§12. 指针进阶

1. 基本概念(复习)
2. 变量与指针(复习)
3. 一维数组与指针(复习)
4. 字符串与指针(复习)
5. 返回指针值的函数(复习)
6. 空指针NULL(复习)

习题课-补1. 引用(复习)

习题课-补2. 不同类型的指针的相互转换(复习)

§12. 指针进阶

7. 多维数组与指针(补充，极其重要!!!)

7.1. 二维数组的地址

★ 一维数组的理解方法(下标法、指针法)

一维数组：

```
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[0])

由等价关系 a[i] ⇔ *(a+i)可得

&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] ：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 地址

a[i] ⇔ *(a+i) 值

第0个元素的特殊表示：

a[0] ⇔ *(a+0) ⇔ *a

&a[0] ⇔ a+0 ⇔ a

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]

§12. 指针进阶

7. 多维数组与指针(补充，极其重要!!!)

7.1. 二维数组的地址

二维数组：

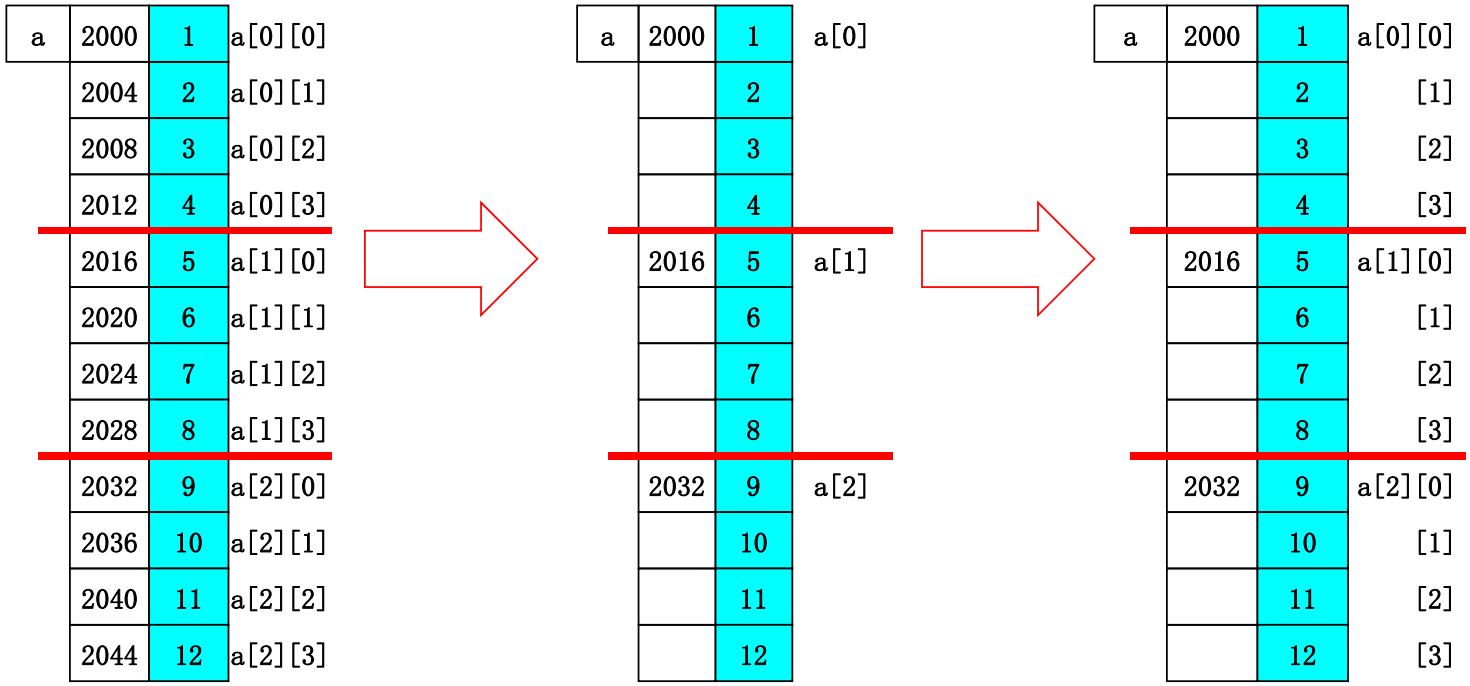
```
int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
    1   2   3   4
    5   6   7   8
    9   10  11  12
```

第5章的内容：
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

理解1：a | [3][4]

理解2：a[3] | [4]



§12. 指针进阶

7. 多维数组与指针(补充，极其重要!!!)

7.1. 二维数组的地址

★ 二维数组加一个下标的理解方法(下标法、指针法)

元素是指
4元素一维数组

