

## § 6. 善于使用指针与引用

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

### 6.1. 基本概念

#### ★ 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:  $0 \sim 2^{16}-1$  (64KB)

32位:  $0 \sim 2^{32}-1$  (4GB)

64位:  $0 \sim 2^{64}-1$  (16EB)

#### ★ 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

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

#### ★ 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)

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

存放地址

存放值

## § 6. 善于使用指针与引用

### 6.2. 变量与指针

#### 6.2.1. 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
    cout << sizeof(short *) << endl; 4
    cout << sizeof(int *)   << endl; 4
    cout << sizeof(long *)  << endl; 4
    cout << sizeof(float *) << endl; 4
    cout << sizeof(double *) << endl; 4
    return 0;
}
```

```
#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

## § 6. 善于使用指针与引用

### 6.2. 变量与指针

#### 6.2.2. 使用

变量名 = 地址

\*变量名 = 值

short i, \*p;

long t, \*q;

p	???	3000
		3003

i	???	2000
		2001

q	???	4000
		4003

t	???	2100
		2103

p=&i;

q=&t;

p	2000	3000
		3003

i	???	2000
		2001

q	2100	4000
		4003

t	???	2100
		2103

\*p=10 ⇔ i=10

\*q=10 ⇔ t=10

p	2000	3000
		3003

i	10	2000
		2001

q	2100	4000
		4003

t	10	2100
		2103

★ 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

## § 6. 善于使用指针与引用

### 6.2. 变量与指针

#### 6.2.3. &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;

# § 6. 善于使用指针与引用

## 6. 2. 变量与指针

### 6. 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?

假设地址A  
=地址A+2 2字节  
假设地址B  
=地址B+1

## § 6. 善于使用指针与引用

### 6.2. 变量与指针

#### 6.2.4. 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; × 不允许

void \*p; ✓

p++; × 因为不知道基类型的大小

p--; ×

★ \*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++): 先取p所指的值(i), p再++(不指向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.162 例6.2

```

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	???	3000
		3003

a	???	2000
		2003

p2	???	3100
		3103

b	???	2100
		2103

p	???	3200
		3203

带地址的图示法

p1	???
----	-----

a	???
---	-----

p2	???
----	-----

b	???
---	-----

p	???
---	-----

不带具体地址的图示法  
(教科书、后续课程)

# P.162 例6.2

```

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	???	3000 3003
----	-----	--------------

p2	???	3100 3103
----	-----	--------------

p	???	3200 3203
---	-----	--------------

a	45	2000 2003
---	----	--------------

b	78	2100 2103
---	----	--------------

带地址的图示法

p1	???
----	-----

p2	???
----	-----

p	???
---	-----

a	45
---	----

b	78
---	----

不带具体地址的图示法  
(教科书、后续课程)



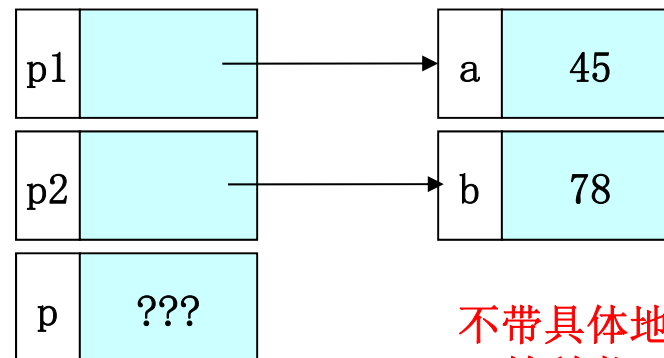
# P.162 例6.2

```

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	2000	3000 3003	a	45	2000 2003
p2	2100	3100 3103	b	78	2100 2103
p	???	3200 3203	带地址的图示法		



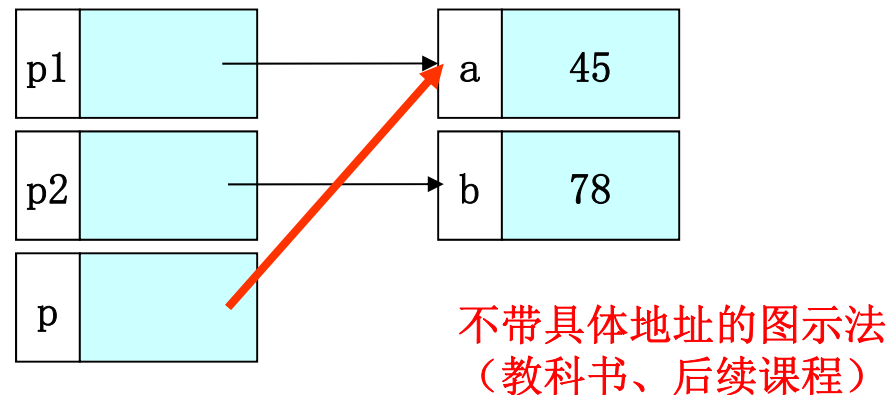
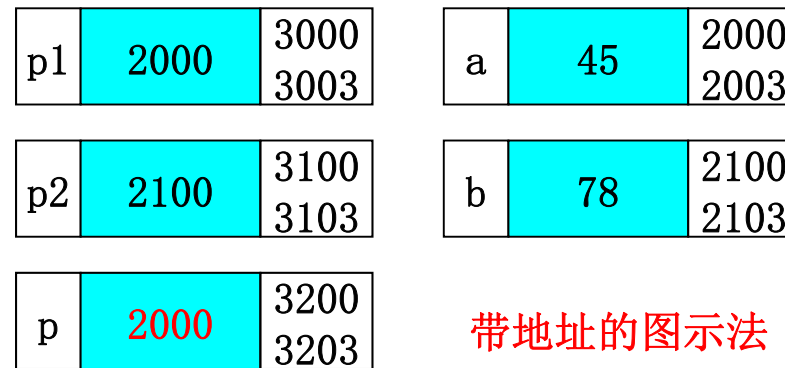
不带具体地址的图示法  
(教科书、后续课程)

# P.162 例6.2

