程序设计基础

第八章: 指针

刘新国

浙江大学计算机学院
浙江大学 CAD&CG 国家重点实验室

November 24, 2021

内容提要

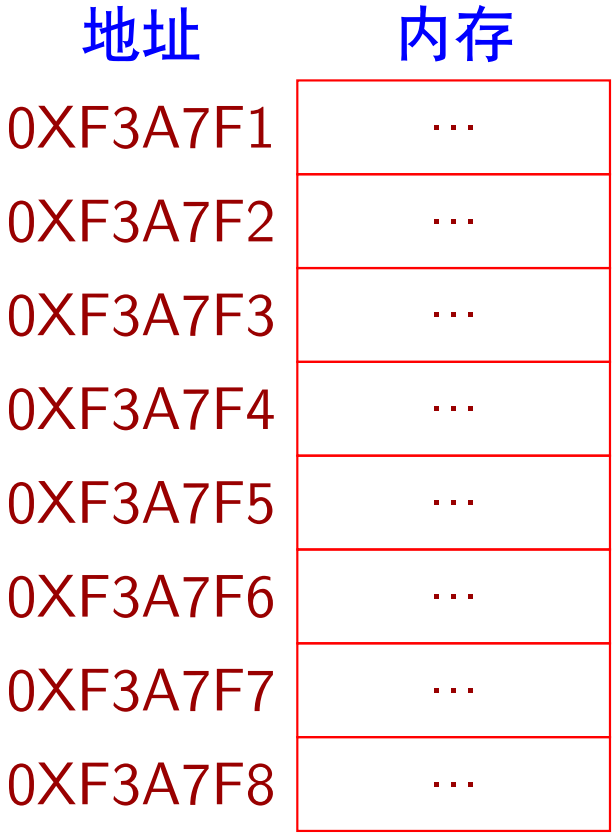
内容提要

地址与指针

地址：内存的地址，或者字节的地址

- 地址也是编号，字节在内存中编号
- 是一个无符号整数，以字节为单位
- 程序的变量和代码保存在内存中
- 通过地址访问内存的数据和代码

```
// 将地址为0XF3A7F5的内存
// 数据移入寄存器ax
mov ax,0XF3A70
mov ds,ax
mov ax,[F5]
```



地址与指针

指针：一种数据类型

- 存储变量的地址：变量的起始字节的地址

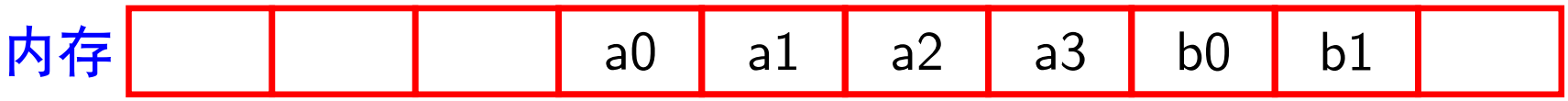
指针变量的定义

类型名 * 指针变量名;

```
int *p;          // 定义了一个整型指针变量：p
float *x, *y;    // 定义了2个浮点数指针变量：x和y
float a, *ap;    // 定义了一个浮点型变量：a
                 // 和一个浮点数指针变量：ap
```

地址与指针

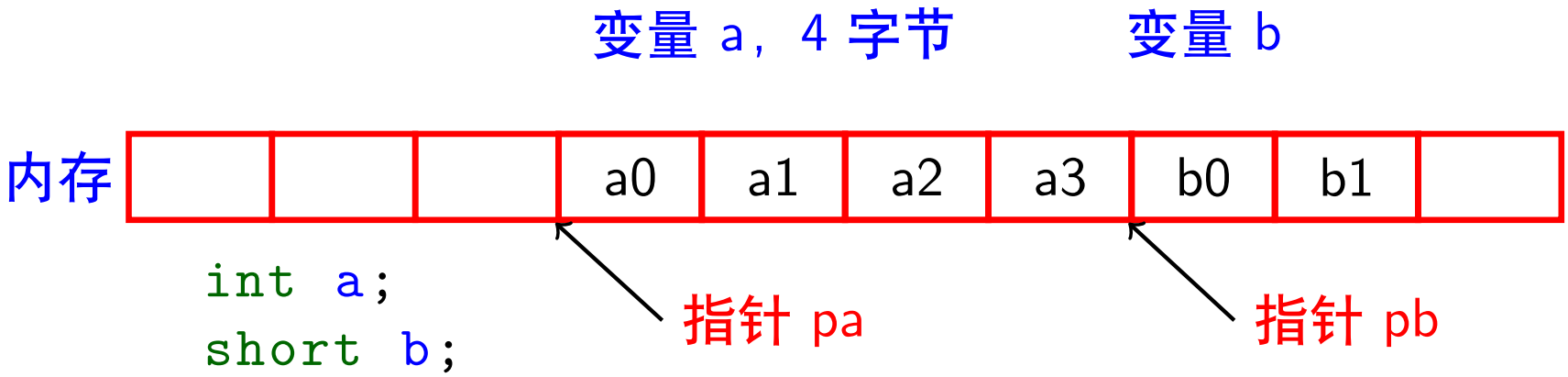
指针存储变量的地址：起始字节的地址




```
int a;
short b;
```

地址与指针

指针存储变量的地址：起始字节的地址

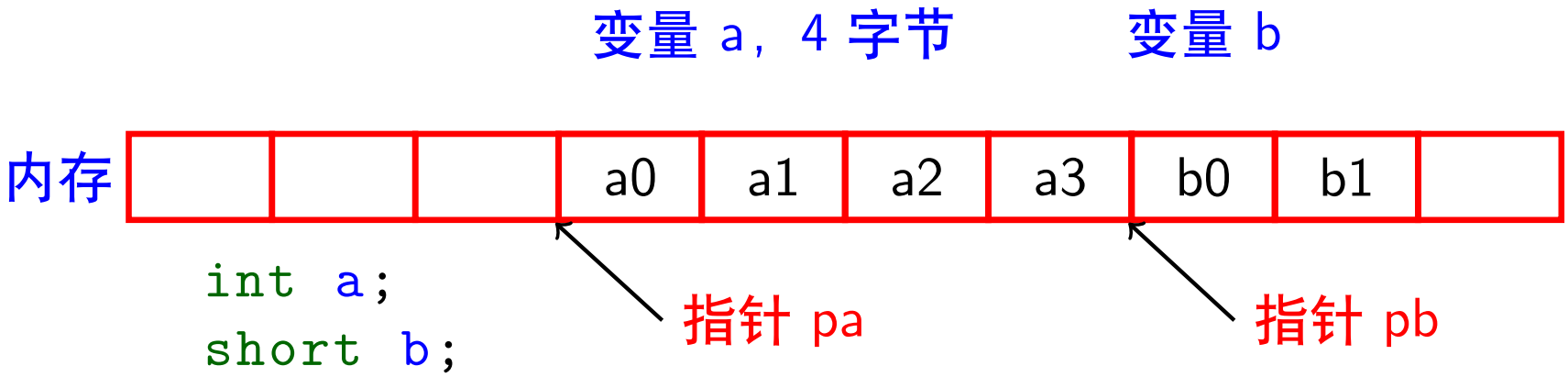


- 指针可以保存变量地址

```
int *pa; short *pb;
pa = & a; pb = & b;
```

地址与指针

指针存储变量的地址：起始字节的地址



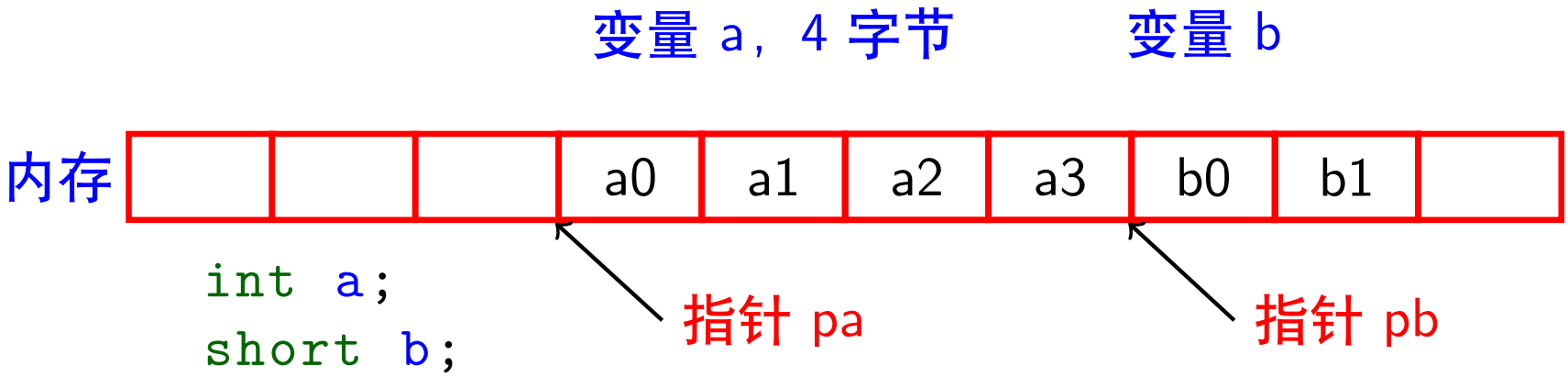
- 指针可以保存变量地址

```
int *pa; short *pb;
pa = & a; pb = & b;
```

- 通过指针访问和使用变量

地址与指针

指针存储变量的地址：起始字节的地址



- 指针可以保存变量地址

```
int *pa; short *pb;
pa = & a; pb = & b;
```

- 通过指针访问和使用变量

```
*pa = 5;   //相当于a=5;
*pb = a;   //相当于b=a;
*pb = *pa;//相当于b=a;
```

指针的属性

- 指针的值：
  - 指针是一种数据类型，可以定以指针变量，指针常量。
  - 指针的值是一个整数，表示内存地址或者变量地址。
  - 我们常说“一个指针指向一个变量”，意义是：这个指针存储了这个变量的地址。

指针的属性

- 指针的值：
  - 指针是一种数据类型，可以定以指针变量，指针常量。
  - 指针的值是一个整数，表示内存地址或者变量地址。
  - 我们常说“一个指针指向一个变量”，意义是：这个指针存储了这个变量的地址。
- 指针的基类型：
  - 指针所指向的变量的类型。
  - 定义指针变量时，需要指定指针基类型
  - 指针的基类型也可以一种指针。这种指针称为二级指针。

指针的基本运算

指向变量：获取地址

```
float a, *p;
p = &a; //运算符 & 获取变量指针
```

指针的基本运算

指向变量：获取地址

```
float a, *p;
p = &a; //运算符 & 获取变量指针
```

获取指针运算符 &：

- 只能作用在拥有内存的存储单元或变量上面
- 表达式&a的类型是指针，值是变量 a 的地址

指针的基本运算

指向变量：获取地址

```
float a, *p;
p = &a; //运算符 & 获取变量指针
```

获取指针运算符 &：

- 只能作用在拥有内存的存储单元或变量上面
- 表达式&a的类型是指针，值是变量 a 的地址

访问变量：通过指针使用变量

```
*p = 1.0;                /* 等价于 a = 1.0; */
*p = *p * *p + *p; /* 等价于 a = a * a + a; */
```

* 运算符：作用在指针上，访问指针所指向的变量

指针与变量地址

```
float x = 3, y = 4;
short a = 1, b = 2, *p = &a;
```

地址	内存	
1920986372	1	a
1920986374	2	b
1920986376	3	x
1920986380	4	y
1920986384	1920986372	p

指针与变量地址

```
float x = 3, y = 4;
short a = 1, b = 2, *p = &a;

printf("&x = %u\n", &x );
printf("&y = %u\n", &y );
printf("&a = %u\n", &a );
printf("&b = %u\n", &b );
printf("&p = %u\n", &p );
printf(" p = %u\n",  p );
```

地址	内存	
1920986372	1	a
1920986374	2	b
1920986376	3	x
1920986380	4	y
1920986384	1920986372	p

指针与变量地址

```
float x = 3, y = 4;
short a = 1, b = 2, *p = &a;

printf("&x = %u\n", &x );
printf("&y = %u\n", &y );
printf("&a = %u\n", &a );
printf("&b = %u\n", &b );
printf("&p = %u\n", &p );
printf(" p = %u\n",  p );
```

地址	内存	
1920986372	1	a
1920986374	2	b
1920986376	3	x
1920986380	4	y
1920986384	1920986372	p

输出：

```
&x = 1920986376
&y = 1920986380
&a = 1920986372
&b = 1920986374
&p = 1920986384
 p = 1920986372
```

程序阅读，输出结果是什么？

```c
#include <stdio.h>
int main( void )
{
    int a, *p;
    p = &a;
    a = 3;
    printf("a = %d, *p = %d\n", a, *p);
    *p = 10;
    printf("a = %d, *p = %d\n", a, *p);
    printf("Enter a:");
    scanf("%d", p); //或 scanf("%d", &a);
    printf("a = %d, *p = %d\n", a, *p);
    (*p) ++;
    printf("a = %d, *p = %d\n", a, *p);
    return 0;
}
```

程序阅读，输出结果是什么？

```c
#include <stdio.h>
int main( void )
{
    int a, *p;
    p = &a;
    a = 3;
    printf("a = %d, *p = %d\n", a, *p);
    *p = 10;
    printf("a = %d, *p = %d\n", a, *p);
    printf("Enter a:");
    scanf("%d", p); //或 scanf("%d", &a);
    printf("a = %d, *p = %d\n", a, *p);
    (*p) ++;
    printf("a = %d, *p = %d\n", a, *p);
    return 0;
}
```

程序输出为：

```
a = 3, *p = 3
a = 10, *p = 10
Enter a:5
a = 5, *p = 5
a = 6, *p = 6
```

程序阅读，输出结果是什么？

详解：(*p)++

- *p 是通过指针 p 获得 p 所指向的变量

程序阅读，输出结果是什么？

详解：(*p)++

- *p 是通过指针 p 获得 p 所指向的变量
- 运算符 ++ 作用在 *p 之上，使之自增

程序阅读，输出结果是什么？

详解：(*p)++

- *p 是通过指针 p 获得 p 所指向的变量
- 运算符 ++ 作用在 *p 之上，使之自增

如果不加 ( ) 呢？　　*p++

程序阅读，输出结果是什么？

详解：(*p)++

- *p 是通过指针 p 获得 p 所指向的变量
- 运算符 ++ 作用在 *p 之上，使之自增

如果不加 ( ) 呢？　　*p++

- * 和 ++ 都是单目运算符，具有相同的优先级

程序阅读，输出结果是什么？

详解：(*p)++

- *p 是通过指针 p 获得 p 所指向的变量
- 运算符 ++ 作用在 *p 之上，使之自增

如果不加 ( ) 呢？　　*p++

- * 和 ++ 都是单目运算符，具有相同的优先级
- 单目运算符从右开始结合，所以结合顺序是 *(p ++)

程序阅读，输出结果是什么？

详解：(*p)++

- *p 是通过指针 p 获得 p 所指向的变量
- 运算符 ++ 作用在 *p 之上，使之自增

如果不加 ( ) 呢？　　*p++

- * 和 ++ 都是单目运算符，具有相同的优先级
- 单目运算符从右开始结合，所以结合顺序是 *(p ++)
- 运算符 ++ 作用在 p 之上，使得指针本身自增，
  即: $p = p + 1$

指针的基本运算

使用指针的注意事项：假设有指针 p

指针的基本运算

使用指针的注意事项：假设有指针 p

- 使用 *p 之前，要保证 p 已经被赋值，并且指向了一个变量或实际的内存单元

指针的基本运算

使用指针的注意事项：假设有指针 p

- 使用 *p 之前，要保证 p 已经被赋值，并且指向了一个变量或实际的内存单元
- 地址值可以是 0，称为空指针。空指针不指向任何实际的内存单元。

指针的基本运算

使用指针的注意事项：假设有指针 p

- 使用 *p 之前，要保证 p 已经被赋值，并且指向了一个变量或实际的内存单元
- 地址值可以是 0，称为空指针。空指针不指向任何实际的内存单元。
- 如果指针 p 的地址值为0，或者没有指向合法的变量，那么执行 *p 将会发生严重运行错误

指针的基本运算

使用指针的注意事项：假设有指针 p

- 使用 *p 之前，要保证 p 已经被赋值，并且指向了一个变量或实际的内存单元
- 地址值可以是 0，称为空指针。空指针不指向任何实际的内存单元。
- 如果指针 p 的地址值为0，或者没有指向合法的变量，那么执行 *p 将会发生严重运行错误
- C 语言标准库里定义了空指针 – NULL

  #define NULL 0

内容提要

变量交换函数

变量交换的三部曲

```
temp = a; a = b; b = temp;
```

变量交换函数

变量交换的三部曲

```
temp = a; a = b; b = temp;
```

设计一个交换函数

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

变量交换函数

变量交换的三部曲

```
temp = a; a = b; b = temp;
```

设计一个交换函数

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

```
int main( void )
{
    int a = 1, b = 2;
    swap(a, b);
    printf("a = %d ", a);
    printf("b = %d ", b);
}
```

变量交换函数

变量交换的三部曲

```
temp = a; a = b; b = temp;
```

设计一个交换函数

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

- 输出是: a = 1 b = 2

```
int main( void )
{
    int a = 1, b = 2;
    swap(a, b);
    printf("a = %d ", a);
    printf("b = %d ", b);
}
```

变量交换函数

变量交换的三部曲

```
temp = a; a = b; b = temp;
```

设计一个交换函数

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

- 输出是: a = 1 b = 2
- a 和 b 没交换, why ?

```
int main( void )
{
    int a = 1, b = 2;
    swap(a, b);
    printf("a = %d ", a);
    printf("b = %d ", b);
}
```

