



§ 6. 指针基础

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 \Leftrightarrow 指针)

二进制大数的表示单位:

2^{10}	= 1024	= 1 KB	(KiloByte)
2^{20}	= 1024 KB	= 1 MB	(MegaByte)
2^{30}	= 1024 MB	= 1 GB	(GigaByte)
2^{40}	= 1024 GB	= 1 TB	(TeraByte)
2^{50}	= 1024 TB	= 1 PB	(PeraByte)
2^{60}	= 1024 PB	= 1 EB	(ExaByte)
2^{70}	= 1024 EB	= 1 ZB	(ZettaByte)
2^{80}	= 1024 ZB	= 1 YB	(YottaByte)
2^{90}	= 1024 YB	= 1 BB	(BrontoByte)
2^{100}	= 1024 BB	= 1 NB	(NonaByte)
2^{110}	= 1024 NB	= 1 DB	(DoggaByte)
2^{120}	= 1024 DB	= 1 CB	(CorydonByte)

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. 指针基础

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) **特别说明: 后续可能以VS/x86模式为基准, 直接说“指针变量占4字节”**

<pre>#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; }</pre>	<pre>#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; }</pre>	<pre>#include <iostream> using namespace std; int main() { cout << sizeof(char *) << endl; 8 cout << sizeof(short *) << endl; 8 cout << sizeof(int *) << endl; 8 cout << sizeof(long *) << endl; 8 cout << sizeof(float *) << endl; 8 cout << sizeof(double *) << endl; 8 return 0; }</pre>
---	--	--

- ★ 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. 指针基础

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=&i;          q=&t;
*p=10 ⇔ i=10  *q=10 ⇔ t=10
```

每步分析

★ 假设32位地址系统，则：

i 占2字节是因为 short型。

t 占2字节是因为 long型。

p/q 占4字节是因为 指针类型。

★ 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

```
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
```

间接访问:

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

注: i=10 / t=10 称为直接访问

p	2000	3000
		3003

i	10	2000
		2001

q	2100	4000
		4003

t	10	2100
		2103

```
#include <iostream>
using namespace std;
int main()
{   char    c, *p;      4: 指针变量的大小
    double d, *q;      1: 指针变量的基类型大小
    p = &c;            4: 指针变量的大小
    q = &d;            8: 指针变量的基类型大小
    cout << sizeof(p)  << endl;  4
    cout << sizeof(*p) << endl;  1
    cout << sizeof(q)  << endl;  4
    cout << sizeof(*q) << endl;  8
    return 0;
}
```



§ 6. 指针基础

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. 指针基础

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 *) int(++p5) << endl;

    return 0;
}
```

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

假设地址A
=地址A+2
假设地址B
=地址B+1

Microsoft Visual Studio 调试控制台

```
00B3FA98
00B3FA9A
烫烫烫烫
烫烫烫烫
```



§ 6. 指针基础

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++): 保留p的旧值到临时变量中，p再++(不指向i)，最后取旧值所指的值(i)

++p ⇔ *(++p): p先++(不指向i)，再取p的值(非i)

(*p)++ ⇔ i++ : 取p所指的值(i)，i值再后缀++

++*p ⇔ ++i : 取p所指的值(i)，i值再前缀++

另一种不够准确的解释：先取p所指的值(i)，p再++(不指向i)
虽然能解释最终结果，但无法解释后缀++优先级高于*，为什么不先做++



例：键盘输入两数，按从大到小的顺序依次输出（指针法处理）

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

```
int main()
{
    int *p1, *p2, *p, a, b;

    cin >> a >> b; （假设键盘输入是45 78）
    p1=&a;
    p2=&b;

    if (a<b) {
        p=p1;
        p1=p2;
        p2=p;
    }

    cout << *p1 << ' ' << *p2 << endl;
    return 0;
}
```

p1	???	3000
		3003

p2	???	3100
		3103

p	???	3200
		3203

a	???	2000
		2003

b	???	2100
		2103

带地址的图示法
给一个虚假的10进制地址
(一般地址是16进制)

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

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

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

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

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

不带具体地址的图示法
(教科书、后续课程)
不具体写明地址，用带
箭头的线段指向来表示



例：键盘输入两数，按从大到小的顺序依次输出（指针法处理）

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

```
int main()
{
    int *p1, *p2, *p, a, b;

    cin >> a >> b; (假设键盘输入是45 78)
    p1=&a;
    p2=&b;

    if (a<b) {
        p=p1;
        p1=p2;
        p2=p;
    }

    cout << *p1 << ' ' << *p2 << endl;
    return 0;
}
```

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)



例：键盘输入两数，按从大到小的顺序依次输出（指针法处理）

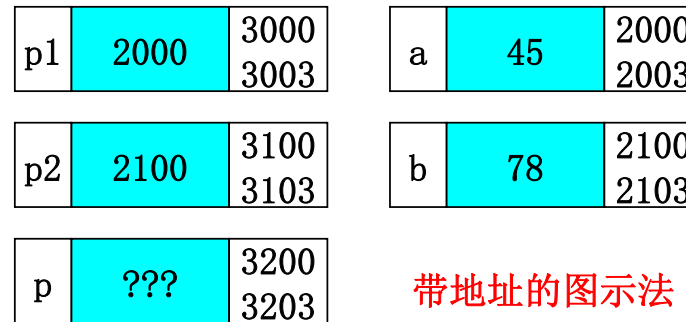
```
#include <iostream>
using namespace std;

int main()
{
    int *p1, *p2, *p, a, b;

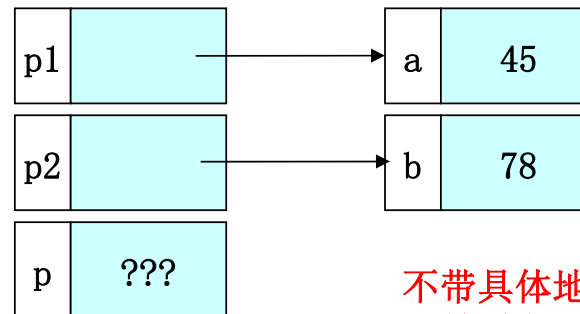
    cin >> a >> b; （假设键盘输入是45 78）
    p1=&a;
    p2=&b;

    if (a<b) {
        p=p1;
        p1=p2;
        p2=p;
    }

    cout << *p1 << ' ' << *p2 << endl;
    return 0;
}
```



带地址的图示法



不带具体地址的图示法
（教科书、后续课程）



例：键盘输入两数，按从大到小的顺序依次输出（指针法处理）

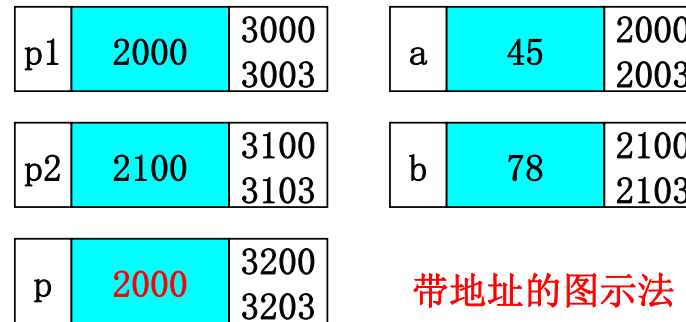
```
#include <iostream>
using namespace std;
```

```
int main()
{
    int *p1, *p2, *p, a, b;

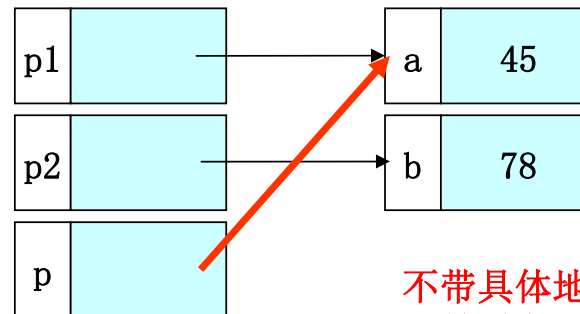
    cin >> a >> b; （假设键盘输入是45 78）
    p1=&a;
    p2=&b;

    if (a<b) {
        p=p1;
        p1=p2;
        p2=p;
    }

    cout << *p1 << ' ' << *p2 << endl;
    return 0;
}
```



带地址的图示法



不带具体地址的图示法
（教科书、后续课程）



例：键盘输入两数，按从大到小的顺序依次输出（指针法处理）

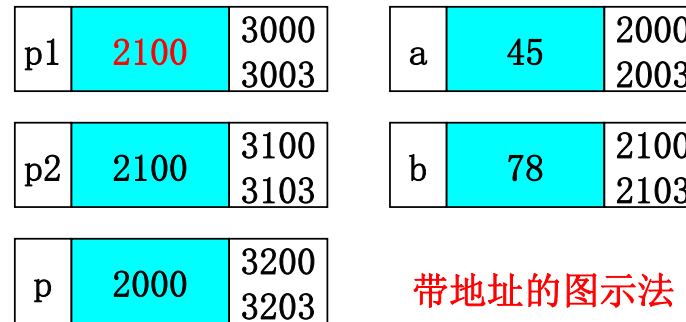
```
#include <iostream>
using namespace std;
```

```
int main()
{
    int *p1, *p2, *p, a, b;

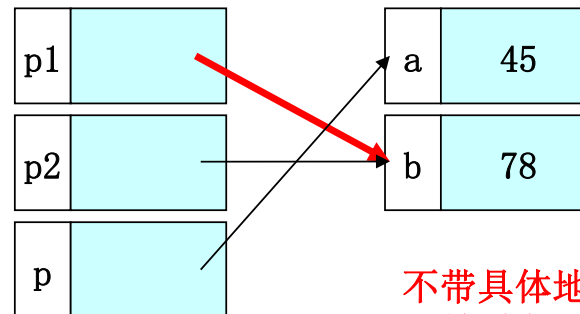
    cin >> a >> b; （假设键盘输入是45 78）
    p1=&a;
    p2=&b;

    if (a<b) {
        p=p1;
        p1=p2;
        p2=p;
    }

    cout << *p1 << ' ' << *p2 << endl;
    return 0;
}
```



带地址的图示法



不带具体地址的图示法
（教科书、后续课程）



例：键盘输入两数，按从大到小的顺序依次输出（指针法处理）

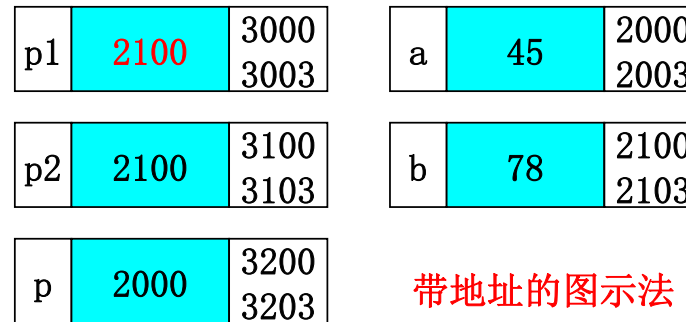
```
#include <iostream>
using namespace std;
```

```
int main()
{
    int *p1, *p2, *p, a, b;

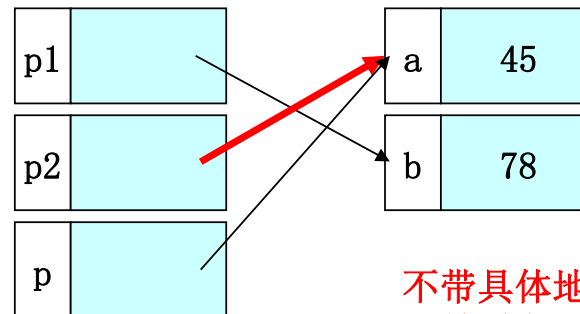
    cin >> a >> b; （假设键盘输入是45 78）
    p1=&a;
    p2=&b;

    if (a<b) {
        p=p1;
        p1=p2;
        p2=p;
    }

    cout << *p1 << ' ' << *p2 << endl;
    return 0;
}
```