```

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 << ...
}

```

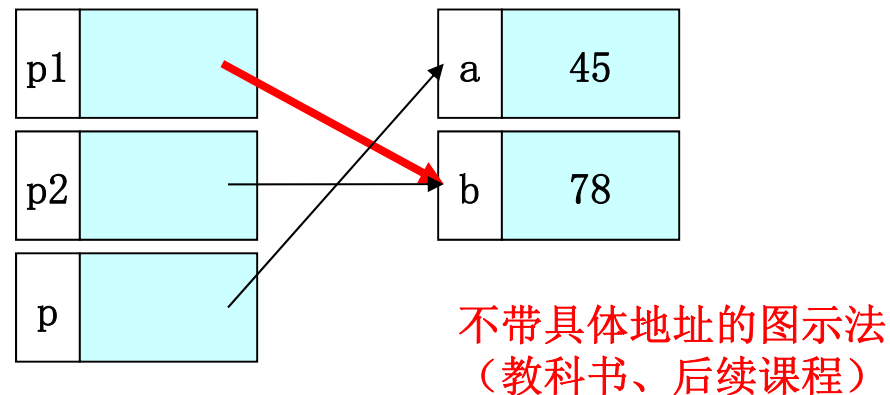
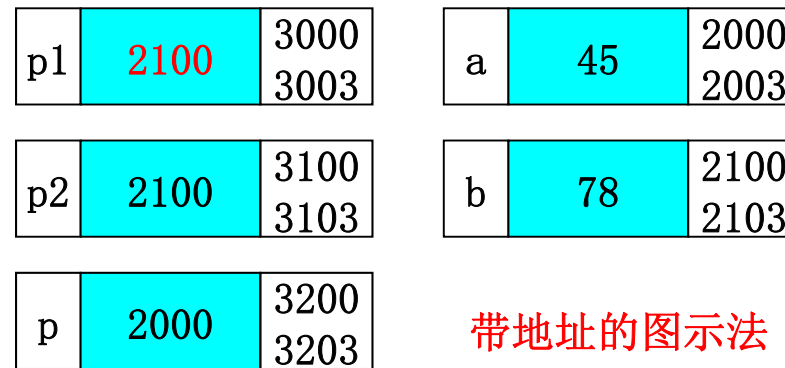


# P.162 例6.2

```

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 << ...
}

```



# P.162 例6.2

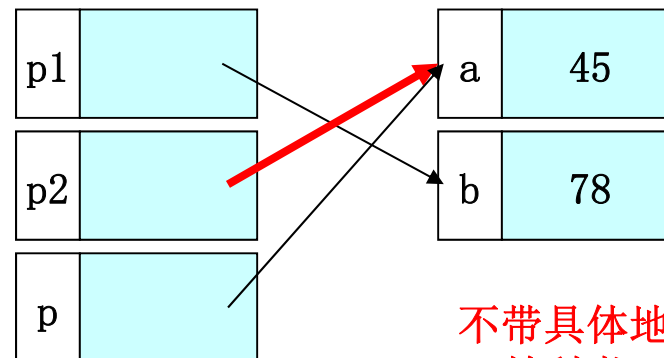
```

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	2100	3000 3003	a	45	2000 2003
p2	2100	3100 3103	b	78	2100 2103
p	2000	3200 3203			

带地址的图示法



不带具体地址的图示法  
(教科书、后续课程)

P.162 例6.2

```
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 << ...
```

```
}
```

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;

## § 6. 善于使用指针与引用

### 6.2. 变量与指针

#### 6.2.5. 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    匹配  
形参：整型简单变量

为什么无法交换？

## § 6. 善于使用指针与引用

### 6.2. 变量与指针

#### 6.2.5. 指针变量作函数的参数

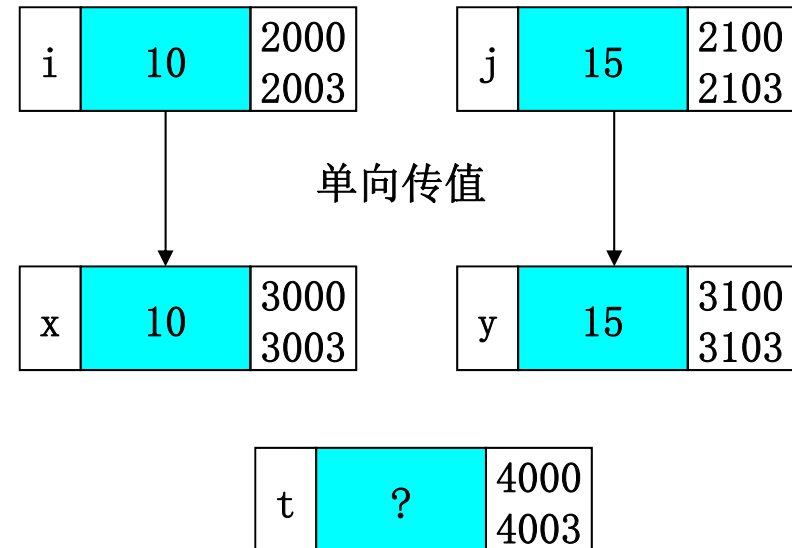
例：编写一个函数，将两个整数进行交换后输出

```
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？

## § 6. 善于使用指针与引用

### 6.2. 变量与指针

#### 6.2.5. 指针变量作函数的参数

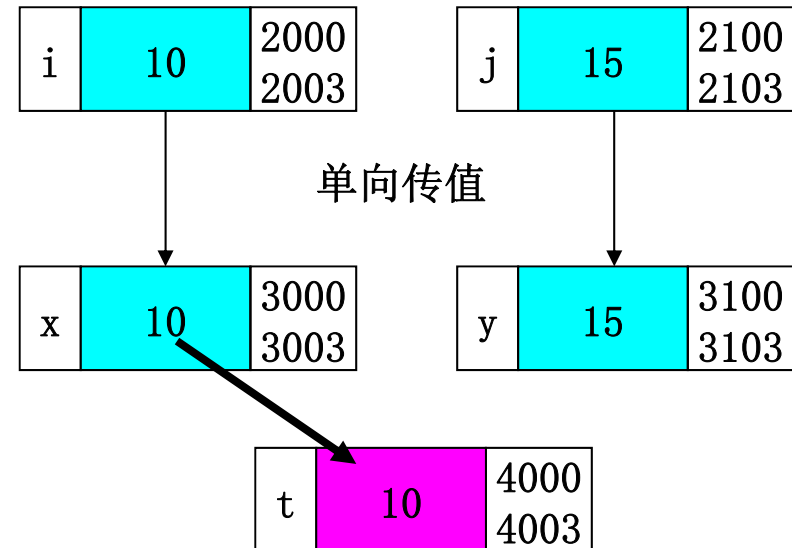
例：编写一个函数，将两个整数进行交换后输出

