§12. 指针进阶

2^10(KB)	KiloByte
20(MB)	MegaByte
30(GB)	GigaByte
40(TB)	TeraByte
50(PB)	PeraByte
60(EB)	ExaByte
70(ZB)	ZetaByte
80(YB)	YottaByte
90(NB)	NonaByte
100(DB)	DoggaByte

1. 基本概念(复习)

★ 数据在内存中的存放
- ● 根据不同的类型存放在动态/静态数据区
- ● 数据所占内存大小由变量类型决定 sizeof(类型)

★ 内存地址
- ● 给内存中每一个字节的编号
- ● 内存地址的表示根据内存地址的大小一般分为16位、32位和64位

（一般称为地址总线的宽度，是CPU的理论最大寻址范围,具体还受限于其它软、硬件）

16位：0 ~ $2^{16}-1$ (64KB)

32位：0 ~ $2^{32}-1$ (4GB)

64位：0 ~ $2^{64}-1$ (16EB)

例如：32位地址总线
4G内存
则：内存地址表示为
0x00000000
|
0xFFFFFFFF

说明：到目前为止，John von Neumann型计算机的地址都表示为一维线性结构

★ 内存中的内容

以字节为单位，用一个或几个字节来表示某个数据的值

（基本数据类型一般都是2的n次方）

★ 内存中内容的访问

存放地址

直接访问：按变量的地址取变量值

间接访问：通过某个变量取另一个变量的地址，再取另一变量的值

存放值

★ 指针变量

存放地址的变量，称为指针变量

★ 指针

某一变量的地址，称为指向该变量的指针 （地址 = 指针）

§12. 指针进阶

2. 变量与指针(复习)

2.1. 定义指针变量

数据类型 *变量名：表示该变量为指针变量，指向某一数据类型

★ 数据类型称为该指针变量的基类型

★ 变量中存放的是指向该数据类型的地址

int *p：p是指针变量(注意，不是*p)
存放一个int型数据的地址
p的基类型是int型

★ 指针变量所占的空间与基类型无关，与系统的地址总线的宽度有关

16位地址：一个指针变量占16位（2字节）

32位地址：一个指针变量占32位（4字节）

64位地址：一个指针变量占64位（8字节）

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

int main()
{
    cout << sizeof(char)   << endl;   1
    cout << sizeof(short)  << endl;   2
    cout << sizeof(int)    << endl;   4
    cout << sizeof(long)   << endl;   4
    cout << sizeof(float)  << endl;   4
    cout << sizeof(double) << endl;   8
    return 0;
}
```

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

int main()
{
    cout << sizeof(char *)   << endl;  4 8
    cout << sizeof(short *)  << endl;  4 8
    cout << sizeof(int *)    << endl;  4 8
    cout << sizeof(long *)   << endl;  4 8
    cout << sizeof(float *)  << endl;  4 8
    cout << sizeof(double *) << endl;  4 8
    return 0;
}
```

Debug x86

Debug x64

```
#include <iostream>
using namespace std;
int main()
{   char   c, *p;
    double d, *q;
    p = &c;
    q = &d;
    cout << sizeof(p)   << endl;   4
    cout << sizeof(*p)  << endl;   1
    cout << sizeof(q)   << endl;   4
    cout << sizeof(*q)  << endl;   8
    return 0;
}
```

★ 基类型的作用是指定通过该指针变量间接访问的变量的类型及占用的内存大小

§12. 指针进阶

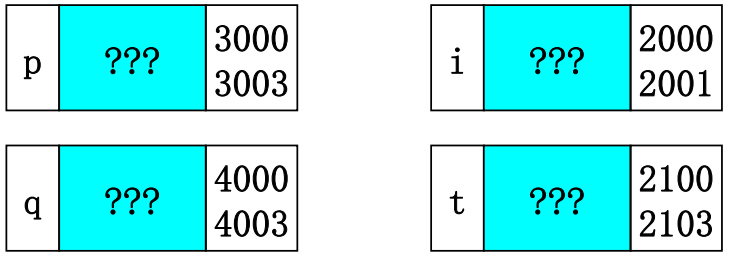
2. 变量与指针(复习)

2.2. 使用

变量名 = 地址

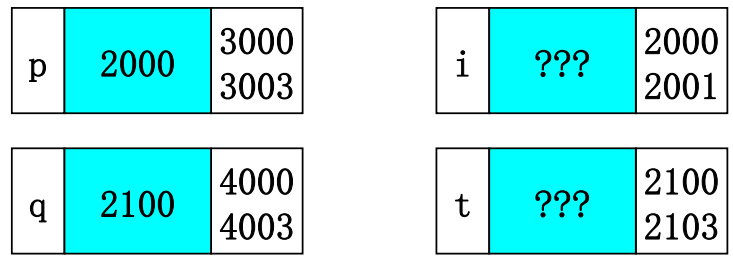
*变量名 = 值

short i, *p;　　long t, *q;



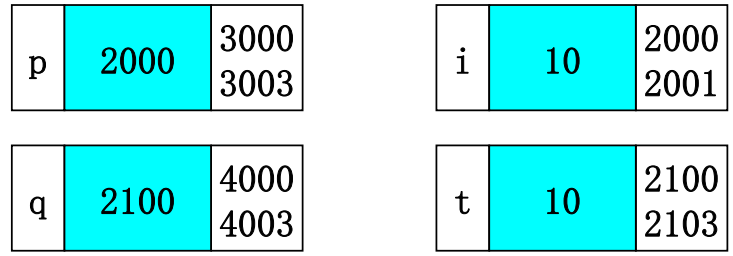
i占2字节是因为_short型_。
t占4字节是因为_long型___。
p/q占4字节是因为_指针类型_。
★ 假设是32位地址系统

p=&i;　　q=&t;



间接访问：
不是将p/q自身空间赋值为10，
而是将p/q中存放的值(&i/&t)
所对应的空间(i/t)赋值为10

*p=10 ⇔ i=10　　*q=10 ⇔ t=10



★ 假设32位地址系统，则p，q均占用4字节
★ 假设p，q中存放的地址为2000/2100，则
*p=10：表示将2000-2001的2个字节赋值为10
*q=10：表示将2100-2103的4个字节赋值为10

问题：p/q中只存放了变量的首地址，如何知道变量所占字节的长度？
★ 基类型的作用是指定通过该指针变量间接访问的变量的类型及占用的内存大小

§12. 指针进阶

2. 变量与指针(复习)

2.3. &与*的使用

★ &表示取变量的地址，*表示取指针变量的值

★ 两者优先级相同，右结合

```
int i=5, *p=&i;
```

&*p ⇔ &i ⇔ p

*&i ⇔ i

p	2000	3000 3003

i	5	2000 2003

变量定义时赋初值

```
int i=5, *p=&i;
```

用赋值语句赋值

```
int i=5, *p;
p=&i;
```

§12. 指针进阶

2. 变量与指针(复习)

2.4. 指针变量的++/--

★ 指针变量的++/--单位是该指针变量的基类型【指针变量++ ⇔ 所指地址+=sizeof(基类型)】

定义	赋值为10	++运算后地址(假设初始地址均为2000)
char *p1;	*p1=10：2000赋值为10	p1++：p1为2001
short *p2;	*p2=10：2000-2001赋值为10	p2++：p2为2002
long *p3;	*p3=10：2000-2003赋值为10	p3++：p3为2004
float *p4;	*p4=10：2000-2003赋值为10	p4++：p4为2004
double *p5;	*p5=10：2000-2007赋值为10	p5++：p5为2008

```
#include <iostream>
using namespace std;
int main()
{
    short  s, *p2 = &s;
    double d, *p5 = &d;

    cout << p2    << endl;   假设地址A
    cout << ++p2  << endl;   =地址A+2   2字节
    cout << p5    << endl;   假设地址B
    cout << ++p5  << endl;   =地址B+8   8字节

    return 0;
}
```

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    short  s, *p2 = &s;
    char   d, *p5 = &d;
    cout << p2    << endl;
    cout << ++p2  << endl;
    cout << hex << (int *)(p5)   << endl;
    cout << hex << (int *)(++p5) << endl;

    return 0;
}
```

问题：直接用
cout << p5 << endl;
cout << ++p5 << endl;
为什么会输出一串乱字符？
为什么输出char型的地址要转为int型？

假设地址A
=地址A+2
假设地址B
=地址B+1

§12. 指针进阶

2. 变量与指针(复习)

2.4. 指针变量的++/--

★ 指针变量的++/--单位是该指针变量的基类型

★ 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，因为不知道该给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
```

★ *与++/--的优先级关系

*比后缀++/--优先级低　　*:3 后缀:2

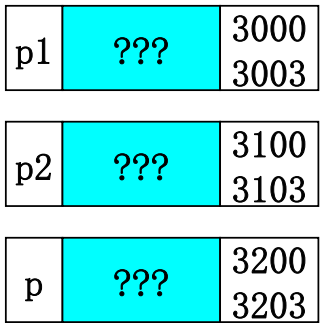
*与前缀++/--优先级相同，右结合　*:3 前缀:3

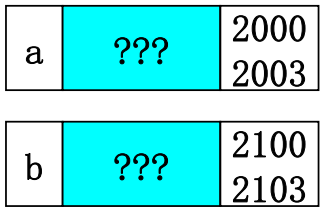
```
int i, *p=&i;
*p++ ⇔ *(p++)：保留p的旧值到临时变量中，p再++(不指向i)，最后取旧值所指的值(i)
*++p ⇔ *(++p)：p先++(不指向i)，再取p的值(非i)
(*p)++ ⇔ i++ ：取p所指的值(i)，i值再后缀++
++*p   ⇔ ++i ：取p所指的值(i)，i值再前缀++
```

*p++的另一种解释：先取p所指的值(i),p再++(不指向i)
=> 虽然能解释最终结果，但无法解释后缀++优先级高于*，
为什么不先做++

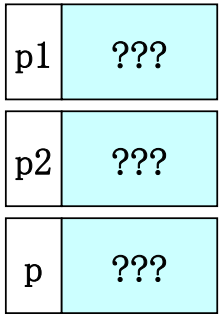
例：键盘输入两数，按从大到小的顺序依次输出（指针法处理）

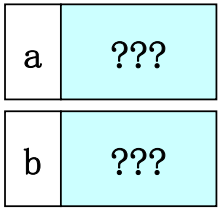
```
int main()
{   int *p1, *p2, *p, a, b;
    cin >> a >> b;  (假设键盘输入是45 78)
    p1=&a;
    p2=&b;
    if (a<b) {
       p=p1;
       p1=p2;
       p2=p;
       }
    cout << *p1 << ' ' << *p2 << endl;
}
```





带地址的图示法

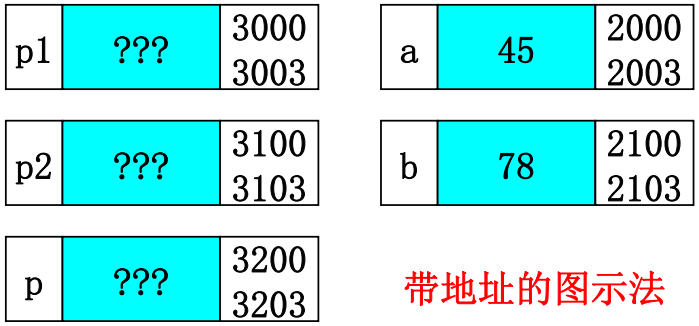




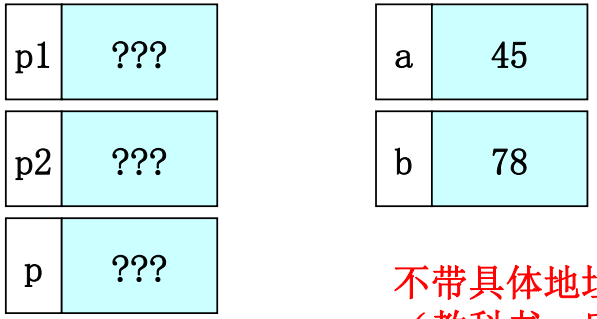
不带具体地址的图示法
（教科书、后续课程）

例：键盘输入两数，按从大到小的顺序依次输出（指针法处理）

```
int main()
{   int *p1, *p2, *p, a, b;
    cin >> a >> b;  (假设键盘输入是45 78)
    p1=&a;
    p2=&b;
    if (a<b) {
       p=p1;
       p1=p2;
       p2=p;
       }
    cout << *p1 << ' ' << *p2 << endl;
}
```



