C程序设计基础

Introduction to C programming
Lecture 11: Pointer II

张振国 zhangzg@sustech.edu.cn

南方科技大学/理学院/地球与空间科学系

Review on L10 Pointer I

Function-Recursion

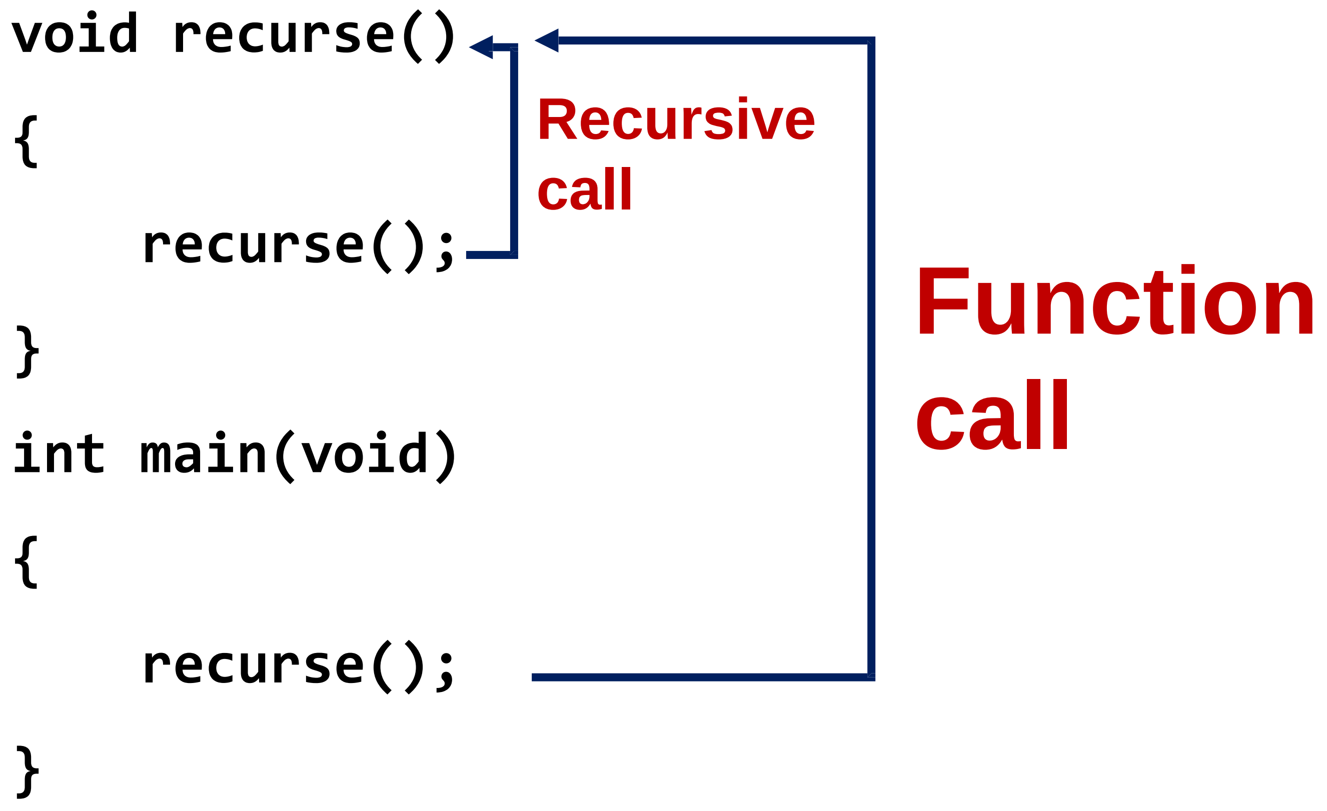
Memory address

Pointer

Pointer and Array

Recursion

Recursion is to repeat the same procedure again and again



```
void recurse()
{
    recurse();
}
int main(void)
{
    recurse();
}
```

Recursion

```
void recurse(int n)
{
   if (n == 0) return;
   recurse(n-1);
}
int main(void)
{
    int n = 100;
    recurse(n);
}
```

Which one can leave?

```
void recurse(int n)
{
    recurse(n-1);
    if (n == 0) return;
}
int main(void)
{
    int n = 100;
    recurse(n);
}
```

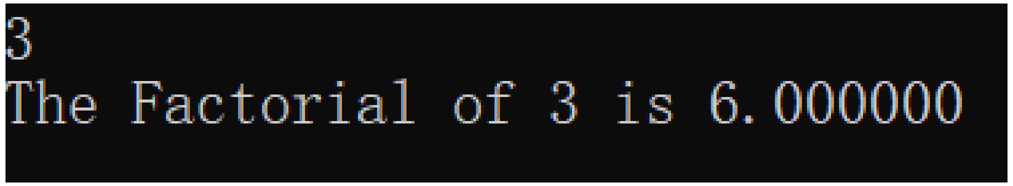
Case study: factorial calculator

```c
#include <stdio.h>

double factorial(int i)
{
    if (i <= 1)
    {
        return 1;
    }
    return i * factorial(i - 1);
}

int main(void)
{
    int input;
    scanf("%d", &input);
    printf("The Factorial of %d is %f\n", input,
factorial(input);
}
```

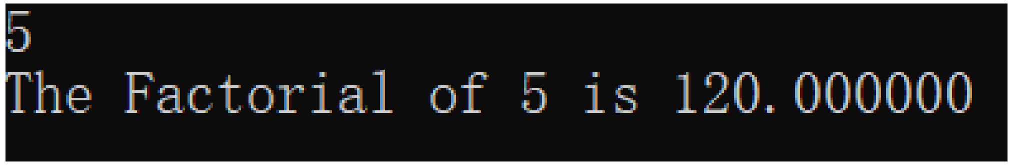
Case: use recursion to design a factorial calculator.

$$n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$$

$$n! = n \times (n-1)!$$



```
3
The Factorial of 3 is 6.000000
```



```
5
The Factorial of 5 is 120.000000
```

Case study: Fibonacci series

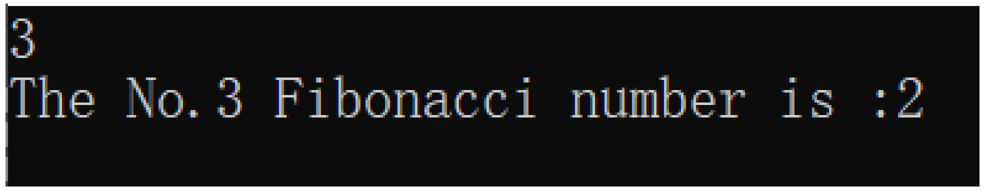
```
 #include <stdio.h>

 int fib(int input) {
        if (input <= 2) {
                return 1;//first two numbers are 1
        }
        return fib(input - 1) + fib(input - 2);
 }

int main(void) {
        int input = 0;
        scanf("%d", &input);
        fib(input);
        printf("The No.%d Fibonacci number is :%d",
input, fib(input));
}
```

Case: use recursion to implement a Fibonacci series.

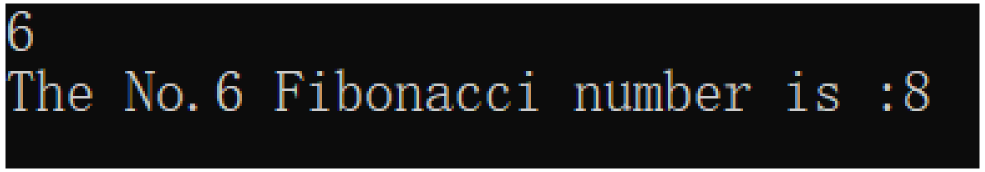
F(1)=1, F(2)=1,

F(n)=F(n - 1)+F(n - 2) （n ≥3， n ∈ N*）

1、1、2、3、5、8、13、21、34、…



```
3
The No.3 Fibonacci number is :2
```



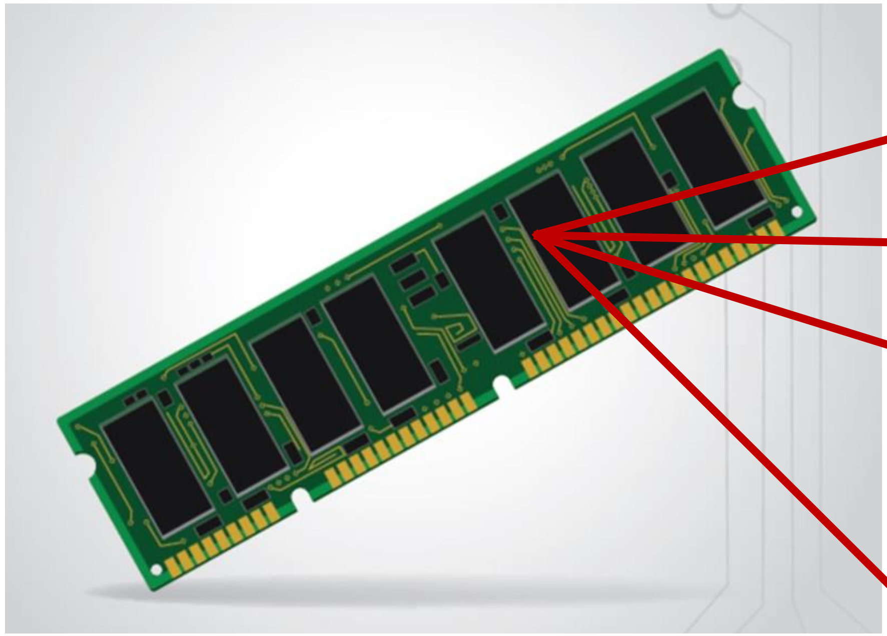
```
6
The No.6 Fibonacci number is :8
```

Memory address

How can we find a person?

Address

Memory address



Address	Content							
ffc1	0	1	0	0	1	0	1	1
ffc2	0	1	0	1	0	0	0	1
ffc3	1	0	0	0	1	1	1	0
⋮	⋮							
ffc9	1	1	0	0	0	0	0	1

Memory address

```
int a = 5;
int b = 2;
int c = 1;
```

Variable	Address	Content
a	ffc1	00000101
b	ffc2	00000010
c	ffc3	00000001

You can find the content by indexing the variable name or its address!

How to check variable address

Use **& (reference operator)** to check the variable address

```
#include <stdio.h>

main ()
{
    int var1;
    float var2;
    char var3;
    printf("Address of var1 variable: %x\n",&var1);
    printf("Address of var1 variable: %x\n",&var2);
    printf("Address of var1 variable: %x\n",&var3);
}
```

What is pointer?

Variable $\longrightarrow$ Address

What is pointer?

Pointer is a variable that stores the address of another variable.

```
type *var;
type *var2 = &var1;
```

```
int a;
float f;     } Stores value
char c;
```

```
int *a;
float *f;    } Stores address
char *c;
```

声明指针变量时必须指定
指针所指向变量的类型

Pointer declaration and definition

```
int a = 10;
int *b;
b = &a;
```

```
int a = 5;
int *b = &a;
```

Variable	Address	Content
a	ffc1	00001010
b	ffc2	ffc1

- a stores the value of 10
- b stores the address of a

How to interpret pointer?

int *b

b has data type int*

```
printf("%x", b);//address
```

int *b

*b has data type int

```
printf("%d", *b);//value
```

Pointer

```
int a = 5;
int *b = &a;
```

Variable	Address	Content
a	ffc1	00001010
b	ffc2	ffc1

难点：

- □指针所指向的值(5→10)；
- □指针的值(ffc1→ffc3，移动指针);
- □取地址(&)
- □取值(*)

Case of value swap

How to swap values between two variables?

```c
void swap(int v1, int v2)
{
    printf("Before: v1=%d, v2=%d\n", v1, v2);
    int temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
    printf("After: v1=%d, v2=%d\n", v1, v2);
}


int main(void)
{
    int a = 10, b = 5;
    printf("Before: a=%d, b=%d\n", a, b);
    swap(a, b);
    printf("After: a=%d, b=%d\n", a, b);
}
```

v1 = 10, v2 = 5

v1 = 5, v2 = 10

Changes inside function cannot influence outside

a = 10, b = 5

a = 10, b = 5

Case of value swap

Procedure

```c
void swap(int v1, int v2)
{
    printf("Before: v1=%d, v2=%d\n", v1, v2);
    int temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
    printf("After: v1=%d, v2=%d\n", v1, v2);
}


int main(void)
{
    int a = 10, b = 5;
    printf("Before: a=%d, b=%d\n", a, b);
    swap(a, b);
    printf("After: a=%d, b=%d\n", a, b);
}
```

Variable	Address	Content
a	ffc1	10
b	ffc2	5

Variable	Address	Content
a	ffc1	10
b	ffc2	5
v1	ffc3	10
v2	ffc4	5

Variable	Address	Content
a	ffc1	10
b	ffc2	5
v1	ffc3	5
v2	ffc4	10
temp	ffc5	10

Case of value swap

How to use pointer to swap values?

```
void swap(int *v1, int *v2)
{
    int temp;
    temp = *v1;
    *v1 = *v2;
    *v2 = temp;
}


int main(void) {
    int a = 10, b = 5;
    printf("Before: a=%d, b=%d\n", a, b);
    swap(&a, &b);
    printf("After: a=%d, b=%d\n", a, b);
}
```

*v1 = 10, *v2 = 5

*v1 = 5, *v2 = 10

Changes made to memory address influence outside

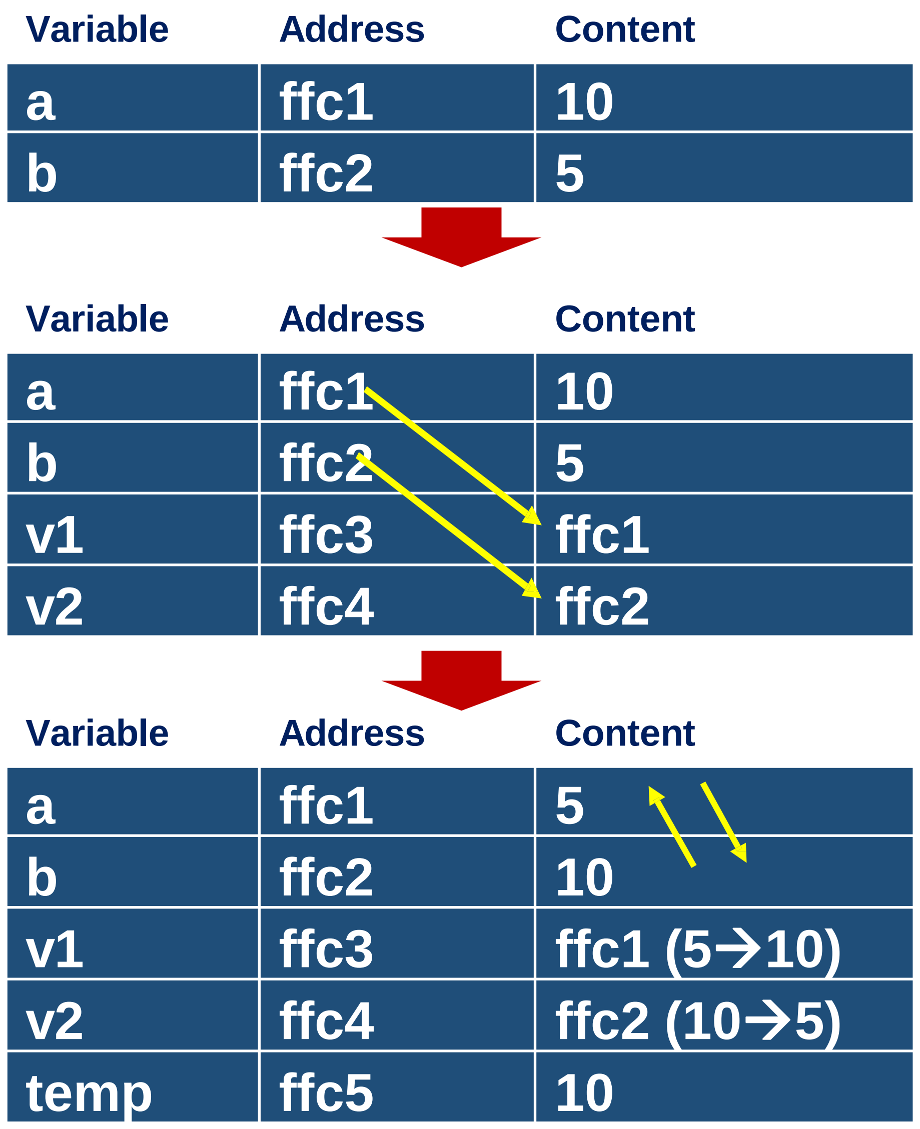
a = 10, b = 5

a = 5, b = 10

Case of value swap

Procedure

```
void swap(int *v1, int *v2)
{
   int temp;
   temp = *v1;
   *v1 = *v2;
   *v2 = temp;
}


int main(void) {
   int a = 10, b = 5;
   printf("Before: a=%d, b=%d\n", a, b);
   swap(&a, &b);
   printf("After: a=%d, b=%d\n", a, b);
}
```