```
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？



## § 6. 善于使用指针与引用

### 6.2. 变量与指针

#### 6.2.5. 指针变量作函数的参数

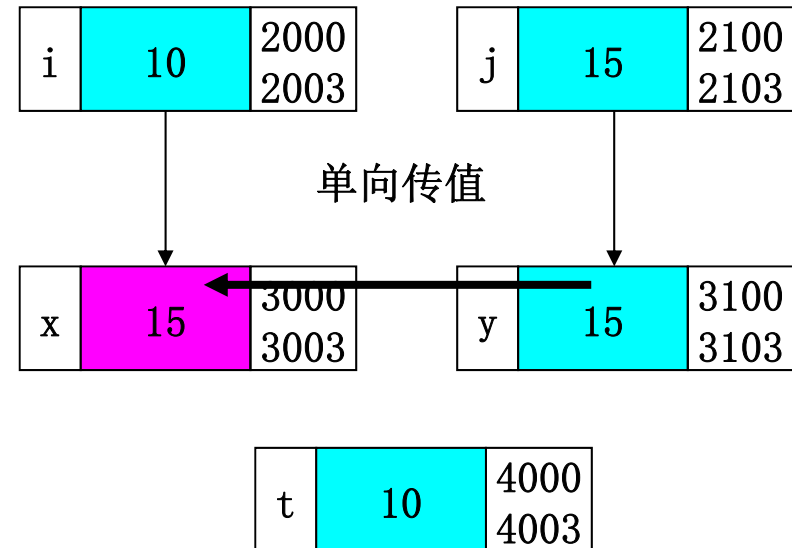
例：编写一个函数，将两个整数进行交换后输出

```
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？

## § 6. 善于使用指针与引用

### 6.2. 变量与指针

#### 6.2.5. 指针变量作函数的参数

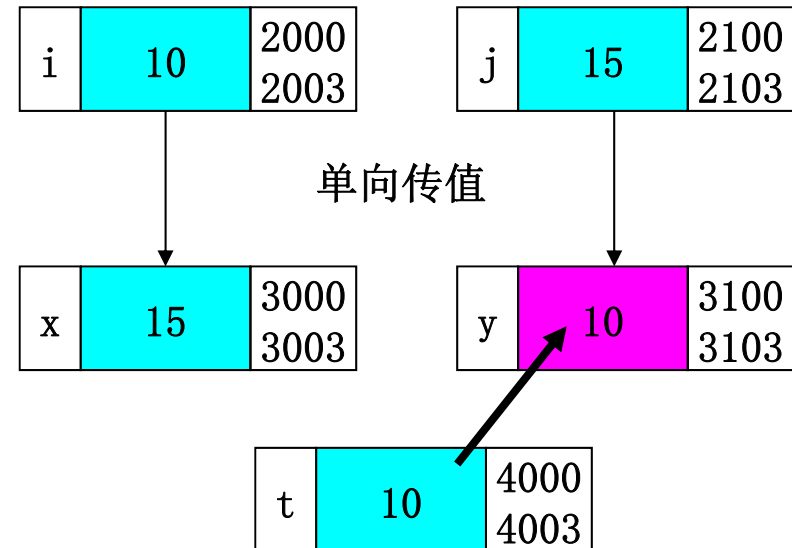
例：编写一个函数，将两个整数进行交换后输出

```
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？

## § 6. 善于使用指针与引用

### 6.2. 变量与指针

#### 6.2.5. 指针变量作函数的参数

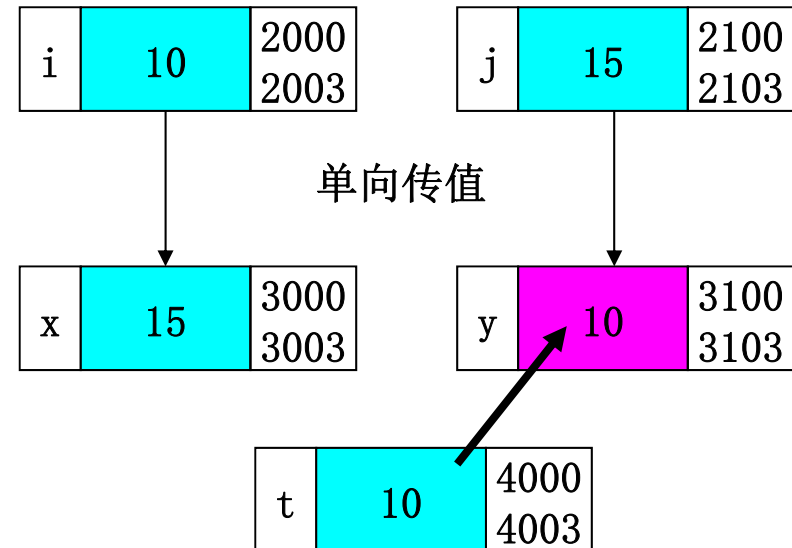
例：编写一个函数，将两个整数进行交换后输出

```
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

## § 6. 善于使用指针与引用

### 6.2. 变量与指针

#### 6.2.5. 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;  
}
```

实参：整型变量地址    匹配  
形参：整型指针变量

```
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

## § 6. 善于使用指针与引用

### 6.2. 变量与指针

#### 6.2.5. 指针变量作函数的参数

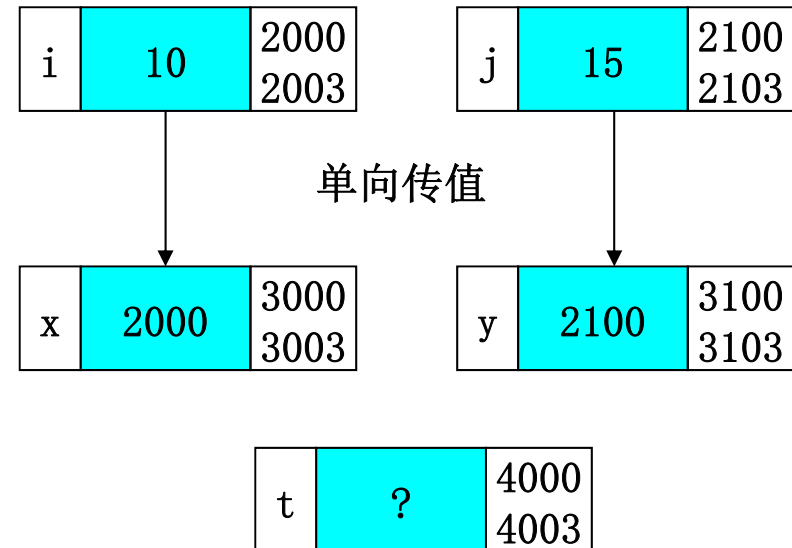
例：编写一个函数，将两个整数进行交换后输出