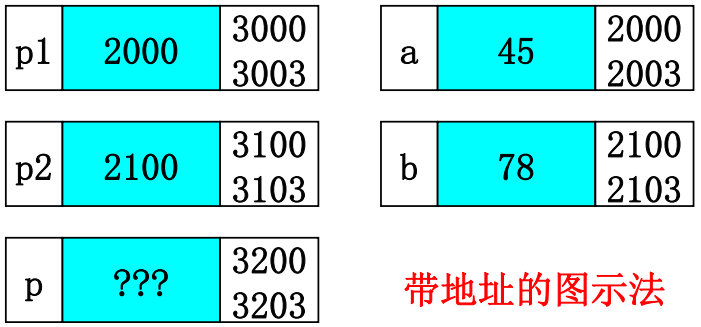
带地址的图示法



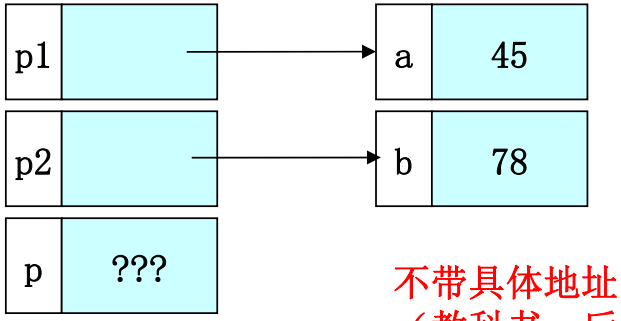
不带具体地址的图示法
（教科书、后续课程）

例：键盘输入两数，按从大到小的顺序依次输出（指针法处理）

```
int main()
{   int *p1, *p2, *p, a, b;
    cin >> a >> b;  (假设键盘输入是45 78)
    p1=&a;
    p2=&b;
    if (a<b) {
       p=p1;
       p1=p2;
       p2=p;
       }
    cout << *p1 << ' ' << *p2 << endl;
}
```



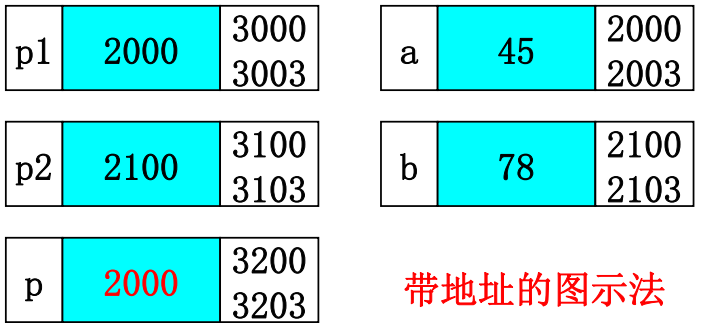
带地址的图示法



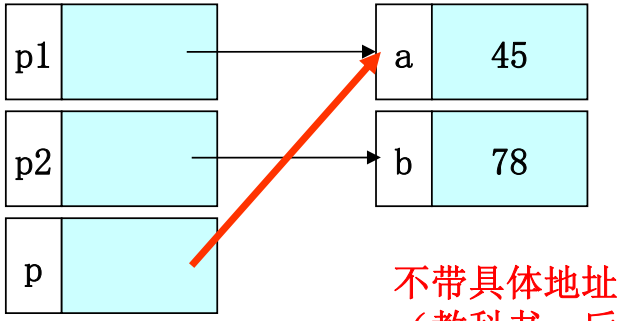
不带具体地址的图示法
（教科书、后续课程）

例：键盘输入两数，按从大到小的顺序依次输出（指针法处理）

```
int main()
{   int *p1, *p2, *p, a, b;
    cin >> a >> b;  (假设键盘输入是45 78)
    p1=&a;
    p2=&b;
    if (a<b) {
       p=p1;
       p1=p2;
       p2=p;
       }
    cout << *p1 << ' ' << *p2 << endl;
}
```



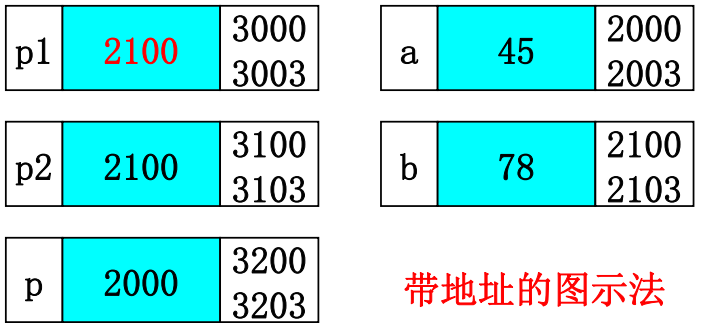
带地址的图示法



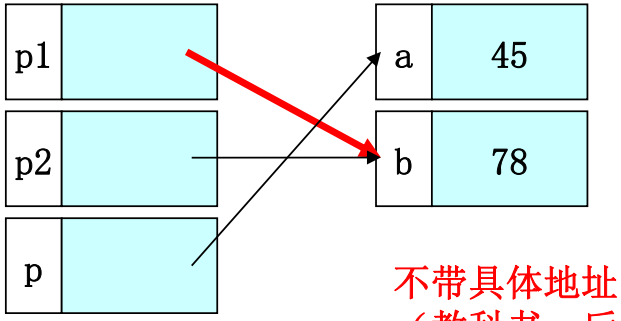
不带具体地址的图示法
（教科书、后续课程）

例：键盘输入两数，按从大到小的顺序依次输出（指针法处理）

```
int main()
{   int *p1, *p2, *p, a, b;
    cin >> a >> b;  (假设键盘输入是45 78)
    p1=&a;
    p2=&b;
    if (a<b) {
       p=p1;
       p1=p2;
       p2=p;
       }
    cout << *p1 << ' ' << *p2 << endl;
}
```



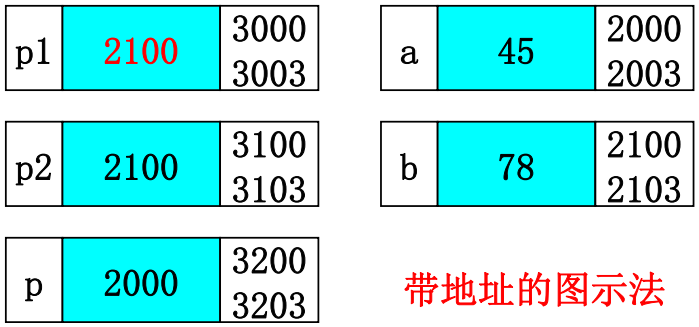
带地址的图示法



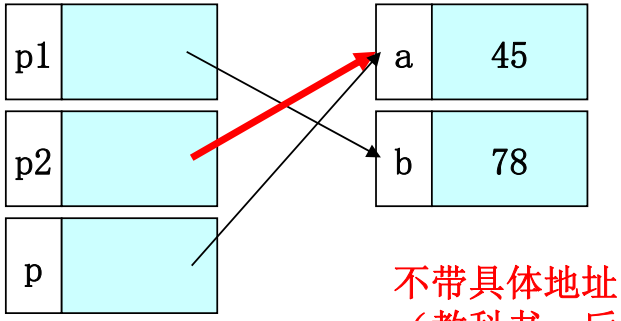
不带具体地址的图示法
（教科书、后续课程）

例：键盘输入两数，按从大到小的顺序依次输出（指针法处理）

```
int main()
{   int *p1, *p2, *p, a, b;
    cin >> a >> b;  (假设键盘输入是45 78)
    p1=&a;
    p2=&b;
    if (a<b) {
       p=p1;
       p1=p2;
       p2=p;
       }
    cout << *p1 << ' ' << *p2 << endl;
}
```



带地址的图示法



不带具体地址的图示法
（教科书、后续课程）

例：键盘输入两数，按从大到小的顺序依次输出（指针法处理）

```
int main()
{   int *p1, *p2, *p, a, b;
    cin >> a >> b;  (假设键盘输入是45 78)
    p1=&a;
    p2=&b;
    if (a<b) {
       p=p1;
       p1=p2;
       p2=p;
       }
    cout << *p1 << ' ' << *p2 << endl;
}
```

赋值语句不能表示为：*p1=&a;
*p2=&b;

若表示为定义时赋初值，则
int a,b,*p1=&a, *p2=&b, *p;

§12. 指针进阶

2. 变量与指针(复习)

2.5. 指针变量作函数的参数

例：编写一个函数，将两个整数进行交换后输出

```
void swap(int x, int y)
{    int t;
     t = x;
     x = y;
     y = t;
}
int main()
{    int i=10, j=15;
     cout << "i=" << i << " j=" << j << endl;    i=10 j=15
     swap(i,j);
     cout << "i=" << i << " j=" << j << endl;    i=10 j=15
}
```

实参：整型简单变量
形参：整型简单变量
匹配

为什么无法交换？

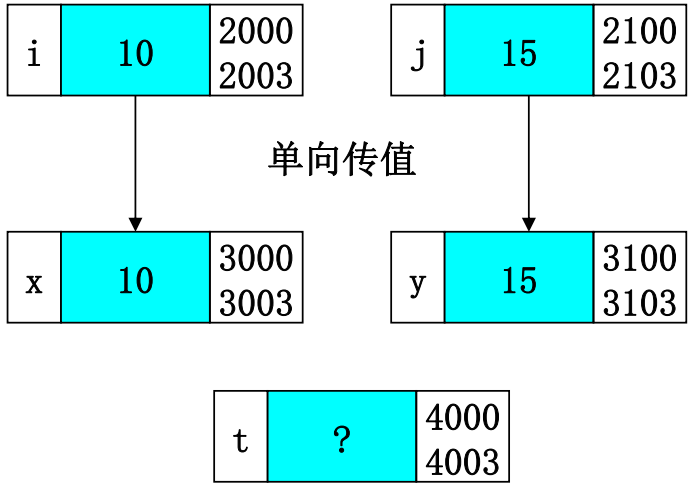
§12. 指针进阶

2. 变量与指针(复习)

2.5. 指针变量作函数的参数

例：编写一个函数，将两个整数进行交换后输出

```
void swap(int x, int y)
{    int t;
     t = x;
     x = y;
     y = t;
}
int main()
{    int i=10, j=15;
     cout << "i=" << i << " j=" << j << endl;    i=10 j=15
     swap(i,j);
     cout << "i=" << i << " j=" << j << endl;    i=10 j=15
}
```



为什么无法交换？

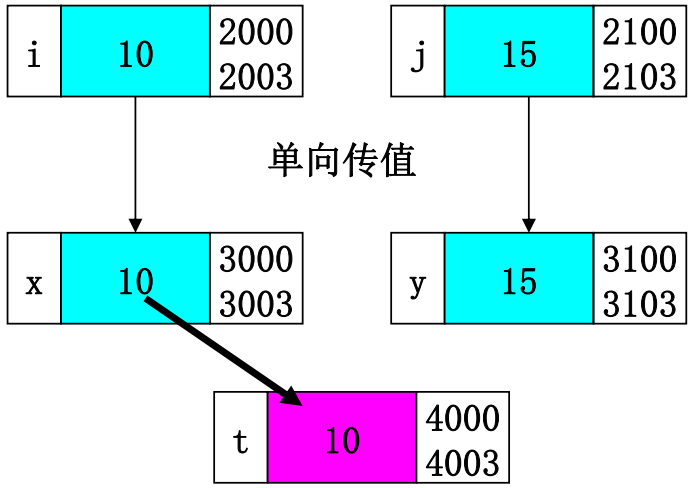
§12. 指针进阶

2. 变量与指针(复习)

2.5. 指针变量作函数的参数

例：编写一个函数，将两个整数进行交换后输出

```
void swap(int x, int y)
{    int t;
     t = x;
     x = y;
     y = t;
}
int main()
{    int i=10, j=15;
     cout << "i=" << i << " j=" << j << endl;    i=10 j=15
     swap(i,j);
     cout << "i=" << i << " j=" << j << endl;    i=10 j=15
}
```



