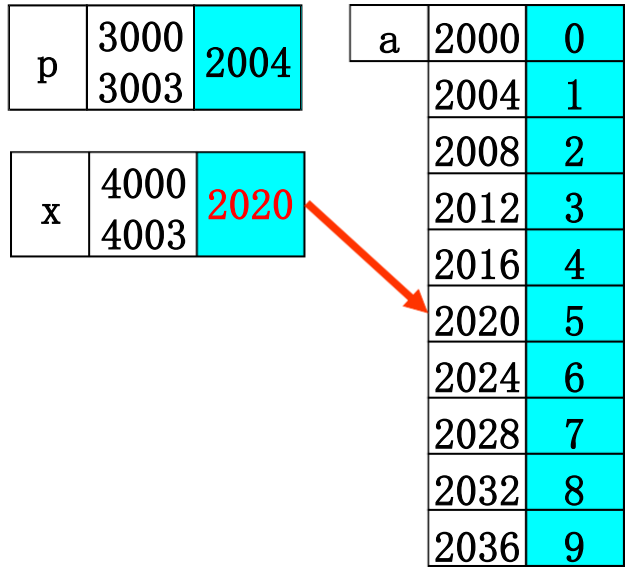
• return中的返回值必须是指针（地址）

```
#include <iostream>
using namespace std;
int *fun(int *x)
{
   x++;
   return x;
}
int main()
{
   int a[10]={0,1,2,3,4,5,6,7,8,9}, *p;
   p=fun(a);
   cout << "*p=" << *p << endl;
   p=fun(a+5);
   cout << "*p=" << *p << endl;
}
```



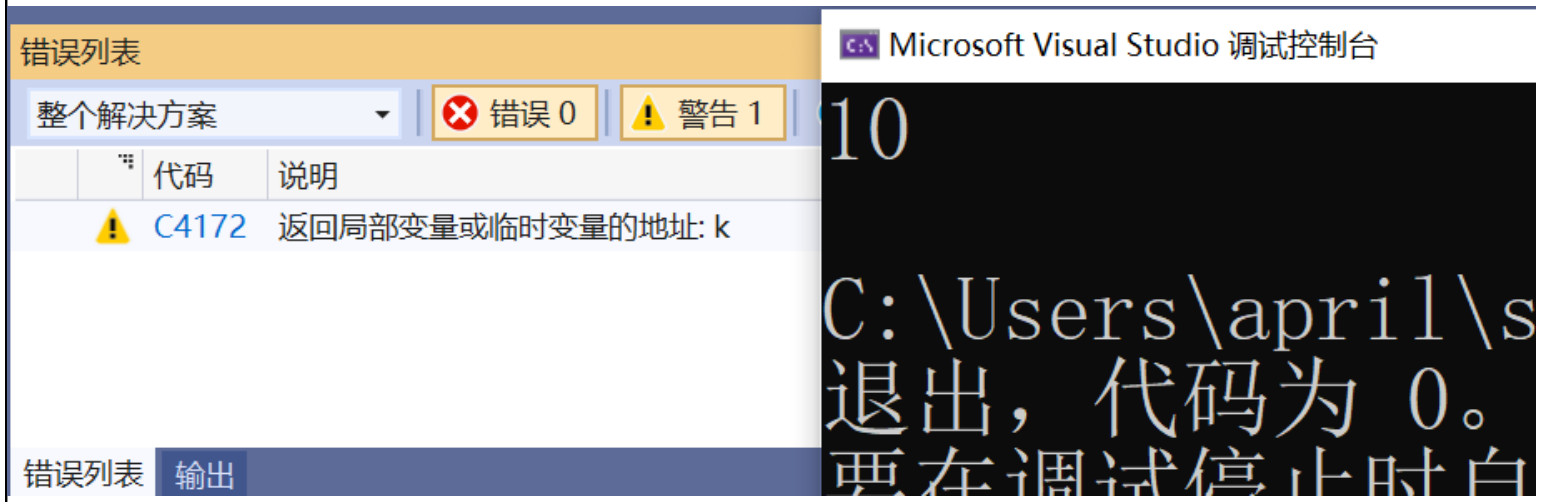
• return中的返回值必须是指针（地址）

```
#include <iostream>
using namespace std;
int *fun(int *x)
{
   x++;
   return x;
}
int main()
{
   int a[10]={0,1,2,3,4,5,6,7,8,9}, *p;
   p=fun(a);
   cout << "*p=" << *p << endl;
   p=fun(a+5);  //与上述过程类似
   cout << "*p=" << *p << endl;
}
```



- 不能返回一个自动变量/形参的地址，否则可能出错

```
#include <iostream>
using namespace std;
int *fun()
{
  int k=10;
  return &k;
}
int main()
{
  int *p;
  p=fun();
  cout << *p << endl;
}
```



1、VS2019下有编译警告
2、有运行结果，不会死机

- 不能返回一个自动变量/形参的地址，否则可能出错

```
#include <iostream>
using namespace std;
int *fun()
{
  int k=10;
  return &k;
}
int main()
{
  int *p;
  p=fun();
  cout << *p << endl;
}
```

k	3000 3003	10
p	2000 2003	?

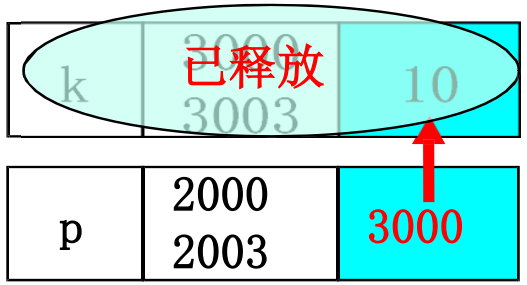
- 不能返回一个自动变量/形参的地址，否则可能出错

```
#include <iostream>
using namespace std;
int *fun()
{
  int k=10;
  return &k;
}
int main()
{
  int *p;
  p=fun();
  cout << *p << endl;
}
```