```
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

## § 6. 善于使用指针与引用

### 6.2. 变量与指针

#### 6.2.5. 指针变量作函数的参数

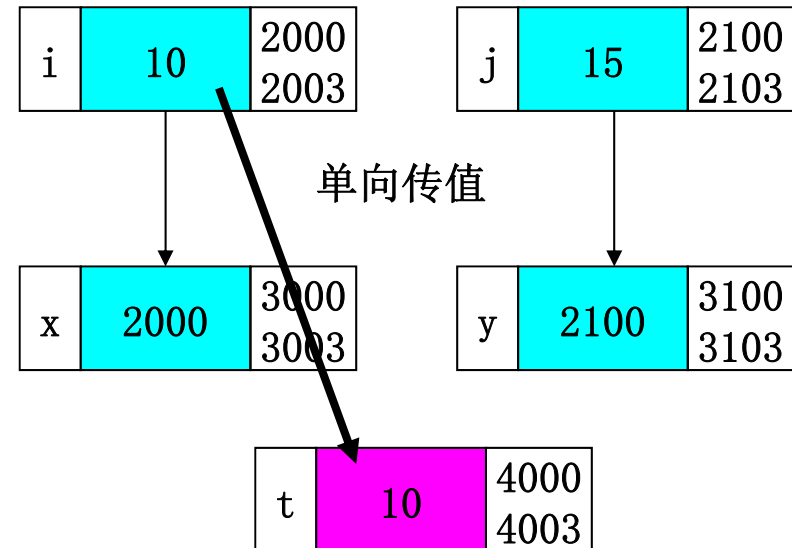
例：编写一个函数，将两个整数进行交换后输出

```
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

## § 6. 善于使用指针与引用

### 6.2. 变量与指针

#### 6.2.5. 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)
```

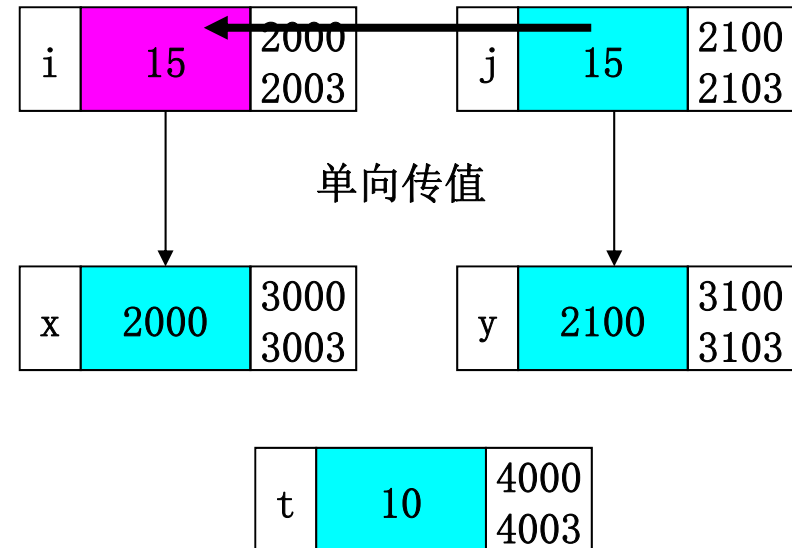
```
{    int t;  
    t = *x;  
    *x = *y;  
    *y = t;
```

```
}
```

```
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



## § 6. 善于使用指针与引用

### 6.2. 变量与指针

#### 6.2.5. 指针变量作函数的参数

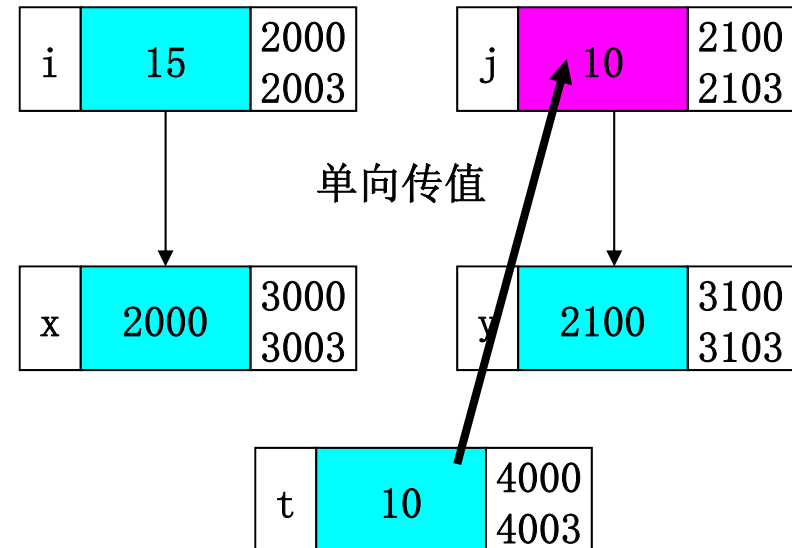
例：编写一个函数，将两个整数进行交换后输出

```
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  
}
```



★ 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



## § 6. 善于使用指针与引用

### 6.2. 变量与指针

#### 6.2.5. 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;  
}
```

实参：整型指针变量    匹配  
形参：整型指针变量

```
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  
}
```

正确的方法

与swap(&i, &j) 等价

## § 6. 善于使用指针与引用

### 6.2. 变量与指针

#### 6.2.5. 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
}
```

## § 6. 善于使用指针与引用

### 6.2. 变量与指针

#### 6.2.5. 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

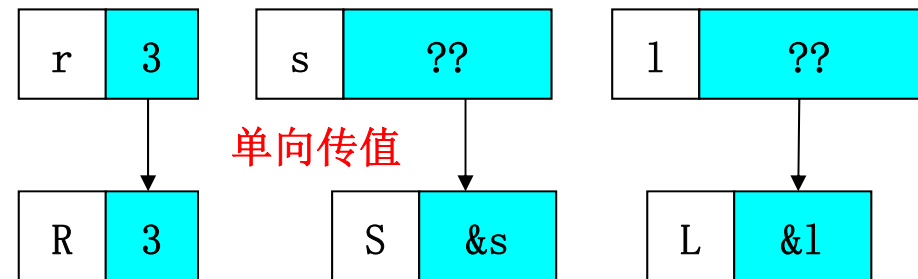
★ 指针变量做参数，可以同时得到多个改变的实参值，从而达到一个函数返回多个值的目的