```
int a[3][4]={1,…,12};
```

3种理解方法

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]	行地址
&a[i]	3元素一维数组的第i个元素的地址	
a+i	同上	
a[i]	3元素一维数组的第i个元素的值 (即4元素一维数组的数组名 4元素一维数组的首元素的地址)	元素地址
*(a+i)	同上	

i:0-2(行)

§12. 指针进阶

7. 多维数组与指针(补充，极其重要!!!)

7.1. 二维数组的地址

★ 二维数组加两个下标的理解方法(下标法、指针法)

从第五章概念可知：
a[i][j]：第i行j列元素的值
&a[i][j]：第i行j列元素的地址

令x表示a[i]，则：
x[j]：第i行j列元素的值
&x[j]：第i行j列元素的地址

由一维数组的等价变换可得：
x[j]：第i行j列元素的值
&x[j]：第i行j列元素的地址
*(x+j)：第i行j列元素的值
x+j ：第i行j列元素的地址

所以，用a[i]替换回x，则可得：
a[i][j]:第i行j列元素的值
&a[i][j]:第i行j列元素的地址
*(a[i]+j):第i行j列元素的值
a[i]+j :第i行j列元素的地址

因为：对一维数组 a[i]⇔*(a+i)
所以：*(a[i]+j) ⇔ *(*(a+i)+j)(值)
a[i]+j ⇔ *(a+i)+j (元素地址)

a[i][j]:第i行j列元素的值
&a[i][j]:第i行j列元素的地址
*(a[i]+j):第i行j列元素的值
a[i]+j :第i行j列元素的地址
((a+i)+j):第i行j列元素的值
*(a+i)+j :第i行j列元素的地址

二维数组元素的值和元素的地址均有三种形式：
a[i][j] ⇔ *(a[i]+j) ⇔ *(*(a+i)+j) 值
&a[i][j] ⇔ a[i]+j ⇔ *(a+i)+j 元素地址

§12. 指针进阶

7. 多维数组与指针(补充，极其重要!!!)

7.1. 二维数组的地址

地址增量的变化规律

对一维数组a：

a+i 实际 a+i*sizeof(基类型)

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]

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)

例：a+1：2016　行地址

a[1]+2：2024　元素地址

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

§12. 指针进阶

7. 多维数组与指针(补充，极其重要!!!)

7.1. 二维数组的地址

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)	值
((a+i)+j)	：值(第i行j列)	

假设 int a[3][4] 存放在2000开始的48个字节中

2000
2016
2016
2016
2016
2024
2024
2024

假设 i=1
j=2

a+1是地址2016，*(a+1)取a+1的值，还是地址2016
a+1是行地址,*(a+1)取a+1的值,是元素地址

a[2]是地址2032，&a[2]取a[2]的地址，还是2032
a[2]是元素地址，&a[2]取a[2]的地址，是行地址

§12. 指针进阶

7. 多维数组与指针(补充，极其重要!!!)

7.1. 二维数组的地址

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]+0	：地址(第i行0列)	
*(a+i)+0	：地址(第i行0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	：地址(第i行j列)	
a[i][j]	：值(第i行j列)	值
*(a[i]+j)	：值(第i行j列)	
((a+i)+j)	：值(第i行j列)	

这两种情况虽然只看到一个下标，但要当做两个下标理解(i行0列的特殊表示)

&a[i]	：地址(第i行)
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)

由：&a[i]：行地址　　a[i]：元素地址

　　a+i　：行地址　　*(a+i)：元素地址

得：*行地址　　=> 元素地址（该行首元素）

如何证明?

&首元素地址 => 行地址　（必须首元素!!!）

如何证明?

进一步思考：

(1) &行地址　是什么？ &&行地址呢？

(2) *元素地址 是什么？ **元素地址呢？

§12. 指针进阶

7. 多维数组与指针(补充，极其重要!!!)

7.1. 二维数组的地址

```
#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

实际运行一次，观察结果并思考!!!

说明：
每组打印地址后，
再打印地址+1，
目的是区分行地址及元素地址

§12. 指针进阶

7. 多维数组与指针(补充，极其重要!!!)

7.1. 二维数组的地址

```
#include <iostream>
using namespace std;
int main()
{   int a[3][4];
    cout << sizeof(a)          << endl;
    cout << sizeof(a+1)        << endl;
    cout << sizeof(*(a+1))     << endl;
    cout << sizeof(*(a+1))     << endl;
    cout << sizeof(**(a+1))    << endl;
    cout << sizeof(a[2])       << endl;
    cout << sizeof(*(a[2]))    << endl;
    cout << sizeof(&a[2])      << endl;
    cout << sizeof(*(&a[2]))   << endl;
    return 0;
}
```

另一种验证方法!!!

48		数组大小
4	即&a[1],是地址(指针)	a+1大小
16	指针基类型是int[4]	a+1基类型
16	即a[1],是数组(4元素)	*(a+1)大小
4	数组元素是int	*(a+1)基类型
16	a[2]是数组(4元素)	a[2]大小
4	数组元素是int	a[2]基类型
4	数组a[2]的地址(指针)	&a[2]大小
16	指针基类型是int[4]	&a[2]基类型
	*&a[2]⇔a[2],是数组(4元素)	

行地址 / 元素地址 / 行地址

同

§12. 指针进阶

7. 多维数组与指针(补充，极其重要!!!)

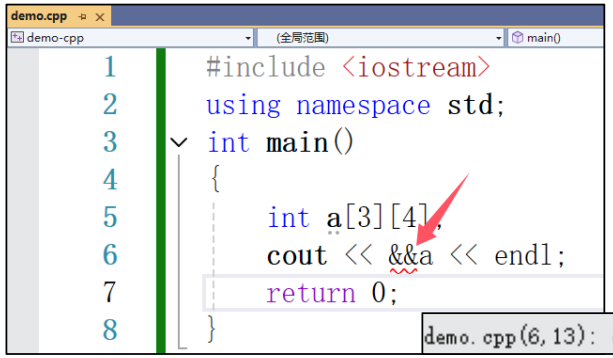
7.1. 二维数组的地址

```
#include <iostream>
using namespace std;
int main()
{
    int a[3][4];

    cout << a << endl;
    cout << a + 1 << endl;
    cout << endl;
    cout << &a << endl;
    cout << &a + 1 << endl;
    return 0;
}
```

```
Microsoft Visual Stud
00DDFD8C
00DDFD9C

00DDFD8C
00DDFDBC
```

地址a
地址a+16

地址a
地址a+48



进一步思考：

(1) &行地址　是什么？　&&行地址呢？

(2) *元素地址　是什么？　**元素地址呢？

拓展思维：多维数组
看不懂不要纠结，放弃！

```
#include <iostream>
using namespace std;
int main()
{
    int a[2][3][4];
    cout << &a[1][2] << endl;
    cout << &a[1][2] + 1 << endl;
    cout << endl;
    cout << &a[1] << endl;
    cout << &a[1] + 1 << endl;
    cout << endl;
    cout << &a << endl;
    cout << &a + 1 << endl;
    return 0;
}
```

```
Microsoft Visual Studio i
00B3FBE8
00B3FBF8

00B3FBC8
00B3FBF8

00B3FB98
00B3FBF8
```

地址a
地址a+16

地址a
地址a+48

地址a
地址a+96

§12. 指针进阶

7. 多维数组与指针(补充，极其重要!!!)