k	3000 3003	10
p	2000 2003	3000

- 不能返回一个自动变量/形参的地址，否则可能出错

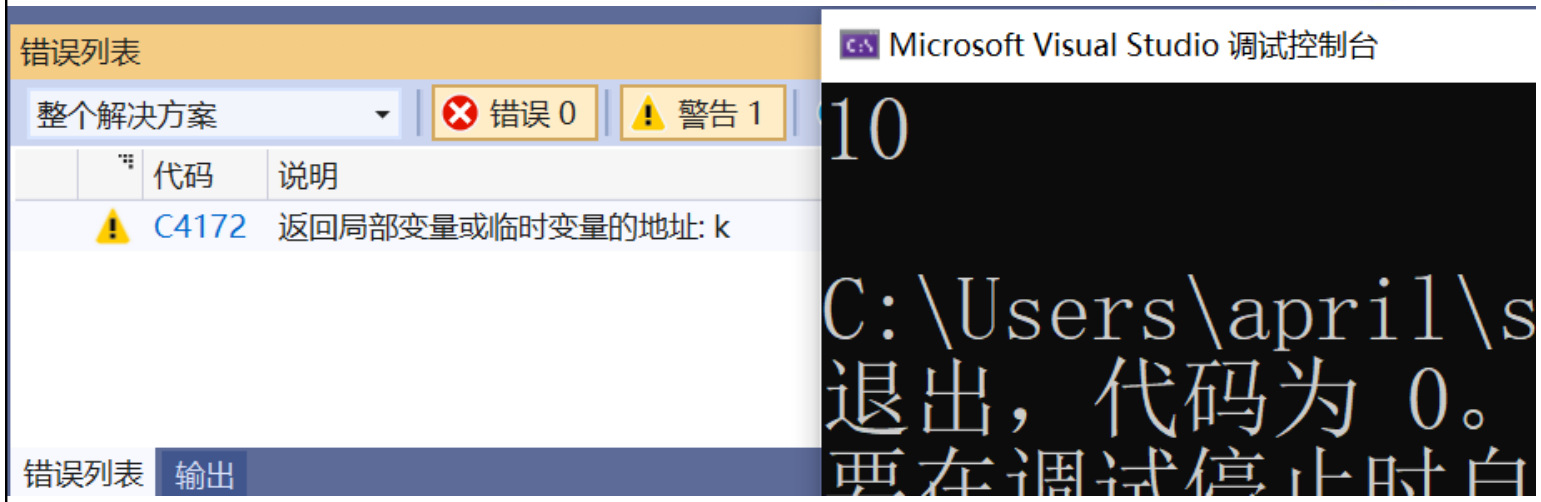
```
#include <iostream>
using namespace std;
int *fun()
{
  int k=10;
  return &k;
}
int main()
{
  int *p;
  p=fun();
  cout << *p << endl;
}
```

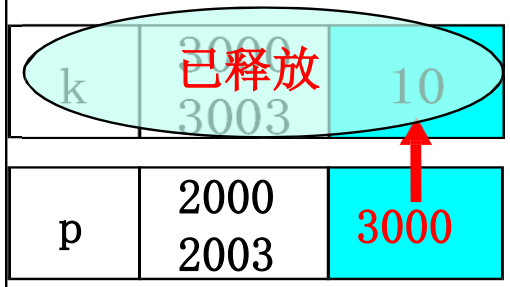


cout时，k所占空间已释放
若未被再次分配并赋值：*p=10
若已被再次分配并赋值：*p=其他

- 不能返回一个自动变量/形参的地址，否则可能出错

```
#include <iostream>
using namespace std;
int *fun()
{
  int k=10;
  return &k;
}
int main()
{
  int *p;
  p=fun();
  cout << *p << endl;
}
```



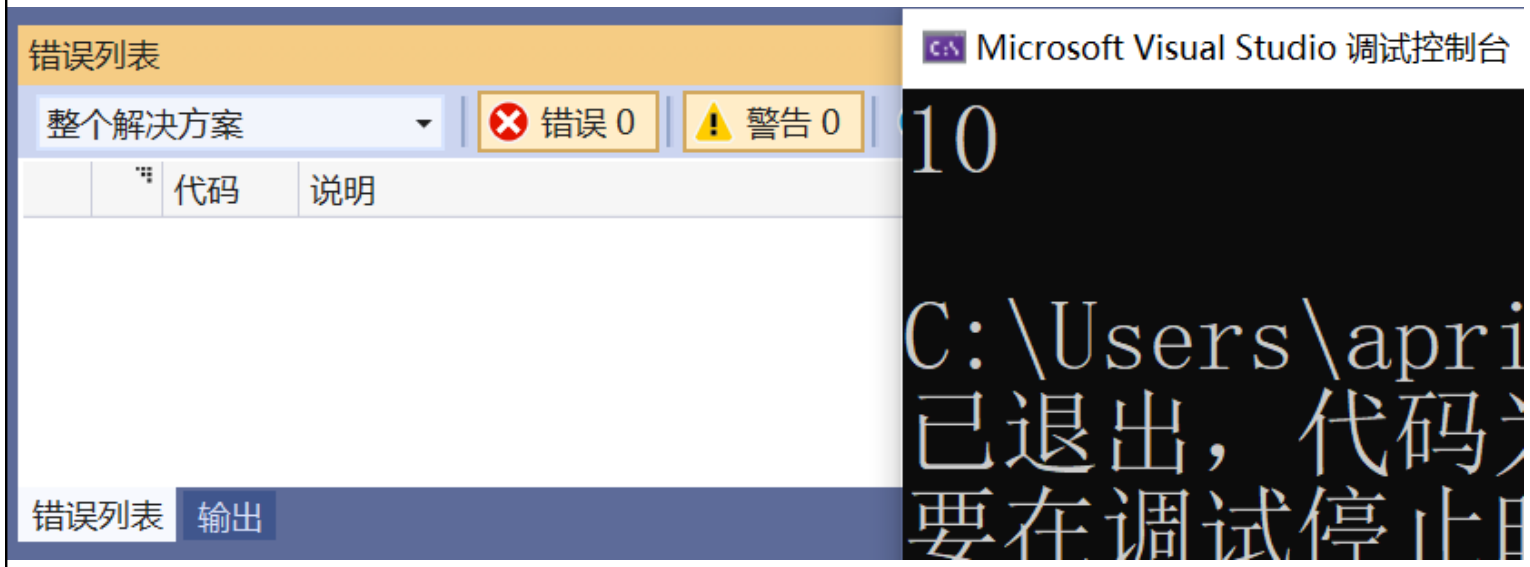


1、VS2019下有编译警告

2、有运行结果，不会死机 ➡ “读”已释放的非法空间而未“写”

