08 指针

通过地址访问对象，连接函数与数组

本讲学习目标

- 区分对象、地址、指针变量与所指对象。
- 定义和初始化指针，通过有效指针读写对象。
- 解释指针参数，编写修改调用者对象的函数。
- 说明数组与指针的关系，确定处理范围并安全遍历。

指针：从数值传递到原对象访问

程序 = 数据对象 + 操作 + 控制 + 组合

- 普通数值参数只提供数值副本。函数要读写调用方指定的原对象，还需要能定位该对象的信息。
- 指针保存对象的地址。函数接收地址后，可以通过**间接访问**读写地址所指的原对象。

1 通过地址修改原变量

定位原变量，通过地址访问和修改

函数怎样修改调用方指定的变量

- 已学过的两种现象
  - 修改普通整数形参，只改变函数中的副本。
  - 数组处理函数可以修改调用方的原数组元素。
- 本次任务
  - main 中有整数变量 value，初值为 7。
  - 调用 set_value 后，应将原变量 value 的值改为 10。
- 把 value 的整数值传给函数，能否完成这项任务?

函数参数按值传递

```c
#include <stdio.h>
void set_value(int x)
{
    x = 10;
}

int main(void)
{
    int value = 7;
    set_value(value);
    printf("%d\n", value);
    return 0;
}
```

