

## § 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.158 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.158 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	45	2000 2003
---	----	--------------

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

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

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

带地址的图示法

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

a	45
---	----

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

b	78
---	----

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

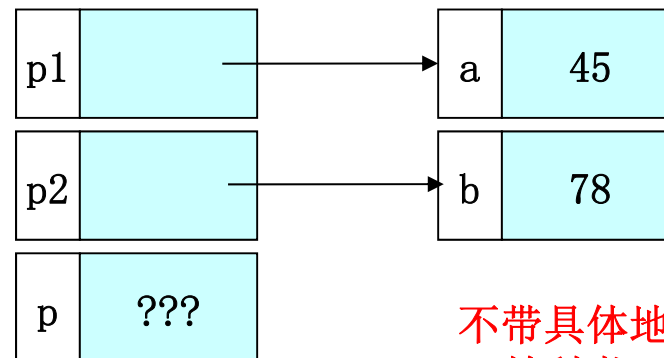
不带具体地址的图示法  
(教科书、后续课程)



P.158 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		



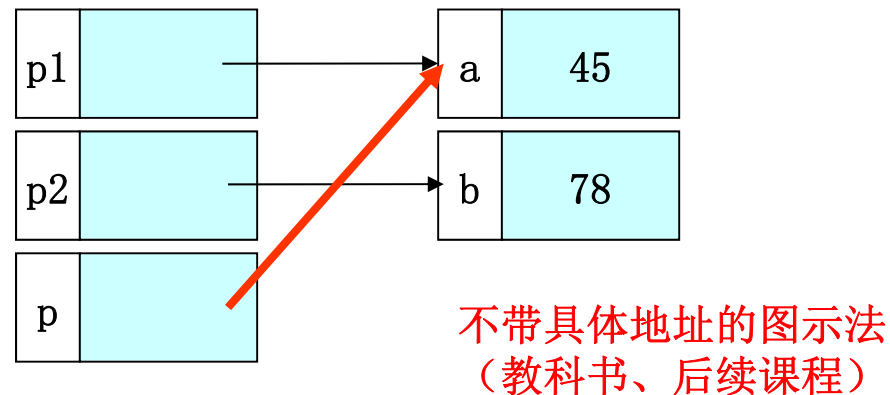
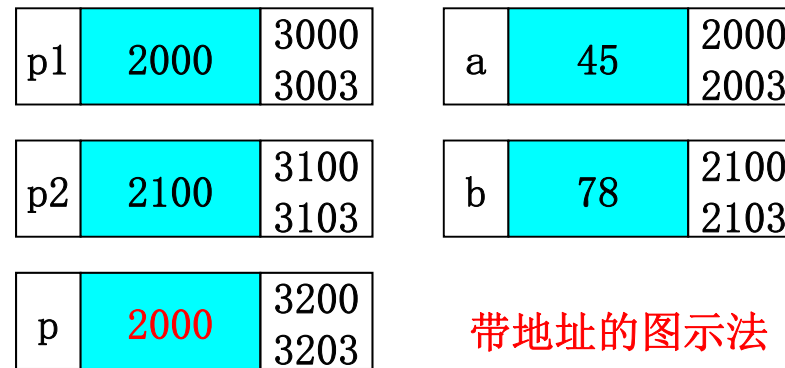
不带具体地址的图示法  
(教科书、后续课程)

# P.158 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 << ...
}

```

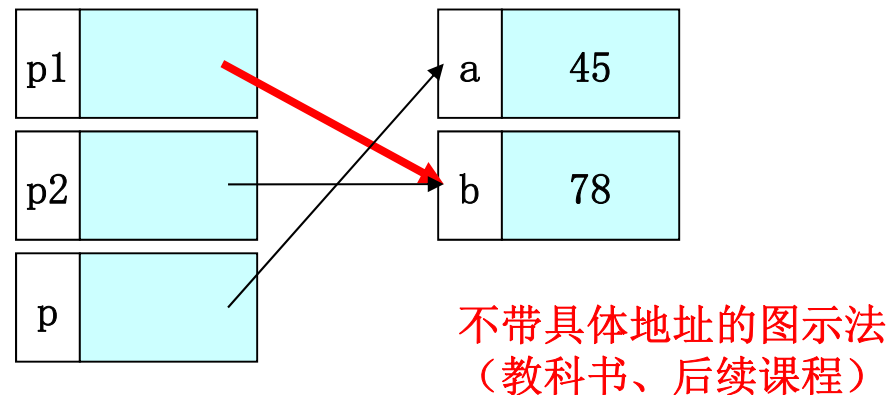
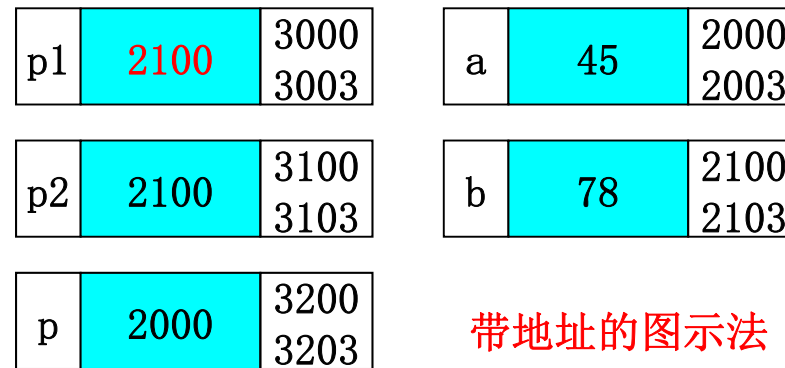


# P.158 例6.2

```

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 << ...
}

```

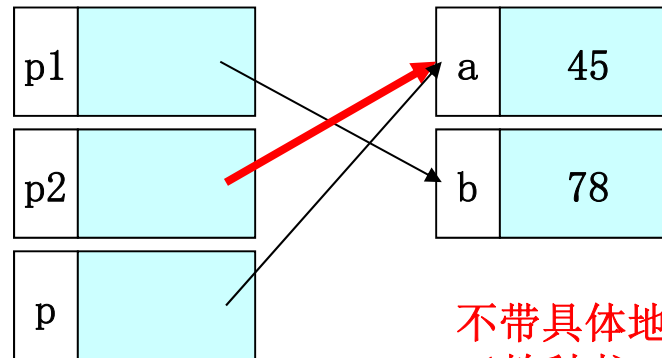


P.158 例6.2

```
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.158 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;  
}
```

实参：整型简单变量    匹配  
形参：整型简单变量

```
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=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

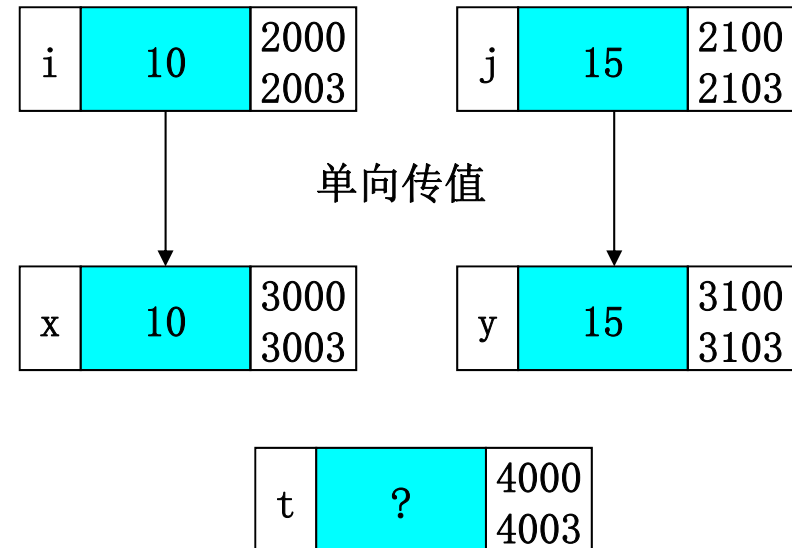
例：编写一个函数，将两个整数进行交换后输出

```
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

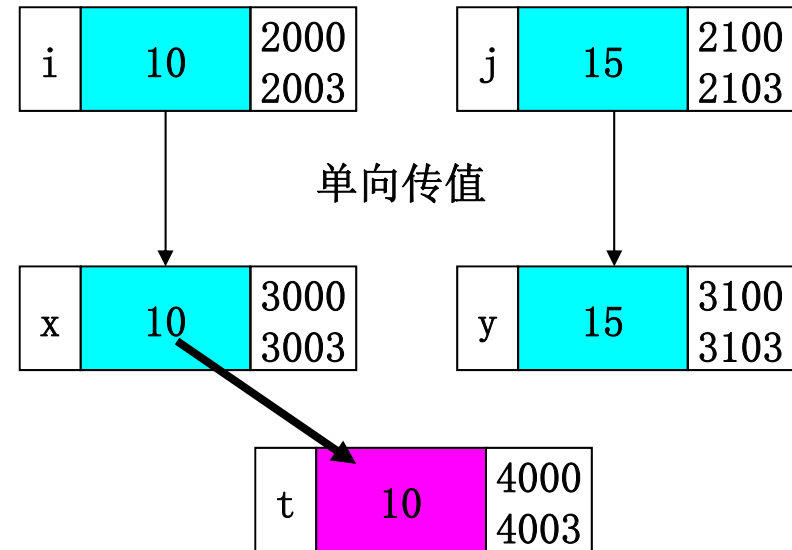
例：编写一个函数，将两个整数进行交换后输出

```
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

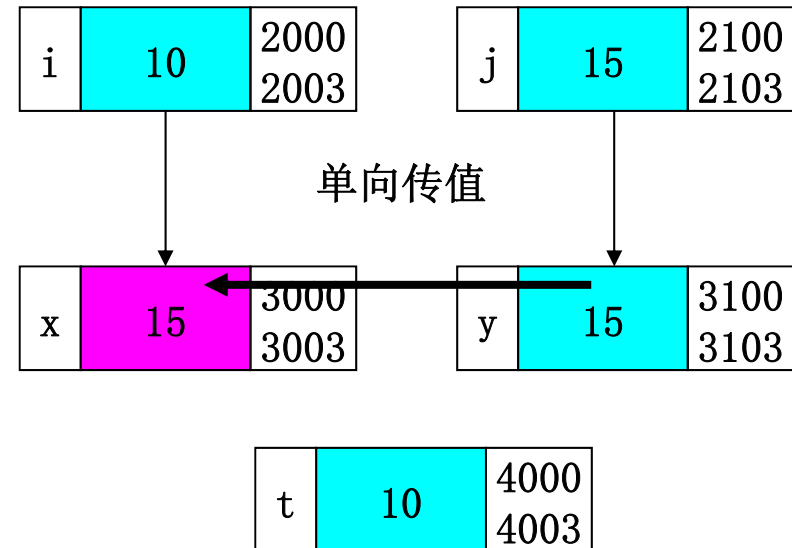
例：编写一个函数，将两个整数进行交换后输出

```
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

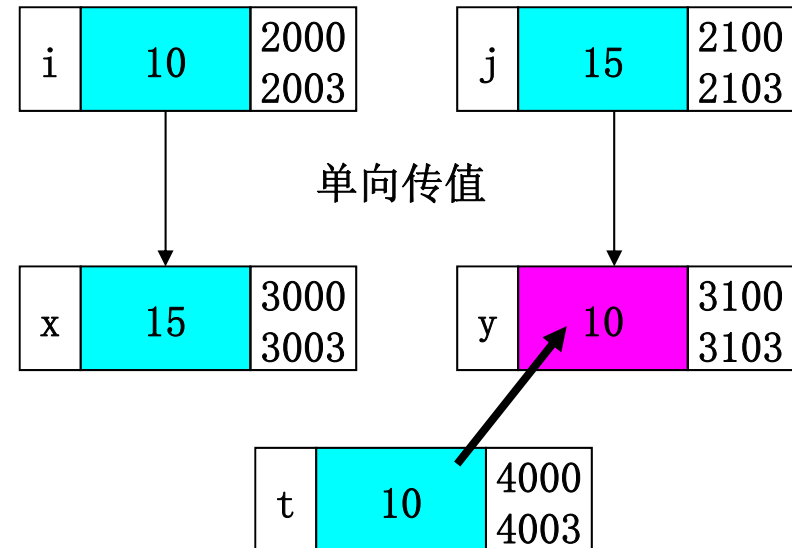
例：编写一个函数，将两个整数进行交换后输出

```
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？

## § 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

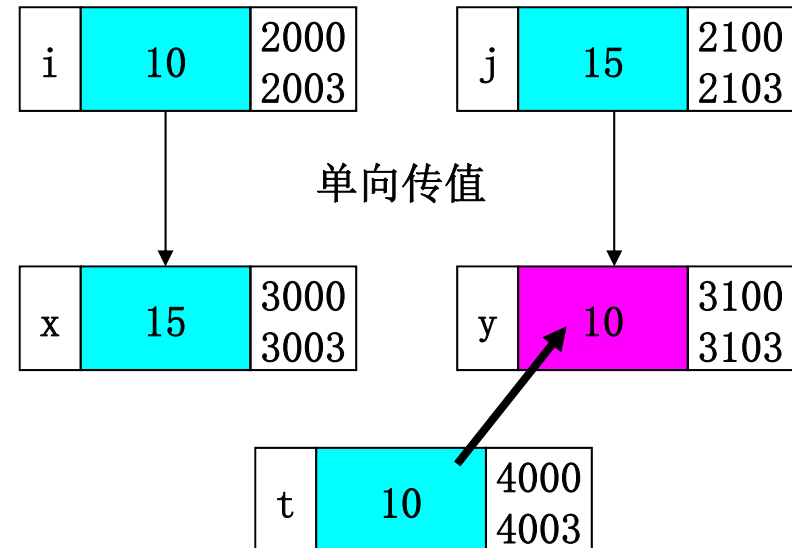
例：编写一个函数，将两个整数进行交换后输出

```
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

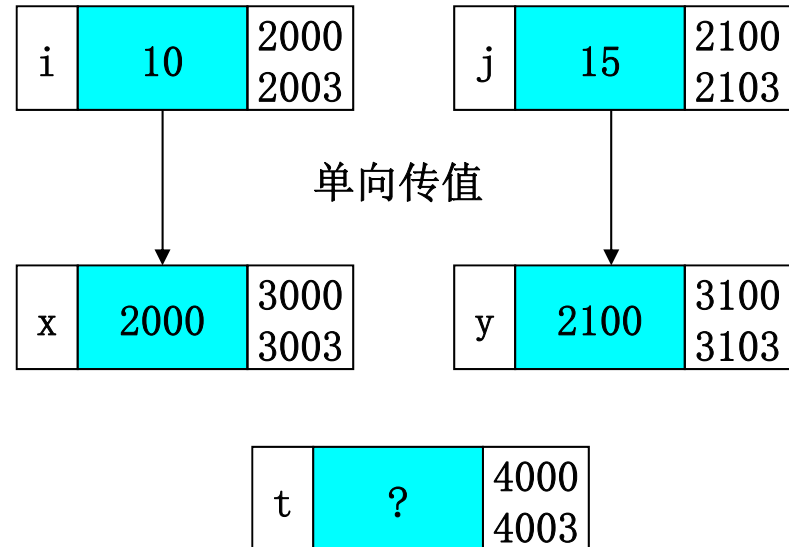
```
void swap(int *x, int *y)
```