- 若返回一个自动变量/形参的地址，可设置为静态局部变量

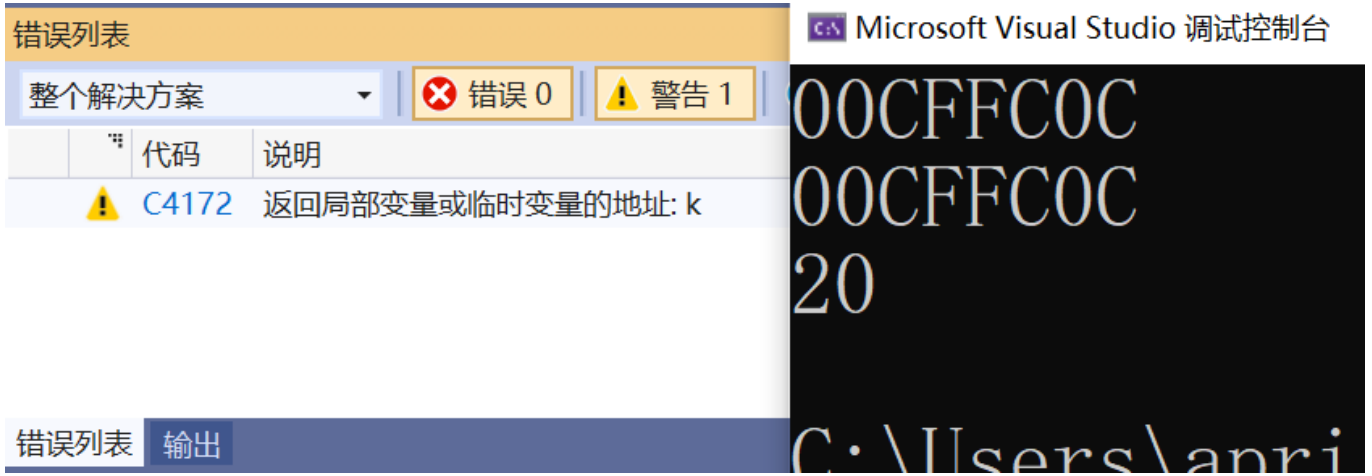
```
#include <iostream>
using namespace std;
int *fun()
{
  static int k=10;
  return &k;
}
int main()
{
  int *p;
  p=fun();
  cout << *p << endl;
}
```



- 不能返回一个自动变量/形参的地址，否则可能出错

```
#include <iostream>
using namespace std;
int *fun()
{
    int k=10;
    cout << &k << endl;
    return &k;
}
void fun2()
{
    int m=20;
    cout << &m << endl;
}
```

```
int main()
{
    int *p;
    p=fun();
    fun2();
    cout << *p << endl;
}
```



- 不能返回一个自动变量/形参的地址，否则可能出错

```
#include <iostream>
using namespace std;
int *fun()
{
    int k=10;
    cout << &k << endl;
    return &k;
}
void fun2()
{
    int m=20;
    cout << &m << endl;
}
```

```
int main()
{
    int *p;
    p=fun();
    fun2();
    cout << *p << endl;
}
```

k	3000 3003	10
p	2000 2003	?

- 不能返回一个自动变量/形参的地址，否则可能出错

```
#include <iostream>
using namespace std;
int *fun()
{
    int k=10;
    cout << &k << endl;
    return &k;
}
void fun2()
{
    int m=20;
    cout << &m << endl;
}
```

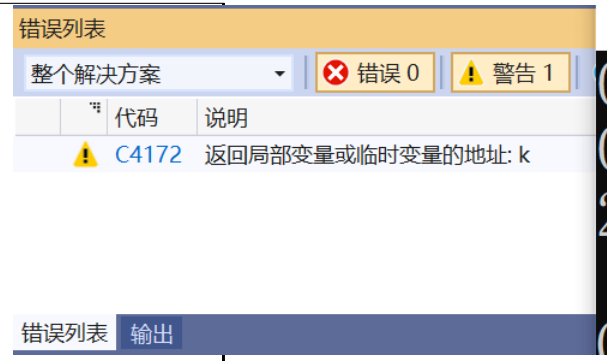
```
int main()
{
    int *p;
    p=fun();
    fun2();
    cout << *p << endl;
}
```

k	3000 3003	10
p	2000 2003	3000

- 不能返回一个自动变量/形参的地址，否则可能出错

```
#include <iostream>
using namespace std;
int *fun()
{
    int k=10;
    cout << &k << endl;
    return &k;
}
void fun2()
{
    int m=20;
    cout << &m << endl;
}
```

```
int main()
{
    int *p;
    p=fun();
    fun2();
    cout << *p << endl;
}
```



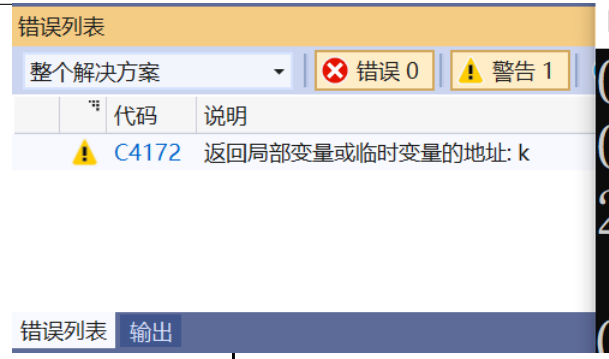
k/m	3000 3003	20
p	2000 2003	3000

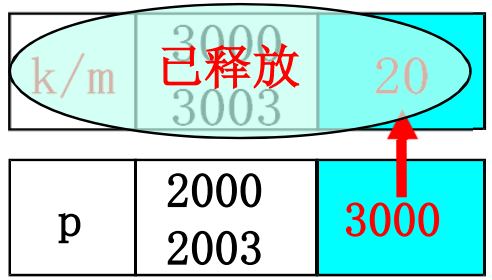
两次打印地址相同：调用fun2时，虽然k所占空间已释放，但是m申请的空间会重复利用原k的空间（VS下常见的内存处理方式）

- 不能返回一个自动变量/形参的地址，否则可能出错

```
#include <iostream>
using namespace std;
int *fun()
{
    int k=10;
    cout << &k << endl;
    return &k;
}
void fun2()
{
    int m=20;
    cout << &m << endl;
}
```

```
int main()
{
    int *p;
    p=fun();
    fun2();
    cout << *p << endl;
}
```





