C 讲义

中文版

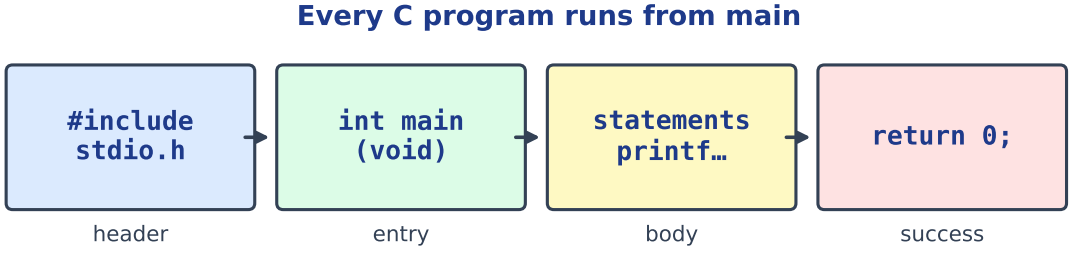
Contents

1 C 基础

1.1 main、printf 与函数

每个 C 程序都从 `main` 开始。`#include <stdio.h>` 引入一个头文件 (header), 这样你就能用 `printf` 来打印。函数 (function) 是一个有名字、可以调用的代码块;`\n` 表示换行。`main` 返回 `0` 表示” 成功”。

```c
#include <stdio.h>

void greet(void) {
    printf("Hello, world!\n");
}

int main(void) {
    greet();
    printf("I am learning C.\n");
    return 0;
}
```



Exam tip: #include for libraries; main returns 0 for success; end statements with ;

Every C program runs from main after including headers

1.2 变量、类型与算术

C 要求每个变量都有类型:`int`(整数,integer)、`double`(浮点数,floating-point)、`char`(单个字符)。`printf` 为每个值使用一个格式说明符 (format specifier)——`%d`、`%f`、`%c`。两个整数之间的 `/` 是整数除法;`%` 是余数 (remainder)。

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

```c
int main(void) {
    int n = 17;
    double pi = 3.14;
    char grade = 'A';
    printf("%d %.2f %c\n", n, pi, grade);   // 17 3.14 A
    printf("%d %d\n", 7 / 2, 7 % 2);        // 3 1
    return 0;
}
```

格式串同时驱动打印和读取:

说明符	类型	示例输出
%d	int	17
%f	double(%.2f = 保留 2 位小数)	3.14
%c	char	A
%s	字符串	Mei
%zu	大小 (来自 sizeof / strlen)	3
%x	int, 按 16 进制打印	ff

scanf("%d", &n) 用同样的说明符读取键盘输入 (注意 &)。它的同伴 sscanf 从字符串里解析值, 所以在哪里都能运行:

```c
#include <stdio.h>

int main(void) {
    char line[] = "Mei 88";
    char name[20];
    int score;
    sscanf(line, "%19s %d", name, &score);   // 从文本解析出值
    printf("%s scored %d\n", name, score);   // Mei scored 88
    return 0;
}
```

1.3 注释与风格

注释 (comment) 是给人看的说明; 编译器会忽略它。单行用 //, 整块用 /* ... */。良好的缩进 (indentation) 和清晰的命名让代码易读。

```c
#include <stdio.h>

int main(void) {
    // 单行注释
    /* 一段
       块注释 */
```

```c
    int total = 3 + 4;        // 清晰的命名有帮助
    printf("%d\n", total);    // 7
    return 0;
}
```

常见错误

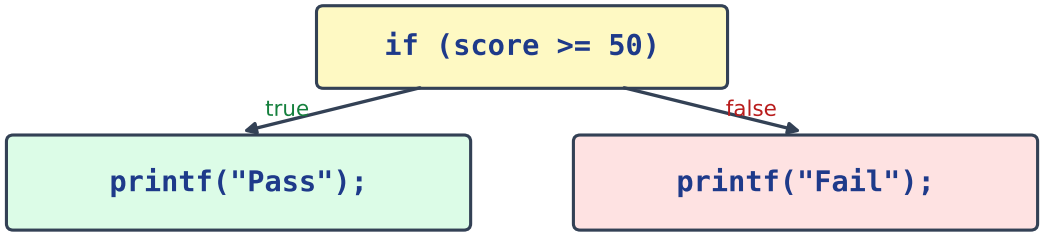
- 每条语句以 ; 结尾,main 应返回 int(return 0;)。
- printf 需要格式串:%d 对应 int,%f 对应 double,\n 换行。
- 格式不匹配, 比如对 double 用 %d, 会打印乱码 —— 类型要对上。