带地址的图示法



不带具体地址的图示法
（教科书、后续课程）



例：键盘输入两数，按从大到小的顺序依次输出（指针法处理）

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    int *p1, *p2, *p, a, b;
```

```
    cin >> a >> b; (假设键盘输入是45 78)
```

```
    p1=&a;
```

```
    p2=&b;
```

赋值语句不能表示为：
`*p1=&a;`
`*p2=&b;`

```
    if (a<b) {  
        p=p1;  
        p1=p2;  
        p2=p;  
    }
```

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

```
    cout << *p1 << ' ' << *p2 << endl;
```

```
    return 0;
```

```
}
```



§ 6. 指针基础

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
形参：整型简单变量 匹配

```
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. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

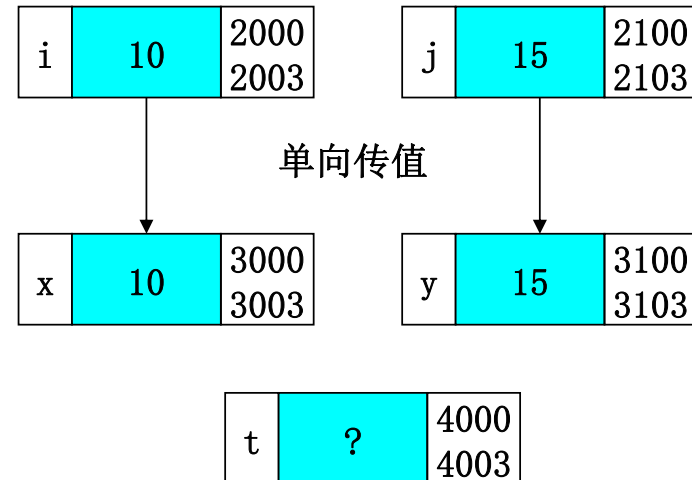
例：编写一个函数，将两个整数进行交换后输出

```
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. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

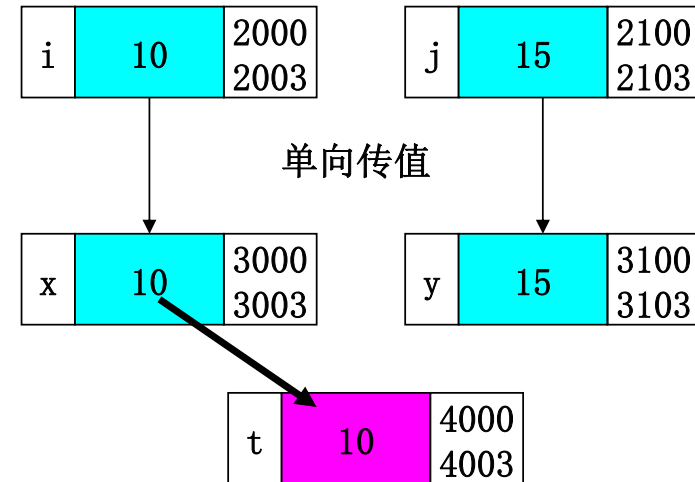
例：编写一个函数，将两个整数进行交换后输出

```
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. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

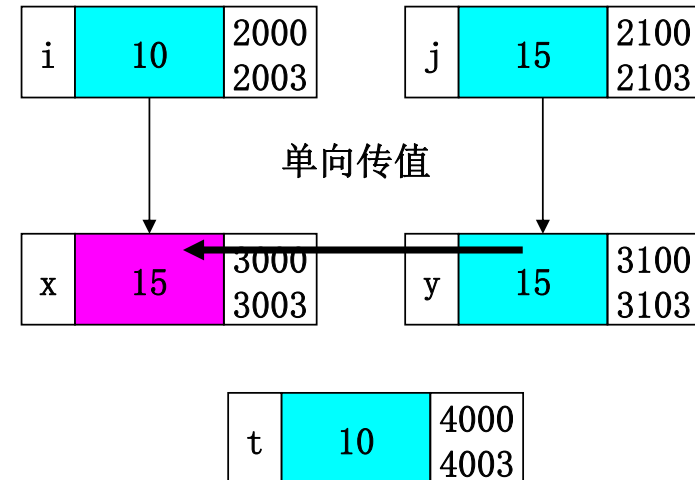
例：编写一个函数，将两个整数进行交换后输出

```
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. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

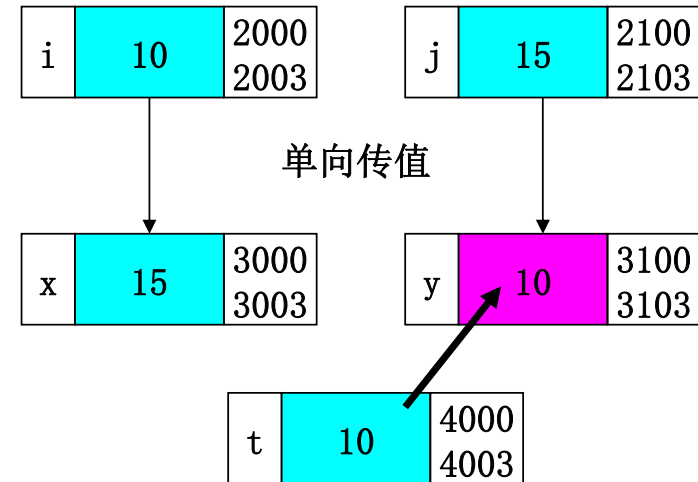
例：编写一个函数，将两个整数进行交换后输出

```
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. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

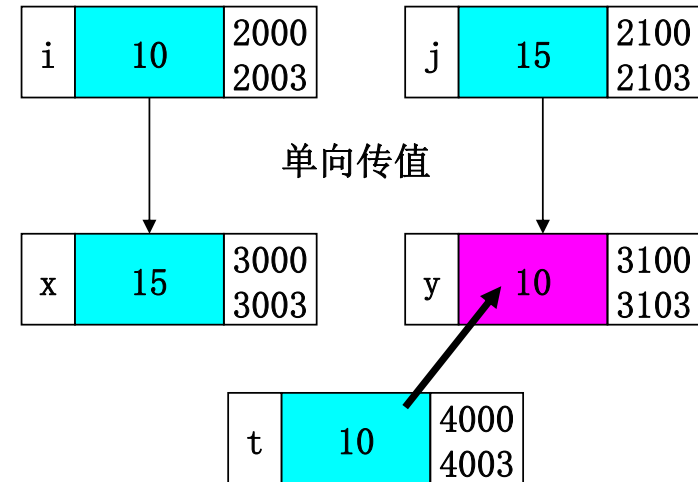
例：编写一个函数，将两个整数进行交换后输出

```
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. 指针基础

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. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

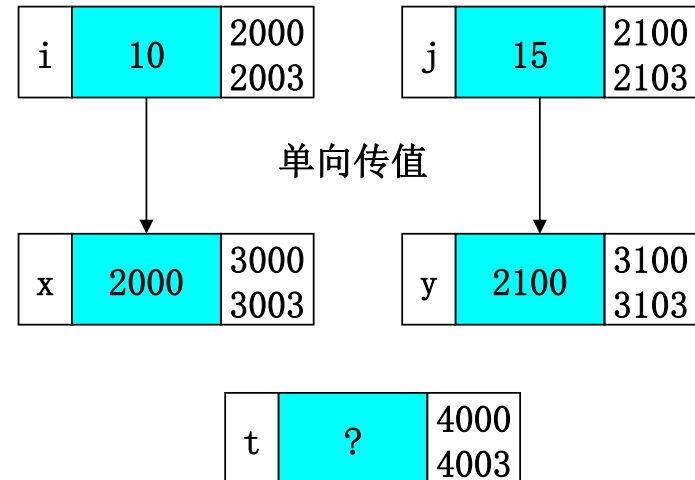
例：编写一个函数，将两个整数进行交换后输出

```
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. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

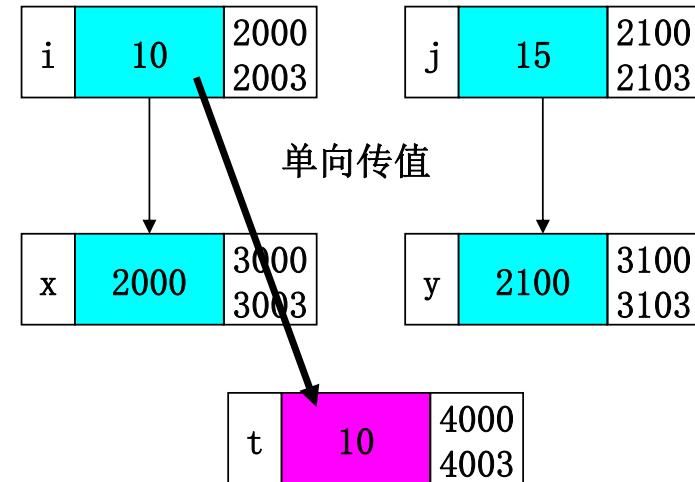
例：编写一个函数，将两个整数进行交换后输出

```
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. 指针基础

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)
```

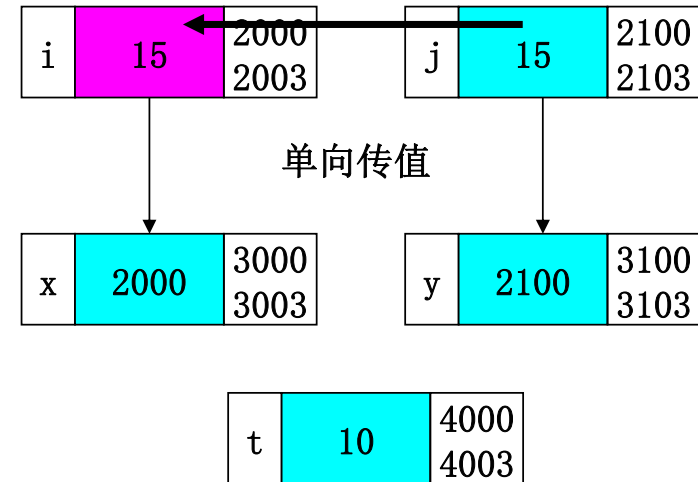
```
{    int t;  
    t = *x;  
    *x = *y;  
    *y = t;
```

```
}
```

```
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. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

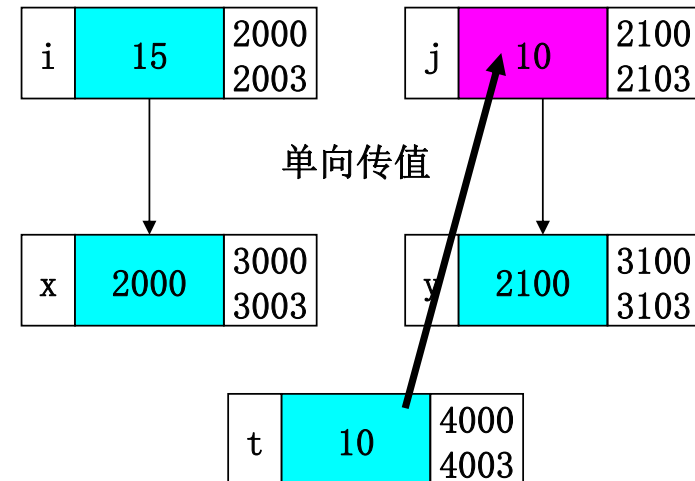
例：编写一个函数，将两个整数进行交换后输出