按值传递

set_value(value) 将 value 的整数值 7 复制给形参 x

x 是函数内部新创建的对象，与 main 中的 value 不同

函数内的修改

x = 10 只修改形参 x 的值

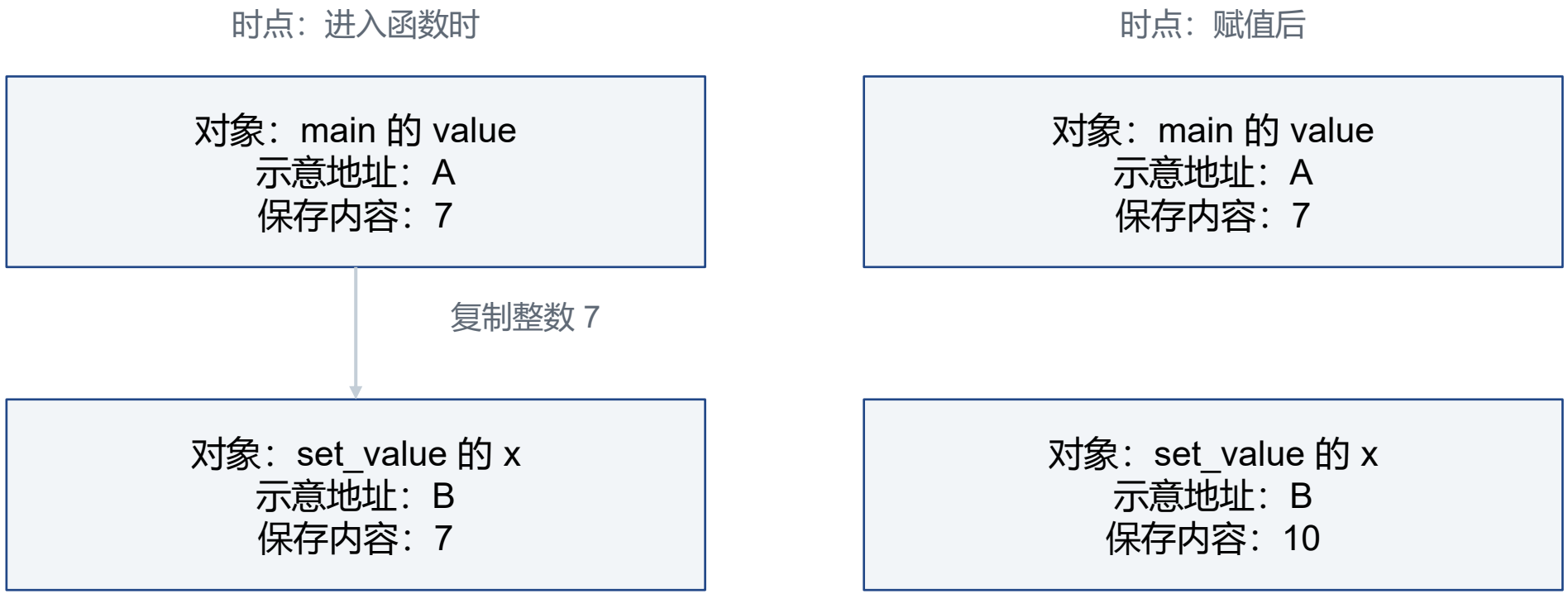
调用后的结果

输出为 7，原变量 value 未改变，未达到改为 10 的目标

整数按值传递：修改的是形参副本

调用 set_value(value) 时，把整数 7 复制给形参 x。

执行 x = 10 后，只改变形参的内容；main 中的 value 仍为 7。



内存对象示意：A、B表示不同对象的位置，不表示实际地址、对象大小或栈的排列。

数值不能说明原变量在哪里

- 前页调用 set_value 时，形参 x 接收 value 的整数值副本 7。
  - 多个变量可以保存相同的值 7，因此仅凭这个数值无法确定要修改哪个变量。
- 函数还需要能定位原变量的信息。
  - 沿用第7讲的定义：地址是用于定位对象的值。
  - 函数保存传入的 value 地址后，便可通过该地址访问原变量。
- 通过地址访问对象包括三步：取得地址、保存地址、间接访问对象。

函数也可以返回结果，由调用方赋值；接下来学习通过参数直接访问原变量的方式。

用储物柜理解变量、数组与指针

```
int price = 99;
int scores[3] = {95, 88, 100};
int *note = &price;
```



柜号仅用于类比，不表示实际机器地址或地址间距

储物柜类比中的变量、数组与指针

```
int price = 99;
int scores[3] = {95, 88, 100};
int *note = &price;
```

把对象看作储物柜：**变量名是标签，地址是柜号，当前值是柜中保存的内容**

- 普通变量：price 的柜中保存整数 99
- 数组：scores 是三个元素组成的整体，各元素分别保存一个整数
- 指针变量：note 的柜中保存 price 的地址，按这个地址可以找到 price
  - &price 取得 price 的地址；*note 指定 price，读取 *note 得到 99

图中的柜号用于类比，不表示实际机器地址

变量的值与变量的地址

```c
int x = 7;
printf("%d\n", x);
printf("%p\n", (void *)&x);
```

对象、值与地址

x 是保存整数的对象，当前值为 7

使用 x 输出整数时，读取的是当前值；&x 取得 x 的地址

取地址运算符 & 不读取 x 中的整数，也不改变 x 的值

输出内容与格式

%d 输出整数值 7

%p 配合 (void *)&x 输出地址；具体地址随运行与环境变化

指针变量保存地址

```
1 int x = 7;
2 int *p = &x;  // 定义时初始化，使 p 指向 x
```

指针类型

int * 表示指向 int 对象的指针类型

指针对象与所指对象

p 保存 x 的地址时，称 p 指向 x

p 与 x 是两个不同对象；定义 p 不会额外创建一个 int 对象

本例先创建 x，再用 &x 初始化 p，已建立有效指向。

通过指针读取和修改对象的值

```
1 int x = 7;
2 int *p = &x;
3 int value = *p;
4 *p = 10;
5 printf("%d\n", x);
```

间接访问的对象

p 保存 x 的地址；*p 指定 p 所指的对象 x

间接访问运算符 * 也称解引用，使用时要求指针有效

读取所指对象的值

int value = *p 读取 x 的值 7，并保存到 value

修改所指对象的值

*p = 10 将 x 的值改为 10，输出为 10

p 保存的地址不变，仍指向 x

通过指针参数修改函数外的变量

```c
#include <stdio.h>
void set_value(int *p)
{
    *p = 10;
}

int main(void)
{
    int value = 7;
    set_value(&value);
    printf("%d\n", value);
    return 0;
}
```