```
#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

## § 6. 善于使用指针与引用

### 6.2. 变量与指针

#### 6.2.5. 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

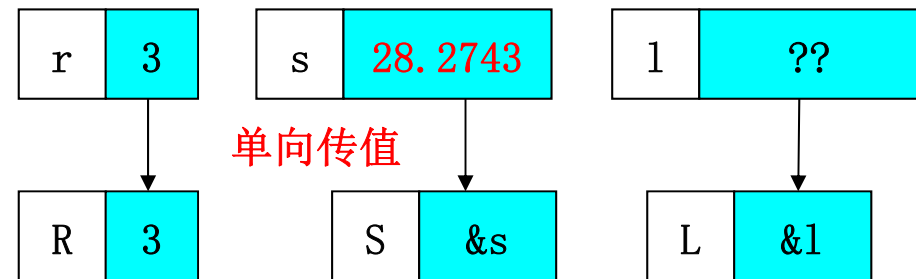
★ 指针变量做参数，可以同时得到多个改变的实参值，从而达到一个函数返回多个值的目的

```
#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
}
```



## § 6. 善于使用指针与引用

### 6.2. 变量与指针

#### 6.2.5. 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

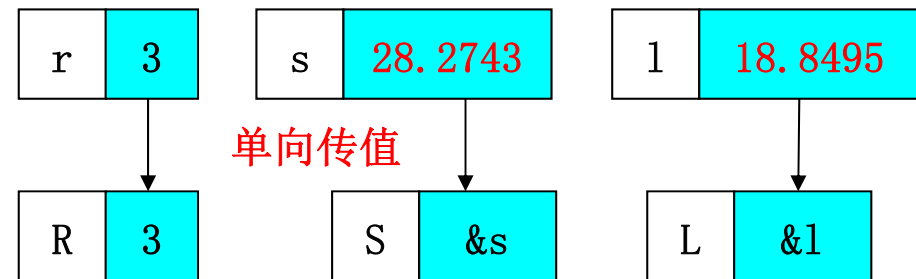
★ 指针变量做参数，可以同时得到多个改变的实参值，从而达到一个函数返回多个值的目的

```
#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
}
```



## § 6. 善于使用指针与引用

### 6.2. 变量与指针

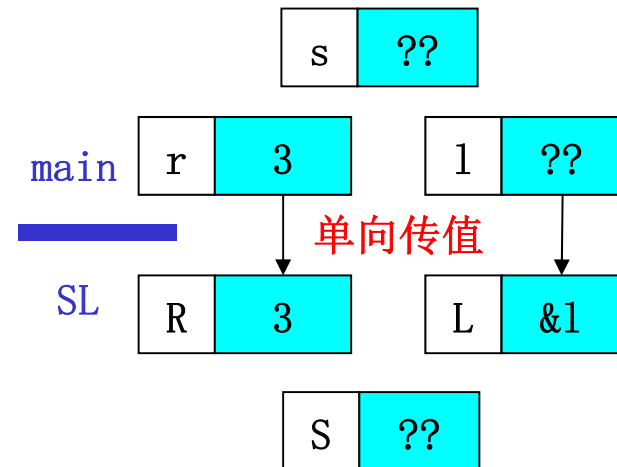
#### 6.2.5. 指针变量作函数的参数

★ 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  
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

## § 6. 善于使用指针与引用

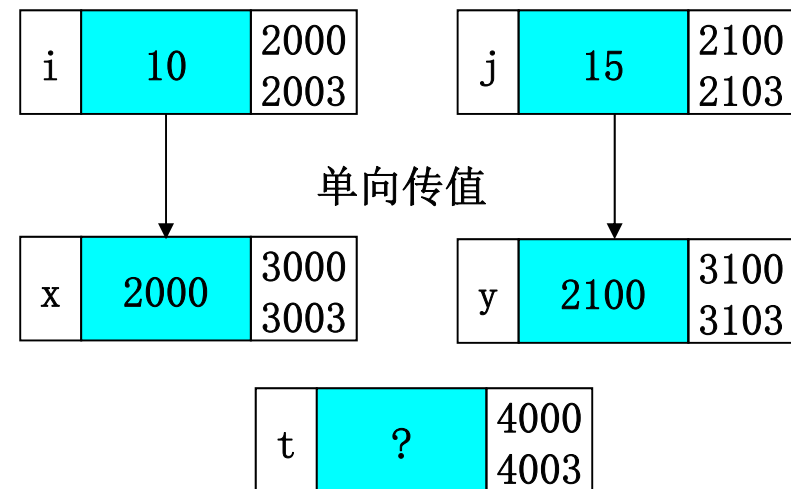
### 6.2. 变量与指针

#### 6.2.5. 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;
    swap(&i, &j);
    cout << "i=" << i << " j=" << j << endl;
}
```



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

## § 6. 善于使用指针与引用

### 6.2. 变量与指针

#### 6.2.5. 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;
}
```

VS2015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?

```
int main()
{
```

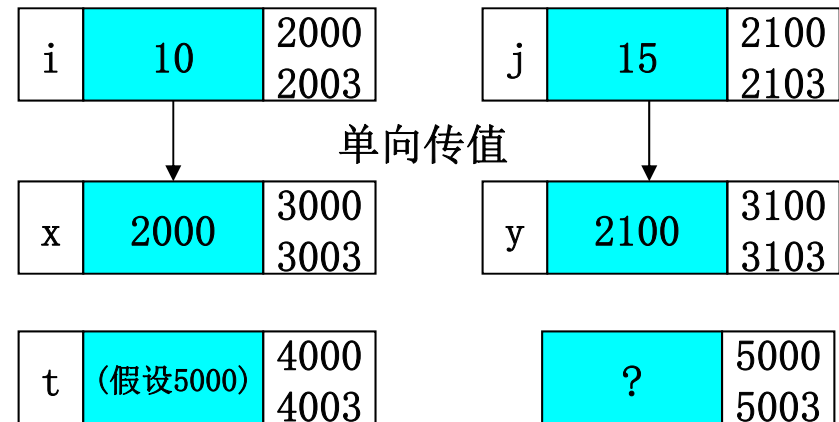
```
    int i=10, j=15;
```

```
    cout << "i=" << i << " j=" << j << endl;
```

```
    swap(&i, &j);
```

```
    cout << "i=" << i << " j=" << j << endl;
```

```
}
```



提示: 5000-5003系统  
是否分配给了程序?

i=10 j=15

i=15 j=10  
或 死机或其它非正常现象



## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.1. 基本概念

数组的指针：数组的起始地址 ( $\Leftrightarrow \&a[0]$ )