Variable	Address	Content
a	ffc1	10
b	ffc2	5

Variable	Address	Content
a	ffc1	10
b	ffc2	5
v1	ffc3	ffc1
v2	ffc4	ffc2

Variable	Address	Content
a	ffc1	5
b	ffc2	10
v1	ffc3	ffc1 (5→10)
v2	ffc4	ffc2 (10→5)
temp	ffc5	10

Case of value swap

How to use pointer to swap values?

```
void swap(int *v1, int *v2)
{
   int *temp;
   temp = v1;
   v1 = v2;
   v2 = temp;
}


int main(void) {
   int a = 10, b = 5;
   printf("Before: a=%d, b=%d\n", a, b);
   swap(&a, &b);
   printf("After: a=%d, b=%d\n", a, b);
}
```

***v1 = 10, *v2 = 5**

***v1 = 5, *v2 = 10**

Changes made to memory address do not influence outside

a = 10, b = 5

a = 10, b = 5

Case of value swap

Procedure

```
void swap(int *v1, int *v2)
{
   int *temp;
   temp = v1;
   v1 = v2;
   v2 = temp;
}


int main() {
   int a = 10, b = 5;
   printf("Before: a=%d, b=%d\n", a, b);
   swap(&a, &b);
   printf("After: a=%d, b=%d\n", a, b);
}
```

Variable	Address	Content
a	ffc1	10
b	ffc2	5

Variable	Address	Content
a	ffc1	10
b	ffc2	5
v1	ffc3	ffc1
v2	ffc4	ffc2

Variable	Address	Content
a	ffc1	10
b	ffc2	5
v1	ffc3	ffc2
v2	ffc4	ffc1
temp	ffc5	ffc1

Content

Pointer points to array

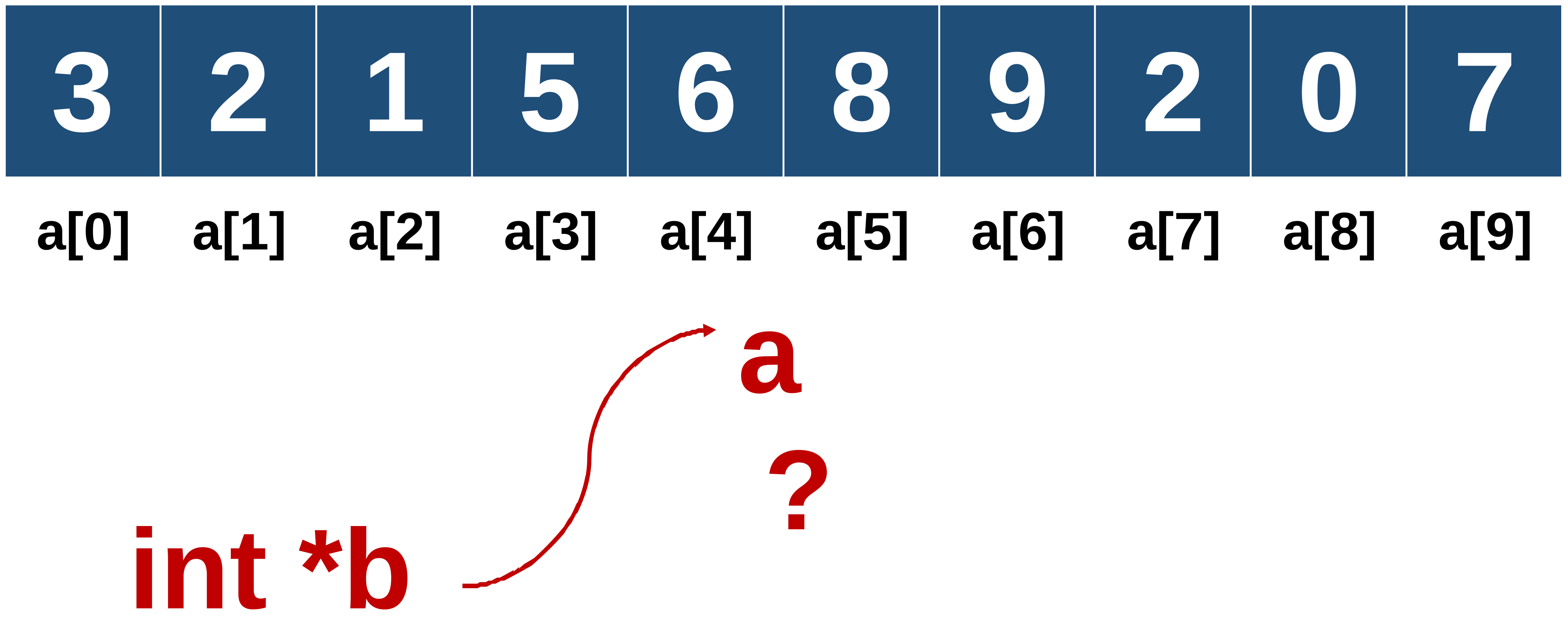
```
int a = 5;
int *b = &a;
```

$\downarrow$

Give the address of a to b!

Pointer points to array

int a[10] = {3, 2, 1, 5, 6, 8, 9, 2, 0, 7}; // length is 10



Pointer points to array

```
int a = 5;
int *b = &a;
```

↓

Give the address of a to b!

```
int a[10];
int *b = a;
```

↓

Give the address of **<u>first element of a</u>** to b!

```
int *b = &a[0];
```

Pointer points to array

```
int a[3]={1,2,3};
```

```
int *b = a;
```

or

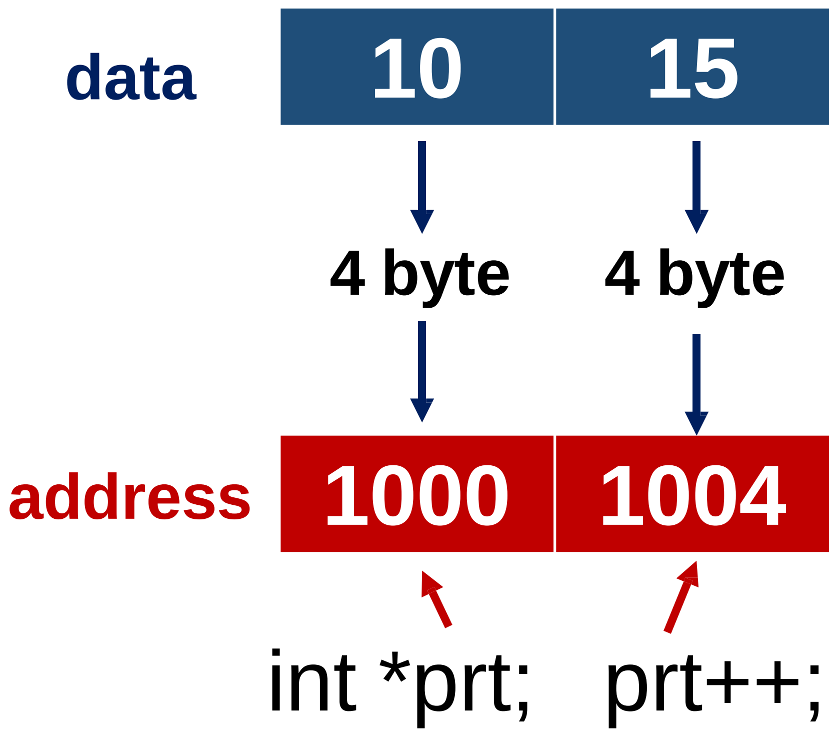
```
int *b = &a[0];
```

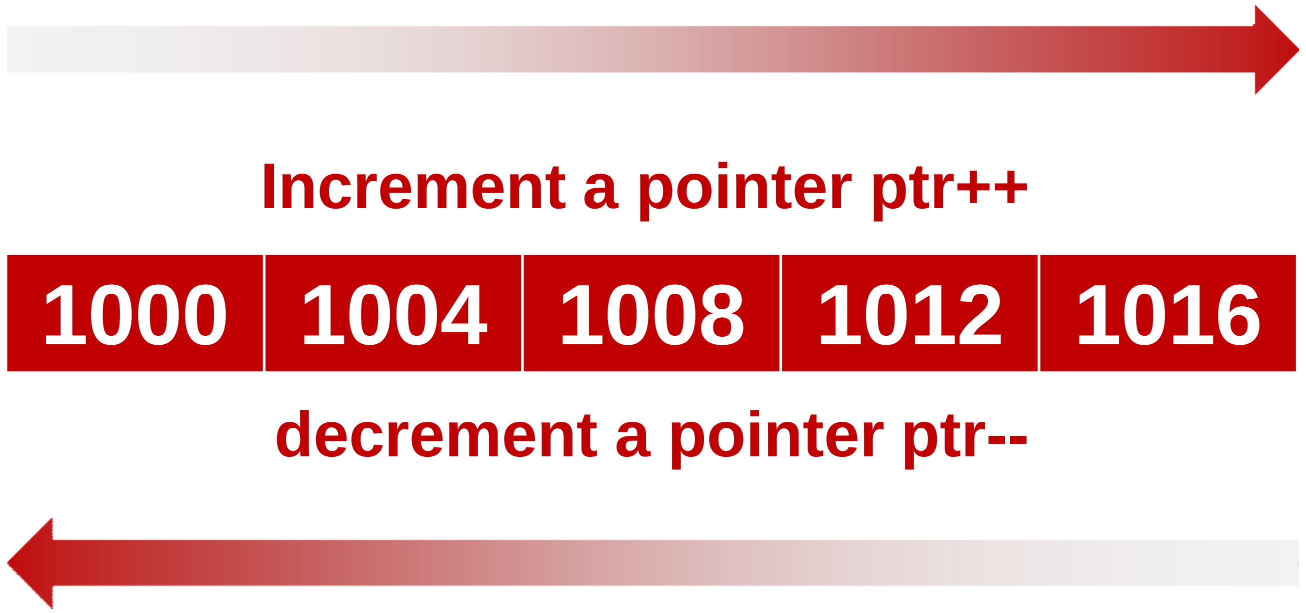
Array	Address	Content
a[0]	17d8f780	1
a[1]	17d8f784	2
a[2]	17d8f788	3
…	…	

Address of the first element is assigned to pointer

Pointer points to array

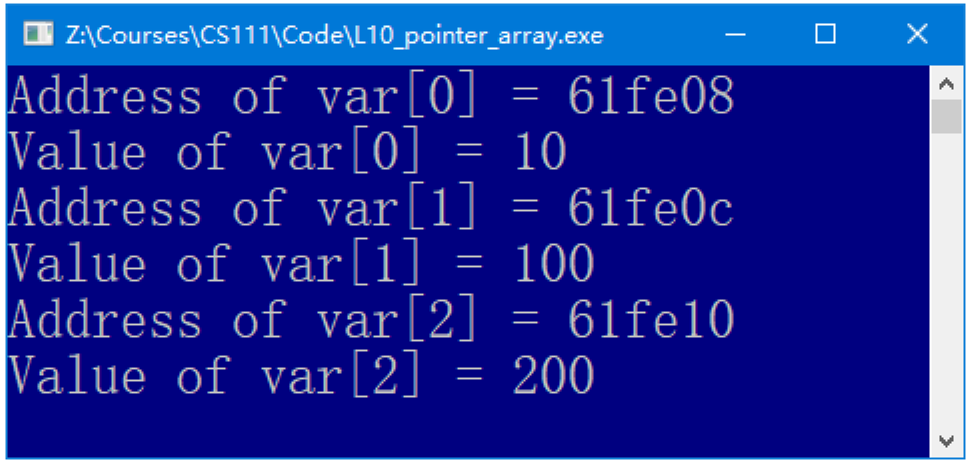
Four arithmetic operators that can be used on pointers: ++, --, +, -





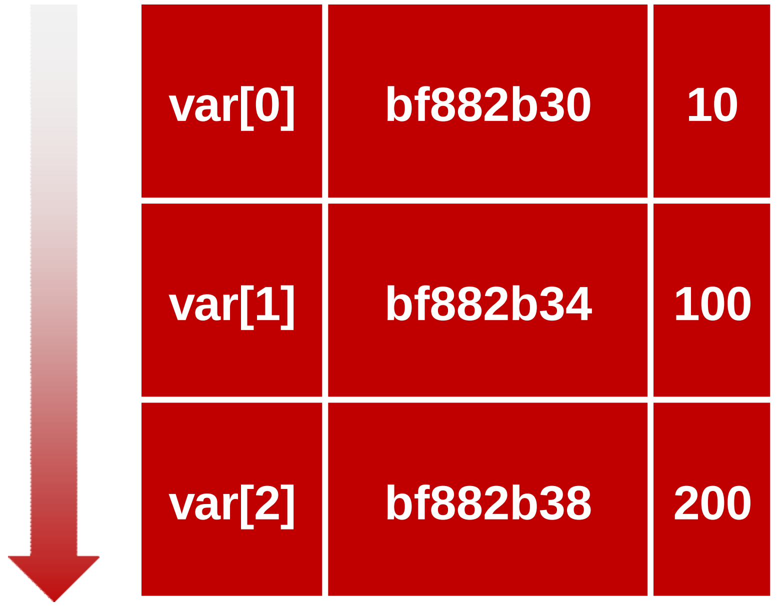
Pointer points to array

```c
#include <stdio.h>

int main (void)
{
    int var[] = {10, 100, 200};
    int *ptr = var; //&var[0]

    for ( int i = 0; i < 3; i++)
    {
        printf("Address of var[%d] = %x\n", i, ptr );
        printf("Value of var[%d] = %d\n", i, *ptr );

        ptr++; /* move to the next location */
    }
}
```



Z:\Courses\CS111\Code\L10_pointer_array.exe

```
Address of var[0] = 61fe08
Value of var[0] = 10
Address of var[1] = 61fe0c
Value of var[1] = 100
Address of var[2] = 61fe10
Value of var[2] = 200
```

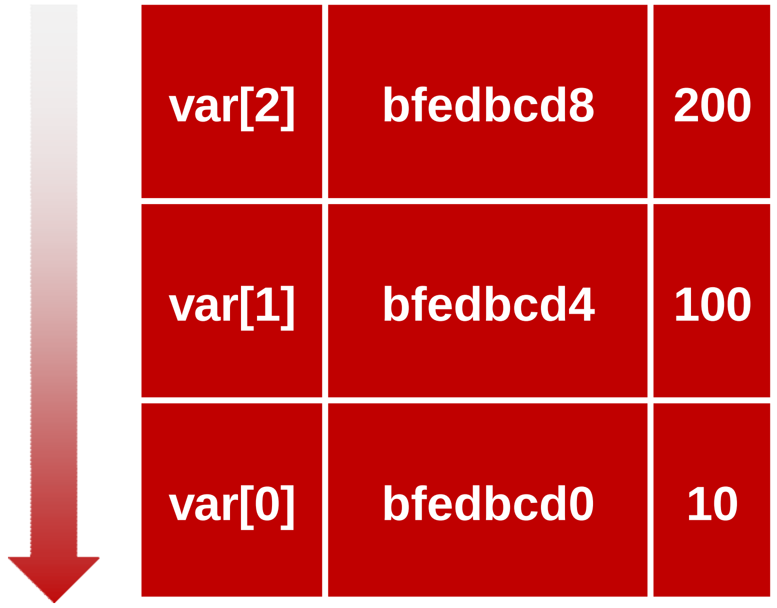


var[0]	bf882b30	10
var[1]	bf882b34	100
var[2]	bf882b38	200

Increments a pointer

Pointer points to array

```
#include <stdio.h>

int main (void)
{
    int var[] = {10, 100, 200};
    int *ptr = &var[3];

    for ( int i = 2; i >= 0; i--)
    {
        printf("Address of var[%d] = %x\n", i, ptr );
        printf("Value of var[%d] = %d\n", i, *ptr );

        ptr--; /* move to the previous location */
    }
}
```