7.1. 二维数组的地址

7.2. 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

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

int main()
{
   int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
   int *p = a[0];
   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
   cout << p      << endl;       地址a        元素[0][0]地址
   cout << p+5    << endl;       地址a+20     元素[1][1]地址
   cout << *(p+5) << endl;       6           a[1][1]的值
}
```

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

假设a的首地址是2000，则区别如下：

p=a[0]：p的值是2000，基类型是int，p+1的值为2004

p=a　：p的值是2000，基类型是int*4，p+1的值为2016

因为p是基类型为int的指针变量，所以：

p+i ⇔ p+i*sizeof(int)

p+5 ⇔ &a[1][1]

§12. 指针进阶

7. 多维数组与指针(补充，极其重要!!!)

7.1. 二维数组的地址

7.2. 指向二维数组元素的指针变量

例：打印二维数组的值（以下四种方法均正确，均是按一维方式顺序循环）

```
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;
}
```

§12. 指针进阶

7. 多维数组与指针(补充，极其重要!!!)

7.3. 指向由m个元素组成的一维数组的指针变量

```
#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

int a[3][4]={…};

int (*p)[4]=a;

(*p)有4个元素

每个元素类型是int

=> p是指向4个元素组成的一维数组的指针

int a[4]

a有4个元素

每个元素类型是int

*p+j / *(p+0)+j:取这个一维数组中的第j个元素

p+i 实际 p+i*4*sizeof(int)

*(p+i)+j 实际 p+(i*4+j)*sizeof(int)

★ 使用：

p：地址(m个元素组成的一维数组的地址)

*p：值(是一维数组的名称，即一维数组的首元素地址)

§12. 指针进阶

7. 多维数组与指针(补充，极其重要!!!)

7.3. 指向由m个元素组成的一维数组的指针变量

int a[3][4]={1,…,12}, (*p)[4] ;

p = a;

p+1 : 行地址2016(a[1])

*p+1 : 元素地址2004(a[0][1])

*(*p+1) : 元素值2(a[0][1])

*(p+1)+2 : 元素地址2024(a[1][2])

((p+1)+2): 元素值7(a[1][2])

p是行地址2000

*p是元素地址2000

p+1是行地址2016

*(p+1)是元素地址2016

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

```
#include <iostream>
using namespace std;
int main()
{  int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
   int (*p)[4] = a;
   cout << sizeof(a)  << endl;
   cout << sizeof(p)  << endl;
   cout << sizeof(*p) << endl;
   cout << p       << endl;
   cout << p+1     << endl;
   cout << *p      << endl;
   cout << *p+1    << endl;
   cout << *(*p+1) << endl;
}
```

```
48          数组大小
4           因为指针
16          因为int[4]
地址a       行地址
地址a+16    +1 = +16
地址a       元素地址
地址a+4     +1 = +4
2           a[0][1]的值
```

```
#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
        }
}
```

§12. 指针进阶

7. 多维数组与指针(补充，极其重要!!!)

7.4. 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;

    return 0;
}
```

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

§12. 指针进阶

7. 多维数组与指针(补充，极其重要!!!)

7.5. 用指向二维数组的指针做函数参数

思考：若f1/f2/f3中为sizeof(**x1/**x2/**x3)
则：结果是多少？为什么？

5.4. 用数组名作函数参数

5.4.3. 用多维数组名做函数实参

当时第5章的说法，都不准确，形参数组不存在
形参的本质是指针变量

★ 形参为相应类型的多维数组

★ 实、形参数组的列必须相等，形参的行可以不指定，或为任意值(实参传入二维数组的首地址，只要知道每行多少列实形参即可对应，不关心行数)

```
#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) << endl;
    f1(a);
    f2(a);
    f3(a);
}
```

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

```
#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) << 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]
```

§12. 指针进阶

7. 多维数组与指针(补充，极其重要!!!)

7.5. 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

★ 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)

例：二维数组名做实参

```
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;
}

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

int p[3][4]
int p[][4]
int p[123][4]
本质都是行指针变量

((p+i)+j)
*(p[i]+j)
p[i][j]
二维数组值的三种形式

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

3. 一维数组与指针中

★ 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)

本处：

★ 对二维数组而言，数组的指针是指向一维数组的指针，数组元素的指针是指向单个元素的指针，两者的本质是完全不同的(基类型不同)

§12. 指针进阶

7. 多维数组与指针(补充，极其重要!!!)

7.5. 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

★ 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)

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])
{
   ...
}
```

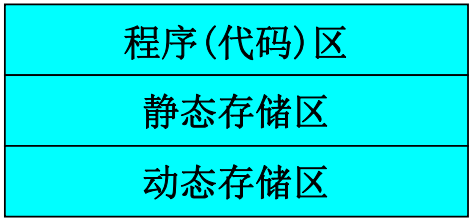
```
//实参是二维数组名
int main()
{   int a[3][4]={...};

    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);
}
```

§12. 指针进阶

8. 函数与指针

8.1. 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

8.2. 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);
```

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)

§12. 指针进阶

8. 函数与指针

8.1. 函数的地址

8.2. 用函数指针变量调用函数

```
//例：简单的函数调用
#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;
}
```

```
//例：简单的函数调用
#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);//定义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

§12. 指针进阶

8. 函数与指针

8.1. 函数的地址

8.2. 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;
}
```

函数名/函数指针
1、带()不带()
2、加*不加*
的含义区别

?

