西北工业大学
NORTHWESTERN POLYTECHNICAL UNIVERSITY

C程序设计

Programming in C

1011014

主讲：姜学锋，计算机学院

数组元素的简洁表示

1、指向一维数组元素的指针

7.3 指针与数组

- 指针与数组有着十分密切的联系，除了用数组下标访问数组元素外，C程序员更偏爱使用指针来访问数组元素，这样做的好处是运行效率高、写法简洁。

7.3.1 指向一维数组元素的指针

- 一个对象占用内存单元有地址，一个数组元素占用内存单元同样有地址。

7.3.1 指向一维数组元素的指针

- 1．一维数组元素的地址
- 数组由若干个元素组成，每个元素都占用内存单元，因而每个元素都有相应的地址，通过取地址运算（&）可以得到每个元素的地址。例如：

```
1 int a[10];
2 int *p=&a[0]; //定义指向一维数组元素的指针
3 p=&a[5]; //指向a[5]
```

- 第2行用a[0]的地址作为指针变量p的初值，则p指向a[0]；第3行将a[5]的地址赋值给指针变量p，则p指向a[5]。

7.3.1 指向一维数组元素的指针

- 数组对象可以看作是一个占用更大存储空间的对象，它也有地址。C语言规定，数组名既代表数组对象，又是数组首元素的地址值，即a与第0个元素的地址&a[0]相同。例如：

```
①p=a;
②p=&a[0];
```

- 是等价的。

7.3.1 指向一维数组元素的指针

- 将数组的首地址看作是数组对象的地址。例如：

```
int a[10];
int *p=a; //p指向数组a
```

7.3.1 指向一维数组元素的指针

- 数组名是地址值，是一个指针常量，因而它不能出现在左值和某些算术运算中，例如：

```
int a[10], b[10], c[10];
a=b; //错误，a是常量不能出现在左值的位置
c=a+b; //错误，a、b是地址值，不允许加法运算
a++; //错误，a是常量不能使用++运算
a>b //正确，表示两个地址的比较，而非两个数组内容的比较
```

7.3.1 指向一维数组元素的指针

- 2．指向一维数组元素的指针变量
- 定义指向一维数组元素的指针变量时，指向类型应该与数组元素类型一致，例如：

```
int a[10], *p1;
double f[10], *p2;
p1=a; //正确
p2=f; //正确
p1=f; //错误，指向类型不同不能赋值
```

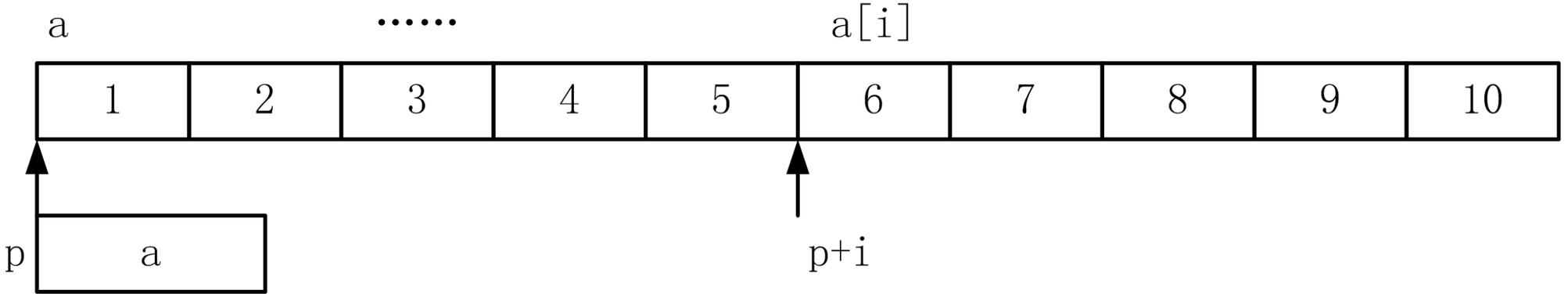
7.3.1 指向一维数组元素的指针

- 3．通过指针访问一维数组元素
- 由于数组元素是连续存储的，其内存地址是规律性增加的。根据指针算术运算规则，可以利用指针及其运算来访问数组元素。
- 设有如下定义：

```
int *p, a[10]={1,2,3,4,5,6,7,8,9,10};
p=a; //p指向数组a
```