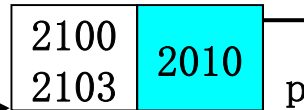
数组元素的指针：数组中某个元素的地址

#### 6.3.2. 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



2000	0
2001	
2002	1
2003	
2004	2
2005	
2006	3
2007	
2008	4
2009	
2010	5
2011	
...	
2019	

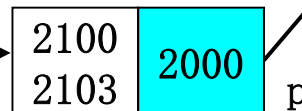
#### 6.3.3. 指向数组的指针变量

```
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

2000	0
2001	
2002	
2003	
2004	
2005	
2006	
2007	
2008	
2009	
2010	
2011	
...	
2019	

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.1. 基本概念

数组的指针：数组的起始地址 ( $\Leftrightarrow \&a[0]$ )

数组元素的指针：数组中某个元素的地址

#### 6.3.2. 指向数组元素的指针变量

#### 6.3.3. 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

$\left\{ \begin{array}{l} p = \&a[0]: \text{数组的第0个元素的地址就是数组的起始地址} \\ p = a : \text{数组名代表首地址} \end{array} \right.$

★ 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( $\text{sizeof}(\text{数组名})/\text{sizeof}(\text{指针})$ 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
}
```

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.4. 指针法引用数组元素

##### 6.3.4.1. 形式

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

```
p=&a[5];
```

```
*p=10 ⇔ a[5]=10
```

指针法    下标法

##### 6.3.4.2. 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

p	2012	3000 3003
---	------	--------------

0	2000 2003
1	2004 2007
2	2008 2011
3	2012 2015
4	2016 2019
5	2020 2023
6	2024 2027
7	2028 2031
8	2032 2035
9	2036 2039

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

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.4. 指针法引用数组元素

##### 6.3.4.2. 下标法与指针法的区别

#### ★ 常见用法与错误 P. 169-170 例6.5及变化

P. 169 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. 169 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. 169 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. 169 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;
}
```

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.4. 指针法引用数组元素

##### 6.3.4.2. 下标法与指针法的区别

#### ★ 常见用法与错误 P. 169-170 例6.5及变化

P. 169 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. 169 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. 169 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  $p+i*\text{sizeof}(\text{int})$

本程序: 只要  $p+\text{sizeof}(\text{int})$

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.4. 指针法引用数组元素

##### 6.3.4.2. 下标法与指针法的区别

#### ★ 常见用法与错误 P. 169-170 例6.5及变化

P. 169 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. 169 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

$p-a < 10 \Leftrightarrow p < a+10$   
表示p和a是否相差10个  
int型的元素

P. 169 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

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.4. 指针法引用数组元素

##### 6.3.4.3. 指向数组的指针变量的运算

#### A. 指针变量 $\pm$ 整数（包括++/--）

指针变量++  $\Leftrightarrow$  所指地址 += sizeof(基类型)

指针变量--  $\Leftrightarrow$  所指地址 -= sizeof(基类型)

指针变量+n  $\Leftrightarrow$  所指地址 + n\*sizeof(基类型)

指针变量-n  $\Leftrightarrow$  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

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.4. 指针法引用数组元素

##### 6.3.4.3. 指向数组的指针变量的运算

###### A. 指针变量 $\pm$ 整数 (包括++/--)

指针变量++  $\Leftrightarrow$  所指地址 += sizeof(基类型)

指针变量--  $\Leftrightarrow$  所指地址 -= sizeof(基类型)

指针变量+n  $\Leftrightarrow$  所指地址 + n\*sizeof(基类型)

指针变量-n  $\Leftrightarrow$  所指地址 - n\*sizeof(基类型)

★ 若指针变量指向数组，则 $\pm n$ 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
```



## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.4. 指针法引用数组元素

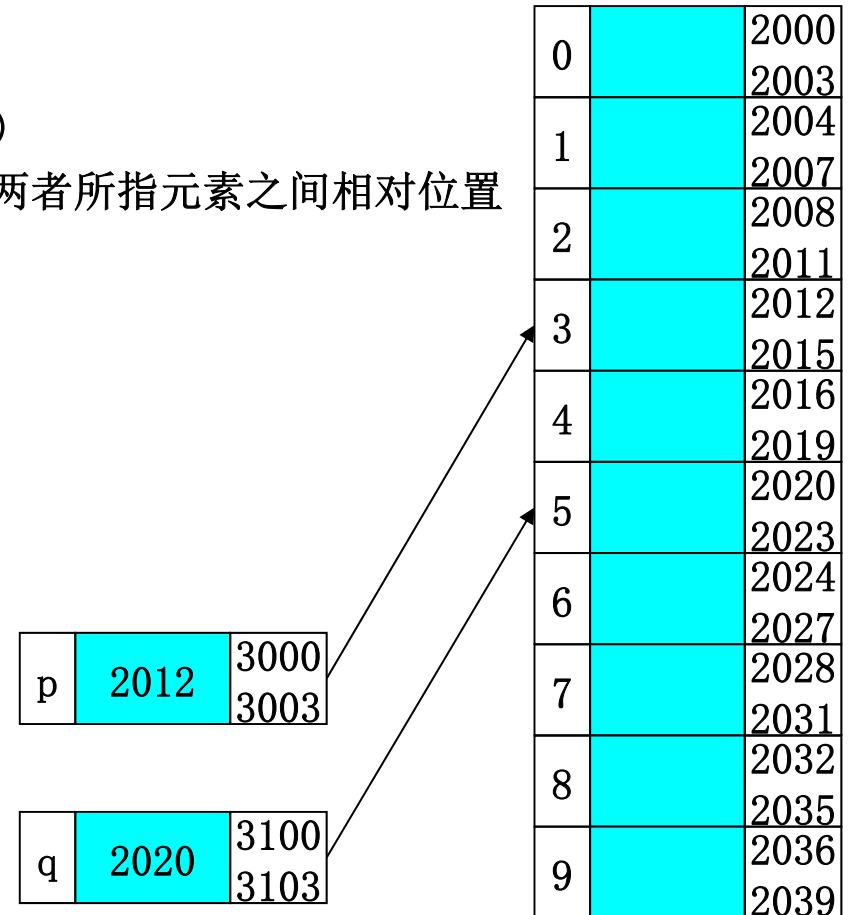
##### 6.3.4.3. 指向数组的指针变量的运算

#### 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  $\Leftrightarrow$  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
}
```



## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.4. 指针法引用数组元素

##### 6.3.4.3. 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  $\Leftrightarrow$  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

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.4. 指针法引用数组元素

##### 6.3.4.3. 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)

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.4. 指针法引用数组元素

##### 6.3.4.3. 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;
}
```