```
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. 指针基础

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
形参：整型指针变量 匹配

```
}
```

```
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
```

```
}
```

与swap(&i, &j)等价

正确的方法



§ 6. 指针基础

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
    return 0;
}
```

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)



§ 6. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

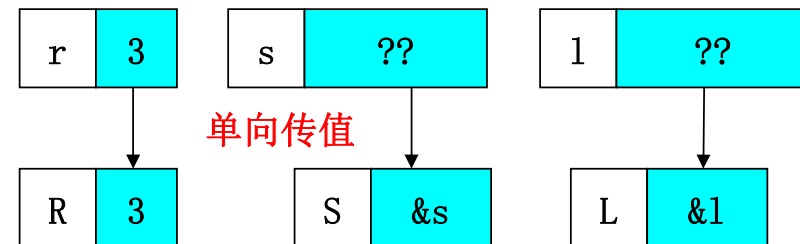
- ★ 指针变量做函数参数，虽然可以通过形参 (**实参地址**) 来**间接访问**实参，从而达到改变实参值的目的，但本质上仍然是**单向传值**，而**不是形参值回传实参**
- ★ 指针变量做参数，可以同时得到多个改变的实参值，从而达到一个函数返回多个值的目的

```
#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
    return 0;
}
```

实参与形参



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)



§ 6. 指针基础

6.2. 变量与指针

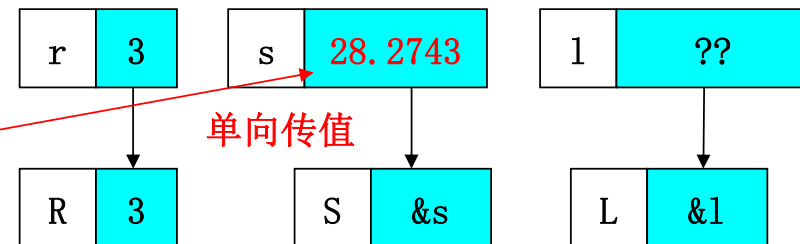
6.2.5. 指针变量作函数的参数

- ★ 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;
    cout << "l=" << l << endl;
    return 0;
}
```



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)



§ 6. 指针基础

6.2. 变量与指针

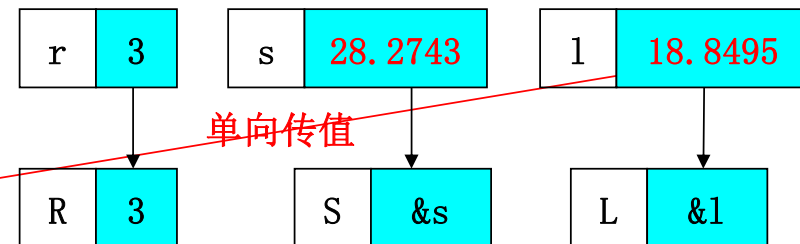
6.2.5. 指针变量作函数的参数

- ★ 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;
    cout << "l=" << l << endl;
    return 0;
}
```



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)



§ 6. 指针基础

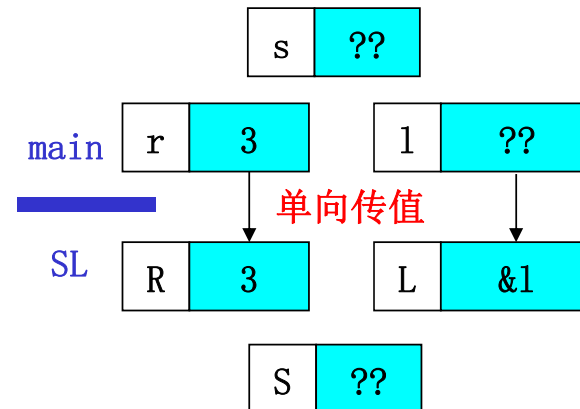
6.2. 变量与指针

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;
    cout << "l=" << l << endl;
    return 0;
}
```

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



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



§ 6. 指针基础

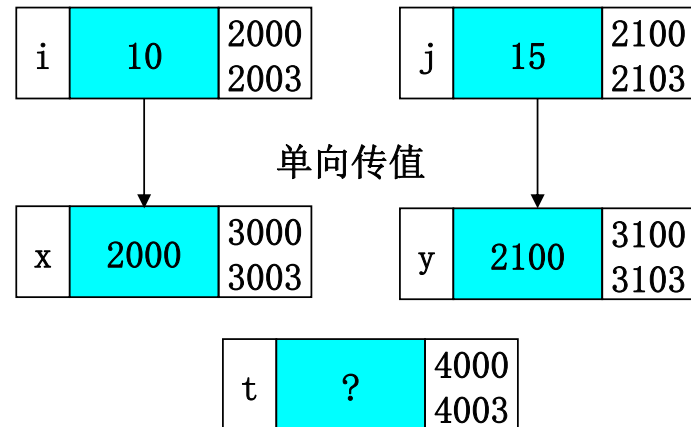
6.2. 变量与指针

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. 指针基础

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;
}
```

VS2019编译报错

-使用了未初始化的局部变量t

其它编译器可能可以运行

初始内存分配如图所示，请自行画出
swap中三句话执行时内存的变化，理解为什么出现严重错误

另1: 哪句是错误的关键?

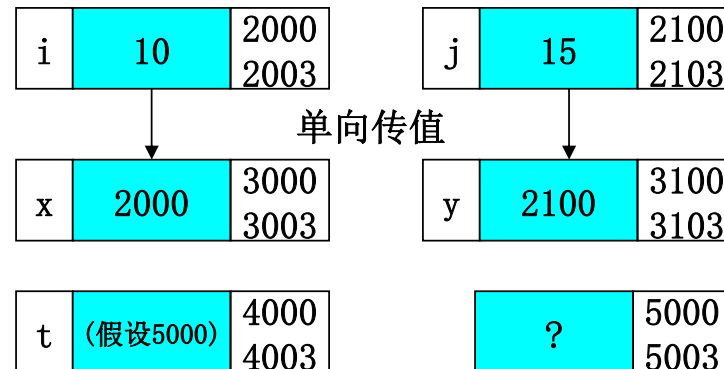
另2: int *t 改为 int tt, *t;

t = &tt;

为什么就正确了?

```
int main()
```

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



单向传值

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. 指针基础

6.3. 一维数组与指针

6.3.1. 基本概念

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

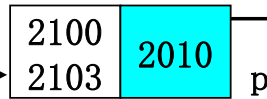
数组元素的指针：数组中某个元素的地址

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	

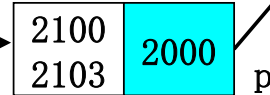
6.3.3. 指向数组的指针变量

```
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. 指针基础

6.3. 一维数组与指针

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($\text{sizeof}(\text{数组名})/\text{sizeof}(\text{指针})$ 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;
    cout << &a[0] << endl;
    cout << sizeof(*a) << endl;
    cout << sizeof(*(&a[0])) << endl;
    cout << sizeof(a) << endl;
    cout << sizeof(&a[0]) << endl;
}
```

换为short, 结果?

数组a的地址	地址a
a[0]元素的地址	地址a
地址a的基类型 \Leftrightarrow a[0]的类型	4
地址&a[0]的基类型 \Leftrightarrow a[0]的类型	4
数组a的大小	40
地址&a[0]的大小	4



§ 6. 指针基础

6.3. 一维数组与指针

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)` 都表示访问数组的第*i*个元素
- ★ `a[i] ⇔ *(a+i)` 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

```
#include <iostream>
using namespace std;
int main()
{
    int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    int *p = &a[3], i;

    for (i = 0; i < 10; i++)
        cout << a[i] << ' ';
    cout << endl;

    for (i = -3; i < 7; i++)
        cout << p[i] << ' ';
    cout << endl;
    return 0;
}
```

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

0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

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

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



§ 6. 指针基础

6.3. 一维数组与指针

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

例：键盘读入10个数并输出

(1)-数组名下标法

```
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;
}
```

(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

(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;
}
```

(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. 指针基础

6.3. 一维数组与指针

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

例：键盘读入10个数并输出

(3) 用指针运算循环数组

```
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

(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

(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. 指针基础

6.3. 一维数组与指针

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

例：键盘读入10个数并输出

```
(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

```
(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;
}
```

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

正确

```
(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. 指针基础

6.3. 一维数组与指针

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

Microsoft Visual Studio 调试控制台

```
0079FCF4--0079FCF8
0079FD08--0079FD04
0079FD00--0079FD0C
0079FD10--0079FD04
```



§ 6. 指针基础

6.3. 一维数组与指针

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)

- ★ 以上每个操作均为独立操作

- ★ 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. 指针基础

6.3. 一维数组与指针

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

地址a
地址a+12
地址a+20
-2
2

Microsoft Visual Studio 调试控制台
009EF9D4
009EF9E0
009EF9E8
-2
2

p	2012	3000
		3003

q	2020	3100
		3103

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



§ 6. 指针基础

6.3. 一维数组与指针

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

不同编译器
结果不相同

Dev: 11, -11

VS: 16, -16

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

不同编译器
结果不相同

Dev: 1, -1

VS: 6, -6

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. 指针基础

6.3. 一维数组与指针

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. 指针基础

6.3. 一维数组与指针

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;
}
```

```
cpp-demo.cpp  x
cpp-demo
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int* p;
6      cout << (p * 2) << endl;
7      cout << (p / 2) << endl;
8  }
```

```
(6,19): error C2296: "*" : 非法，左操作数包含 "int *" 类型
(7,19): error C2296: "/" : 非法，左操作数包含 "int *" 类型
```

- ★ 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;
}
```

```
cpp-demo.cpp  x
cpp-demo
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int* p, *q;
6      cout << (p + q) << endl;
7      cout << (p * q) << endl;
8      cout << (p / q) << endl;
9  }
```

```
(6,19): error C2110: "+" : 不能添加两个指针
(7,19): error C2296: "*" : 非法，左操作数包含 "int *" 类型
(7,19): error C2297: "*" : 非法，右操作数包含 "int *" 类型
(8,19): error C2296: "/" : 非法，左操作数包含 "int *" 类型
(8,19): error C2297: "/" : 非法，右操作数包含 "int *" 类型
```



§ 6. 指针基础

6.3. 一维数组与指针

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

★ 两个不同基类型的指针变量不能进行包括减及比较在内的任何运算 (编译报错)

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

```
cpp-demo.cpp  x
cpp-demo  (全局范围)
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int* p;
6      unsigned int* q;
7      cout << (p-q) << endl;
8      cout << (p-q < 2) << endl;
9      cout << (p+q) << endl;
10     cout << (p*q) << endl;
11     cout << (p/q) << endl;
12 }
```