打印*p时，k/m所占空间已释放
只读已释放的非法空间

目录

- 多维数组与指针
- 函数与指针
- 指针数组
- 指向指针的指针
- 其它指针
- 数组的引用
- 总结与应用

2.3 指针数组

- 含义：元素类型是指针的数组
- 定义：数据类型 *数组名[数组长度]

```
int *p[4];
```
（一层层看）

p是数组名，有4个元素

每个元素是int *

- 指针数组与指向m个元素的一维数组的指针的比较

```
int (*p)[4];
```

p是指针变量名

指向由4个元素组成的一维数组

2.3 指针数组

- 二维字符数组和一维指针数组的区别

二维字符数组：

```
char a[3][10] = {"china", "student", "s"};
```

a[0]	2000	c
	2001	h
	2002	i
	2003	n
	2004	a
	2005	\0
	2006	
	2007	
	2008	
	2009	

a[1]	2010	s
	2011	t
	2012	u
	2013	d
	2014	e
	2015	n
	2016	t
	2017	\0
	2018	
	2019	

a[2]	2020	s
	2021	\0
	2022	
	2023	
	2024	
	2025	
	2026	
	2027	
	2028	
	2029	

字符串常量

赋初值的方法 "china"(无名)

a[0]	2000	c
	2001	h
	2002	i
	2003	n
	2004	a
	2005	\0
	2006	
	2007	
	2008	
	2009	

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

2.3 指针数组

- 二维字符数组和一维指针数组的区别

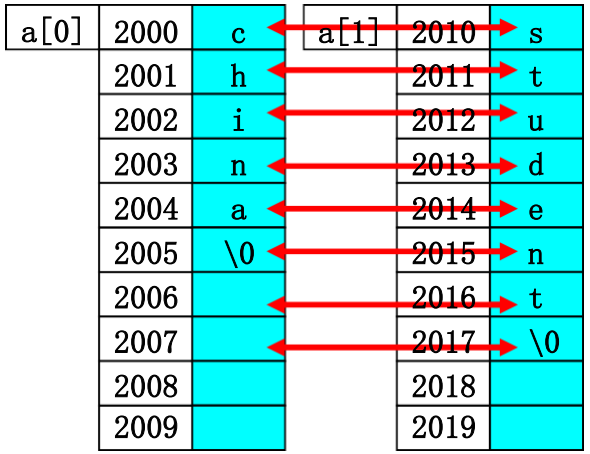
二维数组交换的方法：

```
char tmp[10];

strcpy(tmp, a[0]);

strcpy(a[0],a[1]);

strcpy(a[1],tmp);
```



优点：(1) 与无名字符串常量分占不同空间
(2) 字符串的值可以修改

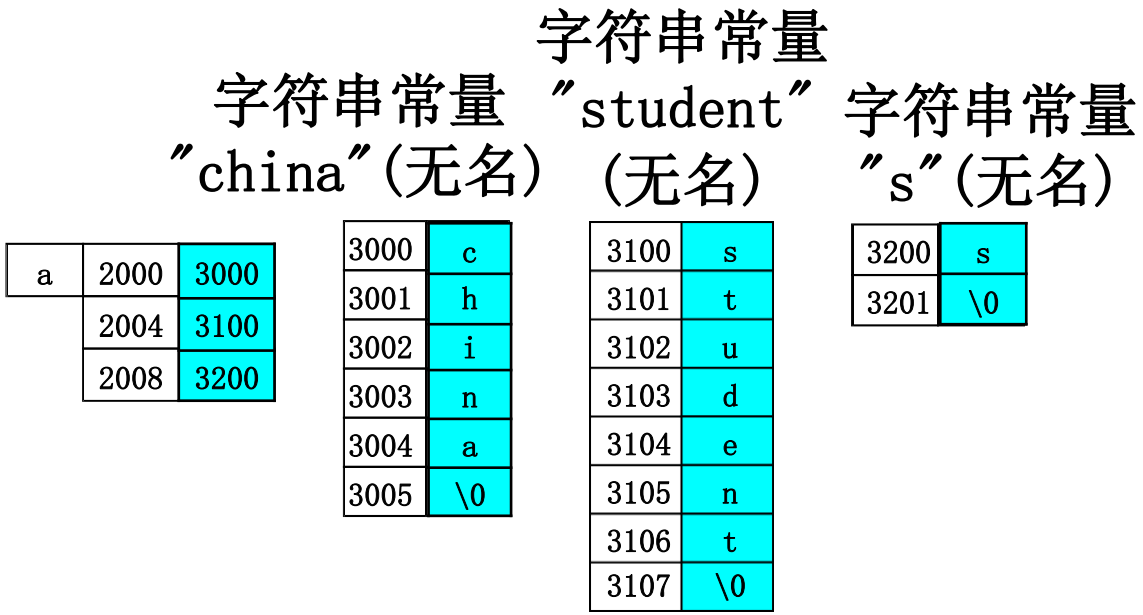
缺点：(1) 有空间浪费
(2) 若要交换元素(例如排序)，则需要整体移动元素

2.3 指针数组

- 二维字符数组和一维指针数组的区别

一维指针数组：

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



2.3 指针数组

- 二维字符数组和一维指针数组的区别

一维指针数组交换的方法：

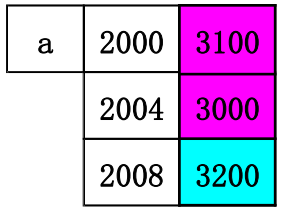
```
char *a[3] = {(char *)"china", (char *) "student", (char *) "s"};

char *tmp;

tmp = a[0];

a[0] = a[1];

a[1] = tmp;
```



优点：(1) 节约空间
(2) 交换是只需交换指针值即可，效率高

缺点：(1) 用指针指向无名字符串常量，无法改变字符串的值

2.3 指针数组

- 二维字符数组和一维指针数组的区别
  - 二维字符数组分配实际的字符串存储空间，在执行过程中可以修改字符串任意位置的值
  - 一维指针数组不分配实际字符串存储空间，只是指向字符串常量，在执行过程中字符串值不能改变

- 二维字符数组在执行过程中可以修改字符串任意位置的值

```
#include <iostream>
using namespace std;
int main()
{    char a[3][10]={"china","student","s"};
     cout << a[0] << endl;   //china
     cout << a[1] << endl;   //student
     cout << a[2] << endl;   //s
     a[0][0]-=32;
     cout << a[0] << endl;   //China
     cout << a[1] << endl;   //student
     cout << a[2] << endl;   //s
     return 0;
}
```

a[0]	2000	c=>C
	2001	h
	2002	i
	2003	n
	2004	a
	2005	\0
	2006	
	2007	
	2008	
	2009	

a[1]	2010	s
	2011	t
	2012	u
	2013	d
	2014	e
	2015	n
	2016	t
	2017	\0
	2018	
	2019	

a[2]	2020	s
	2021	\0
	2022	
	2023	
	2024	
	2025	
	2026	
	2027	
	2028	
	2029	

- 一维指针数组在执行过程中字符串值不能改变

```
#include <iostream>
using namespace std;
int main()
{    char *a[3]= {(char *)"china", (char *)"student", (char *)"s"};
     cout << a[0] << endl;  //china
     cout << a[1] << endl;  //student
     cout << a[2] << endl;  //s
     a[0][0]-=32;  //编译不错运行错
     cout << a[0] << endl;
     cout << a[1] << endl;
     cout << a[2] << endl;
     return 0;
}
```