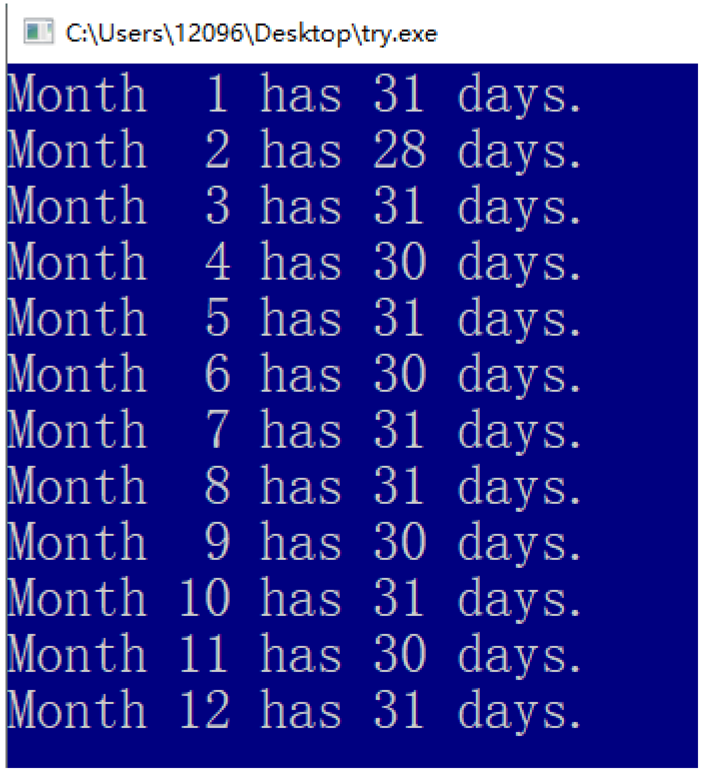


var[2]	bfedbcd8	200
var[1]	bfedbcd4	100
var[0]	bfedbcd0	10

Decrements a pointer

Pointer points to array

```
#include <stdio.h>
#define MONTHS 12

int main(void) {
	int days[MONTHS] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
	int index;

	for (index = 0; index < MONTHS; index++) {
		printf("Month %2d has %d days.\n", index + 1, *(days + index));
	}
	return 0;
}
```


C:\Users\12096\Desktop\try.exe

```
Month  1 has 31 days.
Month  2 has 28 days.
Month  3 has 31 days.
Month  4 has 30 days.
Month  5 has 31 days.
Month  6 has 30 days.
Month  7 has 31 days.
Month  8 has 31 days.
Month  9 has 30 days.
Month 10 has 31 days.
Month 11 has 30 days.
Month 12 has 31 days.
```

days + index 与(days + index)不一样！！！

间接运算符（*）优先级高于+

Pointer points to array

Use pointer to compare memory address: >, <, ==

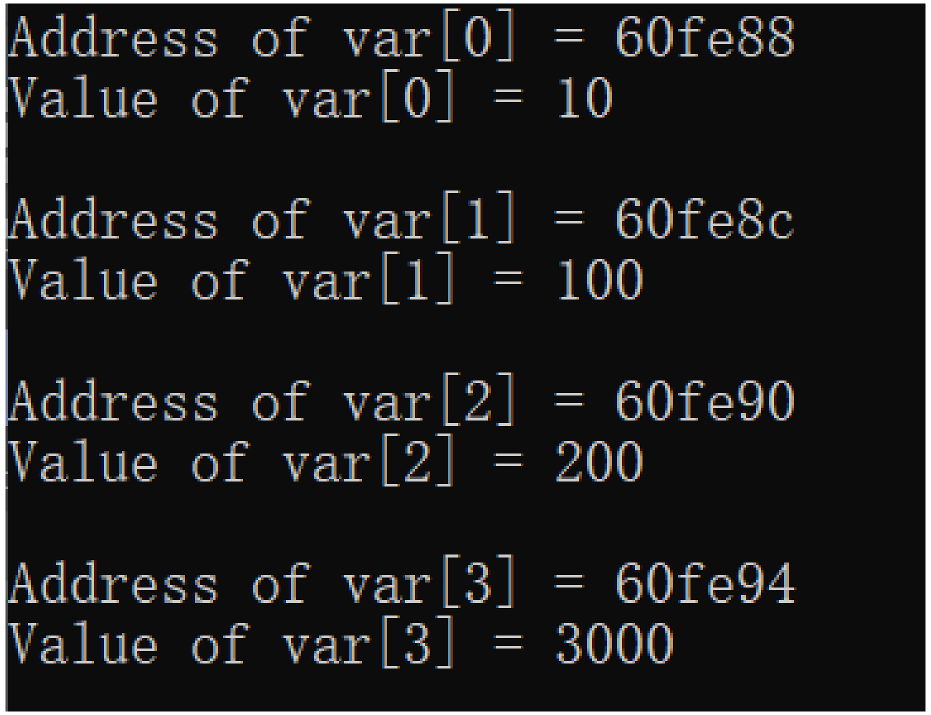
```
int var[] = {10, 100, 200};

int *ptr1 = &var[0];
int *ptr2 = &var[0];
int *ptr3 = &var[1];
```

```
ptr1 == ptr2                ptr1 < ptr3
```

Pointer points to array

Use pointer to compare memory address: >, <, ==

```c
#include <stdio.h>

int main (void)
{
   int  var[] = {10, 100, 200, 3000};
   int  i, *ptr;

   ptr = var;
   i = 0;
   while (ptr <= &var[3])
   {
      printf("Address of var[%d] = %x\n", i, ptr );
      printf("Value of var[%d] = %d\n\n", i, *ptr );
      ptr++;
      i++;
   }
   return 0;
}
```

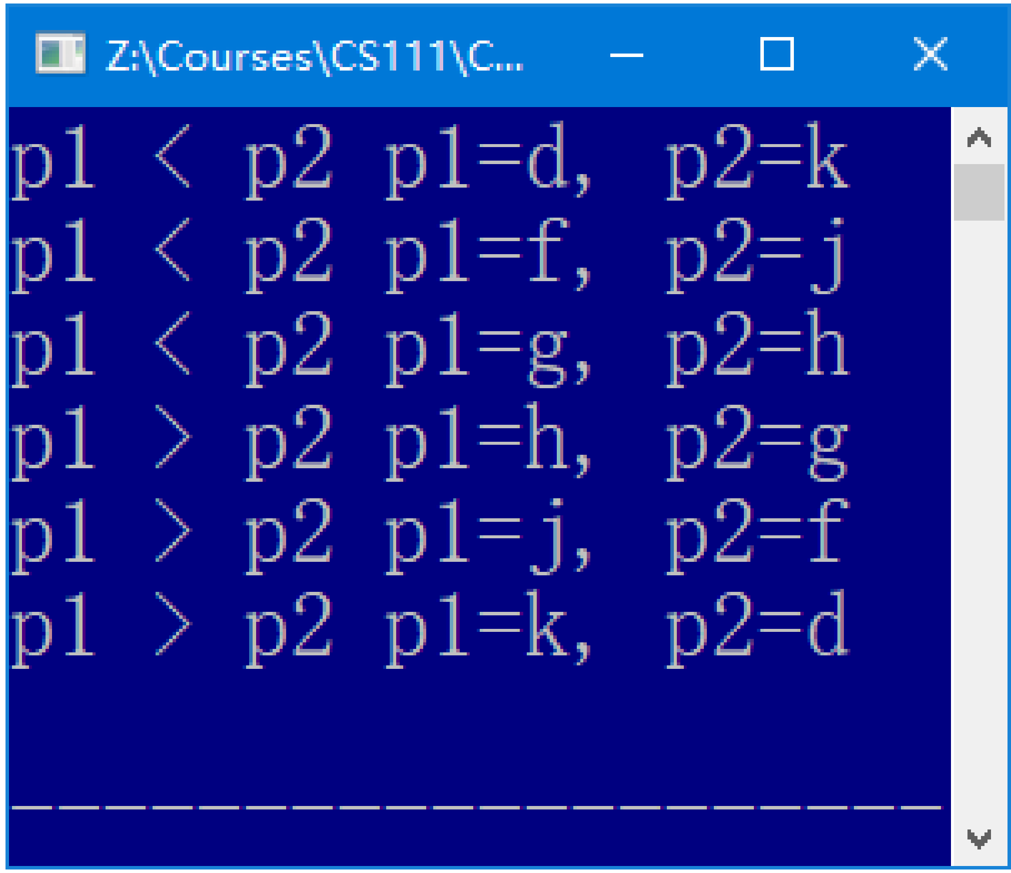


```
Address of var[0] = 60fe88
Value of var[0] = 10

Address of var[1] = 60fe8c
Value of var[1] = 100

Address of var[2] = 60fe90
Value of var[2] = 200

Address of var[3] = 60fe94
Value of var[3] = 3000
```

Pointer points to array

Use pointer to compare memory address: >, <, ==

```c
#include <stdio.h>
main()
{
    char str[7] = "dfghjk", * p1, * p2;
    p1 = str;
    p2 = p1 + 5;

    for (int i = 0; i < 6; i++)
    {
        if (p1 < p2)
        {
            printf("p1 < p2\t");
            printf("p1=%c, p2=%c\n", *p1, *p2);
        }
        else {
            printf("p1 > p2\t");
            printf("p1=%c, p2=%c\n", *p1, *p2);
        }
         *p1++;    //p1++
         *p2--;    //p2--
    }
}
```



```
p1 < p2 p1=d, p2=k
p1 < p2 p1=f, p2=j
p1 < p2 p1=g, p2=h
p1 > p2 p1=h, p2=g
p1 > p2 p1=j, p2=f
p1 > p2 p1=k, p2=d

-------------------------
```

L10_compare.c

Pointer points to array

* and ++/--

数组用法

```
a[i++]=j;
```

指针用法??

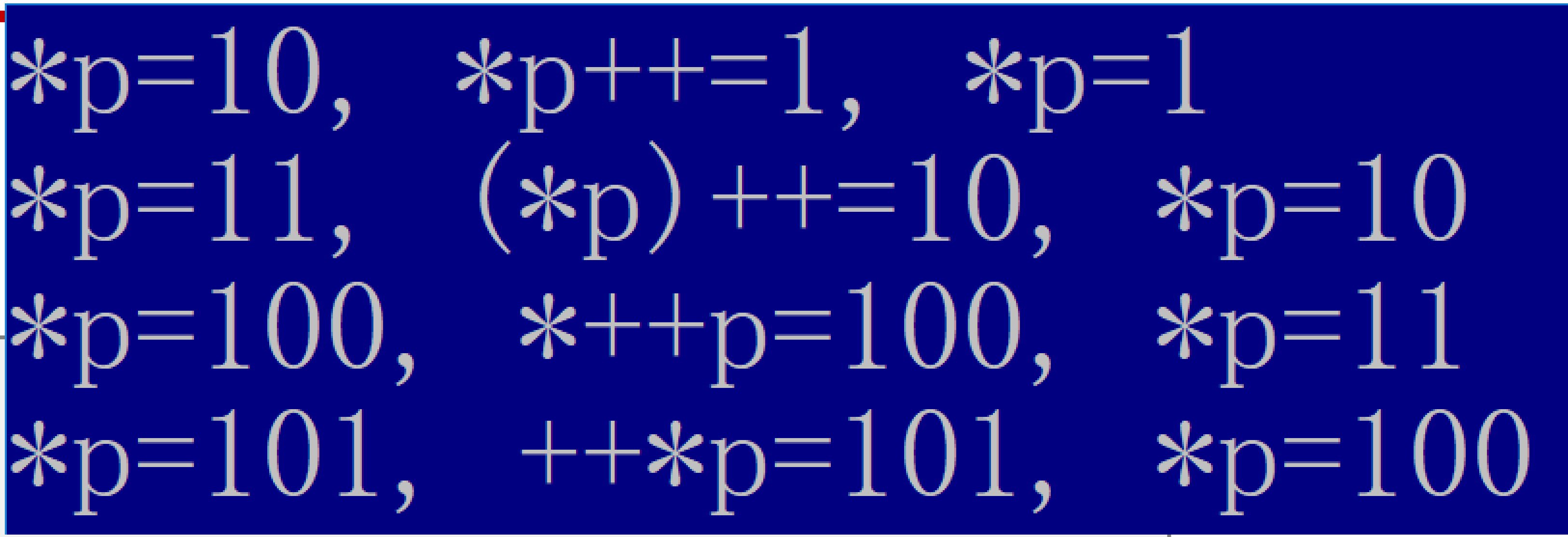
```
p = a;

*p++=j;     ✓

(*p)++=j;   ✗

*(p++)=j;   ✓
```

Pointer points to array

* and ++/--

`*p++ or *p(++)`	先使用p(表达式取指针值)，后移动指针(自增)
`(*p)++`	先使用p(表达式取指针值)，后自增指针指向的值
`*++p or *(++p)`	先移动指针使用(自增)，后使用(取值)
`++*p or ++(*p)`	先取值，后自增指针指向的值

Pointer points to array

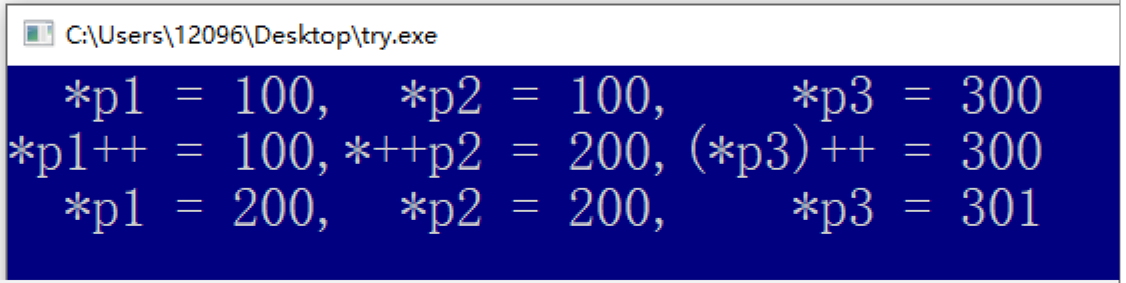


```
*p=10,  *p++=1,  *p=1
*p=11,  (*p)++=10,  *p=10
*p=100,  *++p=100,  *p=11
*p=101,  ++*p=101,  *p=100
```

```c
#include <stdio.h>

int main(void) {
     int a[4] = {1, 10, 100, 1000};
     int *p = a;

     printf("*p=%d, *p++=%d, *p=%d\n",   *p, *p++,   *p);
     printf("*p=%d, (*p)++=%d, *p=%d\n", *p, (*p)++, *p);
     printf("*p=%d, *++p=%d, *p=%d\n",   *p,  *++p,  *p);
     printf("*p=%d, ++*p=%d, *p=%d\n",   *p,  ++*p,  *p);
}
```

Pointer points to array

```c
#include <stdio.h>

int data[2] = {100, 200};
int moredata[2] = {300, 400};

int main(void) {
        int *p1, *p2, *p3;
        p1 = p2 = data;
        p3 = moredata;
        printf("  *p1 = %d,   *p2 = %d,     *p3 = %d\n", *p1, *p2, *p3);
        printf("*p1++ = %d,*++p2 = %d,(*p3)++ = %d\n", *p1++, *++p2, (*p3)++);
        printf("  *p1 = %d,   *p2 = %d,     *p3 = %d\n", *p1, *p2, *p3);
        return 0;
}
```


C:\Users\12096\Desktop\try.exe

```
  *p1 = 100,   *p2 = 100,     *p3 = 300
*p1++ = 100,*++p2 = 200,(*p3)++ = 300
  *p1 = 200,   *p2 = 200,     *p3 = 301
```

Pointer points to array

输出数组元素的三种方法：

下标法

```
void  main() {
        int a[10];
        int i;
        for (i = 0; i < 10; i++)
                scanf("%d", &a[i]);
        printf("\n");
        for (i = 0; i < 10; i++)
                printf("%d", a[i]);
}
```

通过数组名计算数组元素地址，找出元素的值

```
void  main() {
        int a[10];
        int i;
        for (i = 0; i < 10; i++)
                scanf("%d", &a[i]);
        printf("\n");
        for (i = 0; i < 10; i++)
                printf("%d",*(a+i));
}
```