变量交换函数

变量交换的三部曲

```
temp = a; a = b; b = temp;
```

设计一个交换函数

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

```
int main( void )
{
    int a = 1, b = 2;
    swap(a, b);
    printf("a = %d ", a);
    printf("b = %d ", b);
}
```

- 输出是: a = 1 b = 2
- a 和 b 没交换, why ?
  - swap 交换的是自己的局部变量 a 和 b，并非调用者 (main 函数) 的 a 和 b

变量交换函数

变量交换的三部曲

```
temp = a; a = b; b = temp;
```

设计一个交换函数

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

```
int main( void )
{
    int a = 1, b = 2;
    swap(a, b);
    printf("a = %d ", a);
    printf("b = %d ", b);
}
```

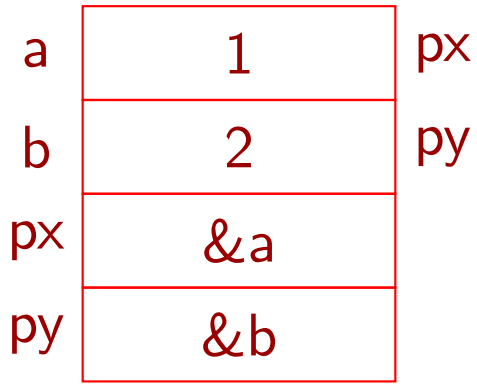
- 输出是: a = 1 b = 2
- a 和 b 没交换, why ?
  - swap 交换的是自己的局部变量 a 和 b，并非调用者 (main 函数) 的 a 和 b
  - 调用者 (main 函数) 把自己 a 和 b 的值传给了 swap 函数

变量交换函数

如何设计一个变量交换函数呢？

可以使用指针实现这个目标

```
void swap(int*px, int*py)
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
int main( void )
{
    int a = 1, b = 2;
    swap(&a, &b);
    printf("a = %d",a);
    printf("b = %d",b);
}
```

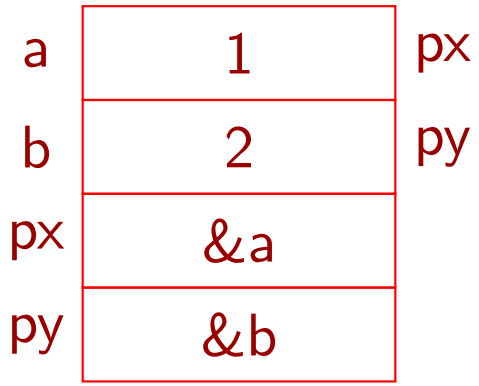


变量交换函数

如何设计一个变量交换函数呢？

可以使用指针实现这个目标

```
void swap(int*px, int*py)
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
int main( void )
{
    int a = 1, b = 2;
    swap(&a, &b);
    printf("a = %d",a);
    printf("b = %d",b);
}
```



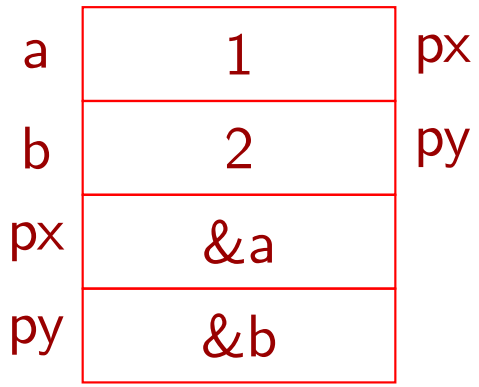
- swap 函数采用指针类型的形式参数

变量交换函数

如何设计一个变量交换函数呢？

可以使用指针实现这个目标

```
void swap(int*px, int*py)
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
int main( void )
{
    int a = 1, b = 2;
    swap(&a, &b);
    printf("a = %d",a);
    printf("b = %d",b);
}
```



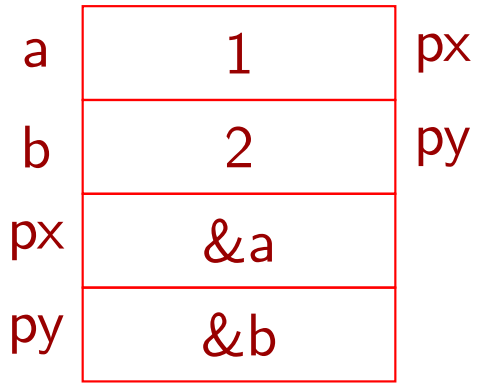
- swap 函数采用指针类型的形式参数
- 调用者 main 函数将自己变量的指针传递给 swap 函数

变量交换函数

如何设计一个变量交换函数呢？

可以使用指针实现这个目标

```
void swap(int*px, int*py)
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
int main( void )
{
    int a = 1, b = 2;
    swap(&a, &b);
    printf("a = %d",a);
    printf("b = %d",b);
}
```



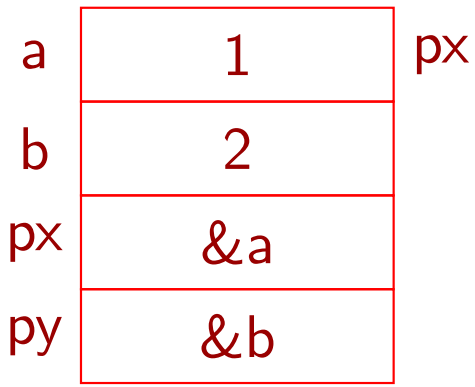
- swap 函数采用指针类型的形式参数
- 调用者 main 函数将自己变量的指针传递给 swap 函数
- swap 函数根据指针访问 main 函数中的变量，并交换它们

变量交换函数

下面的交换函数可以吗?

直接交换指针可以达到目的吗？

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

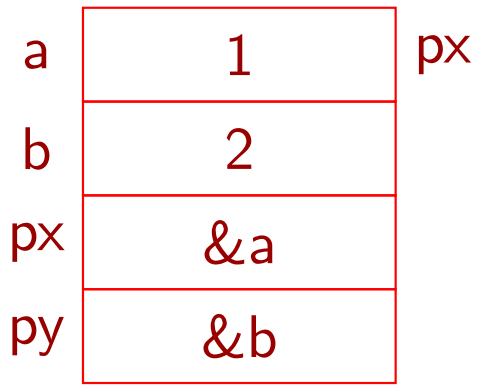


变量交换函数

下面的交换函数可以吗?

直接交换指针可以达到目的吗？

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



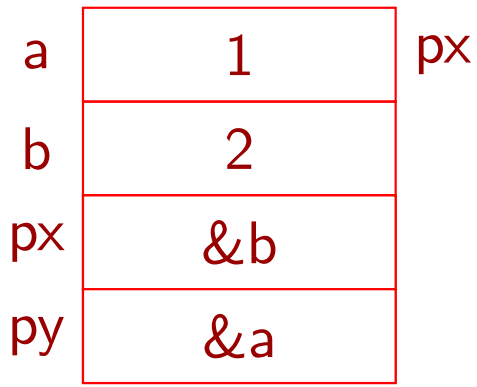
显然是不行的！只交换了指针变量 a,b 的地址，并没有通过指针交换 main 函数中对应的变量 a 和 b

变量交换函数

下面的交换函数可以吗?

直接交换指针可以达到目的吗？

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



显然是不行的！只交换了指针变量 a,b 的地址，并没有通过指针交换 main 函数中对应的变量 a 和 b

用指针当函数的参数，传递计算结果 – 例 8-4

输入年份和该年的第几天：2021 330
2021年的第330天是：2021年11月26日

用指针当函数的参数，传递计算结果 – 例 8-4

输入年份和该年的第几天：2021 330
2021年的第330天是：2021年11月26日

```
void month_day(int year, int yearday, int *
    pMonth, int *pDay);
```

用指针当函数的参数，传递计算结果 – 例 8-4

```
输入年份和该年的第几天：2021 330
2021年的第330天是：2021年11月26日
```

```c
void month_day(int year, int yearday, int *
    pMonth, int *pDay);

int main(void)
{
    int day, month, year, yearday;
    printf("输入年份和该年的第几天：");
    scanf("%d%d", &year, &yearday);
    month_day(year, yearday, &month, &day);
    printf("%d年的第%d天是：%d年%d月%d日\n",
        year, yearday, year, month, day);
    return 0;
}
```

用指针当函数的参数，传递计算结果 – 例 8-4

```
void month_day(int year, int yearday, int *pMonth,
    int *pDay)
{
    int k, leap = year%4==0 && year%100!=0 || year
        %400==0;
    int days[][13] = { /* 每个月的天数 */
        {0,31,28,31,30,31,30,31,31,30,31,30,31},
        {0,31,29,31,30,31,30,31,31,30,31,30,31}};
    for( k = 1; yearday>days[leap][k]; k++ )
        yearday -= days[leap][k]; //减去本月天数
    *pMonth = k;     //保存结果到指针指向的变量
    *pDay = yearday;//保存结果到指针指向的变量
}
```

- 类似于 swap 函数，monthday 有两个参数是指针类型

用指针当函数的参数，传递计算结果 – 例 8-4

```
void month_day(int year, int yearday, int *pMonth,
    int *pDay)
{
    int k, leap = year%4==0 && year%100!=0 || year
        %400==0;
    int days[][13] = { /* 每个月的天数 */
        {0,31,28,31,30,31,30,31,31,30,31,30,31},
        {0,31,29,31,30,31,30,31,31,30,31,30,31}};
    for( k = 1; yearday>days[leap][k]; k++ )
        yearday -= days[leap][k]; //减去本月天数
    *pMonth = k;      //保存结果到指针指向的变量
    *pDay = yearday;//保存结果到指针指向的变量
}
```

- 类似于 swap 函数，monthday 有两个参数是指针类型
- 调用时，实际参数指向调用者的变量，回传计算结果
  month_day(year, yearday, &month, &day);

内容提要

数组名具有两重意义

数组名具有两重意义

- 作为变量，代表整个数组本身

指针与数组

数组名具有两重意义

- 作为变量，代表整个数组本身
- 作为常量，也是一个指针，指向数组的首元素

数组名具有两重意义

- 作为变量，代表整个数组本身
- 作为常量，也是一个指针，指向数组的首元素
  - 地址值不能改变

数组名具有两重意义

- 作为变量，代表整个数组本身
- 作为常量，也是一个指针，指向数组的首元素
  - 地址值不能改变
  - 但是，所指向的变量（首元素）的值是可以改变的

数组名具有两重意义

- 作为变量，代表整个数组本身
- 作为常量，也是一个指针，指向数组的首元素
  - 地址值不能改变
  - 但是，所指向的变量（首元素）的值是可以改变的

指针与数组

数组名具有两重意义

- 作为变量，代表整个数组本身
- 作为常量，也是一个指针，指向数组的首元素
  - 地址值不能改变
  - 但是，所指向的变量（首元素）的值是可以改变的

```
int a[100];
```

- a 即是整个数组本身 (有 100 个元素)

指针与数组

数组名具有两重意义

- 作为变量，代表整个数组本身
- 作为常量，也是一个指针，指向数组的首元素
  - 地址值不能改变
  - 但是，所指向的变量（首元素）的值是可以改变的

```
int a[100];
```

- a 即是整个数组本身 (有 100 个元素)
- a 也是一个指针，指向数组的首元素 a[0]

指针与数组

数组名具有两重意义

- 作为变量，代表整个数组本身
- 作为常量，也是一个指针，指向数组的首元素
  - 地址值不能改变
  - 但是，所指向的变量（首元素）的值是可以改变的

```
int a[100];
```

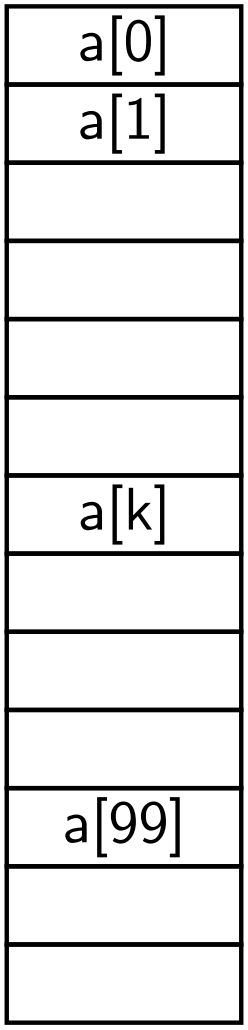
- a 即是整个数组本身 (有 100 个元素)
- a 也是一个指针，指向数组的首元素 a[0]

数组名作为指针时，不能被重新赋值指向其他变量

```
int a[100], c, *p;
a = &c; /* 编译时会报错 */
p = a;  /* 可以，p指向a[0] */
```