按值传递的内容

set_value(&value) 将 value 的地址值复制给指针形参 p

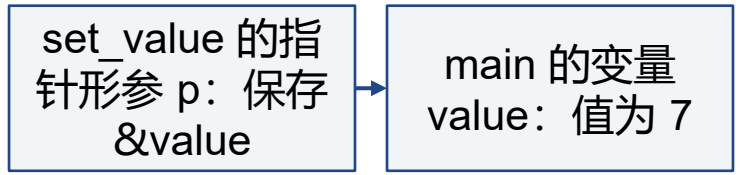
形参 p 指向 main 中仍存在、可写的 value（图示为写入前）

通过形参访问原对象

*p 指定 main 中的 value，*p = 10 将其值改为 10

调用后的结果

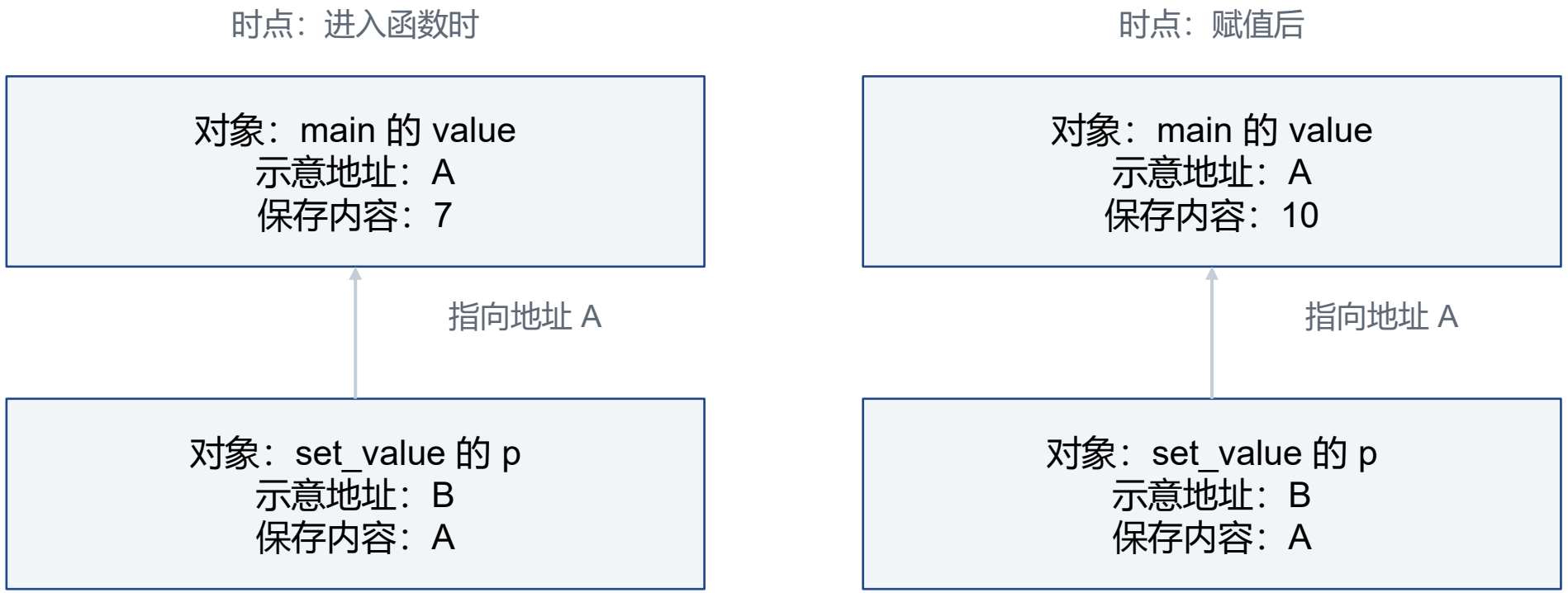
输出为 10，原变量 value 已修改；地址值仍按值传递



地址按值传递：访问同一个原对象

调用 set_value(&value) 时，形参 p 得到 value 的地址 A 的副本。

执行 *p = 10 后，地址 A 处的 value 改为10；p仍保存A，参数传递方式仍是按值传递。



内存对象示意：A、B仅表示不同位置；箭头表示指向，value的内容改变，p中的地址不变。

scanf 通过地址写入变量

```c
int count = 0;
if (scanf("%d", &count) == 1) {
    printf("%d\n", count);
}
```

地址实参与类型要求

&count 定位 count 对象；不能把 count 的整数值当成地址

%d 对应的实参须为 int *，&count 的类型符合要求

输入结果的写入位置

scanf 将转换所得整数写入 count

返回值与结果使用

返回值为 1 表示本次成功读入一个整数

条件成立后才输出本次输入结果

scanf 用返回值报告转换结果，用 &count 指定数据的写入位置

通过指针交换两个变量的值

```c
#include <stdio.h>
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
int main(void) {
    int x = 3, y = 8;
    swap(&x, &y);
    printf("%d %d\n", x, y);
    return 0;
}
```