```
{    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  
}
```

单向传值



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

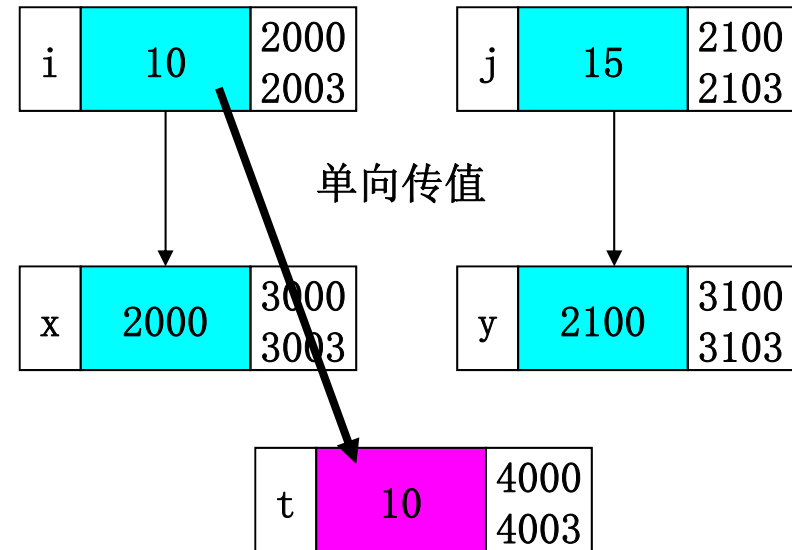
例：编写一个函数，将两个整数进行交换后输出

```
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)
```

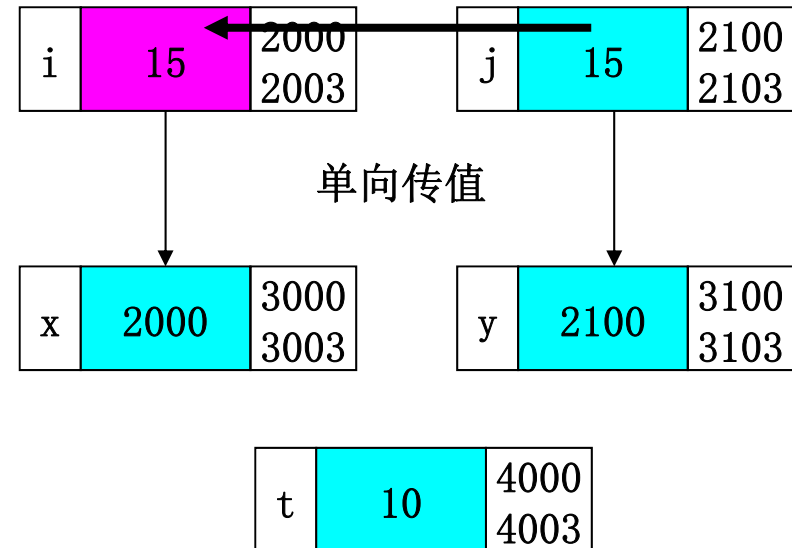
```
{    int t;  
    t = *x;  
    *x = *y;  
    *y = t;
```

```
}
```

```
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

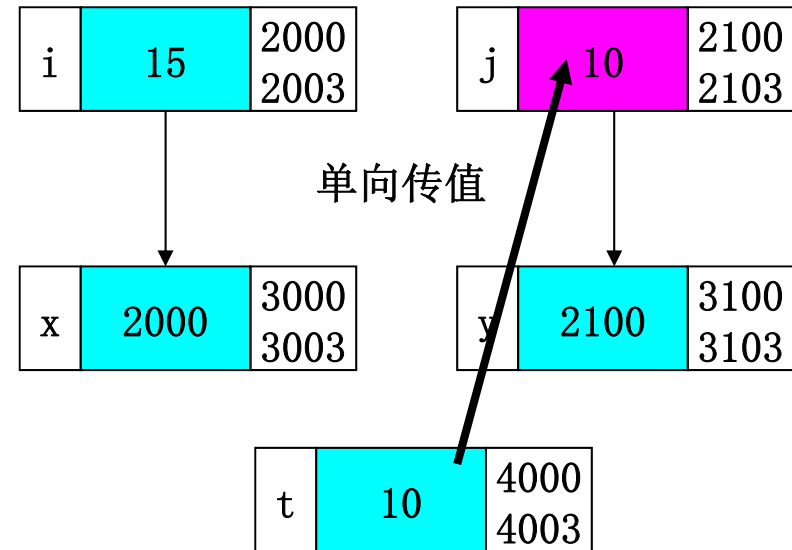
例：编写一个函数，将两个整数进行交换后输出

```
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

★ 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

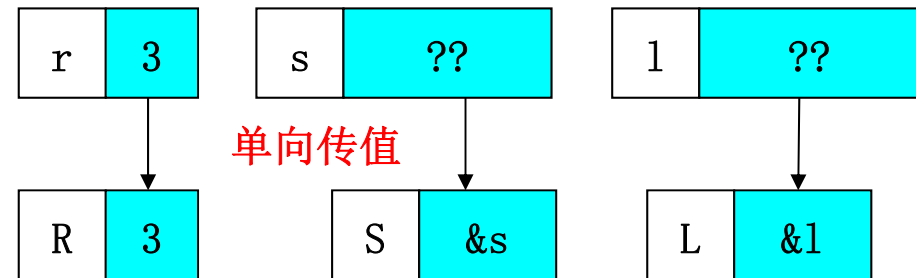
★ 指针变量做参数，可以同时得到多个改变的实参值，从而达到一个函数返回多个值的目的

```
#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

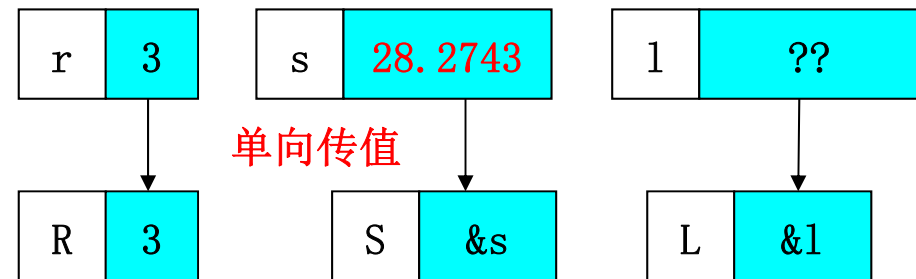
★ 指针变量做参数，可以同时得到多个改变的实参值，从而达到一个函数返回多个值的目的

```
#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

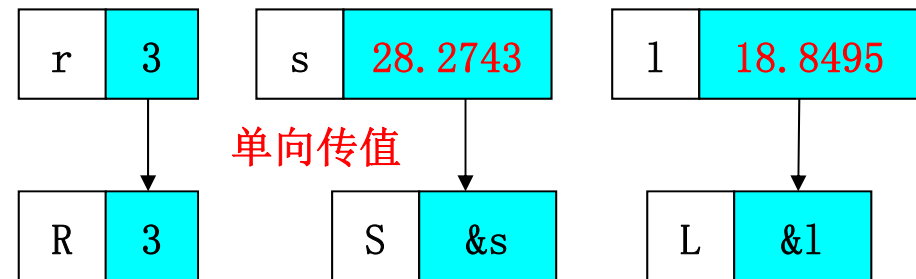
★ 指针变量做参数，可以同时得到多个改变的实参值，从而达到一个函数返回多个值的目的

```
#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

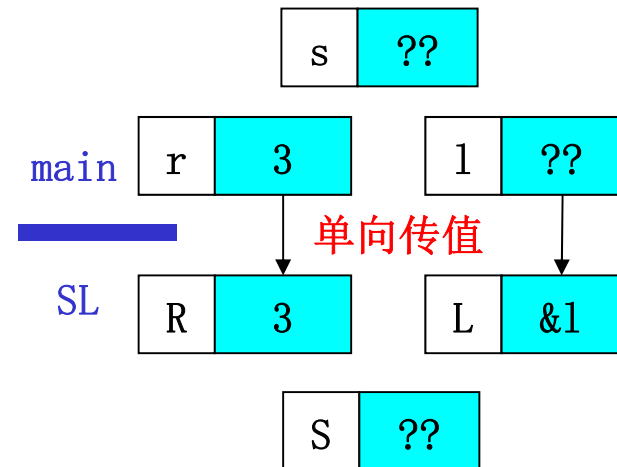
#### 6.2.5. 指针变量作函数的参数

★ 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

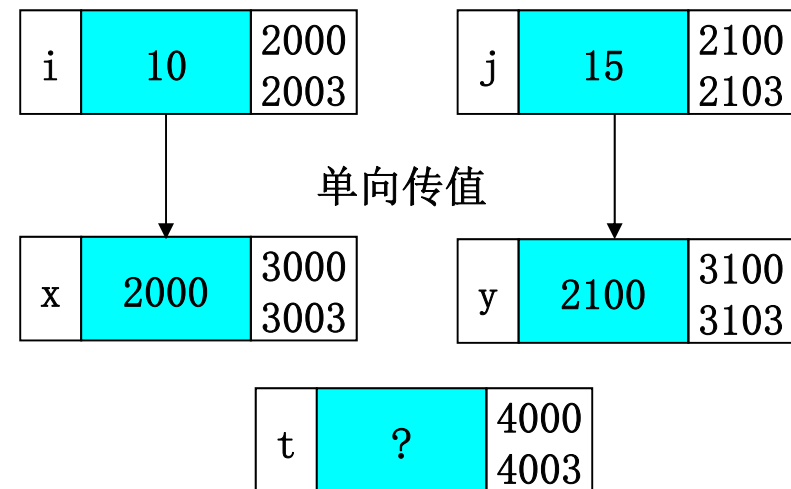
### 6.2. 变量与指针

#### 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;
}
```

VS2017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()
{
```

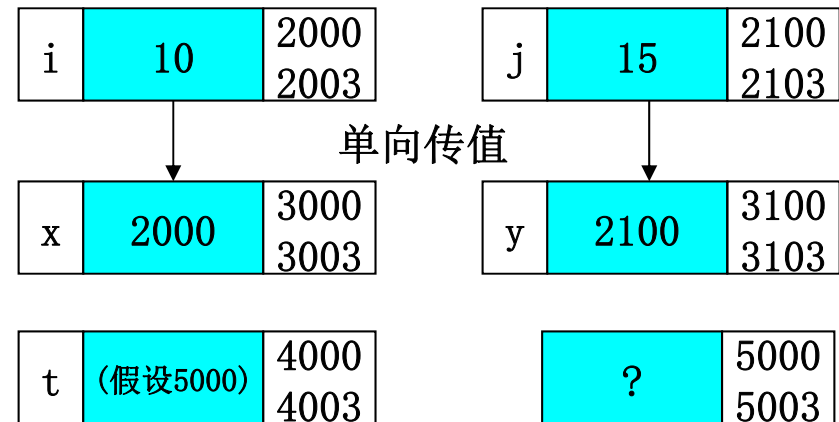
```
    int i=10, j=15;
```

```
    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 ( $\Leftrightarrow \&a[0]$ )

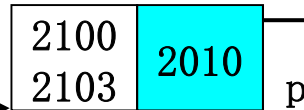
数组元素的指针：数组中某个元素的地址

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

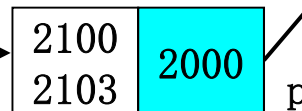
#### 6.3.3. 指向数组的指针变量

```
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

`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
}
```

## § 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. 165-166 例6.5及变化

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;
}
```

## § 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. 165-166 例6.5及变化

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

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. 165-166 例6.5及变化

P. 165 例6.5 (3)-变化

```
int main()
{   int a[10], i, *p=a;
    for(b=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(b=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. 165 例6.5 (3)-变化

```
int main()
{   int a[10], i, *p=a;
    for(b=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

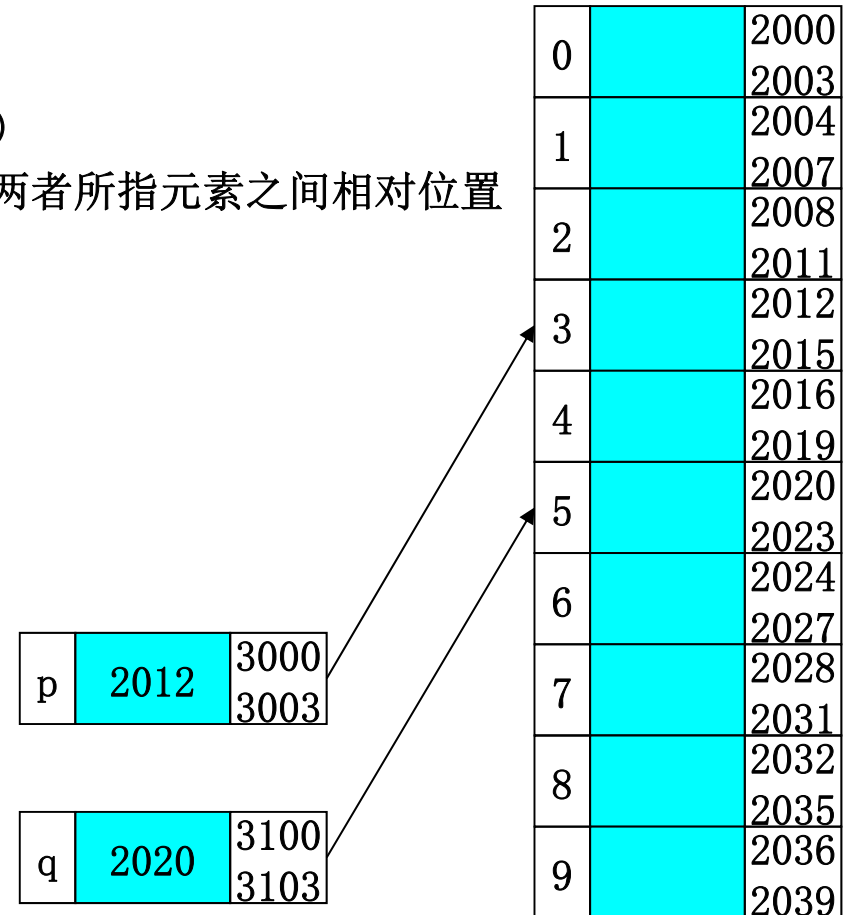
##### 6.3.4.3. 指向数组的指针变量的运算

#### 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

P. 166-167 例6.6 (和P. 135例5.7对比)

#### 6.3.5. 用指针变量作函数参数接收数组地址

```
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. 135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. 135