为什么无法交换？

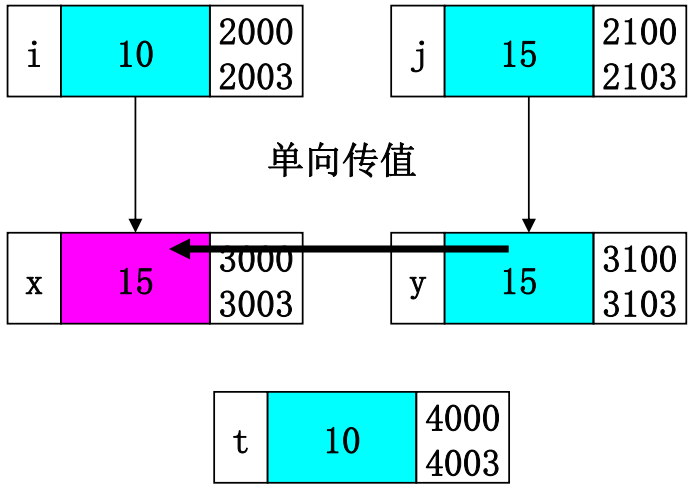
§12. 指针进阶

2. 变量与指针(复习)

2.5. 指针变量作函数的参数

例：编写一个函数，将两个整数进行交换后输出

```
void swap(int x, int y)
{    int t;
     t = x;
     x = y;
     y = t;
}
int main()
{    int i=10, j=15;
     cout << "i=" << i << " j=" << j << endl;    i=10 j=15
     swap(i,j);
     cout << "i=" << i << " j=" << j << endl;    i=10 j=15
}
```



为什么无法交换？

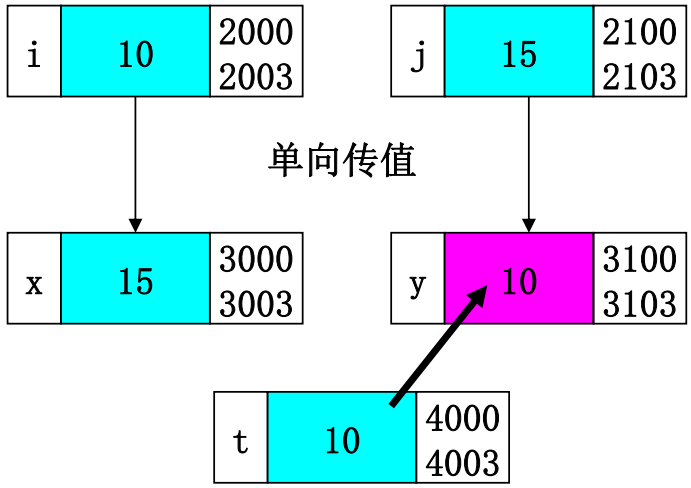
§12. 指针进阶

2. 变量与指针(复习)

2.5. 指针变量作函数的参数

例：编写一个函数，将两个整数进行交换后输出

```
void swap(int x, int y)
{    int t;
     t = x;
     x = y;
     y = t;
}
int main()
{    int i=10, j=15;
     cout << "i=" << i << " j=" << j << endl;    i=10 j=15
     swap(i,j);
     cout << "i=" << i << " j=" << j << endl;    i=10 j=15
}
```



为什么无法交换？

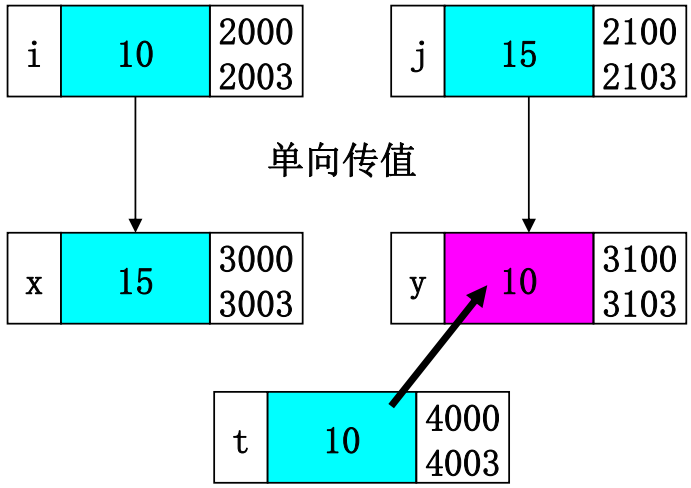
§12. 指针进阶

2. 变量与指针(复习)

2.5. 指针变量作函数的参数

例：编写一个函数，将两个整数进行交换后输出

```
void swap(int x, int y)
{    int t;
     t = x;
     x = y;
     y = t;
}
int main()
{    int i=10, j=15;
     cout << "i=" << i << " j=" << j << endl;    i=10 j=15
     swap(i,j);
     cout << "i=" << i << " j=" << j << endl;    i=10 j=15
}
```



为什么无法交换？

错误原因：C/C++中函数参数是单向传值，
形参的改变不能影响实参

§12. 指针进阶

2. 变量与指针(复习)

2.5. 指针变量作函数的参数

例：编写一个函数，将两个整数进行交换后输出

```
void swap(int *x, int *y)
{    int t;
     t = *x;
     *x = *y;
     *y = t;
}
int main()
{    int i=10, j=15;
     cout << "i=" << i << " j=" << j << endl;    i=10 j=15
     swap(&i,&j);
     cout << "i=" << i << " j=" << j << endl;    i=15 j=10
}
```

实参：整型变量地址
形参：整型指针变量　匹配

正确的方法

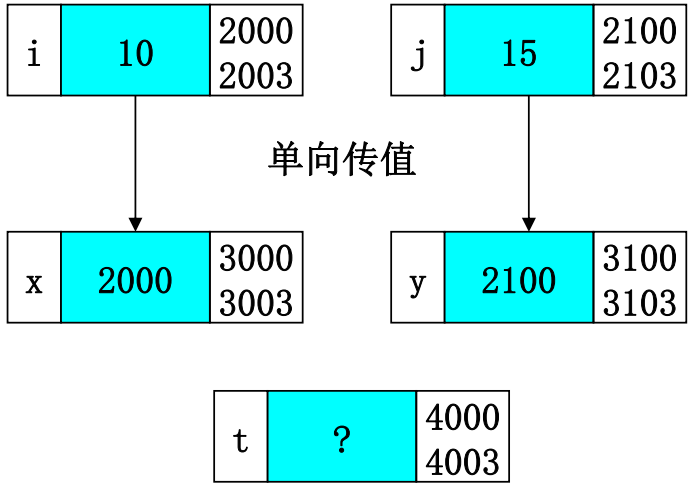
§12. 指针进阶

2. 变量与指针(复习)

2.5. 指针变量作函数的参数

例：编写一个函数，将两个整数进行交换后输出

```
void swap(int *x, int *y)
{    int t;
     t = *x;
     *x = *y;
     *y = t;
}
int main()
{    int i=10, j=15;
     cout << "i=" << i << " j=" << j << endl;    i=10 j=15
     swap(&i,&j);
     cout << "i=" << i << " j=" << j << endl;    i=15 j=10
}
```



正确的方法

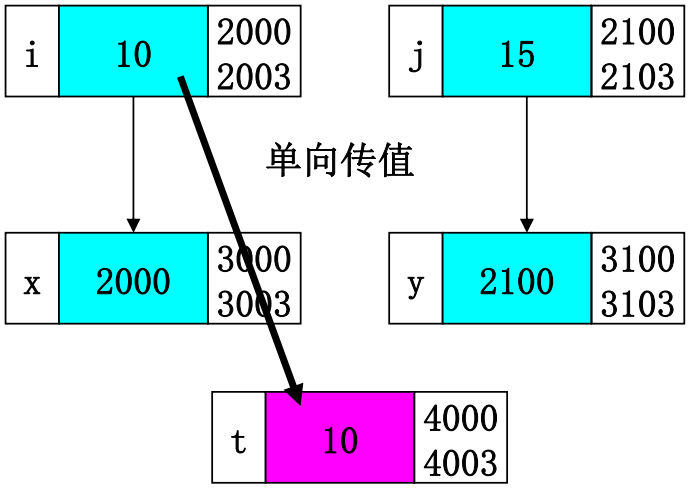
§12. 指针进阶

2. 变量与指针(复习)

2.5. 指针变量作函数的参数

例：编写一个函数，将两个整数进行交换后输出

```
void swap(int *x, int *y)
{    int t;
     t = *x;
     *x = *y;
     *y = t;
}
int main()
{    int i=10, j=15;
     cout << "i=" << i << " j=" << j << endl;    i=10 j=15
     swap(&i,&j);
     cout << "i=" << i << " j=" << j << endl;    i=15 j=10
}
```



正确的方法

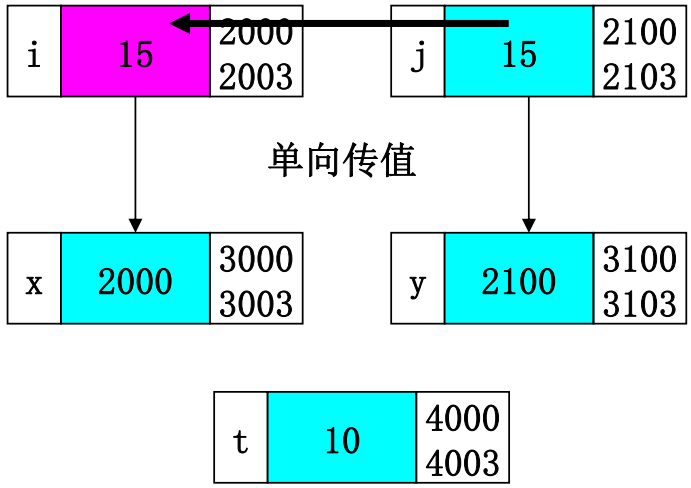
§12. 指针进阶

2. 变量与指针(复习)

2.5. 指针变量作函数的参数

例：编写一个函数，将两个整数进行交换后输出

```
void swap(int *x, int *y)
{    int t;
     t = *x;
     *x = *y;
     *y = t;
}
int main()
{    int i=10, j=15;
     cout << "i=" << i << " j=" << j << endl;    i=10 j=15
     swap(&i,&j);
     cout << "i=" << i << " j=" << j << endl;    i=15 j=10
}
```



正确的方法

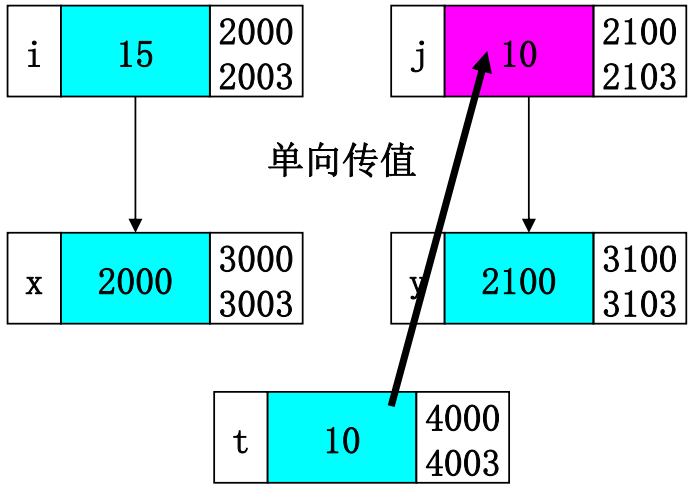
§12. 指针进阶

2. 变量与指针(复习)

2.5. 指针变量作函数的参数

例：编写一个函数，将两个整数进行交换后输出

```
void swap(int *x, int *y)
{    int t;
     t = *x;
     *x = *y;
     *y = t;
}
int main()
{    int i=10, j=15;
     cout << "i=" << i << " j=" << j << endl;    i=10 j=15
     swap(&i,&j);
     cout << "i=" << i << " j=" << j << endl;    i=15 j=10
}
```



正确的方法

★ 指针变量做函数参数，虽然可以通过形参(实参地址)来间接访问实参，从而达到改变实参值的目的，但本质上仍然是单向传值，而不是形参值回传实参

§12. 指针进阶

2. 变量与指针(复习)