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 main()
{    char *a[3]= {(char *)"china", (char *)"student", (char *)"s"};
     char b[10]="hello";
     cout << a[0] << endl;  //china
     cout << a[1] << endl;  //student
     cout << a[2] << endl;  //s
     a[0]= b; //a[0]存放数组b的首地址
     cout << a[0] << endl;  //hello
     cout << a[1] << endl;  //student
     cout << a[2] << endl;  //s
     return 0;
}
```

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

a	2000	3000->3300
	2004	3100
	2008	3200

数组b

3300	h=>H
3301	e
3302	l
3303	l
3304	o
3305	\0
3306	
3307	
3308	
3309	

```
     a[0][0] -= 32; //修改数组b[0]的元素值
     cout << a[0] << endl;  //Hello
     cout << a[1] << endl;  //student
     cout << a[2] << endl;  //s
```

```
int main()
{
        char a[3][10] = {"china","student","s"};
        char b[10]="hello";
        cout << a[0] << endl;
        cout << a[1] << endl;
        cout << a[2] << endl;
        a[0] = b;                    //编译错误，非运行错误！！
        cout << a[0] << endl;
        cout << a[1] << endl;
        cout << a[2] << endl;
        a[0][0]-=32;
        cout << a[0] << endl;
        cout << a[1] << endl;
        cout << a[2] << endl;
        return 0;
}
```

```
//谭书P.173 例6.12（一维指针数组）
void sort(char *name[], int n)
{
  char *temp;
  int i,j,k;
  for(i=0; i<n-1; i++) {
     k=i;
    for(j=i+1; j<n; j++)
      if (strcmp(name[k],name[j])>0)
        k=j;
    if (k!=i) {
      temp=name[i];
      name[i]=name[k];
      name[k]=temp;
      }
    }
}
```

main函数中

```
char *name[]={"BASIC",…};
```

```
//谭书P.178 例6.12（二维字符数组）
void sort(char name[][8], int n)
{
  char temp[8];
  int i,j,k;
  for(i=0; i<n-1; i++) {
     k=i;
    for(j=i+1; j<n; j++)
      if (strcmp(name[k],name[j])>0)
        k=j;
    if (k!=i) {
      strcpy(temp,name[i]);
      strcpy(name[i],name[k]);
      strcpy(name[k],temp);
      }
    }
}
```

main函数中

```
char name[][8]={"BASIC",…};
```

目录

- 多维数组与指针
- 函数与指针
- 指针数组
- 指向指针的指针
- 其它指针
- 数组的引用
- 总结与应用

2.4 指向指针的指针

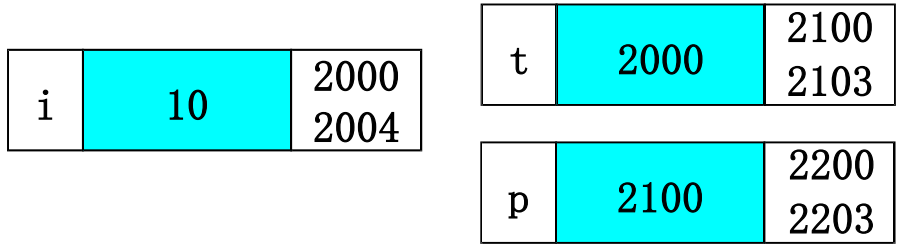
- 定义：

  数据类型 **指针变量名

  int **p;　（一层层看）

  p是指针变量

  指向一个指针变量，该指针变量的基类型是int



- 使用：　int i=10, *t, **p;
  t=&i;　(普通变量的地址)
  p=&t;　(指针变量的地址)

⇨ 定义时赋初值的写法：
int i=10,*t=&i,**p=&t;

p=地址（指向普通变量的指针变量的地址）
*p=地址(普通变量的地址)
**p=值（普通变量）

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

p	2100	2000

a	2000	3000
	2004	3100
	2008	3200

//示例：

```
#include <iostream>
using namespace std;
int main()
{
    char *a[3] = {(char *)"china", (char *)"student", (char *)"s"}, **p;
    p=a;
    cout << p << endl;                  //地址2000
    cout << p+1 << endl;                //地址2004   因为指针
    cout << hex << int(*p) << endl;     //地址3000   串首地址
    cout << *p << endl;                 //china
    cout << *p+3 << endl;               //na
    cout << *(*p+3) << endl;            //n
}
```

Microsoft Visual Studio 调试控制台

```
00AFFC94
00AFFC98
119b30
china
na
n
```

思考：为什么是 cout << hex << int(*p) << endl;

//复习：

```
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

目录

- 多维数组与指针
- 函数与指针
- 指针数组
- 指向指针的指针
- 其它指针
- 数组的引用
- 总结与应用

2.5.1 const指针

- 引入：
  - 数据共享带来便捷，但可能承担非预期的改变从而带来的错误。

    如：通过指针改变计划外的局部变量数值
  - 既能达到数据共享，又不会因误操作而改变 ➡ 共用数据保护
  - 使用const关键词，把有关数据定义为常量

2.5.1 const指针

1）指向常量的指针变量

- 形式

  const 数据类型 *指针变量名 （看const修饰谁来理解，后同）

  或 数据类型 const *指针变量名

- 作用：
  - 不能通过指针修改变量的值(仍可以通过变量修改)
  - 指针变量可以指向其它同类型变量(不必在定义时初始化)
  - 用于不希望通过指针修改变量值的情况

- 指向常量的指针变量

```
#include <iostream>
using namespace std;
int main()
{
    int a=12, b=15;
    const int *p;//int const *p;
    p = &a;
    cout << *p << endl;
    *p = 10;        //编译报错
    a = 10;         //正确
    cout << *p << endl;
    p = &b;
    cout << *p << endl;
}
        //只是不能通过p改变a的值
```

应用举例

```
#include <iostream>
using namespace std;
void fun(const int *x)
{
      x++ / x+=2 等操作：    可以
      *x=10 / (*x)++ 等操作：不可以
}
int main()
{
    int a[10]={1,2,3,4,5,6,7,8,9,10};
    fun(a);
}
        //保证在fun函数中仅能访问而不会
        改变实参数组的值(防止误操作)
```

2.5.1 const指针

1）指向常量的指针变量

- 指向常量的指针变量可以指向常变量、普通变量
- 普通指针不能指向常变量

int a;	const int a = 10;
const int b;	int *p1;
const int *p;	const int *p2;
p = &a; //正确	p1 = &a; //编译错
p = &b; //正确	p2 = &a; //正确