2 选择

2.1 if / else

`if` 在条件 (condition) 为真时运行一个代码块。C 默认没有真正的布尔值 (boolean) 类型:0 为假, 任何非零值为真。用 `else if` 和 `else` 串联多个选择。用 `==`、`!=`、`<`、`>`、`<=`、`>=` 比较。

```c
#include <stdio.h>

int main(void) {
    int score = 72;
    if (score >= 90) {
        printf("A\n");
    } else if (score >= 60) {
        printf("Pass\n");
    } else {
        printf("Fail\n");
    }
    return 0;
}
```



Exam tip: braces { } group the body; only one branch runs; else is optional

if chooses the true branch; else the false branch

2.2 switch

`switch` 根据一个整数值选择一个 `case`。每个 `case` 都需要一个 `break` 来跳出 (jump out)—— 否则 C 会” 贯穿” 到下一个 case。当没有匹配时运行 `default`。

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

```c
int main(void) {
    int day = 3;
    switch (day) {
        case 1: printf("Mon\n"); break;
        case 2: printf("Tue\n"); break;
        case 3: printf("Wed\n"); break;
        default: printf("Other\n");
    }
    return 0;
}
```

常见错误

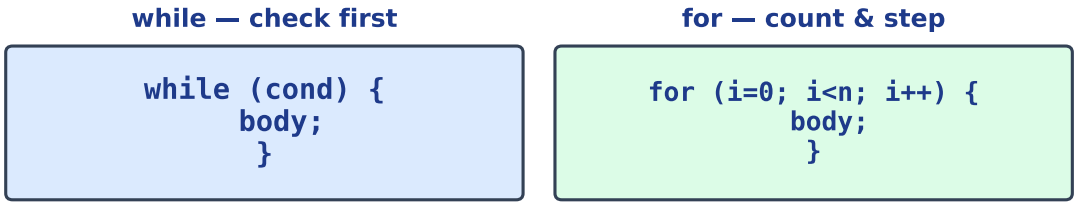
- `if (x = 5)` 是赋值, 永远为真; 比较要用 `==`。
- 每个 `switch` 分支都要 `break;`, 否则会掉进下一个。
- 在条件里 0 是假, 任何非零值都是真。

3 循环

3.1 while 与 for 循环

循环 (loop) 重复运行一个代码块。`while` 循环在条件为真时一直运行。`for` 循环把初始化、判断和自增 (increment,`i++`) 打包到一行 —— 在知道重复次数时最好用。

```c
#include <stdio.h>

int main(void) {
    int i = 1;
    while (i <= 3) {
        printf("%d ", i);
        i++;
    }
    printf("\n");                       // 1 2 3
    for (int j = 0; j < 5; j++) {
        printf("%d ", j);
    }
    printf("\n");                       // 0 1 2 3 4
    return 0;
}
```



Exam tip: both need a way to end; for packs init, test, step; while is open-ended

while checks a condition; for counts with a step

3.2 累加

一个常见模式: 让累加器 (accumulator) 从 `0` 开始, 然后在循环里不断累加。同样的思路可以用来计数或求累计总和。

```c
#include <stdio.h>

int main(void) {
    int total = 0;
    for (int i = 1; i <= 5; i++) {
        total += i;            // 1 + 2 + 3 + 4 + 5
    }
    printf("%d\n", total);     // 15
    return 0;
}
```

3.3 嵌套循环与图案

一个循环里面再放一个循环就是嵌套 (nested) 循环。外层循环每走一步，内层循环都会完整地跑完 —— 非常适合网格和图案。

```c
#include <stdio.h>

int main(void) {
    for (int row = 0; row < 3; row++) {
        for (int col = 0; col < 3; col++) {
            printf("*");
        }
        printf("\n");
    }
    return 0;
}
```