数组与指针和地址的关系

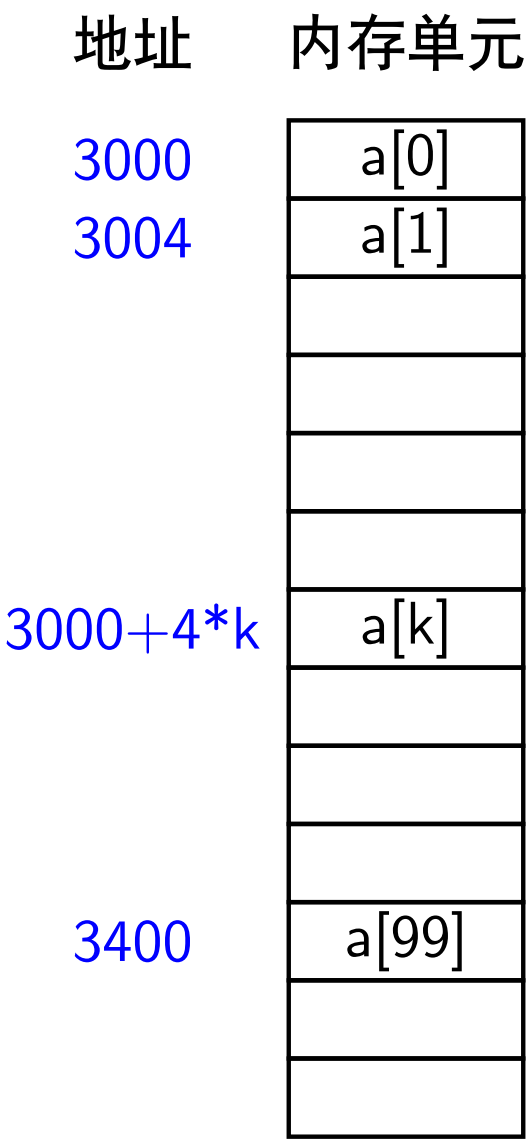
地址　　内存单元



指针运算

指针移动

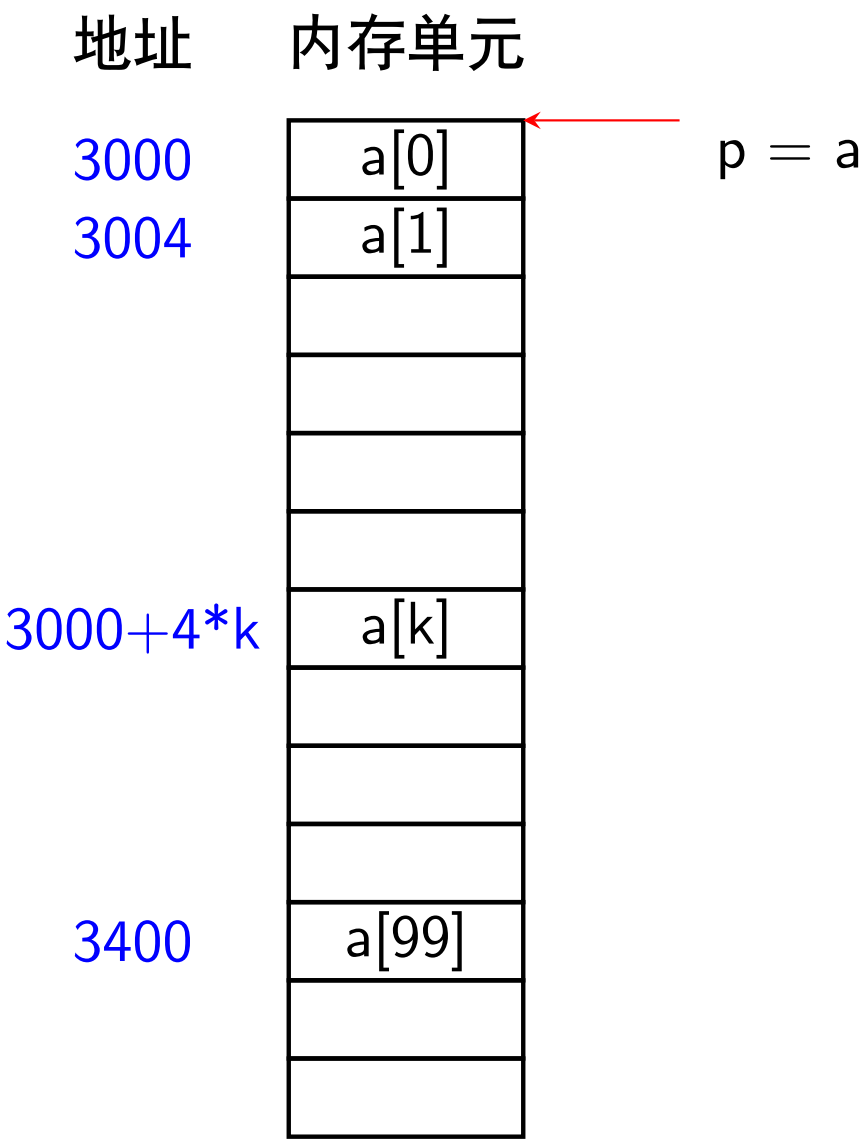
数组与指针和地址的关系



指针运算

指针移动

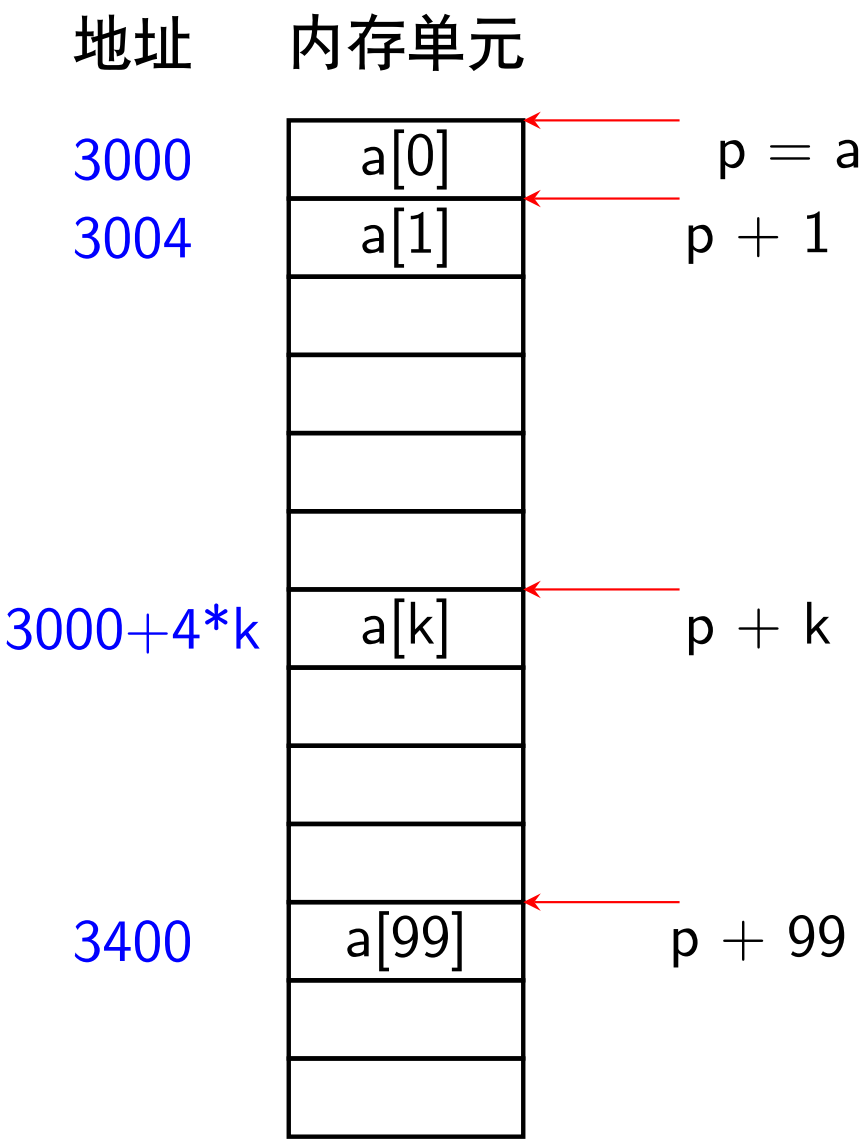
数组与指针和地址的关系



指针运算

指针移动

数组与指针和地址的关系

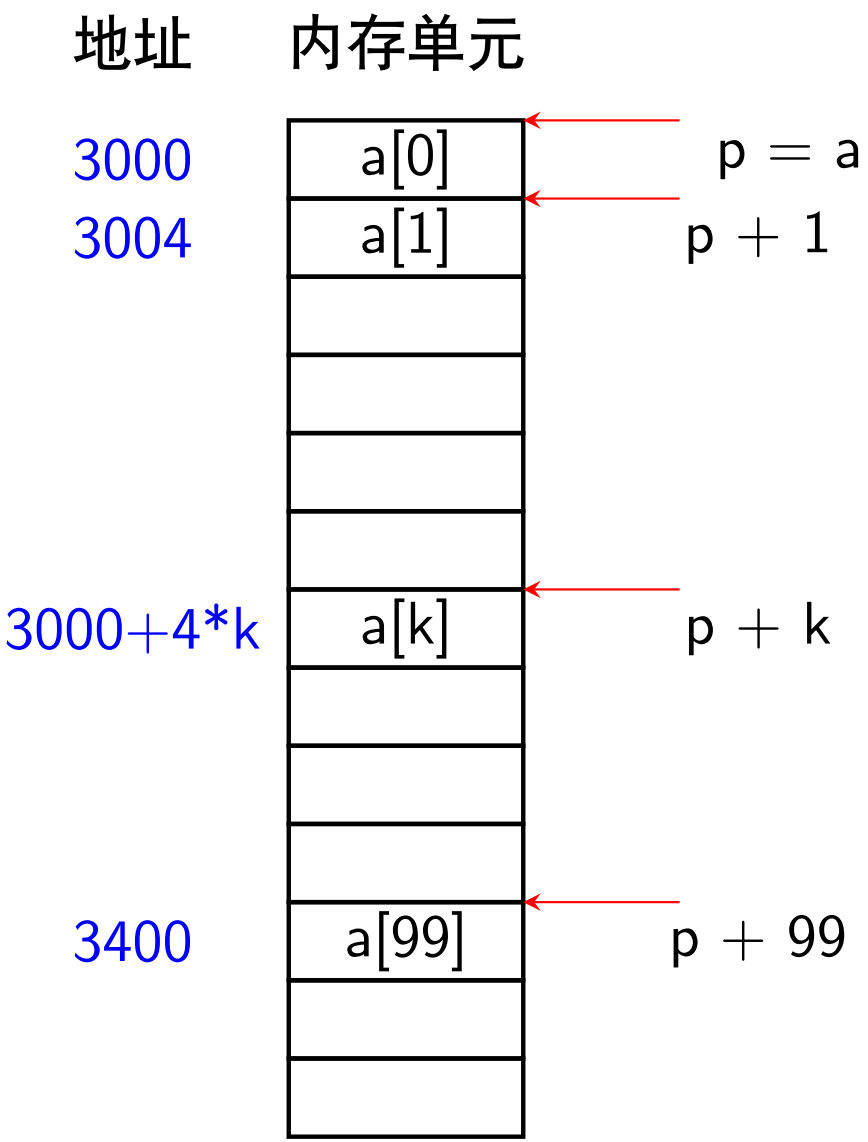


指针运算

- 指针加法：p + 整数表达式

指针移动

数组与指针和地址的关系

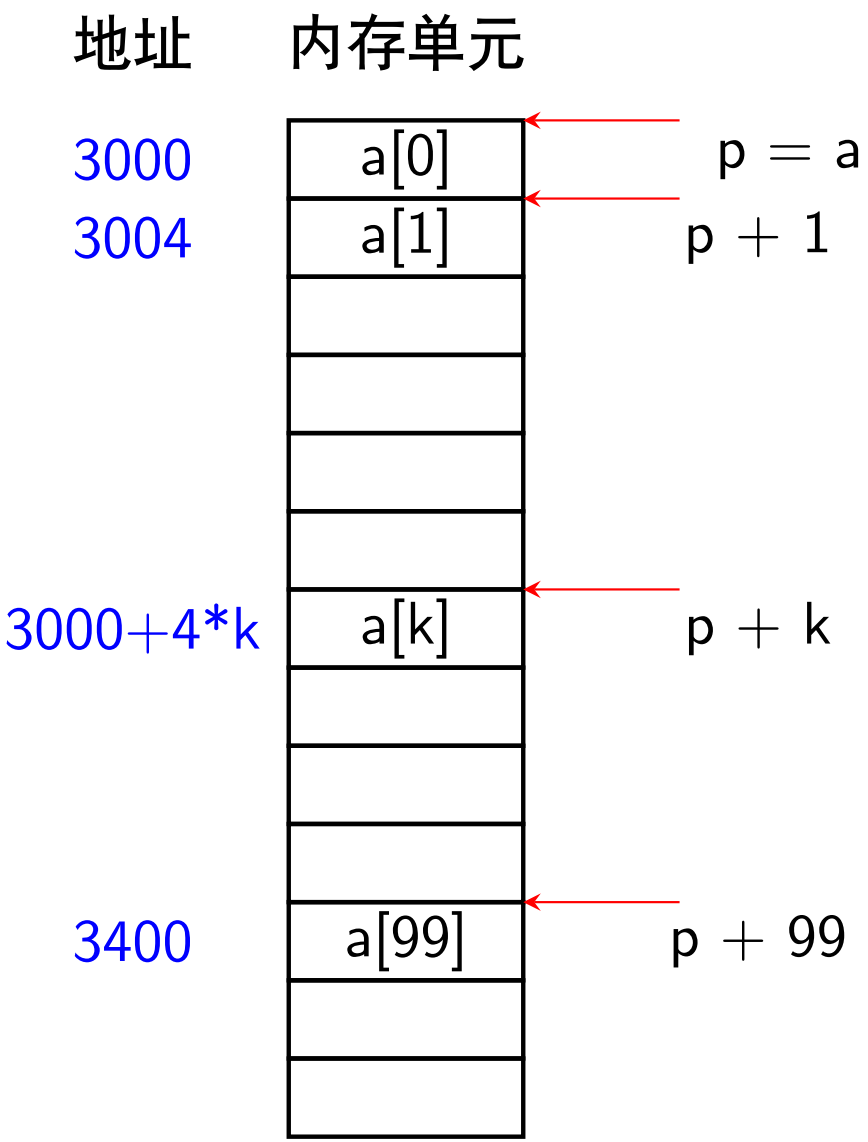


指针运算

- 指针加法：p + 整数表达式
  - 指针之后的若干元素

指针移动

数组与指针和地址的关系

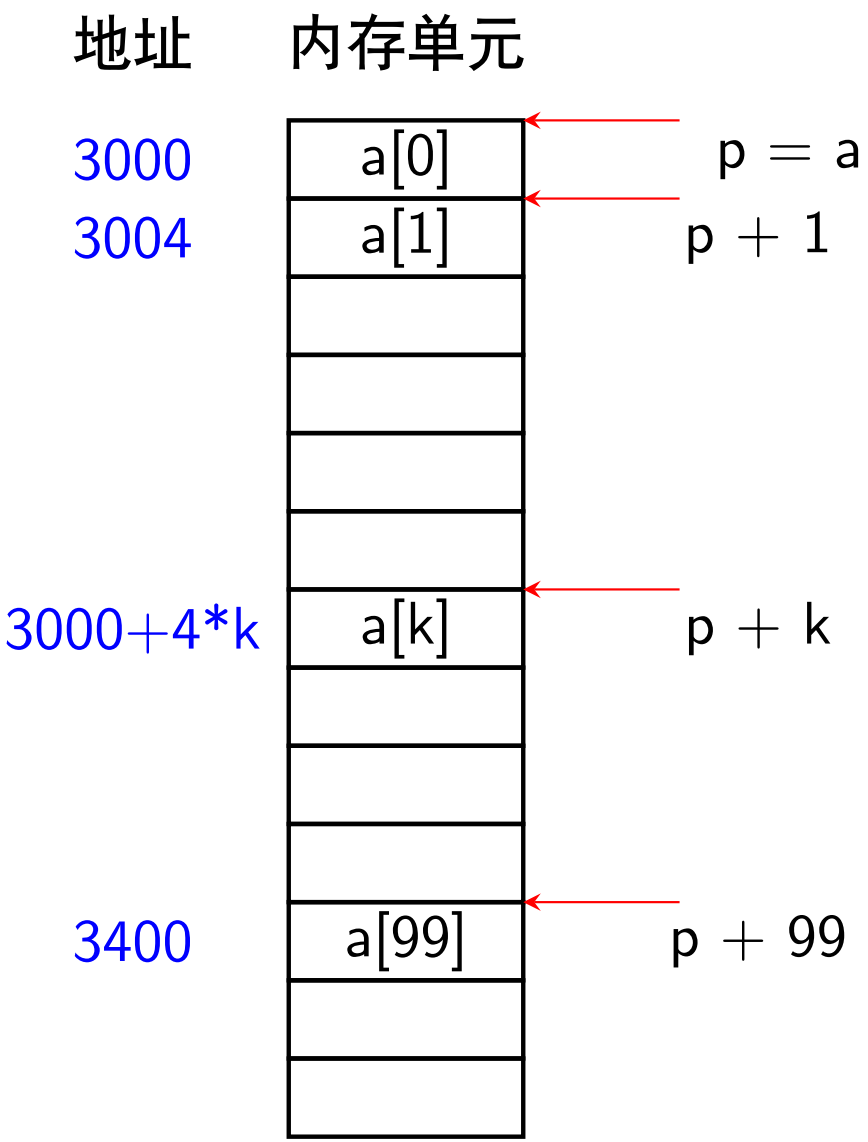


指针运算

- 指针加法：p + 整数表达式
  - 指针之后的若干元素
- 指针减法：p – 整数表达式

指针移动

数组与指针和地址的关系

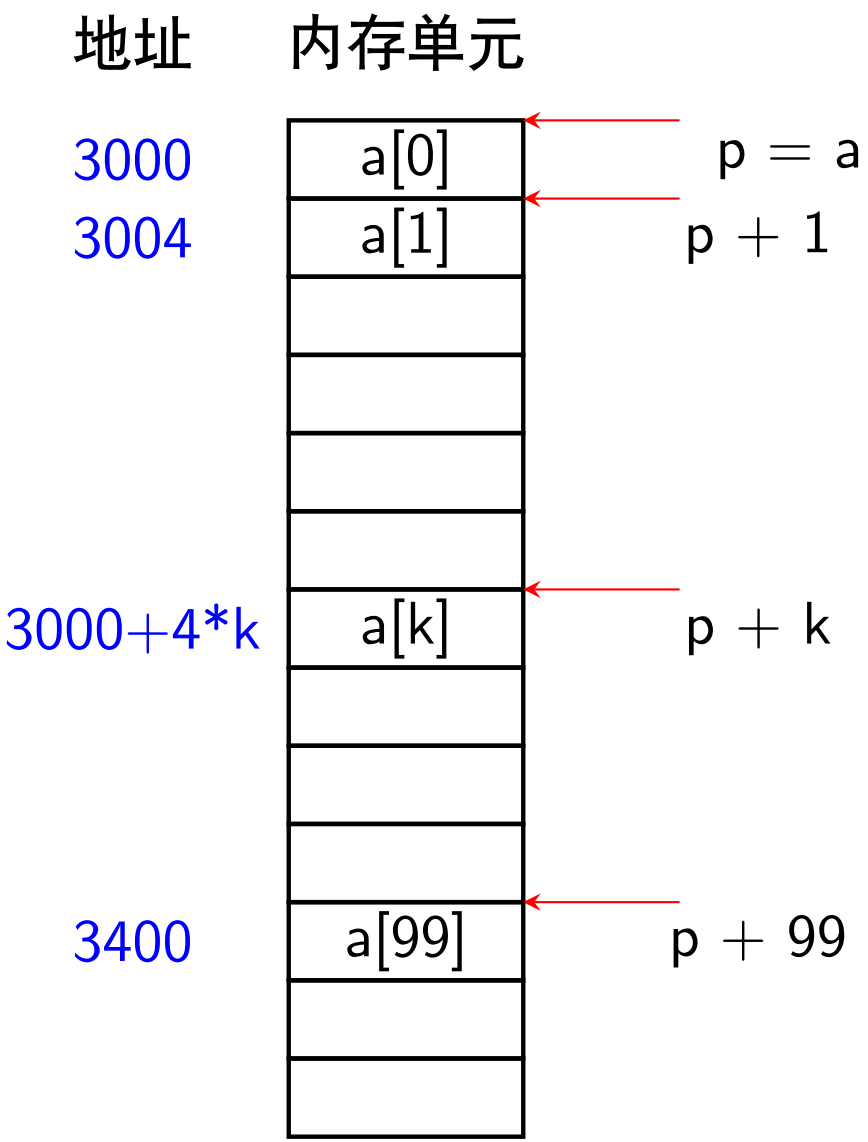


指针运算

- 指针加法：p + 整数表达式
  - 指针之后的若干元素
- 指针减法：p – 整数表达式
  - 指针之前的若干元素

指针移动

数组与指针和地址的关系

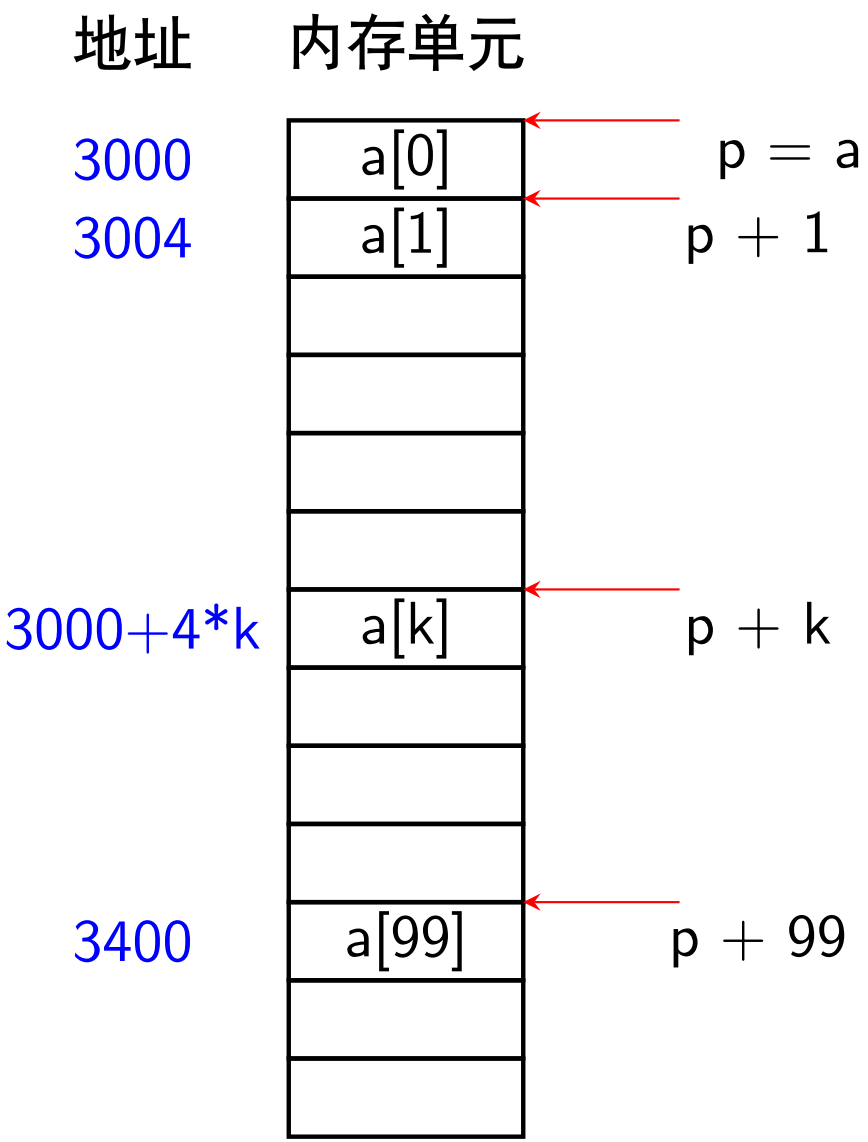


指针运算

- 指针加法：p + 整数表达式
  - 指针之后的若干元素
- 指针减法：p − 整数表达式
  - 指针之前的若干元素
- 指针相减：p − q

指针移动

数组与指针和地址的关系

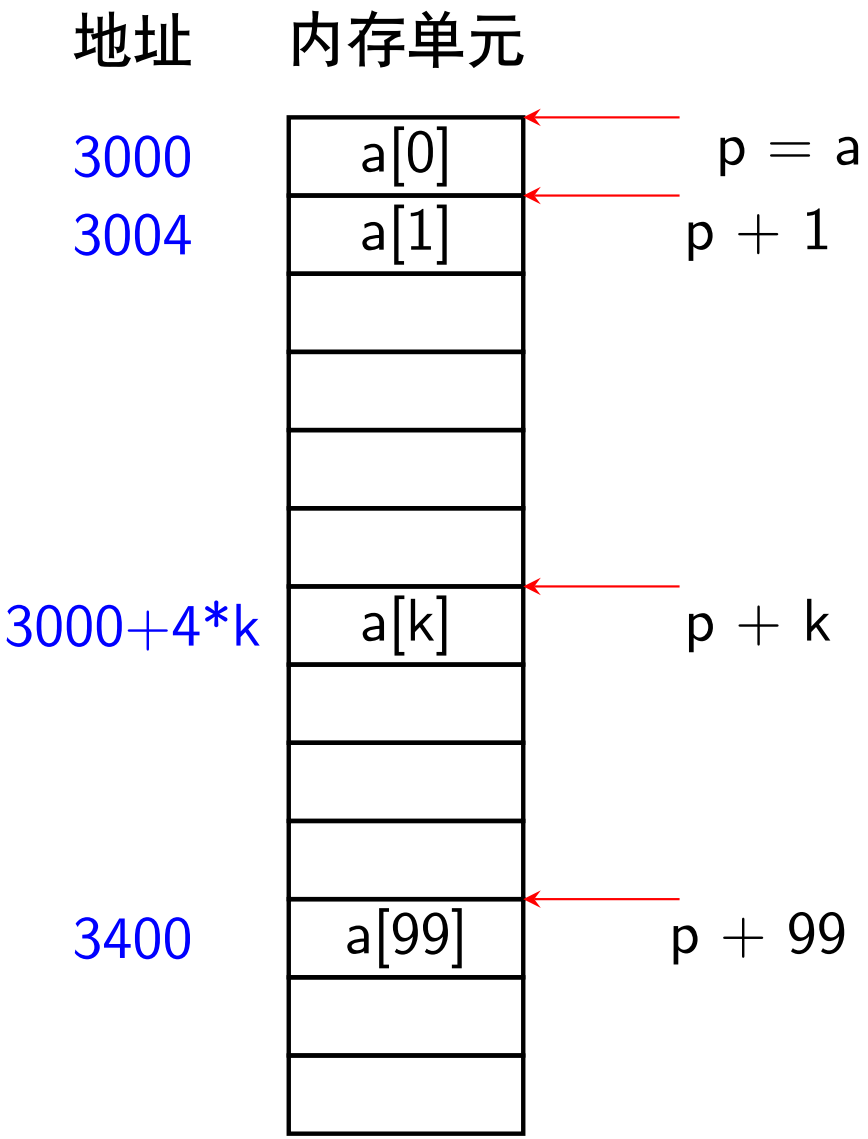


指针运算

- 指针加法：p + 整数表达式
  - 指针之后的若干元素
- 指针减法：p − 整数表达式
  - 指针之前的若干元素
- 指针相减：p − q
  - p 和 q 间元素个数

指针移动

数组与指针和地址的关系



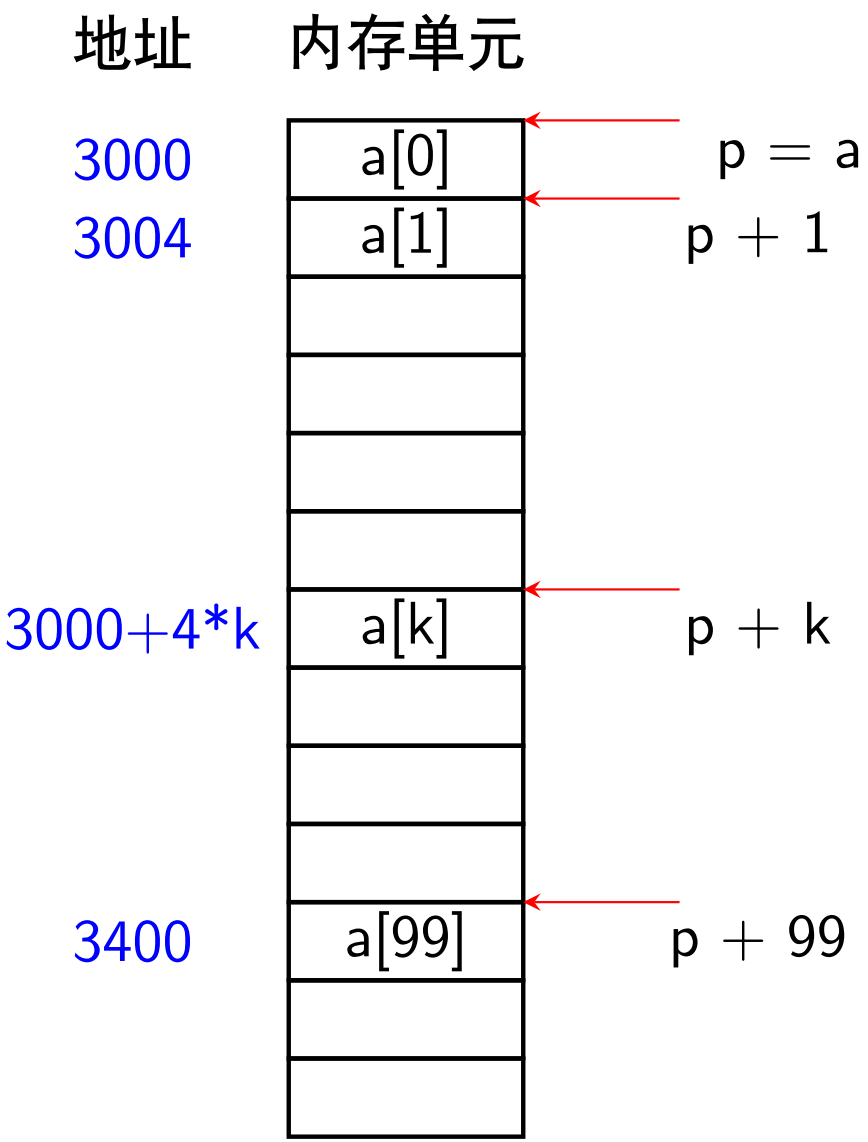
指针运算

- 指针加法：p + 整数表达式
  - 指针之后的若干元素
- 指针减法：p − 整数表达式
  - 指针之前的若干元素
- 指针相减：p − q
  - p 和 q 间元素个数

指针移动

- p = p + 整数表达式

数组与指针和地址的关系



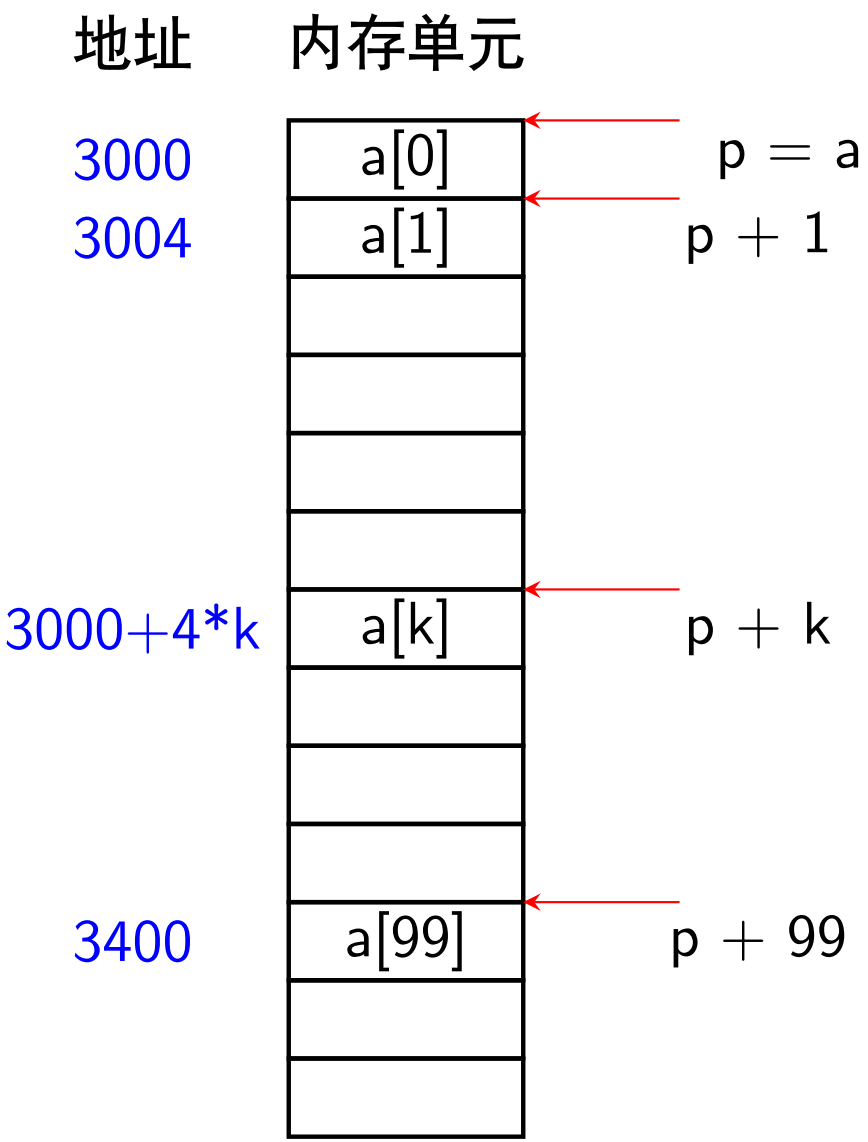
指针运算

- 指针加法：p + 整数表达式
  - 指针之后的若干元素
- 指针减法：p − 整数表达式
  - 指针之前的若干元素
- 指针相减：p − q
  - p 和 q 间元素个数

指针移动

- p = p + 整数表达式
  - +=, -=

数组与指针和地址的关系



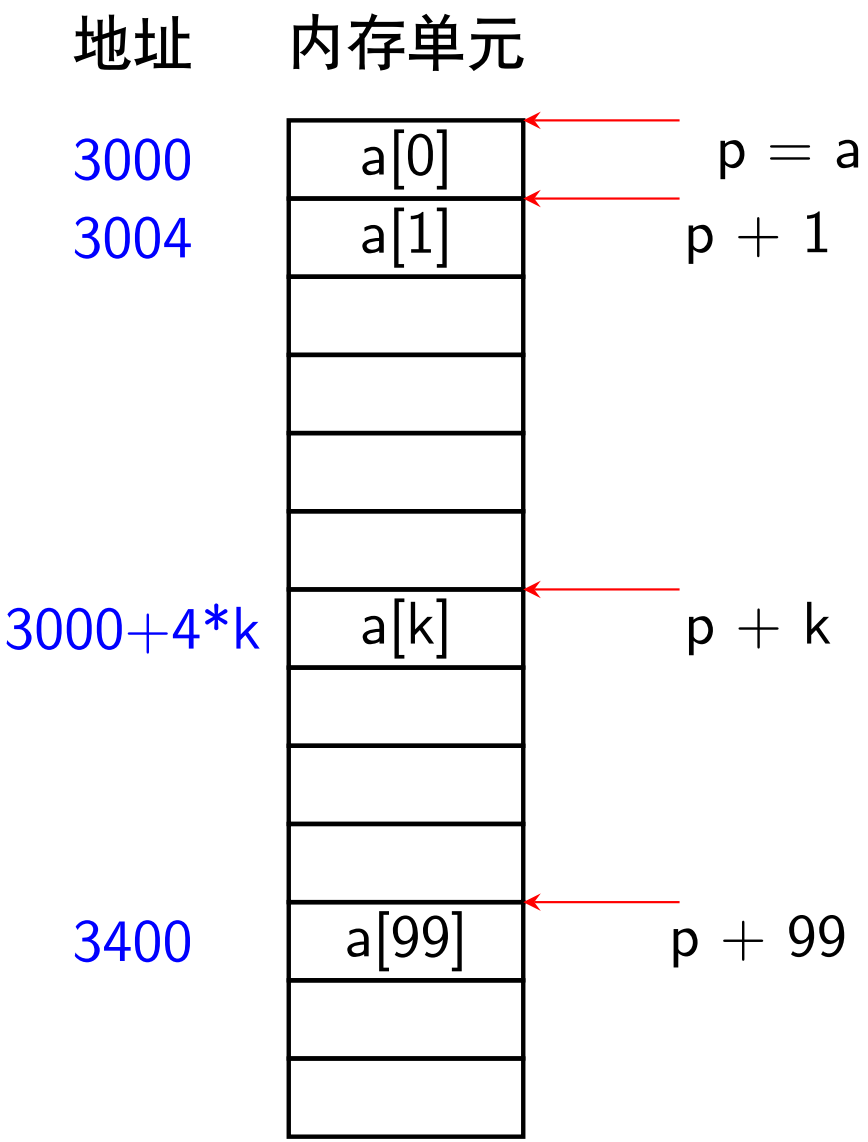