§12. 指针进阶

8. 函数与指针

8.1. 函数的地址

8.2. 用函数指针变量调用函数

★ 指向函数的指针的型参表声明时，与被调用函数的型参表类型、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) {   ...   }
main()
{
    int (*p)(int, int); //不带形参变量名
    p=max;
}
```

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

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

§12. 指针进阶

8. 函数与指针

8.1. 函数的地址

8.2. 用函数指针变量调用函数

★ 指向函数的指针的型参表声明时，与被调用函数的型参表类型、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

★ 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;
}
```

```
#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;
   return 0;
}
```

编译出错，因为p/q指向的两个函数的形参表列及返回值不完全相同

§12. 指针进阶

8. 函数与指针
8.1. 函数的地址
8.2. 用函数指针变量调用函数
8.3. 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
★ 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)

```
void f1(int x, int y)
{
    cout << x+y << endl;
}
void f2(int x, int y)
{
    cout << x-y << endl;
}
void f3(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(f1);    //25
   fun(f2);    //-5
   fun(f3);    //150
   return 0;
}
```

§12. 指针进阶

8. 函数与指针
8.1. 函数的地址
8.2. 用函数指针变量调用函数
8.3. 指向函数的指针做函数参数
8.4. 指向类对象的成员函数的指针

§7. 结构体、类和对象
7.6. 对象指针
7.6.1. 指向对象的指针
7.6.2. 指向对象成员的指针
7.6.2.1. 指向对象的数据成员的指针
7.6.2.2. 指向对象的成员函数的指针(后续课程内容，略)

§12. 指针进阶

8. 函数与指针
8.1. 函数的地址
8.2. 用函数指针变量调用函数
8.3. 指向函数的指针做函数参数
8.4. 指向类对象的成员函数的指针

```
/* 8.3.指向普通函数的指针 */
#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
}
```

全局函数的指针：
(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() {    //构造
       hour=0;
       }
    void display() { //打印
       cout << hour << endl;
       }
};
int main()
{  Time t1;
   void (*p)();
   p=t1.display;//赋值,错误
   p();         //调用,错误
}
```

```
#include <iostream>
using namespace std;
class Time {
  private:
    int hour;
  public:
    Time() {    //构造
       hour=0;
       }
    void display() { //打印
       cout << hour << endl;
       }
};
int main()
{  Time t1;
   void (Time::*p)();
   p=&Time::display;//赋值,正确
   (t1.*p)();       //调用,正确
}
```

成员函数的指针：
(1) 返回类型匹配
(2) 形参表匹配
(3) 类匹配

§12. 指针进阶

8. 函数与指针

8.1. 函数的地址

8.2. 用函数指针变量调用函数

8.3. 指向函数的指针做函数参数

8.4. 指向类对象的成员函数的指针

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

★ 对象的成员函数必须是public

使用：

(对象名.*指针变量名)(实参表)

```
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
```

§12. 指针进阶

9. 指针数组和指向指针的指针

9.1. 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];

是一个数组

数组的元素类型是指针

指针的基类型是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是数组名，有4个元素，每个元素是int *

p+1实际+4，因为数组类型为指针

int (*p)[4];

p是指针变量名，指向由4个元素组成的一维数组

p+1实际+16，因为p的基类型为int*4

使用：保证数组的每个元素为基类型为数据类型的指针，使用时匹配即可，可进行所允许的任何运算

```
#include <iostream>
using namespace std;
int main()
{
    int a=10,b[3]={11,12,13},c=27, *d=&c;
    int *p[4] = {&a, b, &b[2], d};
    cout << *p[0] << endl;
    *(p[1] + 1) = 32;
    cout << b[1] << endl;
    cout << p[2] - b << endl;
    cout << (*p[3] - *p[0]) << endl;

    return 0;
}
```

?

§12. 指针进阶

9. 指针数组和指向指针的指针

9.1. 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	

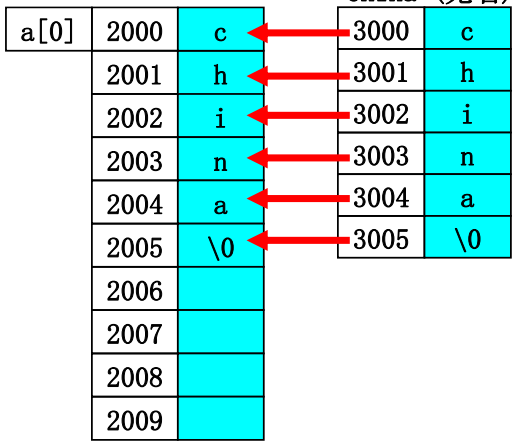
优点：(1) 与无名字符串常量分占不同空间

(2) 字符串的值可以修改

缺点：(1) 有空间浪费

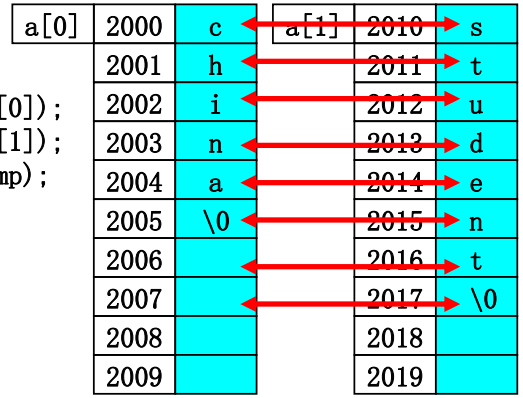
(2) 若要交换元素(例如排序)，则需要要整体移动元素

赋初值方法



交换的方法：