```
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. 167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. 135



## § 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

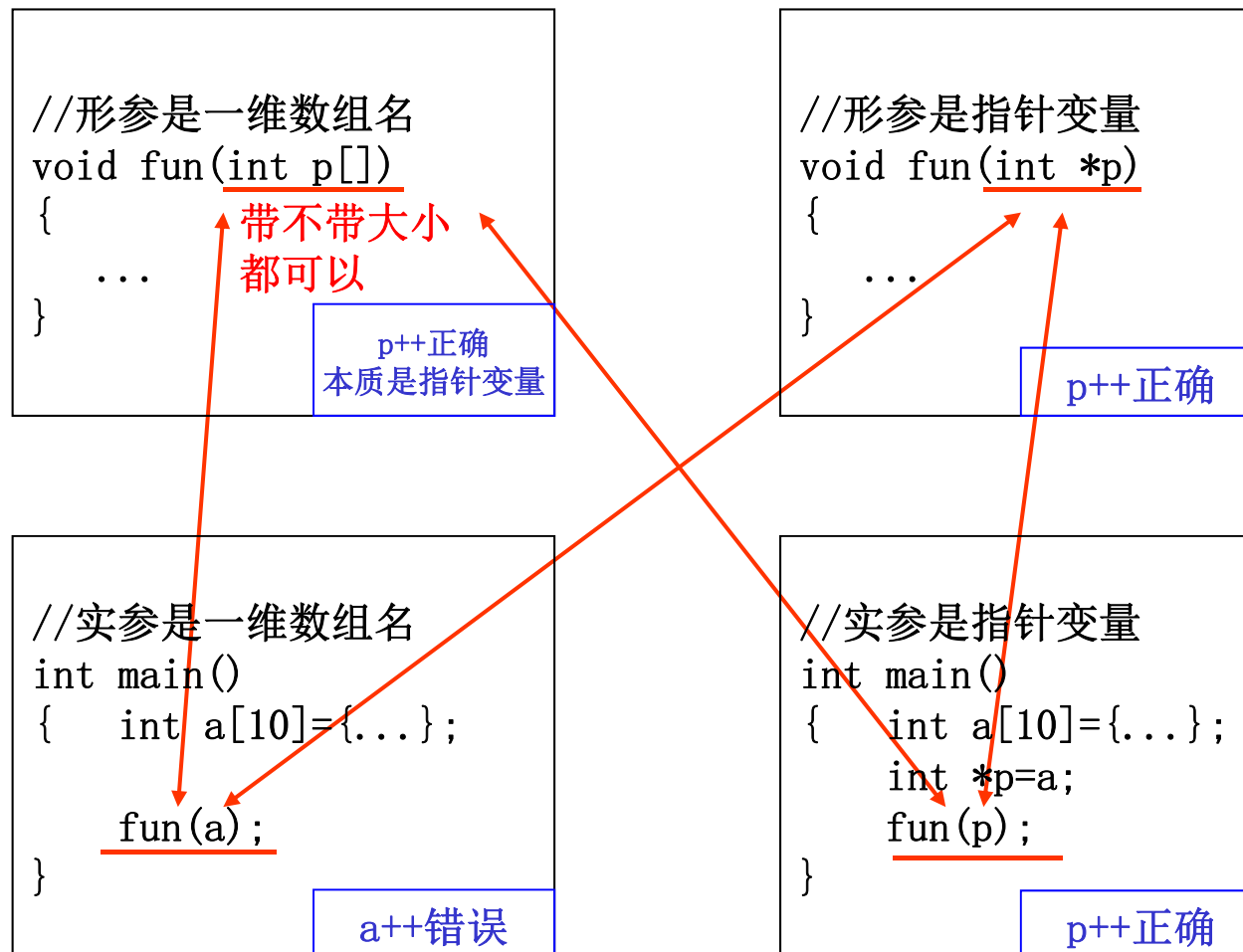
### 6.3. 数组与指针

#### 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. 168 表6.1 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:

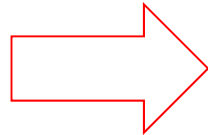
```
int a[3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

1    2    3    4

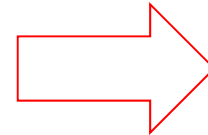
5    6    7    8

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

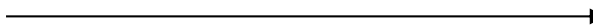
### 6.3. 数组与指针

#### 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)
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!!!)

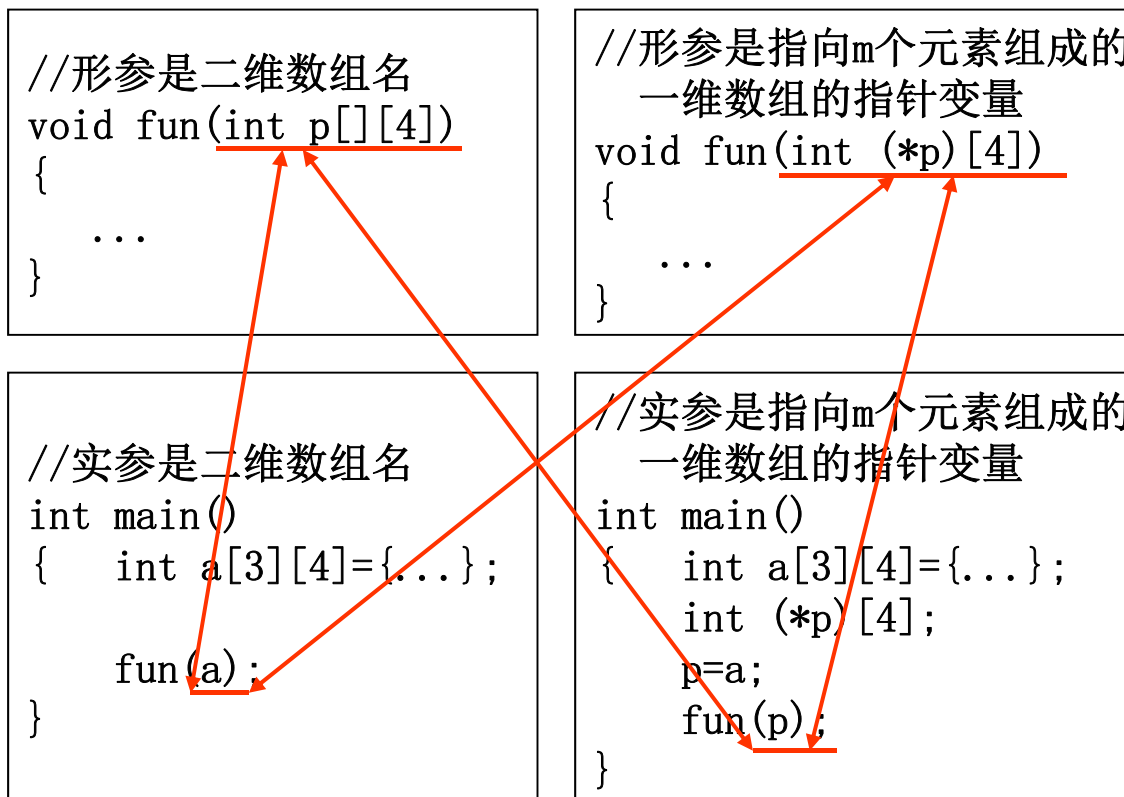
##### 6.3.6.5. 用指向二维数组的指针做函数参数

★ 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



## § 6. 善于使用指针与引用

### 6. 4. 字符串与指针

#### 6. 4. 1. 字符指针的定义及使用 (重复6. 3, 类型为char)

char \*指针变量名

char \*p;

指向简单变量的指针:

char ch='A', \*p;

p=&ch; 地址

赋值语句

char ch, \*p=&ch; 定义时赋初值

\*p='B'; 值, 相当于ch='B'

指向一维数组的指针:

char ch[10], \*p;

p=&ch[3]; 指向数组的第3个元素

p=ch / p=&ch[0]; 指向数组的开始

指向二维数组的指针:

char ch[10][20];

char \*p1; //指向元素的指针

char (\*p2)[20]; //指向一维数组的指针

p1 = &ch[0][0]; / p1 = ch[0]; / p1 = \*ch;

p2 = ch; / p2 = &ch[0];

## § 6. 善于使用指针与引用

### 6. 4. 字符串与指针

#### 6. 4. 2. 用字符指针指向字符串

第5章中:

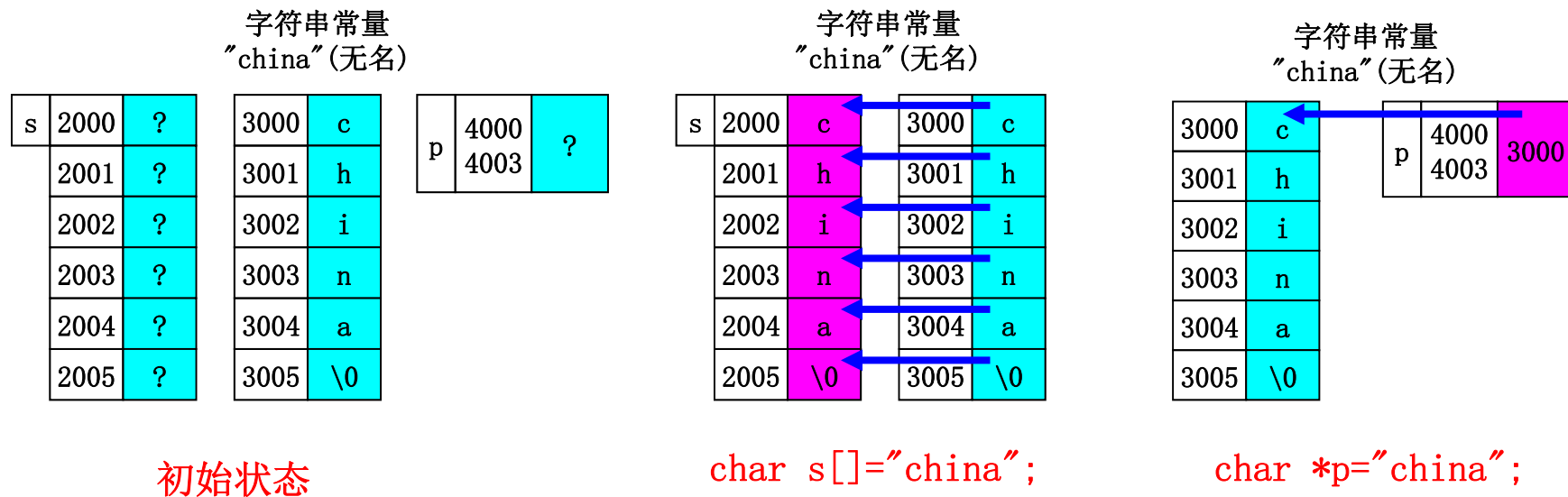
```
char s[]="china"; //一维字符数组, 缺省为6  
string s="china"; //C++类变量表示, 暂不讨论
```

本章:

```
char *p="china"; //字符指针指向字符串
```

区别: s是一个一维字符数组名, 占用一定的内存空间, 编译时确定地址不变 (s++错)

p是基类型为char的指针变量, 表示一个字符(串首字符)的地址, 在程序的运行中可以改变



## § 6. 善于使用指针与引用

### 6. 4. 字符串与指针

#### 6. 4. 2. 用字符指针指向字符串

定义时赋初值

```
char *p="china"; ✓
```

用赋值语句赋值

```
char *p;
```

```
p="china"; ✓ p表示取地址，将字符串常量的首地址赋给p
```

```
*p="china"; ✗ *p表示取值，基类型是char，因此不能是字符串
```

```
*p='c'; ? 编译正确，能否正确执行视情况而定，具体例子后面会给出
```

定义时赋初值

```
char s[]="china"; ✓
```

用赋值语句赋值

```
char s[10];
```

```
s="china"; ✗
```

数组不能整体赋值

要用strcpy(s, "China");

```
*s="china"; ✗
```

\*s↔s[0], char型值

```
*s='c'; ✓
```

#### 6. 4. 3. 下标法与指针法处理字符串

```
char a[]="china", *p;
```

```
p=a; / p=&a[0];
```

```
a[i]      p[i]
```

```
*(a+i)    *(p+i)
```

```
*a++      *p++
```

```
(*a)++    (*p)++
```

8种表示方式中有一种是错误的？哪种？

是否正确？  
后续6. 4. 5中解决

## § 6. 善于使用指针与引用

### 6. 4. 字符串与指针

#### 6. 4. 3. 下标法与指针法处理字符串

P. 170 例6. 10

```
int main()
{ char str1[]="I Love CHINA!", str2[20];
  char *p1,*p2;
  p1=str1;
  p2=str2;
  for(;*p1!='\0';p1++,p2++)
      *p2=*p1;
  *p2='\0';
  p1=str1;
  p2=str2;
  cout ...
  return 0;
}
```

P. 170 例6. 10

```
int main()
{ char str1[]="I Love CHINA!", str2[20];
  int i;
  for(i=0;str1[i]!='\0';i++)
      str2[i] = str1[i];
  str2[i]='\0';
  cout << str1 << endl;
  cout << str2 << endl;
  return 0;
}
```

数组法实现

P. 170 例6. 10