指针运算

- 指针加法：p + 整数表达式
  - 指针之后的若干元素
- 指针减法：p − 整数表达式
  - 指针之前的若干元素
- 指针相减：p − q
  - p 和 q 间元素个数

指针移动

- p = p + 整数表达式
  - +=, −=
- p −− 或者 −−p

数组与指针和地址的关系



指针运算

- 指针加法：p + 整数表达式
  - 指针之后的若干元素
- 指针减法：p − 整数表达式
  - 指针之前的若干元素
- 指针相减：p − q
  - p 和 q 间元素个数

指针移动

- p = p + 整数表达式
  - +=, −=
- p −− 或者 −−p
- p ++ 或者 ++p

数组与指针和地址的关系

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

数组与指针和地址的关系

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

指针相减

```
printf("%d", q - p);
```

数组与指针和地址的关系

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

指针相减

```
printf("%d", q - p);
```

输出为: 5
why？指针相减，二者间的元素个数

数组与指针和地址的关系

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

指针相减

```
printf("%d", q - p);
```

输出为: 5
why？指针相减，二者间的元素个数

指针的地址相减

```
printf("%d",(int)q-(int)p);
```

数组与指针和地址的关系

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

指针相减

```
printf("%d", q - p);
```

输出为: 5
why？指针相减，二者间的元素个数

指针的地址相减

```
printf("%d",(int)q-(int)p);
```

输出为: 20
why? 地址值的偏移量

数组与指针和地址的关系

指针与数组的关系

- a 与 &a[0] ———————— 地址相等

数组与指针和地址的关系

指针与数组的关系

- a 与 &a[0] ———— 地址相等
- a + k 与 &a[k] ———— 地址相等

数组与指针和地址的关系

指针与数组的关系

- a 与 &a[0] ———————— 地址相等
- a + k 与 &a[k] ———————— 地址相等
- *(a + k) 与 a[k]———————— 同一个变量/元素

数组与指针和地址的关系

指针与数组的关系

- a 与 &a[0] ———— 地址相等
- a + k 与 &a[k] ———— 地址相等
- *(a + k) 与 a[k]———— 同一个变量/元素
- &a 是什么？———— 一个指针，指向数组 a 的指针

