

CS100 Recitation 5

C 语言内存管理：生命周期、指针与动态分配

Speaker Chaofan Li

CS100 Teaching Assistant



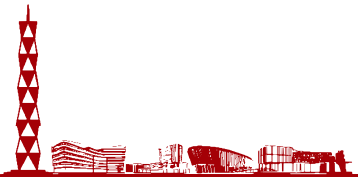
上海科技大学

ShanghaiTech University

今天我们要回答什么？

1. 我们为什么需要指针？
2. 运行时才知道需要多少空间，怎么办？
3. 这些对象通常放在哪些 memory 区域？
4. 一块动态分配的内存，最后应该由谁释放？

看到一个指针，先问：它指向谁？那个对象还活着吗？



目录

1 我们为什么需要指针?

2 运行时需要空间：动态分配

3 Memory 区域与对象的存储期

4 谁负责 free：指针、别名与所有权

我们为什么需要指针？

我们为什么需要指针？先看一次函数调用

目标：让 set_number 把调用者的 n 改为 42。

```c
#include <stdio.h>
void set_number(int x) {
    x = 42;
    printf("inside: %d\n", x);
}
int main(void) {
    int n = 10;
    set_number(n);
    printf("outside: %d\n", n);
    return 0;
}
```

先猜输出：函数里的 x 变了，main 里的 n 会跟着变吗？

立志成才 报国裕民

实参传给形参的是值的副本



- 输出 `inside: 42`，随后输出 `outside: 10`。
- `x` 与 `n` 是两个不同对象；修改 `x` 只改了副本。
- **生命周期（lifetime）**：形参 `x` 属于这次调用，返回后生命周期结束；此时调用者的 `n` 仍然活着。

我们需要让函数找到调用者的那个对象，而不只是拿到它当前的值。

把地址传进去，函数才能找到原来的对象


```c
#include <stdio.h>
void set_number(int *p) {
    *p = 42;
}
int main(void) {
    int n = 10;
    set_number(&n);
    printf("%d\n", n);
    return 0;
}
```

- `&n`：取得 `n` 的地址，作为实参。
- `int *p`：形参 `p` 保存这个地址值的副本。
- `*p = 42`：通过地址修改 `n`，输出 42。

C 仍然按值传递，这次复制的值是地址。

若只需要返回一个结果，也可以 `return` 后由调用者赋值；这里演示直接修改调用者指定的对象。

指针变量和它指向的对象，各有生命周期



- `set_number` 返回：形参 `p` 结束生命周期，`n` 继续存在。
- 反过来也可能发生：指针变量还在，所指对象已经结束生命周期。
- **能否通过指针访问对象，取决于所指对象此时是否仍然有效。**

接下来：如果把**函数内部局部变量的地址**带出去，会怎样？

立志成才 报国裕民

尝试写一个“构造函数”：create_point


```c
struct Point { int x; int y; };
struct Point *create_point(int x, int y) {
    struct Point point = {x, y};
    return &point;              /* wrong: local object */
}
int main(void) {
    struct Point *p = create_point(10, 15);
    struct Point new_point = {20, 30};
    p->x = 99;                  /* UB: dangling pointer */
    printf("%d\n", new_point.x); /* no guaranteed output */
    return 0;
}
```

- 想用 create_point 创建并初始化结构体，再把指针交给调用者。
- 但 point 是局部对象，函数返回后生命周期结束，返回的指针悬空。
- 通过悬空指针访问成员是 **UB**；new_point.x 的结果没有保证。

立志成才 报国裕民

为什么新变量可能“无辜被改”？

第一次调用：对象仍然活着

某块栈空间
`point = {10, 15}`

取得 `&point`

返回 main：后续对象可能复用空间

原来的 `point` 已不存在
新对象 `new_point = {20, 30}`

`p->x = 99`（悬空写入，UB）

- 常见栈实现会复用空间，错误写入**可能破坏后来创建的对象**。
- **复用同一个地址，不会让悬空指针重新变得有效。**
- 这是可能的故障过程，不保证地址复用，也不保证 `new_point.x` 最终为 99。

如果需要让对象在函数返回后继续存在，就要选择合适的存储期。下面看动态分配。

运行时需要空间：动态分配

为什么需要 Dynamic allocation?