常见错误

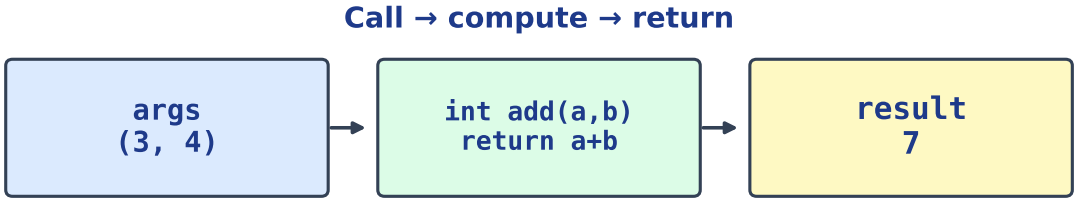
- `for (i = 0; i < n; i++)` 运行 `n` 次;`for (...)` 后紧跟分号会得到空循环。
- 条件永远不为假的 `while` 会无限循环。
- 计数器 (`int i`) 要在 `for` 头部之前或之中声明。

4 函数

4.1 参数与返回值

函数接收参数 (parameter, 即输入), 并交回一个返回值 (return value)。返回类型写在最前面 (`int`、`double`……);`void` 表示不返回任何东西。

```c
#include <stdio.h>

int square(int x) {          // x 是一个参数
    return x * x;            // 交回一个值
}

int main(void) {
    int r = square(5);
    printf("%d\n", r);       // 25
    return 0;
}
```



Exam tip: declare the return type; parameters need types; return ends the function

Parameters in; return value out

4.2 函数原型与作用域

C 从上往下读, 所以函数在被调用之前必须先被认识。函数原型 (prototype)—— 就是头部那一行加上 `;` —— 提前声明它, 这样你就能把 `main` 放在最前面。变量的作用域 (scope) 是它所在的代码块: 它是局部 (local) 的, 代码块结束时就消失。

```c
#include <stdio.h>

int add(int a, int b);       // 函数原型: 在使用前声明
```

```c
int main(void) {
    printf("%d\n", add(3, 4));   // 7
    return 0;
}

int add(int a, int b) {     // 定义放在后面
    int sum = a + b;        // sum 是 add 的局部变量
    return sum;
}
```

常见错误

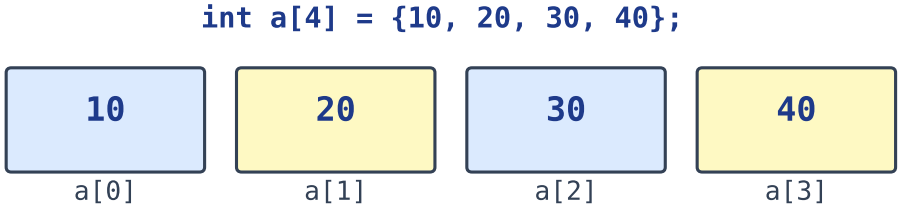
- 在定义之前使用的函数, 需要在 `main` 上方写原型 (prototype)。
- 参数是**按值**传递 —— 函数内的修改不影响调用者, 除非传入指针。
- 在函数内声明的变量是它的局部变量。

5 数组

5.1 一维数组

数组 (array) 保存同一类型的多个值。索引 (index) 从 0 开始。C **不会**存储数组的长度, 所以有一个常用技巧来计算元素 (element) 个数:`sizeof(a) / sizeof(a[0])`。

```c
#include <stdio.h>

int main(void) {
    int scores[3] = {88, 71, 95};
    printf("%d\n", scores[0]);     // 88
    scores[1] = 100;
    printf("%d\n", scores[1]);     // 100
    int n = sizeof(scores) / sizeof(scores[0]);
    printf("%d\n", n);             // 3
    return 0;
}
```



Exam tip: indices start at 0; a[i] is one element; length is fixed at creation

Array slots are indexed from 0

5.2 数组算法

用 `for` 循环遍历 (traverse) 数组, 求最大值、总和或计数。始终从 0 循环到 `n - 1`。

```c
#include <stdio.h>

int main(void) {
    int a[] = {3, 9, 2, 7};
    int n = sizeof(a) / sizeof(a[0]);
    int max = a[0], total = 0;
    for (int i = 0; i < n; i++) {
```

```
        if (a[i] > max) max = a[i];
        total += a[i];
    }
    printf("%d %d\n", max, total);   // 9 21
    return 0;
}
```

5.3 二维数组

二维数组是由行 (row) 和列 (column) 组成的网格:`grid[row][col]`。用两个嵌套循环访问每个单元格。

```
#include <stdio.h>

int main(void) {
    int grid[2][3] = {{1, 2, 3}, {4, 5, 6}};
    printf("%d\n", grid[1][2]);    // 6
    for (int r = 0; r < 2; r++) {
        for (int c = 0; c < 3; c++) {
            printf("%d ", grid[r][c]);
        }
    }
    printf("\n");                  // 1 2 3 4 5 6
    return 0;
}
```