调用与修改目标

swap(&x, &y) 使 a 指向 x、b 指向 y，修改的是 x、y 的值

这里应传入 &x 和 &y，而不是整数值 x 和 y

交换过程

temp = *a 保存原值 3，避免覆盖后丢失

*a = *b 将 x 改为 8；*b = temp 将 y 改为 3

输出为 8 3；a、b 保存的地址未改变

调用前提

a、b 均须指向可读写的有效 int 对象

2 指针的指向与访问条件

区分指针与所指对象的修改，追踪共同指向，并判断访问是否合法

指针定义中的类型与初始化

```
1 int x = 7;
2 int *p = &x;
3
4 double t = 21.5;
5 double *pt = &t;
6
7 int *later;    // 也可以先定义，再赋值
8 later = &x;    // 赋值前不能用 later 访问对象
```

指针类型与所指对象类型

p 的类型是 int *，所指对象 x 的类型是 int

pt 的类型是 double *，所指对象 t 的类型是 double

初始化时的类型匹配

p 用 &x 初始化，pt 用 &t 初始化，各自指向对应的对象

double 对象的地址不能直接作为 int * 使用

later 先定义，再赋值为 &x；赋值前不能用它访问对象

定义中的星号

* 声明指针，不是乘法或一次对象访问

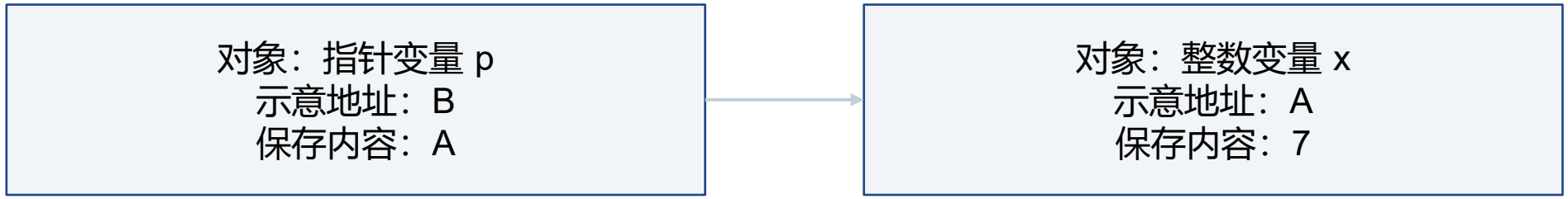
int *p, q; 只把 p 定义为指针，q 仍是 int

对象、地址与指向关系

代码：int x = 7; int *p = &x;

p 与 x 是两个对象；箭头表示 p 中保存的地址定位 x。

复制地址没有复制 x，x 仍保存整数 7。



A、B仅为示意地址。p自身位于B，保存指向x的地址A；读取*p得到x中的整数7。

改变指针的指向与修改所指对象的值

```
1 int x = 7, y = 9;
2 int *p = &x;
3 *p = 10;    // 修改所指对象 x 的值
4 p = &y;     // 改变 p 的指向，x、y 的值不变
5 *p = 20;    // 此时 p 指向 y，修改 y 的值
```

修改所指对象的值

*p = 10 修改 x，p 仍指向 x

*p = 20 执行时 p 已指向 y，因此修改 y

改变指针的指向

p = &y 修改 p 保存的地址，使 p 改为指向 y

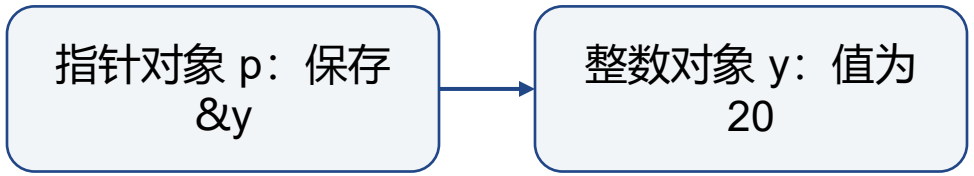
这条语句不改变 x、y 的值

最终结果

x 为 10，y 为 20；p 指向 y

重新指向后的访问对象

- 前页先用 *p = 10 把 x 改为 10，再用 p = &y 使 p 指向 y。
- 图示为执行 *p = 20 后的状态：y 为 20，x 仍为 10。
- p = &y 修改指针对象；*p = 20 修改当前所指的 y。