- 输入告诉你有多少个元素，编译时还不知道。
- 刚才的 `create_point`：我们希望创建出的结构体在函数返回后继续存在。
- 数组需要扩容，或者需要逐个建立链表节点。

	automatic 局部对象	**动态分配对象**
创建	进入对应块时自动管理	调用 `malloc/calloc` 等
结束	离开对应块	调用 `free/realloc` 等
责任	注意作用域与生命周期	额外设计释放责任

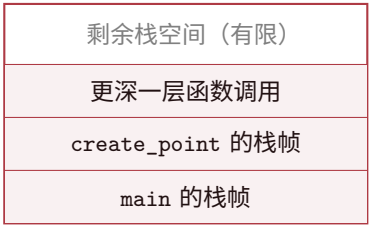
选择依据是大小和生命周期需求，不是“heap 一定比 stack 好”。

立志成才 报国裕民

先认识 Stack 与 Heap：一层层调用，与一块块申请

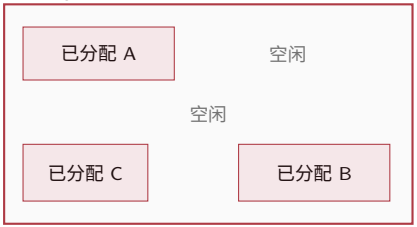
上海科技大学 ShanghaiTech University

Stack（栈）：一层层栈帧



- 调用进入一层，返回退出一层。
- 局部对象通常随作用域自动管理。

Heap（堆）：相对较大的分配空间



- 可以想成“比较散的一块空间”，有空闲块，也有已分配块。
- 要用就 `malloc` 申请，不用了 `free`；释放不必按调用顺序。

Stack 按调用层次组织；Heap 按需申请，使用完毕后释放。

立志成才 报国裕民

运行时才知道 n，直接写 int a[n] 可以吗？


```c
/* Suppose n > 0; headers omitted. */
void show_first(size_t n) {
    int a[n];       /* VLA: size known at run time */
    a[0] = 42;
    printf("%d\n", a[0]);
}                   /* a stops existing here */
/* Reading example only: VLAs are banned in homework. */
```

- 这里的 n 不是整型常量表达式，a 是 **variable-length array（VLA，变长数组）**。
- 大小在运行时确定，但它是 automatic 局部对象，通常占用 **stack** 空间。
- 离开作用域后生命周期结束，也不能通过改变 n 来扩容。

VLA 不是 malloc 式的动态分配。本课程所有作业都禁止使用 VLA。

为什么作业禁用 VLA？空间不足时能否处理？

malloc：可以检查分配失败

```c
int *a = malloc(n * sizeof(int));
if (a == NULL) {
    return; /* handle failure */
}
/* use a, then free(a) */
```

- 空间申请失败时返回 NULL。
- 可以报告错误、清理资源，再结束当前操作。

VLA：没有 NULL 失败返回机制

```c
/* n comes from input */
int a[n];
/* Too large for the stack? */
/* No NULL result to check. */
```

- n 太大，可能直接耗尽栈空间，导致爆栈或崩溃。
- 没有类似 malloc 的返回值可检查，存在安全风险。

类比：发射中心要建在酒泉，而不是紫禁城

大小不可控的数组，不要挤进空间有限、承担函数调用的栈里。

所有作业禁止使用 VLA。运行时分配用 malloc 等接口，并检查失败。

立志成才 报国裕民

create_point 续集：让对象活到函数返回之后

创建对象：申请、检查、初始化

```c
#include <stdio.h>
#include <stdlib.h>
struct Point { int x; int y; };
struct Point *create_point(int x, int y) {
    struct Point *point;
    point = malloc(sizeof(struct Point));
    if (point == NULL) return NULL;
    point->x = x;
    point->y = y;
    return point;
}
```

调用者：使用完毕后释放

```c
int main(void) {
    struct Point *p = create_point(10, 15);
    if (p == NULL) return EXIT_FAILURE;
    struct Point new_point = {20, 30};
    p->x = 99;
    printf("%d %d\n", p->x, p->y);
    printf("%d %d\n",
           new_point.x, new_point.y);
    free(p);
    return 0;
}
```