```
int main()
{ char str1[]="I Love CHINA!", str2[20];
  int i;
  for(i=0;*(str1+i]!='\0';i++)
      *(str2+i) = *(str1+i);
  *(str2+i]='\0';
  cout << str1 << endl;
  cout << str2 << endl;
  return 0;
}
```

等价指针法



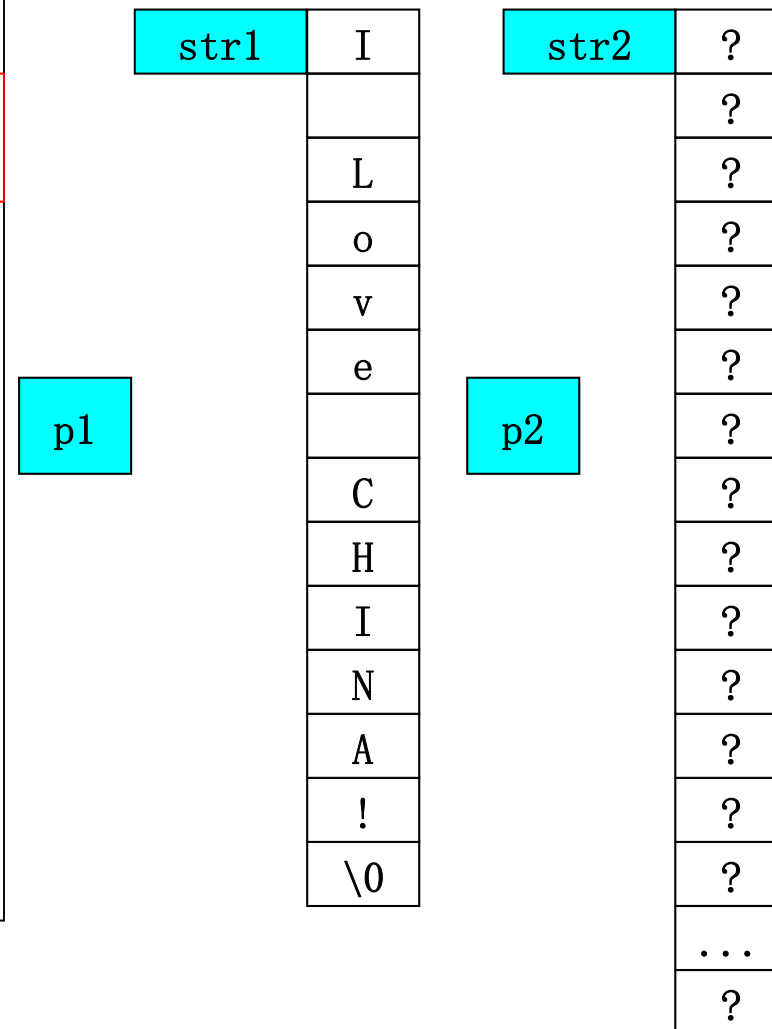
## § 6. 善于使用指针与引用

### 6. 4. 字符串与指针

#### 6. 4. 3. 下标法与指针法处理字符串

P. 170 例6. 10

```
int main()
{
    char str1[]="I Love CHINA!", str2[20];
    char *p1, *p2;
    p1=str1;
    p2=str2;
    for(; *p1!='\0'; p1++, p2++)
        *p2=*p1;
    *p2='\0';
    p1=str1;
    p2=str2;
    cout ...
    return 0;
}
```



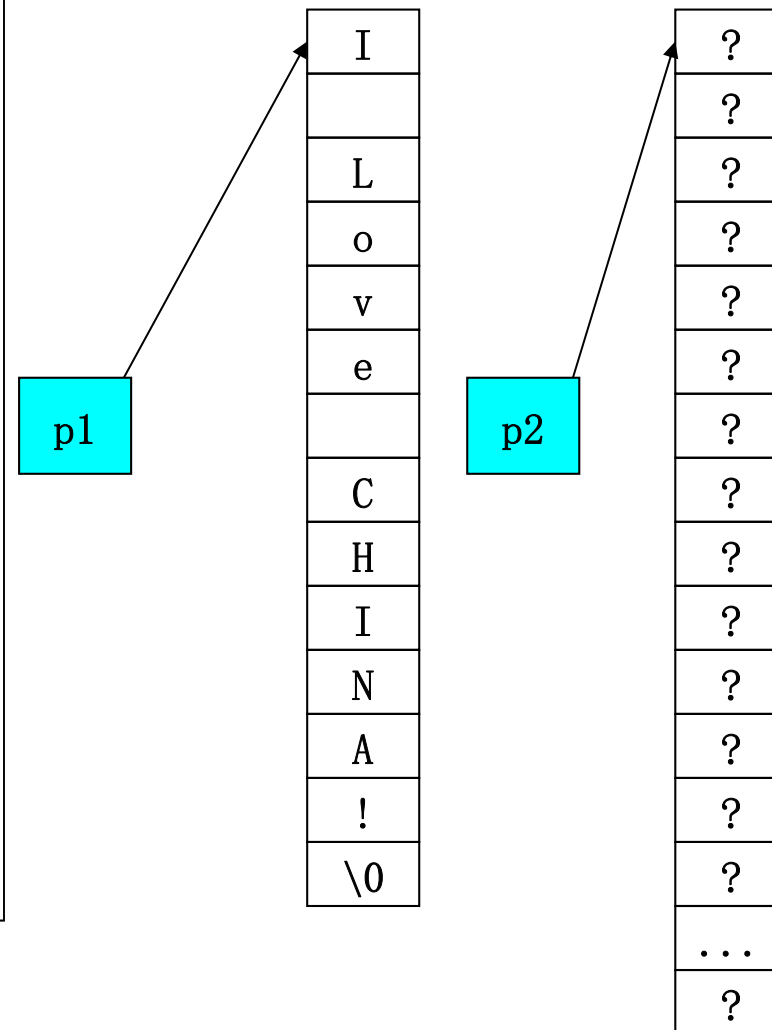
## § 6. 善于使用指针与引用

### 6. 4. 字符串与指针

#### 6. 4. 3. 下标法与指针法处理字符串

P. 170 例6. 10

```
int main()
{ char str1[]="I Love CHINA!", str2[20];
  char *p1,*p2;
  p1=str1;
  p2=str2;
  for(;*p1!='\0';p1++,p2++)
    *p2=*p1;
  *p2='\0';
  p1=str1;
  p2=str2;
  cout ...
  return 0;
}
```



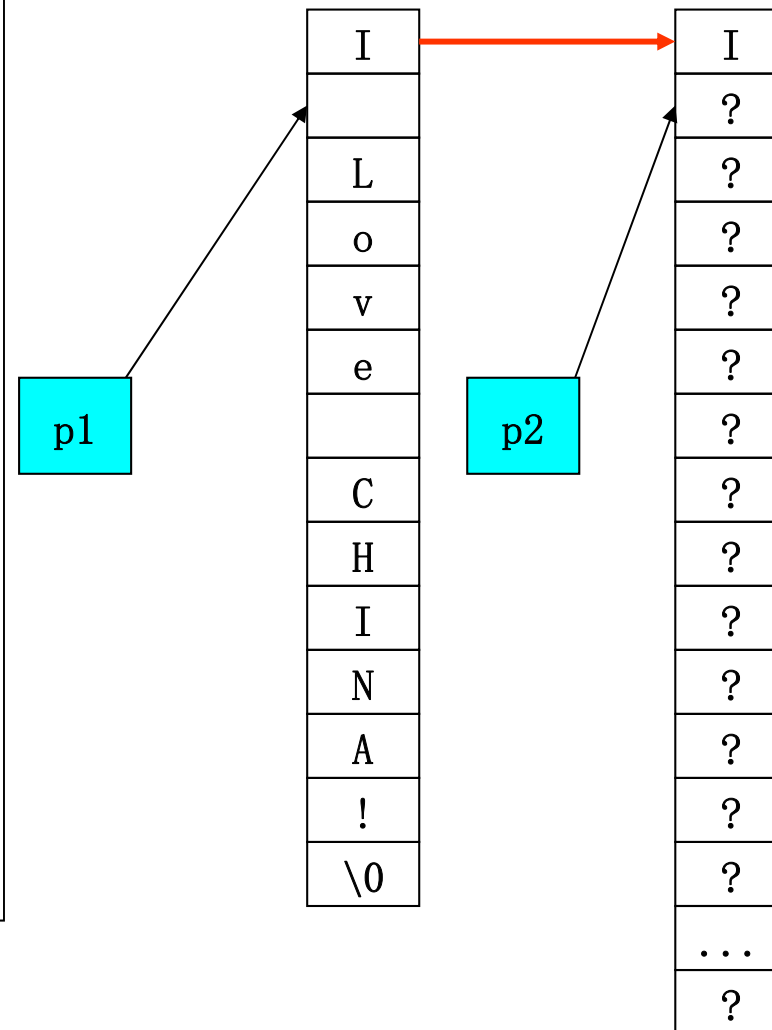
## § 6. 善于使用指针与引用

### 6. 4. 字符串与指针

#### 6. 4. 3. 下标法与指针法处理字符串

P. 170 例6. 10

```
int main()
{ char str1[]="I Love CHINA!", str2[20];
  char *p1,*p2;
  p1=str1;
  p2=str2;
  for(;*p1!='\0';p1++,p2++)
    *p2=*p1;
  *p2='\0';
  p1=str1;
  p2=str2;
  cout ...
  return 0;
}
```



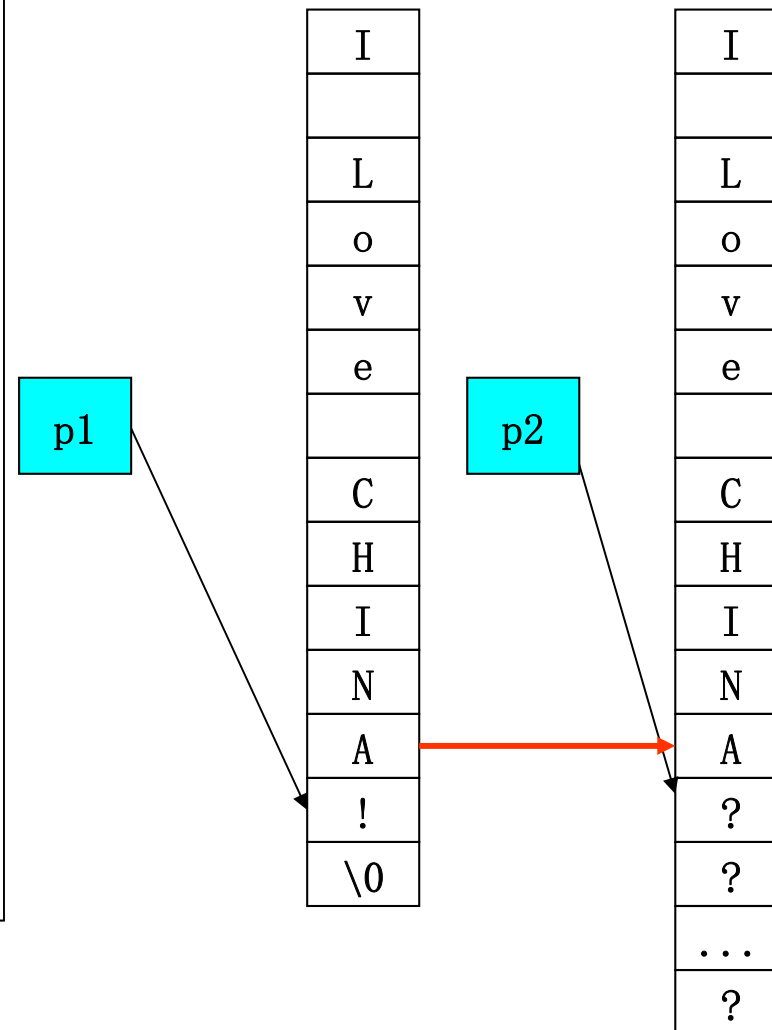
## § 6. 善于使用指针与引用

### 6. 4. 字符串与指针

#### 6. 4. 3. 下标法与指针法处理字符串

P. 170 例6. 10

```
int main()
{ char str1[]="I Love CHINA!", str2[20];
  char *p1,*p2;
  p1=str1;
  p2=str2;
  for(;*p1!='\0';p1++,p2++)
    *p2=*p1;
  *p2='\0';
  p1=str1;
  p2=str2;
  cout ...
  return 0;
}
```



## § 6. 善于使用指针与引用

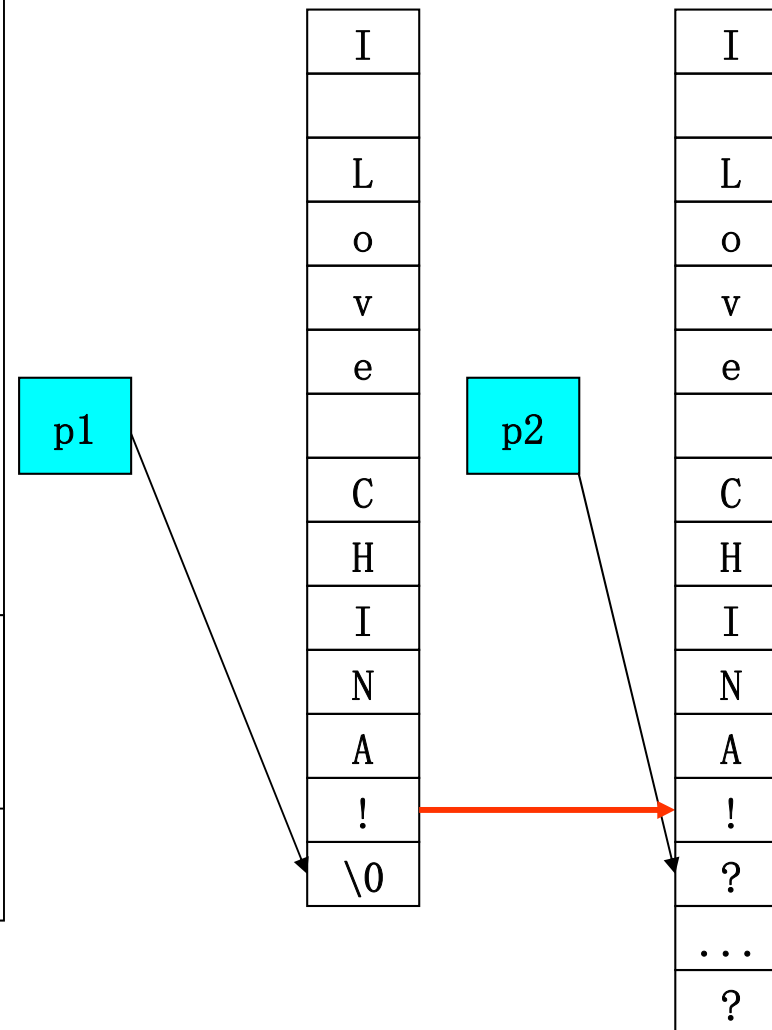
### 6. 4. 字符串与指针

#### 6. 4. 3. 下标法与指针法处理字符串

P. 170 例6. 10