```
(7,19): error C2440: “-”: 无法从“short*”转换为“int*”
(8,20): error C2440: “-”: 无法从“short*”转换为“int*”
(9,19): error C2110: “+”: 不能添加两个指针
(10,19): error C2296: “*”: 非法, 左操作数包含“int*”类型
(10,19): error C2297: “*”: 非法, 右操作数包含“short*”类型
(11,19): error C2296: “/”: 非法, 左操作数包含“int*”类型
(11,19): error C2297: “/”: 非法, 右操作数包含“short*”类型
```

★ 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)
}
```

```
cpp-demo.cpp  x
cpp-demo  (全局范围)
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      void* p, *q;
6      cout << (p+2) << endl; //编译错(void无大小)
7      cout << (q--) << endl; //编译错(void无大小)
8      cout << (p-q) << endl; //编译错(void无大小)
9      cout << (p < q+1) << endl; //编译错(void无大小)
10     cout << (p < q) << endl; //编译错(pq未初始化)
11 }
```

```
(6,19): error C2036: “void*” : 未知的大小
(7,15): error C2036: “void*” : 未知的大小
(8,19): error C2036: “void*” : 未知的大小
(9,23): error C2036: “void*” : 未知的大小
```



§ 6. 指针基础

6.3. 一维数组与指针

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++`, 保留`p`旧值, `p`指向下一个数组元素的地址, 再取`p`旧值所指元素的值 (`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. 指针基础

6.3. 一维数组与指针

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

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

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

//第05模块例子

```
#include <iostream>
```

```
using namespace std;
```

```
void f1(int x1[]) //形参数组不指定大小
```

```
{  
  cout << "x1_size=" << sizeof(x1) << endl;  
}
```

```
void f2(int x2[10]) //形参数组大小与实参相同
```

```
{  
  cout << "x2_size=" << sizeof(x2) << endl;  
}
```

```
void f3(int x3[1234]) //形参数组大小与实参不同
```

```
{  
  cout << "x3_size=" << sizeof(x3) << endl;  
}
```

```
int main()
```

```
{  
  int a[10];
```

```
  cout << "a_size=" << sizeof(a) << endl;
```

```
  f1(a);
```

```
  f2(a);
```

```
  f3(a);
```

```
}
```

a_size=40

x1_size=4 因为int*

x2_size=4 因为int*

x3_size=4 因为int*

(为什么是4指针部分会解释)



§ 6. 指针基础

6.3. 一维数组与指针

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

例：选择排序（下标法和指针法对比）

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

数组法

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

语句理解为数组法
访问指针变量

形参是指针变量
其余同左

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

数组法访问
改成
指针法访问

指针法

```
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;
    }
}
```

array换名为p,
其余同上



§ 6. 指针基础

6.3. 一维数组与指针

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

//第05模块, 数组名及简单变量做函数参数, 验证形参修改是否影响实参

```
#include <iostream>
```

```
using namespace std;
```

```
void fun(int x[], int y)
```

```
{
```

```
    x[2]=37; //修改形参数组某元素的值
```

```
    y=45;    //修改简单变量形参的值
```

```
}
```

```
int main()
```

```
{    int a[10] = {7, -2, 18, 25}, w=19;
```

```
    cout << a[2] << ' ' << w << endl;
```

```
    fun(a, w);
```

```
    cout << a[2] << ' ' << w << endl;
```

```
    return 0;
```

```
}
```

Microsoft Visual Studio 调试控制台

```
18 19
37 19
```

//第05模块, 数组名及简单变量做函数参数, 验证形参修改是否影响实参

```
#include <iostream>
```

```
using namespace std;
```

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

```
{
```

```
    *(x+2)=37; //x[2]=37;
```

```
    y=45;
```

```
}
```

```
int main()
```

```
{    int a[10] = {7, -2, 18, 25}, w=19;
```

```
    cout << a[2] << ' ' << w << endl;
```

```
    fun(a, w);
```

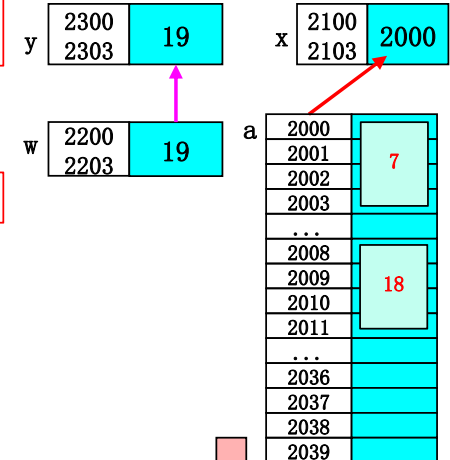
```
    cout << a[2] << ' ' << w << endl;
```

```
    return 0;
```

```
}
```

形参本质是指针，
用指针法表示x[2]

传参时，函数传地址，简单变量传值



//第05模块, 数组名及简单变量做函数参数, 验证形参修改是否影响实参

```
#include <iostream>
```

```
using namespace std;
```

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

```
{
```

```
    *(x+2)=37; //x[2]=37;
```

```
    y=45;
```

```
}
```

```
int main()
```

```
{    int a[10] = {7, -2, 18, 25}, w=19;
```

```
    cout << a[2] << ' ' << w << endl;
```

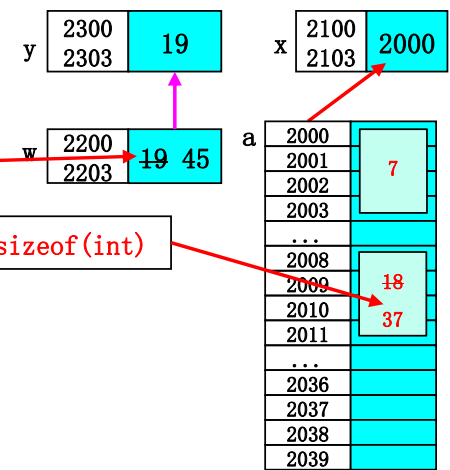
```
    fun(a, w);
```

```
    cout << a[2] << ' ' << w << endl;
```

```
    return 0;
```

```
}
```

2000+2*sizeof(int)