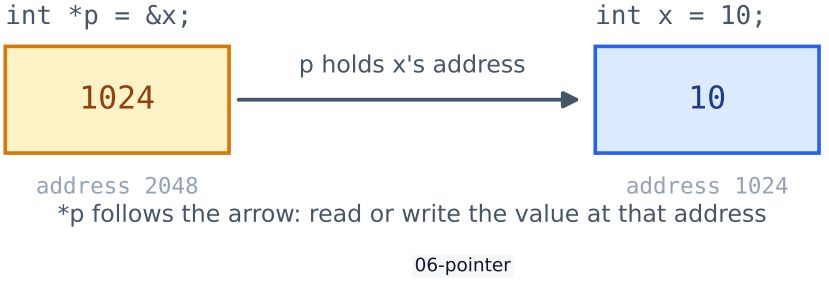
常见错误

- C **不**检查数组越界: 对大小为 `n` 的数组写 `a[n]` 是未定义行为。
- 合法下标是 `0` 到 `n - 1`。
- 数组名是指向首元素的指针;`sizeof` 只在数组自身的作用域里才给出大小。

6 指针

6.1 &、* 与按指针传递

指针 (pointer) 存储一个变量的地址 (address)。`&x` 给出 `x` 的地址;`*p` 对指针解引用 (dereference)—— 读取或写入存放在那里的值。传递指针能让函数修改调用者的变量 (按指针传递,pass-by-pointer)。



指针存储一个地址; 解引用就是沿着箭头找到那个值

```c
#include <stdio.h>

void addOne(int *p) {      // p 保存一个地址
    *p = *p + 1;           // 修改那个地址处的值
}

int main(void) {
    int x = 10;
    int *ptr = &x;         // & 取得 x 的地址
    printf("%d\n", *ptr); // * 读取值:10
    addOne(&x);
    printf("%d\n", x);     // 11 -- 通过指针被修改了
    return 0;
}
```

最经典的用法是交换函数。按值传递只是拷贝 —— 交换会丢失。传指针才能让函数够到调用者的变量:

```c
#include <stdio.h>

void swap_values(int a, int b) {      // 拷贝: 调用者看不到变化
```

```
    int t = a; a = b; b = t;
}

void swap_pointers(int *a, int *b) {  // 地址：交换是真的
    int t = *a; *a = *b; *b = t;
}

int main(void) {
    int x = 1, y = 2;
    swap_values(x, y);
    printf("%d %d\n", x, y);   // 1 2  (没变!)
    swap_pointers(&x, &y);
    printf("%d %d\n", x, y);   // 2 1  (交换了)
    return 0;
}
```

数组名的行为就像指向第一个元素的指针，而指针运算 (pointer arithmetic) 按整个元素来步进：

```
#include <stdio.h>

int main(void) {
    int a[] = {10, 20, 30};
    int *p = a;                 // 等同于 &a[0]
    printf("%d\n", *p);         // 10
    printf("%d\n", *(p + 2));   // 30 -- 等同于 a[2]
    return 0;
}
```

- NULL 是一个”什么都不指”的指针 —— 解引用之前先检查它。

常见错误

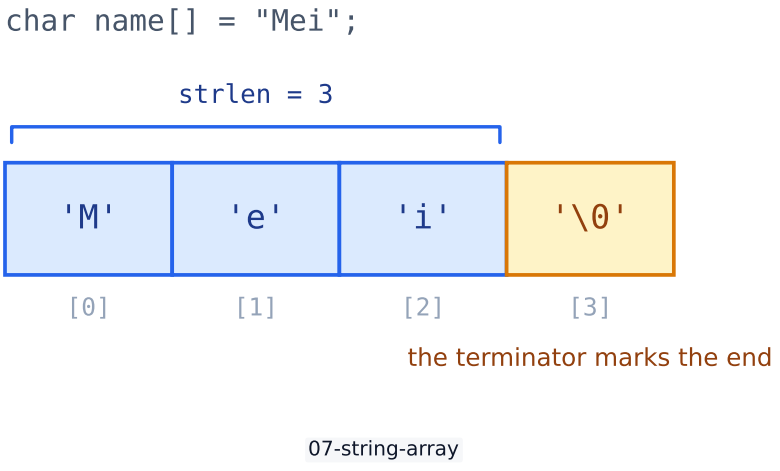
- &x 是 x 的地址;*p 是 p 指向的值。
- 绝不要对未初始化或 NULL 的指针用 *p—— 会崩溃或破坏内存。
- 要改调用者的变量，就传它的地址，并通过指针写入。