7.3.1 指向一维数组元素的指针

图7.11 指向一维数组的指针



p=a使得p指向了数组a[0]元素的地址，即与p=&a[0]等价。那么，数组a[i]元素的地址既可以写为&a[i]，又可以写为p+i（指向a[0]元素后面的第i个元素），则a[i]元素可以写为*(p+i)。

7.3.1 指向一维数组元素的指针

- 同理，由于数组名表示数组首地址，a[i]元素的地址还可以写为a+i（a[0]元素后面的第i个元素的地址），则a[i]元素可以写为*(a+i)。
- 再者，重新考查a[i]的表示法，其形式可以归纳为：

地址[下标]

- 因此，a[i]还可以写为p[i]。

7.3.1 指向一维数组元素的指针

- 根据以上叙述，访问一个数组元素a[i]，可以用：
  - ①数组下标法：a[i]；
  - ②指针下标法：p[i]；
  - ③地址引用法：*(a+i)；
  - ④指针引用法：*(p+i)。
- 其中a是一维数组名，p是指向一维数组的指针变量且p=a。

7.3.1 指向一维数组元素的指针

【例7.8】

用多种方法遍历一维数组元素。

①下标法。

②通过地址间接访问数组元素。

③通过指向数组的指针变量间接访问元素。

7.3.1 指向一维数组元素的指针

例7.8

```
1 #include <stdio.h>
2 int main()
3 {
4   int a[10], i;
5   for(i=0;i<10;i++) scanf("%d",&a[i]); //实参是a[i]的地址
6   for(i=0;i<10;i++) printf("%d ",a[i]); //输出a[i]的值
7   return 0;
8 }
```

①下标法遍历一维数组元素。

7.3.1 指向一维数组元素的指针

例7.8

```
1 #include <stdio.h>
2 int main()
3 {
4   int a[10], i;
5   for (i=0;i<10;i++) scanf("%d", a+i); //实参是a[i]的地址
6   for (i=0;i<10;i++) printf("%d ",*(a+i)); //输出a[i]的值
7   return 0;
8 }
```

②通过地址间接访问数组元素。

7.3.1 指向一维数组元素的指针

例7.8

```
1 #include <stdio.h>
2 int main()
3 {
4   int a[10], *p;
5   for (p=a;p<a+10;p++) scanf("%d",p); //p值即a[i]的地址
6   for (p=a;p<a+10;p++) printf("%d ",*p); //*p即a[i]的值
7   return 0;
8 }
```

③通过指向数组的指针变量间接访问元素。

7.3.1 指向一维数组元素的指针

例7.8

以上3个程序的运行情况均如下：

```
1 2 3 4 5 6 7 8 9 10
```

```
1 2 3 4 5 6 7 8 9 10↙
```

7.3.1 指向一维数组元素的指针

- ①下标法遍历一维数组元素。

```
4 int a[10], i;
5 for (i=0;i<10;i++) scanf("%d",&a[i]);
6 for (i=0;i<10;i++) printf("%d ",a[i]);
```

- ②通过地址间接访问数组元素。

```
4 int a[10], i;
5 for (i=0;i<10;i++) scanf("%d", a+i);
6 for (i=0;i<10;i++) printf("%d ", *(a+i));
```

第②种方法中，a+i为a[i]的地址，等价于&a[i]，*(a+i)等价于a[i]。因此第①种方法和第②种方法完全一样。

7.3.1 指向一维数组元素的指针

- ③通过指向数组的指针变量间接访问元素。

```
4 int a[10], *p;
5 for (p=a;p<a+10;p++) scanf("%d",p);
6 for (p=a;p<a+10;p++) printf("%d ",*p);
```

- 如果第6行换为：

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

- 则输出的是数组元素的地址。

scanf函数的实参需要元素的地址，即p的值，printf函数输出元素的值，即p所指向的元素的值。

7.3.2 指向一维数组元素的指针

- ③通过指向数组的指针变量间接访问元素。

```
4 int a[10], *p;
5 for (p=a;p<a+10;p++) scanf("%d",p);
6 for (p=a;p<a+10;p++) printf("%d ",*p);
```