§ 6. 指针基础

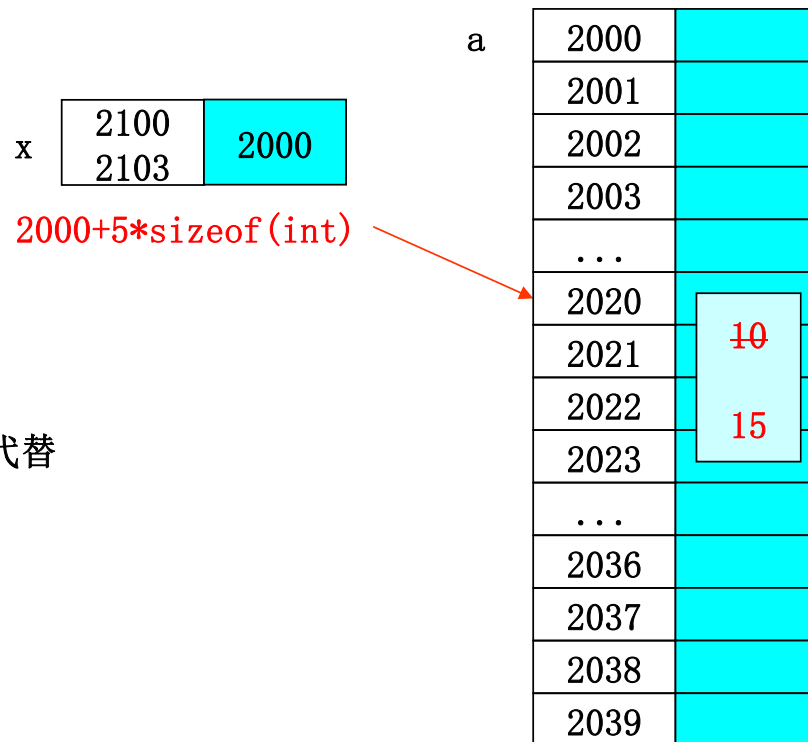
6.3. 一维数组与指针

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
  ...
}
```



★ 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. 指针基础

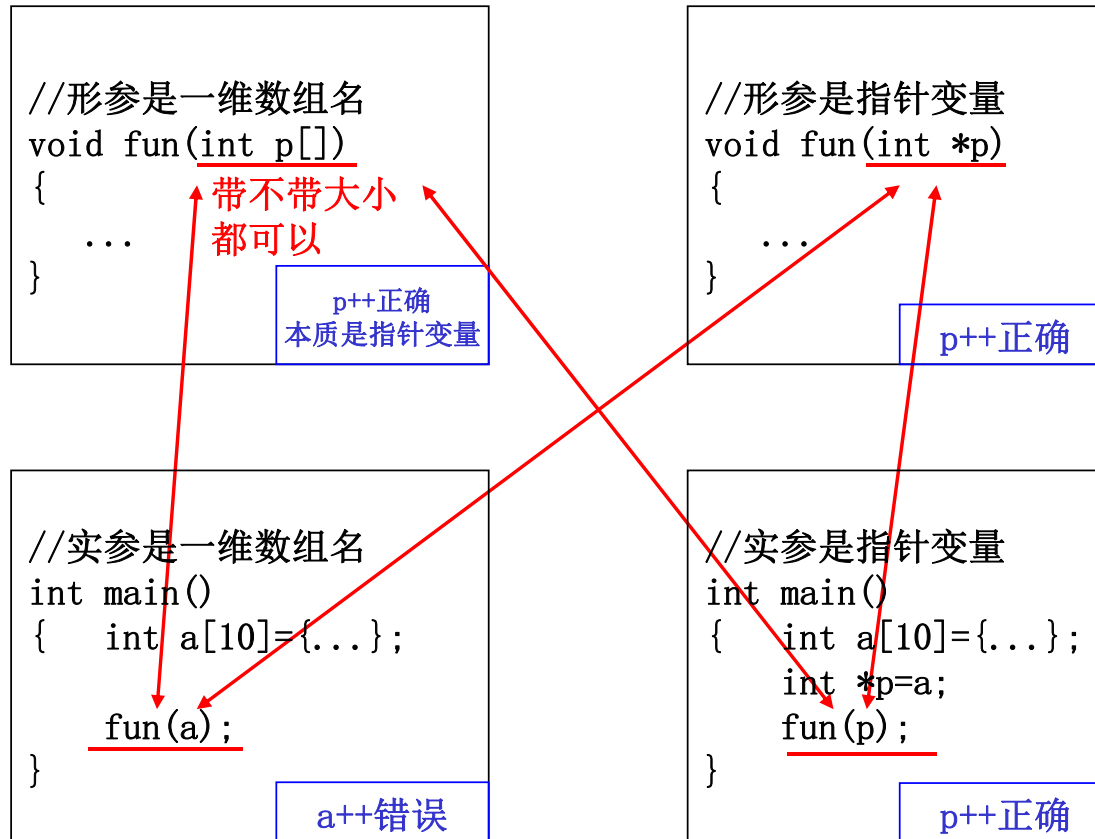
6.3. 一维数组与指针

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

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

这些内容与第04模块中简单变量做实形参的概念不同，第06模块指针中再详细解释

★ 形参为相应类型的一维数组

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

★ 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，因此实形参类型必须完全相同，否则编译错

形参指针变量p的基类型必须与实参数组的类型一致，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. 指针基础

6.3. 一维数组与指针

6.3.4. 指针法引用一维数组元素

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
- ★ 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
- ★ 实参数组也可以用指向它的指针变量来代替
- ★ 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

二维/多维数组与指针，是荣誉课内容，本课程不再展开



§ 6. 指针基础

6. 4. 字符串与指针

6. 4. 1. 字符指针的定义及使用 (重复之前的内容, 基类型为char/unsigned 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

*p='B'; 值, 相当于ch='B'

赋值语句

char ch, *p=&ch; 定义时赋初值

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



§ 6. 指针基础

6. 4. 字符串与指针

6. 4. 2. 用字符指针指向字符串

数组模块中:

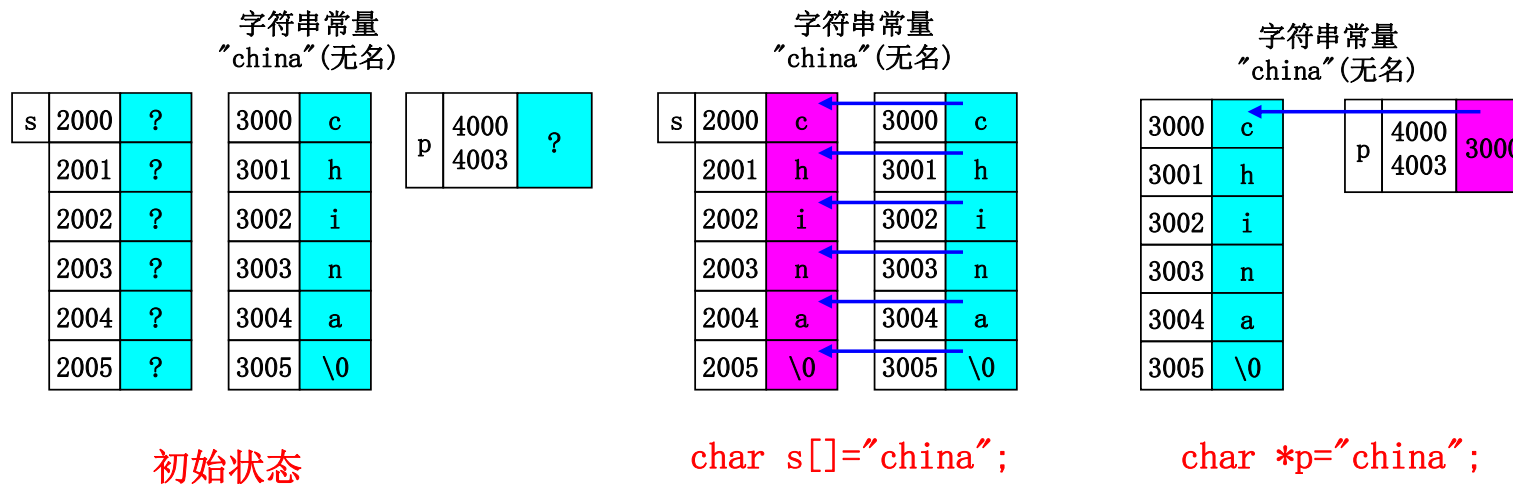
```
char s[]="china"; //一维字符数组, 缺省为6  
string s="china"; //C++类变量表示, 暂不讨论
```

本模块:

```
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. 指针基础

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=(char *)"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?
后续解决

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'; ✓
```

```
cpp-demo.cpp - 2 x (全局范围)
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      char s[10];
6      s = "china";
7
8      *s = "china";
9
10     *s = 'c';
11 }
```

(6,4): error C3863: 不可指定数组类型“char [10]”
(8,14): error C2440: “=”: 无法从“const char [6]”转换为“char”
(8,7): message : 没有使该转换得以执行的上下文



§ 6. 指针基础

6. 4. 字符串与指针

6. 4. 3. 字符指针的打印

```
#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>
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型的地址要转为非char？

假设地址A
=地址A+2 2字节
假设地址B
=地址B+1 1字节

★ 本页重复6. 2. 4的内容



§ 6. 指针基础

6. 4. 字符串与指针

6. 4. 4. 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]	8种表示方式中有一种 是错误的？哪种？
*(a+i)	*(p+i)	
*a++	*p++	
(*a)++	(*p)++	



§ 6. 指针基础

6.4. 字符串与指针

6.4.4. 下标法与指针法处理字符串

例：将str1的内容复制给str2

```
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 << "str1:" << p1 << endl
        << "str2:" << p2 << endl;
  return 0;
}
```

例：将str1的内容复制给str2

```
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

例：将str1的内容复制给str2

```
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. 指针基础

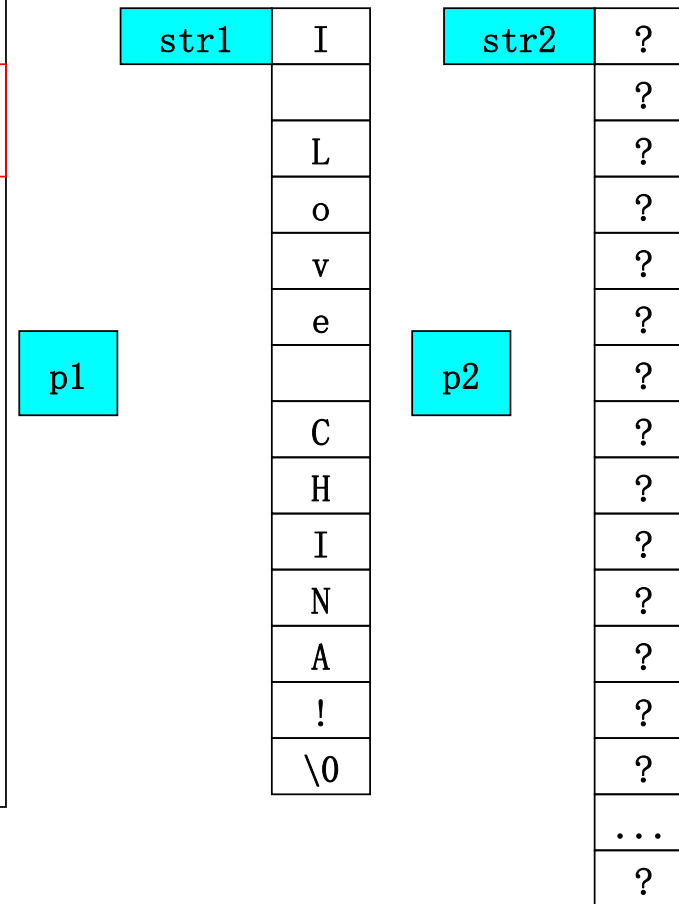
6. 4. 字符串与指针

6. 4. 4. 下标法与指针法处理字符串

例：将str1的内容复制给str2

```
int main()
{
    char str1[]="I Love CHINA!";
    char *p1,*p2;

    p1=str1;
    p2=str2;
    for(;*p1!='\0';p1++,p2++)
        *p2=*p1;
    *p2='\0';
    p1=str1;
    p2=str2;
    cout ...
    return 0;
}
```





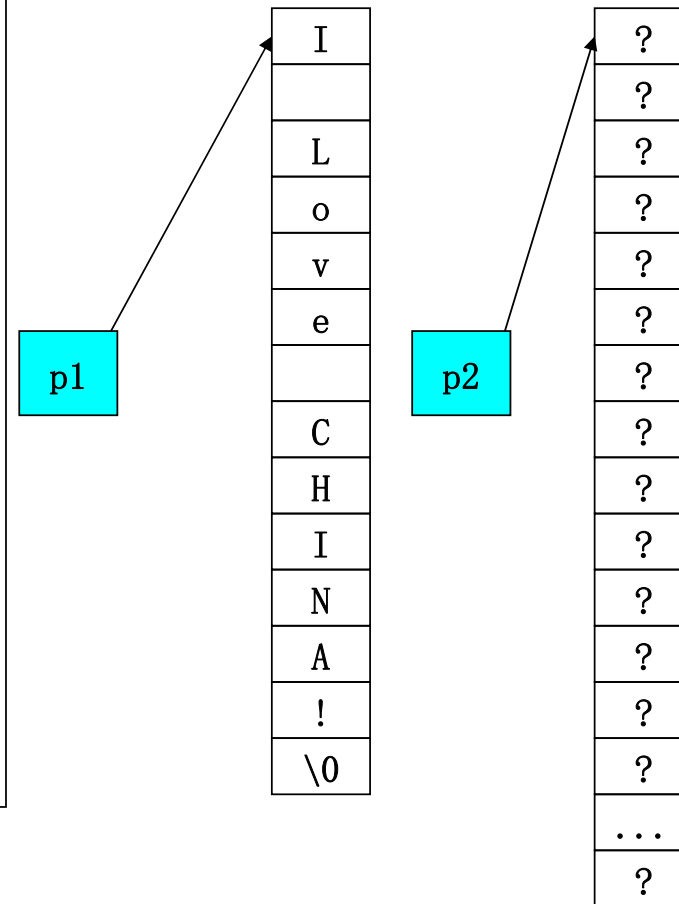
§ 6. 指针基础

6. 4. 字符串与指针

6. 4. 4. 下标法与指针法处理字符串

例：将str1的内容复制给str2

```
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;
}
```





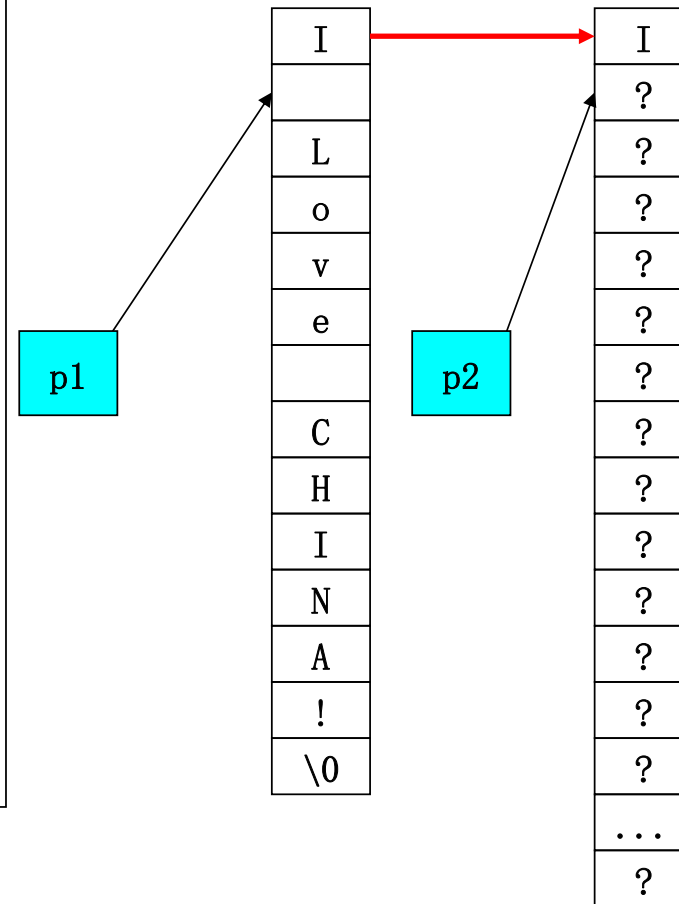
§ 6. 指针基础

6. 4. 字符串与指针

6. 4. 4. 下标法与指针法处理字符串

例：将str1的内容复制给str2

```
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;
}
```