7 字符串

7.1 字符数组与空终止符

C 的字符串 (string) 是一个 `char` 数组。每个字符串都以一个隐藏的空终止符 (null terminator)`'\0'` 结尾，它标记字符串在哪里结束。`strlen`(来自 `<string.h>`) 统计到那个 `'\0'` 为止的字符数。用 `%s` 打印整个字符串，用 `%c` 打印单个字符 (character)。

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

int main(void) {
    char name[] = "Mei";            // 'M', 'e', 'i', '\0'
    printf("%s\n", name);           // Mei
    printf("%zu\n", strlen(name));  // 3 (到 '\0' 为止)
    printf("%c\n", name[0]);        // M
    return 0;
}
```



C 字符串是一个 *char* 数组，以隐藏的空终止符结尾

你可以通过循环到 `'\0'` 为止来遍历一个字符串。

```c
#include <stdio.h>

int main(void) {
    char word[] = "banana";
    int count = 0;
    for (int i = 0; word[i] != '\0'; i++) {
        if (word[i] == 'a') count++;
```

```
    }
    printf("%d\n", count);   // 3
    return 0;
}
```

7.2 string.h 函数库

`#include <string.h>` 提供日常要用的字符串工具:

函数	作用
`strlen(s)`	长度, 不算结束符
`strcpy(dst, src)`	复制 ——dst 必须够大
`strcat(dst, src)`	把 src 接到 dst 末尾
`strcmp(a, b)`	比较: 相等为 0, 否则为负 / 正

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

int main(void) {
    char full[40];
    strcpy(full, "Mei");           // full 现在是 "Mei"
    strcat(full, " Chen");         // 拼接 -> "Mei Chen"
    printf("%s (%zu)\n", full, strlen(full));       // Mei Chen (8)
    printf("%d\n", strcmp("apple", "apple") == 0);  // 1 (相等)
    return 0;
}
```

- `strcmp` 相等时返回 0, 所以判断写成 `strcmp(a, b) == 0`—— 绝不能写 `a == b`。
- 目标数组要装得下结果, **再加上**结束符。

常见错误

- C 字符串要多留一个字节给结束符 `\0`:5 个字母的词需要 `char[6]`。
- 复制和比较字符串用 `strcpy` / `strcmp`, 不要用 `=` / `==`。
- 少了 `\0`,`printf("%s", ...)` 会读过文本末尾。

8 结构体

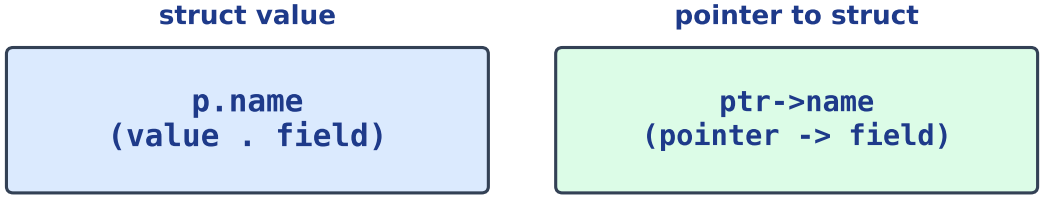
8.1 struct、typedef、. 与 ->

结构体 (struct) 把相关的值组合成一个类型。其中每个值是一个成员 (member)。`typedef` 给结构体起一个短名字，这样你就能写 `Dog` 而不用写 `struct Dog`。对结构体值用 `.`，但对指向结构体的指针用 `->`。

```c
#include <stdio.h>

typedef struct {
    char name[20];
    int age;
} Dog;

int main(void) {
    Dog d = {"Rex", 3};
    printf("%s is %d\n", d.name, d.age);   // Rex is 3   (对值用 .)

    Dog *p = &d;
    p->age = 4;                            // 对指针用 ->
    printf("%s is %d\n", d.name, d.age);   // Rex is 4
    return 0;
}
```

常见错误

- 对结构体值用 `.`，对结构体指针用 `->`。
- `typedef` 让你声明时省去 `struct` 这个词；没有它就要写 `struct Point p;`。
- 把一个结构体赋给另一个会复制它的所有字段。