多个指针可以访问同一对象

```
1 int x = 7;
2 int *p = &x;
3 int *q = p;
4 *q += 2;
5 printf("%d\n", *p);
```

复制地址与共同指向

int *q = p 将 p 的指针值复制给 q

p 与 q 是不同指针对象，但都指向已有的 x

复制地址没有复制所指对象

修改与读取同一对象

*q += 2 将 x 从 7 改为 9

p 也指向 x，因此 *p 读到 9，输出为 9

练习：追踪指针的指向与变量的值

```
1 int x = 2, y = 8;
2 int *p = &x;
3 *p = 5;
4 p = &y;
5 *p = *p + 3;
```

指针的指向

p 最后指向 x 还是 y?

变量的值

x、y 最终各是多少?

最后一次赋值的目标

*p = *p + 3 读取并修改哪个变量?

根据当时 p 的指向说明理由

指针指向与变量值的变化过程

下表记录前页练习中每条语句执行后的指向与变量值

执行位置	p 指向	x	y
初始化 p 后	x	2	8
*p = 5 后	x	5	8
p = &y 后	y	5	8
*p = *p + 3 后	y	5	11

指针的指向：p = &y 后，p 改为指向 y

变量的值：最终 x 为 5，y 为 11

最后一次赋值的目标

当时 p 指向 y，*p = *p + 3 读取 y 的值 8，再将 11 写回 y

p = &y 只改变指向；修改 y 的值发生在下一条赋值语句

未初始化的局部指针不能用于访问对象

```
1 void invalid_access(void)
2 {
3     int *p;
4     *p = 10;
5 }
```

错误原因

局部指针 p 尚未初始化，因此不能用它确定访问目标

定义指针变量只创建指针对象，不会同时创建目标对象

出错语句

*p = 10 试图写入 p 指向的对象，但 p 尚无可用的目标地址

正确做法

先定义对象，再令指针保存其地址

例如 int x = 0; int *p = &x;

空指针不指向对象，不能间接访问

```c
#include <stddef.h>
int *p = NULL;  // 已初始化，暂时没有目标
```

空指针的含义

空指针不指向任何对象或函数；stddef.h 提供 NULL

NULL 可以作为初始值，表示已初始化但暂时没有目标

空指针的使用约定与访问条件

- **入门使用约定**
  - 有目标时，让指针保存有效对象的地址
  - 暂时没有目标时，将指针设为 NULL
- 判断与访问
  - p == NULL 检查 p 是否为空；p 为空时，不能通过 *p 读写对象
  - p != NULL 只排除空值，仍须确认所指对象可合法访问

通过指针访问对象的条件

- 指针已初始化，指向仍存在的对象。
- 指针类型与所指对象相容，访问未超出有效范围。
- 读取时对象已有有效值；写入时对象允许修改。
- 空指针、失效地址和越界访问不能靠一次运行结果判断合法。

未定义行为表示 C 不规定该操作会产生什么结果；可能崩溃，也可能暂时看不出问题。

练习：修改指针形参后的调用结果

```c
#include <stddef.h>

void reset_pointer(int *q)
{
    q = NULL;
}

int main(void)
{
    int x = 7;
    int *p = &x;
    reset_pointer(p);
    return 0;
}
```

调用者指针

调用结束后，p 是否为 NULL?

调用者变量

x 的值是否改变?