数组与指针和地址的关系

指针与数组的关系

- a 与 &a[0] ———— 地址相等
- a + k 与 &a[k] ———— 地址相等
- *(a + k) 与 a[k]———— 同一个变量/元素
- &a 是什么？———— 一个指针，指向数组 a 的指针

数组与指针和地址的关系

指针与数组的关系

- a 与 &a[0] ———— 地址相等
- a + k 与 &a[k] ———— 地址相等
- *(a + k) 与 a[k]———— 同一个变量/元素
- &a 是什么？———— 一个指针，指向数组 a 的指针

思考下列代码输出 (假设数组 a 的地址是 c183cb50)

```
printf("%x\n", &a);
printf("%x %x\n",a,a+1);
printf("%x %x\n",&a[0],&a[1]);
```

数组与指针和地址的关系

指针与数组的关系

- a 与 &a[0] ———— 地址相等
- a + k 与 &a[k] ———— 地址相等
- *(a + k) 与 a[k]———— 同一个变量/元素
- &a 是什么？———— 一个指针，指向数组 a 的指针

思考下列代码输出 (假设数组 a 的地址是 c183cb50)

```
printf("%x\n", &a);
printf("%x %x\n",a,a+1);
printf("%x %x\n",&a[0],&a[1]);
```

输出为:

```
c183cb50
c183cb50 c183cb54
c183cb50 c183cb54
```

数组求和 – 例程 8-6

输入一个正整数 n (n<100), 再输入 n 个整数作为数组元素，分别使用数组和指针计算并输出数据元素的和。

首先输入数组元素

```
#include<stdio.h>
int main(void)
{
    int i, n, a[100], *p;
    long sum = 0;

    printf("输入一个整数 n(n<100): ");
    scanf("%d", &n);
    printf("输入 %d 个整数: ", n);
    for( i = 0; i<n; i++ )
        scanf("%d", &a[i]);
```

数组求和 – 例程 8-6

遍历数组元素，并求和

```
for(sum=0,i=0; i<n; i++)
    sum += *(a+i); //使用数组求和
    // *(a+i)与a[i]是等价的表达式
printf("数组求和结果 = %ld\n",sum);
```

通过指针，遍历数组元素，并求和

```
for(sum=0,p=a; p<a+n; p++)
    sum += *p; //使用指针求和
printf("指针求和结果 = %ld\n",sum);
```

- 指针比较：p < a + n（比较地址值的大小关系）

数组求和 – 例程 8-6

遍历数组元素，并求和

```
for(sum=0,i=0; i<n; i++)
    sum += *(a+i); //使用数组求和
    // *(a+i)与a[i]是等价的表达式
printf("数组求和结果 = %ld\n",sum);
```

通过指针，遍历数组元素，并求和

```
for(sum=0,p=a; p<a+n; p++)
    sum += *p; //使用指针求和
printf("指针求和结果 = %ld\n",sum);
```

- 指针比较：p < a + n（比较地址值的大小关系）
- 指针加法 a + n，指向数组 a 末尾元素的后面

数组求和 – 例程 8-6

遍历数组元素，并求和

```
for(sum=0,i=0; i<n; i++)
    sum += *(a+i); //使用数组求和
    // *(a+i)与a[i]是等价的表达式
printf("数组求和结果 = %ld\n",sum);
```

通过指针，遍历数组元素，并求和

```
for(sum=0,p=a; p<a+n; p++)
    sum += *p; //使用指针求和
printf("指针求和结果 = %ld\n",sum);
```

- 指针比较：p < a + n（比较地址值的大小关系）
- 指针加法 a + n，指向数组 a 末尾元素的后面
  - 刚刚超出了数组 a 的范围，常用在循环条件里

数组求和 – 例程 8-6

遍历数组元素，并求和

```
for(sum=0,i=0; i<n; i++)
    sum += *(a+i); //使用数组求和
    // *(a+i)与a[i]是等价的表达式
printf("数组求和结果 = %ld\n",sum);
```

通过指针，遍历数组元素，并求和

```
for(sum=0,p=a; p<a+n; p++)
    sum += *p; //使用指针求和
printf("指针求和结果 = %ld\n",sum);
```

- 指针比较：p < a + n（比较地址值的大小关系）
- 指针加法 a + n，指向数组 a 末尾元素的后面
  - 刚刚超出了数组 a 的范围，常用在循环条件里
- 或写成 p < &a[n]，或 p <= &a[n-1]

指针减法 – 例程 8-7

计算两个指针之间的元素个数和字节数

```
double a[100], *p, *q;

p = &a[0];
q = &a[20];
printf("p、q间元素的个数 = %ld\n", q - p);
printf("p、q间字节的个数 = %ld\n", (long)q
    - (long)p);
```

- 指针相减：q - p

指针减法 – 例程 8-7

计算两个指针之间的元素个数和字节数

```
double a[100], *p, *q;

p = &a[0];
q = &a[20];
printf("p、q间元素的个数 = %ld\n", q - p);
printf("p、q间字节的个数 = %ld\n", (long)q
    - (long)p);
```

- 指针相减：q - p
  - 结果是元素个数

指针减法 – 例程 8-7

计算两个指针之间的元素个数和字节数

```
double a[100], *p, *q;

p = &a[0];
q = &a[20];
printf("p、q间元素的个数 = %ld\n", q - p);
printf("p、q间字节的个数 = %ld\n", (long)q
    - (long)p);
```

- 指针相减：q - p
  - 结果是元素个数
- 指针地址值相减：(long)q - (long)p

指针减法 – 例程 8-7

计算两个指针之间的元素个数和字节数

```
double a[100], *p, *q;

p = &a[0];
q = &a[20];
printf("p、q间元素的个数 = %ld\n", q - p);
printf("p、q间字节的个数 = %ld\n", (long)q
    - (long)p);
```

- 指针相减：q - p
  - 结果是元素个数
- 指针地址值相减：(long)q - (long)p
  - (long) 是类型转换运算符

指针减法 – 例程 8-7

计算两个指针之间的元素个数和字节数

```
double a[100], *p, *q;

p = &a[0];
q = &a[20];
printf("p、q间元素的个数 = %ld\n", q - p);
printf("p、q间字节的个数 = %ld\n", (long)q
    - (long)p);
```

- 指针相减：q - p
  - 结果是元素个数
- 指针地址值相减：(long)q - (long)p
  - (long) 是类型转换运算符
  - 这里 (long) 将 p、q 从指针类型转化为 long

指针减法 – 例程 8-7

计算两个指针之间的元素个数和字节数

```
double a[100], *p, *q;

p = &a[0];
q = &a[20];
printf("p、q间元素的个数 = %ld\n", q - p);
printf("p、q间字节的个数 = %ld\n", (long)q
    - (long)p);
```

- 指针相减：q - p
  - 结果是元素个数
- 指针地址值相减：(long)q - (long)p
  - (long) 是类型转换运算符
  - 这里 (long) 将 p、q 从指针类型转化为 long
  - 转换结果为指针的地址值

数组名作为函数参数

```
int sum(int a[], int n)
/* int a[] 等价于int *a,
   定义一个指针类型的形式参数 */
{
    int i, s;
    for( s = 0, i = 0; i<n; i++ )
        s += a[i];
    return s;
}
```

数组名作为函数参数

```
int sum(int a[], int n)
/* int a[] 等价于int *a,
   定义一个指针类型的形式参数 */
{
    int i, s;
    for( s = 0, i = 0; i<n; i++ )
        s += a[i];
    return s;
}
```

数组作为实际参数

数组名作为函数参数

```
int sum(int a[], int n)
/* int a[] 等价于int *a,
   定义一个指针类型的形式参数 */
{
    int i, s;
    for( s = 0, i = 0; i<n; i++ )
        s += a[i];
    return s;
}
```

数组作为实际参数

假设定义数组：int b[100]; 那么：

- sum(b, 100)

数组名作为函数参数

```
int sum(int a[], int n)
/* int a[] 等价于int *a,
   定义一个指针类型的形式参数 */
{
    int i, s;
    for( s = 0, i = 0; i<n; i++ )
        s += a[i];
    return s;
}
```

数组作为实际参数

假设定义数组：int b[100]; 那么：

- sum(b, 100)　　求和 b[0] + b[1] + ... + b[99]

数组名作为函数参数

```
int sum(int a[], int n)
/* int a[] 等价于int *a,
   定义一个指针类型的形式参数 */
{
    int i, s;
    for( s = 0, i = 0; i<n; i++ )
        s += a[i];
    return s;
}
```

数组作为实际参数

假设定义数组：int b[100]; 那么：

- sum(b, 100)　　求和 b[0] + b[1] + ... + b[99]
- sum(b+10, 50)

数组名作为函数参数

```
int sum(int a[], int n)
/* int a[] 等价于int *a,
   定义一个指针类型的形式参数 */
{
    int i, s;
    for( s = 0, i = 0; i<n; i++ )
        s += a[i];
    return s;
}
```

数组作为实际参数

假设定义数组：int b[100]; **那么**：

- sum(b, 100)　　**求和** b[0] + b[1] + ... + b[99]
- sum(b+10, 50)　**求和** b[10] + b[11] + ... + b[59]

数组名作为函数参数

```
int sum(int a[], int n)
/* int a[] 等价于int *a,
   定义一个指针类型的形式参数 */
{
    int i, s;
    for( s = 0, i = 0; i<n; i++ )
        s += a[i];
    return s;
}
```

数组作为实际参数

假设定义数组：int b[100]; 那么：

- sum(b, 100)　　求和 b[0] + b[1] + ... + b[99]
- sum(b+10, 50)　求和 b[10] + b[11] + ... + b[59]

数组名既代表了数组变量本身，又可作为首元素的指针

内容提要

冒泡排序算法

回顾选择排序算法

```
for( k = 0; k<N-1; k++ ) {
    找到a[k], ..., a[N-1]中最小的元素
    将其交换到最前面 (即与a[k]交换)
}
```

冒泡排序算法

回顾选择排序算法

```
for( k = 0; k<N-1; k++ ) {
    找到a[k], ..., a[N-1]中最小的元素
    将其交换到最前面 (即与a[k]交换)
}
```

转换成 C 语言代码

```
for( k = 0; k<N-1; k++ ) {
    /*找a[k],...,a[N-1]中的最小元素 */
    for( m = k, i = k+1; i<N; i++ )
        if( a[i] < a[m] ) m = i;
            temp = a[m]; a[m] = a[i]; a[i] =
                temp; /*交换a[m]与a[i]交换 */
}
```

冒泡排序算法

冒泡排序算法的核心 – 冒泡，将大元素交换到后面

冒泡排序算法

冒泡排序算法的核心 – 冒泡，将大元素交换到后面

对数组 a[0], a[1], ..., a[N-1] 冒泡，步骤如下：

- 比较 a[0] 和 a[1]，若 a[0] 大，则交换 a[0] 和 a[1]

冒泡排序算法

冒泡排序算法的核心 – 冒泡，将大元素交换到后面

对数组 a[0], a[1], ..., a[N-1] 冒泡，步骤如下：

- 比较 a[0] 和 a[1]，若 a[0] 大，则交换 a[0] 和 a[1]
- 比较 a[1] 和 a[2]，若 a[1] 大，则交换 a[1] 和 a[2]

冒泡排序算法

冒泡排序算法的核心 – 冒泡，将大元素交换到后面

对数组 a[0], a[1], ..., a[N-1] 冒泡，步骤如下：

- 比较 a[0] 和 a[1]，若 a[0] 大，则交换 a[0] 和 a[1]
- 比较 a[1] 和 a[2]，若 a[1] 大，则交换 a[1] 和 a[2]
- ……

冒泡排序算法

冒泡排序算法的核心 – 冒泡，将大元素交换到后面

对数组 a[0], a[1], ..., a[N-1] 冒泡，步骤如下：

- 比较 a[0] 和 a[1]，若 a[0] 大，则交换 a[0] 和 a[1]
- 比较 a[1] 和 a[2]，若 a[1] 大，则交换 a[1] 和 a[2]
- ……
- 比较 a[k] 和 a[k+1]，若 a[k] 大，则交换 a[k] 和 a[k+1]

冒泡排序算法

冒泡排序算法的核心 – 冒泡，将大元素交换到后面

对数组 a[0], a[1], ..., a[N-1] 冒泡，步骤如下：

- 比较 a[0] 和 a[1]，若 a[0] 大，则交换 a[0] 和 a[1]
- 比较 a[1] 和 a[2]，若 a[1] 大，则交换 a[1] 和 a[2]
- ……
- 比较 a[k] 和 a[k+1]，若 a[k] 大，则交换 a[k] 和 a[k+1]
- ……

冒泡排序算法

冒泡排序算法的核心 – 冒泡，将大元素交换到后面

对数组 a[0], a[1], ..., a[N-1] 冒泡，步骤如下：

- 比较 a[0] 和 a[1]，若 a[0] 大，则交换 a[0] 和 a[1]
- 比较 a[1] 和 a[2]，若 a[1] 大，则交换 a[1] 和 a[2]
- ……
- 比较 a[k] 和 a[k+1]，若 a[k] 大，则交换 a[k] 和 a[k+1]
- ……
- 比较 a[N-2] 和 a[N-1]，若 a[N-2] 大，则交换 a[N-2] 和 a[N-1]

冒泡排序算法

冒泡排序算法的核心 – 冒泡，将大元素交换到后面

对数组 a[0], a[1], ..., a[N-1] 冒泡，步骤如下：

- 比较 a[0] 和 a[1]，若 a[0] 大，则交换 a[0] 和 a[1]
- 比较 a[1] 和 a[2]，若 a[1] 大，则交换 a[1] 和 a[2]
- ……
- 比较 a[k] 和 a[k+1]，若 a[k] 大，则交换 a[k] 和 a[k+1]
- ……
- 比较 a[N-2] 和 a[N-1]，若 a[N-2] 大，则交换 a[N-2] 和 a[N-1]
- 比较 a[0] 和 a[1]，若 a[0] 大，则交换 a[0] 和 a[1]

冒泡排序算法

冒泡排序算法的核心 – 冒泡，将大元素交换到后面

对数组 a[0], a[1], ..., a[N-1] 冒泡，步骤如下：

```
for( k = 0; k<N-1; k++ ) {
    if( a[k]>a[k+1] )
        交换a[k]和a[k+1]
}
```

冒泡排序算法

冒泡排序过程

- 对 a[0], . . ., a[N-1] 冒泡（**最大元素被交换到** a[N-1]）

冒泡排序算法

冒泡排序过程

- 对 a[0], . . ., a[N-1] 冒泡（最大元素被交换到 a[N-1]）
- 对 a[0], . . ., a[N-2] 冒泡（最大元素被交换到 a[N-2]）

冒泡排序算法

冒泡排序过程

- 对 a[0], . . ., a[N-1] 冒泡（最大元素被交换到 a[N-1]）
- 对 a[0], . . ., a[N-2] 冒泡（最大元素被交换到 a[N-2]）

. . . . . .

冒泡排序算法

冒泡排序过程

- 对 a[0], . . ., a[N-1] 冒泡（最大元素被交换到 a[N-1]）
- 对 a[0], . . ., a[N-2] 冒泡（最大元素被交换到 a[N-2]）

  . . ..
- 对 a[0], . . ., a[N-k] 冒泡（最大元素被交换到 a[N-k]）

冒泡排序算法

冒泡排序过程

- 对 a[0], . . ., a[N-1] 冒泡（最大元素被交换到 a[N-1]）
- 对 a[0], . . ., a[N-2] 冒泡（最大元素被交换到 a[N-2]）

  . . ..
- 对 a[0], . . ., a[N-k] 冒泡（最大元素被交换到 a[N-k]）

  . . . . . .

冒泡排序算法

冒泡排序过程

- 对 a[0], . . ., a[N-1] 冒泡（最大元素被交换到 a[N-1]）
- 对 a[0], . . ., a[N-2] 冒泡（最大元素被交换到 a[N-2]）

  . . ..
- 对 a[0], . . ., a[N-k] 冒泡（最大元素被交换到 a[N-k]）

  . . ..
- 对 a[0], a[1] 冒泡（最大元素被交换到 a[1]）

冒泡排序算法

冒泡排序过程

- 对 a[0], . . ., a[N-1] 冒泡（最大元素被交换到 a[N-1]）
- 对 a[0], . . ., a[N-2] 冒泡（最大元素被交换到 a[N-2]）

  . . ..
- 对 a[0], . . ., a[N-k] 冒泡（最大元素被交换到 a[N-k]）

  . . ..
- 对 a[0], a[1] 冒泡（最大元素被交换到 a[1]）

冒泡排序算法

冒泡排序过程

- 对 a[0], . . ., a[N-1] 冒泡（最大元素被交换到 a[N-1]）
- 对 a[0], . . ., a[N-2] 冒泡（最大元素被交换到 a[N-2]）

  . . ..
- 对 a[0], . . ., a[N-k] 冒泡（最大元素被交换到 a[N-k]）

  . . ..
- 对 a[0], a[1] 冒泡（最大元素被交换到 a[1]）

冒泡排序算法伪代码

```
for( j = 0; j<N-1; j++ ) {
    对a[0], ..., a[N-j]进行冒泡
}
```

冒泡排序算法

冒泡排序算法伪代码

```
for( j = 0; j<N-1; j++ ) {
    对a[0], ..., a[N-j]进行冒泡
}
```

冒泡排序算法伪代码 – 展开冒泡操作

```
for( j = 0; j<N-1; j++ ) {
    // 对a[0], ..., a[N-j]进行冒泡
    for( k = 0; k<N-j; k++ ) {
        if( a[k]>a[k+1] )
            交换a[k]和a[k+1];
    }
}
```

冒泡排序算法

冒泡排序函数 bubble

```
// a - 待排序数组首元素指针
// N - 待排序元素个数
void bubble(int a[], int N)
{
    int j, k, temp;
    for( j = 1; j<N; j++ ) {
        // 对a[0], ..., a[N-j]进行冒泡
        for( k = 0; k<N-j; k++ )
            if( a[k]>a[k+1] ) {
                temp = a[k];
                a[k] = a[k+1];
                a[k+1] = temp;
            }
    }
}
```

冒泡排序算法

计算量分析

- 对 a[0], . . ., a[N-1] 冒泡，需要N - 1 次比较

冒泡排序算法

计算量分析

- 对 a[0], . . ., a[N-1] 冒泡，需要N - 1 次比较
- 对 a[0], . . ., a[N-2] 冒泡，需要N - 2 次比较

冒泡排序算法

计算量分析

- 对 a[0], . . ., a[N-1] 冒泡，需要N - 1 次比较
- 对 a[0], . . ., a[N-2] 冒泡，需要N - 2 次比较

. . . . . .