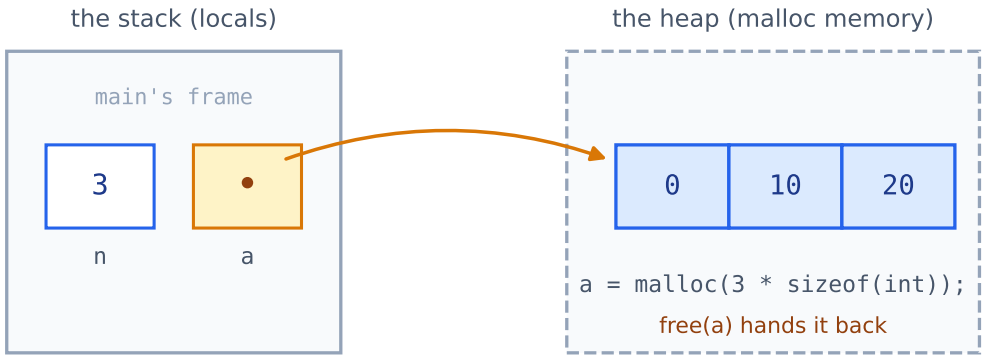
Exam tip: use . on a struct; use -> on a pointer (same as (*ptr).field)

. for struct values; -> for pointers to structs

9 动态内存

9.1 malloc、free 与 realloc

`malloc` 在运行时从堆 (heap) 上分配 (allocate) 内存，并返回一个指向它的指针。`realloc` 调整这块内存的大小;`free` 把它归还。每个 `malloc` 都需要一个对应的 `free`，否则就会发生内存泄漏 (memory leak)。这些函数在 `<stdlib.h>` 里。



09-stack-heap

局部变量住在栈上;*malloc* 的内存住在堆上，直到你 *free* 它

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

int main(void) {
    int n = 3;
    int *a = malloc(n * sizeof(int));    // 容纳 3 个 int 的空间
    for (int i = 0; i < n; i++) a[i] = i * 10;

    a = realloc(a, 4 * sizeof(int));     // 扩大到 4 个 int
    a[3] = 30;

    for (int i = 0; i < 4; i++) printf("%d ", a[i]);
    printf("\n");                        // 0 10 20 30

    free(a);                             // 把内存归还
    return 0;
}
```

- `calloc(n, size)` 像 `malloc` 一样分配数组，并把它填成 0。

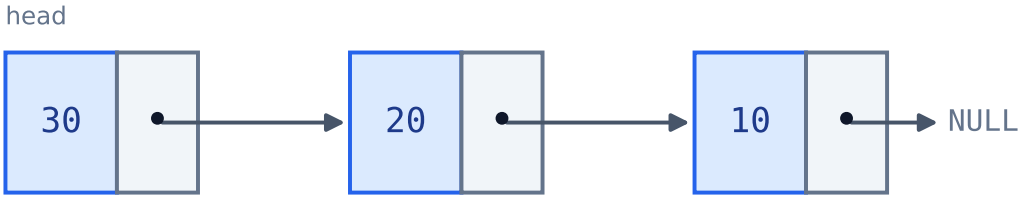
常见错误

- 每个 `malloc` 都要有对应的 `free`; 忘记 free 会内存泄漏。
- `free` 之后再用那块内存 (use-after-free) 是严重的错误。
- 使用内存前, 先检查 `malloc` 没有返回 `NULL`。

10 数据结构

10.1 链表

链表 (linked list) 是一串节点 (node)。每个节点保存一个值和一个指向 `next`(下一个) 节点的指针; 最后一个指向 `NULL`。与数组不同, 它用 `malloc` 一次增加一个节点。用完后一定要 `free` 每个节点。



10-linked-list

链表: 每个节点保存一个值和指向下一个节点的指针, 最后指向 *NULL*

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

typedef struct Node {
    int value;
    struct Node *next;
} Node;

int main(void) {
    Node *head = NULL;
    for (int v = 10; v <= 30; v += 10) {   // 依次头插 10、20、30
        Node *n = malloc(sizeof(Node));
        n->value = v;
        n->next = head;
        head = n;
    }
    for (Node *p = head; p != NULL; p = p->next) {
        printf("%d ", p->value);           // 30 20 10
    }
    printf("\n");

    while (head != NULL) {                 // 释放整个链表
        Node *t = head;
```

```
        head = head->next;
        free(t);
    }
    return 0;
}
```