2.5. 指针变量作函数的参数

例：编写一个函数，将两个整数进行交换后输出

```
void swap(int *x, int *y)
{    int t;
     t = *x;
     *x = *y;
     *y = t;
}
int main()
{    int i=10, j=15, *p1=&i, *p2=&j;
     cout << "i=" << i << " j=" << j << endl;    i=10 j=15
     swap(p1,p2);
     cout << "i=" << i << " j=" << j << endl;    i=15 j=10
}
```

实参：整型指针变量
形参：整型指针变量
匹配

正确的方法

与swap(&i,&j)等价

§12. 指针进阶

2. 变量与指针(复习)

2.5. 指针变量作函数的参数

★ 指针变量做函数参数，虽然可以通过形参(实参地址)来间接访问实参，从而达到改变实参值的目的，但本质上仍然是单向传值，而不是形参值回传实参

★ 指针变量做参数，可以同时得到多个改变的实参值，从而达到一个函数返回多个值的目的

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

#define PI 3.14159

void SL(double R, double *S, double *L)
{   *S = PI*R*R;
    *L = 2*PI*R;    函数执行后同时得到周长及面积
}                   (都是指针变量做函数形参)
int main()
{   double s,l, r=3;
    SL(r,&s,&l);
    cout << "s=" << s << endl;  s=28.2743
    cout << "l=" << l << endl;  l=18.8495
}
```

§12. 指针进阶

2. 变量与指针(复习)

2.5. 指针变量作函数的参数

★ 指针变量做函数参数，虽然可以通过形参(实参地址)来间接访问实参，从而达到改变实参值的目的，但本质上仍然是单向传值，而不是形参值回传实参

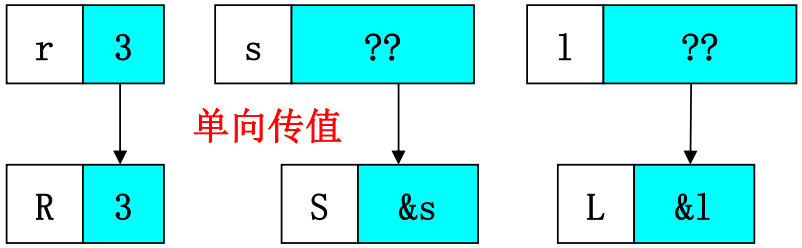
★ 指针变量做参数，可以同时得到多个改变的实参值，从而达到一个函数返回多个值的目的

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

#define PI 3.14159

void SL(double R, double *S, double *L)
{   *S = PI*R*R;
    *L = 2*PI*R;
}
int main()
{   double s,l, r=3;
    SL(r,&s,&l);
    cout << "s=" << s << endl;  s=28.2743
    cout << "l=" << l << endl;  l=18.8495
}
```

实参与形参



§12. 指针进阶

2. 变量与指针(复习)

2.5. 指针变量作函数的参数

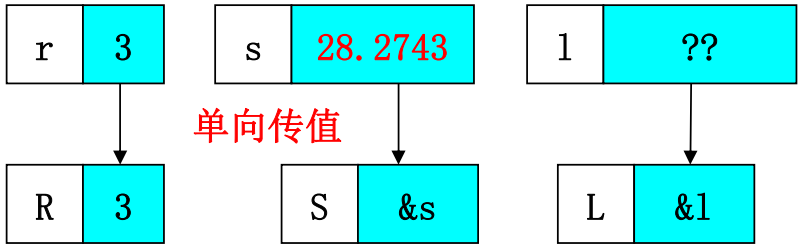
★ 指针变量做函数参数，虽然可以通过形参(实参地址)来间接访问实参，从而达到改变实参值的目的，但本质上仍然是单向传值，而不是形参值回传实参

★ 指针变量做参数，可以同时得到多个改变的实参值，从而达到一个函数返回多个值的目的

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

#define PI 3.14159

void SL(double R, double *S, double *L)
{   *S = PI*R*R;
    *L = 2*PI*R;
}
int main()
{   double s,l, r=3;
    SL(r,&s,&l);
    cout << "s=" << s << endl;  s=28.2743
    cout << "l=" << l << endl;  l=18.8495
}
```



§12. 指针进阶

2. 变量与指针(复习)

2.5. 指针变量作函数的参数

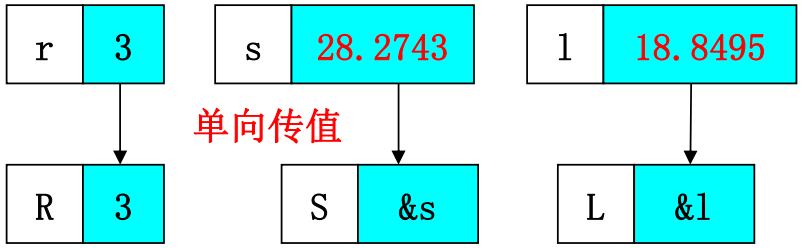
★ 指针变量做函数参数，虽然可以通过形参(实参地址)来间接访问实参，从而达到改变实参值的目的，但本质上仍然是单向传值，而不是形参值回传实参

★ 指针变量做参数，可以同时得到多个改变的实参值，从而达到一个函数返回多个值的目的

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

#define PI 3.14159

void SL(double R, double *S, double *L)
{   *S = PI*R*R;
    *L = 2*PI*R;
}
int main()
{   double s,l, r=3;
    SL(r,&s,&l);
    cout << "s=" << s << endl;  s=28.2743
    cout << "l=" << l << endl;  l=18.8495
}
```



§12. 指针进阶

2. 变量与指针(复习)

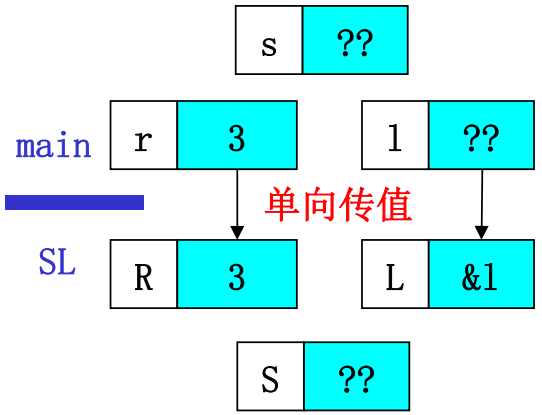
2.5. 指针变量作函数的参数

★ 指针变量做函数参数，虽然可以通过形参(实参地址)来间接访问实参，从而达到改变实参值的目的，但本质上仍然是单向传值，而不是形参值回传实参

★ 指针变量做参数，可以同时得到多个改变的实参值，从而达到一个函数返回多个值的目的

```
#include <iostream>
using namespace std;
#define PI 3.14159
double SL(double R, double *L)
{   double S;
    S = PI*R*R;
    *L = 2*PI*R;
    return S;
}
int main()
{   double s,l, r=3;
    s=SL(r,&l);
    cout << "s=" << s << endl;   s=28.2743
    cout << "l=" << l << endl;   l=18.8495
}
```

函数执行后同时得到周长及面积
周长：指针变量做形参方式
面积：函数返回值方式
注:函数的return只能带一个返回值!!



初始内存分配如图所示
请自行画出SL中三句话
执行时内存的变化
理解最后的输出结果

§12. 指针进阶

2. 变量与指针(复习)

2.5. 指针变量作函数的参数

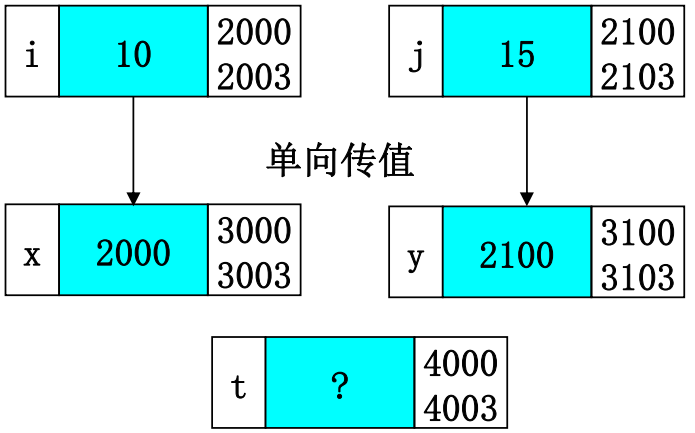
★ 指针变量做函数参数，虽然可以通过形参(实参地址)来间接访问实参，从而达到改变实参值的目的，但本质上仍然是单向传值，而不是形参值回传实参

★ 指针变量做参数，可以同时得到多个改变的实参值，从而达到一个函数返回多个值的目的

★ 必须通过改变形参指针变量所指变量(即实参)值的方法来达到改变实参值的目的，仅通过改变形参指针变量的值的方法是无效的

```
void swap(int *x, int *y)
{    int *t;
     t = x;
     x = y;
     y = t;
}

int main()
{
     int i=10, j=15;
     cout << "i=" << i << " j=" << j << endl;    i=10 j=15
     swap(&i,&j);
     cout << "i=" << i << " j=" << j << endl;    i=10 j=15
}
```



初始内存分配如图所示
请自行画出swap中三句话执行时内存的变化
理解为什么无法交换

§12. 指针进阶

2. 变量与指针(复习)

2.5. 指针变量作函数的参数

★ 指针变量做函数参数，虽然可以通过形参(实参地址)来间接访问实参，从而达到改变实参值的目的，但本质上仍然是单向传值，而不是形参值回传实参

★ 指针变量做参数，可以同时得到多个改变的实参值，从而达到一个函数返回多个值的目的

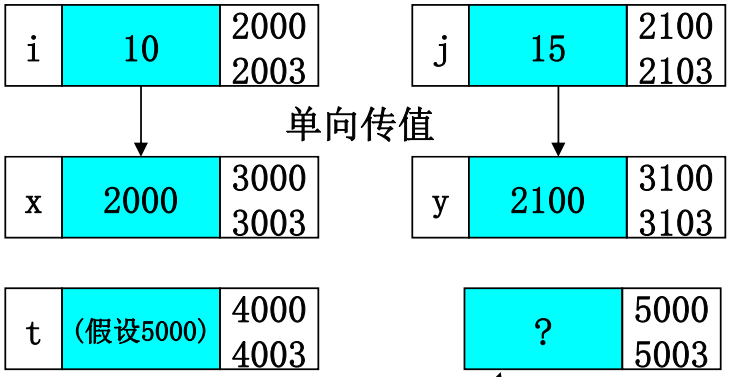
★ 必须通过改变形参指针变量所指变量(即实参)值的方法来达到改变实参值的目的，仅通过改变形参指针变量的值的方法是无效的

★ 指针变量的使用，一定要有确定的值，否则会出现错误

```
void swap(int *x, int *y)
{    int *t;
     *t = *x;
     *x = *y;
     *y = *t;
}

int main()
{
     int i=10, j=15;
     cout << "i=" << i << " j=" << j << endl;
     swap(&i,&j);
     cout << "i=" << i << " j=" << j << endl;
}
```

VS2019编译报错
-使用了未初始化的局部变量t
其它编译器可能可以运行
初始内存分配如图所示，请自行画出swap中三句话执行时内存的变化，理解为什么出现严重错误
另1：哪句是错误的关键？
另2：int *t 改为 int tt, *t;
t = &tt;
为什么就正确了？



i=10 j=15

i=15 j=10

或 死机或其它非正常现象

§12. 指针进阶

3. 一维数组与指针(复习)

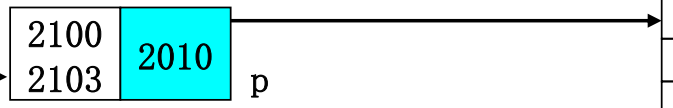
3.1. 基本概念

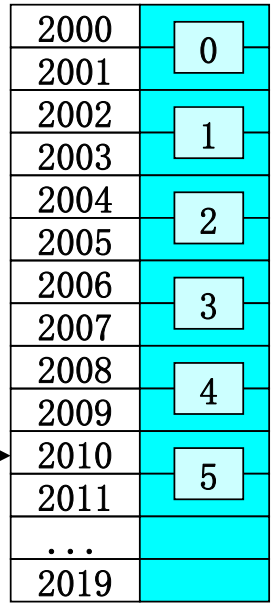
数组的指针：数组的起始地址(⇔&a[0])