这次返回的是动态分配对象的地址，main 可以继续使用它。

输出两行：99 15、20 30。new_point 是另一个对象，不会被这次有效写入修改。

同一个故事：改了什么，生命周期为什么不同？



create_point 的局部指针 point
函数返回后已经结束生命周期

- 错误版：结构体本身是局部对象，返回时就不存在了。
- malloc 版：局部的是**指针变量**；动态分配的**结构体对象**继续存在。
- 新责任：调用者必须检查 NULL，并在不再使用时 free(p)。

立志成才 报国裕民

动态分配的四个核心接口


```c
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t count, size_t size);
void *realloc(void *ptr, size_t new_size);
void free(void *ptr);
```

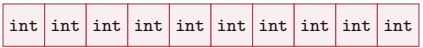
- `size_t`：用于表示大小的无符号整数类型。
- `malloc`：申请指定字节数；`calloc`：申请数组并将所有字节清零。
- `realloc`：调整已有分配大小；`free`：结束该次分配。
- 本节示例均使用**正数大小**，避免零大小分配的特殊规则。

立志成才 报国裕民

sizeof：明确写出要申请的对象类型


```c
int *p = malloc(10 * sizeof(int)); /* 10 ints */
/* NOT: malloc(10 * sizeof(int *)) */
```

10 * sizeof(int)：10 个 int 对象



sizeof(int *)：仅 1 个指针变量本身



（64 位平台通常为 8 字节，与所指对象大小无关）

- sizeof(int)：一个 int 对象占多少字节；sizeof(struct Point)：一个结构体对象占多少字节。
- C 中 void * 可转换成对象指针，malloc 返回值不需要强制类型转换。

申请 10 个 int，用 10 * sizeof(int)；不要把所指对象的大小写成指针的大小。

立志成才 报国裕民

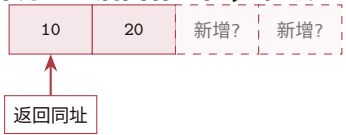
calloc：按元素数量申请，并把所有位清零


```c
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    size_t n = 5;
    int *a = calloc(n, sizeof(int));
    if (a == NULL) return EXIT_FAILURE;
    printf("%d %d\n", a[0], a[4]);
    free(a);
    return 0;
}
```

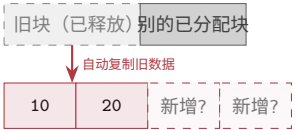
- `calloc(count, size)` 按元素个数申请空间，并把所有字节清零。
- 这里的 `int` 数组初始值全是 0，输出 `0 0`。

realloc：原地扩容，还是搬家？

情况 A：后面有空位，原地延长



情况 B：后面被挡住，搬去新块



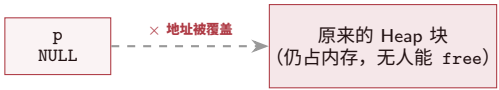
- 成功：保留原有数据，新增区域未初始化；**旧指针及指向旧块的别名全部失效**。
- 失败且请求大小非零：返回 NULL，**原分配仍然有效**（不会自动释放）。

立志成才 报国裕民

为什么不能直接 p = realloc(p, ...)?


```c
/* p owns an allocated block */
p = realloc(p, 100 * sizeof(int));
if (p == NULL) {
    /* Where is the old block now? */
}
```

- 失败返回 NULL，直接赋给 p 会覆盖旧地址。
- 原内存块还活着，但钥匙丢了 → **memory leak**。
- 正确做法：先用临时指针 tmp 接住返回值。



立志成才 报国裕民

安全扩容：先接住，再更新


```c
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    int *p = malloc(2 * sizeof(int));
    if (p == NULL) return EXIT_FAILURE;
    p[0] = 10;
    p[1] = 20;
    int *tmp = realloc(p, 4 * sizeof(int));
    if (tmp == NULL) {
        free(p);
        return EXIT_FAILURE;
    }
    p = tmp;
    p[2] = 30;
    p[3] = 40;
    printf("%d %d %d %d\n", p[0], p[1], p[2], p
     [3]);
    free(p);
    return 0;
}
```