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.4. 指针法引用数组元素

##### 6.3.4.3. 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）

★ 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)
}
```

```
#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;
}
```

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.4. 指针法引用数组元素

##### 6.3.4.4. 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++)`:  $\Leftrightarrow$  `*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++

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.4. 指针法引用数组元素

#### 6.3.5. 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

本质都是指针变量

<pre>main() { int a[10];   ...   fun(a);   ... }</pre>	<pre>fun(int x[10]) {   int x[123]   ...   ... }</pre>	<pre>fun(int x[]) {   ...   ... }</pre>	<pre>fun(int *x) {   ...   ... }</pre>
--	--	---	--

实参是数组名，传入数组的首地址

形参是数组名(带大小)

形参是数组名(不带大小)

形参是指针变量

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第6章会解释)

```
//第5章中的例子
#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);
}
```

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.4. 指针法引用数组元素

#### 6.3.5. 用指针变量作函数参数接收数组地址

P. 170-171 例6.6 (和P. 139例5.7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;
    }
}
```

P. 139的写法

```
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P. 139

```
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

P. 171的写法

```
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;
    }
}
```

形参是指针变量  
其余同P. 139



## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.4. 指针法引用数组元素

#### 6.3.5. 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
    ...
}
```

x

2100	2000
2103	

2000+5\*sizeof(int)

a

2000	
2001	
2002	
2003	
...	
2020	15
2021	
2022	
2023	
...	
2036	
2037	
2038	
2039	

★ 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);
    ...
}
```

## § 6. 善于使用指针与引用

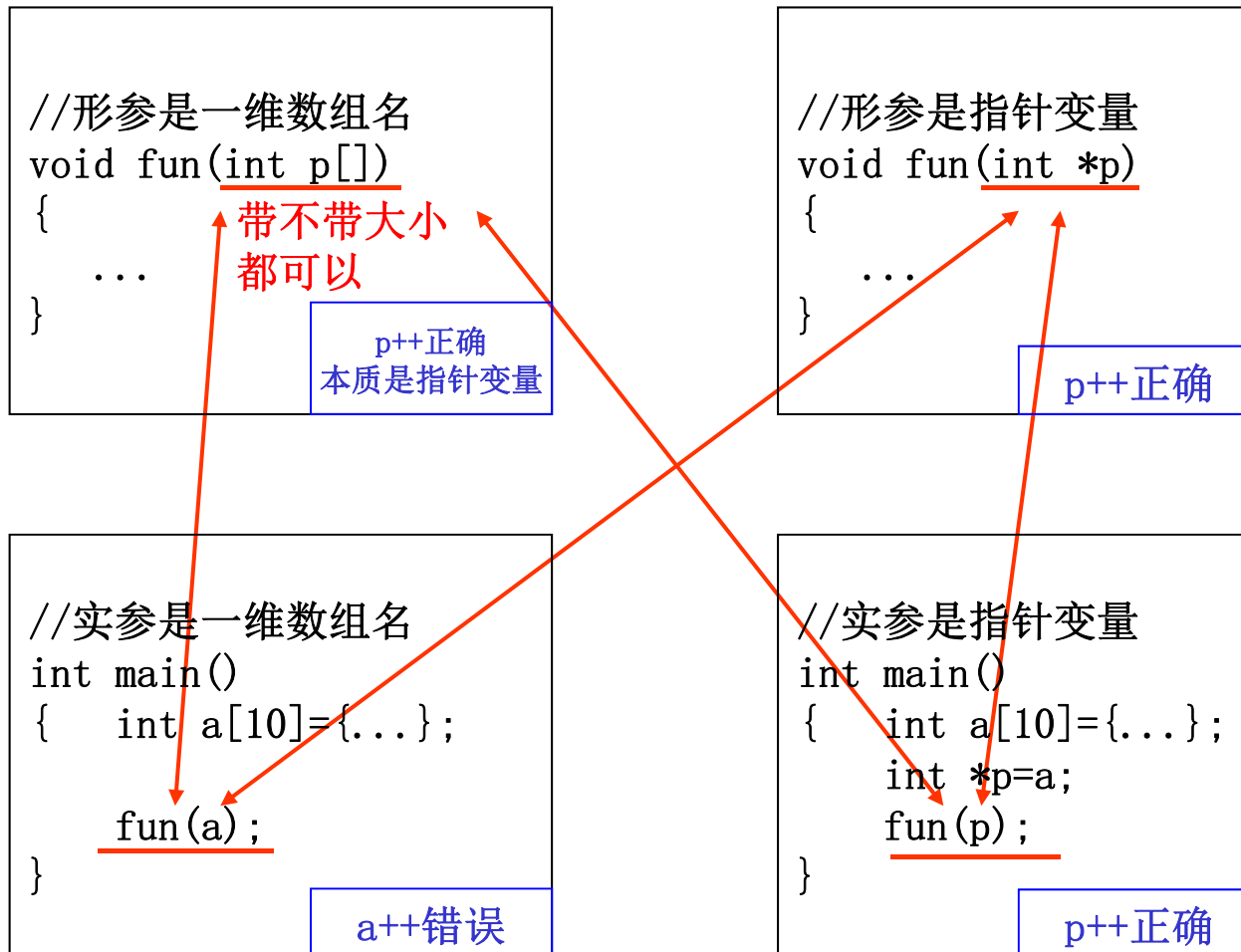
### 6.3. 数组与指针

#### 6.3.4. 指针法引用数组元素

#### 6.3.5. 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

#### P. 172 实参/形参的四种组合



## § 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

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.6. 多维数组与指针 (补充, 极其重要!!!)

##### 6.3.6.1. 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 ( $\Leftrightarrow \&a[0]$ )

由等价关系  $a[i] \Leftrightarrow *(a+i)$  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)

$\&a[i] \Leftrightarrow a+i$  地址

$a[i] \Leftrightarrow *(a+i)$  值

第0个元素的特殊表示:

$a[0] \Leftrightarrow *(a+0) \Leftrightarrow *a$

$\&a[0] \Leftrightarrow a+0 \Leftrightarrow a$

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]

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.6. 多维数组与指针 (补充, 极其重要!!!)