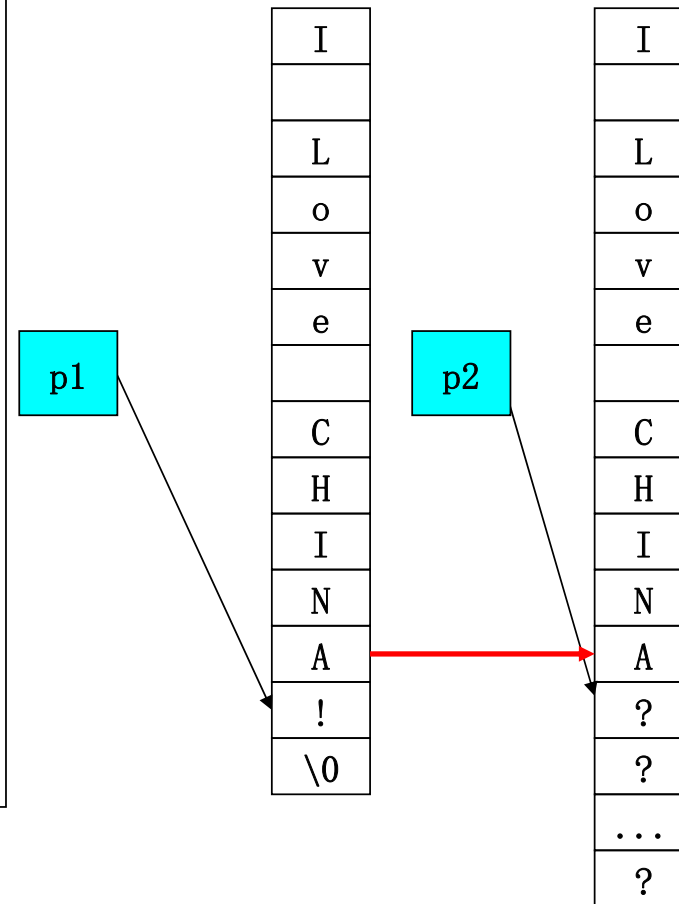
§ 6. 指针基础

6. 4. 字符串与指针

6. 4. 4. 下标法与指针法处理字符串

例：将str1的内容复制给str2

```
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. 指针基础

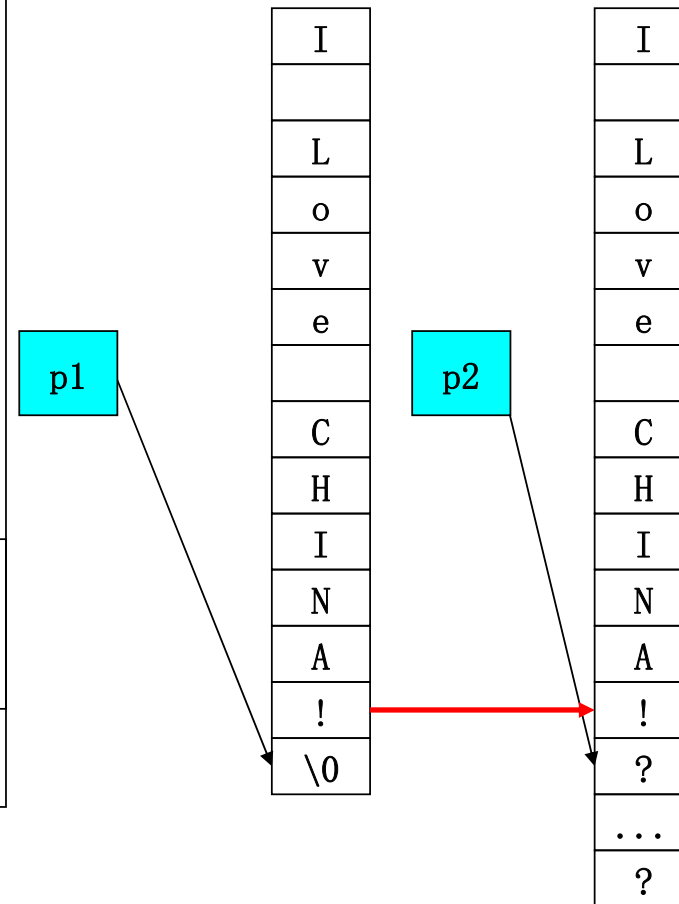
6. 4. 字符串与指针

6. 4. 4. 下标法与指针法处理字符串

例：将str1的内容复制给str2

```
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. 指针基础

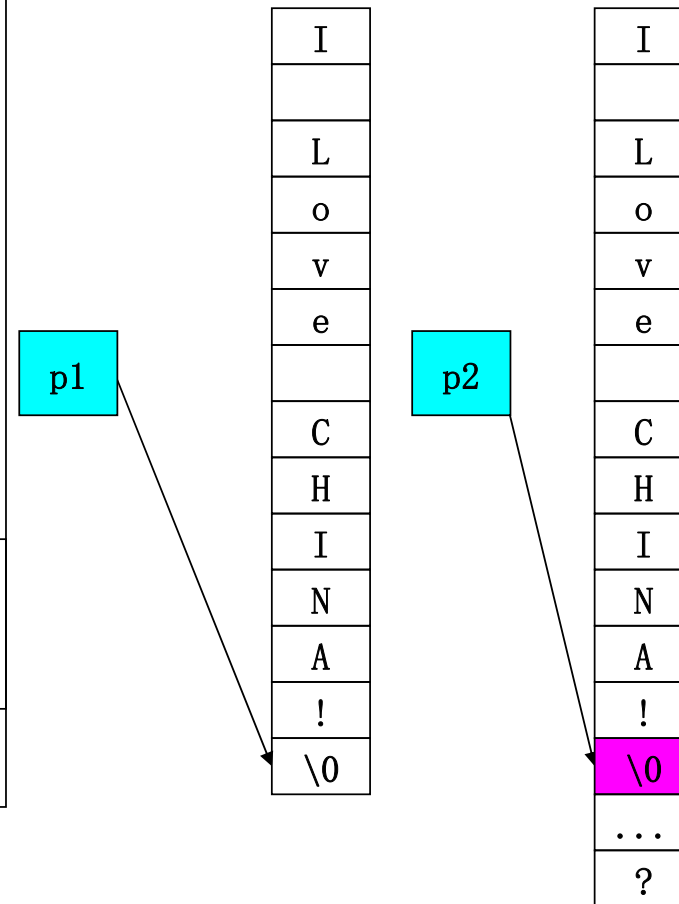
6. 4. 字符串与指针

6. 4. 4. 下标法与指针法处理字符串

例：将str1的内容复制给str2

```
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. 指针基础

6. 4. 字符串与指针

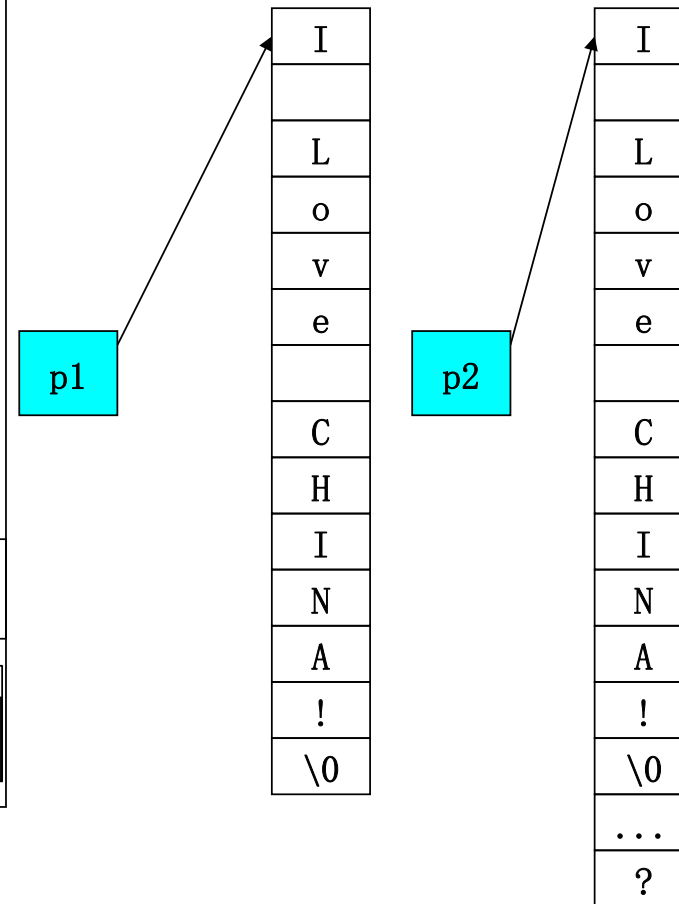
6. 4. 4. 下标法与指针法处理字符串

例：将str1的内容复制给str2

```
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
并打印

Microsoft Visual Studio 调试控制台
str1=I Love CHINA!
str2=I Love CHINA!





§ 6. 指针基础

6. 4. 字符串与指针

6. 4. 4. 下标法与指针法处理字符串

例：将str1的内容复制给str2

```
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;
}
```

1、判断
2、赋值
3、++到下一元素
循环结束,\0未赋

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

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

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

几种等价表示



§ 6. 指针基础

6. 4. 字符串与指针

6. 4. 4. 下标法与指针法处理字符串

例：将str1的内容复制给str2

```
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

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

虽不完全等价
但都是正确的

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. 指针基础

6. 4. 字符串与指针

6. 4. 4. 下标法与指针法处理字符串

例：将str1的内容复制给str2

```
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`，则读非法，但写仍在`str2`的合理
范围内，错误可能不体现

若`x>19`，则读非法，且对`str2`的赋值会
超过数组长度20，会导致非法写，
出错概率大于`x<=19`



§ 6. 指针基础

6. 4. 字符串与指针

6. 4. 4. 下标法与指针法处理字符串

<pre>#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; }</pre>	<p>完全正确 地址str2</p> <p>地址str2+13 13 I love CHINA!</p>	<pre>#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; }</pre>	<p>完全正确 地址str2</p> <p>地址str2 0 空行</p>
<pre>#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; }</pre>	<p>目前正确 地址str2</p> <p>地址str2+13 13 I love CHINA!</p>	<pre>#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; }</pre>	<p>运气好正确 本质不正确!!! 地址str2</p> <p>地址str2+x x (不确定) 空行</p>



§ 6. 指针基础

6. 4. 字符串与指针

6. 4. 5. 指向字符数组的指针作函数参数 (四种组合, 同前)