失败：原块仍有效，清理后退出。成功：更新指针，初始化新增元素。

Memory 区域与对象的存储期

先猜输出：每次进入函数，变量都会重新开始吗？


```c
#include <stdio.h>
int global = 10;
void tick(void) {
    int local = 0;
    static int count = 0;
    ++local;
    ++count;
    ++global;
    printf("%d %d %d\n", local, count, global);
}
int main(void) {
    tick();
    tick();
    return 0;
}
```

答案：作用域与生命周期是两件事

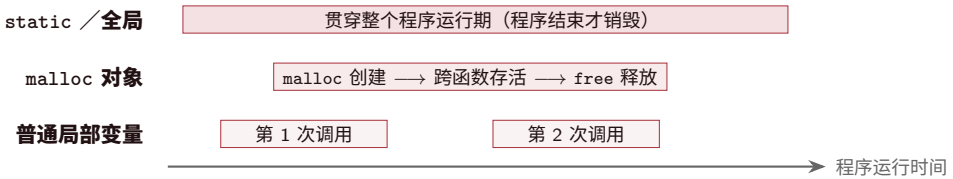
- 第一次：`1 1 11`；第二次：`1 2 12`。
- `local`：每次进入函数体，开始新的 automatic 对象生命周期。
- `count`：具有 static storage duration，整个程序执行期间都存在。
- `global`：同样具有 static storage duration，值不会在函数返回时重置。

`count` 只能在函数体内用名字访问，但它不会随着函数返回而消失。

立志成才 报国裕民

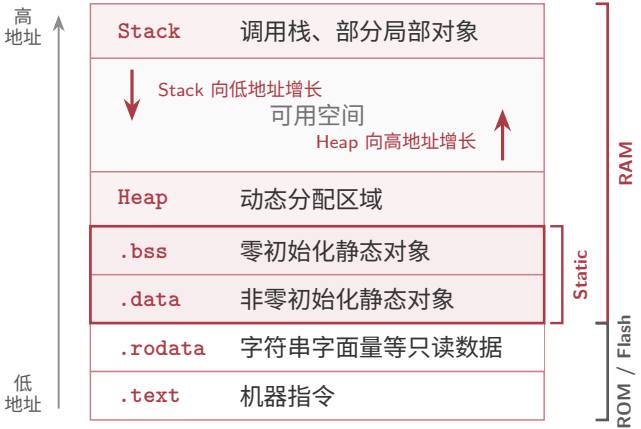
Scope（作用域）与 Lifetime（生命周期）

- **Scope（作用域）**：源码中，哪些位置能用这个名字？
- **Lifetime（生命周期）**：对象在程序运行期间何时存在、何时可以访问？



立志成才 报国裕民

Memory layout：一种典型的程序存储布局



RAM 与 ROM ／ Flash

- **RAM**：运行时可写数据，包括 Stack、Heap 与 Static 区。
- **ROM ／ Flash**：保存代码和只读数据，如.text 与.rodata。
- **Static 区**：.bss 与.data 中的对象，存储期贯穿程序。

嵌入式系统中，.data 的初值可保存在 Flash，启动时复制到 RAM；.bss 在启动时清零。

图示高地址在上，箭头为常见增长方向；具体地址、介质与布局由平台决定。

立志成才 报国裕民

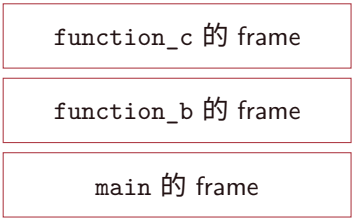
这些对象都在哪？没写初值时会怎样？


```c
int global_uninit;
int global_init = 7;
static int static_global = 8;
void demo(void) {
    static int static_local = 9;
    static int static_zero;
    int local;
    const char *p = "hello";
}
```

对象／数据	初始值	通常所在区域
global_uninit	0	.bss（Static）
global_init	7	.data（Static）
static_global	8	.data（Static）
static_local	9	.data（Static）
static_zero	0	.bss（Static）
local	不确定值	Stack ／寄存器
p	字面量的地址	Stack ／寄存器
"hello"	字符及结尾零	.rodata

- **默认初始化行为，看存储期：**没写初值的全局／ static 对象会**零初始化**；普通局部 `int local;` 不会自动置零，**先赋值，再读取**。
- 非零初值通常进入.data；显式 = 0 也通常进入.bss。static 局部对象的生命周期贯穿程序。
- **p 保存地址，"hello" 是所指数据；字符串字面量不能修改。**