用指针变量指向数组元素

```
void  main() {
        int a[10];
        int *p, i;
        for (i = 0; i < 10; i++)
                scanf("%d", &a[i]);
        printf("\n");
        for (p = a; p < (a + 10); p++)
                printf("%d", *p);
}
```

Pointer points to 2D/ND array

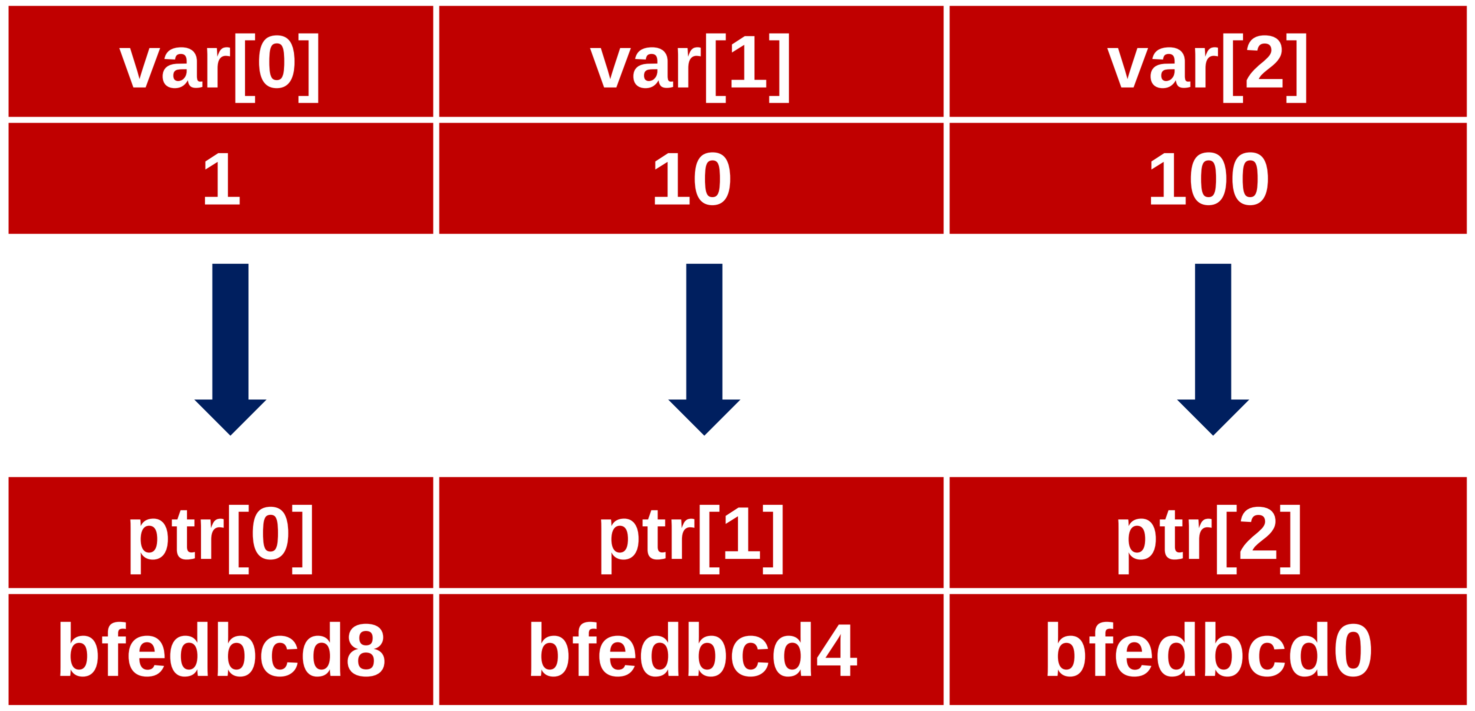
```
#include <stdio.h>

int main(void) {
     int *p;
     int a[4][2]={{2,4},{6,8},{1,3},{5,7}};

     p = a;
     printf("Value of p: %d\n", *p);
}
```

2	4
6	8
1	3
5	7

[错误] 无法转换 'int [3][2]' 到 'int*' 在赋值时

Array of pointers

An array to store pointers



int val[3];

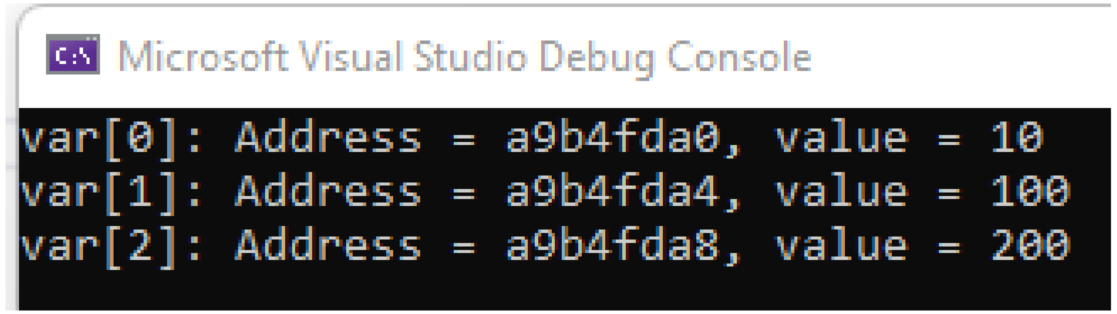
var[0]	var[1]	var[2]
1	10	100

int *ptr[3];

ptr[0]	ptr[1]	ptr[2]
bfedbcd8	bfedbcd4	bfedbcd0

Array of pointers

```c
#include <stdio.h>

int main (void)
{
    int var[] = {10, 100, 200};
    int *ptr[3];
    for (int i = 0; i < 3; i++)
    {
        ptr[i] = &var[i];
        printf("Address of var[%d] = %d\n", i, ptr[i]);
        printf("Value of var[%d] = %d\n", i,  *ptr[i]);
    }
}
```

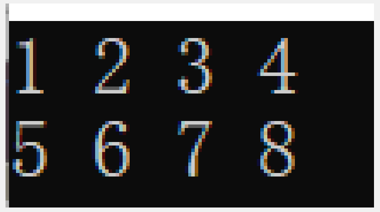

Microsoft Visual Studio Debug Console

```
var[0]: Address = a9b4fda0, value = 10
var[1]: Address = a9b4fda4, value = 100
var[2]: Address = a9b4fda8, value = 200
```

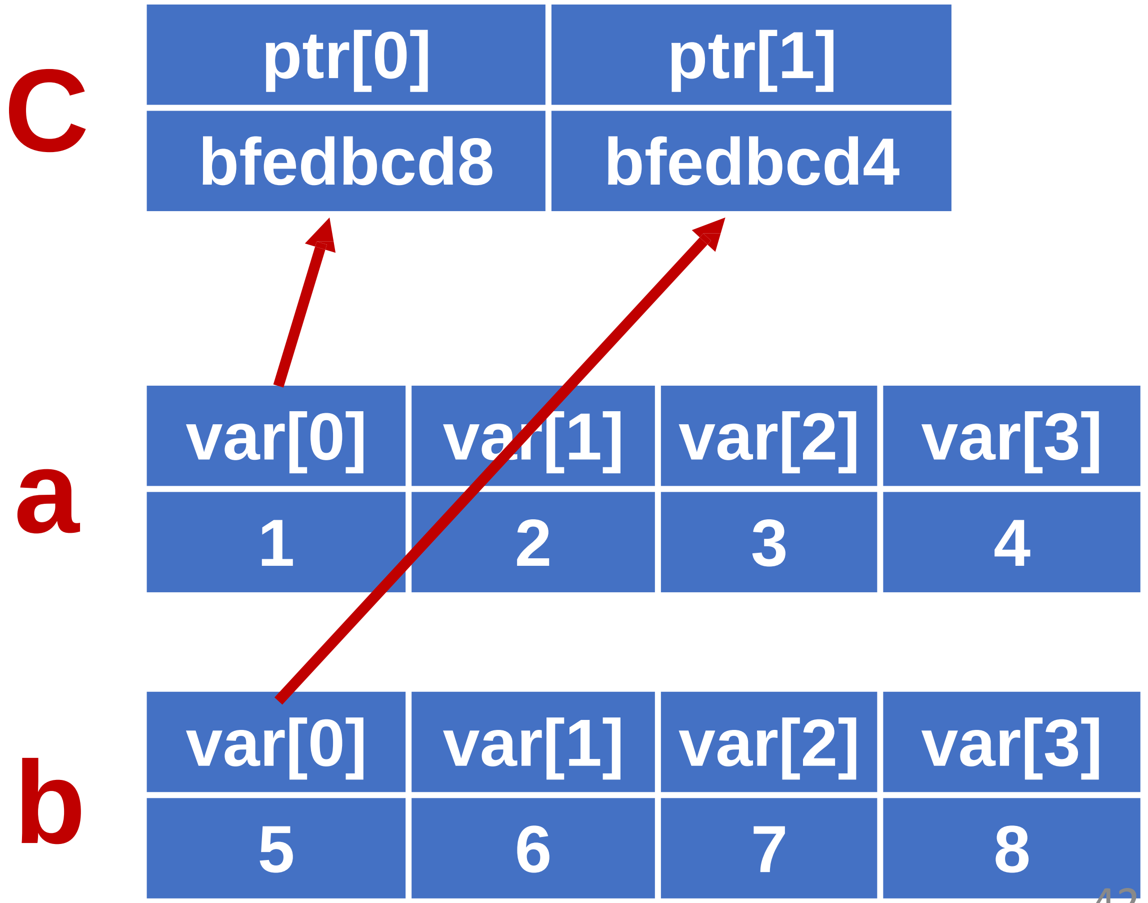
Array of pointers

We can use a 1-D array of pointers to store a 2-D int array

```
#include<stdio.h>

int main(void)
{
  int a[4]={1,2,3,4};
  int b[4]={5,6,7,8};
  int* c[]={a,b};//指针数组
  for(int i = 0; i < 2; i++)
{
    for(int j = 0;j < 4;j++)
    printf("%d ",c[i][j]);
    printf("\n");
}
return 0;
}
```

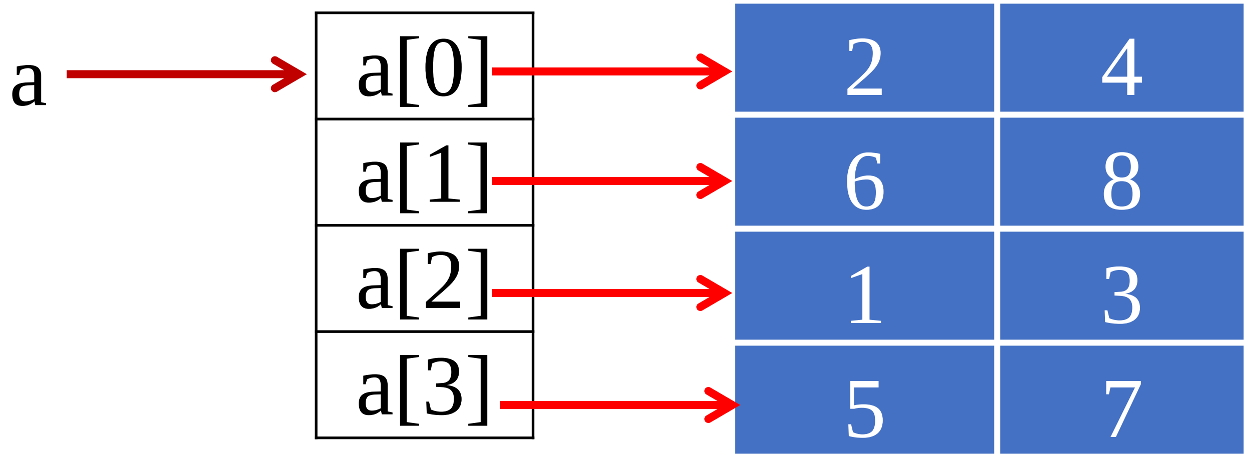


```
1 2 3 4
5 6 7 8
```



Pointer points to 2D array

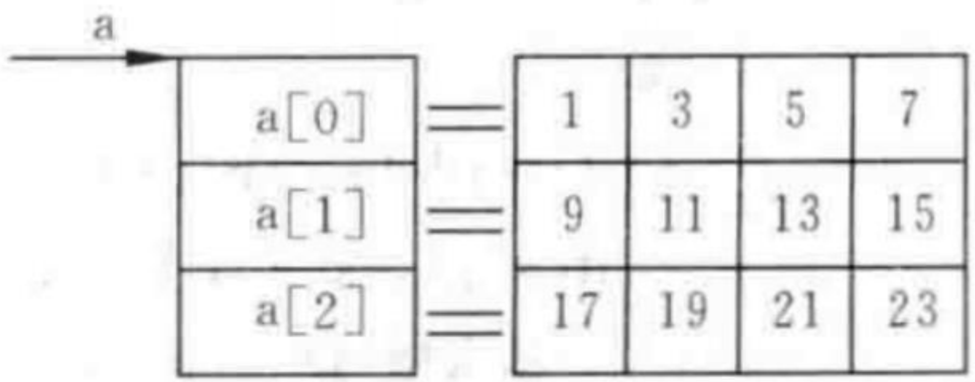
```
int a[4][2]={{2,4},{6,8},{1,3},{5,7}};
```



a[0]、a[1]、a[2]、a[3]分别表示为一维数组

Pointer points to 2D array

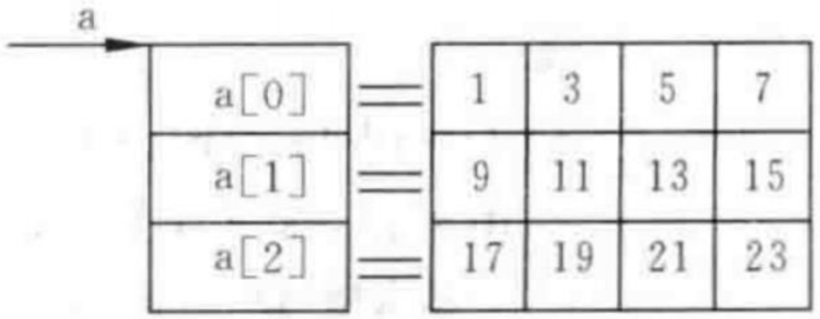
```
int a[3][4]={{1,3,5,7},{9,11,13,15},{17,19,21,23}}
```



a是二维数组名。a数组包含3行，即3个行元素: a[0]，a[1]，a[2]。而每一个行元素又是一个一维数组，它包含4个元素（即4个列元素）。可以认为二维数组是“数组的数组”，即二维数组a有三个一维数组构成

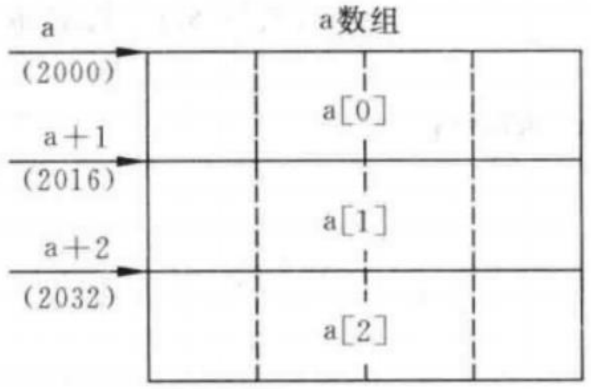
Pointer points to 2D array

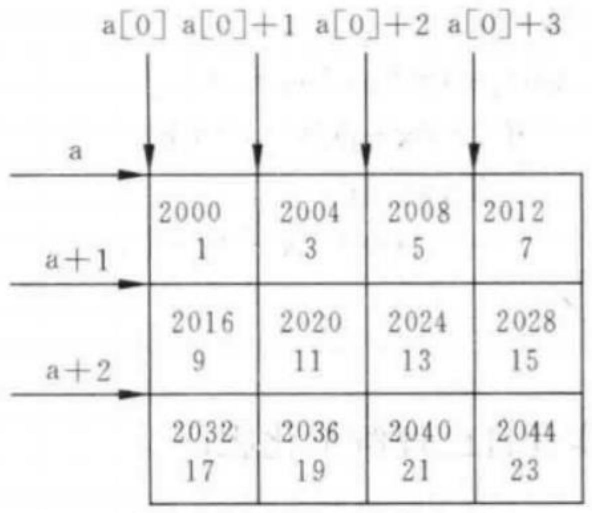
```
int a[3][4]={{1,3,5,7},{9,11,13,15},{17,19,21,23}}
```