数组元素的指针：数组中某个元素的地址

3.2. 指向数组元素的指针变量

```
short a[10], *p;
p = &a[5];
```

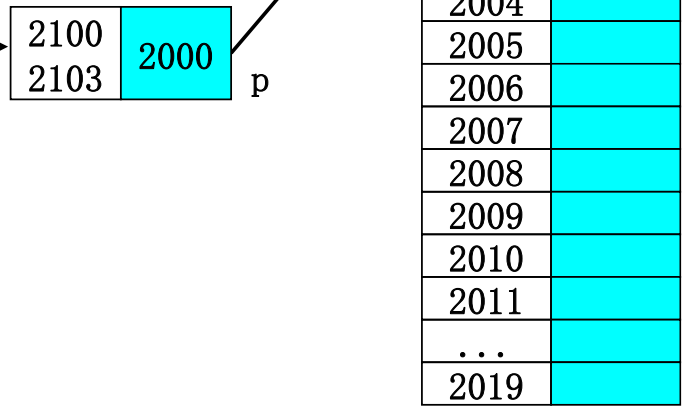




表示p指向a数组的第5个(从0开始)元素

3.3. 指向数组的指针变量

```
short a[10], *p;
p = &a[0]：数组的第0个元素的地址就是数组的起始地址
p = a    ：数组名代表首地址
```



§12. 指针进阶

3. 一维数组与指针(复习)

3.1. 基本概念

数组的指针：数组的起始地址(⇔&a[0])

数组元素的指针：数组中某个元素的地址

3.2. 指向数组元素的指针变量

3.3. 指向数组的指针变量

short a[10], *p;

p = &a[5]：表示p指向a数组的第5个(从0开始)元素

p = &a[0]：数组的第0个元素的地址就是数组的起始地址

p = a　　：数组名代表首地址

★ 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)

```
#include <iostream>
using namespace std;
int main()
{   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
}
```

§12. 指针进阶

3. 一维数组与指针(复习)

3.4. 指针法引用数组元素

3.4.1. 形式

int a[10], *p;

p=&a[5];

*p=10 ⇔ a[5]=10

指针法　下标法

3.4.2. 下标法与指针法的区别

若int a[10], *p=a

★ p[i] ⇔ *(p+i)

★ a[i] ⇔ *(a+i)

都表示访问数组的第i个元素
等价关系，非常重要!!!

★ 数组的首地址不可变，指针的值可以改变

a++ ×　　p++ ✓

★ C/C++语言对指针/数组下标的越界不做检查，因此必须保证引用有效的数组元素，否则可能产生错误

int a[10], *p=a;

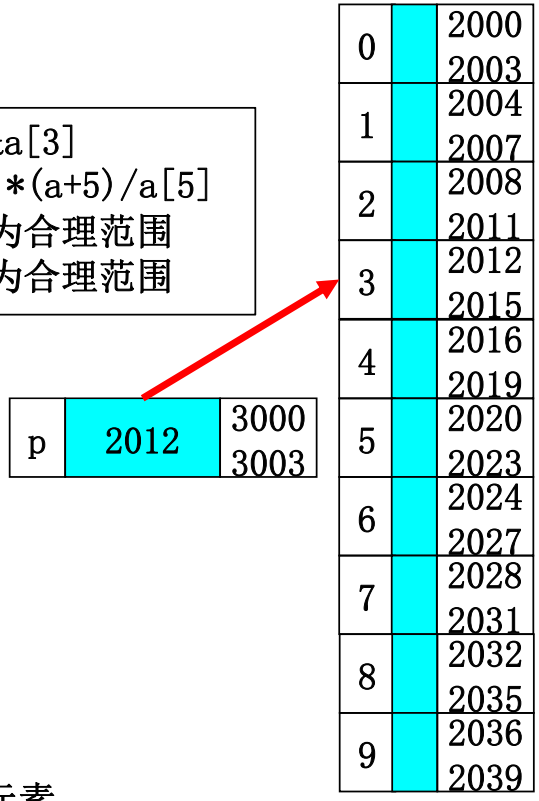
p[100]/*(p+100)/a[100]/*(a+100)

编译正确，使用出错

★ p[i]/*(p+i)/a[i]/*(a+i)的求值过程

取p/a的地址为基地址，则p[i]/*(p+i)/a[i]/*(a+i)的地址为 基地址+i*sizeof(基类型)

若：int a[10], *p=&a[3]
则：*(p+2)/p[2] ⇔ *(a+5)/a[5]
a[0] - a[9] 为合理范围
p[-3] - p[6] 为合理范围



C++源程序文件中的下标形式在可执行文件中都按指针形式处理，即 a[i] 按 *(a+i) 的方式处理，因此可以理解为可执行文件中已经无下标的概念，也就不会对下标越界进行检查

● VS2019会有IntelliSense(智能提示)，但不够准确，经常误判

§12. 指针进阶

3. 一维数组与指针(复习)

3.4. 指针法引用数组元素

3.4.2. 下标法与指针法的区别

★ 常见用法与错误

例：键盘读入10个数并输出

P.165 例6.5 (1)-数组名用下标法

```
int main()
{   int a[10], i;
    for(i=0; i<10; i++)
       cin >> a[i];
    cout << endl; //先输出一个换行，和输入分开
    for(i=0; i<10; i++)
       cout << a[i] << " ";
    cout << endl;
    return 0;
}
```

P.165 例6.5 (2)-数组名用指针法

```
int main()
{   int a[10], i;
    for(i=0; i<10; i++)
       cin >> *(a+i);
    cout << endl; //先输出一个换行，和输入分开
    for(i=0; i<10; i++)
       cout << *(a+i) << " ";
    cout << endl;
    return 0;
}
```

4种方法都正确，效率相同

P.165 例6.5 (1)变化-指针用下标法

```
int main()
{   int a[10], i, *p=a;
    for(i=0; i<10; i++)
       cin >> p[i];
    cout << endl; //先输出一个换行，和输入分开
    for(i=0; i<10; i++)
       cout << p[i] << " ";
    cout << endl;
    return 0;
}
```

P.165 例6.5 (2)变化-指针用指针法

```
int main()
{   int a[10], i, *p=a;
    for(i=0; i<10; i++)
       cin >> *(p+i);
    cout << endl; //先输出一个换行，和输入分开
    for(i=0; i<10; i++)
       cout << *(p+i) << " ";
    cout << endl;
    return 0;
}
```

§12. 指针进阶

3. 一维数组与指针(复习)

3.4. 指针法引用数组元素

3.4.2. 下标法与指针法的区别

★ 常见用法与错误

例：键盘读入10个数并输出

```
P.165 例6.5 (3)
int main()
{   int a[10], i, *p=a;
    for(i=0; i<10; i++)
       cin >> *(p+i);
    cout << endl; //先输出一个换行，和输入分开
    for(p=a; p<a+10; p++)
       cout << *p << " ";
    cout << endl;
    return 0;
}
```

p<a+10 表示p和a是否相差10个int型元素

```
P.165 例6.5 (3)-变化
int main()
{   int a[10], i, *p=a;
    for(i=0; i<10; i++)
       cin >> *(p+i);
    cout << endl; //先输出一个换行，和输入分开
    for(p=a; p<a+10; p++)
       cout << *p << " ";
    cout << endl;
    return 0;
}
```

去掉阴影中语句正确

```
P.165 例6.5 (3)-变化
int main()
{   int a[10], i, *p=a;
    for(p=a; p<a+10; p++)
       cin >> *p;
    cout << endl; //先输出一个换行，和输入分开
    for(p=a; p<a+10; p++)
       cout << *p << " ";
    cout << endl;
    return 0;
}
```

去掉后2处阴影中语句错误
为什么？哪个不能去？

执行效率高于前4种实现方式
(前4种的效率相同)
前几种：每次计算p+i*sizeof(int)
本程序：只要 p+sizeof(int)

§12. 指针进阶

3. 一维数组与指针(复习)

3.4. 指针法引用数组元素

3.4.2. 下标法与指针法的区别

★ 常见用法与错误

例：键盘读入10个数并输出

P.165 例6.5 (3)-变化

```
int main()
{   int a[10], i, *p=a;
    for(p=a; p<a+10;)
       cin >> *p++;
    cout << endl; //先输出一个换行，和输入分开
    for(p=a; p<a+10;)
       cout << *p++ << " ";
    cout << endl;
    return 0;
}
```

正确

P.165 例6.5 (3)-变化

```
int main()
{   int a[10], i, *p=a;
    for(p=a; p-a<10;)
       cin >> *p++;
    cout << endl; //先输出一个换行，和输入分开
    for(p=a; p-a<10;)
       cout << *p++ << " ";
    cout << endl;
    return 0;
}
```

正确

P.165 例6.5 (3)-变化

```
int main()
{   int a[10], i, *p=a;
    for(p=a; p-a<10; p++)
       cin >> *p;
    cout << endl; //先输出一个换行，和输入分开
    for(p=a; p-a<10; p++)
       cout << *p << " ";
    cout << endl;
    return 0;
}
```

正确

p-a<10 ⇔ p<a+10
表示p和a是否相差10个int型的元素

§12. 指针进阶

3. 一维数组与指针(复习)

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

A. 指针变量 ± 整数 (包括++/--)

指针变量++ ⇔ 所指地址 += sizeof(基类型)

指针变量-- ⇔ 所指地址 -= sizeof(基类型)

指针变量+n ⇔ 所指地址 + n*sizeof(基类型)

指针变量-n ⇔ 所指地址 - n*sizeof(基类型)

```
#include <iostream>
using namespace std;
int main()
{
   int a[10], *p=a;
   cout << a << "--" << ++p << endl;        地址a--地址a+4
   p = &a[5];
   cout << &a[5] << "--" << --p << endl;    地址a+20--地址a+16
   p = &a[3];
   cout << p << "--" << (p+3) << endl;      地址a+12--地址a+24
   p = &a[7];
   cout << p << "--" << (p-3) << endl;      地址a+28--地址a+16
}
```

实际运行一次，观察打印出来的地址间的关系

§12. 指针进阶

3. 一维数组与指针(复习)

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

A. 指针变量 ± 整数 (包括++/--)

指针变量++ ⇔ 所指地址 += sizeof(基类型)

指针变量-- ⇔ 所指地址 -= sizeof(基类型)

指针变量+n ⇔ 所指地址 + n*sizeof(基类型)

指针变量-n ⇔ 所指地址 - n*sizeof(基类型)

★ 若指针变量指向数组，则±n表示前/后的n个元素

注意不要超出数组的范围，否则无意义

假设：int a[10], *p=&a[3];

p++ : p指向a[4]

p-- : p指向a[2]

p+5 : a[8]的地址(p未变)

p-3 : a[0]的地址(p未变)

p+=3 : p指向a[6]

p-=2 : p指向a[1]

p+9 : a[12]的地址(已越界)

★ 若指针变量指向简单变量，则语法正确，但无实际意义

假设：int a, b, *p=&a, *q=&b;

p++ :若a的地址为2000，则p指向2004(不再指向a)

q-=3:若b的地址为2100，则q指向2088(不再指向b)

1. 可以运算并得到结果，但结果无实际意义

2. 即使p++/q--指向其它简单变量也没有实际意义

§12. 指针进阶

3. 一维数组与指针(复习)