Stack frame：函数返回后发生了什么？



- 函数调用通常建立 stack frame（栈帧）。
- 栈帧可能保存局部变量、返回地址与部分参数。
- 参数也可能通过寄存器传递。
- 函数返回后，普通局部对象生命周期结束。

内存里的旧字节暂时没被覆盖，也不代表原来的对象还活着。

立志成才 报国裕民

谁负责 free：指针、别名与所有权

free：释放的是分配，不是指针变量

上海科技大学
ShanghaiTech University

```c
int *p = malloc(sizeof(int));
if (p == NULL) return EXIT_FAILURE;
*p = 42;
free(p);
p = NULL;
```

1. free(p); 之后



2. p = NULL; 之后



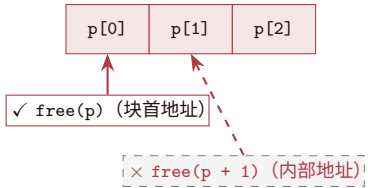
- free 结束该块的分配，但**不会自动把 p 改为 NULL**。
- free(NULL) 安全无操作；主动置 p = NULL 可避免误用悬空指针。

立志成才 报国裕民

哪几种地址能传给 free?


```c
int global;
void f(void) {
    int local = 0;
    int *p = malloc(3 * sizeof(int));
    if (p == NULL) return;
    /* free(&local);  wrong */
    /* free(&global); wrong */
    /* free(p + 1);   wrong */
    free(p);          /* correct */
    /* free(p);       wrong: already freed */
}
```

整租一套房，退房必须交整套房钥匙

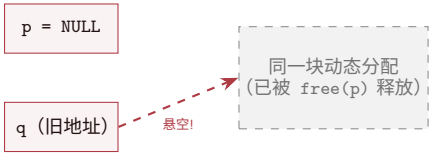


只能传 NULL，或与仍有效的动态分配返回值完全匹配的块首指针。

局部变量（Stack）、全局变量（Static）、块内部偏移地址（p + 1）和已释放的地址都不能传给 free。

两个指针指向同一块分配：p = NULL 后，q 安全吗？


```c
int *p = malloc(sizeof(int));
if (p == NULL) return EXIT_FAILURE;
*p = 42;
int *q = p;
free(p);
p = NULL;
/* printf("%d\n", *q); wrong */
/* free(q);            wrong */
q = NULL;
```



- `int *q = p;` 只复制地址，没有复制对象。
- `p = NULL` 不会自动更新别名 `q`。
- **释放次数看分配次数，不看有几个指针。**

Memory leak：指针没了，堆上的分配还在


```c
void f(void) {
    int *p = malloc(100 * sizeof(int));
    if (p == NULL) return;
    p[0] = 42;
} /* p disappears; nobody kept the allocation
     */
```



- p 是 automatic 局部指针，离开函数 f 后自动销毁。
- 但 Heap 里的对象不会自动释放；丢失唯一地址后，程序再也无法 free 它。

“指针生命周期结束”不等于“所指对象自动释放”。

立志成才 报国裕民

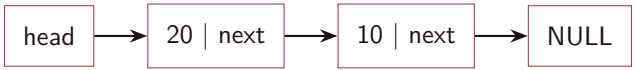
回看 create_point：约定谁负责 free


```c
int main(void) {
    struct Point *p = create_point(10, 15);
    if (p == NULL) return EXIT_FAILURE;
    struct Point new_point = {20, 30};
    p->x = 99;
    printf("%d %d\n", p->x, p->y);
    printf("%d %d\n",
           new_point.x, new_point.y);
    free(p);
    return 0;
}
```

- `create_point` 把动态对象交给调用者（转移 **ownership**）；返回 `NULL` 表示失败。
- `main` 接管返回的指针，使用结束后负责 `free(p)`。
- 局部结构体 `new_point` 随作用域自动结束生命周期，不能 `free(&new_point)`。