函数内的赋值目标

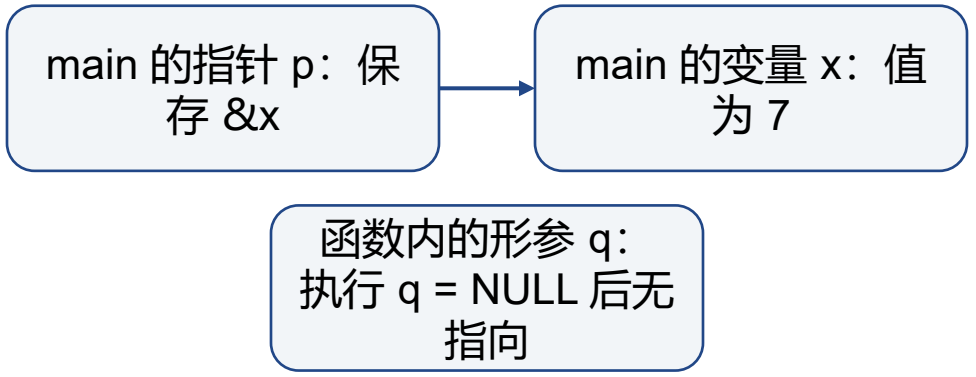
q = NULL 修改哪个对象？根据参数传递过程说明理由

修改指针形参只改变形参自身

- 传递的地址值：调用时，p 保存的地址值被复制给形参 q，x 没有被复制
- 函数内的修改：q = NULL 只改变 q，不改变 main 中的 p；本例没有通过 *q 写入 x
- 调用后的结果：p 仍指向 x，x 仍为 7

图示为 q = NULL 执行后、函数返回前的关系；返回后形参 q 不再存在

q 有效且指向 x 时，给 *q 赋值会修改 x；给 q 赋值只修改形参本身



常见错误：通过指针访问已失效的局部变量

```
1 int *p = NULL;
2 {
3     int x = 7;
4     p = &x;
5 }
6 *p = 10;  // 错误：x 已失效，不能再通过 p 写入
```

对象的生存期

块中的局部对象 x 只在该块执行期间存在

p 保存 x 的地址不会延长 x 的生存期

块结束后的访问限制

x 已不存在，原来指向 x 的指针值已失效

不要读取、比较或解引用这个失效的指针值；可给 p 重新赋予有效值

函数返回时的同类问题

不能返回生存期随调用结束的普通局部对象地址供继续访问

3 指针与数组

用首元素地址和显式范围传递、访问数组

通过指针访问数组的首元素

```
1 int values[3] = {4, 7, 9};
2 int *p = values;
3 *p = 10;
```

数组对象与指针对象

values 是包含三个 int 元素的数组；p 是独立的指针对象

首元素地址的取得

初始化 p 时，values 转换为指向首元素的指针

这里得到的指针值与 &values[0] 相同

修改原数组元素

*p 指定 values[0]，*p = 10 将它改为 10

数组变为 {10, 7, 9}

数组与指针的区别

```c
int values[3] = {4, 7, 9};
int *p = values;
printf("%zu\n", sizeof values);
printf("%zu\n", sizeof p);
```

对象与名称

values 是数组名，该数组包含三个 int 元素

p 是指针变量，保存首元素地址；数组名不是指针常量

赋值的区别

数组元素可分别修改，但不能用 values = p 给整个数组赋值

p 可以重新赋值，改为指向另一个有效 int 对象

sizeof 的区别

sizeof values 查询整个数组的字节数，此处不转换为指针

sizeof p 查询指针对象的字节数，不能由它推算数组元素个数

数组形参的两种等价写法

```
// 只读元素
// int sum_array(const int values[], int count);
int sum_array(const int *values, int count);

// 修改元素
// void increment_array(int values[], int count);
void increment_array(int *values, int count);
```

- 数组形参实际是指针形参
  - 形参声明中的 [] 写法调整为指针类型，注释中的写法与下方声明等价
  - 本页的写入接口使用 int *，只读接口使用 const int *

数组参数的读写职责与处理范围

- const 限制的是元素访问
  - int * 允许通过参数修改元素；const int * 表示通过参数只读元素
  - 这里的 const 修饰所指的 int，不限制指针形参本身改变指向
- 调用时传入地址与处理个数
  - 调用中的数组表达式转换为首元素指针，地址值复制给形参
  - 函数通过这个地址访问原数组；地址不携带元素个数，因此通常还需传入 count
  - count 指定要处理的元素个数，可以小于数组的元素总数

sizeof 数组形参得到的是指针大小

```c
#include <stdio.h>
// 等价的数组形参写法:
// void show_size(const int values[])
void show_size(const int *values)
{
    printf("%zu\n", sizeof values);
}

int main(void)
{
    int data[3] = {4, 7, 9};
    printf("%zu\n", sizeof data);
    show_size(data);
    return 0;
}
```

调用者中的查询

data 是数组，sizeof data 为 3 * sizeof(int)