10.2 栈与队列

栈 (stack) 是 LIFO 后进先出 (last in, first out): 在同一端入栈和出栈。队列 (queue) 是 FIFO 先进先出 (first in, first out): 在尾部加入, 从头部移除。两者都可以用数组加索引变量轻松实现。

```
#include <stdio.h>

int main(void) {
    int stack[10];
    int top = 0;                        // 下一个空位
    stack[top++] = 1;                   // 入栈
    stack[top++] = 2;
    stack[top++] = 3;
    while (top > 0) {
        printf("%d ", stack[--top]);    // 出栈:3 2 1
    }
    printf("\n");
    return 0;
}
```

```
#include <stdio.h>

int main(void) {
    int queue[10];
    int front = 0, back = 0;
    queue[back++] = 1;                      // 入队
    queue[back++] = 2;
    queue[back++] = 3;
    while (front < back) {
        printf("%d ", queue[front++]);      // 出队:1 2 3
    }
    printf("\n");
    return 0;
}
```

常见错误

- 在链表中绝不能弄丢 `head` 指针, 否则整条链表都找不回来。

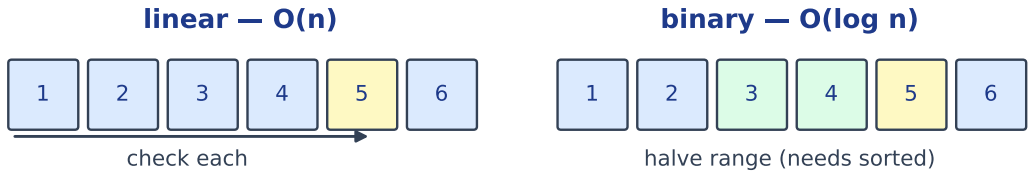
- 用完后释放每个节点, 解开链接前先遍历。
- pop 之前先检查空链表 (`head` 为 `NULL`)。

11　查找、排序与递归

11.1　线性查找与二分查找

线性查找 (linear search) 依次检查每个元素 —— 它对任何数组都适用。二分查找 (binary search) 快得多, 但需要一个已排序 (sorted) 的数组: 它检查中间值, 每一步丢掉一半。两者都返回索引, 找不到则返回 `-1`。

```c
#include <stdio.h>

int linear(int a[], int n, int target) {
    for (int i = 0; i < n; i++)
        if (a[i] == target) return i;
    return -1;
}

int binary(int a[], int n, int target) {
    int lo = 0, hi = n - 1;
    while (lo <= hi) {
        int mid = (lo + hi) / 2;
        if (a[mid] == target) return mid;
        else if (a[mid] < target) lo = mid + 1;
        else hi = mid - 1;
    }
    return -1;
}

int main(void) {
    int a[] = {2, 5, 8, 12, 16, 23};
    int n = sizeof(a) / sizeof(a[0]);
    printf("%d %d %d\n", linear(a, n, 12), binary(a, n, 12), binary(a,
↪   n, 9));
    return 0;   // 3 3 -1
}
```



Exam tip: binary search needs a sorted array; linear works on any order

Linear scans all; binary halves a sorted range

11.2 排序

冒泡排序 (bubble sort) 反复比较相邻元素, 并交换 (swap) 任何顺序不对的一对。每一趟之后, 最大的值就 “ 冒泡 ” 到末尾。

```c
#include <stdio.h>

int main(void) {
    int a[] = {5, 2, 9, 1, 7};
    int n = sizeof(a) / sizeof(a[0]);
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - 1 - i; j++) {
            if (a[j] > a[j + 1]) {
                int t = a[j]; a[j] = a[j + 1]; a[j + 1] = t;
            }
        }
    }
    for (int i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");   // 1 2 5 7 9
    return 0;
}
```

11.3 递归

递归 (recursion) 是一个调用自己的函数。它需要一个基准情形 (base case) 来停止, 以及一个朝它靠近的递归调用。没有基准情形它就永不结束。

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

```c
int factorial(int n) {
    if (n <= 1) return 1;           // 基准情形
    return n * factorial(n - 1);    // 递归调用
}

int main(void) {
    printf("%d\n", factorial(5));  // 120
    return 0;
}
```