```
int main()
{ char str1[]="I Love CHINA!", str2[20];
  char *p1,*p2;
  p1=str1;
  p2=str2;
  for(;*p1!='\0';p1++,p2++)
    *p2=*p1;
  *p2='\0';
  p1=str1;
  p2=str2;
  cout ...
  return 0;
}
```

! 赋值完成后,  
循环结束  
注意: \0未赋



## § 6. 善于使用指针与引用

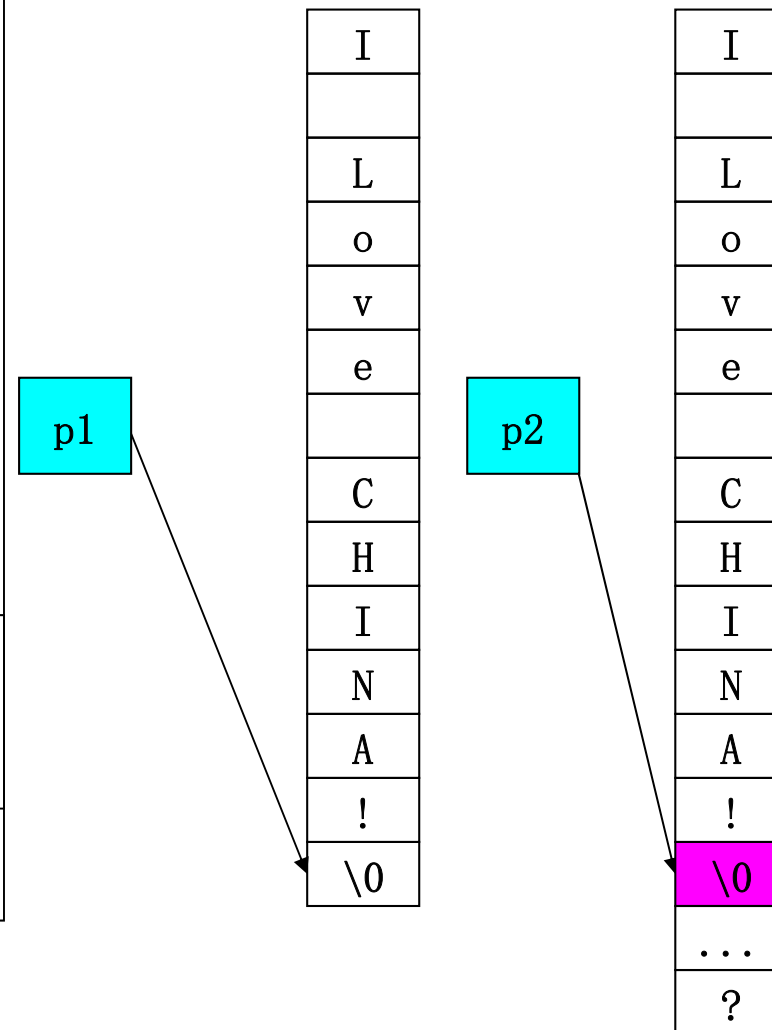
### 6. 4. 字符串与指针

#### 6. 4. 3. 下标法与指针法处理字符串

P. 170 例6. 10

```
int main()
{ char str1[]="I Love CHINA!", str2[20];
  char *p1,*p2;
  p1=str1;
  p2=str2;
  for(;*p1!='\0';p1++,p2++)
    *p2=*p1;
  *p2='\0';
  p1=str1;
  p2=str2;
  cout ...
  return 0;
}
```

! 赋值完成后,  
循环结束  
注意: \0未赋



## § 6. 善于使用指针与引用

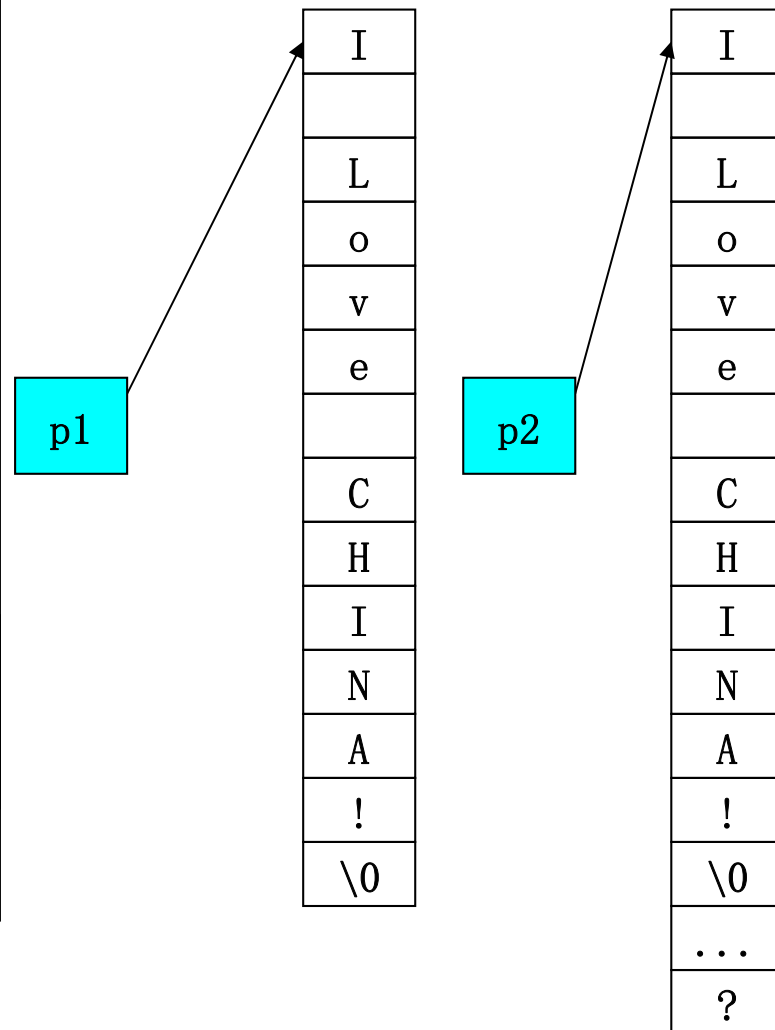
### 6. 4. 字符串与指针

#### 6. 4. 3. 下标法与指针法处理字符串

P. 170 例6. 10

```
int main()
{ char str1[]="I Love CHINA!", str2[20];
  char *p1,*p2;
  p1=str1;
  p2=str2;
  for(;*p1!='\0';p1++,p2++)
    *p2=*p1;
  *p2='\0';
  p1=str1;
  p2=str2;
  cout ...
  return 0;
}
```

重新指向



## § 6. 善于使用指针与引用

### 6. 4. 字符串与指针

#### 6. 4. 3. 下标法与指针法处理字符串

P. 170 例6. 10

```
int main()
{ char str1[]="I Love CHINA!", str2[20];
  char *p1,*p2;
  p1=str1;
  p2=str2;
  for(;*p1!='\0';p1++,p2++)
    *p2=*p1;
  *p2='\0';
  p1=str1;
  p2=str2;
  cout ...
  return 0;
}
```

```
for(;*p1!='\0';)
    *p2++ = *p1++;
```

```
for(;*p1;)
    *p2++ = *p1++;
```

```
while(*p1)
    *p2++ = *p1++;
```

1、判断  
2、赋值  
3、++到下一元素  
循环结束,\0未赋

几种等价表示



## § 6. 善于使用指针与引用

### 6. 4. 字符串与指针

#### 6. 4. 3. 下标法与指针法处理字符串

P. 170 例6. 10

```
int main()
{ char str1[]="I Love CHINA!", str2[20];
  char *p1,*p2;
  p1=str1;
  p2=str2;
  while ((*p2=*p1) != '\0') {
    p1++; p2++;
  }
  *p2='\0';
  p1=str1;
  p2=str2;
  cout ...
  return 0;
}
```

1、赋值  
2、判断  
3、++到下一元素  
循环结束, \0已赋

虽不完全等价  
但都是正确的

```
for (; *p1 != '\0'; p1++, p2++)
    *p2 = *p1;
```

1、判断  
2、赋值  
3、++到下一元素  
循环结束, \0未赋

```
while (*p2 = *p1) {
    p1++; p2++;
}
```

1、赋值  
2、++到下一元素  
3、判断旧值  
循环结束, \0已赋

```
while (*p2++ = *p1++);
```

```
while (*p2++ = *p1++)
    ;
```

## § 6. 善于使用指针与引用

### 6. 4. 字符串与指针

#### 6. 4. 3. 下标法与指针法处理字符串

P. 170 例6. 10

```
int main()
{
    char str1[]="I Love CHINA!", str2[20];
    char *p1,*p2;
    p1=str1;
    p2=str2;
    do {
        *p2++ = *p1++;
    } while(*p1);
    *p2='\0';
    p1=str1;
    p2=str2;
    cout ...
    return 0;
}
```

1、赋值  
2、++到下一元素  
3、判断新值  
循环结束, \0未赋

问：改为do-while循环后是否正确？

答：就本例而言，是正确的

若 `char str1[]=""`，即空串的情况下有可能错误

- 1、`str2[0] = str1[0]`  
`'\0' <= '\0'`
- 2、++到 `str1[1]`
- 3、判断`str1[1]`是否`\0`

至此，`str1[1]`已越界，但程序会继续执行  
若`str1[1]!='\0'`，则  
`str2[1] = str1[1]`  
依次类推到`str1[x]`是`'\0'`为止

若`x<=19`，则仍在`str2`的合理范围内，  
不会错

若`x>19`，则对`str2`的赋值会超过数组  
长度20，会导致非法赋值

## § 6. 善于使用指针与引用

### 6. 4. 字符串与指针

#### 6. 4. 3. 下标法与指针法处理字符串

```
#include <iostream>
using namespace std;
int main()
{ char str1[]="I love CHINA!", str2[20];
  char *p1 = str1, *p2 = str2;
  cout << hex << int(str2) << endl;
  for(;; *p1!='\0'; p1++, p2++)
    *p2=*p1;
  *p2='\0';
  cout << hex << int(p2) << endl;
  cout << dec << p2-str2 << endl;
  cout << str2 << endl;
}
```

完全正确  
地址str2

地址str2+13  
13  
I love CHINA!

```
#include <iostream>
using namespace std;
int main()
{ char str1[]="", str2[20];
  char *p1 = str1, *p2 = str2;
  cout << hex << int(str2) << endl;
  for(;; *p1!='\0'; p1++, p2++)
    *p2=*p1;
  *p2='\0';
  cout << hex << int(p2) << endl;
  cout << dec << p2-str2 << endl;
  cout << str2 << endl;
}
```

完全正确  
地址str2

地址str2  
0  
空行

```
#include <iostream>
using namespace std;
int main()
{ char str1[]="I Love CHINA!", str2[20];
  char *p1 = str1, *p2 = str2;
  cout << hex << int(str2) << endl;
  do {
    *p2++ = *p1++;
  } while(*p1);
  *p2='\0';
  cout << hex << int(p2) << endl;
  cout << dec << p2-str2 << endl;
  cout << str2 << endl;
}
```

目前正确  
地址str2

地址str2+13  
13  
I love CHINA!

```
#include <iostream>
using namespace std;
int main()
{ char str1[]="", str2[20];
  char *p1 = str1, *p2 = str2;
  cout << hex << int(str2) << endl;
  do {
    *p2++ = *p1++;
  } while(*p1);
  *p2='\0';
  cout << hex << int(p2) << endl;
  cout << dec << p2-str2 << endl;
  cout << str2 << endl;
}
```

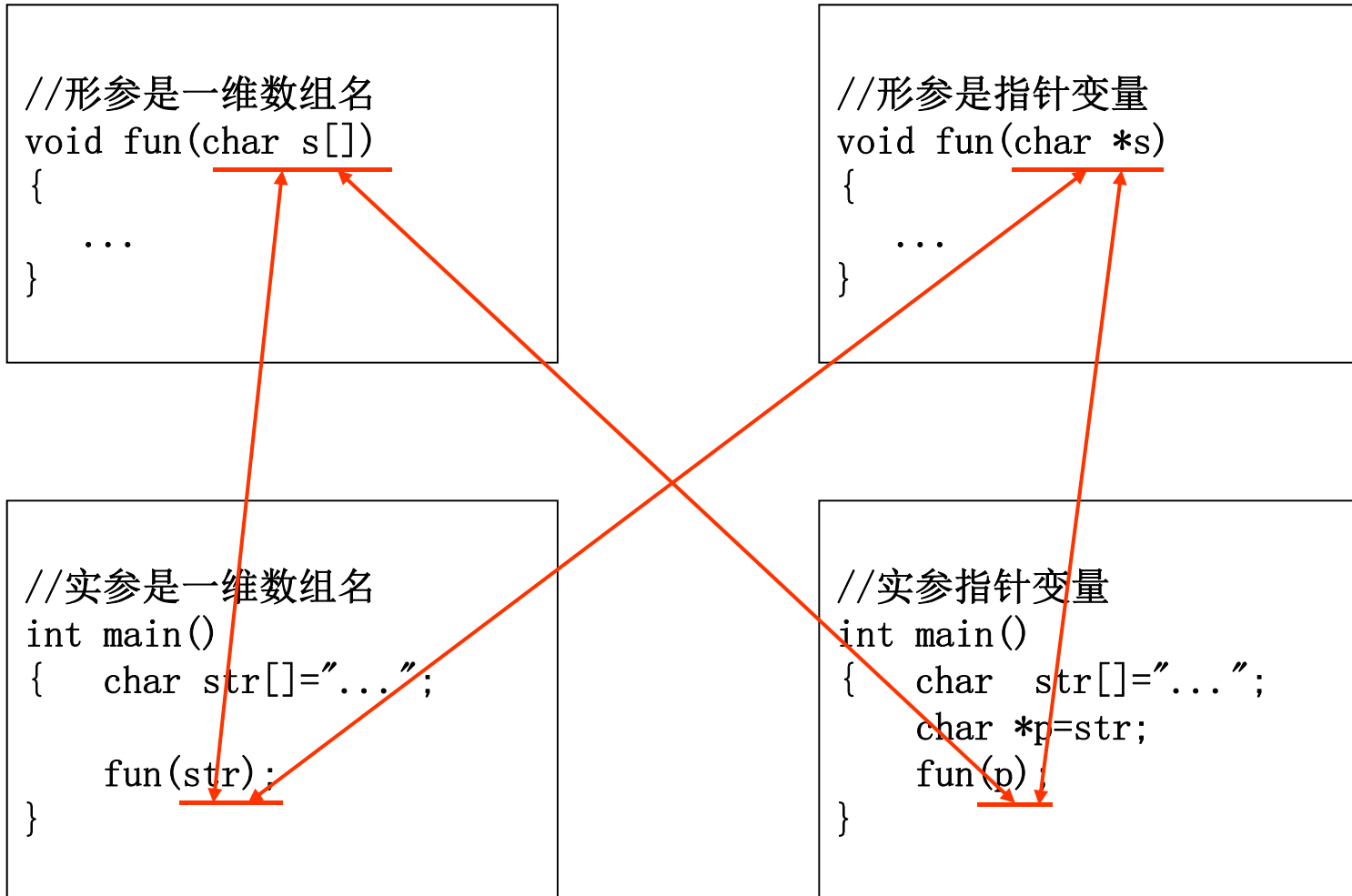
运气好正确  
本质不正确!!!  
地址str2

地址str2+12  
12  
空行

## § 6. 善于使用指针与引用

### 6. 4. 字符串与指针

#### 6. 4. 4. 指向字符数组的指针作函数参数(四种组合, 同前)



## § 6. 善于使用指针与引用

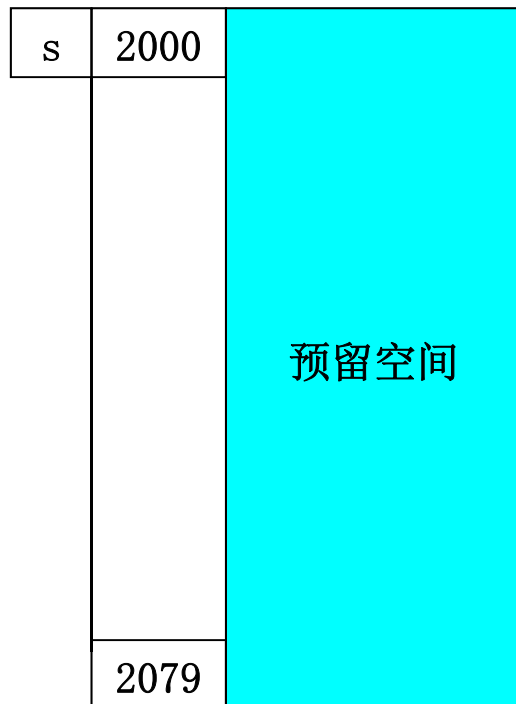
### 6. 4. 字符串与指针

#### 6. 4. 5. 字符指针与字符数组的区别

★ 字符数组占用一定的连续存储空间，而指针仅有一个有效地址的存储空间

```
char s[80];  
cin >> s;
```

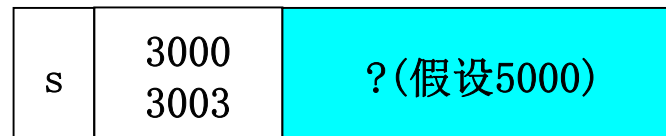
当输入小于80个字符时，正确



```
char *s;  
cin >> s;
```

VS2017编译报error

其他编译器能运行，但运行出错



设键盘输入10个字符，则会覆盖  
5000-5009的空间(非法占用)

如何使正确?