函数内部的查询

values 是指针形参，sizeof values 查询指针对象的字节数

即使声明写为 values[]，这里查询的仍是指针

查询结果与处理范围

两次输出即使碰巧相同，也不表示查询了同类对象

元素个数须在数组对象所在处计算，再另外传给处理函数

通过指针参数求数组元素之和

```c
#include <stdio.h>
int sum_array(const int *values, int count) {
    int sum = 0;
    for (int i = 0; i < count; i++) {
        sum += values[i];
    }
    return sum;
}
int main(void) {
    int values[3] = {4, 7, 9};
    int count = sizeof values /
                sizeof values[0];
    printf("%d\n", sum_array(values, count));
    return 0;
}
```

参数与处理范围

调用中的 values 提供首元素地址，count 在数组所在处计算，本例为 3

const int * 表示通过参数只读元素；values[i] 的下标为 0 ~ count-1

求和与调用结果

sum 累加元素值并返回；读取 {4, 7, 9}，输出为 20

调用前提

count 应非负，前 count 个元素应可读；超出可访问范围会导致越界

每次累加结果均可用 int 表示；指针值不自动携带数组长度

通过指针参数将数组元素加一

```c
#include <stdio.h>
void increment_array(int *values, int count) {
    for (int i = 0; i < count; i++) {
        values[i] += 1;
    }
}
int main(void) {
    int values[3] = {4, 7, 9};
    increment_array(values, 3);
    for (int i = 0; i < 3; i++) {
        printf("%d ", values[i]);
    }
    return 0;
}
```

参数与处理范围

调用时，values 提供首元素地址，实参 3 指定处理三个元素

int * 允许写入元素，不能改为 const int * 后继续写入

原数组的修改与结果

values[i] += 1 修改原数组元素，没有复制完整数组

{4, 7, 9} 变为 {5, 8, 10}，随后输出 5 8 10

调用前提

count 非负，前 count 个元素可读写

每个值加一均可用 int 表示

指针的加减

```
1 int values[3] = {4, 7, 9};
2 int *p = values;
3 p += 2;
4 p--;
```

指针移动的单位

移动单位由所指元素类型决定；int * 加一跨过一个 int 元素

跨过的字节数不是固定为 1

指向的逐步变化

p += 2 后指向 values[2]，*p 为 9

p-- 后指向 values[1]，*p 为 7

运算范围

指针运算只在同一数组及其尾后位置范围内进行

尾后指针用作结束标记

```
1 int values[3] = {4, 7, 9};
2 int *begin = values;
3 int *end = values + 3;
```

起点与尾后位置

begin 指向 values[0]

end 为 values + 3，位于末元素之后，称为**尾后位置**

结束标记与允许范围

end 可用于判断遍历何时结束

values + 4 已超出允许形成的偏移范围

可以访问的元素范围

最后一个可访问元素为 values[2]

尾后位置不对应数组元素，因此不能通过 *end 读写

通过指针逐个访问数组元素

```c
int values[3] = {4, 7, 9};
int *end = values + 3;
for (int *p = values; p != end; p++) {
    printf("%d ", *p);
}
printf("\n");
```

遍历的起点与界限

p 从首元素开始，end 标记尾后位置

每轮执行过程

先判断 p != end，再通过 *p 读取当前元素

p++ 移动到下一个元素，继续判断

结束条件与输出

p == end 时循环结束，尾后指针不会被解引用

输出为 4 7 9

4 应用与回收

独立编写函数与调用，核对类型、对象与范围

练习：编写带指针参数的函数并调用

```c
void add_to(int *value, int delta);
int count_positive(const int *values, int count);
void clear_array(int *values, int count);
```

根据以下声明，独立编写三个函数的函数体及测试调用；可查语法：

- add_to：给 value 所指的整数加上 delta
  - 令 x 初值为 5，加 3 后检查 x 的值
- count_positive：统计正数个数
  - 使用 {-2, 0, 7, 3} 测试，检查返回值
- clear_array：将前 count 个元素清零
  - 检查范围外的元素是否保持原值

练习：测试处理范围并说明参数条件

- 数组函数的测试与说明
  - 分别测试处理 0 个、1 个元素的情况，说明只读与写入参数的区别
- 参数条件
  - 指针有效，count 非负且不超出可访问范围，整数加法结果可表示