冒泡排序算法

计算量分析

- 对 a[0], . . ., a[N-1] 冒泡，需要N - 1 次比较
- 对 a[0], . . ., a[N-2] 冒泡，需要N - 2 次比较

  . . . . . .
- 对 a[0], . . ., a[N-k] 冒泡，需要N - k 次比较

冒泡排序算法

计算量分析

- 对 a[0], . . ., a[N-1] 冒泡，需要N - 1 次比较
- 对 a[0], . . ., a[N-2] 冒泡，需要N - 2 次比较

  . . . . . .
- 对 a[0], . . ., a[N-k] 冒泡，需要N - k 次比较

  . . . . . .

冒泡排序算法

计算量分析

- 对 a[0], . . ., a[N-1] 冒泡，需要N - 1 次比较
- 对 a[0], . . ., a[N-2] 冒泡，需要N - 2 次比较

  . . . . . .
- 对 a[0], . . ., a[N-k] 冒泡，需要N - k 次比较

  . . . . . .
- 对 a[0], a[1] 冒泡，需要1 次比较

冒泡排序算法

计算量分析

- 对 a[0], . . ., a[N-1] 冒泡，需要N - 1 次比较
- 对 a[0], . . ., a[N-2] 冒泡，需要N - 2 次比较

  . . . . . .
- 对 a[0], . . ., a[N-k] 冒泡，需要N - k 次比较

  . . . . . .
- 对 a[0], a[1] 冒泡，需要1 次比较

冒泡排序算法

计算量分析

- 对 a[0], . . ., a[N-1] 冒泡，需要N - 1 次比较
- 对 a[0], . . ., a[N-2] 冒泡，需要N - 2 次比较

. . . . . .

- 对 a[0], . . ., a[N-k] 冒泡，需要N - k 次比较

. . . . . .

- 对 a[0], a[1] 冒泡，需要1 次比较

比较次数共计

$$(N-1)+(N-2)+\cdots+2+1=N(N-1)/2\sim O(N^2)$$

内容提要

线性查找算法

线性查找 – 逐个比对，直到找到或者数组结束

```
int find(int a[], int n, int x)
{
    int k;
    for( k = 0; k<n; k++ )
        if( a[k]==x ) return k;
    return -1;
}
```

线性查找算法

线性查找 – 逐个比对，直到找到或者数组结束

```
int find(int a[], int n, int x)
{
    int k;
    for( k = 0; k<n; k++ )
        if( a[k]==x ) return k;
    return -1;
}
```

- 算法的平均复杂度为 O(n)

线性查找算法

线性查找 – 逐个比对，直到找到或者数组结束

```
int find(int a[], int n, int x)
{
    int k;
    for( k = 0; k<n; k++ )
        if( a[k]==x ) return k;
    return -1;
}
```

- 算法的平均复杂度为 O(n)
- 最好的情况，只要比对 1 次

线性查找算法

线性查找 – 逐个比对，直到找到或者数组结束

```
int find(int a[], int n, int x)
{
    int k;
    for( k = 0; k<n; k++ )
        if( a[k]==x ) return k;
    return -1;
}
```

- 算法的平均复杂度为 O(n)
- 最好的情况，只要比对 1 次
- 最坏的情况，需要比对 n 次

线性查找算法

线性查找 – 逐个比对，直到找到或者数组结束

```
int find(int a[], int n, int x)
{
    int k;
    for( k = 0; k<n; k++ )
        if( a[k]==x ) return k;
    return -1;
}
```

- 算法的平均复杂度为 O(n)
- 最好的情况，只要比对 1 次
- 最坏的情况，需要比对 n 次
- 如果需要查找的元素在每个位置出现的概率是一样的，那么平均需要比对 n/2 次

二分查找算法

通过更好地组织数据（排序），实现高效率的二分查找算法

```
int binary_search(int a[], int n, int x)
{   /* a[]具有从小到大顺序 */

    初始化搜索区间为[0, n-1]
    while( 搜索区间不为空 ) {
        查看搜索区间中间元素
        if( 中间元素==x )return 该元素index;
        if( 中间元素<x ) 搜索区间缩为 右 半边
        else             搜索区间缩为 左 半边
    }
    return (没找到);
}
```

二分查找算法

通过更好地组织数据（排序），实现高效率的二分查找算法

```
int binary_search(int a[], int n, int x)
{   /* a[]具有从小到大顺序 */

    int low = 0, high = n-1, mid;
    while( low <= high ) {
        mid = (low+high)/2;
        if( a[mid] == x )      return mid;
        else if( a[mid] < x ) low = mid + 1;
        else                  high = mid - 1;
    }
    return (-1);
}
```

二分查找算法复杂度

每执行一次比较，将搜索空间降为一半

算法的复杂度为 $log_2 n$

二分查找算法复杂度

每执行一次比较，将搜索空间降为一半

算法的复杂度为 $log_2 n$

- 最好的情况，只要比对 1 次

二分查找算法复杂度

每执行一次比较，将搜索空间降为一半

算法的复杂度为 $log_2 n$

- 最好的情况，只要比对 1 次
- 最坏的情况，需要比对 $log_2 n$

二分查找算法复杂度

每执行一次比较，将搜索空间降为一半

算法的复杂度为 $log_2 n$

- 最好的情况，只要比对 1 次
- 最坏的情况，需要比对 $log_2 n$

查找算法

二分查找算法复杂度

每执行一次比较，将搜索空间降为一半

算法的复杂度为 $log_2 n$

- 最好的情况，只要比对 1 次
- 最坏的情况，需要比对 $log_2 n$

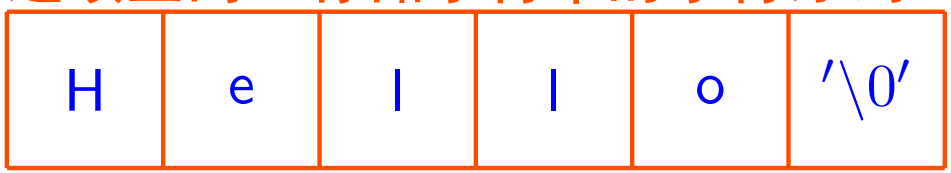
二分查找算法说明：合理组织数据的重要性!!!

内容提要

字符串的存储

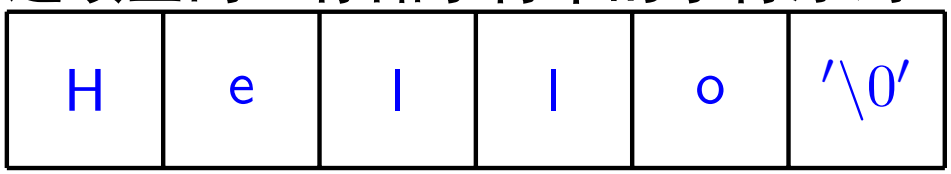
字符串的存储

- 连续空间、存储字符串的字符序列



字符串的存储

- 连续空间、存储字符串的字符序列

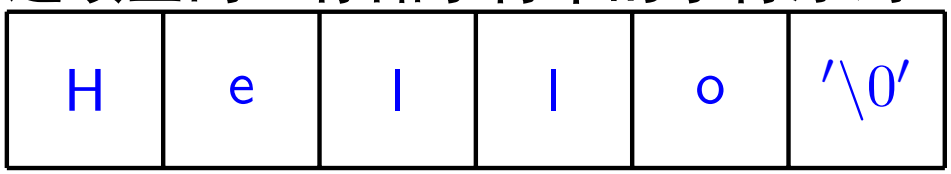


- 末尾必须添加一个 $'\backslash 0'$，作为结束标志字符

字符串和指针

字符串的存储

- 连续空间、存储字符串的字符序列



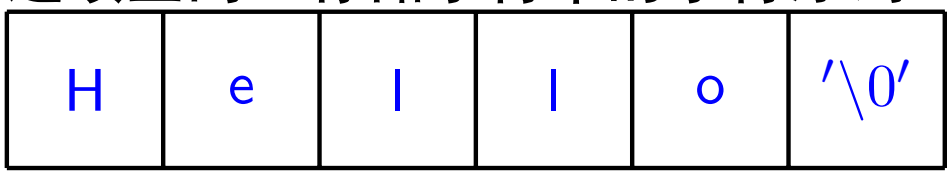
- 末尾必须添加一个 $'\backslash 0'$，作为结束标志字符

字符串的表示

字符串和指针

字符串的存储

- 连续空间、存储字符串的字符序列



- 末尾必须添加一个 $'\backslash 0'$，作为结束标志字符

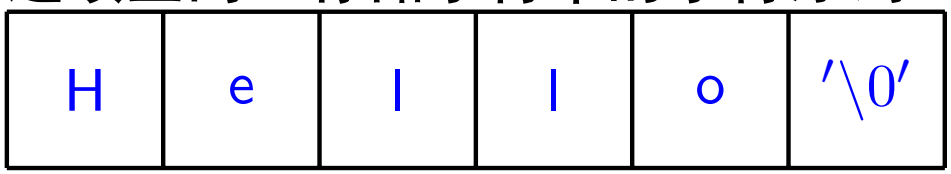
字符串的表示

- 因此，表示一个字符串，只需给出指向字符串首字符的指针

字符串和指针

字符串的存储

- 连续空间、存储字符串的字符序列



- 末尾必须添加一个 $'\backslash 0'$，作为结束标志字符

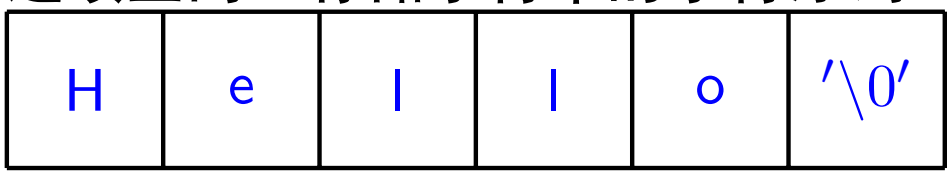
字符串的表示

- 因此，表示一个字符串，只需给出指向字符串首字符的指针
- 字符串在 C 语言里的类型是：char * 类型

字符串和指针

字符串的存储

- 连续空间、存储字符串的字符序列



- 末尾必须添加一个 $'\backslash 0'$，作为结束标志字符

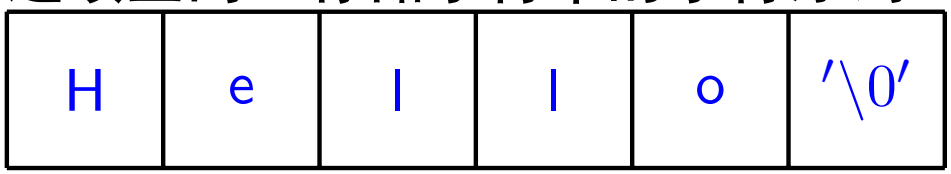
字符串的表示

- 因此，表示一个字符串，只需给出指向字符串首字符的指针
- 字符串在 C 语言里的类型是：char * 类型
- 字符串常量，例如"Hello Mike"，是一个指向字符串首字符的字符指针常量

字符串和指针

字符串的存储

- 连续空间、存储字符串的字符序列



- 末尾必须添加一个 $'\backslash 0'$，作为结束标志字符

字符串的表示

- 因此，表示一个字符串，只需给出指向字符串首字符的指针
- 字符串在 C 语言里的类型是：char * 类型
- 字符串常量，例如"Hello Mike"，是一个指向字符串首字符的字符指针常量
  - 所谓指针常量，其存储的地址值不可更改

字符串举例

字符串举例

```
/* 定义字符数组，初始化为字符串常量 */
char sa[] = "array";
/* 定义字符指针，保存字符串常量 */
char *sp = "point";
```

字符串举例

```
/* 定义字符数组，初始化为字符串常量 */
char sa[] = "array";
/* 定义字符指针，保存字符串常量 */
char *sp = "point";

/* 输出字符串 */
printf("%s\n", sa); //数组名sa作为字符串指针
printf("%s\n", sp);
printf("%s\n", "string"); //字符串常量就是它首
    字符指针
```

字符串举例

```
/* 定义字符数组，初始化为字符串常量 */
char sa[] = "array";
/* 定义字符指针，保存字符串常量 */
char *sp = "point";

/* 输出字符串 */
printf("%s\n", sa); //数组名sa作为字符串指针
printf("%s\n", sp);
printf("%s\n", "string"); //字符串常量就是它首
    字符指针
```

- **输出结果为**：array point string

字符串举例

```
/* 定义字符数组，初始化为字符串常量 */
char sa[] = "array";
/* 定义字符指针，保存字符串常量 */
char *sp = "point";

/* 输出字符串 */
printf("%s\n", sa+2);
printf("%s\n", sp+3);
printf("%s\n", "string" + 1); //"string" + 1
    是一个字符指针，指向字符 't'
```

字符串举例

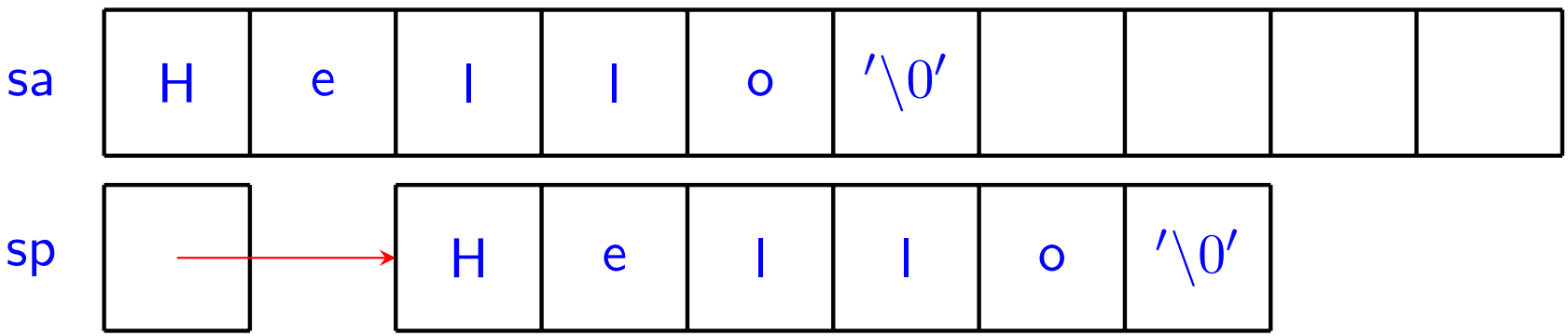
```
/* 定义字符数组，初始化为字符串常量 */
char sa[] = "array";
/* 定义字符指针，保存字符串常量 */
char *sp = "point";

/* 输出字符串 */
printf("%s\n", sa+2);
printf("%s\n", sp+3);
printf("%s\n", "string" + 1); //"string" + 1
    是一个字符指针，指向字符 't'
```

- **输出结果为**：ray nt tring

用指针和数组处理字符串的区别

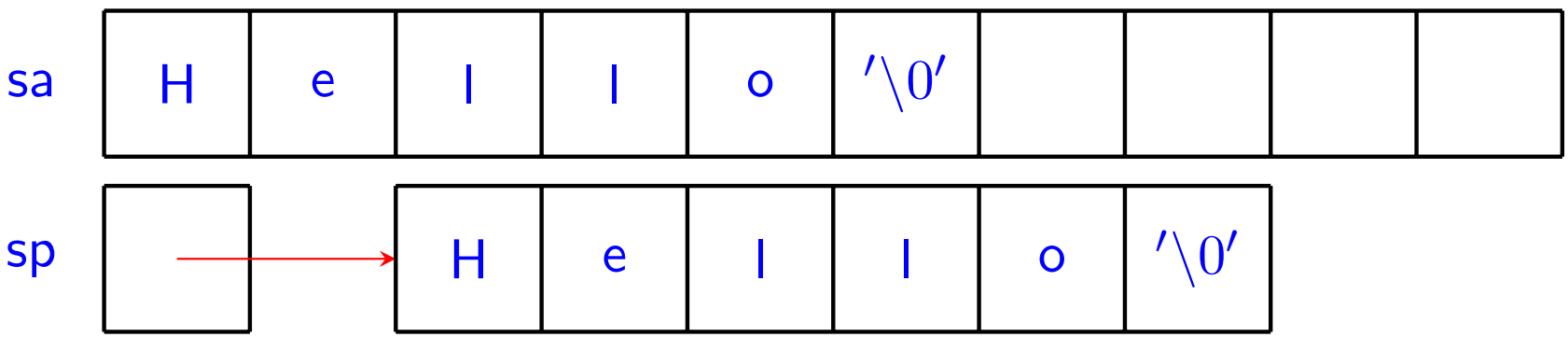
```
/* 定义字符数组 */
char sa[10] = "Hello";
/* 定义字符指针 */
char *sp = "Hello";
```