```
char ss[80], *s;  
s=ss; //使s指向确定空间  
cin >> s;
```

## § 6. 善于使用指针与引用

### 6. 4. 字符串与指针

#### 6. 4. 5. 字符指针与字符数组的区别

★ 字符数组占用一定的连续存储空间，而指针仅有一个有效地址的存储空间

★ 赋初值的方式相同，但含义不同

```
char s[]="china";
```

```
char *p="china";
```

s	2000	c	字符串常量 "china"(无名)	3000	c
	2001	h		3001	h
	2002	i		3002	i
	2003	n		3003	n
	2004	a		3004	a
	2005	\0		3005	\0

p	4000	3000
	4003	

★ 字符数组在执行语句中赋值时只能逐个进行，而字符指针仅能整体赋首址

```
char s[80];
```

```
s="china"; ✗
```

数组不允许整体赋值

```
s[0]='c'; ✓
```

```
s[1]='h'; ✓
```

```
s[2]='i'; ✓
```

```
s[3]='n'; ✓
```

```
s[4]='a'; ✓
```

```
s[5]='\0'; ✓
```

s	2000	预留空间
	2079	

```
char *s;
```

```
s="china"; ✓
```

```
s[0]='c'; ✗ 非法占用
```

```
s[1]='h'; ✗
```

```
s[2]='i'; ✗
```

```
s[3]='n'; ✗
```

```
s[4]='a'; ✗
```

```
s[5]='\0'; ✗
```

s	3000	?(假设5000)
	3003	

```
char ss[80], *s;  
s=ss;  
指向确定地址后正确
```

★ 数组首地址的值不可变，指针的值可变

## § 6. 善于使用指针与引用

### 6. 4. 字符串与指针

#### 6. 4. 5. 字符指针与字符数组的区别

##### 6. 4. 2 中的遗留问题:

定义时赋初值

```
char *p="china"; ✓
```

用赋值语句赋值

```
char *p;
```

```
p="china"; ✓ p表示取地址, 将字符串常量的首地址赋给p
```

```
*p="china"; ✗ *p表示取值, 基类型是char, 因此不能是字符串
```

```
*p='c'; ? 编译正确, 能否正确执行视情况而定, 具体例子后面会给出
```

是否正确?

后续6. 4. 5中解决

```
char *p;      *p='c';
```

正确/错误的各种情况

```
int main()
{
    char *p;
    *p='c';
    return 0;
} //编译有警告, 运行错
   p指向未确定空间
```

```
int main()
{
    char ch, *p=&ch;
    *p='c';
    return 0;
} //正确, ch被赋值为'c'
```

```
int main()
{
    char *p="Hello";
    *p='c';
    return 0;
} //编译不错运行错
   p已指向常量
```

```
int main()
{
    char ch[]="Hello";
    char *p=ch;
    *p='c';
    return 0;
} //正确, ch变为"cello"
```

## § 6. 善于使用指针与引用

### 6. 5. 函数与指针

#### 6. 5. 1. 函数的地址

程序(代码)区
静态存储区
动态存储区

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. 5. 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)



## § 6. 善于使用指针与引用

### 6.5. 函数与指针

#### 6.5.1. 函数的地址

#### 6.5.2. 用函数指针变量调用函数

```
//P.171 例6.11
#include <iostream>
using namespace std;

int max(int x, int y)
{
    return (x>y?x:y);
}

int main()
{
    int a,b,m;
    cin >> a >> b;
    m=max(a,b);
    cout << "max=" << m << endl;
    return 0;
}
```

```
//P.172 例6.11
#include <iostream>
using namespace std;

int max(int x, int y)
{
    return (x>y?x:y);
}

int main()
{
    int a,b,m;
    int (*p)(int,int); //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

## § 6. 善于使用指针与引用

### 6.5. 函数与指针

#### 6.5.1. 函数的地址

#### 6.5.2. 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



## § 6. 善于使用指针与引用

### 6.5. 函数与指针

#### 6.5.1. 函数的地址

#### 6.5.2. 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;
}
```

## § 6. 善于使用指针与引用

### 6.5. 函数与指针

#### 6.5.1. 函数的地址

#### 6.5.2. 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

## § 6. 善于使用指针与引用

### 6.5. 函数与指针

#### 6.5.1. 函数的地址

#### 6.5.2. 用函数指针变量调用函数

#### 6.5.3. 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

```
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;
}
```

★ 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）

## § 6. 善于使用指针与引用

### 6.6. 返回指针值的函数

#### 6.6.1. 定义

返回基类型 \*函数名 (形参表)

int \*fun(int x)

float \*function(char ch)

#### 6.6.2. 使用

★ return中的返回值必须是指针 (地址)

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

int *fun(int *x)
{
    x++;
    return x;
}

int main()
{
    int a[10]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, *p;
    p=fun(a);
    cout << "*p=" << *p << endl; *p=1
    p=fun(a+5);
    cout << "*p=" << *p << endl; *p=6
}
```

★ 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)

p	3000	?
	3003	

x	4000	?
	4003	

a	2000	0
	2004	1
	2008	2
	2012	3
	2016	4
	2020	5
	2024	6
	2028	7
	2032	8
	2036	9

## § 6. 善于使用指针与引用

### 6.6. 返回指针值的函数

#### 6.6.1. 定义

返回基类型 \*函数名 (形参表)

int \*fun(int x)

float \*function(char ch)

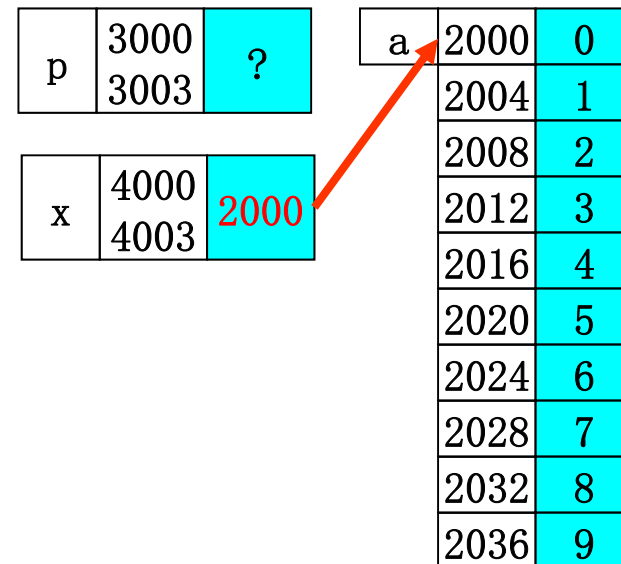
#### 6.6.2. 使用

★ return中的返回值必须是指针 (地址)

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

int *fun(int *x)
{
    x++;
    return x;
}

int main()
{
    int a[10]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, *p;
    p=fun(a);
    cout << "*p=" << *p << endl; *p=1
    p=fun(a+5);
    cout << "*p=" << *p << endl; *p=6
}
```



## § 6. 善于使用指针与引用

### 6.6. 返回指针值的函数

#### 6.6.1. 定义

返回基类型 \*函数名 (形参表)

int \*fun(int x)

float \*function(char ch)

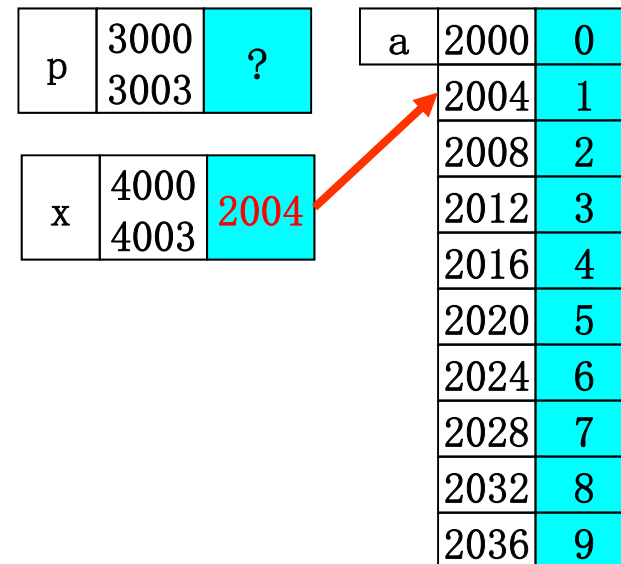
#### 6.6.2. 使用

★ return中的返回值必须是指针 (地址)

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

int *fun(int *x)
{
    x++;
    return x;
}

int main()
{
    int a[10]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, *p;
    p=fun(a);
    cout << "*p=" << *p << endl; *p=1
    p=fun(a+5);
    cout << "*p=" << *p << endl; *p=6
}
```





## § 6. 善于使用指针与引用

### 6.6. 返回指针值的函数

#### 6.6.1. 定义

返回基类型 \*函数名 (形参表)

int \*fun(int x)

float \*function(char ch)

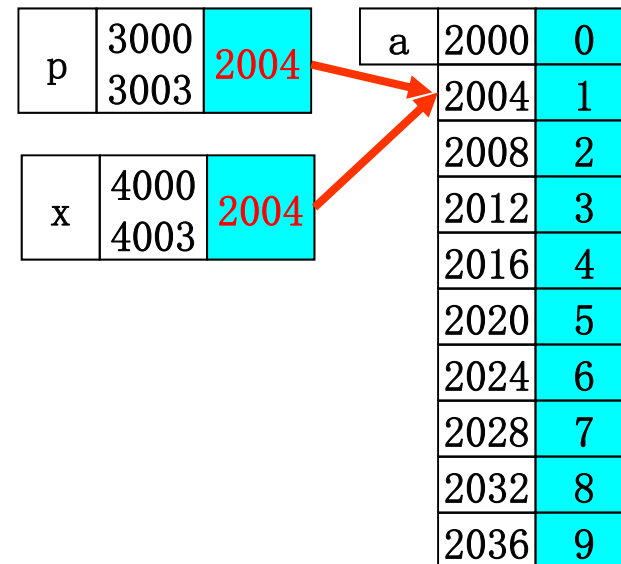
#### 6.6.2. 使用

★ return中的返回值必须是指针 (地址)

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

int *fun(int *x)
{
    x++;
    return x;
}

int main()
{
    int a[10]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, *p;
    p=fun(a);
    cout << "*p=" << *p << endl; *p=1
    p=fun(a+5);
    cout << "*p=" << *p << endl; *p=6
}
```



## § 6. 善于使用指针与引用

### 6.6. 返回指针值的函数

#### 6.6.1. 定义

返回基类型 \*函数名 (形参表)

int \*fun(int x)

float \*function(char ch)

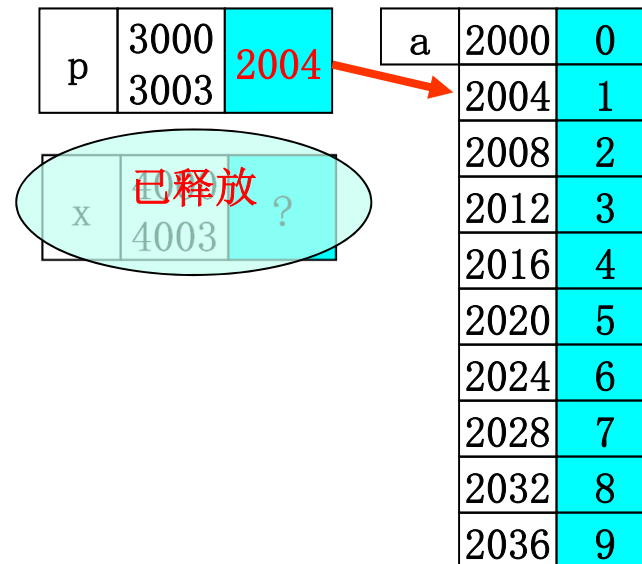
#### 6.6.2. 使用

★ return中的返回值必须是指针 (地址)

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

int *fun(int *x)
{
    x++;
    return x;
}

int main()
{
    int a[10]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, *p;
    p=fun(a);
    cout << "*p=" << *p << endl; *p=1
    p=fun(a+5);
    cout << "*p=" << *p << endl; *p=6
}
```



## § 6. 善于使用指针与引用

### 6.6. 返回指针值的函数

#### 6.6.1. 定义

返回基类型 \*函数名 (形参表)

int \*fun(int x)

float \*function(char ch)

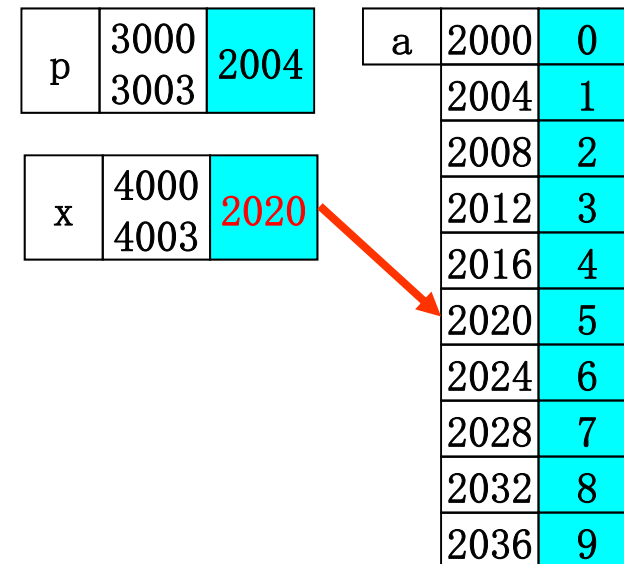
#### 6.6.2. 使用

★ return中的返回值必须是指针 (地址)