3.4. 指针法引用数组元素

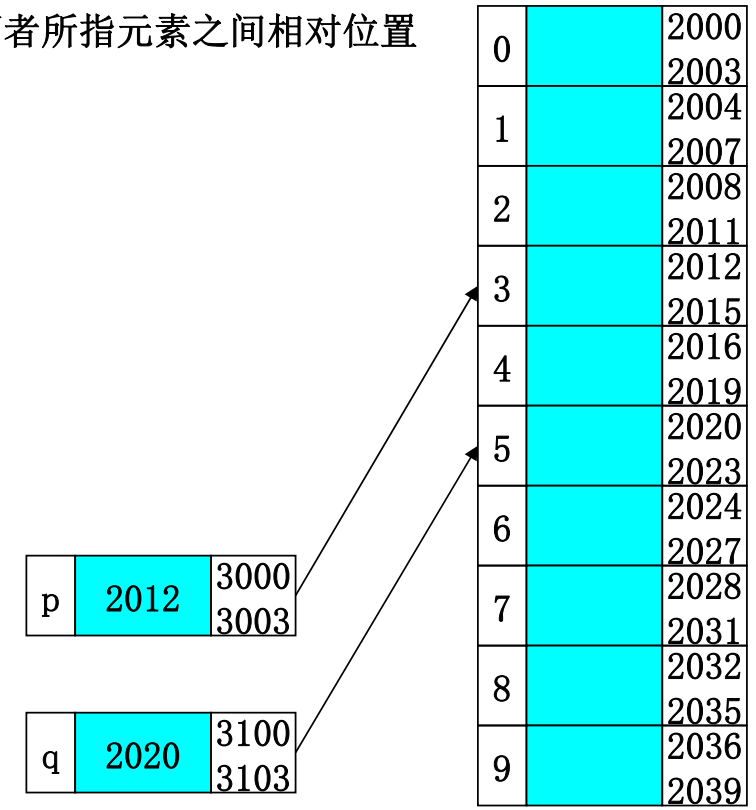
3.4.3. 指向数组的指针变量的运算

B. 两个基类型相同的指针变量相减

指针变量1-指针变量2 ⇔ 地址差/sizeof(基类型)

★ 若两个指针变量都指向同一个数组，则差值表示两者所指元素之间相对位置

```
#include <iostream>
using namespace std;
int main()
{   int a[10], *p=&a[3], *q=&a[5];
    cout << a << endl;          地址a
    cout << p << endl;          地址a+12
    cout << q << endl;          地址a+20
    cout << (p-q) << endl;      -2
    cout << (q-p) << endl;      2
}
```



§12. 指针进阶

3. 一维数组与指针(复习)

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

B. 两个基类型相同的指针变量相减

指针变量1-指针变量2 ⇔ 地址差/sizeof(基类型)

★ 若两个指针变量都指向同一个数组，则差值表示两者所指元素之间相对位置

★ 若两个指针变量分别指向不同的数组，则语法正确，但无实际意义

```
#include <iostream>
using namespace std;
int main()
{   int a[10], *p=&a[3];
    int b[20], *q=&b[12];

    cout << (p-q) << endl;   16
    cout << (q-p) << endl;   -16
}
```

不同编译器结果不相同

假设：int a[10], *p=&a[3];
int b[20], *q=&b[12];
则：
cout<<(p-q); 某值x(正负不确定)
cout<<(q-p); -x

可以运算并得到确定结果，但结果无实际意义

★ 若两个指针变量分别指向不同的简单变量，则语法正确，但无实际意义

```
#include <iostream>
using namespace std;
int main()
{   int i, *p=&i;
    int j, *q=&j;

    cout<< (p-q) <<endl;   6
    cout<< (q-p) <<endl;   -6
}
```

不同编译器结果不相同

假设：int i, *p=&i;
int j, *q=&j;
则：
cout<<(p-q); 某值x(正负不确定)
cout<<(q-p); -x

可以运算并得到确定结果，但结果无实际意义

§12. 指针进阶

3. 一维数组与指针(复习)

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

指针变量1-指针变量2 比较运算符 n

⇔

指针变量1-指针变量2 比较运算符 n*sizeof(基类型)

指针变量1-指针变量2 比较运算符 n　　(p-q < 2)

等价变换

指针变量1 比较运算符 指针变量2 + n　　(p < q+2)

```
#include <iostream>
using namespace std;
int main()
{
     int a[10], *p=&a[3], *q=&a[5];

     cout << (q-p == 2) << endl;     1
     cout << (q == p+2) << endl;     1

     cout << (q-p <= 2) << endl;     1
     cout << (q <= p+2) << endl;     1

     cout << (p-q < 0)  << endl;     1
     cout << (p < q)    << endl;     1
}
```

```
int a[10]={...}, *p=a;
for(p=a; p-a<10;)
     cout << *p++ << " ";

10表示p和a之间差10个int型元素
(实际地址差40)
```

```
int a[10]={...}, *p=a;
for(p=a; p<a+10;)
     cout << *p++ << " ";

10表示p和a之间差10个int型元素
(实际地址差40)
```

§12. 指针进阶

3. 一维数组与指针(复习)

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

★ 只有当两个指针变量都指向同一个数组时才有意义，若两个指针变量分别指向不同的数组或不同的简单量，则语法正确，但无实际意义（与B相似，不再举例）

★ 指针变量与整数不能进行乘除运算(编译报错)

```
#include <iostream>
using namespace std;
int main()
{   int *p;
    cout << (p*2) << endl;
    cout << (p/2) << endl;
}
```

★ 两个基类型相同的指针变量之间不能进行加/乘/除运算(编译报错)

```
#include <iostream>
using namespace std;
int main()
{   int *p, *q;
    cout << (p+q) << endl;
    cout << (p*q) << endl;
    cout << (p/q) << endl;
}
```

§12. 指针进阶

3. 一维数组与指针(复习)

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

★ 只有当两个指针变量都指向同一个数组时才有意义，若两个指针变量分别指向不同的数组或不同的简单量，则语法正确，但无实际意义（与B相似，不再举例）

★ 指针变量与整数不能进行乘除运算(编译报错)

★ 两个基类型相同的指针变量之间不能进行加/乘/除运算(编译报错)

★ 两个不同基类型的指针变量不能进行包括减及比较在内的任何运算(编译报错)

```
#include <iostream>
using namespace std;
int main()
{   int   *p;
    short *q;
    cout << (p-q)   << endl;
    cout << (p-q<2) << endl;
    cout << (p+q)   << endl;
    cout << (p*q)   << endl;
    cout << (p/q)   << endl;
}
```

★ void型的指针变量不能进行相互运算(不知道基类型)

```
#include <iostream>
using namespace std;
int main()
{   void *p, *q;
    cout << (p+2)   << endl;   //编译错(void无大小)
    cout << (q--)   << endl;   //编译错(void无大小)
    cout << (p-q)   << endl;   //编译错(void无大小)
    cout << (p<q+1) << endl;   //编译错(void无大小)
    cout << (p<q)   << endl;   //编译错(pq未初始化)
}
```

§12. 指针进阶

3. 一维数组与指针(复习)

3.4. 指针法引用数组元素

3.4.4. 指针变量的各种表示

```
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++)：⇔ *p++，表示取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　：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++

§12. 指针进阶

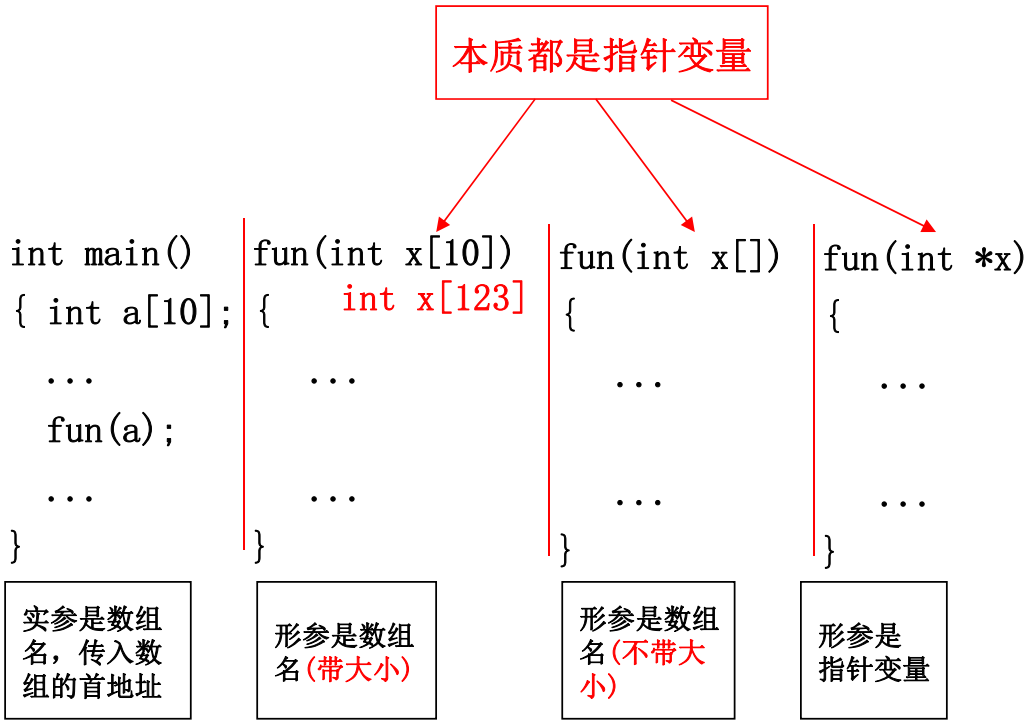
3. 一维数组与指针(复习)

3.4. 指针法引用数组元素

3.5. 用指针变量作函数参数接收数组地址

★ C/C++语言将形参数组作为一个指针变量来处理

★ 开始等于实参数组的首地址，执行过程中可以改变




```
//上学期的例子
#include <iostream>
using namespace std;
void f1(int x1[])      //形参数组不指定大小
{   cout << "x1_size=" << sizeof(x1) << endl;
}
void f2(int x2[10])    //形参数组大小与实参相同
{   cout << "x2_size=" << sizeof(x2) << endl;
}
void f3(int x3[1234]) //形参数组大小与实参不同
{   cout << "x3_size=" << sizeof(x3) << endl;
}
int main()
{   int a[10];
    cout << "a_size=" << sizeof(a) << endl;
    f1(a);
    f2(a);
    f3(a);
}
```

```
a_size=40
x1_size=4   因为int*
x2_size=4   因为int*
x3_size=4   因为int*
(为什么是4指针部分会解释)
```

§12. 指针进阶

3. 一维数组与指针(复习)

3.4. 指针法引用数组元素

3.5. 用指针变量作函数参数接收数组地址

例：选择排序(下标法和指针法对比)

```
void select_sort(int array[], int n)
{   int i,j,k,t;
    for(i=0; i<n-1; i++) {
       k=i;
       for(j=i+1; j<n; j++)
          if (array[j] < array[k])
             k=j;
       t=array[k];
       array[k]=array[i];
       array[i]=t;
       }
}
```

数组法

```
void select_sort(int *array, int n)
{   int i,j,k,t;
    for(i=0; i<n-1; i++) {
       k=i;
       for(j=i+1; j<n; j++)
          if (array[j] < array[k])
             k=j;
       t=array[k];
       array[k]=array[i];
       array[i]=t;
       }
}
```

语句理解为数组法访问指针变量

形参是指针变量其余同左

```
void select_sort(int *p, int n)
{   int i,j,k,t;
    for(i=0; i<n-1; i++) {
       k=i;
       for(j=i+1; j<n; j++)
          if (p[j] < p[k])
             k=j;
       t= p[k];
       p[k] = p[i];
       p[i] =t;
       }
}
```

array换名为p，其余同上