```
char tmp[10];
strcpy(tmp, a[0]);
strcpy(a[0],a[1]);
strcpy(a[1],tmp);
```



§12. 指针进阶

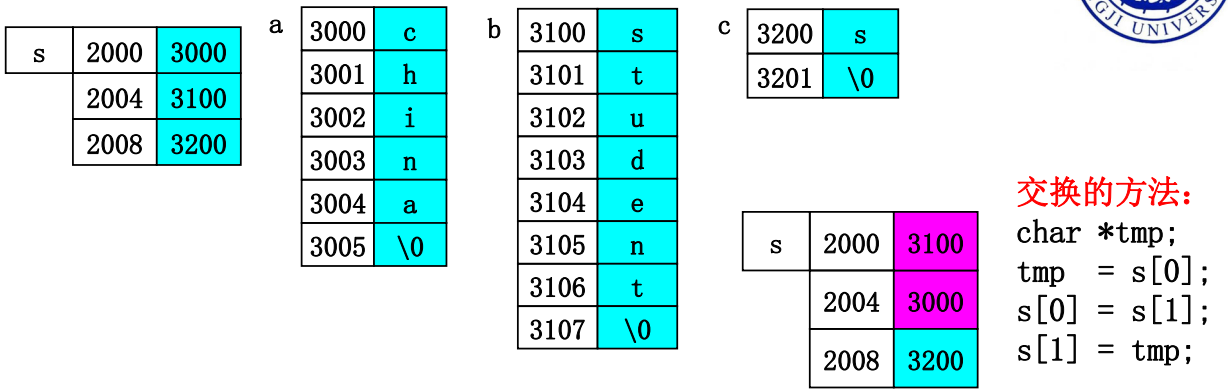
9. 指针数组和指向指针的指针

9.1. 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[]="china", b[]="student", c[]="s";
char *s[3] = {a, b, c};
```

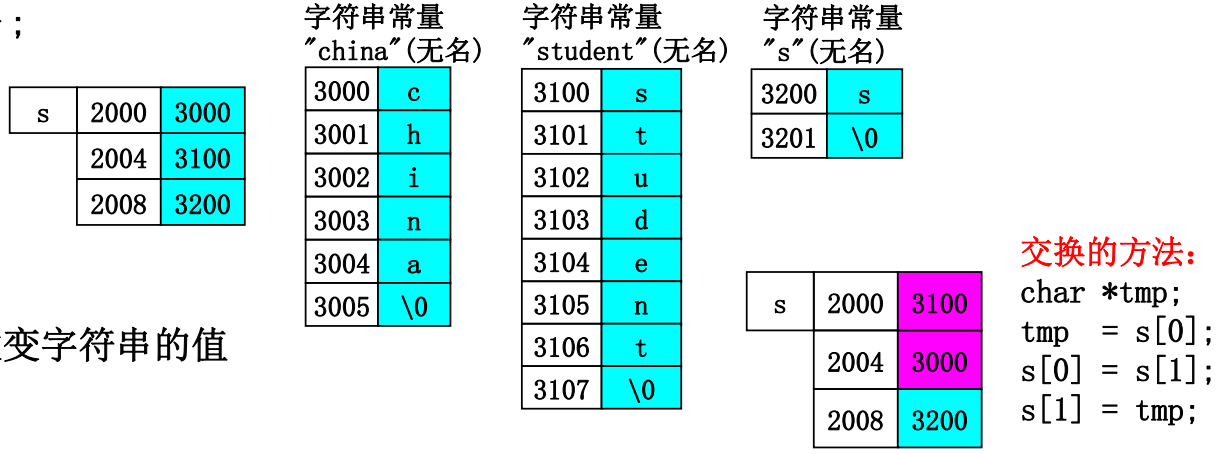


交换的方法：

```
char *tmp;
tmp  = s[0];
s[0] = s[1];
s[1] = tmp;
```

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

注：因为用字符串常量初始化，要加const



交换的方法：

```
char *tmp;
tmp  = s[0];
s[0] = s[1];
s[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) 如果指针指向无名字符串常量，则无法改变字符串的值

§12. 指针进阶

9. 指针数组和指向指针的指针

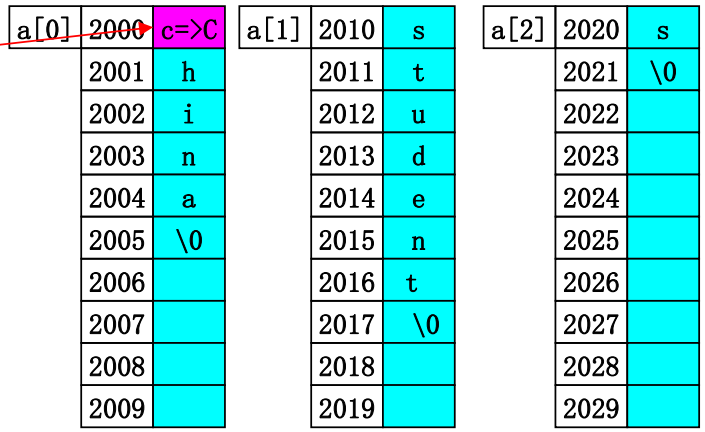
9.1. 指针数组

★ 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的字符串存储空间，只是指向字符串首址，在执行过程中字符串值能否改变不确定

（建议用于不改变的场景）

```
#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;
}
```



§12. 指针进阶

9. 指针数组和指向指针的指针

9.1. 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的字符串存储空间，只是指向字符串首址，在执行过程中字符串值能否改变不确定

（建议用于不改变的场景）