2.5.1 const指针

1）指向常量的指针变量

- 形参：普通指针，实参：普通变量地址/指针　　正确
- 形参：普通指针，实参：常变量地址/指针　　错误

```
void f(int *p)
{   return;
}
int main()
{   int x;
    const int *p;
    p = &x;
    f(p); //编译错
}
```

```
void f(int *p)
{   return;
}
int main()
{   int x;
    f(&x);
} //正确
```

```
void f(int *p)
{   return;
}
int main()
{   const int x = 10;
    f(&x); //编译错
}
```

2.5.1 const指针

1）指向常量的指针变量

- 形参：常变量指针，实参：普通变量地址/指针　　正确
- 形参：常变量指针，实参：常变量地址/指针　　正确

```
void f(const int *p)
{   return;
}
int main()
{   int x;
    f(&x);
} //正确
```

```
void f(const int *p)
{   return;
}
int main()
{   const int x = 10;
    f(&x);
} //正确
```

2.5.1 const指针

2）常指针

- 形式：数据类型 *const 指针变量名
- 作用：
  - 可以通过指针修改变量的值
  - 指针变量指向固定变量(必须在定义时初始化)后，不能再指向其它同类型变量
  - 适用于希望指针始终指向某个变量的情况

- 常指针

```
#include <iostream>
using namespace std;
int main()
{
    int a=12, b=15;
    int *const p = &a;
      //定义时必须初始化
    cout << *p << endl;
    *p = 10;
    cout << *p << endl;
    p = &b;       //编译错
    cout << *p << endl;
    return 0;
}
```

应用举例

```
#include <iostream>
using namespace std;
void fun(int *const x)
{
    x++ / x+=2 等操作：                    不可以
    if (x+2 < 另一个指针) 等操作：          可以
    *(x+2)=10 / (*x)++ / *x==10 等操作：可以
}
int main()
{   int a[10]={1,2,3,4,5,6,7,8,9,10}, *p=&a;
    fun(p);
    return 0;
}
```

//保证在fun函数中p始终指向a
并能读写a数组(防止误操作)

//实参：常指针
//形参：普通指针

```
void f(int *p)
{
   return;
}
int main()
{
   int x;
   int *const p =&x;
   f(p);
   return 0;
}
```

//实参：常指针
//形参：常指针

```
void f(int *const p)
{
   return;
}
int main()
{
   int x;
   int *const p = &x;
   f(p);
   return 0;
}
```

//实参：普通变量地址
//形参：普通指针

```
void f(int *p)
{
   return;
}
int main()
{
   int x;
   f(&x);
}
```

//实参：普通变量地址
//形参：常指针

```
void f(int *const p)
{
   return;
}
int main()
{
   int x;
   f(&x);
}
```

//常指针不能指向常变量

```
const int x = 10;
int *const p = &x;
//编译错
```

```
void f(int *const p)
{
    return;
}
int main()
{
    const int x=10;
    f(&x);
    return 0;
} //编译错
```

2.5.1 const指针

3）指向常量的常指针

- 形式：const 数据类型 *const 指针变量名
- 作用：
  - 不能通过指针值修改变量的值
  - 指针变量指向固定变量(必须在定义时初始化)后，不能再指向其它同类型变量
  - 适用于既希望始终指向固定变量，又希望不能通过指针修改变量值的情况

- 指向常量的常指针

```
#include <iostream>
using namespace std;
int main()
{
    int a=12, b=15;
    const int *const p=&a; //必须定义时初始化
    cout << *p << endl;
    *p = 10;     //编译错
    a = 10;
    cout << *p << endl;
    p = &b;      //编译错
    cout << *p << endl;
}
```

- 指向常量的常指针

```
void f(int const *const p)
{
    return;
}
```

```
int main()
{
    int x;
    f(&x);
}
```

√

```
int main()
{
    const int x = 10;
    f(&x);
}
```

√

```
void f(int *p)
{
    return;
}
```

```
int main()
{
    int x;
    int const *const p = &x;
    f(p);
}
```

×

```
int main()
{
    const int x = 10;
    int const *const p = &x;
    f(p);
}
```

×

2.5.1 const指针

- 按读写/只读方式区分，实形参的组合一共四种

实参只读 => 形参只读

实参只读 => 形参读写 //错误

实参读写 => 形参只读

实参读写 => 形参读写

思考题：c++中const的作用（可以修饰谁？）

1. const用于定义常量

```
#define PI 3.1415926          //符号常量
const double PI=3.1414926     //常变量        本质区别？
```

2. const修饰函数形参

```
void fun(A a);                //值传递
void fun(A const &a);         //const &传递      效率区别？
```

3. const修饰函数的返回值

```
const char *Getchar(void){};//返回值不能被直接修改
char *ch=Getchar();           //错误
const char *ch=Getchar();     //正确，只能赋值给同类型指针
```

4. const修饰类的成员函数

```
int GetCount(void) const; //不会修改数据成员
```

2.5.2 void指针

- 含义：
  指向空类型的指针变量
- 使用：
  - 不能直接通过void指针访问数据(不知道基类型)，必须强制转换为某种确定数据类型后才能访问
  - 非void型的指针可直接赋值给void类型，void类型赋值给非void类型时必须强制转换

```
#include <iostream>
using namespace std;
int main()
{
    int i=10, *p1=&i, *p3;
    void *p2;
    cout << p1 << endl;  //地址i
    cout << *p1 << endl; //10
    cout << *p2 << endl; //编译错误
    p2 = p1;
    p3 = p2; //编译错误
                  改为：p3=(int *)p2
    cout << *p3 << endl; //10
}
```

2.5.2 void指针

- void可以声明指针类型，但不能++/--
- void不能声明变量，但可以是函数的形参及返回值

```
void k;          //错误，不允许
void *p;         //正确
++p;   p--;      //错误，因为不知道基类型的大小
```

- void型的指针变量不能进行相互运算(因为不知道基类型)

```
void *p, *q;
cout << (p+2) << endl;    //编译错
cout << (q--) << endl;    //编译错
cout << (p-q) << endl;    //编译错
cout << (p<q+1) << endl; //编译错
```