```
void select_sort(int *p, int n)
{   int i,j,k,t;
    for(i=0; i<n-1; i++) {
       k=i;
       for(j=i+1; j<n; j++)
          if (*(p+j) < *(p+k))
             k=j;
       t= *(p+k);
       *(p+k) = *(p+i);
       *(p+i) =t;
       }
}
```

数组法访问改成指针法访问

指针法

§12. 指针进阶

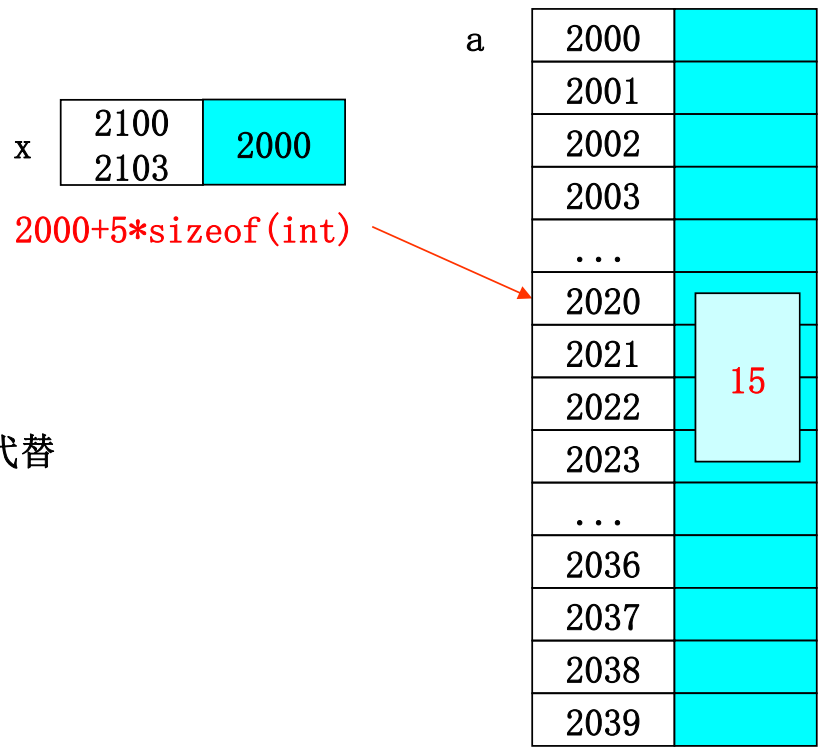
3. 一维数组与指针(复习)

3.4. 指针法引用数组元素

3.5. 用指针变量作函数参数接收数组地址

★ 可以通过改变该指针变量所指的变量的值来达到改变实参数组值的目的

```
void fun(int *x)
{  *(x+5)=15;
}
int main()
{ int a[10];
  ...
  a[5]=10;
  cout << "a[5]=" << a[5]; a[5]=10
  fun(a);
  cout << "a[5]=" << a[5]; a[5]=15
  ...
}
```



★ 实参数组也可以用指向它的指针变量来代替

```
fun(int *x)
{
    ...
}
int main()
{ int a[10], *p;
  p=a;
  ...
  fun(p);
  ...
}
```

§12. 指针进阶

3. 一维数组与指针(复习)

3.4. 指针法引用数组元素

3.5. 用指针变量作函数参数接收数组地址

★ 形参无论表示为数组名形式还是指针变量形式，本质都是一个指针变量

实参/形参的四种组合

```
//形参是一维数组名
void fun(int p[])
{
   ...
}
```

带不带大小都可以

p++正确
本质是指针变量

```
//形参是指针变量
void fun(int *p)
{
   ...
}
```

p++正确

```
//实参是一维数组名
int main()
{   int a[10]={...};

    fun(a);
}
```

a++错误

```
//实参是指针变量
int main()
{   int a[10]={...};
    int *p=a;
    fun(p);
}
```

p++正确

§5. 数组

5.4. 用数组名作函数参数

5.4.1. 用数组元素做函数实参

5.4.2. 用一维数组名做函数实参

对第5章的解释

★ 形参为相应类型的一维数组

★ 实参传递时，将实参数组的首地址(数组名表示数组的首地址)传给形参，因此实、形参数组的内存地址重合(实参占用空间，形参不占用空间)

★ 形参数组值的改变会影响到实参(与简单参数不同)

★ 因为形参数组不分配空间，因此数组大小可不指定

★ 因为形参数组不分配空间，因此实形参的类型必须完全相同，否则可能导致错误

形参本质是指针变量，只是可以用数组法表示，当然没有大小

形参只是指向实参的指针变量，因此可通过访问形参所指变量值的方式来访问实参

形参指针变量p的基类型必须与实参数组的类型的一致，这样p++/p--/*(p+i)/p[i]等操作才等价于访问实参数组的元素

§12. 指针进阶

4. 多维数组与指针(补充，极其重要!!!)

4.1. 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

4. 多维数组与指针(补充，极其重要!!!)

4.1. 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};

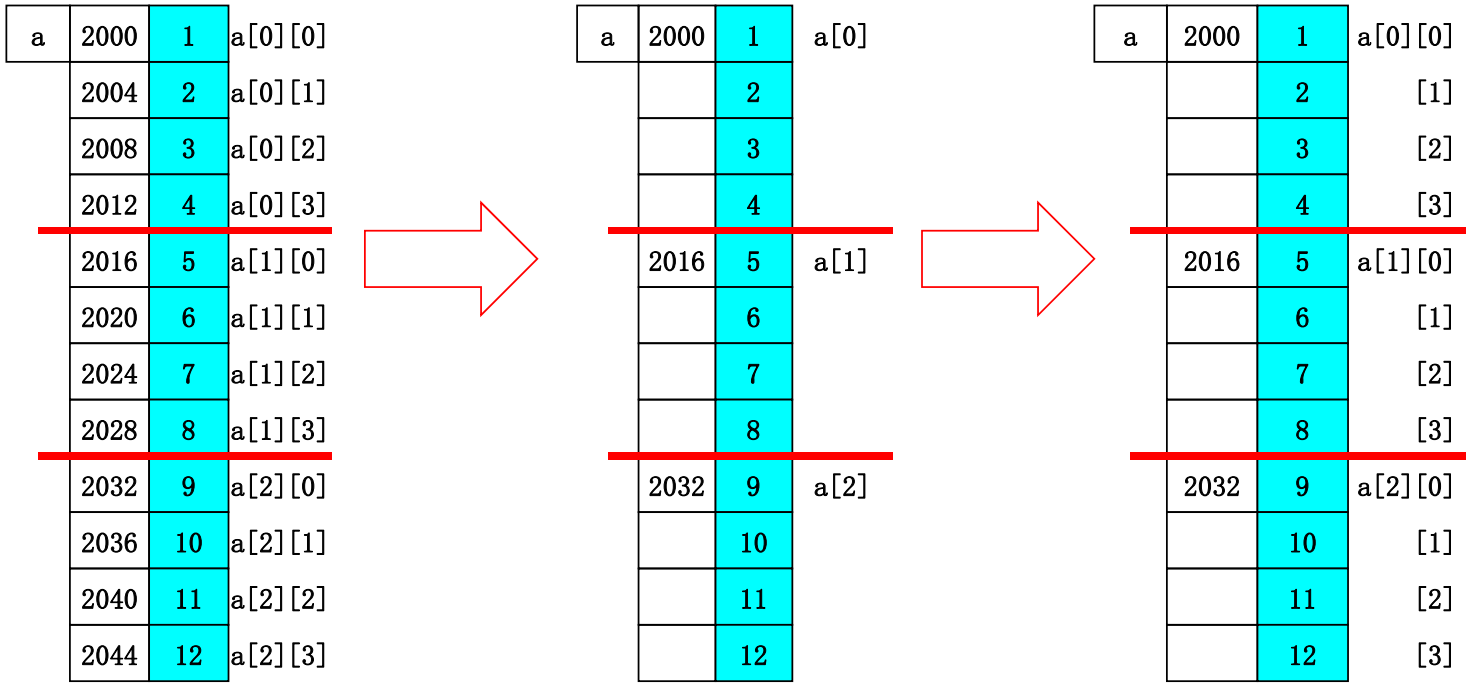
1	2	3	4
5	6	7	8
9	10	11	12

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

4. 多维数组与指针(补充，极其重要!!!)

4.1. 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]

&a[i] ： 3元素一维数组的第i个元素的地址

a+i ： 同上

行地址

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)

*(a+i) ： 同上

元素
地址

i:0-2(行)

§12. 指针进阶

4. 多维数组与指针(补充，极其重要!!!)

4.1. 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

4. 多维数组与指针(补充，极其重要!!!)

4.1. 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

4. 多维数组与指针(补充，极其重要!!!)

4.1. 二维数组的地址

a	：地址(二维数组/第0行)	行地址
&a[i]	：地址(第i行)	
a+i	：地址(第i行)	
a[i]	：地址(第i行0列)	元素地址
*(a+i)	：地址(第i行0列)	
&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)	

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

4. 多维数组与指针(补充，极其重要!!!)

4.1. 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

4. 多维数组与指针(补充，极其重要!!!)

4.1. 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

4. 多维数组与指针(补充，极其重要!!!)

4.1. 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

4. 多维数组与指针(补充，极其重要!!!)

4.1. 二维数组的地址

4.2. 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

4. 多维数组与指针(补充，极其重要!!!)

4.1. 二维数组的地址

4.2. 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

4. 多维数组与指针(补充，极其重要!!!)

4.3. 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

4. 多维数组与指针(补充，极其重要!!!)

4.3. 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是行地址2000

*p是元素地址2000

*(*p+1)　　：元素值2(a[0][1])

*(p+1)+2　　：元素地址2024(a[1][2])　　p+1是行地址2016

*(p+1)是元素地址2016

((p+1)+2)：元素值7(a[1][2])

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

4. 多维数组与指针(补充，极其重要!!!)

4.4. 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

4. 多维数组与指针(补充，极其重要!!!)

4.5. 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

4. 多维数组与指针(补充，极其重要!!!)

4.5. 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

4. 多维数组与指针(补充，极其重要!!!)

4.5. 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

5. 字符串与指针(复习，略)

核心：

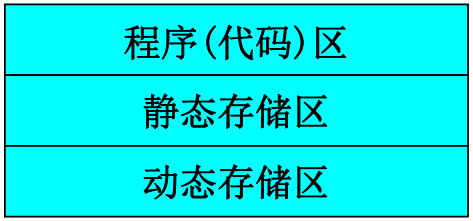
1、char *s 和 char s[80] 有本质区别

2、字符串复制的多种方法及错误分析

§12. 指针进阶

6. 函数与指针

6.1. 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

6.2. 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

6. 函数与指针

6.1. 函数的地址

6.2. 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

6. 函数与指针

6.1. 函数的地址

6.2. 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

6. 函数与指针

6.1. 函数的地址

6.2. 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

6. 函数与指针

6.1. 函数的地址

6.2. 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

6. 函数与指针
6.1. 函数的地址
6.2. 用函数指针变量调用函数
6.3. 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

6. 函数与指针
6.1. 函数的地址
6.2. 用函数指针变量调用函数
6.3. 指向函数的指针做函数参数
6.4. 指向类对象的成员函数的指针

§9. 类和对象基础
9.6. 对象指针
9.6.2. 指向对象成员的指针
9.6.2.1. 指向对象的数据成员的指针
9.6.2.2. 指向对象的成员函数的指针(荣誉课内容，略)

§12. 指针进阶

6. 函数与指针
6.1. 函数的地址
6.2. 用函数指针变量调用函数
6.3. 指向函数的指针做函数参数
6.4. 指向类对象的成员函数的指针

```
/* 6.3.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

6. 函数与指针

6.1. 函数的地址

6.2. 用函数指针变量调用函数

6.3. 指向函数的指针做函数参数

6.4. 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

7. 返回指针值的函数(复习，略)

7.1. 定义

返回基类型 *函数名（形参表）

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

★ 与指向函数的指针变量的区别

```
int *p(int x);
```

p是函数名

(函数形参为一个int型，返回类型 int *)

```
int (*p)(int x);
```

p是指针变量名，指向函数

(函数形参为一个int型，返回类型 int)

7.2. 使用

★ return中的返回值必须是指针（地址）

★ 不能返回一个自动变量/形参的地址，否则可能出错

§12. 指针进阶

7. 返回指针值的函数(复习，略)