```
#include <iostream>
using namespace std;
int main()
{   const char *a[3]= {"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;
    cout << a[1] << endl;
    cout << a[2] << endl;
    return 0;
}
```

VS编译：error
Dev编译：error

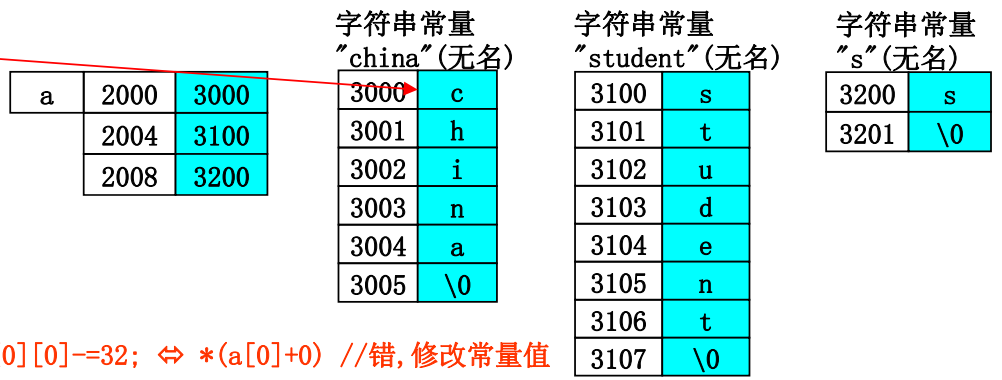
VS：

demo.cpp(9,11): error C3892: “a”: 不能给常量赋值

Dev：

In function 'int main()':

[Error] assignment of read-only location '* a[0]'



a[0][0]-=32; ⇔ *(a[0]+0) //错,修改常量值

§12. 指针

9. 指针数组和指向指针的指针

9.1. 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
- 一维指针数组不分配实际的字符串存储空间，只是指向字符串

不确定

```
int main()
{
    char *a[3]= {"china","student","s"};//无const
    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
    a[0][0]-=32;        //修改数组b[0]元素的值
    cout << a[0] << endl;       Hello
    cout << a[1] << endl;       student
    cout << a[2] << endl;       s
    return 0;
}
```

VS编译：error
Dev编译：warning

VS：

```
error C2440: “初始化”: 无法从“const char [6]”转换为“char *”
message : 从字符串文本转换将丢失 const 限定符(请参阅 /Zc:strictStrings)
error C2440: “初始化”: 无法从“const char [8]”转换为“char *”
message : 从字符串文本转换将丢失 const 限定符(请参阅 /Zc:strictStrings)
error C2440: “初始化”: 无法从“const char [2]”转换为“char *”
message : 从字符串文本转换将丢失 const 限定符(请参阅 /Zc:strictStrings)
```

Dev：

```
[Warning] ISO C++ forbids converting a string constant to 'char*' [-Wwrite-strings]
[Warning] ISO C++ forbids converting a string constant to 'char*' [-Wwrite-strings]
[Warning] ISO C++ forbids converting a string constant to 'char*' [-Wwrite-strings]
```

（建议用于不改变的场景）

a	2000	3000->3300
	2004	3100
	2008	3200

```
china
student
s
hello
student
s
Hello
student
s

Process exited after 0.0557 seconds with return value 0
请按任意键继续. . .
```

Dev有warning但可运行，且表面现象正常，实际呢？

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

数组b[10]

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

§12. 指针进阶

9. 指针数组和指向指针的指针

9.1. 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的字符串存储空间，只是指向字符串首址，在执行过程中字符串值能否改变不确定

思考题

```
int main()
{   char  a[3][10] = {"china","student","s"}; //二维数组
    char  b[10]="hello";
    cout << a[0] << endl;
    cout << a[1] << endl;
    cout << a[2] << endl;
    a[0] = b;
    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;
}
```

编译
- 正确，执行
  - 正确，运行结果？
  - 错误，哪句运行出错，为什么？
- 错误，哪句错？为什么？

如何修改，使正确运行并且运行结果与上例相同？

§12. 指针进阶

9. 指针数组和指向指针的指针

9.1. 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的字符串存储空间，只是指向字符串首址，在执行过程中字符串值能否改变不确定

```
//例：字符串进行选择排序（指针数组形式）
#define N 4
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;
            }
        }
}
int main()
{   int i;
    const char *name[N] = { "PHP", "C++", "Python","Java"};
    for (i=0; i<N; i++)
        cout << name[i] << endl;
    sort(name, N);
    for (i=0; i<N; i++)
        cout << name[i] << endl;
}
```

对比并体会红蓝部分差异

```
Microsoft Visual Studio 调试控制台
PHP
C++
Python
Java
C++
Java
PHP
Python
```

思考：
左右两侧name中，字符串的长度有限制吗？

```
//例：字符串进行选择排序（二维字符数组形式）
#define N 4
#define LEN 8
void sort(char name[][LEN], int n)
{
    char temp[LEN];
    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);
            }
        }
}
int main()
{   int i;
    char name[N][LEN] = {"PHP", "C++", "Python", "Java"};
    for (i=0; i<N; i++)
        cout << name[i] << endl;
    sort(name, N);
    for (i=0; i<N; i++)
        cout << name[i] << endl;
}
```

§12. 指针进阶

9. 指针数组和指向指针的指针

9.1. 指针数组

9.2. 指向指针的指针

定义：数据类型 **指针变量名

int **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;

t	2000	2100 2103
i	10	2000 2001
p	2100	2200 2203

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）

```
#include <iostream>
using namespace std;
int main()
{
    const char *a[3] = {"china", "student", "s"}, **p;
    p=a;
    cout << p << endl;
    cout << p+1 << endl;
    cout << (int *)(*p) <<endl;
    cout << *p << endl;
    cout << *p+3 << endl;
    cout << *(*p+3) << endl;
}
```