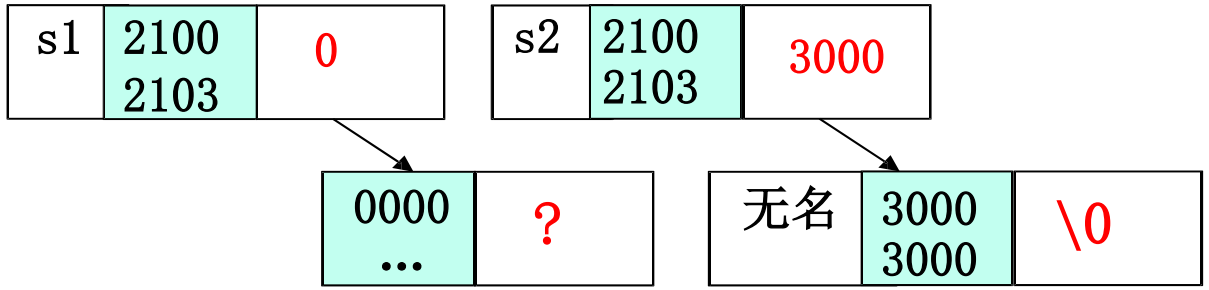
2.5.3 空指针NULL

- 基本概念
  - 指针允许有空值NULL(系统宏定义#define NULL 0)，表示不指向任何变量(若定义指针变量未赋初值，则随机指向，称为野指针)
  - NULL与空字符串的区别：
    - char *s1 = NULL; //s1是指针，存放地址0，地址0中的内容不一定是'\0'，即strlen(s1)不一定为0
    - char *s2 = ""; //s2是指针，存放一个长度为0的无名字符串常量的首地址(非0)，strlen(s2)为0
  - 系统的字符串操作函数若传入参数为NULL则会出错
    - 包括strcpy/strcat/strcmp/strlen/strncpy/strncmp等，以及未出现过的同类函数

2.5.3 空指针NULL

NULL与空字符串的区别：

```
char *s1 = NULL;        //s1是指针，存放地址0，地址0中的内容不一定是'\0'，
                        //即strlen(s1)不一定为0
char *s2 = "";          //s2是指针，存放一个长度为0的无名字符串常量的首地址(非0),
                        //strlen(s2)为0
```




```
char s3[]="";    //正确
char s4[]=NULL; //错，不能用无{}的一个数字初始化
```

```
#include <iostream>
using namespace std;
int main()
{   char s3[]="";
    char s4[]=NULL; //编译报错
}
```

2.5.3 空指针NULL

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{   char *s1 = NULL;
    int len;
    len=strlen(s1);
}
```

错

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;
int main()
{   char *s1 = NULL;
    char s2[80]="Hello";
    strcat(s2, s1);
}
```

错

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;
int main()
{   char *s1 = NULL;
    char s2[80]="Hello";
    strcpy(s2, s1);
}
```

错

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{   char *s1 = NULL;
    char *s2 = NULL;
    int k=strcmp(s1,s2);
}
```

错

目录

- 多维数组与指针
- 函数与指针
- 指针数组
- 指向指针的指针
- 其它指针
- 数组的引用
- 总结与应用

2.6 数组的引用

- 数组引用做函数参数
  - 引入：数组长度不是参数类型的一部分，函数不知道传递给它的数组的实际长度。当编译器对实参类型进行参数类型检查时，并不检查数组的长度。后续的数组操作不得当时可能发生越界等运行错误。
  - 问题：如何知道数组的长度？
  - 方法：

    1）提供一个含有数组长度的额外参数

    2）将参数声明为数组的引用

- 数组引用做函数参数

```
#include <iostream>
using namespace std;
void test1(char *s1, const char *s2)
{
    cout << sizeof(s1) << endl; //4
    cout << sizeof(s2) << endl; //4
}
void test2(char (&s1)[10], const char *s2)
{
    cout << sizeof(s1) << endl; //10
    cout << sizeof(s2) << endl; //4
}
```

```
int main()
{
    char s[10], t[]="This is a
    pencil.";
    test1(s, t);
    test2(s, t);
    return 0;
}
```

当形参为实参数组的引用时，编译器不会将数组实参转化为指针，而是传递数组的引用本身。在这种情况下，数组大小成为形参与实参类型的一部分，编译器检查数组实参的大小与形参的大小是否匹配。

- 数组引用做函数参数

```
#include <iostream>
using namespace std;

char *tj_strcpy(char *s1, const char *s2)
{
    char *p1=s1;
    const char *p2=s2;

    while(*p1++ = *p2++)
      ;   依次赋值，含尾零
    return s1;
}
```

```
int main()
{
    char s[10], t[]="This is a pencil.";
    tj_strcpy(s, t);        //运行出错!!!
    cout << s << endl;
    return 0;
}
```

错误原因：函数中无法知道s大小，只能依据t的\0来判断结束

- 数组引用做函数参数

```
#include <iostream>
using namespace std;

char *tj_strcpy(char (&s1)[10], const char *s2)
{
    int i;
    for (i=0; s2[i] != '\0' && i < sizeof(s1)-1; i++)
        s1[i] = s2[i]; //可以做到安全copy，不会越界

    s1[i] = '\0';
    return s1;
}
```

```
int main()
{
    char s[10], t[]="This is a pencil.";
    tj_strcpy(s, t);
    cout << s << endl;
    return 0;
}
```

- 数组引用做函数参数

```
#include <iostream>
using namespace std;
char *tj_strcpy(char (&s1)[10], const char *s2)
{
     int i;
     for (i=0; s2[i] != '\0' && i < sizeof(s1)-1; i++)
          s1[i] = s2[i];
     s1[i] = '\0';
     return s1;
}
char *tj_strcpy(char *s1, const char *s2)
{
     char *p1=s1;
     const char *p2=s2;
     while(*p1++ = *p2++)
     ;
     return s1;
}

int main()
{
     char s[10], t[]="This is a pencil.";
     tj_strcpy_s(s, t);
     cout << s << endl;
     return 0;
}
```

E0308 有多个 重载函数 "tj_strcpy" 实例与参数列表匹配:

C2668 "tj_strcpy": 对重载函数的调用不明确

- 形参是数组的引用，则实参必须是数组，且大小一致，不能是数组的指针

```
#include <iostream>
using namespace std;
void test2(char (&s1)[10])
{
    cout << sizeof(s1) << endl;
}
int main()
{
    char s[10], *p = s;
    test2(s);
    test2(p);    //编译错
}
```

//无法将参数1从char *转为char(&)[10]

```
#include <iostream>
using namespace std;
void test2(char (&s1)[10])
{
    cout << sizeof(s1) << endl;
}
void test1(char s1[])
{
    cout << sizeof(s1) << endl;
    test2(s1);    //编译错
}
int main()
{
    char s[10]; test1(s);
}
```