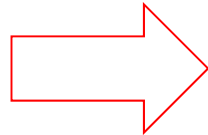
##### 6.3.6.1. 二维数组的地址

二维数组:

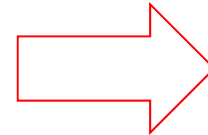
```
int a[3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

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

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]



a	2000	1	a[0]
		2	
		3	
		4	
	2016	5	a[1]
		6	
		7	
		8	
	2032	9	a[2]
		10	
		11	
		12	



a	2000	1	a[0][0]
		2	[1]
		3	[2]
		4	[3]
	2016	5	a[1][0]
		6	[1]
		7	[2]
		8	[3]
	2032	9	a[2][0]
		10	[1]
		11	[2]
		12	[3]

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]

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.6. 多维数组与指针 (补充, 极其重要!!!)

##### 6.3.6.1. 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)

```
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

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

元素是指  
4元素一维数组

行地址

元素  
地址

i:0-2(行)

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.6. 多维数组与指针 (补充, 极其重要!!!)

##### 6.3.6.1. 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

$a[i][j]$ : 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:

$a[i][j] \Leftrightarrow *(a[i]+j) \Leftrightarrow (*(a+i)+j)$  值  
 $\&a[i][j] \Leftrightarrow a[i]+j \Leftrightarrow *(a+i)+j$  元素地址

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

## § 6. 善于使用指针与引用

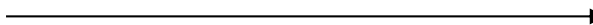
### 6.3. 数组与指针

#### 6.3.6. 多维数组与指针 (补充, 极其重要!!!)

##### 6.3.6.1. 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  $a+i*\text{sizeof}(\text{基类型})$  

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  $a+i*n*\text{sizeof}(\text{基类型})$

$a[i]+j$  实际  $a+(i*n+j)*\text{sizeof}(\text{基})$

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]	
<hr/>				
	2016	5	a[1][0]	
	2020	6	[1]	
	2024	7	[2]	
	2028	8	[3]	
<hr/>				
	2032	9	a[2][0]	
	2036	10	[1]	
	2040	11	[2]	
	2044	12	[3]	



## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.6. 多维数组与指针 (补充, 极其重要!!!)

##### 6.3.6.1. 二维数组的地址

假设 `int a[3][4]` 存放在2000开始的48个字节中

<code>a</code>	: 地址(二维数组/第0行)	2000
<code>&amp;a[i]</code>	: 地址(第i行)	2016
<code>a+i</code>	: 地址(第i行)	2016
<code>a[i]</code>	: 地址(第i行0列)	2016
<code>*(a+i)</code>	: 地址(第i行0列)	2016
<code>&amp;a[i][j]</code>	: 地址(第i行j列)	2024
<code>a[i]+j</code>	: 地址(第i行j列)	2024
<code>*(a+i)+j</code>	: 地址(第i行j列)	2024
<code>a[i][j]</code>	: 值(第i行j列)	
<code>*(a[i]+j)</code>	: 值(第i行j列)	
<code>*(*(a+i)+j)</code>	: 值(第i行j列)	

行地址

元素  
地址

值

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

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.6. 多维数组与指针 (补充, 极其重要!!!)

##### 6.3.6.1. 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?

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.6. 多维数组与指针 (补充, 极其重要!!!)

##### 6.3.6.1. 二维数组的地址

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

实际运行一次, 观察结果并思考!!!

```
{   int a[3][4];
行 cout <<  a      << endl;  地址a
地 cout <<  (a+1)   << endl;  地址a+16
址 cout <<  (a+1)+1 << endl;  地址a+32
元 cout << *(a+1)    << endl;  地址a+16
素 cout << *(a+1)+1  << endl;  地址a+20
地 cout <<  a[2]     << endl;  地址a+32
址 cout <<  a[2]+1   << endl;  地址a+36
行 cout << &a[2]     << endl;  地址a+32
地 cout << &a[2]+1   << endl;  地址a+48(已超范围)
址 return 0;
}
```

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

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.6. 多维数组与指针 (补充, 极其重要!!!)

##### 6.3.6.1. 二维数组的地址

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

另一种验证方法!!!

```
{   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;
```

```
}
```

48

4 即&a[1], 是地址(指针)

16 指针基类型是int[4]

16 即a[1], 是数组(4元素)

4 数组元素是int

16 a[2]是数组(4元素)

4 数组元素是int

4 数组a[2]的地址(指针)

16 指针基类型是int[4]

\*&a[2] ⇔ a[2], 是数组(4元素)

数组大小

a+1大小

a+1基类型

\*(a+1)大小

\*(a+1)基类型

a[2]大小

a[2]基类型

&a[2]大小

&a[2]基类型

} 同

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.6. 多维数组与指针 (补充, 极其重要!!!)

##### 6.3.6.1. 二维数组的地址

##### 6.3.6.2. 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
}
```

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]

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]

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.6. 多维数组与指针 (补充, 极其重要!!!)

##### 6.3.6.1. 二维数组的地址

##### 6.3.6.2. 指向二维数组元素的指针变量

例: 补充资料P.2 例6.7 打印二维数组的值 (以下四种方法均正确)

```
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;
}
```

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.6. 多维数组与指针 (补充, 极其重要!!!)

##### 6.3.6.3. 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];
}
```

#### ★ 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)

```
int a[3][4]={...};
```

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

(\*p) 有4个元素

每个元素类型是int

```
int a[4]
```

a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

\*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)

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.6. 多维数组与指针 (补充, 极其重要!!!)

##### 6.3.6.3. 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;    48    数组大小
    cout << sizeof(p) << endl;    4      因为指针
    cout << sizeof(*p) << endl;    16    因为int[4]
    cout << p << endl;            地址a    行地址
    cout << p+1 << endl;          地址a+16 +1 = +16
    cout << *p << endl;           地址a    元素地址
    cout << *p+1 << endl;         地址a+4 +1 = +4
    cout << *(*p+1) << endl;      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 (*p1)[4], *p;
    for (p1=a; p1 < a+3; p1++) { //行指针
        for (p=*p1; p < *p1+4; p++) //元素指针
            cout << *p << ' ';
        cout << endl; //每行一个回车
    }
}
```



## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.6. 多维数组与指针 (补充, 极其重要!!!)

##### 6.3.6.4. 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

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.6. 多维数组与指针 (补充, 极其重要!!!)

##### 6.3.6.5. 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

★ 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参即可对应, 不关心行数)

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

```
#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int\*  
x2\_size=4 因为int\*  
x3\_size=4 因为int\*

```
#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]

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.6. 多维数组与指针 (补充, 极其重要!!!)

##### 6.3.6.5. 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)

补充资料P.4 例6.9

```
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

6.3.1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)

## § 6. 善于使用指针与引用

### 6.3. 数组与指针

#### 6.3.6. 多维数组与指针 (补充, 极其重要!!!)

##### 6.3.6.5. 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