第③种方法比第①、②种方法快，因为指针变量直接指向数组元素，不必每次重新计算元素地址。类似p++的自增运算快于加法运算，大大提高了数组元素访问效率。

7.3.1 指向一维数组元素的指针

- ③通过指向数组的指针变量间接访问元素。

```
4 int a[10], *p;
5 for (p=a;p<a+10;p++) scanf("%d",p);
6 for (p=a;p<a+10;p++) printf("%d ",*p);
```

在第5行执行完成后，p++运算后指向了“a[10]”，对于数组a来说，“a[10]”不是已知对象，因此若继续进行p++运算，则指针已经是无效的。所以第6行开始输出数组元素前，再次将p指向数组a，确保p指针是有效的。

7.3.1 指向一维数组元素的指针

- 使用指针访问数组元素，指针本身是可以指向数组之外的，运行时一旦进行指针间接引用，往往会导致程序的严重错误（相当于数组越界使用）。由于这样的程序编译器不会给出任何提示（语法是正确的），因此这种错误比较隐蔽，难于发现。

7.3.1 指向一维数组元素的指针

- 实际编程中，若程序出现了崩溃性的严重错误，多数情况下是因为程序欲存取一个未知对象。例如：

```
int a[10], *p=a, i=10, *p1;
a[10]=5; //错误，数组a只有a[0]...a[9]，a[10]是未知对象
*(p+20)=5; //错误，等价于a[20]，a[20]是未知对象
p--; //p指向a[-1]
*p=5; //错误，等价于a[-1]=5，a[-1]是未知对象
*p1=5; //错误，p1未初始化或未赋值，引用未知对象
```

7.3.1 指向一维数组元素的指针

- 4．数组元素访问方法的比较
- （1）使用下标法访问数组元素，程序写法比较直观，能直接知道访问的是第几个元素，例如a[3]是数组第3个元素（从0开始计）。用地址法或指针法就不直观，需要结合程序上下文才能判断是哪一个元素。

7.3.1 指向一维数组元素的指针

- （2）下标法与地址引用法运行效率相同。实际上，编译器总是将a[i]转换为*(a+i)、&a[i]转换为a+i处理的。即访问元素前需要先计算元素地址。使用指针引用法，指针变量直接指向元素，不必每次都重新计算地址，能提高运行效率。

7.3.1 指向一维数组元素的指针

- （3）a[i]和p[i]的运行效率相同，但两者还是有本质的区别。数组名a是数组元素首地址，它是一个指针常量，其值在程序运行期间是固定不变的，例如：

```
a++; //错误，a是常量不能作自增运算
```

- 而p是一个指针变量，可以用p++使p值不断改变从而指向不同的元素。

7.3.1 指向一维数组元素的指针

- 一旦p值不再是数组首地址，则a[i]和p[i]就不一定是相同的元素了。例如：

```
int a[10], *p=a;
p[5]=10; //此时的p[5]实际是a[5]
p++;
p[5]=10; //此时的p[5]实际是a[6]
```

7.3.1 指向一维数组元素的指针

- （4）将自增和自减运算用于指针变量十分有效，可以使指针变量自动向前或向后指向数组的下一个或前一个元素。例如遍历数组的100个元素，程序代码如下：

```
int a[100],*p=a;
while (p<a+100) *p++=0; //数组每个元素都赋值为0
```

7.3.1 指向一维数组元素的指针

- （5）需要注意指针变量各种运算形式的含义。
- ①*p++。由于++和*优先级相同，结合性自右向左，因此它等价于*(p++)，其作用是表达式先得到p所指向的元素的值（即*p），然后再使p指向下一个。若p初值为a，则*p++的结果是a[0]，p指向a[1]。

7.3.1 指向一维数组元素的指针

- ②*(p++)和*(++p)不同。前者是先取*p值，然后p加1。后者是先使p加1，再取*p。若p初值为a，则*(p++)的结果是a[0]，*(++p)的结果是a[1]，运算后p均指向a[1]。

7.3.1 指向一维数组元素的指针

- ③(*p)++表示p所指向的元素加1。若p初值为a，(*p)++等价于a[0]++，运算后p值不变。

- ④假定p指向数组a中的第i个元素，即p=&a[i]，则：
  - a. *(p++)等价于a[i++]；
  - b. *(++p)等价于a[++i]；
  - c. *(p--)等价于a[i--]；
  - d. *(--p)等价于a[--i]。


程序设计