从二维数组的角度来看，a代表二维数组首元素的地址，**现在的首元素不是一个简单的整型元素，是由4个整型元素所组成的一维数组**，因此a代表的是首行（即序号为0的行）的起始地址。a＋1代表序号为1的行的起始地址。

如果二维数组的首行的起始地址为2000，一个整型数据占4个字节，则a＋1的值应该是2000＋4×4=2016（因为第0行有4个整型数据）。a＋1指向a[1]，或者说，a+1的值是a[1]的起始地址。a＋2代表a[2]的起始地址，它的值是2032。





Pointer points to 2D array

```
int a[3][4]={{1,3,5,7},{9,11,13,15},{17,19,21,23}}
```

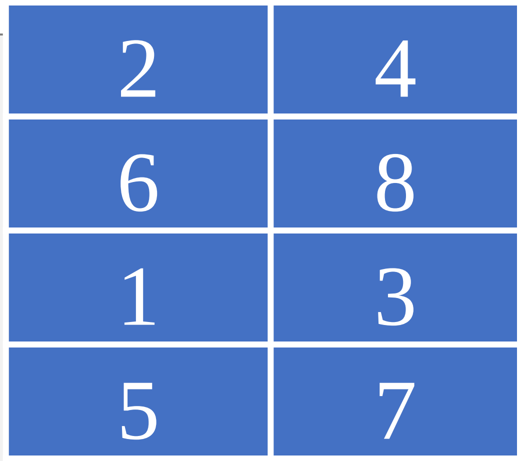
表示形式	含义	值
a	二维数组名，指向一维数组a[0]，即0行起始地址	2000
a[0], *(a+0), *a	0行0列元素地址	2000
a+1, &a[1]	1行起始地址	2016
a[1], *(a+1)	1行0列元素a[1][0]的地址	2016
a[1]+2, *(a+1)+2, &a[1][2]	1行2列元素a[1][2]的地址	2024
*(a[1]+2), *(*(a+1)+2), a[1][2]	1行2列元素a[1][2]的值	元素值，13

注意：

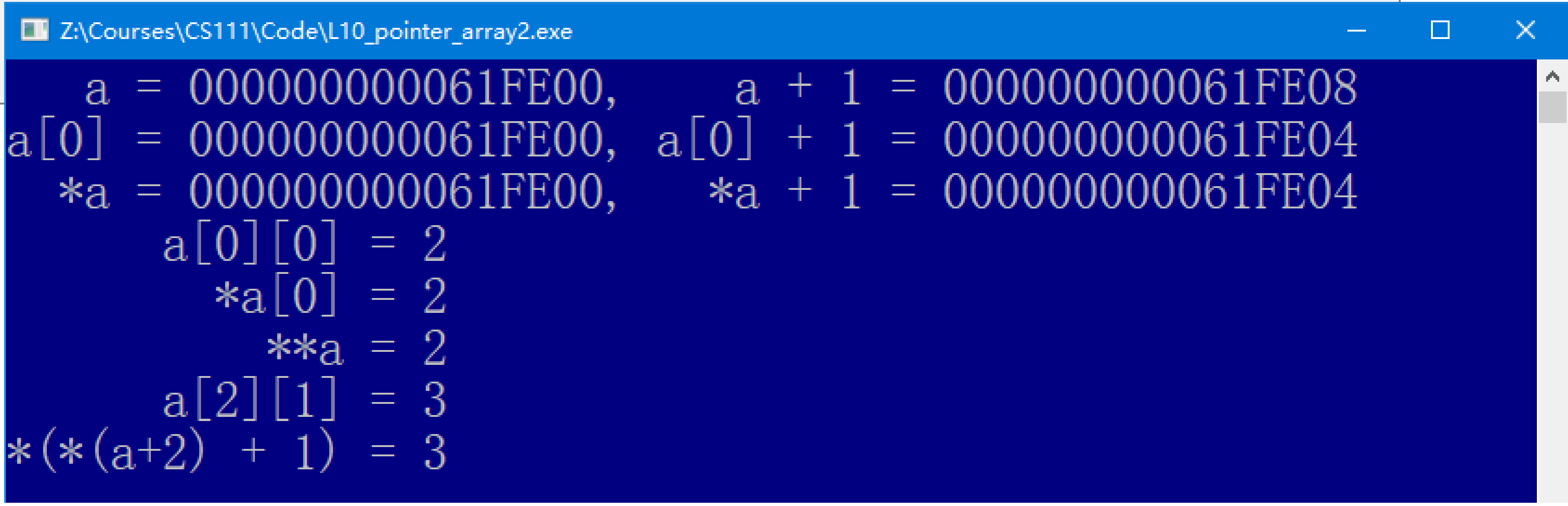
- **二维数组名(如a)是指向行(一维数组)的。一维数组名（如a[0]）是指向列元素的。**在指向行的指针前面加一个*，就转换为指向列的指针。反之，在指向列的指针前面加&，就成为指向行的指针。
- 不要把&a[i]简单地理解为a[i]元素的存储单元的地址，它只是一种**地址**的计算方法，能得到第i行的起始地址。
- &a[i]和a[i]的值是一样的，但它们的基类型是不同的。&a[i]或a+i指向行，而 a[i]或*(a+i)指向列。

Pointer points to 2D array

```
int main(void) {
        int a[4][2] = { { 2, 4 }, { 6, 8 }, { 1, 3 },{ 5, 7 }};
        printf("   a = %p,     a + 1 = %p\n", a, a + 1);
        printf("a[0] = %p, a[0] + 1 = %p\n", a[0], a[0] + 1);
        printf("  *a = %p,    *a + 1 = %p\n", *a, *a + 1);
        printf("      a[0][0] = %d\n", a[0][0]);
        printf("        *a[0] = %d\n", *a[0]);
        printf("          **a = %d\n", **a);
        printf("      a[2][1] = %d\n", a[2][1]);
        printf("*(*(a+2) + 1) = %d\n", *(*(a + 2) + 1));
        return 0;
}
```



2	4
6	8
1	3
5	7


Z:\Courses\CS111\Code\L10_pointer_array2.exe

```
   a = 000000000061FE00,     a + 1 = 000000000061FE08
a[0] = 000000000061FE00, a[0] + 1 = 000000000061FE04
  *a = 000000000061FE00,    *a + 1 = 000000000061FE04
      a[0][0] = 2
        *a[0] = 2
          **a = 2
      a[2][1] = 3
*(*(a+2) + 1) = 3
```

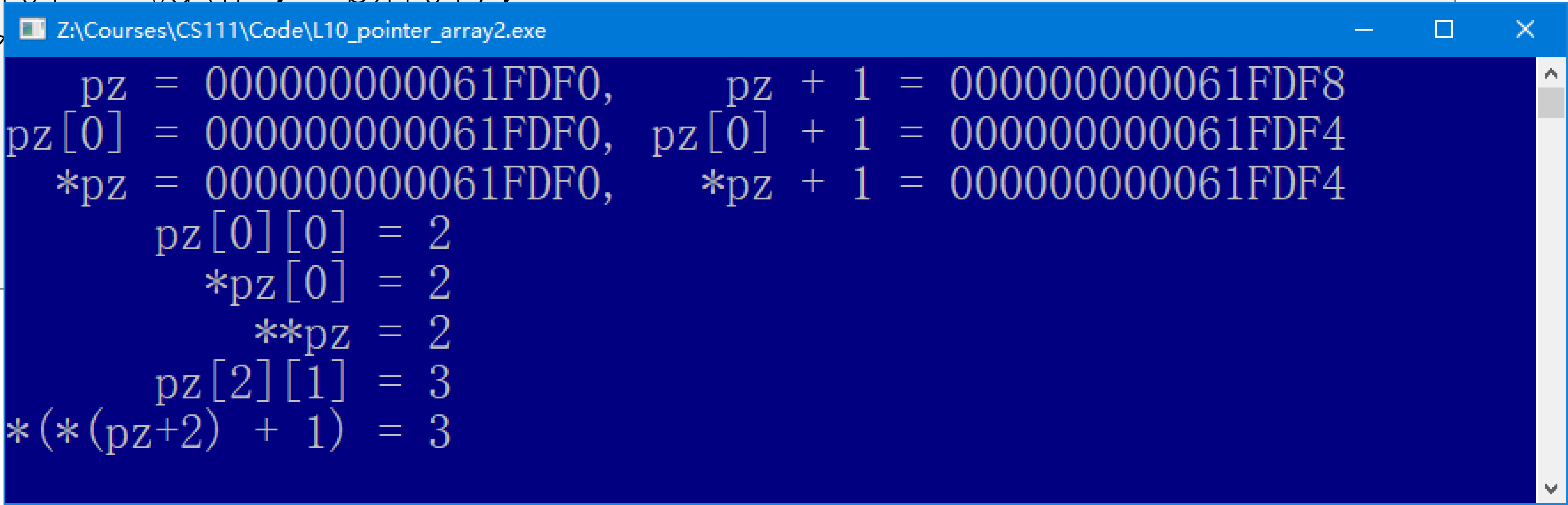
Pointer points to 2D array

声明指向数组的指针：（数组指针）

```
type  (*name)[2]
```

2	4
6	8
1	3
5	7

```
int main(void)  {
        int a[4][2] = { { 2, 4 }, { 6, 8 }, { 1, 3 }, { 5, 7 } };
        int(*pz)[2];
        pz = a;
        printf("    pz = %p,     pz + 1 = %p\n", pz, pz + 1);
        printf("pz[0] = %p, pz[0] + 1 = %p\n", pz[0], pz[0] + 1);
        printf("   *pz = %p,    *pz + 1 = %p\n", *pz, *pz + 1);
        printf("       pz[0][0] = %d\n", pz[0][0]);
        printf("         *pz[0] = %d\n", *pz[0]);
        printf("           **
        printf("       pz[2]
        printf("*(*(pz+2) +
        return 0;
}
```



Z:\Courses\CS111\Code\L10_pointer_array2.exe

```
    pz = 000000000061FDF0,     pz + 1 = 000000000061FDF8
pz[0] = 000000000061FDF0, pz[0] + 1 = 000000000061FDF4
   *pz = 000000000061FDF0,    *pz + 1 = 000000000061FDF4
        pz[0][0] = 2
          *pz[0] = 2
            **pz = 2
        pz[2][1] = 3
*(*(pz+2) + 1) = 3
```

Pointer points to 2D array

假设有如下声明：

```
int * pt;
int (*pa)[3];
int ar1[2][3];
int ar2[3][2];
int **p2; // 一个指向指针的指针
```

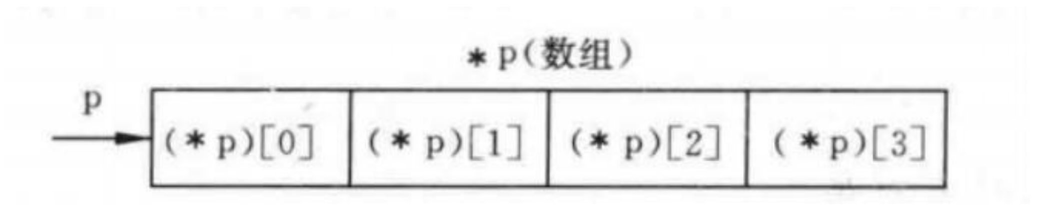
有如下的语句：

```
pt = &ar1[0][0]; // 都是指向int的指针
pt = ar1[0];     // 都是指向int的指针
pt = ar1;        // 无效
pa = ar1;        // 都是指向内含3个int类型元素数组的指针
pa = ar2;        // 无效
p2 = &pt;        // 都是指向int *的指针
*p2 = ar2[0];    // 都是指向int的指针
p2 = ar2;        // 无效
```

?

Pointer points to 2D array

- 指向由m个元素组成的一维数组的指针变量



```
int zippo[3][4]
```

假设指针变量p指向二维数组zippo，声明：

```
int (* p)[4] //pz指向一个内含两个int类型值的一维数组
p = zippo
```

数组名zippo是数组首元素的地址，**该元素是一个内含四个int类型值的一维数组**。因此，p必须指向一个内含两个int类型值的数组，而不是指向一个int类型值。

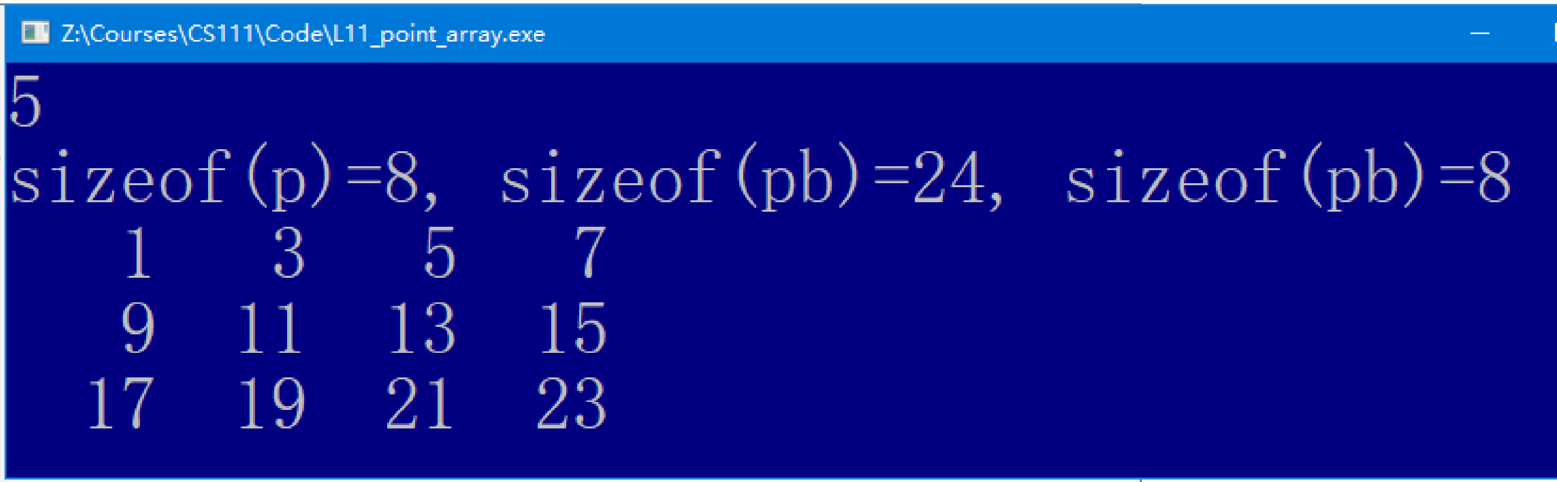
```
int * pax[2]   // pax是一个内含两个指针元素的数组，每个元素都指向int的指针
```

Pointer points to 2D array

```
#include <stdio.h>
int main(void) {
      int a[3][4] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23};
      int *p, *pa[3], (*pb)[4];
      pb = a;
      pa[0] = a[0];
      pa[1] = a[1];
      pa[2] = a[2];
      printf("%d \n", *(pa[0] + 2 ));
      printf("sizeof(p)=%d, sizeof(pb)=%d, sizeof(pb)=%d", \
             sizeof(p), sizeof(pa), sizeof(pb));
      for (p = a[0]; p < a[0] + 12; p++) {
            if ((p - a[0]) % 4 == 0)
                  printf("\n");
            printf("%4d", *p);
      }
      printf("\n");
      return 0;}
```

Pointer points to 2D array

```
#include <stdio.h>
int main(void) {
        int a[3][4] = {1, 3, 5, 7
        int *p, *pa[3], (*pb)[4];
        pb = a;
        pa[0] = a[0];
        pa[1] = a[1];
        pa[2] = a[2];
        printf("%d \n", *(pa[0] + 2 ));
        printf("sizeof(p)=%d, sizeof(pb)=%d, sizeof(pb)=%d", \
                sizeof(p), sizeof(pa), sizeof(pb));
        for (p = a[0]; p < a[0] + 12; p++) {
                if ((p - a[0]) % 4 == 0)
                        printf("\n");
                printf("%4d", *p);
        }
        printf("\n");
        return 0;}
```



Z:\Courses\CS111\Code\L11_point_array.exe

```
5
sizeof(p)=8, sizeof(pb)=24, sizeof(pb)=8
   1   3   5   7
   9  11  13  15
  17  19  21  23
```