链表：每个节点都是一次独立分配


```c
struct Node {
    int data;
    struct Node *next;
};
```



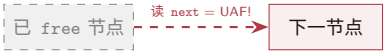
释放 head 指向的节点，不会自动释放 next 指向的节点。

立志成才 报国裕民

释放链表：为什么必须先保存 next，再 free？

```c
void free_list(struct Node *head) {
    while (head != NULL) {
        struct Node *next = head->next;
        free(head);
        head = next;
    }
}
/* caller: free_list(head); head = NULL; */
```

× 错误：先 free(head)，桥已经炸断



✓ 正确：先用 next 探路，再拆当前桥



先保存 next 指针，再 free 当前节点；否则读取已释放节点的 next 就是 use-after-free。

内存错误小结：先问哪条规则被破坏了

错误类型	破坏了哪条规则	本节典型现场
Out-of-bounds（越界）	下标超出有效分配范围	申请 `n` 个元素却访问 `a[n]`
Uninitialized read	读取前未写入确定值	局部 `int local;` 或 `malloc` 后直接读
Dangling / UAF	所指对象生命周期已结束	返回 `&point`、先 `free(head)` 再读 `next`
Double free（重复释放）	同一块分配被 `free` 两次	别名 `free(p)` 后又执行 `free(q)`
Memory leak（内存泄漏）	丢失地址，遗漏释放路径	`p = realloc(p, ...)` 失败覆盖原指针

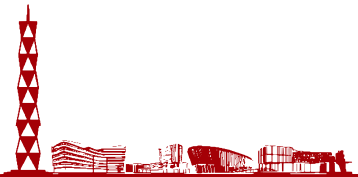
这些问题没有按 stack、heap、global 划分的固定严重程度。

越界、use-after-free、double free 等问题，可结合**上节课的 AddressSanitizer** 检查。

带走这四个问题

- **在哪里？**区分指针变量与所指对象；布局是实现模型。
- **活多久？**区分 scope、automatic/static/allocated 存储期。
- **能访问吗？**检查初始化、边界、NULL 与生命周期。
- **谁释放？**每次分配都有明确责任；失败路径也要清理。

地址还在，不代表对象还在；函数返回，也不代表动态对象被释放。



Thank You

课后练习：指针与宏

课后练习一：判断对错，并说明理由

- A. 普通局部变量一定活到整个程序结束。
- B. 动态分配的对象可以跨越一次函数调用继续存在。
- C. 给指针赋值 `NULL` 会自动释放它原先指向的块。
- D. 全局变量可以通过 `free(&global)` 释放。
- E. 对一个仍有效的分配，多个指针可以同时指向它。

先独立判断，再用“对象的生命周期”解释。

课后练习一答案：B、E 正确

- A 错：普通 automatic 局部对象在离开对应块时结束生命周期。
- B 对：动态分配对象的存在不依赖创建它的函数是否已经返回。
- C 错：赋值改变的是指针变量；原分配可能因此泄漏。
- D 错：全局对象不来自动态分配，不能这样释放。
- E 对：多个别名共享同一个对象，但必须协调访问与释放责任。

课后练习二：移动形参，会改变调用者吗？


```c
#include <stdio.h>
void move(int *p) {
    p = p + 1;
}
int main(void) {
    int a[2] = {4, 6};
    int *p = &a[0];
    move(p);
    printf("%d %d\n", *a, *p);
    return 0;
}
```

输出是 4 4，还是 4 6？画出调用前后的两个 p。

课后练习二答案：输出 4 4

- C 函数参数按值传递，`move` 得到地址值的副本。
- 函数里的 `p = p + 1` 只改变形参，不改变调用者的 `p`。
- `*a` 仍然是 `a[0]`；调用者的 `p` 也仍指向 `a[0]`。

```
1 void move_caller(int **p) {
2     *p = *p + 1;
3 }
4 /* caller: move_caller(&p); */
```

改对象内容传对象地址；改调用者的指针，就传那个指针的地址。

立志成才 报国裕民

课后练习三：加 2，同时返回原值

补全函数：调用后 `x` 加 2，返回修改前的值。

```
int add_two( /* fill in */ ) {
    /* one statement per line; no comma operator */
}
/* int x = 0; int old = add_two(&x); */
/* expected: old == 0, x == 2 */
```