```
地址a
地址a+4
地址b        串首地址
china        输出串
na           输出串
n            输出字符
```

1、p+1只加了4，证明p不是china的地址(不够)

2、*p的值(地址b)，是无名字符串常量"china"的首址

3、由2反证1中的地址a应该是数组元素a[0]的地址

§12. 指针进阶

9. 指针数组和指向指针的指针

9.2. 指向指针的指针

```
char *a[3] = {"china", "student", "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

§12. 指针进阶

10. const指针

10.1. 共用数据的保护

一个数据可以通过不同的方式进行共享访问，因此可能导致数据因为误操作而改变，为了达到既能共享，又不会因误操作而改变，引入共用数据保护的概念

``#include <iostream>``` ```using namespace std;``` ```void fun(int *p)``` ```{``` ``` *p=10;``` ```}``` ```int main()``` ```{``` ``` int k=15;``` ``` fun(&k);``` ```}```	```#include <iostream>``` ```using namespace std;``` ```void fun(int *p)``` ```{``` ``` if (*(p+5)=10) //原意是 *(p+5)==10``` ```}``` ```int main()``` ```{``` ``` int a[]={…};``` ``` fun(a);``` ```}```
通过指针，fun中改变了main的局部变量k的值，如果这种改变不是预期中的，则可能会带来错误	通过指针，fun中改变了main的数组a中元素的值（手误导致非预期结果）

§12. 指针进阶

10. const指针

10.2. 指向常量的指针变量

形式：　　const 数据类型 *指针变量名

　　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)

★ 适用于不希望通过指针修改变量值的情况

```
#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;
    a = 10;
    cout << *p << endl;
    p = &b;
    cout << *p << endl;
    return 0;
}
```

a并不是常量，
但无法通过p改变a的值
=> 理解为p指向一个常量

1、哪一个语句会编译报错?
2、注释掉报错语句后，
三句cout的输出是什么？

问：假设 a=10; 也不允许，
如何操作？
答：

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

void fun(const int *x)
{
    x++    / x+=2 / *x=10   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);
    return 0;
}
```

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）

假设main由甲完成，fun由乙完成，数组a
为传递的参数，甲希望乙只能读a而不能
修改a中的值

§12. 指针进阶

10. const指针

10.2. 指向常量的指针变量

形式：　　const 数据类型 *指针变量名

　　或　　数据类型 const *指针变量名

作用：

★ 指向常量的指针变量可以指向常变量、普通变量，但是普通指针不能指向常变量

```
error C2440: “=”: 无法从“const int *”转换为“int *”
message : 转换丢失限定符
```

```
int a;
const int b;
const int *p;
p = &a; //正确
p = &b; //正确
```

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

=> 推论：应用于函数的实形参对应，则规律如下：

形参	属性	实参	属性	正确性
普通指针	RW	普通变量地址/指针	RW	对
普通指针	RW	常变量地址/指针	R	错
常变量指针	R	普通变量地址/指针	RW	对
常变量指针	R	常变量地址/指针	R	对

基本原则：赋权不能大于原权!!!

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

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

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

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

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

§12. 指针进阶

10. const指针

10.3. 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

```
#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;
    cout << *p << endl;
    return 0;
}
```

1、哪一个语句会编译报错?
2、注释掉报错语句后，
三句cout的输出是什么？

```
#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数组而不能被改变
(防止误操作)

假设main由甲完成，fun由乙完成，指针p为传递的参数，
甲希望乙可以通过p读写，但不要改变p的指向

§12. 指针进阶

10. const指针

10.3. 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

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

?

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

?

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

?

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

?

★ 常指针不能指向常变量（右侧两个例子均编译错）

```
const int x = 10;
int *const p = &x;
```

error C2440: “初始化”：无法从“const int *”转换为“int *”
message : 转换丢失限定符

问：如何解决?
答：见10.4

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

error C2664: “void f(int *const)”：无法将参数 1 从“const int *”转换为“int *const ”
message : 转换丢失限定符
message : 参见“f”的声明

§12. 指针进阶

10. const指针

10.2. 指向常量的指针变量

10.3. 常指针

10.4. 指向常量的常指针(同时满足10.2+10.3)

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

```
#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;
    a = 10;
    cout << *p << endl;
    p = &b;
    cout << *p << endl;
    return 0;
}
```

1、哪两个语句会编译报错?
2、注释掉报错语句后，
三句cout的输出是什么？

```
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);
}
```
?

§12. 指针进阶

10. const指针

10.1. 共用数据的保护

10.2. 指向常量的指针变量

形式：　　const 数据类型 *指针变量名

　　或　　数据类型 const *指针变量名

10.3. 常指针

形式：数据类型 *const 指针变量名

10.4. 指向常量的常指针(同时满足10.2+10.3)

形式：const 数据类型 *const 指针变量名

问：在引入const指针的情况下，实形参之间参数传递的基本规则?

答：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

实参读写 => 形参只读

实参读写 => 形参读写

哪种有错?