Content

1. Memory address
2. Pointer
3. Pointer and Array
4. Pointer and function
5. Memory management(advanced uses)

function、array and pointer

```
#include <stdio.h>
#define SIZE 10
int sum(int ar[], int n);

int main(void) {
      ...
      int marbles[SIZE] = {20, 10, 5, \
   39, 4, 16, 19, 26, 31, 20};
      answer = sum(marbles, SIZE);
      ...
}

int sum(int ar[10], int n) {
      ...
}
```

- 实参数组与形参数组类型应一致（例子中都为int型）。
- 在定义sum函数时，声明形参数组大小为10，但实际上，指定其大小不起任何作用因为C语言编译系统并不检查形参数组大小，**只是将实参数组的首元素的地址传给形参数组名。**形参数组名获得了实参数组的首元素地址，可以认为，形参数组首元素（ar[0]）和实参数组首元素（ marbles[0]）具有同一地址，它们共占同一存储单元， marbles[n]和 ar[n]指的是同一单元，marbles[n]和ar[n]具有相同的值。
- 形参数组可以不指定大小，在定义数组名后面跟一个空的方括号。

function、array and pointer

声明数组形参：

数组名是该数组首元素的地址，作为实际参数的数组名要求形式参数是一个与之匹配的指针。

下面两种形式的函数定义等价：

```
int sum(int *ar,int n)
{ ...
}

int sum(int ar[],int n)
{ ...
}
```

function、array and pointer

假设flizny是一个数组：

flizny == &flizny[0]

数组名即为元素的首地址。

```
int sum(int *ar) {
    int i;
    int total = 0;
    for (i = 0; i < 10; i++) {
        total += ar[i];
    }
    return total;
}
```

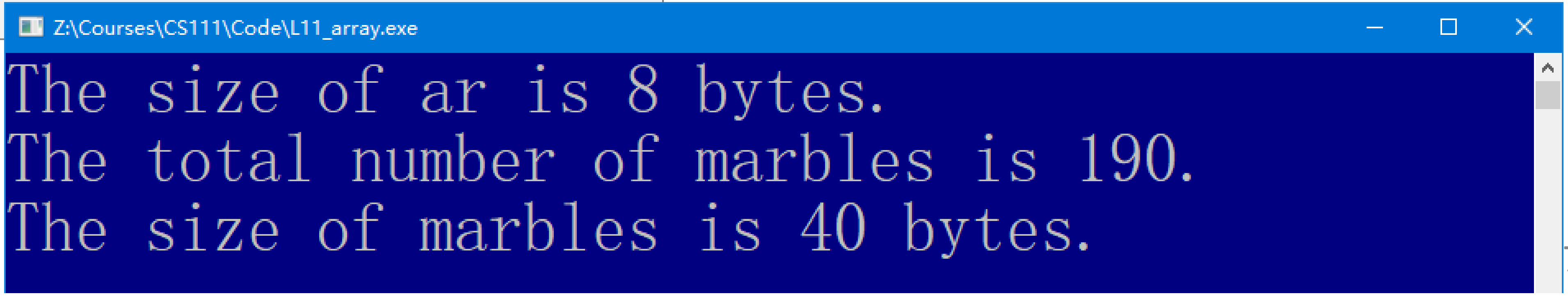
int *ar 和int ar[] 的形式都表示ar是一个指向int 的指针!!!

function、array and pointer

数组在函数内部当作指针

```
#include <stdio.h>
#define SIZE 10
int sum(int ar[], int n);

int main(void) {
   int marbles[SIZE] = {20, 10, 5, 39, 4, 16, 19, 26, 31, 20};
   long answer;
   answer = sum(marbles, SIZE);
   printf("The total number of marbles is %d.\n", answer);
   printf("The size of marbles is %d bytes.\n", sizeof (marbles));
   return 0;
}
```

```
int sum(int ar[], int n) {
   int i;
   int total = 0;

   for (i = 0; i < n; i++) {
      total += ar[i];
   }
   printf("The size of ar is %d bytes.\n", sizeof(ar));
   return total;
}
```


Z:\Courses\CS111\Code\L11_array.exe

```
The size of ar is 8 bytes.
The total number of marbles is 190.
The size of marbles is 40 bytes.
```

function、array and pointer

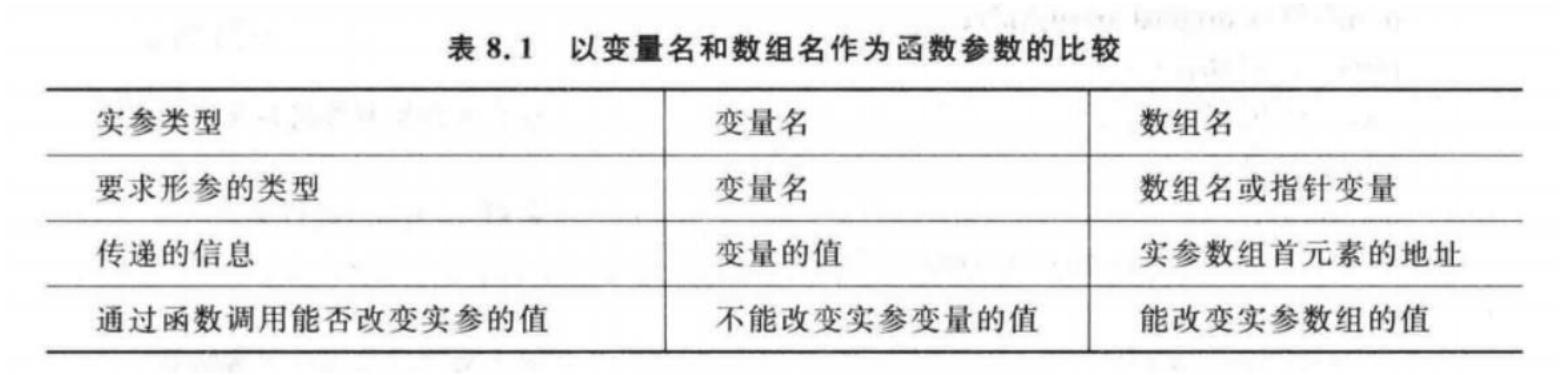


表 8.1　以变量名和数组名作为函数参数的比较

实参类型	变量名	数组名
要求形参的类型	变量名	数组名或指针变量
传递的信息	变量的值	实参数组首元素的地址
通过函数调用能否改变实参的值	不能改变实参变量的值	能改变实参数组的值

注意: 实参数组名代表一个固定的地址，或者说是指针常量，但形参数组名并不是一个固定的地址，而是按指针变量处理。

```
void fun(arr[],int n){
        printf("%d\n",*arr);
        arr = arr+3;            //形参数组名在函数执行期间可以再被赋值
        printf("%d\n",*arr);
}
```

function、array and pointer

如果有一个实参数组，要想在函数中改变此数组中的元素的值，实参与形参的对应关系：

形参和实参都用数组名

```
int main() {
	int a[10];
	...
	f(a, 10);
	...
}

int f(int x[], int n) {
	...
}
```

实参用数组名，形参用指针变量

```
int main() {
	int a[10];
	...
	f(a, 10);
	...
}

int f(int *x, int n) {
	...
}
```

形参和实参都用指针变量

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

int f(int *x, int n) {
	...
}
```

实参为指针变量，形参为数组名

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

int f(int x[], int n) {
	...
}
```

function、array and pointer

用指针作为实参并表明数组的开始与结束：

```
#include <stdio.h>
#define SIZE 10
int sump(int *start, int *end);
int main(void) {
        int marbles[SIZE] = { 20, 10, 5, 39, 4, 16, 19, 26, 31, 20 };
        long answer;
        answer = sump(marbles, marbles + SIZE);
        printf("The total number of marbles is %ld.\n", answer);
        return 0;
}
/* 使用指针算法 */
int sump(int *start, int *end) {
        int total = 0;
        while (start < end) {
                total += *start; // 把数组元素的值加起来
                start++;} // 让指针指向下一个元素
        return total;
}
```

total += *start++

因为while循环的测试条件是一个不相等的关系，所以**end指向的位置实际上在数组最后一个元素的后面**。C保证在给数组分配空间时，**指向数组后面第一个位置的指针仍是有效的指针。**

注意：用指针作实参必须先使指针变量有确定值，指向一个已定义的对象

function、array and pointer

多维数组名作为函数参数

```
int sum2d(int ar[][4], int rows)   ✔
int sum2d(int ar[][], int rows)
```

编译器会把数组表示法转换成指针表示法。例如，编译器会把 ar[1]转换成 ar+1。编译器对ar+1求值，要知道ar所指向的对象大小。上述**第一种表示ar指向一个内含4个int类型值的数组**，所以ar+1的意思是“该地址加上16字节”。如果第2对方括号是空的，编译器就不知道该怎样处理。

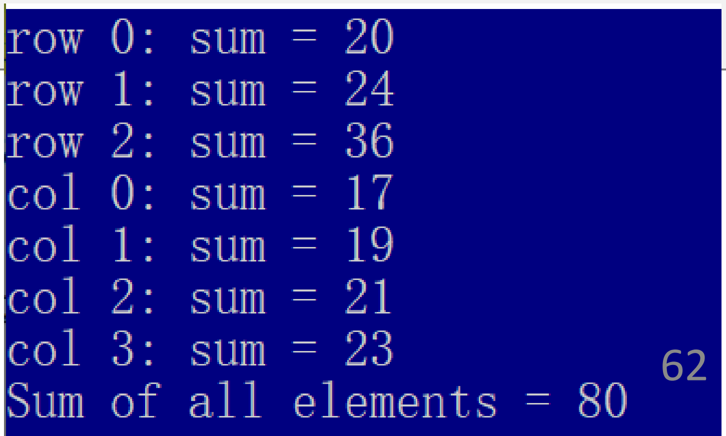
```
int sum2d(int ar[3][4], int rows)   ✔
```

有效声明，但是编译器会忽略3。

function、array and pointer

```
#define ROWS 3
#define COLS 4
void sum_rows(int ar[][COLS], int rows);
void sum_cols(int [][COLS], int); // 省略形参名，没问题
int sum2d(int(*ar)[COLS], int rows); // 另一种语法
int main(void) {
    int junk[ROWS][COLS] = {{ 2, 4, 6, 8 },{ 3, 5, 7, 9 },{ 12, 10, 8, 6 }};
    sum_rows(junk, ROWS);
    sum_cols(junk, ROWS);
    printf("Sum of all elements = %d\n", sum2d(junk, ROWS));
    return 0;
}
void sum_rows(int ar[][COLS], int rows) {
    int r, c, tot;
    for (r = 0; r < rows; r++) {
        tot = 0;
        for (c = 0; c < COLS; c++)
            tot += ar[r][c];
        printf("row %d: sum = %d\n", r, tot);}}
```

```
void sum_cols(int ar[][COLS], int rows) {
    int r, c, tot=0;
    for (c = 0; c < COLS; c++) {
        tot = 0;
        for (r = 0; r < rows; r++)
            tot += ar[r][c];
        printf("col %d: sum = %d\n", c, tot);}}
int sum2d(int ar[][COLS], int rows) {
        int r, c, tot;
        for (r = 0; r < rows; r++)
            for (c = 0; c < COLS; c++)
                tot += ar[r][c];
        return tot;
}
```

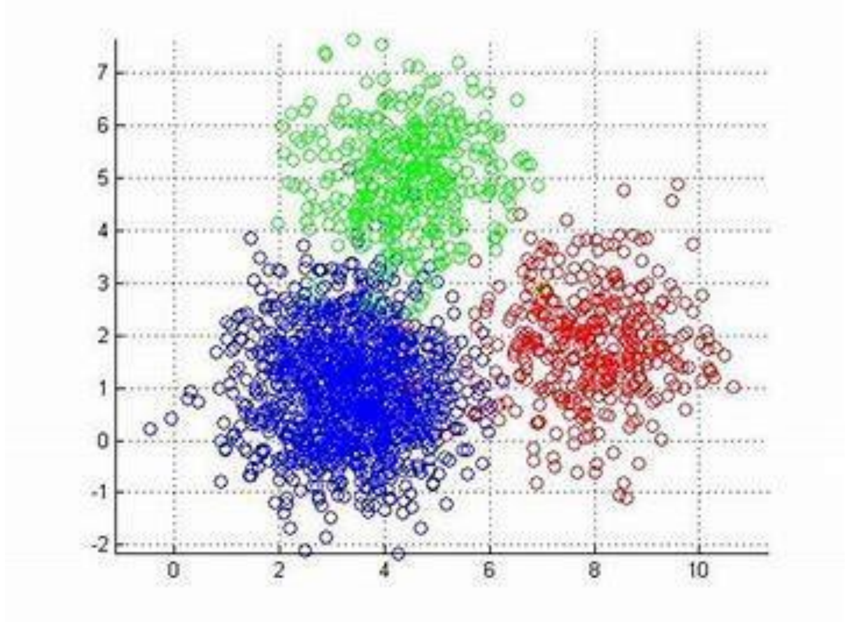


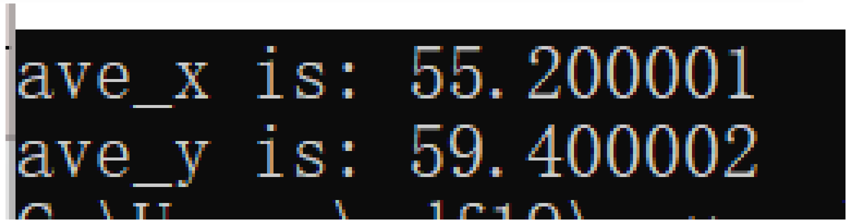
```
row 0: sum = 20
row 1: sum = 24
row 2: sum = 36
col 0: sum = 17
col 1: sum = 19
col 2: sum = 21
col 3: sum = 23
Sum of all elements = 80
```

Case study: calculate coordinate mean

```c
void cal_ave_coor(int points[][2],int num, float* ave_x,
float* ave_y)
{
    int sum_x = 0;
    int sum_y = 0;
    for (int i = 0; i < num; i++)
    {
        sum_x += points[i][0];
        sum_y += points[i][1];
    }
    *ave_x = (float)sum_x / num;
    *ave_y = (float)sum_y / num;
}
int main(void){
    float ave_x, ave_y;
    int points[5][2] =
    { 23,43,65,67,78,53,74,85,36,49 };
    cal_ave_coor(points, 5, &ave_x, &ave_y);
    printf("ave_x is: %f\nave_y is: %f",ave_x,ave_y);
}
```

Case: calculate mean of 2D points





```
ave_x is: 55.200001
ave_y is: 59.400002
```

Function can return pointer

一个函数可以带回一个整型值、字符值、实型值等，也可以带回指针型的数据，即地址。其概念与以前类似，只是带回的值的类型是指针类型而已。

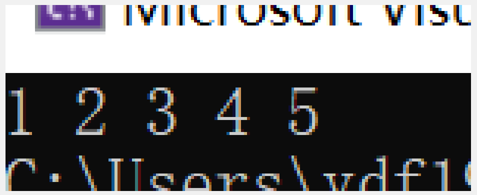
一般定义形式为：

类型名 *函数名（参数列表）

Function can return pointer

```
int * myFunction()
{

 ...

}
```

```c
int* merge(int a, int b, int c, int d, int e)
{
int* array = (int*)malloc(sizeof(int) * 5);
array[0] = a;
array[1] = b;
array[2] = c;
array[3] = d;
array[4] = e;
return array;
}

int main()
{
int* array = merge(1, 2, 3, 4, 5);
for (int i = 0; i < 5; i++)
printf("%d ", array[i]);
return 0;
}
```