- 数组 sa 中的字符串内容可以修改

用指针和数组处理字符串的区别

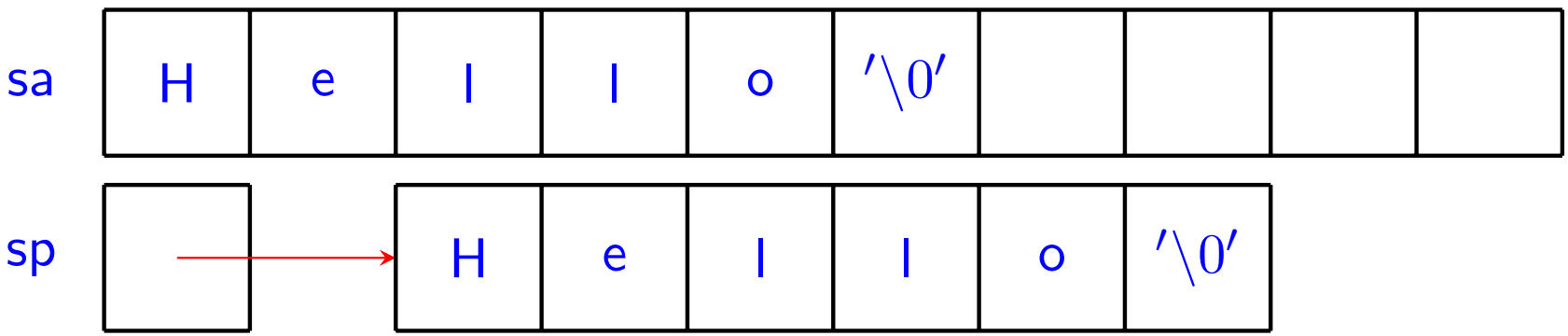
```
/* 定义字符数组 */
char sa[10] = "Hello";
/* 定义字符指针 */
char *sp = "Hello";
```



- 数组 sa 中的字符串内容可以修改
  - sa[0] = 'h';

用指针和数组处理字符串的区别

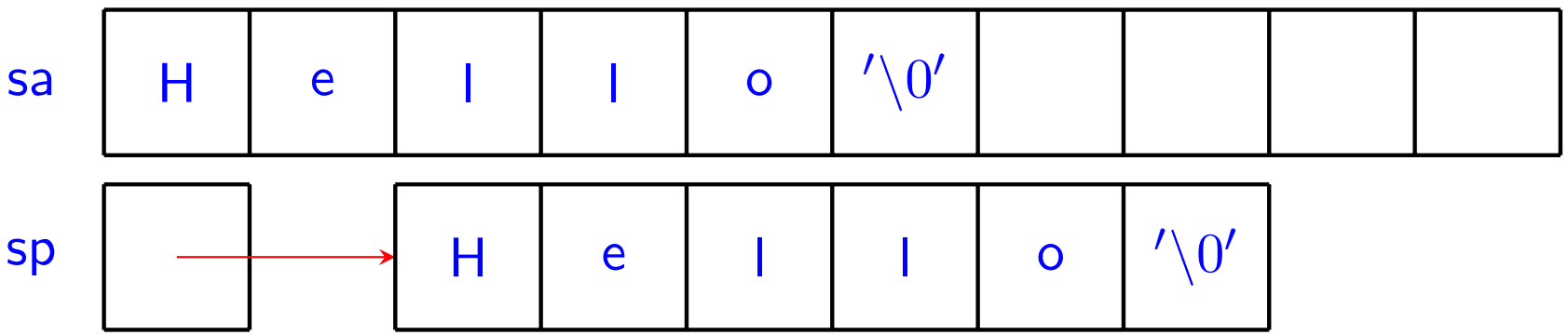
```
/* 定义字符数组 */
char sa[10] = "Hello";
/* 定义字符指针 */
char *sp = "Hello";
```



- 数组 sa 中的字符串内容可以修改
  - sa[0] = 'h';
- 字符串 sp 中的内容不可修改，该字符串是常量

用指针和数组处理字符串的区别

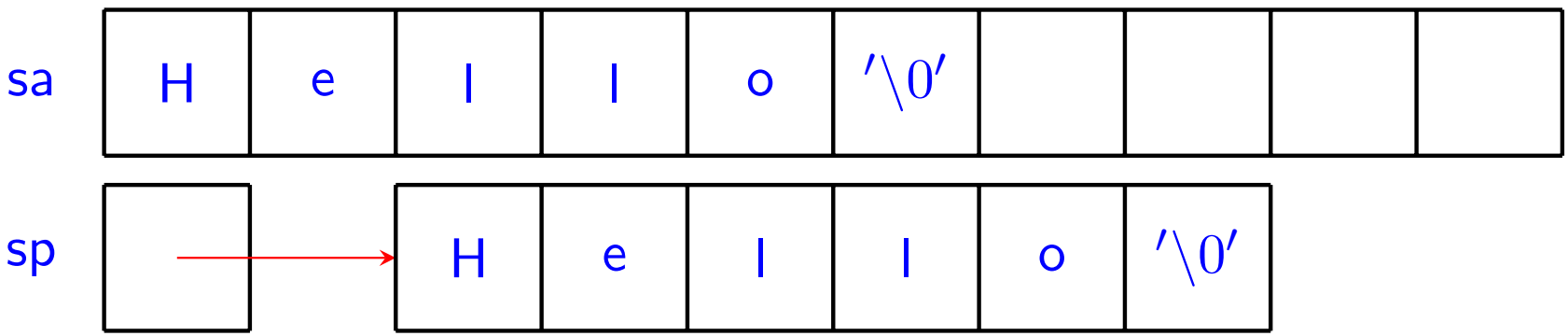
```
/* 定义字符数组 */
char sa[10] = "Hello";
/* 定义字符指针 */
char *sp = "Hello";
```



- 数组 sa 中的字符串内容可以修改
  - sa[0] = 'h';
- 字符串 sp 中的内容不可修改，该字符串是常量
  - 但是可以修改指针 sp（的地址值），使其指向别的字符串

用指针和数组处理字符串的区别

```
/* 定义字符数组 */
char sa[10] = "Hello";
/* 定义字符指针 */
char *sp = "Hello";
```



- 数组 sa 中的字符串内容可以修改
  - sa[0] = 'h';
- 字符串 sp 中的内容不可修改，该字符串是常量
  - 但是可以修改指针 sp（的地址值），使其指向别的字符串
    sp = "World";

字符串处理函数

- scanf/printf 输入输出

字符串处理函数

- scanf/printf 输入输出
- gets/puts 输入输出

字符串处理函数

- scanf/printf 输入输出
- gets/puts 输入输出
- strcpy 字符串拷贝复制

字符串处理函数

- scanf/printf 输入输出
- gets/puts 输入输出
- strcpy 字符串拷贝复制
- strcat 字符串连接

字符串处理函数

- scanf/printf 输入输出
- gets/puts 输入输出
- strcpy 字符串拷贝复制
- strcat 字符串连接
- strcmp 字符串比较

字符串处理函数

- scanf/printf 输入输出
- gets/puts 输入输出
- strcpy 字符串拷贝复制
- strcat 字符串连接
- strcmp 字符串比较
- strlen 求字符串长度

字符串处理函数

- scanf/printf 输入输出
- gets/puts 输入输出
- strcpy 字符串拷贝复制
- strcat 字符串连接
- strcmp 字符串比较
- strlen 求字符串长度

字符串处理函数

- scanf/printf 输入输出
- gets/puts 输入输出
- strcpy 字符串拷贝复制
- strcat 字符串连接
- strcmp 字符串比较
- strlen 求字符串长度

需要包含相应的头文件

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

字符串处理函数

用 scanf 输入字符串

字符串处理函数

用 scanf 输入字符串

- scanf("%s", p);

字符串处理函数

用 scanf 输入字符串

- scanf(”%s”, p);

字符串处理函数

用 scanf 输入字符串

- scanf("%s", p);
  - 要求 p 是一个字符数组，或者字符指针并且指向了实际内存单元

字符串处理函数

用 scanf 输入字符串

- scanf("%s", p);
  - 要求 p 是一个字符数组，或者字符指针并且指向了实际内存单元
  - 碰到空格、制表符、回车停止读入

字符串处理函数

用 scanf 输入字符串

- scanf("%s", p);
  - 要求 p 是一个字符数组，或者字符指针并且指向了实际内存单元
  - 碰到空格、制表符、回车停止读入
  - 末尾添加'\0'，作为字符串结束标志

字符串处理函数

用 scanf 输入字符串

- scanf("%s", p);
  - 要求 p 是一个字符数组，或者字符指针并且指向了实际内存单元
  - 碰到空格、制表符、回车停止读入
  - 末尾添加'\0'，作为字符串结束标志

用 printf 输出字符串

字符串处理函数

用 scanf 输入字符串

- scanf("%s", p);
  - 要求 p 是一个字符数组，或者字符指针并且指向了实际内存单元
  - 碰到空格、制表符、回车停止读入
  - 末尾添加'\0'，作为字符串结束标志

用 printf 输出字符串

- printf("%s", p);

字符串处理函数

用 scanf 输入字符串

- scanf("%s", p);
  - 要求 p 是一个字符数组，或者字符指针并且指向了实际内存单元
  - 碰到空格、制表符、回车停止读入
  - 末尾添加'\0'，作为字符串结束标志

用 printf 输出字符串

- printf("%s", p);
  - p 可以是字符数组，或者字符指针

字符串处理函数

用 scanf 输入字符串

- scanf("%s", p);
  - 要求 p 是一个字符数组，或者字符指针并且指向了实际内存单元
  - 碰到空格、制表符、回车停止读入
  - 末尾添加'\0'，作为字符串结束标志

用 printf 输出字符串

- printf("%s", p);
  - p 可以是字符数组，或者字符指针
  - 都要求 p 指向了实际字符串

字符串处理函数

用 scanf 输入字符串

- scanf("%s", p);
  - 要求 p 是一个字符数组，或者字符指针并且指向了实际内存单元
  - 碰到空格、制表符、回车停止读入
  - 末尾添加'\0'，作为字符串结束标志

用 printf 输出字符串

- printf("%s", p);
  - p 可以是字符数组，或者字符指针
  - 都要求 p 指向了实际字符串
  - 并且以'\0' 结尾

字符串处理函数

printf 函数与字符指针

```
char * p = "Hello";
```

字符串处理函数

printf 函数与字符指针

char * p = "Hello";

- printf("%s", p)：输出指针所指向的字符串

字符串处理函数

printf 函数与字符指针

char * p = "Hello";

- printf("%s", p)：输出指针所指向的字符串
- printf("%d", p)：输出指针的地址值（作为整数）

字符串处理函数

printf 函数与字符指针

char * p = "Hello";

- printf("%s", p): 输出指针所指向的字符串
- printf("%d", p): 输出指针的地址值（作为整数）
- printf("%u", p): 输出指针的地址值（作为无符号整数）

字符串处理函数

printf 函数与字符指针

char * p = "Hello";

- printf("%s", p): 输出指针所指向的字符串
- printf("%d", p): 输出指针的地址值（作为整数）
- printf("%u", p): 输出指针的地址值（作为无符号整数）
- printf("%x", p): 输出指针的地址值（16 进制）

字符串处理函数

printf 函数与字符指针

char * p = "Hello";

- printf("%s", p): 输出指针所指向的字符串
- printf("%d", p): 输出指针的地址值（作为整数）
- printf("%u", p): 输出指针的地址值（作为无符号整数）
- printf("%x", p): 输出指针的地址值（16 进制）

scanf 函数与字符指针

char sa[100], *p, *p1 = sa, *p2 = "Hello";

字符串处理函数

printf 函数与字符指针

char * p = "Hello";

- printf("%s", p): 输出指针所指向的字符串
- printf("%d", p): 输出指针的地址值（作为整数）
- printf("%u", p): 输出指针的地址值（作为无符号整数）
- printf("%x", p): 输出指针的地址值（16 进制）

scanf 函数与字符指针

char sa[100], *p, *p1 = sa, *p2 = "Hello";

- printf("%d", p1); 读入一个字符串

字符串处理函数

printf 函数与字符指针

char * p = "Hello";

- printf("%s", p): 输出指针所指向的字符串
- printf("%d", p): 输出指针的地址值（作为整数）
- printf("%u", p): 输出指针的地址值（作为无符号整数）
- printf("%x", p): 输出指针的地址值（16 进制）

scanf 函数与字符指针

char sa[100], *p, *p1 = sa, *p2 = "Hello";

- printf("%d", p1); 读入一个字符串
- scanf("%s", p2); 错!，因为 p2 指向的是字符串常量

字符串处理函数

printf 函数与字符指针

char * p = "Hello";

- printf("%s", p): 输出指针所指向的字符串
- printf("%d", p): 输出指针的地址值（作为整数）
- printf("%u", p): 输出指针的地址值（作为无符号整数）
- printf("%x", p): 输出指针的地址值（16 进制）

scanf 函数与字符指针

char sa[100], *p, *p1 = sa, *p2 = "Hello";

- printf("%d", p1); 读入一个字符串
- scanf("%s", p2); 错!，因为 p2 指向的是字符串常量
- printf("%u", p); 不可预测的输出!

字符串处理函数

printf 函数与字符指针

char * p = "Hello";

- printf("%s", p): 输出指针所指向的字符串
- printf("%d", p): 输出指针的地址值（作为整数）
- printf("%u", p): 输出指针的地址值（作为无符号整数）
- printf("%x", p): 输出指针的地址值（16 进制）

scanf 函数与字符指针

char sa[100], *p, *p1 = sa, *p2 = "Hello";

- printf("%d", p1); 读入一个字符串
- scanf("%s", p2); 错!，因为 p2 指向的是字符串常量
- printf("%u", p); 不可预测的输出!

因为 p 没有指向实际的内存空间

字符串指针的常见错误

指针没有指向实际内存单元

```
char *s;
scanf("%s", s);
```

字符串指针的常见错误

指针没有指向实际内存单元

```
char *s;
scanf("%s", s);
```

字符串指针的常见错误

指针没有指向实际内存单元

```
char *s;
scanf("%s", s);
```

可做如下修正

```
char a[100], *s=a;
scanf("%s", s);
scanf("%s", a);
```

字符串处理函数 – 练习

下面哪一句会有问题？

```
int  n;
char a[100];
char s, *p;
scanf("%d",  n);
scanf("%d", &n);
scanf("%s",  a);
scanf("%s", &a);
scanf("%s", &s);
scanf("%s", p);
p = a;
scanf("%s", p);
```

gets/puts – 字符串输入/输出函数

```
char * gets( char * s ); /* 读入字符串 */
```

- s 指向一段可写入字符串的内存的起始地址

gets/puts – 字符串输入/输出函数

```
char * gets( char * s ); /* 读入字符串 */
```

- s 指向一段可写入字符串的内存的起始地址

  s 通常是一个字符数组，或者指向了一个字符数组首元素

gets/puts – 字符串输入/输出函数

```
char * gets( char * s ); /* 读入字符串 */
```

- s 指向一段可写入字符串的内存的起始地址

  s 通常是一个字符数组，或者指向了一个字符数组首元素
- 返回值为：指针 s

gets/puts – 字符串输入/输出函数

```
char * gets( char * s ); /* 读入字符串 */
```

- s 指向一段可写入字符串的内存的起始地址

  s 通常是一个字符数组，或者指向了一个字符数组首元素
- 返回值为：指针 s

gets/puts – 字符串输入/输出函数

```
char * gets( char * s ); /* 读入字符串 */
```

- s 指向一段可写入字符串的内存的起始地址
  s 通常是一个字符数组，或者指向了一个字符数组首元素
- 返回值为：指针 s

```
int puts( char * s );     /* 输出字符串 */
```

- s 为指向字符串的指针

gets/puts – 字符串输入/输出函数

```
char * gets( char * s ); /* 读入字符串 */
```

- s 指向一段可写入字符串的内存的起始地址
  s 通常是一个字符数组，或者指向了一个字符数组首元素
- 返回值为：指针 s

```
int puts( char * s );     /* 输出字符串 */
```

- s 为指向字符串的指针
- 返回值为：输出字符的个数

gets/puts – 字符串输入/输出函数

```
char * gets( char * s ); /* 读入字符串 */
```

- s 指向一段可写入字符串的内存的起始地址
  s 通常是一个字符数组，或者指向了一个字符数组首元素
- 返回值为：指针 s

```
int puts( char * s );     /* 输出字符串 */
```

- s 为指向字符串的指针
- 返回值为：输出字符的个数

gets/puts – 字符串输入/输出函数

```
char * gets( char * s ); /* 读入字符串 */
```

- s 指向一段可写入字符串的内存的起始地址
  s 通常是一个字符数组，或者指向了一个字符数组首元素
- 返回值为：指针 s

```
int puts( char * s );     /* 输出字符串 */
```

- s 为指向字符串的指针
- 返回值为：输出字符的个数

```
char s[100], *p;
gets(s); /* 读入一个字符串 */
gets(p); /* 错，指针p没有指向内存 */
puts(s); /* 输出字符串s */
```

gets/puts – 字符串输入/输出函数

自己写一个 gets 函数

```
char * gets( char * s )
{
    char *s0 = s;
    while( (*s = getchar())!='\n' )
        s++;
    *s = '\0';
    return s0;
}
```

gets/puts – 字符串输入/输出函数

自己写一个 gets 函数

```
char * gets( char * s )
{
    char *s0 = s;
    while( (*s = getchar())!='\n' )
        s++;
    *s = '\0';
    return s0;
}
```