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

int *fun(int *x)
{
    x++;
    return x;
}

int main()
{
    int a[10]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, *p;
    p=fun(a);
    cout << "*p=" << *p << endl; *p=1
    p=fun(a+5);
    cout << "*p=" << *p << endl; *p=6
}
```



请自行画出打印 \*p=6 的示例图

## § 6. 善于使用指针与引用

### 6.6. 返回指针值的函数

#### 6.6.2. 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

k	3000	10
	3003	

p	2000	?
	2003	

k	3000	10
	3003	

p	2000	3000
	2003	

k	3000	10
	3003	

p	2000	3000
	2003	

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、VS2017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

## § 6. 善于使用指针与引用

### 6.6. 返回指针值的函数

#### 6.6.2. 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;
}
```

k	3000	10
	3003	

p	2000	?
	2003	

k	3000	10
	3003	

p	2000	3000
	2003	

k/m	3000	20
	3003	

p	2000	3000
	2003	

调用fun2时，k所占空间已释放，  
因此m申请的空间会重复利用  
原k的空间

打印\*p时，k/m所占空间已释放

k/m	3000	20
	3003	

p	2000	3000
	2003	

地址a  
地址a  
20

## § 6. 善于使用指针与引用

### 6. 7. 指针数组和指向指针的指针

#### 6. 7. 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;
}
```

?

## § 6. 善于使用指针与引用

### 6. 7. 指针数组和指向指针的指针

#### 6. 7. 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	a[1]	2010	s	a[2]	2020	s
	2001	h		2011	t		2021	\0
	2002	i		2012	u		2022	
	2003	n		2013	d		2023	
	2004	a		2014	e		2024	
	2005	\0		2015	n		2025	
	2006			2016	t		2026	
	2007			2017	\0		2027	
	2008			2018			2028	
	2009			2019			2029	

##### 赋初值方法

字符串常量  
"china"(无名)

a[0]	2000	c	3000	c
	2001	h	3001	h
	2002	i	3002	i
	2003	n	3003	n
	2004	a	3004	a
	2005	\0	3005	\0
	2006			
	2007			
	2008			
	2009			

优点: (1) 与无名字符常量分占不同空间

(2) 字符串的值可以修改

缺点: (1) 有空间浪费

(2) 若要交换元素(例如排序),  
则需要整体移动元素

##### 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	a[1]	2010	s
	2001	h		2011	t
	2002	i		2012	u
	2003	n		2013	d
	2004	a		2014	e
	2005	\0		2015	n
	2006			2016	t
	2007			2017	\0
	2008			2018	
	2009			2019	

## § 6. 善于使用指针与引用

### 6.7. 指针数组和指向指针的指针

#### 6.7.1. 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"};
```

a	2000	3000
	2004	3100
	2008	3200

字符串常量  
"china"(无名)

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

字符串常量  
"student"(无名)

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

字符串常量  
"s"(无名)

3200	s
3201	\0

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

a	2000	3100
	2004	3000
	2008	3200

##### 交换的方法:

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



## § 6. 善于使用指针与引用

### 6.7. 指针数组和指向指针的指针

#### 6.7.1. 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;
}
```

a[0]	2000	c=>C	a[1]	2010	s	a[2]	2020	s
	2001	h		2011	t		2021	\0
	2002	i		2012	u		2022	
	2003	n		2013	d		2023	
	2004	a		2014	e		2024	
	2005	\0		2015	n		2025	
	2006			2016	t		2026	
	2007			2017	\0		2027	
	2008			2018			2028	
	2009			2019			2029	

## § 6. 善于使用指针与引用

### 6.7. 指针数组和指向指针的指针

#### 6.7.1. 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; //编译不错运行错，为什么？
    cout << a[0] << endl;
    cout << a[1] << endl;
    cout << a[2] << endl;
    return 0;
}
```

a	2000	3000
	2004	3100
	2008	3200

字符串常量  
"china"(无名)

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

字符串常量  
"student"(无名)

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

字符串常量  
"s"(无名)

3200	s
3201	\0

## § 6. 善于使用指针与引用

### 6.7. 指针数组和指向指针的指针

#### 6.7.1. 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;
}
```

a	2000	3000->3300
	2004	3100
	2008	3200

数组b[10]

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

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

## § 6. 善于使用指针与引用

### 6.7. 指针数组和指向指针的指针

#### 6.7.1. 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?

## § 6. 善于使用指针与引用

### 6. 7. 指针数组和指向指针的指针

#### 6. 7. 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

//P. 173 例6. 12 (一维指针数组)

```
void sort(char *name[], int n)
{
    char *temp;
    int i, j, k;
    for(i=0; i<n-1; i++) {
        k=i;
        for(j=i+1; j<n; j++)
            if (strcmp(name[k], name[j])>0)
                k=j;
        if (k!=i) {
            temp=name[i];
            name[i]=name[k];
            name[k]=temp;
        }
    }
}
```

main函数中  
char \*name[]={"BASIC", ...};

//P. 178 例6. 12 (二维字符数组)

```
void sort(char name[][8], int n)
{
    char temp[8];
    int i, j, k;
    for(i=0; i<n-1; i++) {
        k=i;
        for(j=i+1; j<n; j++)
            if (strcmp(name[k], name[j])>0)
                k=j;
        if (k!=i) {
            strcpy(temp, name[i]);
            strcpy(name[i], name[k]);
            strcpy(name[k], temp);
        }
    }
}
```

main函数中  
char name[][8]={"BASIC", ...};

## § 6. 善于使用指针与引用

### 6. 7. 指针数组和指向指针的指针

#### 6. 7. 1. 指针数组

#### 6. 7. 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;
```

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）

i	10	2000
		2001

t	2000	2100
		2103

p	2100	2200
		2203

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

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

- 1、p+1只加了4，证明0x28fef0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

## § 6. 善于使用指针与引用

### 6. 7. 指针数组和指向指针的指针

#### 6. 7. 2. 指向指针的指针

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

```
p=a;
```

p+1 : a[1]的地址 (地址2004)

p++ : p指向a[1] (p的值变为地址2004)

\*p : 取a[0]的值3000 (字符串"china"的首地址)

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

\*p++ : 取a[0]的值3000, p指向a[1] (地址2004)

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

\*p+3 : 取a[0]的值3000, 再+3为3003 (字符'n'的地址)

\*(p+3) : 取a[0]的值3000, 再+3为3003 (字符'n')

p	2100	2000
---	------	------

a	2000	3000
	2004	3100
	2008	3200

字符串常量  
"china"(无名)

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

字符串常量  
"student"(无名)

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

字符串常量  
"s"(无名)

3200	s
3201	\0

## § 6. 善于使用指针与引用

### 6.8. const指针

#### 6.8.0. 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);
}
```

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**

```
#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的数组a中元素的值



## § 6. 善于使用指针与引用

### 6.8. const指针

#### 6.8.1. 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;
}
```

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;
```

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)

```
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);
}
```

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

## § 6. 善于使用指针与引用

### 6.8. const指针

#### 6.8.1. 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

=> 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

<pre>int a; const int b; const int *p; p = &amp;a; //正确 p = &amp;b; //正确</pre>	<pre>const int a = 10; int *p1; const int *p2; p1 = &amp;a; //编译错 p2 = &amp;a; //正确</pre>
--	---

<pre>void f(int *p) { return; } int main() {   int x;     f(&amp;x); }</pre>	<pre>void f(int *p) { return; } int main() {   const int x = 10;     f(&amp;x); }</pre>	<pre>void f(int *p) { return; } int main() {   int x;     const int *p;     p = &amp;x;     f(p); }</pre>
<pre>void f(const int *p) { return; } int main() {   int x;     f(&amp;x); }</pre>	<pre>void f(const int *p) { return; } int main() {   const int x = 10;     f(&amp;x); }</pre>	

## § 6. 善于使用指针与引用

### 6.8. const指针

#### 6.8.2. 常指针

形式：数据类型 \*const 指针变量名

作用：

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

```
#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;
}
```

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
    if (x+2 < 另一个指针)
    *(x+2)=10 / (*x)++ / *x==10
}
int main()
{
    int a[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, *p=&a;
    fun(p);
}
```

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)

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

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

## § 6. 善于使用指针与引用

### 6.8. const指针

#### 6.8.2. 常指针

形式：数据类型 \*const 指针变量名

作用：

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

<pre>void f(int *p) { return; } int main() {   int x;     int *const p = &amp;x;     f(p); }</pre>	<pre>void f(int *const p) { return; } int main() {   int x;     int *const p = &amp;x;     f(p); }</pre>
<pre>void f(int *p) { return; } int main() {   int x;     f(&amp;x); }</pre>	<pre>void f(int *const p) { return; } int main() {   int x;     f(&amp;x); }</pre>

- ★ 常指针不能指向常变量（下面两个例子均编译错）

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

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

问：如何解决？

答：见6.8.3

## § 6. 善于使用指针与引用

### 6.8. const指针

#### 6.8.1. 指向常量的指针变量

#### 6.8.2. 常指针

#### 6.8.3. 指向常量的常指针 (同时满足6.8.1+6.8.2)

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;
}
```

- 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);
}
```

?

## § 6. 善于使用指针与引用

### 6.8. const指针

#### 6.8.1. 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`

#### 6.8.2. 常指针

形式: `数据类型 *const 指针变量名`

#### 6.8.3. 指向常量的常指针(同时满足6.8.1+6.8.2)

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?

## § 6. 善于使用指针与引用

### 6.9. void指针类型

#### 6.2. 变量与指针

##### 6.2.4. 指针变量的++/--

★ void可以声明指针类型，但不能++/--

(void不能声明变量，但可以是函数的形参及返回值)

void k; ✗ 不允许

void \*p; ✓

p++; ✗ 因为不知道基类型的大小

p--; ✗

#### 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

★ void型的指针变量不能进行相互运算 (不知道基类型)

```
void *p, *q;
```

```
cout << (p+2) << endl; //编译错
```

```
cout << (q--) << endl; //编译错
```

```
cout << (p-q) << endl; //编译错
```

```
cout << (p<q+1) << endl; //编译错
```

含义：指向空类型的指针变量

使用：

★ 不能直接通过void指针访问数据(不知道基类型)，  
必须强制转换为某种确定数据类型后才能访问

★ 非void型的指针可直接赋值给void类型，void类型  
赋值给非void类型时必须强制转换

```
#include <iostream>
using namespace std;
int main()
{
    int i=10, *p1=&i, *p3;
    void *p2;

    cout << p1 << endl; // 地址i
    cout << *p1 << endl; // 10
    cout << *p2 << endl; // 编译错误
    p2 = p1;
    p3 = p2; // 编译错误, 改为: p3=(int *)p2
    cout << *p3 << endl; // 10
}
```

## § 6. 善于使用指针与引用

### 6. 10. 有关指针的数据类型和指针运算的小结

★ P. 180 表6.2 各种类型的指针变量

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

思考:

`int *(*p)()` : 1、指出这4种情况中的`p`是?  
(指针/数组/函数)

`int *(*p)[n]` : 2、如果是指针, 指向什么?  
3、如果是数组, 数组元素  
是什么类型?

`int (*p[n])()` : 4、如果是函数, 函数的形参  
及返回类型是什么?

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

(右侧为示例, 全部正确) 

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

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

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

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



## § 6. 善于使用指针与引用

### 6. 10. 有关指针的数据类型和指针运算的小结

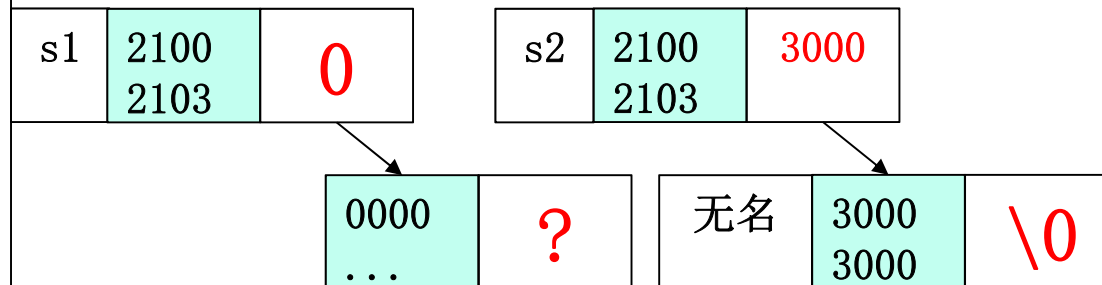
★ P. 180 表6.2 各种类型的指针变量

★ 指针允许有空值 NULL (系统宏定义 `#define NULL 0`)，表示不指向任何变量 (若定义指针变量未赋初值，则随机指向)

NULL与空字符串的区别：

`char *s1 = NULL;` //s1是指针，存放地址0，地址0中的内容不一定是'\0'，  
//即`strlen(s1)`不一定为0

`char *s2 = "";` //s2是指针，存放一个长度为0的无名字符串常量的首地址(非0)，  
//`strlen(s2)`为0



`char s3[] = "";` //正确

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

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

## § 6. 善于使用指针与引用

### 6. 10. 有关指针的数据类型和指针运算的小结

★ P. 180 表6. 2 各种类型的指针变量

★ 指针允许有空值 NULL (系统宏定义 `#define NULL 0`)，表示不指向任何变量 (若定义指针变量未赋初值，则随机指向)

★ 系统的字符串操作函数若传入参数为NULL则会出错

(包括 `strcpy/strcat/strcmp/strlen/strncpy/strncmp`等，以及未出现过的同类函数)

<pre>#include &lt;iostream&gt; #include &lt;cstring&gt; using namespace std; int main() {   char *s1 = NULL;     int len;     len=strlen(s1); }</pre> 错	<pre>#include &lt;iostream&gt; #include &lt;cstring&gt; using namespace std; int main() {   char *s1 = NULL;     char s2[80]="Hello";     strcpy(s2, s1); }</pre> 错
<pre>#include &lt;iostream&gt; #include &lt;cstring&gt; using namespace std; int main() {   char *s1 = NULL;     char s2[80]="Hello";     strcat(s2, s1); }</pre> 错	<pre>#include &lt;iostream&gt; #include &lt;cstring&gt; using namespace std; int main() {   char *s1 = NULL;     char *s2 = NULL;     int k=strcmp(s1, s2); }</pre> 错

=> 自行实现类似功能的字符串处理函数时，可以对NULL进行特殊处理(具体见作业)

- 求长度时为0
- 复制、连接、拷贝时当做空串进行处理

## § 6. 善于使用指针与引用

### 6. 10. 有关指针的数据类型和指针运算的小结

★ P. 180 表6.2 各种类型的指针变量

★ 指针允许有空值 NULL (系统宏定义 `#define NULL 0`), 表示不指向任何变量 (若定义指针变量未赋初值, 则随机指向)

★ 系统的字符串操作函数若传入参数为NULL则会出错

(包括 `strcpy/strcat/strcmp/strlen/strncpy/strncmp`等, 以及未出现过的同类函数)

★ 不同类型的指针变量不能相互赋值, 若要赋值, 则需要强制类型转换

70000=00000000 00000001 00010001 01110000

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

    return 0;
}
```