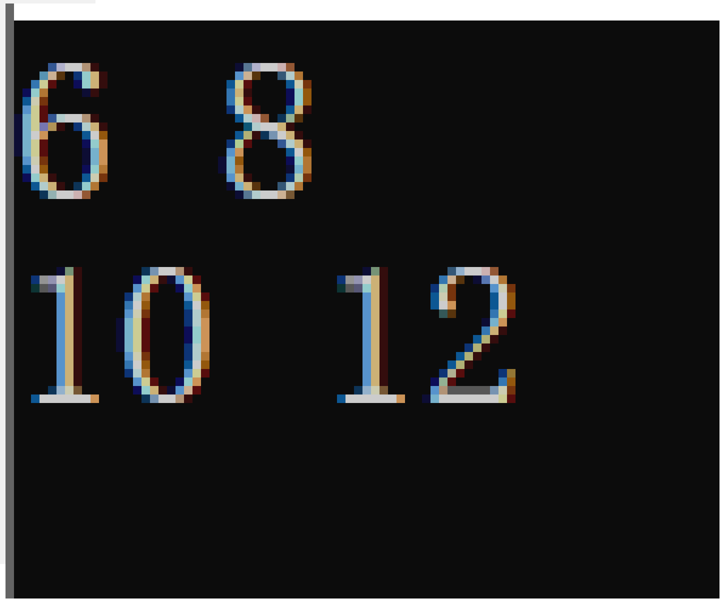


```
1 2 3 4 5
C:\Users\ydf1
```

Case study: Matrix addition

Case: add 2 matrices

```
int *add_mats(int *A, int *B, int rows, int cols) {
        int *C = (int *)malloc(sizeof(int) * rows * cols);
         for (int i = 0; i < rows; i++)
         for (int j = 0; j < cols; j++)
            C[i * cols + j] = B[i * cols + j] +  A[i * cols + j];
        return C;
}
int main(void){
         int A[4] = {1,2,3,4};
         int B[4] = {5,6,7,8};
         int *C = add_mats(A, B, 2, 2);
         for (int i = 0; i < 2; i++){
                 for (int j = 0; j < 2; j++){
                   printf("%d ",C[i * 2 + j]);
                 }
                 printf("\n");
         }
}
```



```
6 8
10 12
```

Pointer of a function

指向函数的指针

- 用指针变量可以指向一个函数。
- 函数在编译时被分配给一个入口地址，函数名代表函数的起始地址。这个函数的入口地址就称为**函数的指针**。

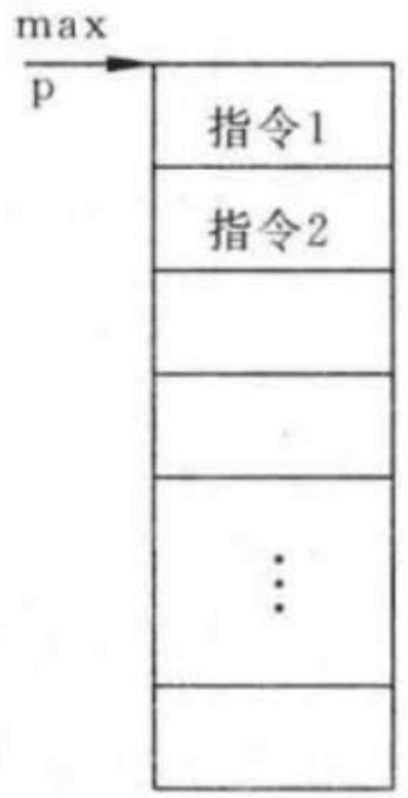
```
int (*p)(int, int);
```

指向函数类型为整形，且有两个整型参数的函数

Pointer of a function

利用指针变量调用它所指向的函数

```c
int main(void) {
        int max(int, int);
        int(*p)(int, int);
        int a, b, c;
        p = max;    //只给出函数名不需要参数
        printf("please enter a and b:");
        scanf("%d %d", &a, &b);
        c = (*p)(a, b);
        printf("a=%d\nb=%d\nmax=%d\n", a, b, c);
        return 0;
}
int max(int x, int y) {
        int z;
        if (x > y)  z = x;
        else   z = y;
        return z;
}
```



指向函数的指针变量**不能**进行算数运算（p+n）！！

Pointer of a function

用指向函数的指针做函数参数

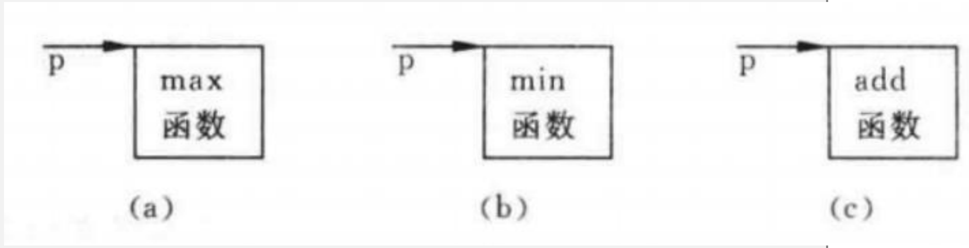
- 指向函数的指针变量的一个重要用途是把函数的入口地址作为参数传递到其他函数。

```
实参函数名      f1                 f2
                ↙                  ↓
void fun(int(*x1)(int), int(*x2)(int, int))//形参是指向函数的指针变量
{
      int a, b, i = 3, j = 5;
      a = (*x1)(i);                             //调用f1函数，i是实参
      b = (*x2)(i, j);                          //调用f2函数，i，j是实参
}
```

Pointer of a function

```
#include <stdio.h>

void main() {
    int max(int, int);
    int min(int, int);
    int add(int, int);
    void process (int, int, int(*fun)());
    int a, b;
    printf("enter a and b:");
    scanf("%d %d", &a, &b);
    printf("max=");
    process(a, b, max);
    printf("min=");
    process(a, b, min);
    printf("sum=");
    process(a, b, add);
}
```

```
int max(int x, int y) {
    return x > y ? x : y;
}
int min(int x, int y) {
    return x < y ? x : y;
}
int add(int x, int y) {
    int z;
    z = x + y;
    return z;
}

void process(int x, int y, int(*fun)(int, int))
{
    int result;
    result = (*fun)(x, y);
    printf("%d\n", result);
}
```



Pointer array

一个数组，若其元素均为指针类型数据，称为**指针数组**，也就是说，指针数组中的每一个元素都相当于一个指针变量。

一维指针数组的定义形式：

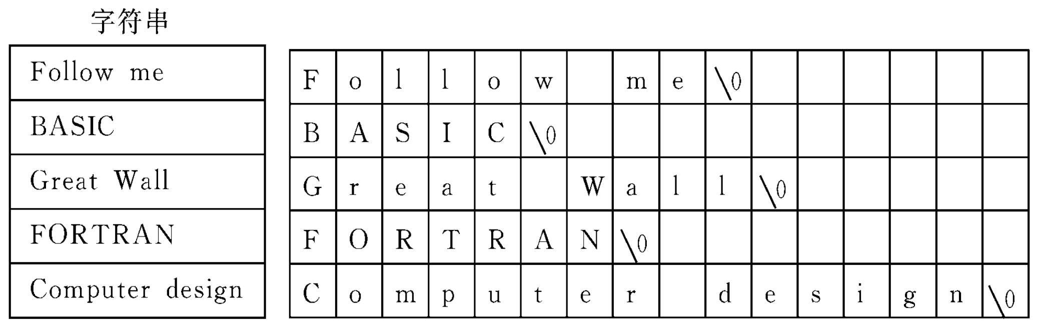
类型名 *数组名[数组长度]

适用于用来指向若干个字符串，使字符串处理更加方便灵活

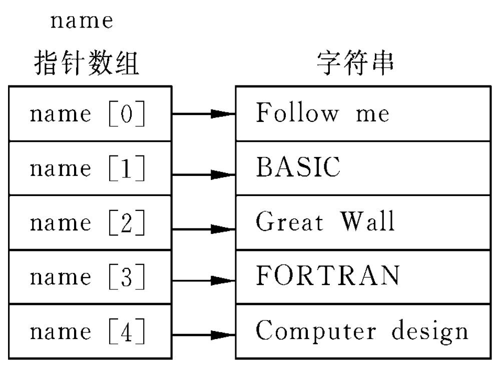
Pointer array

```
int  main(void) {
        void sort(char *name[], int n);
        void print(char *name[], int n);
        char *name[] = {"Follow me", "BASIC", "Great Wall", \
        "FORTRAN", "Computer design"};
        int n = 5;
        sort(name, n);
        print(name, n);}
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;}
                }   }
```

```
void print(char *name[], int n) {
                int i;
                for (i = 0; i < n; i++)
                                printf("%s\n", name[i]);
}
```



(a) (b)



(c)

Pointer array

指针数组作main函数的形参

指针数组的一个重要应用是作为main函数的形参。在以往的程序中，main函数的第一行一般写成以下形式：int **main**（），或者int **main**（void）,括号中没有参数。

main函数也可以有参数：

```
int main(int argc, char *argv[])
```

其中，argc和 argv就是main函数的形参，它们是程序的“命令行参数”。argc(argumentcount的缩写,意思是**参数个数**)，argv(argument vector缩写，意思是参数向量)，它是一个* char指针数组,数组中每一个元素(其值为指针)指向**命令行中的一个字符串的首字符**。

Pointer array

命令行一般形式：

命令名 参数1 参数2 ...参数n

例如一个名为file 1 的文件，它包含以下的main函数:

```
void  main(int argc， char *argv[]) {
            while (argc > 1) {
                        ++argv;
                        printf("%s\n", argv);
                        --argc;
            }
}
```

在DOS命令状态下输入的命令行为
file1 China Beijing
则执行以上命令行将会输出以下信息：

```
China
Beijing
```

Character pointer

用字符数组存放一个字符串，
然后输出该字符串

```
int main(void) {
        char string[] = "I love China!";
        printf("%s\n", string);
        return 0;
}
```

用字符指针存放一个字符串，
然后输出该字符串

```
int main(void) {
        char *string = "I love China!";
        printf("%s\n", string);
        return 0;
}
```

Character pointer

```
int main(void) {
        char *string = "I love China!";
        printf("%s\n", string);
        return 0;
}
```

对字符指针变量string初始化,实际上是**把字符串第1个元素的地址(即存放字符串的字符数组的首元素地址)赋给指针变量string**,使 string指向字符串的第1个字符,由于字符串常量"I love China!"已由系统分配在内存中连续的14个字节中,因此, string 就指向了该字符串的第一个字符。

```
char *string;
*srting = "I love China!"
```

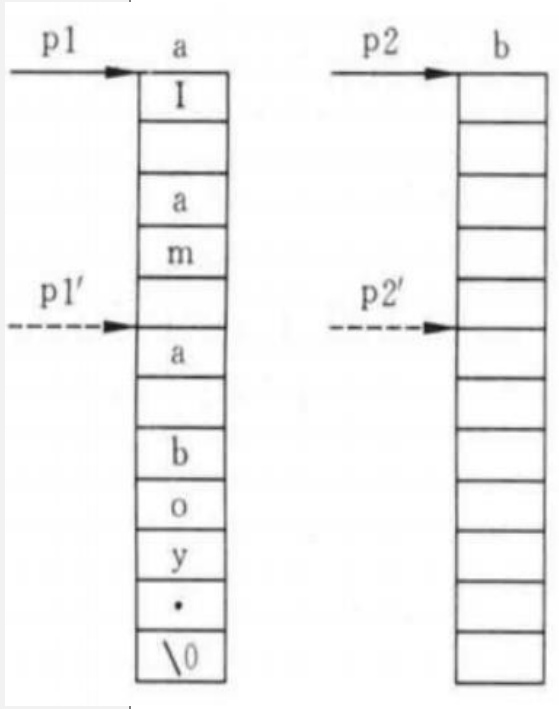
```
char *string;
srting = "I love China!"
```

Character pointer

将字符串a复制为字符串b,然后输出字符串b。

```
#include<stdio.h>
int main(void) {
        char a[] = "I am a student.", b[20];
        int i;
        for (i = 0; * (a + i) != '\0'; i++)
                *(b + i) = *(a + i);
        *(b + i) = '\0';
        printf("string a is: %s\n", a);
        printf("srting b is:");
        for (i = 0; b[i] != '\0'; i++)
                printf("%c", b[i]);
        printf("\n");
        return 0;
}
```

```
#include<stdio.h>
int main(void) {
        char a[] = "I am a student.", b[20];
        char *p1, *p2;
        p1 = a;
        p2 = b;
        for (; *p1 != '\0'; p1++, p2++)
                *p2 = *p1;
        *p2 = '\0';
        printf("string a is: %s\n", a);
        printf("string b is: %s", b );
        return 0;
}
```



Character pointer

字符指针变量 VS 字符数组

1.字符数组由若干个元素组成，每个元素中放一个字符，而字符指针变量存放的是地址（字符串第1个字符的地址），决不是将字符串放到字符指针变量中。

2.赋值方式。**可以对字符指针变量赋值，但不能对数组名赋值。**

```
char *a;
a = "I love China!";//合法赋给a的不是字符串而是第一个元素的地址

char  str[14];
str＝"I love China"; //非法！数组名是地址不是常量，不能被赋值
```

3.初始化含义。

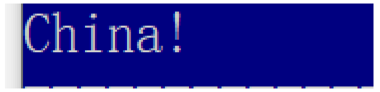
```
char *a;
a = "I love China!";//把字符串第一个元素的地址赋给a

char  str[14]="I love China!";//定义字符数组str，并把字符串赋值给数组中各个元素
```

Character pointer

4.存储单元的内容。编译时为字符数组分配若干存储单元，以存放各元素的值，而对字符指针变量，只分配了一个存储单元。

5.指针变量的值是可以改变的，而字符数组名代表一个固定的值（数组首元素的地址），不能改变。

```
char *a = "I love China!";
a = a + 7;
printf("%s", a);
```



```
China!
```

6.字符数组中各元素的值是可以改变的（可以对它们再赋值），但字符指针变量指向的字符串常量中的内容是不可以被取代的（不能对它们再赋值）。

```
char *a = "House";
a[2] = 'r';  //非法，字符串常量不能改变
```

void pointer

- C99允许使用基类型为 void 的指针类型。
- 可以定义一个基类型为void 的指针变量(即void *型变量)，它不指向任何类型的数据。可以理解为“指向空类型”或“不指向确定的类型”的数据，只提供一个纯地址。
- 它可以用来指向一个抽象的类型的数据，在将它的值赋给另一指针变量时要进行强制类型转换使之适合于被赋值的变量的类型。

```
char *p1;
void *p2;
p1 = (char *)p2;
```

- 主要应用于调用动态存储分配函数时出现。如：

```
pt=(int *)malloc(100);
```

- 也可以应用于函数中：

```
void *fun(char ch1, char ch2)
p1 = (char*)fun(ch1,ch2);
```

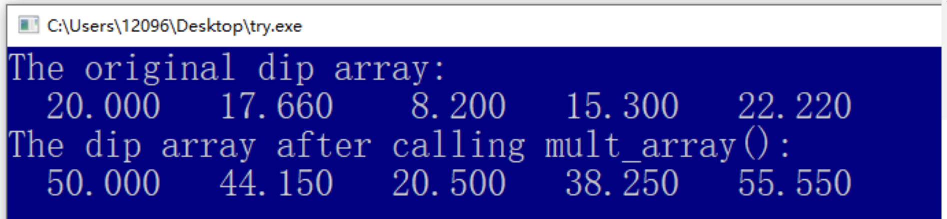
const

编写处理基本类型（如int）的函数时，要选择传递int类型的值还是传递指向int的指针。对于数组来说，必须传递指针，这样效率高。但是传递指针会导致函数可能会将原始数据修改，因此可以在函数原型和函数定义中声明形式参数时使用关键字const。如：

int sum(const int ar[],int n);

```
#include <stdio.h>
#define SIZE 5
void show_array(const double ar[], int n);//不改变数组
void mult_array(double ar[], int n, double mult);//改变数组

int main(void) {
    double dip[SIZE] = { 20.0, 17.66, 8.2, 15.3, 22.22 };
    printf("The original dip array:\n");
    show_array(dip, SIZE);
    mult_array(dip, SIZE, 2.5);
    printf("The dip array after calling mult_array():\n");
    show_array(dip, SIZE);
    return 0;
}
```

```
/* 显示数组的内容 */
void show_array(const double ar[], int n) {
        int i;
        for (i = 0; i < n; i++)
                printf("%8.3f ", ar[i]);
        putchar('\n');
}

/* 把数组的每个元素都乘以相同的值 */
void mult_array(double ar[], int n, double mult) {
        int i;
        for (i = 0; i < n; i++)
                ar[i] *= mult;
```


C:\Users\12096\Desktop\try.exe

```
The original dip array:
  20.000    17.660     8.200    15.300    22.220
The dip array after calling mult_array():
  50.000    44.150    20.500    38.250    55.550
```

const

假设已有：

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double locked[4]={0.08,0.075,0.0725,0.07};
```

1.指针为const类型，表明不能通过指针修改其所向指的值。