自己写一个 puts 函数

```
int puts( char * s )
{
    int n = 0;
    while( s[n] ) putchar( s[n++] );
    return n;
}
```

strcpy – 字符串复制函数

```
char * strcpy( char * s1, char * s2 );
```

- 等价于赋值操作：字符串 s1 = 字符串 s2

strcpy – 字符串复制函数

```
char * strcpy( char * s1, char * s2 );
```

- 等价于赋值操作：字符串 s1 = 字符串 s2
- 参数要求：

strcpy – 字符串复制函数

```
char * strcpy( char * s1, char * s2 );
```

- 等价于赋值操作：字符串 s1 = 字符串 s2
- 参数要求：
  - s2 指向一个以 0 为结尾的字符串

strcpy – 字符串复制函数

```
char * strcpy( char * s1, char * s2 );
```

- 等价于赋值操作：字符串 s1 = 字符串 s2
- 参数要求：
  - s2 指向一个以 0 为结尾的字符串
  - s1 指向一段可写入字符串的内存的起始地址

strcpy – 字符串复制函数

```
char * strcpy( char * s1, char * s2 );
```

- 等价于赋值操作：字符串 s1 = 字符串 s2
- 参数要求：
  - s2 指向一个以 0 为结尾的字符串
  - s1 指向一段可写入字符串的内存的起始地址
  - s1 所指向的内存足以容纳字符串 s2

strcpy – 字符串复制函数

```
char * strcpy( char * s1, char * s2 );
```

- 等价于赋值操作：字符串 s1 = 字符串 s2
- 参数要求：
  - s2 指向一个以 0 为结尾的字符串
  - s1 指向一段可写入字符串的内存的起始地址
  - s1 所指向的内存足以容纳字符串 s2
- 返回值为：字符串指针 s1

strcpy – 字符串复制函数

```
char * strcpy( char * s1, char * s2 );
```

- 等价于赋值操作：字符串 s1 = 字符串 s2
- 参数要求：
  - s2 指向一个以 0 为结尾的字符串
  - s1 指向一段可写入字符串的内存的起始地址
  - s1 所指向的内存足以容纳字符串 s2
- 返回值为：字符串指针 s1

strcpy – 字符串复制函数

```
char * strcpy( char * s1, char * s2 );
```

- 等价于赋值操作：字符串 s1 = 字符串 s2
- 参数要求：
  - s2 指向一个以 0 为结尾的字符串
  - s1 指向一段可写入字符串的内存的起始地址
  - s1 所指向的内存足以容纳字符串 s2
- 返回值为：字符串指针 s1

```
char s[100];
strcpy(s, "Hello");
printf("%s", s);
```

strcpy – 字符串复制函数

```
char * strcpy( char * s1, char * s2 );
```

- 等价于赋值操作：字符串 s1 = 字符串 s2
- 参数要求：
  - s2 指向一个以 0 为结尾的字符串
  - s1 指向一段可写入字符串的内存的起始地址
  - s1 所指向的内存足以容纳字符串 s2
- 返回值为：字符串指针 s1

```
char s[100];
strcpy(s, "Hello");
printf("%s", s);
```

或者

```
char s[100];
printf("%s", strcpy(s, "Hello"));
```

strcat – 字符串连接函数

```
char * strcat( char * s1, char * s2 );
```

- 等价于符合赋值操作：字符串 s1 += 字符串 s2

strcat – 字符串连接函数

```
char * strcat( char * s1, char * s2 );
```

- 等价于符合赋值操作：字符串 s1 += 字符串 s2
- 参数要求：

strcat – 字符串连接函数

```
char * strcat( char * s1, char * s2 );
```

- 等价于符合赋值操作：字符串 s1 += 字符串 s2
- 参数要求：
  - s1 和 s2 都指向以 0 为结尾的字符串

strcat – 字符串连接函数

```
char * strcat( char * s1, char * s2 );
```

- 等价于符合赋值操作：字符串 s1 += 字符串 s2
- 参数要求：
  - s1 和 s2 都指向以 0 为结尾的字符串
  - 字符串 s1 之后还有足够空间，可容纳字符串 s2

strcat – 字符串连接函数

```
char * strcat( char * s1, char * s2 );
```

- 等价于符合赋值操作：字符串 s1 += 字符串 s2
- 参数要求：
  - s1 和 s2 都指向以 0 为结尾的字符串
  - 字符串 s1 之后还有足够空间，可容纳字符串 s2
- 返回值为：字符串指针 s1

strcat – 字符串连接函数

```
char * strcat( char * s1, char * s2 );
```

- 等价于符合赋值操作：字符串 s1 += 字符串 s2
- 参数要求：
  - s1 和 s2 都指向以 0 为结尾的字符串
  - 字符串 s1 之后还有足够空间，可容纳字符串 s2
- 返回值为：字符串指针 s1

strcat – 字符串连接函数

```
char * strcat( char * s1, char * s2 );
```

- 等价于符合赋值操作：字符串 s1 += 字符串 s2
- 参数要求：
  - s1 和 s2 都指向以 0 为结尾的字符串
  - 字符串 s1 之后还有足够空间，可容纳字符串 s2
- 返回值为：字符串指针 s1

```
char s[100] = "water";
strcat(s, "mellon"); /* s += "mellon" */
puts(s); /* 输出是什么？*/
```

strcat – 字符串连接函数

```
char * strcat( char * s1, char * s2 );
```

- 等价于符合赋值操作：字符串 s1 += 字符串 s2
- 参数要求：
  - s1 和 s2 都指向以 0 为结尾的字符串
  - 字符串 s1 之后还有足够空间，可容纳字符串 s2
- 返回值为：字符串指针 s1

```
char s[100] = "water";
strcat(s, "mellon"); /* s += "mellon" */
puts(s); /* 输出是什么？*/
```

或者

```
char s[100] = "water";
puts(strcat(s, "mellon"));
```

strcat – 字符串连接函数

```
char * strcat( char * s1, char * s2 );
```

- 等价于符合赋值操作：字符串 s1 += 字符串 s2
- 参数要求：
  - s1 和 s2 都指向以 0 为结尾的字符串
  - 字符串 s1 之后还有足够空间，可容纳字符串 s2
- 返回值为：字符串指针 s1

```
char s[100] = "water";
strcat(s, "mellon"); /* s += "mellon" */
puts(s); /* 输出是什么？*/
```

或者

```
char s[100] = "water";
puts(strcat(s, "mellon"));
```

下面的调用可以吗？为什么？

```
strcat("hot", s));
```

strcpy 和 strcat 函数定义

自己写一个 strcpy 函数

```
char * strcpy( char * s1, char * s2 )
{
    char * s0 = s1;
    while( *s1++ = *s2++ )
        ;
    return s0;
}
```

strcpy 和 strcat 函数定义

自己写一个 strcpy 函数

```
char * strcpy( char * s1, char * s2 )
{
    char * s0 = s1;
    while( *s1++ = *s2++ )
        ;
    return s0;
}
```

自己写一个 strcat 函数

```
char * strcat( char * s1, char * s2 )
{
    char * s0 = s1;
    while( *s1 ) s1++; /* 移动到字符串末尾 */
    while( *s1++ = *s2++ ) ; /* 复制 s2 */
    return s0;
}
```

字符串比较规则

给定两个字符串 s1 和 s2，比较规则如下：

- 从第 0 个字符，逐个字符对进行比较，直到

字符串比较规则

给定两个字符串 s1 和 s2，比较规则如下：

- 从第 0 个字符，逐个字符对进行比较，直到
1. 出现不一样的字符。假设第 k 个字符不一样

字符串比较规则

给定两个字符串 s1 和 s2，比较规则如下：

- 从第 0 个字符，逐个字符对进行比较，直到
1. 出现不一样的字符。假设第 k 个字符不一样
   - 那么，字符串 s1 和 s2 大小关系定义为：s1[k] 和 s2[k] 大小关系

字符串比较规则

给定两个字符串 s1 和 s2，比较规则如下：

- 从第 0 个字符，逐个字符对进行比较，直到

1. 出现不一样的字符。假设第 k 个字符不一样
   - 那么，字符串 s1 和 s2 大小关系定义为：s1[k] 和 s2[k] 大小关系
2. 两个字符串都结束了

字符串比较规则

给定两个字符串 s1 和 s2，比较规则如下：

- 从第 0 个字符，逐个字符对进行比较，直到

1. 出现不一样的字符。假设第 k 个字符不一样
   - 那么，字符串 s1 和 s2 大小关系定义为：s1[k] 和 s2[k] 大小关系
2. 两个字符串都结束了
   - 那么，字符串 s1 和 s2 完全相等

字符串比较规则

给定两个字符串 s1 和 s2，比较规则如下：

- 从第 0 个字符，逐个字符对进行比较，直到
1. 出现不一样的字符。假设第 k 个字符不一样
   - 那么，字符串 s1 和 s2 大小关系定义为：s1[k] 和 s2[k] 大小关系
2. 两个字符串都结束了
   - 那么，字符串 s1 和 s2 完全相等
- 例如："abc" < "b"

字符串比较规则

给定两个字符串 s1 和 s2，比较规则如下：

- 从第 0 个字符，逐个字符对进行比较，直到
1. 出现不一样的字符。假设第 k 个字符不一样
   - 那么，字符串 s1 和 s2 大小关系定义为：s1[k] 和 s2[k] 大小关系
2. 两个字符串都结束了
   - 那么，字符串 s1 和 s2 完全相等
- 例如："abc" < "b"
- 例如："Abc" < "abc"

strcmp – 字符串比较函数

```
int strcmp( char * s1, char * s2 );
```

- 可以理解为：字符串 s1 - 字符串 s2

strcmp – 字符串比较函数

```
int strcmp( char * s1, char * s2 );
```

- 可以理解为：字符串 s1 - 字符串 s2
- 要求：s1 和 s2 都指向以 0 为结尾的字符串

strcmp – 字符串比较函数

```
int strcmp( char * s1, char * s2 );
```

- 可以理解为：字符串 s1 - 字符串 s2
- 要求：s1 和 s2 都指向以 0 为结尾的字符串
- 返回值为：

strcmp – 字符串比较函数

```
int strcmp( char * s1, char * s2 );
```

- 可以理解为：字符串 s1 - 字符串 s2
- 要求：s1 和 s2 都指向以 0 为结尾的字符串
- 返回值为：
  - 负数 – 当字符串 s1 < 字符串 s2

strcmp – 字符串比较函数

```
int strcmp( char * s1, char * s2 );
```

- 可以理解为：字符串 s1 - 字符串 s2
- 要求：s1 和 s2 都指向以 0 为结尾的字符串
- 返回值为：
  - 负数 – 当字符串 s1 < 字符串 s2
  - 为零 – 当字符串 s1 == 字符串 s2

strcmp – 字符串比较函数

```
int strcmp( char * s1, char * s2 );
```

- 可以理解为：字符串 s1 - 字符串 s2
- 要求：s1 和 s2 都指向以 0 为结尾的字符串
- 返回值为：
  - 负数 – 当字符串 s1 < 字符串 s2
  - 为零 – 当字符串 s1 == 字符串 s2
  - 正数 – 当字符串 s1 > 字符串 s2

strcmp – 字符串比较函数

```
int strcmp( char * s1, char * s2 );
```

- 可以理解为：字符串 s1 - 字符串 s2
- 要求：s1 和 s2 都指向以 0 为结尾的字符串
- 返回值为：
  - 负数 – 当字符串 s1 $<$ 字符串 s2
  - 为零 – 当字符串 s1 $==$ 字符串 s2
  - 正数 – 当字符串 s1 $>$ 字符串 s2

strcmp – 字符串比较函数

```
int strcmp( char * s1, char * s2 );
```

- 可以理解为：字符串 s1 - 字符串 s2
- 要求：s1 和 s2 都指向以 0 为结尾的字符串
- 返回值为：
  - 负数 – 当字符串 s1 < 字符串 s2
  - 为零 – 当字符串 s1 == 字符串 s2
  - 正数 – 当字符串 s1 > 字符串 s2

自己写一个 strcmp 函数

```
int strcmp( char * s1, char * s2 )
{
    while( *s1 == *s2 && *s1 )
        s1++, s2++;
    return *s1 - *s2;
}
```

strlen – 求字符串长度

```
int strlen( char * s );
```

- 参数 s 指向以 0 为结尾的字符串

strlen – 求字符串长度

```
int strlen( char * s );
```

- 参数 s 指向以 0 为结尾的字符串
- 返回值：字符串 s 的长度，即 s 中的字符个数

strlen – 求字符串长度

```
int strlen( char * s );
```

- 参数 s 指向以 0 为结尾的字符串
- 返回值：字符串 s 的长度，即 s 中的字符个数
  - 不包括末尾的'\0'

strlen – 求字符串长度

```
int strlen( char * s );
```

- 参数 s 指向以 0 为结尾的字符串
- 返回值：字符串 s 的长度，即 s 中的字符个数
  - 不包括末尾的'\0'

strlen 应用举例

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

strlen – 求字符串长度

```
int strlen( char * s );
```

- 参数 s 指向以 0 为结尾的字符串
- 返回值：字符串 s 的长度，即 s 中的字符个数
  - 不包括末尾的'\0'

strlen 应用举例

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

- strlen(s) 等于多少？

strlen – 求字符串长度

```
int strlen( char * s );
```

- 参数 s 指向以 0 为结尾的字符串
- 返回值：字符串 s 的长度，即 s 中的字符个数
  - 不包括末尾的'\0'

strlen 应用举例

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

- strlen(s) 等于多少？ 5

strlen – 求字符串长度

```
int strlen( char * s );
```

- 参数 s 指向以 0 为结尾的字符串
- 返回值：字符串 s 的长度，即 s 中的字符个数
  - 不包括末尾的'\0'

strlen 应用举例

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

- strlen(s) 等于多少？ 5
- sizeof(s) 等于多少？

strlen – 求字符串长度

```
int strlen( char * s );
```

- 参数 s 指向以 0 为结尾的字符串
- 返回值：字符串 s 的长度，即 s 中的字符个数
  - 不包括末尾的'\0'

strlen 应用举例

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

- strlen(s) 等于多少？ 5
- sizeof(s) 等于多少？ 6

strlen – 求字符串长度

```
int strlen( char * s );
```

- 参数 s 指向以 0 为结尾的字符串
- 返回值：字符串 s 的长度，即 s 中的字符个数
  - 不包括末尾的'\0'

strlen 应用举例

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

- strlen(s) 等于多少？ 5
- sizeof(s) 等于多少？ 6
- strlen(s+2) 等于多少？

strlen – 求字符串长度

```
int strlen( char * s );
```

- 参数 s 指向以 0 为结尾的字符串
- 返回值：字符串 s 的长度，即 s 中的字符个数
  - 不包括末尾的'\0'

strlen 应用举例

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

- strlen(s) 等于多少？ 5
- sizeof(s) 等于多少？ 6
- strlen(s+2) 等于多少？ 3

strlen – 求字符串长度

自己写一个 strlen 函数

```
int strlen( char * s )
{
    char * p = s;
    while( *p ) p++;
    return p - s;
}
```

字符串压缩 - 例 8-8

输入一个字符串，按规则对字符串进行压缩，并输出压缩后的字符串。压缩规则：

- 如果一个字符 x 连续出现 n（n>1），那么将它们替换为 nx

字符串压缩 - 例 8-8

输入一个字符串，按规则对字符串进行压缩，并输出压缩后的字符串。压缩规则：

- 如果一个字符 x 连续出现 n（n>1），那么将它们替换为 nx
  否则，保持不变

字符串压缩 - 例 8-8

输入一个字符串，按规则对字符串进行压缩，并输出压缩后的字符串。压缩规则：

- 如果一个字符 x 连续出现 n（n>1），那么将它们替换为 nx
  否则，保持不变
- 例如：aaabbcddddddddddddddde 压缩后为：3a2bc15de

字符串压缩 - 例 8-8

输入一个字符串，按规则对字符串进行压缩，并输出压缩后的字符串。压缩规则：

- 如果一个字符 x 连续出现 n（n>1），那么将它们替换为 nx
  否则，保持不变
- 例如：aaabbcddddddddddddddde 压缩后为：3a2bc15de

字符串压缩 - 例 8-8

输入一个字符串，按规则对字符串进行压缩，并输出压缩后的字符串。压缩规则：

- 如果一个字符 x 连续出现 n（n>1），那么将它们替换为 nx
  否则，保持不变
- 例如：aaabbcddddddddddddddde 压缩后为：3a2bc15de

```
#define MAXLINE 80
void zipstring(char *p);
int main(void)
{
    char line[MAXLINE];
    printf("输入字符串");
    gets(line);
    zipstring(line);
    puts(line);
    return 0;
}
```

字符串压缩 - 例 8-8

```
char*int2str(char *q, int n);
void zipstring(char *p)
{
    char *q = p;
    int n;
    while( *p ) {
        for( n = 1; *p==p[n]; n++ )
            ; //统计连续重复的次数
        printf("n=%d\n",n);
        if( n>1 )
            q = int2str(q,n);//保存n
        *q++ = p[n-1];
        p = p + n;
    }
    *q = '\0';
}
```

字符串压缩 - 例 8-8

```
char *int2str(char *q, int n)
{
    int power = 1, m = n;
    while( m>9 ) {
        power *= 10;
        m /= 10;
    }
    while( power>0 ) {
        *q++ = n/power + '0';
        n %= power;
        power /= 10;
    }
    return q;
}
```

总结

- 变量、内存单元、地址，他们是什么关系。
- 如何定义、怎样使用指针变量？指针的作用？
- 指针变量的初始化、基本运算
- 指针类型的形式参数
- 指针与数组
- 冒泡排序与二分查找
- 动态内存分配（下学期学习）
- 字符串与字符指针
- 字符串处理函数

今天到此为止