强制  
类型  
转换

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

    return 0;
}
```

p	2000
---	------

p1	2000
----	------

p2	2000
----	------

a: 低位在前存放	
2000	01110000
2001	00010001
2002	00000001
2003	00000000

70000 (2000-2003)

4464 (2000-2001)

p (2000)

## § 6. 善于使用指针与引用

### 6.11. 引用 (C++新增)

#### 6.11.1. 引用的基本概念

含义：变量的别名

声明：int a, &b=a; //a和b表示同一个变量

★ 引用不分配单独的空间 (指针变量有单独的空间)

变量的定义：分配空间

变量的声明：不分配空间

★ 引用需在声明时进行初始化，指向同类型的变量，在整个生存期内不能再指向其它变量

```
int a, &c=a; //正确
int b, &c=b; //错
      &c=b; //错
c已是a的别名, 不能再
无论定义/赋值均不行
```

```
int a, &c=a, b;
      c=b/b=c ⇔ a=b/b=a
int a, &c=a, b[10];
      c=b[3] ⇔ a=b[3]
      都正确
```

★ 不能声明指向数组的引用、引用数组和指向引用的指针，但可声明数组元素的引用和指向指针的引用

```
int &b[3];           //错误，不能声明指向数组的引用
int &*p;             //错误，不能定义指向引用的指针
int a[5], (&b)[5]=a; //正确，引用指向整个数组
int a[5], &b=a[3];   //正确，引用指向数组元素
int *a, *&b=a;      //正确，指向指针的引用
```

## § 6. 善于使用指针与引用

### 6.11. 引用 (C++新增)

#### 6.11.1. 引用的基本概念

含义：变量的别名

声明：int a, &b=a;    //a和b表示同一个变量

★ 引用不分配单独的空间 (指针变量有单独的空间)

★ 引用需在声明时进行初始化，指向同类型的简单变量，在整个生存期内不能再指向其它变量

★ 不能声明指向数组的引用、引用数组和指向引用的指针，但可声明数组元素的引用和指向指针的引用

★ &的理解

定义语句新变量名前：引用声明符

```
int a, &b=a;
```

其它 (定义语句已定义变量名，执行语句)： 取地址运算符

```
int a, *p=&a;
```

```
p=&a;
```

引用的简单使用：出现在普通变量可出现的任何位置

```
//P.184 例6.15
#include <iostream>
using namespace std;

int main()
{
    int a=10, &b=a;
    a=a*a;
    cout << a << " " << b << endl;    100  100
    b=b/5;
    cout << a << " " << b << endl;    20   20
    return 0;
}
```

## § 6. 善于使用指针与引用

### 6.11. 引用 (C++新增)

#### 6.11.1. 引用的基本概念

#### 6.11.2. 引用作函数参数

P.184 – 186 例6.16 6.17 6.18 两数交换

```
void swap(int a, int b)
{
    int t;
    t = a;
    a = b;
    b = t;
}
```

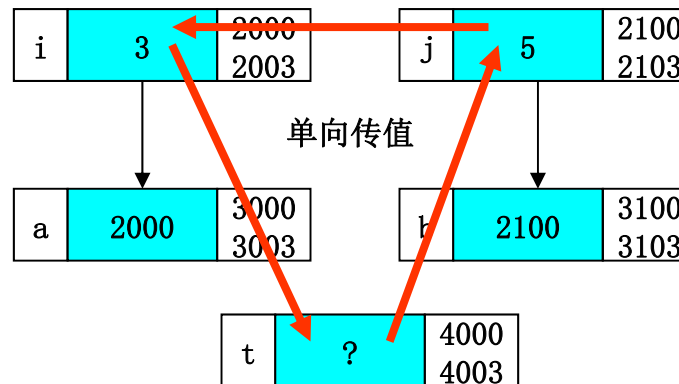
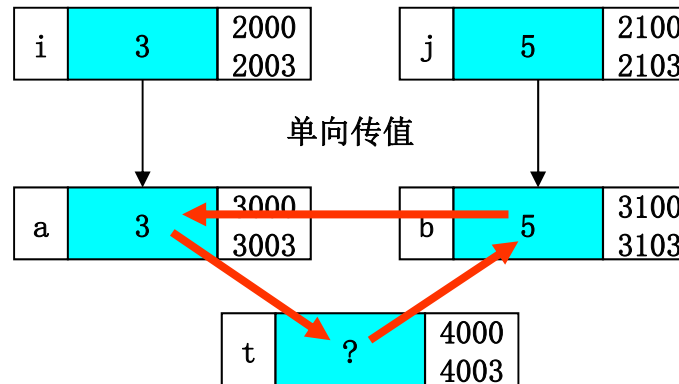
直接传值  
错误

```
int main()
{
    int i=3, j=5;
    swap(i, j);
    cout << i << " " << j << endl;
    return 0;
}
```

```
void swap(int *a, int *b)
{
    int t;
    t = *a;
    *a = *b;
    *b = t;
}
```

传地址  
正确

```
int main()
{
    int i=3, j=5;
    swap(&i, &j);
    cout << i << " " << j << endl;
    return 0;
}
```



## § 6. 善于使用指针与引用

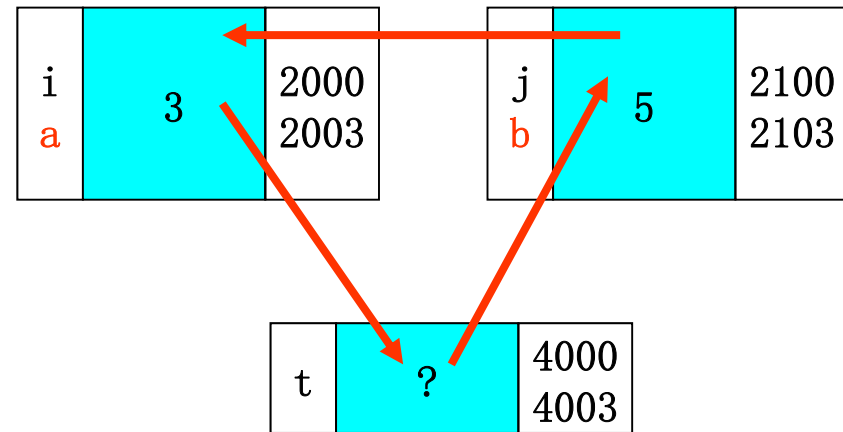
### 6.11. 引用 (C++新增)

#### 6.11.1. 引用的基本概念

#### 6.11.2. 引用作函数参数

P. 184 - 186 例6.16 6.17 6.18 两数交换

```
void swap(int &a, int &b)  引用做形参  
                           正确  
{   int t;  
    t = a;  
    a = b;  
    b = t;  
}  
  
int main()  
{   int i=3, j=5;  
    swap(i, j);  
    cout << i << " " << j << endl;  
    return 0;  
}
```



- ★ 当形参是引用时，不需要声明时初始化，调用时，形参不分配空间，只是当作实参的别名，因此对形参的访问就是对实参的访问
- ★ 实参虽然是变量名，但传递给形参的实际上是实参的地址，同时形参不单独分配空间，只是虚实结合 (P. 187 地址传递方式①-②的解释，没错，但不好理解，可忽略不看)

## § 6. 善于使用指针与引用

### 6.11. 引用 (C++新增)

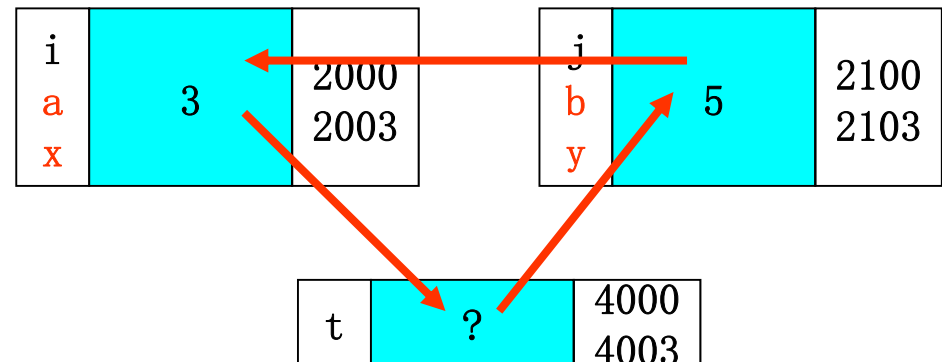
#### 6.11.2. 引用作函数参数

- ★ 当形参是引用时，不需要声明时初始化，调用时，形参不分配空间，只是当作实参的别名，因此对形参的访问就是对实参的访问
- ★ 实参虽然是变量名，但传递给形参的实际上是实参的地址，同时形参不单独分配空间，只是虚实结合 (P. 187 地址传递方式①-②)的解释，没错，但不好理解，可忽略不看)
- ★ 引用允许传递

```
void swap1(int &x, int &y)
{
    int t;
    t = x;
    x = y;
    y = t;
}

void swap(int &a, int &b)
{
    swap1(a, b);
}

int main()
{
    int i=3, j=5;
    swap(i, j);
}
```





## § 6. 善于使用指针与引用

### 6.11. 引用 (C++新增)

#### 6.11.2. 引用作函数参数

- ★ 当形参是引用时，不需要声明时初始化，调用时，形参不分配空间，只是当作实参的别名，因此对形参的访问就是对实参的访问
- ★ 实参虽然是变量名，但传递给形参的实际上是实参的地址，同时形参不单独分配空间，只是虚实结合 (P. 187 地址传递方式①-②的解释，没错，但不好理解，可忽略不看)
- ★ 引用允许传递
- ★ 当引用做函数形参时，实参不允许是常量/表达式，否则编译错误  
(形参为const引用时实参可为常量/表达式)

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

void fun(int x)
{
    cout << x << endl;
}

int main()
{
    int i=10;
    fun(i);    //正确
    fun(15);   //正确
    return 0;
}
```

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

void fun(int &x)
{
    cout << x << endl;
}

int main()
{
    int i=10;
    fun(i);    //正确
    fun(15);   //编译错
    return 0;
}
```

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

void fun(const int &x)
{
    cout << x << endl;
}

int main()
{
    int i=10;
    fun(i);    //正确
    fun(15);   //正确
    return 0;
}
```

## § 6. 善于使用指针与引用

### 6.11. 引用 (C++新增)

#### 6.11.2. 引用作函数参数

#### ★ 数组引用在做函数参数时的作用

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

void test1(char *s1, const char *s2)
{
    cout << sizeof(s1) << endl; 4
    cout << sizeof(s2) << endl; 4
}

void test2(char (&s1)[10], const char *s2)
{
    cout << sizeof(s1) << endl; ???
    cout << sizeof(s2) << endl; 4
}

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

当形参为实参数组的引用时，  
能否取得实参数组的大小？

## § 6. 善于使用指针与引用

### 6.11. 引用 (C++新增)

#### 6.11.2. 引用作函数参数

#### ★ 数组引用在做函数参数时的作用

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

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

    while(*p1++ = *p2++)
        ;           依次赋值，含尾零

    return s1;
}

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

错误原因：函数中无法知道s大小，只能依据t的\0来判断结束

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

char *tj_strcpy_s(char (&s1)[10], const char *s2)
{
    int i;

    for (i=0; s2[i] != '\0' && i < sizeof(s1)-1; i++)
        s1[i] = s2[i];           可以做到安全copy，不会越界

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

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

问题：

- 1、能否和原tj\_strcpy同名(重载)
- 2、可以实用吗？

## § 6. 善于使用指针与引用

### 6.11. 引用 (C++新增)

#### 6.11.2. 引用作函数参数

#### ★ 数组引用在做函数参数时的作用

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

char *tj_strcpy_s(char (&s1)[10], const char *s2)
{
    int i;

    for (i=0; s2[i] != '\0' && i < sizeof(s1)-1; i++)
        s1[i] = s2[i];

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

int main()
{
    char str1[10], str2[]="This is a pencil.";
    tj_strcpy_s(str1, str2); //正确
    cout << str1 << endl;
    char str3[20], str4[]="This is a pen.";
    tj_strcpy_s(str3, str4); //错误
    cout << str3 << endl;
    return 0;
}
```

s1: 形参是实参数组的引用，实形参数组的大小必须一致，无法适应不同长度的实参数组  
s2: 形参是指针，初值指向实参数组的首元素，没有大小，因此能适应不同长度的实参数组

for (i=0; s2[i] != '\0' && i < sizeof(s1)-1; i++)  
s1[i] = s2[i]; 可以做到安全copy，不会越界

问题:  
2、可以实用吗?  
tj\_strcpy\_s() 尚不能投入实用!!!  
如何解决? 自行思考 (需运用后续知识)

- 形参是一维数组的引用，则形参声明的大小必须与实参数组的定义大小一致  
=> 推论：形参是二维数组的引用，则形参声明与实参定义的行列大小必须完全一致

## § 6. 善于使用指针与引用

### 6.11. 引用 (C++新增)

#### 6.11.2. 引用作函数参数

#### ★ 数组引用在做函数参数时的作用

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

void test2(char (&s1)[10])
{
    cout << sizeof(s1) << endl;
}

int main()
{
    char s[10], *p = s;
    test2(s);
    test2(p);
}
```

哪句编译报错?

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

void test2(char (&s1)[10])
{
    cout << sizeof(s1) << endl;
}

void test1(char s1[])
{
    cout << sizeof(s1) << endl;
    test2(s1);
}

int main()
{
    char s[10];
    test1(s);
}
```

哪句编译报错?

- 形参是数组的引用，则实参必须是数组，且大小一致，不能是数组的指针  
=> 推论：数组的引用在函数间传递时，必须始终保持数组引用方式

## § 6. 善于使用指针与引用

### 6.11. 引用 (C++新增)

#### 6.11.3. 关于引用的特别说明

- ★ 引用在需要**改变实参值**的函数调用时比指针方式更容易理解，形式也更简洁，不容易出错
  - ★ 引用不能完全替代指针(可以将指针理解为if-else，引用理解为switch-case)
  - ★ 引用是C++新增的，纯C的编译器不支持，后续工作学习中接触的大量**底层代码**仍是由C编写的，此时无法使用引用 (VS2017是C++编译器，兼容编译纯C，以后缀名.c/c..来区分如何编译，而Linux下gcc/c++区分的较清楚)
- ★ 对于本专业而言，仍需要透彻理解指针!!!