§12. 指针进阶

11. void指针类型

2. 变量与指针
2.4. 指针变量的++/--
★ 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，不知道该给k分配几字节的空间
void *p; ✓ 正确，因为知道p大小是4字节
p++; × 错误，因为不知道基类型的大小
p--; × 错误，同上

3. 一维数组与指针
3.4. 指针法引用数组元素
3.4.3. 指向数组的指针变量的运算
C. 两个基类型相同的指针变量相减后与整数做比较运算
★ void型的指针变量不能进行相互运算(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
```

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
    return 0;
}
```

```
(10,13): error C2100: 非法的间接寻址
(10,10): error C2088: “<<”: 对于 class 非法
(12,12): error C2440: “=”: 无法从“void *”转换为“int *”
(12,10): message : 从“void*”到指向非“void”的指针的强制转换要求显式类型强制转换
```

§12. 指针进阶

12. 有关指针的数据类型和指针运算的小结

12.1. 各种类型的指针变量

```
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
```

```
int *(*p)()    :

int *(*p)[n]   :
int (*p[n])()  :
int *(*p[n])():
```

思考：
1、指出这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？

(下面的示例全部正确)

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

```
int main ()
{  int *a[10],*b[3][10];
   int *(*p)[10];
   p = &a; //特别关注!!!
   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;
}
```

§12. 指针进阶

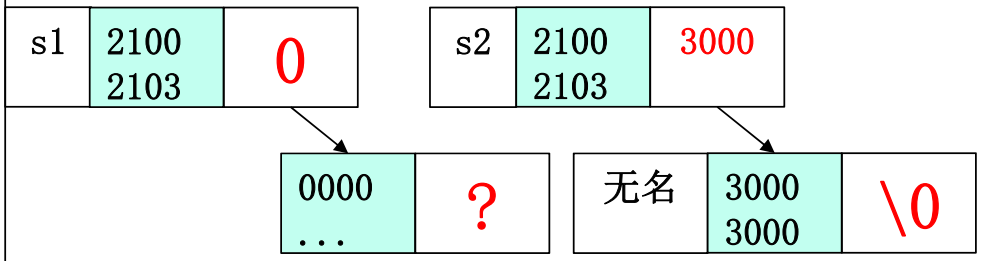
12. 有关指针的数据类型和指针运算的小结

12.2. 空指针NULL(复习)

★ 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)

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
}
```

```
error C2440: "初始化": 无法从"int"转换为"char []"
message : 没有转换为数组类型，但有转换为数组的引用或指针
```

§12. 指针进阶

11. 有关指针的数据类型和指针运算的小结

11.2. 空指针NULL(复习)

★ 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，称野指针)

★ 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)

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

```
#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[80]="Hello";
    strcat(s2, s1);
}
```
错

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

=> 自行实现类似功能的字符串处理函数时，可以对NULL进行特殊处理(具体见作业，这不是标准，只是为了强行与系统函数不同)

- 求长度时为0
- 复制、连接、拷贝时当做空串进行处理

§12. 指针进阶

12. 有关指针的数据类型和指针运算的小结

12.1. 各种类型的指针变量

12.2. 空指针NULL(复习)

12.3. 不同基类型指针的相互赋值(习题课，复习)

★ 不同类型的指针变量不能相互赋值，若要赋值，则需要进行强制类型转换

- char s[]="abcd"; int *p=(int *)s; cout << hex << *p;
  为什么输出 64636261
- int a=0x414243; char *s=(char *)&a; cout << s;
  为什么输出 CBA
- 为什么 float d=1.23e4; 无截断警告
  flaot d=1.23; 有截断警告

§12. 指针进阶

13. 引用(C++新增)

13.1. 引用的基本概念(习题课，复习)

13.2. 简单变量的引用作函数参数(习题课，复习)

§12. 指针进阶

13. 引用(C++新增)

13.3. 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;   ???
    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，能否取得实参数组的大小？

§12. 指针进阶

13. 引用(C++新增)

13.3. 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++)
        ;

    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;
}
```

依次赋值，含尾零

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

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

char *tj_strcpy_s(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;
}

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

可以做到安全copy，不会越界

问题：
1、能否和原tj_strcpy同名(重载)
2、可以实用吗？

§12. 指针进阶

13. 引用(C++新增)

13.3. 数组引用做函数参数

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

char *tj_strcpy_s(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 str1[10], str2[]="This is a pencil.";
    tj_strcpy_s(str1, str2); //正确
    cout << str1 << endl;

    char str3[20], str4[]="This is a pen.";
    tj_strcpy_s(str3, str4); //错误
    cout << str3 << endl;
    return 0;
}
```

s1：形参是实参数组的引用，实形参数组的大小必须一致，无法适应不同长度的实参数组

s2：形参是指针，初值指向实参数组的首元素，没有大小，因此能适应不同长度的实参数组

问题：

2、可以实用吗？

tj_strcpy_s()尚不能投入实用!!!

如何解决？使数组的引用声明时，长度可变

具体方法自行查阅资料（作业）

- 形参是一维数组的引用，则形参声明的大小必须与实参数组的定义大小一致

  => 推论：形参是二维数组的引用，则形参声明与实参定义的行列大小必须完全一致

§12. 指针进阶

13. 引用(C++新增)

13.3. 数组引用做函数参数

```
#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);
}
```

哪句编译报错？

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

void test2(char (&s1)[10])
{
    cout << sizeof(s1) << endl;
}
void test1(char s1[])
{
    cout << sizeof(s1) << endl;
    test2(s1);
}
int main()
{
    char s[10];
    test1(s);
}
```

哪句编译报错？

- 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

  => 推论：数组的引用在函数间传递时，必须始终保持数组引用方式

§12. 指针进阶

13. 引用(C++新增)

13.4. 关于引用的特别说明

★ 引用在需要改变实参值的函数调用时比指针方式更容易理解，形式也更简洁，不容易出错

★ 引用不能完全替代指针

(可以将指针理解为if-else，引用理解为switch-case)

★ 引用是C++新增的，纯C的编译器不支持，后续工作学习中接触的大量底层代码仍是由C编写的，此时无法使用引用

(VS和Dev都是C++编译器，兼容纯C，用后缀名区分如何编译)

(Linux下gcc/c++区分的较清楚)

★ 对于码类专业而言，仍需要透彻理解指针