三个函数的参考实现

```c
void add_to(int *value, int delta) {
    *value += delta;
}
int count_positive(const int *values,
                   int count) {
    int total = 0;
    for (int i = 0; i < count; i++) {
        if (values[i] > 0) total++;
    }
    return total;
}
void clear_array(int *values, int count) {
    for (int i = 0; i < count; i++) {
        values[i] = 0;
    }
}
```

add_to：修改所指变量

*value += delta 将 delta 加到函数外的整数变量

count_positive：读取并统计

读取前 count 个元素，返回 total 记录的正数个数

只处理 {7} 时返回 1，只处理 {-2} 时返回 0

clear_array：修改指定范围

将前 count 个元素写为 0，范围外元素不变

参数使用 int *，范围内元素必须可写

count 为 0 时

两个数组函数的循环均不执行；

count_positive 返回 0，clear_array 不修改元素

参考调用与测试结果

```
int x = 5;
add_to(&x, 3);
int a[4] = {-2, 0, 7, 3};
int n = count_positive(a, 4);
int empty = count_positive(a, 0);
int one = count_positive(a + 2, 1);
clear_array(a, 2);
int single[1] = {7};
clear_array(single, 0);
clear_array(single, 1);
```

修改一个整数

add_to(&x, 3) 通过 x 的地址修改原变量，x 从 5 变为 8

只读统计

处理全部 4 个元素，n 为 2；处理 0 个元素，empty 为 0

a + 2 指向元素 7，只处理这 1 个元素，one 为 1

三次统计均不修改 a

按范围清零

清零 a 的前 2 个元素后，a 为 {0, 0, 7, 3}，其余元素不变

对 single 处理 0 个元素，仍为 {7}；随后处理 1 个元素，变为 {0}

练习：分析并修正指针代码中的错误

```c
/* A */ int x = 7; int *p = x;
/* B */ int *p = NULL; *p = 10;
/* C */ int a[3] = {4, 7, 9}; int *p = a + 3; *p = 0;
/* D */ int sum_array(const int *a, int count) {
    return sizeof a / sizeof a[0];
}
```

以下是四个独立片段；只分析，不运行：

- 分析 A ~ D
  - 分别指出错误语句，从类型、有效指向、访问范围或函数职责说明原因
- 修正 B 或 C
  - 任选一个片段，在保留写入目的的前提下改成合法代码
  - 说明修改后写入的是哪个对象

不要根据错误片段推断确定的运行输出

错误原因与修改方法

- 错误原因
  - A：x 提供整数值 7，不能在此用于初始化 int *；若要 p 指向 x，应使用 &x
  - B：p 为 NULL，没有可供 *p 写入的对象
  - C：a + 3 是尾后位置，不能通过它写入元素
  - D：sizeof a 查询指针大小，不能求出数组长度；函数也未执行求和
- 合法修改示例
  - B：定义 int x = 0;，令 p = &x;，再执行 *p = 10;
  - C：若要修改末元素，令 p = a + 2;，再执行 *p = 0;

类型、指向和访问范围须逐项确认；偶然输出或未崩溃不能证明代码合法

本讲总结：指针、函数与数组

- **指针保存地址，*p 访问所指对象。** 给 p 赋值可改变指向，给 *p 赋值会修改所指对象的值。
- **函数参数仍按值传递。** 形参得到地址值的副本后，可以通过该地址访问调用方的原对象。
- **数组组织元素，指针保存地址。** 数组名在多数表达式中转换为首元素指针；数组形参调整为指针形参。
- **数组处理须明确范围与读写职责。** 处理个数通常另用 count 传入，const int * 限制通过参数修改元素。
- **通过指针访问对象须满足有效性条件。** 所指对象须仍存在，类型和权限须相容，访问不得越界；空指针和尾后指针不能解引用。

教材阅读与复习

本讲内容	《C Primer Plus》第六版对应
指针定义、取地址与间接访问	§9.7
通过指针参数修改函数外变量	§9.6 ~ 9.7
数组与指针的关系、数组参数	§10.3 ~ 10.4
指针运算与数组遍历	§10.5
const 参数与只读访问	§10.6、§10.6.1

对照课件代码，说明指针保存什么、访问哪个对象，以及处理范围如何确定

下一讲学习内存与变量生存期；字符串在第10讲展开