- 调用者希望自己的 `x` 被修改，需要传什么？
- 如何保存修改前的值？
- 前提：`x` 指向有效 `int` 对象，原值加 2 不发生有符号溢出。

课后练习三答案：先保存，再修改


```c
#include <stdio.h>
int add_two(int *x) {
    int old = *x;
    *x += 2;
    return old;
}
int main(void) {
    int x = 0;
    int old = add_two(&x);
    printf("%d %d\n", old, x);
    return 0;
}
```

输出 0 2。**指针让函数找到调用者的对象；它本身仍然是按值传递。**

课后练习四：宏不会替你补括号


```c
#define MUL(a, b) a * b
/* What is 16 / MUL(1 + 1, 3 - 1)? */
```

- 先做 macro expansion（宏展开），再按运算符优先级计算。
- 展开后是 `16 / 1 + 1 * 3 - 1`。
- 它和 `16 / ((1 + 1) * (3 - 1))` 一样吗？
- 怎么修改 `MUL`，让这一次计算得到期望结果？

立志成才 报国裕民

课后练习四答案：原式为 18，修复后为 4


```c
#include <stdio.h>
#define BAD_MUL(a, b) a * b
#define MUL(a, b) ((a) * (b))
int main(void) {
    printf("%d\n", 16 / BAD_MUL(1 + 1, 3 - 1));
    printf("%d\n", 16 / MUL(1 + 1, 3 - 1));
    return 0;
}
```

- 参数各自加括号，保证每个实参被当作完整表达式。
- 整体再加括号，保证整个展开结果参与外围运算。

课后练习五：用条件运算符写 MIN


```c
#define MIN(a, b) /* fill in */
/* MIN(3, 5) should be 3 */
/* condition ? value_if_true : value_if_false */
```

- 比较 a 与 b，较小时选择哪一个?
- 每个参数和整个表达式都要考虑括号。
- 追问：MIN(i++, j) 会让 i 增加几次?

立志成才 报国裕民

课后练习五答案：括号解决不了重复求值


```c
#include <stdio.h>
#define MIN(a, b) ((a) < (b) ? (a) : (b))
static inline int min_int(int a, int b) {
    return a < b ? a : b;
}
int main(void) {
    int i = 1;
    int result = MIN(i++, 5);
    printf("%d %d\n", result, i);
    printf("%d\n", min_int(3, 5));
    return 0;
}
```

这里先比较 `1 < 5`，再求真分支 `i++`，输出 `2 3`。

避免把带副作用的表达式传给这类宏；普通整数比较可用函数。

课后练习六：打印类型名和 sizeof 结果

写一个宏 PS(type)，例如 PS(long long)。

```
1 /* Typical output: Size of long long: 8 */
2 #define PS(type) /* fill in */
```

- sizeof 的结果类型是 size_t，使用%zu。
- 如何把宏参数 long long 变成字符串？提示：#type。
- **示例里的 8 只是常见结果，不是所有平台的固定答案。**

立志成才 报国裕民

课后练习六答案：# 把参数字符串化


```c
#include <stdio.h>
#define PS(type) \
    printf("Size of " #type ": %zu\n", sizeof(type))
int main(void) {
    PS(long long);
    PS(int);
    return 0;
}
```

- #type 是 stringification（字符串化）。
- 相邻字符串字面量在翻译时会拼接。
- PS(int); 相当于打印"Size of " "int" ": %zu n" 并传入 sizeof(int)。

课后综合题：这段代码有哪些问题？


```c
int *p = malloc(2 * sizeof(int));
int *q = p;
p[0] = 10;
p[2] = 20;
free(p);
p = NULL;
printf("%d\n", q[0]);
free(q);
```

假定已包含头文件，代码位于 `main` 中。

逐行分析；不要尝试预测未定义行为后的输出。

立志成才 报国裕民

课后综合题答案：沿着分配与生命周期找问题

- 没有检查 `malloc` 是否返回 `NULL`。
- 只申请两个 `int`，合法下标是 0 和 1；`p[2]` 越界。
- `free(p)` 之后 `q` 也失去有效所指对象；读取 `q[0]` 是 use-after-free。
- 最后 `free(q)` 重复释放同一分配。
- `p = NULL` 只更新 `p`，无法修复 `q`。

正确顺序：检查分配 → 初始化 → 范围内访问 → 统一释放 → 停止使用旧别名。

立志成才 报国裕民