```
const double *pd = rates;
*pd = 29.89;          //× 不允许
pd[2] = 222.22;       //×不允许
rates[0] = 99.99; // √允许，因为rates未被const限定
```

可以让pd指向别处

```
pd++;  /* 让pd指向rates[1] -- 没问题 */
```

2.把const数据或非const数据的地址初始化为指向const的指针或为其赋值是合法的

```
const double *pc = rates;  // √有效
pc = locked;     //√有效
pc = &rates[3]; //√有效
```

const

假设已有：

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double locked[4]={0.08,0.075,0.0725,0.07};
```

3.可以声明并初始化一个不能指向别处的指针

```
double *const pc = rates; // pc指向数组的开始
pc = &rates[2]; // ×不允许，因为该指针不能指向别处
*pc = 92.99;    // √没问题 -- 更改rates[0]的值
```

4.在创建指针时还可以使用const两次，该指针既不能更改它所指向的地址，也不能修改指向地址上的值

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double *const pc = rates;
// double double *const pc = rates; the same
pc = &rates[2]; //×不允许
*pc = 92.99;    //×不允许
```

Content

Memory management

C provides several functions for memory allocation and management.

function	Description
calloc(int num, int size)	Allocate an array of **num** elements each with **size (in byte)**
malloc(int num)	Allocate an array of num bytes and leave them initialized
realloc(void *addr, int newsize)	Re-allocate memory at **addr**ess with **newsize**
free(void *addr)	Release a block of memory at **addr**ess

Memory management

You must use this library to use memory management function

```
#include <stdlib.h>
```

calloc() function

Fixed array size, fixed memory

```
char name[100];
```

```
char *name;
name = (char*)calloc(100, sizeof(char));
```

Dynamic memory at address of name (100 bytes)

calloc() function

Fixed array size, fixed memory

```
int name[100];
```

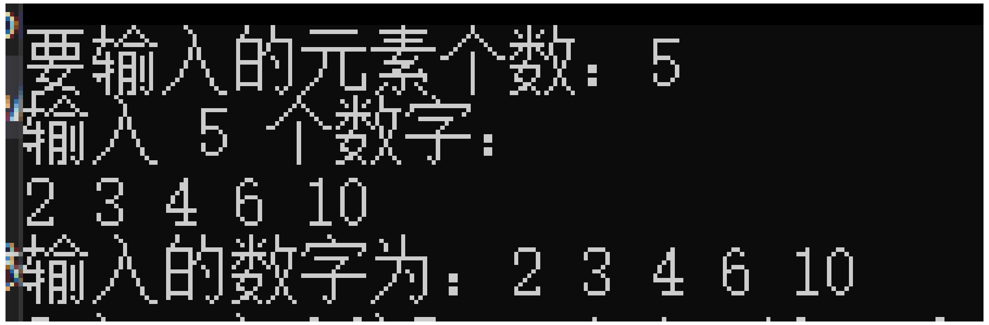
```
int *name;
name = (int*)calloc(100, sizeof(int));
```

How many bytes in total???

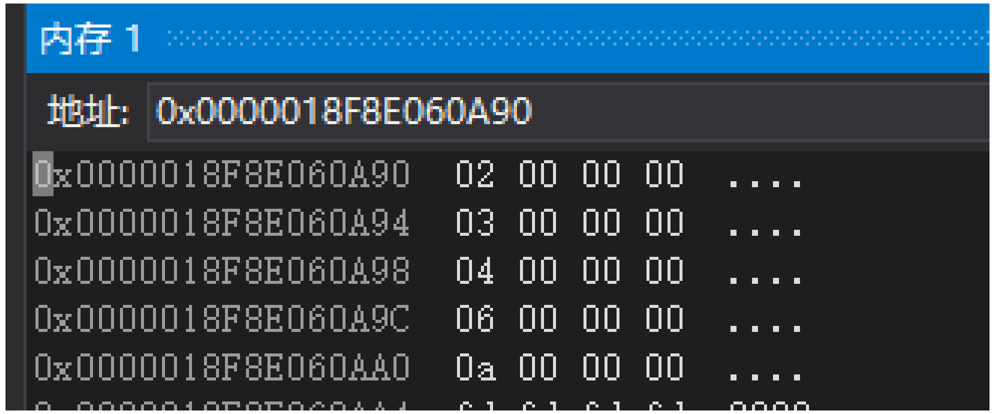
calloc() function

How to use calloc() to allocate memory for a pointer?

```c
#include <stdio.h>
#include <stdlib.h>
main()
{
        int n;
        printf("要输入的元素个数: ");
        scanf("%d", &n);
        int* test_array = (int*)calloc(n, sizeof(int));
        printf("输入 %d 个数字: \n", n);
        for (int i = 0; i < n; i++){
                scanf("%d", &test_array[i]);
        }
        printf("输入的数字为: ");
        for (int i = 0; i < n; i++) {
                printf("%d ", test_array[i]);
        }
        free(test_array);
}
```



```
要输入的元素个数: 5
输入 5 个数字:
2 3 4 6 10
输入的数字为: 2 3 4 6 10
```



```
内存 1
地址: 0x0000018F8E060A90
0x0000018F8E060A90  02 00 00 00  ....
0x0000018F8E060A94  03 00 00 00  ....
0x0000018F8E060A98  04 00 00 00  ....
0x0000018F8E060A9C  06 00 00 00  ....
0x0000018F8E060AA0  0a 00 00 00  ....
```

malloc() function

Fixed array size, fixed memory

```
char name[100];
```

```
char *name;
name = (char*)malloc(100*sizeof(char));
```

Dynamic memory at address of name (100 bytes)

malloc() function

Fixed array size, fixed memory

```
float name[30];
```

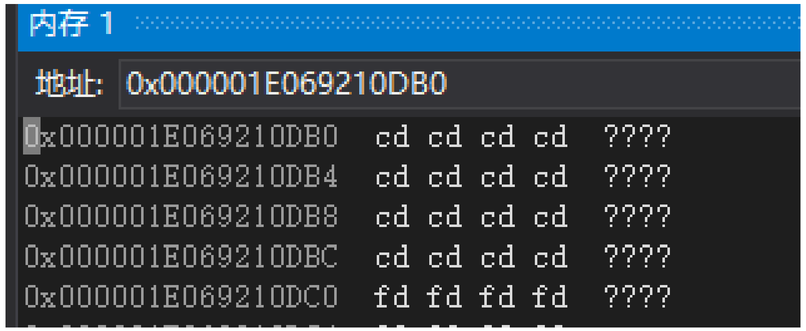
```
float *name;
name = (float*)malloc(30*sizeof(float));
```

How many bytes in total?

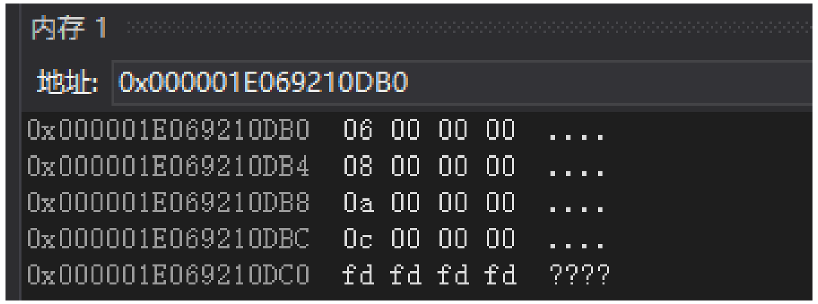
malloc() function

How to use malloc( ) to allocate memory for a pointer?

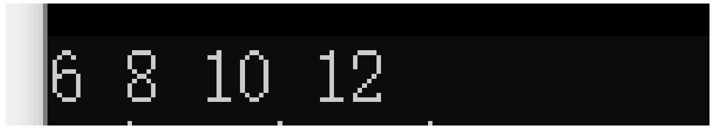
```
int* add_mats(int A[], int B[],int n)
{
    int* C = (int*)malloc(sizeof(int) * n);
    for (int i = 0; i < n; i++)
        C[i] = B[i] + A[i];
    return C;
}

main(void)
{
    int A[4] = {1,2,3,4};
    int B[4] = {5,6,7,8};
    int *C = add_mats(A,B,4);
    for (int i = 0; i < 4 ; i++){
        printf("%d ",C[i]);
    }
}
```



```
内存 1
地址: 0x000001E069210DB0
0x000001E069210DB0  cd cd cd cd  ????
0x000001E069210DB4  cd cd cd cd  ????
0x000001E069210DB8  cd cd cd cd  ????
0x000001E069210DBC  cd cd cd cd  ????
0x000001E069210DC0  fd fd fd fd  ????
```



```
内存 1
地址: 0x000001E069210DB0
0x000001E069210DB0  06 00 00 00  ....
0x000001E069210DB4  08 00 00 00  ....
0x000001E069210DB8  0a 00 00 00  ....
0x000001E069210DBC  0c 00 00 00  ....
0x000001E069210DC0  fd fd fd fd  ????
```



```
6 8 10 12
```

calloc() & malloc()

```
char *name;

name = (char*)calloc(200, sizeof(char));

name = (char*)malloc(200*sizeof(char));
```

calloc() & malloc()

calloc()

contiguous/连续的
allocation

↓

allocates memory and
initializes all bits to zero

malloc()

memory
allocation

↓

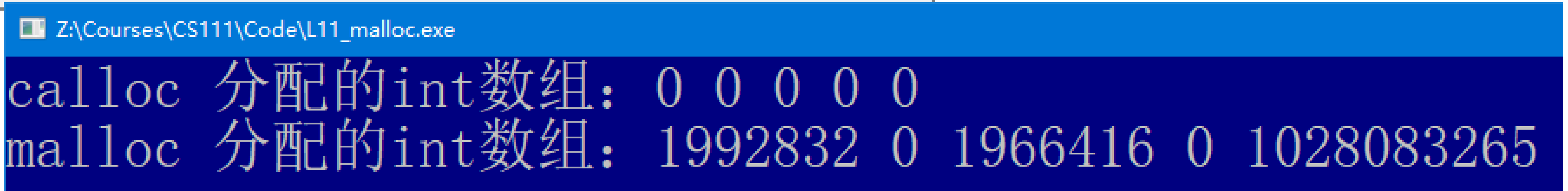
allocates memory and leaves
the memory uninitialized,
faster

Comparison on malloc() & calloc()

```
#include<stdio.h>
#include<stdlib.h>
int main(void)
{
    int n;
    printf("要输入的元素个数: ");
    scanf("%d", &n);
    int *test_array = (int*)calloc(n, sizeof(int));
    int *test_array1 = (int*)malloc(sizeof(int)*n);
    printf("calloc 分配的int数组: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", test_array[i]);
    }
    printf("\nmalloc 分配的int数组: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", test_array1[i]);
    }
}
```

calloc(): The space is initialized to zero.

malloc(): The space is randomly initialized.



Z:\Courses\CS111\Code\L11_malloc.exe

```
calloc 分配的int数组: 0 0 0 0 0
malloc 分配的int数组: 1992832 0 1966416 0 1028083265
```

realloc() function

realloc()

Repeatedly allocation

Re-allocates memory to the pointer variable

Increase memory

Decrease memory

realloc() function

Allocate memory at address of name (200 bytes)

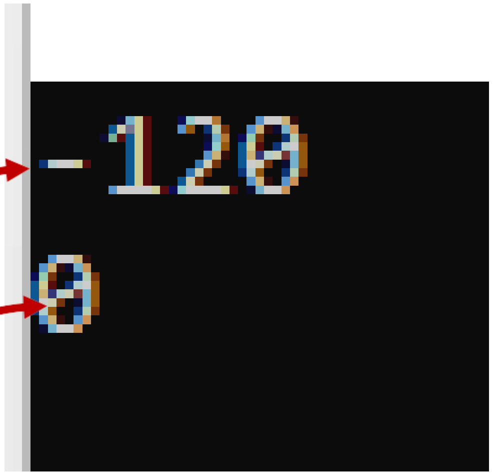
```
char *name;
name = (char*)malloc(200*sizeof(char));


name = (char*)realloc(name, 100*sizeof(char));
```

Resize the merry at address of name (100 bytes)

realloc() function

Why using realloc()?

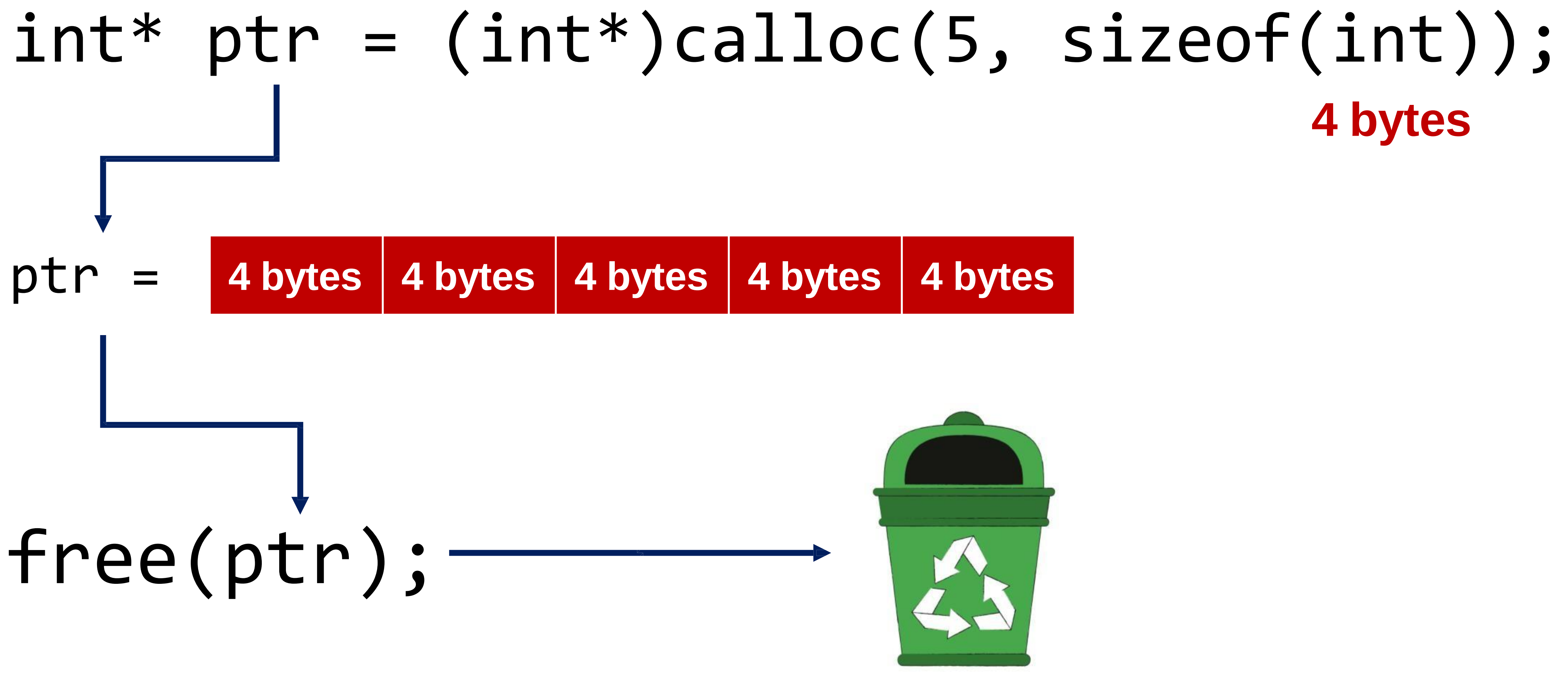
```
#include<stdio.h>
#include<stdlib.h>

int main()
{
    int *ptr;
    ptr = malloc(5 * sizeof(int));
    ptr[3] = -120;

    ptr = realloc(ptr, 10 * sizeof(int));
    printf("%d\n", ptr[3]);
    ptr = calloc( 5, sizeof(int));

    printf("%d", ptr[3]);

    return 0;
}
```



```
-120
0
```

If using malloc, the value will be erased!

free() function



```
int* ptr = (int*)calloc(5, sizeof(int));
```

```
free(ptr);
```

Summary

- **Pointer** is a variable that stores the address of another variable.
- We can access the **memory address** directly using the pointer.
- By changing the pointer value, the value stored at the address can be modified, typically useful for **functions** in transferring values.
- Pointer has **arithmetic and logical operations** (++, --, ==, >, <) on manipulating the memory address.
- We can **manage the memory** using C provided functions in **stdlib.h**.