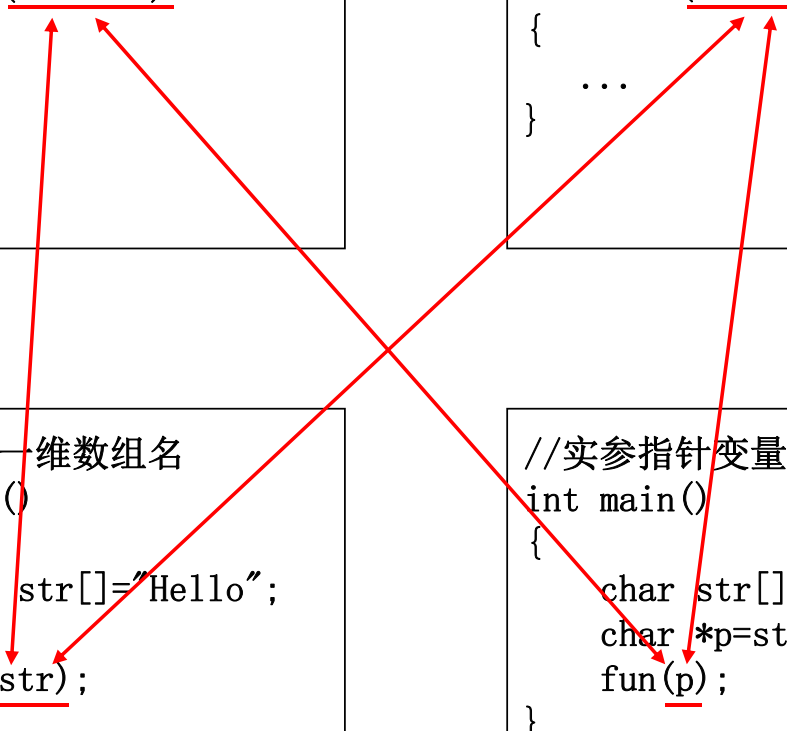
```
//形参是指针变量  
void fun(char *s)  
{  
    ...  
}
```

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

三种形式均可
[] : 空
[6] : 与实参大小一致
[123]: 与实参大小不一致

```
//实参是一维数组名  
int main()  
{  
    char str[]="Hello";  
    ...  
    fun(str);  
}
```

```
//实参指针变量  
int main()  
{  
    char str[]="Hello";  
    char *p=str;  
    fun(p);  
}
```





§ 6. 指针基础

6.4. 字符串与指针

6.4.6. 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

s	2000	预留空间
	2079	

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

VS2019编译报error

其他编译器能运行，但本质是错的

s	3000 3003	?(假设5000)
---	--------------	-----------

设键盘输入10个字符，则会覆盖
5000-5010的11字节的空间(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;
```

```
cpp-demo.cpp  x  
cpp-demo (全局范围)  
1  #include <iostream>  
2  using namespace std;  
3  int main()  
4  {  
5      char *s;  
6      cin >> s;  
7      return 0;  
8  }  
(6): error C4700: 使用了未初始化的局部变量“s”
```

```
cpp-demo.cpp  
1  #include <iostream>  
2  using namespace std;  
3  int main()  
4  {  
5      char *s;  
6      cin >> s;  
7      return 0;  
8  }  
编译及运行结果  
暂时看不出问题  
Hello  
Process exited after 6.662 seconds with return value 0  
请按任意键继续. . .
```




§ 6. 指针基础

6.4. 字符串与指针


6.4.6. 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
	2001	h
	2002	i
	2003	n
	2004	a
	2005	\0

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

p	4000 4003	3000
---	--------------	------

- ★ 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"; ✗
```

```
char *s;
```

```
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	

s[0]='c'; **✗ 修改常量值**

s[1]=' h' ; ✗

```
s[2]=' i' ; ✗
```

s[3]='n' ; x

s[4]=' a' ; **x**

s[5]=' \0' ; **x**

s	2100 2103	3000
---	--------------	------

字符串常量
"china"(无名)

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

- ### ★ 数组首地址的值不可变，指针的值可变



§ 6. 指针基础

6. 4. 字符串与指针

6. 4. 6. 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="china"; ✗
```

```
*p='c'; ?
```

是否正确?
后续解决

p表示取地址, 将字符串常量的首地址赋给p

*p表示取值, 基类型是char, 因此不能是字符串

编译正确, 能否正确执行视情况而定, 具体例子后面会给出

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
```

error C4700: 使用了未初始化的局部变量“p”

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

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

已退出, 代码为 -1073741819。

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



§ 6. 指针基础

6.5. 返回指针值的函数

6.5.1. 定义

返回基类型 *函数名（形参表）

```
int *fun(int x)
```

```
float *function(char ch)
```

6.5.2. 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
}
```

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. 指针基础

6.5. 返回指针值的函数

6.5.1. 定义

返回基类型 *函数名 (形参表)

```
int *fun(int x)
```

```
float *function(char ch)
```

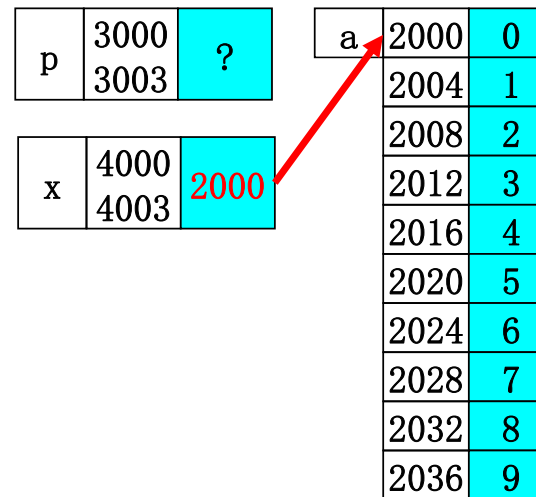
6.5.2. 使用

★ 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. 指针基础

6.5. 返回指针值的函数

6.5.1. 定义

返回基类型 *函数名 (形参表)

int *fun(int x)

float *function(char ch)

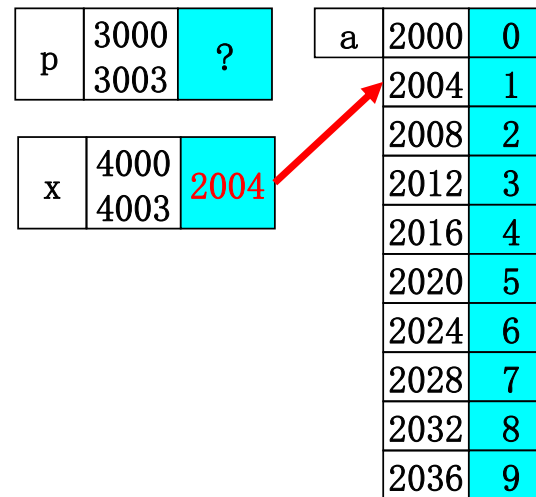
6.5.2. 使用

★ 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. 指针基础

6.5. 返回指针值的函数

6.5.1. 定义

返回基类型 *函数名 (形参表)

```
int *fun(int x)
```

```
float *function(char ch)
```

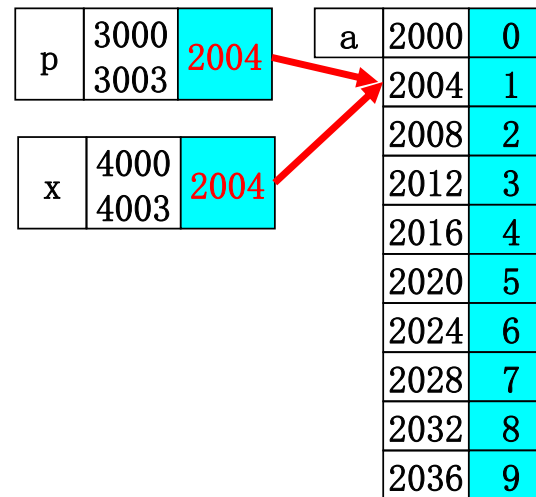
6.5.2. 使用

★ 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. 指针基础

6.5. 返回指针值的函数

6.5.1. 定义

返回基类型 *函数名 (形参表)

```
int *fun(int x)
```

```
float *function(char ch)
```

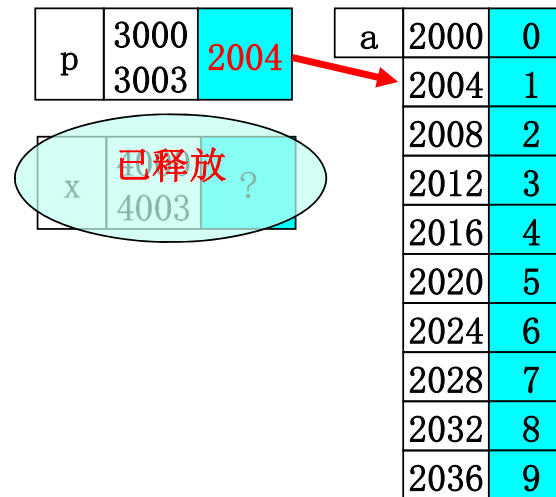
6.5.2. 使用

★ 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. 指针基础

6.5. 返回指针值的函数

6.5.1. 定义

返回基类型 *函数名 (形参表)

```
int *fun(int x)
```

```
float *function(char ch)
```

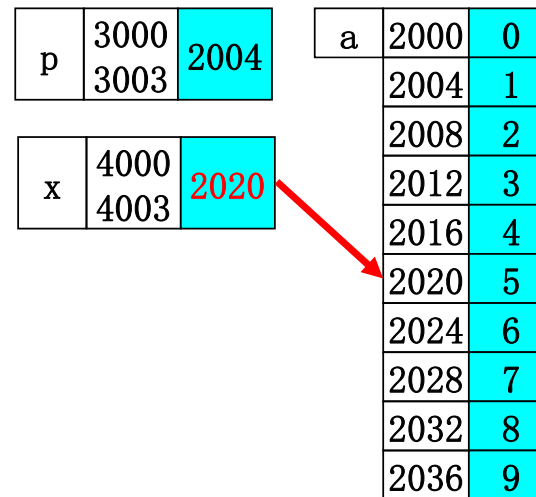
6.5.2. 使用

★ 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. 指针基础

6.5. 返回指针值的函数

6.5.1. 定义

返回基类型 *函数名（形参表）

```
int *fun(int x)
```

```
float *function(char ch)
```

6.5.2. 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（具体分析略）

```
int *fun()
{
    int k=10;
    return &k; //不允许
}
```

```
int *fun(int k)
{
    return &k; //不允许
}
```