//无法将参数1从char[]转为char(&)[10]

目录

- 多维数组与指针
- 函数与指针
- 指针数组
- 指向指针的指针
- 其它指针
- 数组的引用
- 总结与应用

2.7 总结与应用

- 声明指针 //书P109-P111
  - typeName * pointerName
- 给指针赋值
  - 将内存地址赋给指针（&，new）
- 对指针解除引用
  - 获得指针指向的值（*，数组下标）
- 区分指针、值、各类指针运算
  - 复习

```
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;
}
```

2.7 总结与应用

- 数组的动态联编和静态联编（第三章讲）
  - 动态：int * pz = new int [size];

    …

    delete [] pz;
  - 静态：int tacos[10];
- 下标法和指针法
  - 下标法：a[i]
  - 指针法：*(a+i)
- 多维数组的理解方法（复习：以二维数组为例）

2.7 总结与应用

- 不同类型的指针变量不能相互赋值，需要进行强制类型转换

70000 = 00000000 00000001 00010001 01110000

```
#include <iostream>
using namespace std;
int main()
{
    long a=70000, *p=&a;
    short *p1;
    char  *p2;
    p1 = p; //编译错
    p2 = p; //编译错
    cout << *p << endl;
    cout << *p1 << endl;
    cout << *p2 << endl;
    return 0;
}
```

强制类型转换

```
#include <iostream>
using namespace std;
int main()
{
    long a=70000, *p=&a;
    short *p1;
    char  *p2;
    p1 = (short *)p;
    p2 = (char *)p;
    cout << *p << endl;
    cout << *p1 << endl;
    cout << *p2 << endl;
    return 0;
}
```

p	2000
p1	2000
p2	2000

a:低位在前存放

2000	01110000
2001	00010001
2002	00000001
2003	00000000

```
//70000 (2000-2003)
//4464  (2000-2001)
//p     (2000)
```

2.7 总结与应用

- 引用的含义：变量的别名
- 引用的声明：int a, &b=a;　　//a和b表示同一个变量
  - 引用不分配单独的空间(指针变量有单独的空间)
  - 引用必须在声明时进行初始化，指向同类型的变量，在整个生存期内不能再指向其它变量
  - 不能声明引用数组和指向引用的指针，但可声明数组的引用、数组元素的引用和指向指针的引用

2.7 总结与应用

- 变量引用

```
#include <iostream>
using namespace std;
int main()
{
    int a = 1, b = 2;
    int &c;      //编译错误，声明引用的同时必须初始化
    int &d = a;
    &d = b;      //编译错误，引用只能在声明的时候被赋值
    int *p;
    *p = 5;      //运行错误，p没有被初始化，是野指针
    return 0;
}
```

```
const float pi = 3.14f;
float f;  //全局变量
float f1(float r=5)
{
    f = r*r*pi;
    return f;//f赋值给临时变量temp，temp由编译器隐式的建立
}
float& f2(float r=5)
{
    f = r*r*pi;
    return f;//f返回给主函数
}
```

```
int main()
{   float a = f1();
    float& b = f1();
    //错误，对临时变量temp进行引用
    float c = f2();
    float& d = f2();
    //正确，最省空间
    d += 1.0f;
    …
    return 0;
}
```

float& d = f2();//正确

注意：所引用变量的有效期

全局变量f的有效期长于d，所以安全

否则出错（如返回局部变量引用）

2.7 总结与应用

- 指针和引用的区别

	指针	引用
初始化	定义时不必初始化，定义后任意地方可重新赋值	创建的同时必须初始化，即引用到一个有效对象
可修改性	任何时候都可以改变指向	初始化后不能改变为另一个对象的引用
存在NULL	指针可以是NULL	不存在NULL引用，必须指向某个对象
测试需要	指针需要经常进行测试	使用引用之前不需要测试合法性
应用场景	存在指向NULL（不指向任何对象）或不同时刻指向不同对象	指向一个对象后不改变指向

引申思考：传引用还是传指针？

2.7 总结与应用

- 传引用比传指针安全
  - 由于不存在空引用，并且引用一旦被初始化为指向一个对象，它就不能被改变为另一个对象的引用，因此引用很安全
  - 指针可以随时指向别的对象，并且可以不被初始化，或为NULL，所以不安全
  - const指针仍然存在空指针，并且有可能产生野指针

2.7 总结与应用

- 野指针：指向垃圾内存的指针（不是NULL指针）
- 野指针成因
  - 指针变量没有被初始化
  - 指针被free或者delete之后，没有置为NULL
- 野指针危害

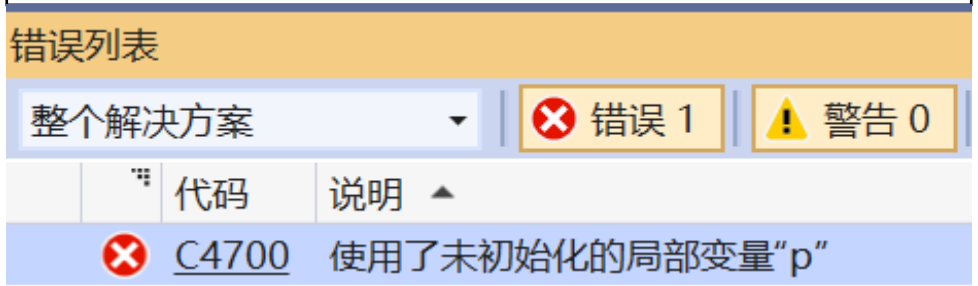
```
short *bufptr;     //未被初始化
short bufarray[20];
short var = 0x20;
*bufptr = var;    //错误，bufptr是野指针，会导致程序运行崩溃
bufarray[0] = var;
```

改为：short *bufptr = (short *) malloc(sizeof(short)); 即正确
//动态内存章节的内容

2.7 总结与应用

- 野指针危害

```
//变量值的交换
void swap (int *p1, int *p2)
{
    int *p;  //野指针
    *p = *p1;
    *p1 = *p2;
    *p2 = *p;
}
//可能导致程序运行时崩溃
```



改为：int *p = (int*)malloc(sizeof(int));
可运行，但需要判断if(p)，否则有warning
建议使用变量而不是指针

```
//变量值的交换
void swap (int *p1, int *p2)
{
    int p;
    p = *p1;
    *p1 = *p2;
    *p2 = *p;
}
//正确
```

本章小结

- 多维数组与指针（熟练）
- 函数与指针（熟练）
- 指针数组（熟练）
- 指向指针的指针（熟练）
- 其它指针（熟练）
- 数组的引用（熟练）
- 总结与应用