7.2. 使用

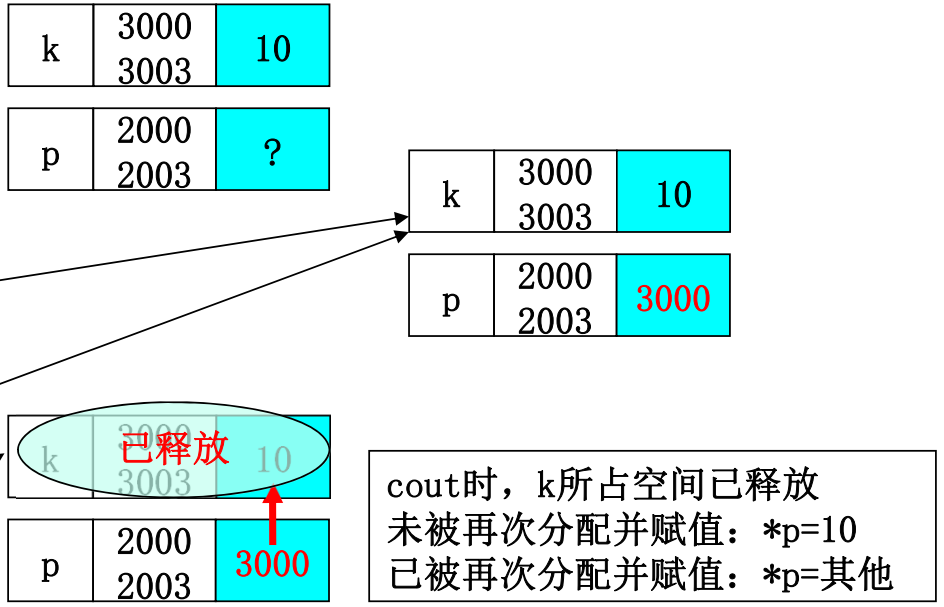
★ return中的返回值必须是指针（地址）

★ 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;//warning
}
int main()
{
    int *p;
    p=fun();
    cout << *p << endl;
}
```

10



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

1、VS2019下有编译警告
2、读非法空间而未写，不会死机

问：如果必须返回局部变量的值，如何做？
答:设为静态局部即可

§12. 指针进阶

7. 返回指针值的函数(复习，略)

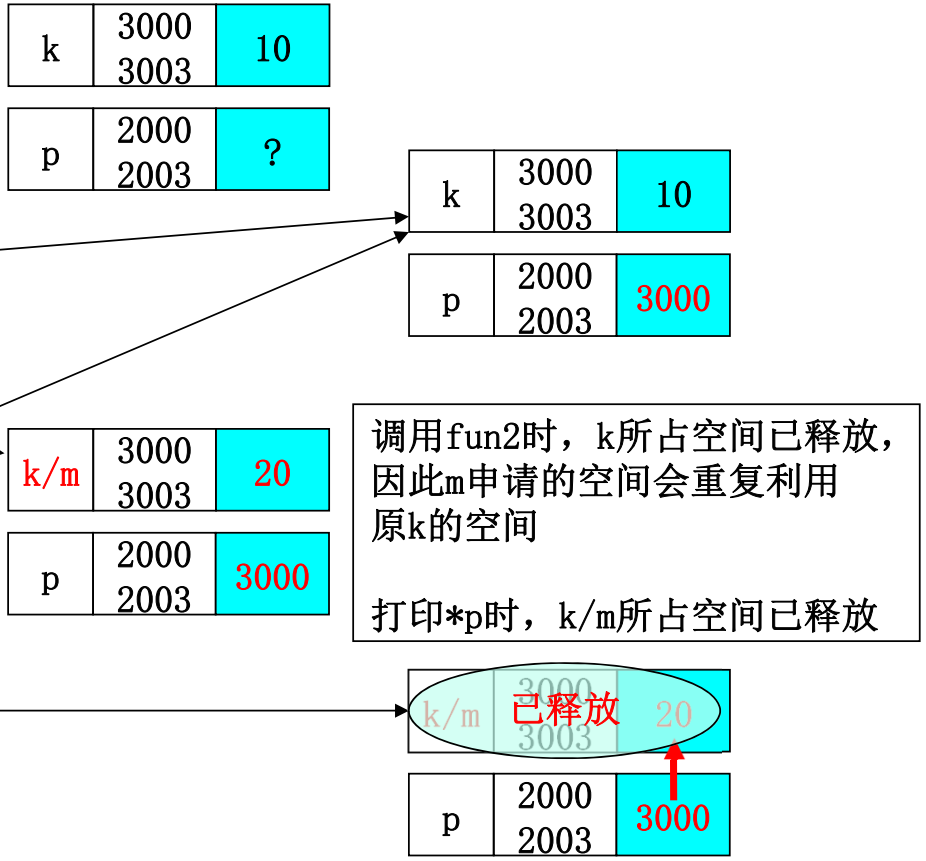
7.2. 使用

★ return中的返回值必须是指针（地址）

★ 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; //warning
}
void fun2()
{   int m=20;
    cout << &m << endl;
}
int main()
{   int *p;
    p=fun();
    fun2();
    cout << *p << endl;
}
```

地址a
地址a
20



§12. 指针进阶

8. 指针数组和指向指针的指针

8.1. 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

8. 指针数组和指向指针的指针

8.1. 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	

赋初值方法

字符串常量
"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

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

交换的方法：

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

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	

§12. 指针进阶

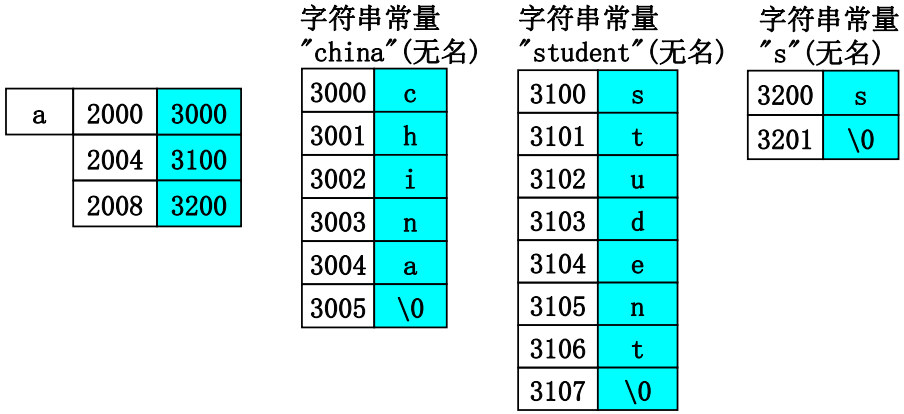
8. 指针数组和指向指针的指针

8.1. 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：

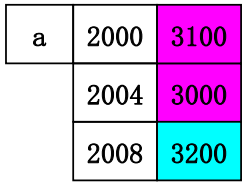
```
char *a[3] = {"china", "student", "s"};
```



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



交换的方法：

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

§12. 指针进阶

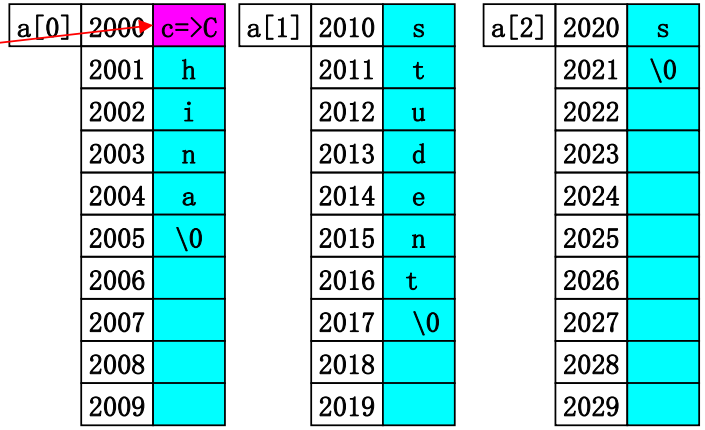
8. 指针数组和指向指针的指针

8.1. 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

```
#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

8. 指针数组和指向指针的指针

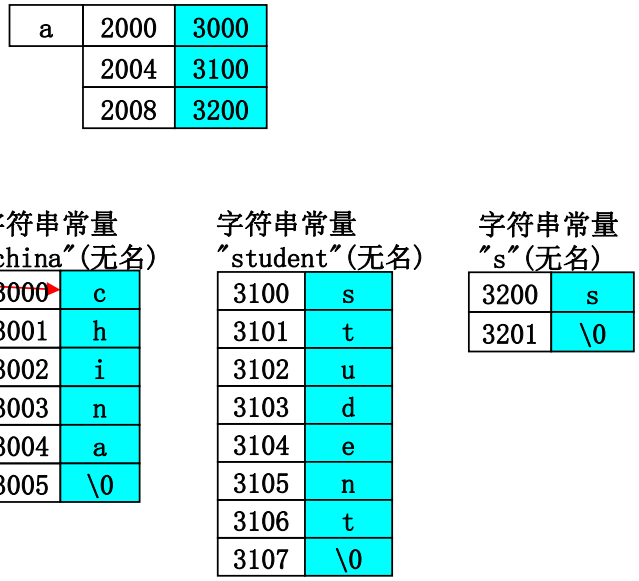
8.1. 指针数组

★ 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

```
#include <iostream>
using namespace std;
int main()
{   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; //Dev有warning运行错，为什么？
    cout << a[0] << endl;          China
    cout << a[1] << endl;          student
    cout << a[2] << endl;          s
    return 0;
}
```

VS编译：error
Dev编译：warning



§12. 指针进阶

8. 指针数组和指向指针的指针

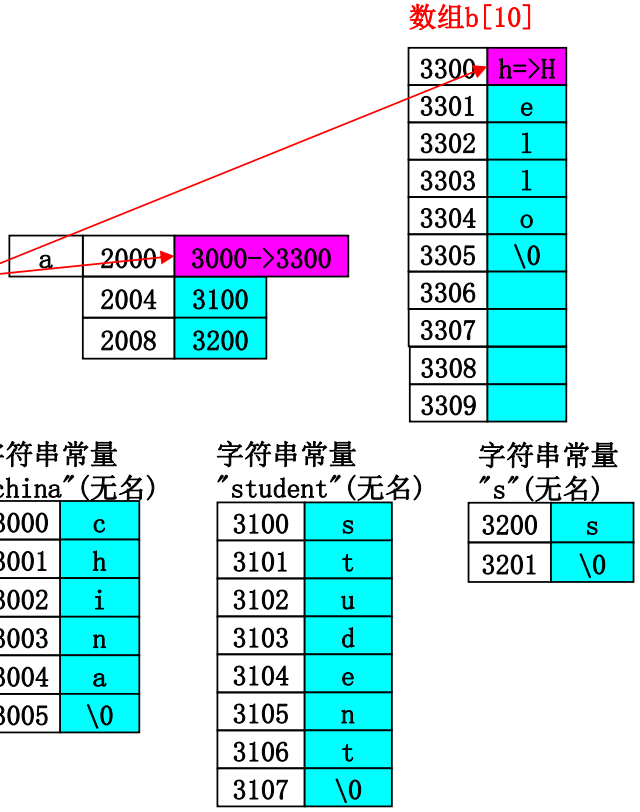
8.1. 指针数组

★ 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

```
int main()
{
    char *a[3]= {"china","student","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
    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 : error
Dev: warning



§12. 指针进阶

8. 指针数组和指向指针的指针

8.1. 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

```
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;
    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

8. 指针数组和指向指针的指针

8.1. 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

```
//例：字符串进行选择排序（指针数组形式）
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",
    "FORTRAN",
    "C++",
    …
    };
```

对比并体会差异

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）
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",
    "FORTRAN",
    "C++",
    …
    };
```

§12. 指针进阶

8. 指针数组和指向指针的指针

8.1. 指针数组

8.2. 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;
}
```

Dev可不要(有warning)

```
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

8. 指针数组和指向指针的指针

8.2. 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)

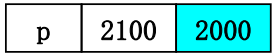
*(p+1) : 取a[1]的值3100(字符串"student"的首地址)

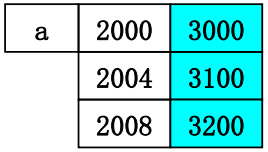
*p++ : 取a[0]的值3000，p指向a[1](地址2004)

(*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')





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

9. const指针

9.1. 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

§12. 指针进阶

9. const指针

9.2. 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    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

9. const指针

9.2. 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

```
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：普通指针　　实参：普通变量地址/指针　对

　　常变量地址/指针　　错

形参：常变量指针　实参：普通变量地址/指针　对

　　常变量地址/指针　　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

9. const指针

9.3. 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

9. const指针

9.3. 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;
```

问：如何解决?
答：见9.4

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

§12. 指针进阶

9. const指针

9.2. 指向常量的指针变量

9.3. 常指针

9.4. 指向常量的常指针(同时满足9.2+9.3)

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

9. const指针

9.1. 共用数据的保护

9.2. 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

9.3. 常指针

形式：数据类型 *const 指针变量名

9.4. 指向常量的常指针(同时满足9.2+9.3)

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?