warning C4172: 返回局部变量或临时变量的地址: k



§ 6. 指针基础

6.5. 返回指针值的函数

6.5.2. 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;
    return 0;
}
```

Microsoft Visual Studio 调试控制台
10

警告 C4172: 返回局部变量或临时变量的地址: k

k	3000	10
p	2000	?

k	3000	10
p	2000	3000

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
=> *p不可信

1、VS2019下有编译警告
2、读非法空间而未写，不会死机

问：如果必须返回局部变量的值，如何做？
答：设为静态局部即可 (k不释放)



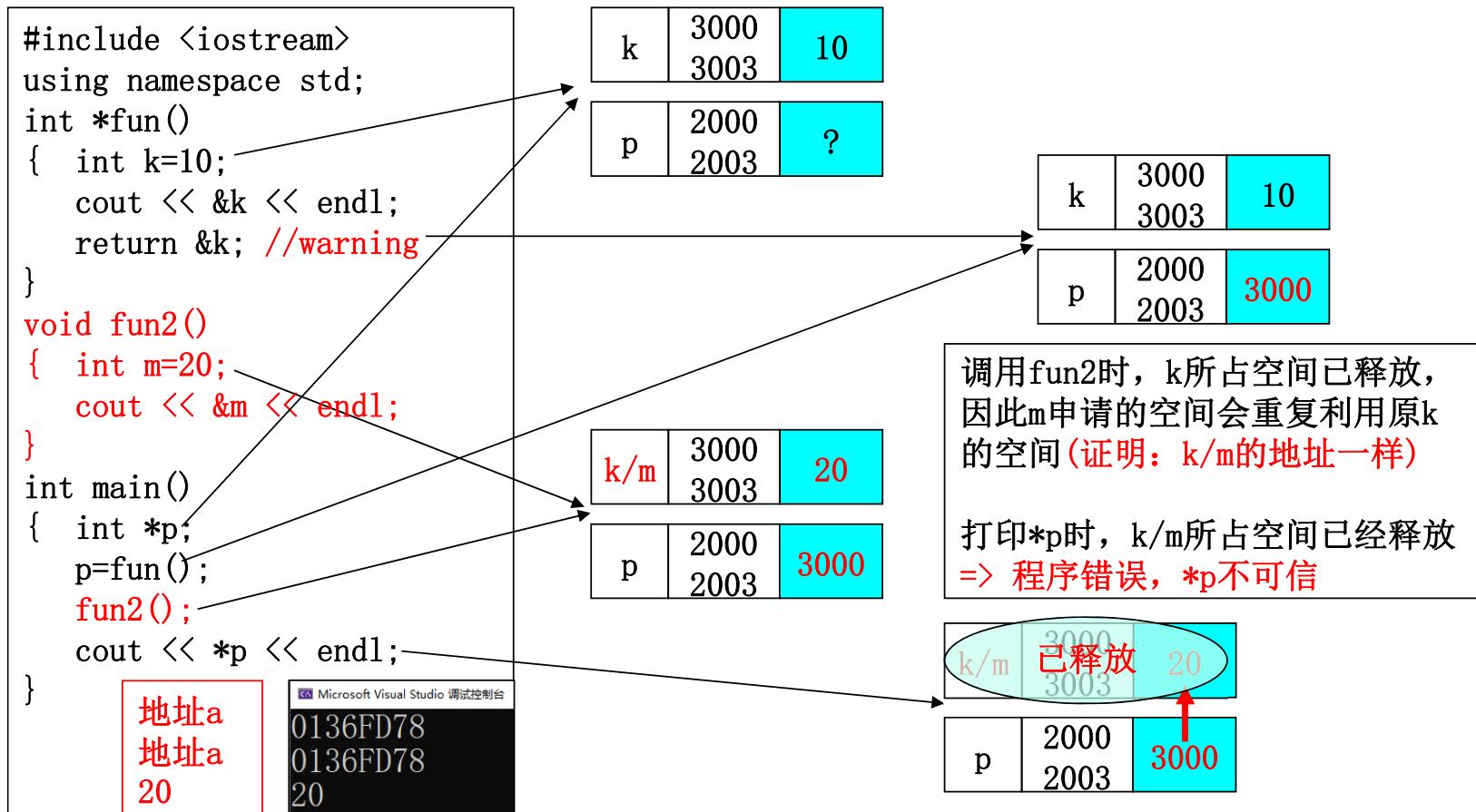
§ 6. 指针基础

6.5. 返回指针值的函数

6.5.2. 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





§ 6. 指针基础

6.5. 返回指针值的函数

6.5.2. 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;

void fun()
{
    int k=10;
    cout << &k << endl;
}

int main()
{
    int i;
    for(i=0; i<10; i++)
        fun();
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
009DFDC8
009DFDC8
009DFDC8
009DFDC8
009DFDC8
009DFDC8
009DFDC8
009DFDC8
009DFDC8
009DFDC8
```

上例证明了自动变量k每次函数调用完成后会释放空间

本例证明了再次调用时，k分配的空间
大概率是上次分配的空间

(04模块中：两次分配不保证同一空间)



§ 6. 指针基础

6.5. 返回指针值的函数

6.5.2. 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
- ★ 可以通过返回数组首地址（指针）的方式来返回整个数组

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

char *my_strcpy(char *dst, const char *src)
{
    int i;
    for (i=0; src[i]; i++)
        dst[i] = src[i];
    dst[i] = 0;
    return dst;
}

int main()
{
    char s1[]="student", s2[]="hello";
    cout << s1 << endl;
    cout << my_strcpy(s1, s2) << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
student
hello
```



§ 6. 指针基础

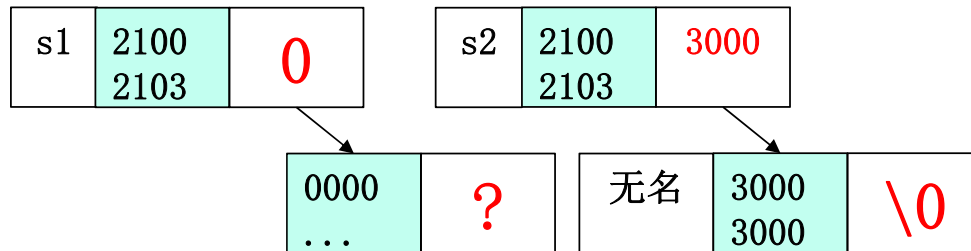
6.6. 空指针NULL

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

NULL与空字符串的区别:

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

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



```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     char s3[] = "";
6     char s4[] = NULL; //编译报错
7 }
```

(6,21): error C2440: “初始化”: 无法从“int”转换为“char []”
(6,15): message : 没有转换为数组类型, 但有转换为数组的引用或指针

`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. 指针基础

6.6. 空指针NULL

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

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

已退出，代码为 -1073741819。

- ★ 自行实现类似功能的字符串处理函数时，
可对NULL进行特殊处理(具体见作业要求)
不是标准，只是为了强行与系统函数不同
- 求长度时为0
 - 复制、连接、拷贝时当做空串进行处理



§ 6. 指针基础

6.7. 引用 (C++新增)

6.7.1. 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b
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
```

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

```
int &b[3];           //错误，不能声明引用数组
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. 指针基础

6.7. 引用 (C++新增)

6.7.1. 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



```
//例：简单变量的引用  
#include <iostream>  
using namespace std;  
int main()  
{   int a=10, &b=a;  
    a=a*a;  
    cout << a << " " << b << endl;  
    b=b/5;  
    cout << a << " " << b << endl;  
    return 0;  
}
```

本例无任何实用价值
1、多定义一个名称
2、两者容易混淆

100 100

20 20



§ 6. 指针基础

6. 7. 引用 (C++新增)

6. 7. 1. 引用的基本概念

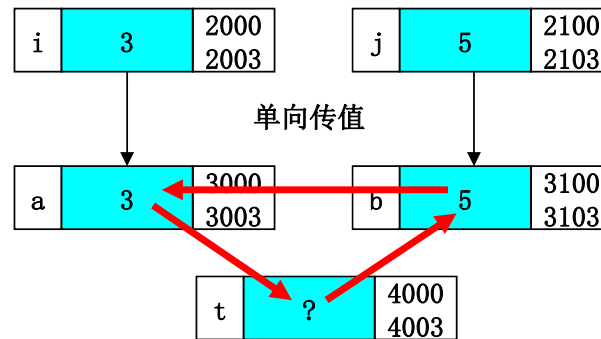
6. 7. 2. 引用作函数参数

例：两数交换

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

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

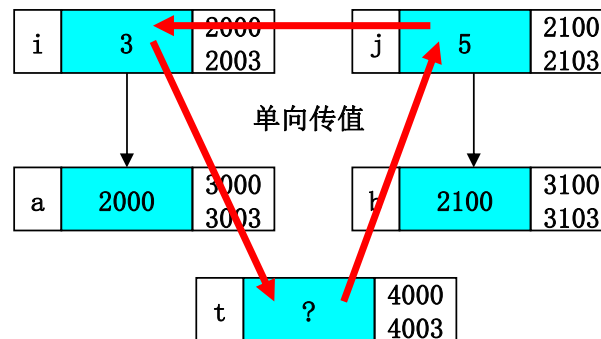
直接传值
错误



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

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

传地址
正确





§ 6. 指针基础

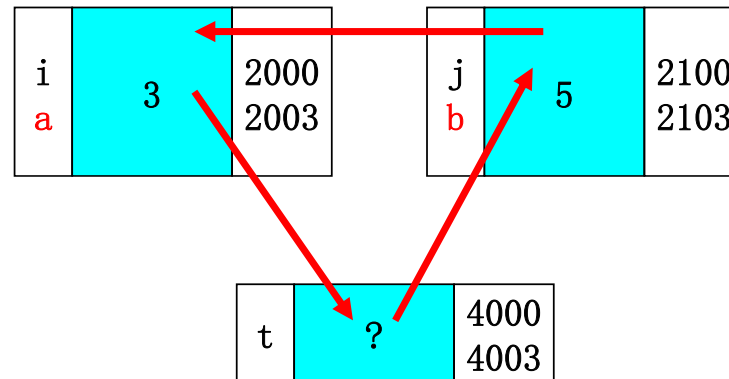
6.7. 引用 (C++新增)

6.7.1. 引用的基本概念

6.7.2. 引用作函数参数

例：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;  
}
```



★ 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 (地址传递方式)



§ 6. 指针基础

6.7. 引用 (C++新增)

6.7.1. 引用的基本概念

6.7.2. 引用作函数参数

★ 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 (地址传递方式)

● C++函数参数传递的两种方式

◆ 传值：单向传值，实形参分占不同空间

```
void fun(int x)
{ ...
}

int main()
{ int k = 10;
  fun(k);
}
```

形参的变化
不影响实参

```
void fun(int *x)
{ ...
}

int main()
{ int k=10;
  fun(&k);
}
```

可通过形参间接
访问实参，但本质
仍是单向传值

◆ 传址：实形参重合，对形参的访问就是对实参的访问

```
void fun(int *x)
{ ...
}

int main()
{ int k[10] = {...};
  fun(k);
}
```

对形参数组
的修改影响
实参数组

```
void fun(int &x)
{ ...
}

int main()
{ int k=10;
  fun(k);
}
```

对形参的访问
就是对实参的访问



§ 6. 指针基础

6.7. 引用 (C++新增)

6.7.1. 引用的基本概念

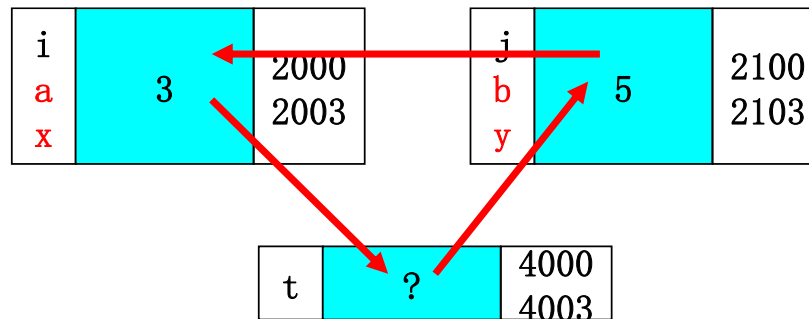
6.7.2. 引用作函数参数

★ 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 (地址传递方式)

★ 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. 指针基础

6.7. 引用 (C++新增)

6.7.2. 引用作函数参数

- ★ 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 (地址传递方式)
- ★ 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. 指针基础

6.7. 引用 (C++新增)

6.7.3. 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
(VS/Dev都是C++编译器，兼容编译纯C，以后缀名.c/.cpp来区分如何编译)
- ★ **对于计算机底层而言，仍需要透彻理解指针!!!**