常见错误

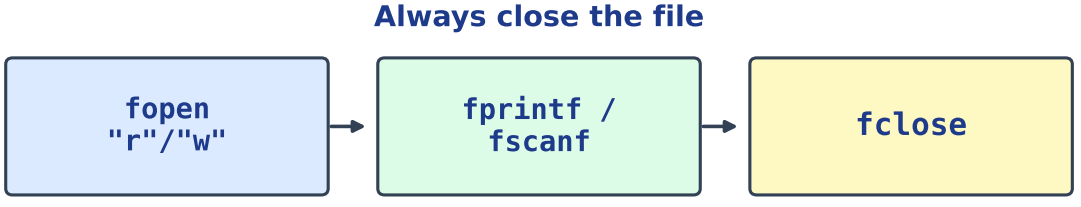
- 二分查找需要**已排序**的数组; 递归需要基准情形。
- 每次递归都要更靠近基准情形, 否则栈会溢出。
- 冒泡和插入排序是 $O(n^2)$—— 学习可以, 规模大了就慢。

12 文件与错误

12.1 文本文件

`fopen` 返回一个文件指针 (file pointer); 模式 (mode) 字符串说明要做什么 —— `"w"` 写、`"r"` 读、`"a"` 追加。用 `fprintf` 写, 用 `fscanf` 读, 用完后一定要 `fclose`。

```c
#include <stdio.h>

int main(void) {
    FILE *out = fopen("scores.txt", "w");   // 打开以写入
    fprintf(out, "Alice 80\nBob 95\n");
    fclose(out);

    FILE *in = fopen("scores.txt", "r");    // 打开以读取
    char name[20];
    int score;
    while (fscanf(in, "%s %d", name, &score) == 2) {
        printf("%s -> %d\n", name, score);
    }
    fclose(in);
    return 0;
}
```



Exam tip: check fopen != NULL; mode "r" reads, "w" writes; fclose when done

fopen → read/write → fclose

12.2 用返回码处理错误

C 没有异常机制。函数改用返回码 (return code) 来报告失败 —— 按惯例 `0` 表示成功, 非零表示错误。调用者在信任结果之前先检查这个码。(`fopen` 遵循同样的思路: 失败时返回 `NULL`。)

```c
#include <stdio.h>

// 成功返回 0，除数为 0 时返回 -1
int safe_divide(int a, int b, int *result) {
    if (b == 0) return -1;          // 错误码
    *result = a / b;
    return 0;                       // 成功
}

int main(void) {
    int r;
    if (safe_divide(10, 2, &r) == 0)
        printf("10 / 2 = %d\n", r);          // 10 / 2 = 5
    if (safe_divide(10, 0, &r) != 0)
        printf("cannot divide by zero\n");   // 报告错误
    return 0;
}
```

常见错误

- 读写前先检查 fopen 没有返回 NULL。
- 用完后一定要 fclose 文件。
- 从 main 返回非零值，向调用者表示出错。

13 位与数据

13.1 位、二进制与位运算符

位 (bit) 是单个的 0 或 1; 数字以二进制 (binary) 存储。按位 (bitwise) 运算符直接作用于这些位:`&` 与、`|` 或、`^` 异或、`<<` 左移 (每次 ×2)、`>>` 右移 (÷2)。

```c
#include <stdio.h>

int main(void) {
    int a = 12;               // 1100
    int b = 10;               // 1010
    printf("%d\n", a & b);    // 8   与   -> 1000
    printf("%d\n", a | b);    // 14  或   -> 1110
    printf("%d\n", a ^ b);    // 6   异或 -> 0110
    printf("%d\n", a << 1);   // 24  左移
    printf("%d\n", a >> 1);   // 6   右移
    return 0;
}
```

十六进制 (hexadecimal,base 16) 是二进制的紧凑写法: 一个十六进制位恰好等于四个二进制位。字面量用 `0x` 开头; 打印用 `%x`:

```c
#include <stdio.h>

int main(void) {
    printf("%d\n", 0xFF);   // 255  (0x 表示十六进制字面量)
    printf("%x\n", 255);    // ff
    printf("%x\n", 12);     // c
    return 0;
}
```

Exam tip: bitwise ops work on bits of integers; << multiplies by 2, >> divides by 2

Bitwise operators work on the bits of integers

13.2 游程编码

游程编码 (run-length encoding,RLE) 是一种简单的压缩 (compression) 方法: 把每一段重复字符的游程 (run) 替换成一个计数加上那个字符。它是无损 (lossless) 的 —— 原始数据可以完全还原。

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

int main(void) {
    char *s = "aaabbc";
    int n = strlen(s);
    for (int i = 0; i < n; ) {
        char c = s[i];
        int run = 0;
        while (i < n && s[i] == c) { run++; i++; }
        printf("%d%c", run, c);      // 3a2b1c
    }
    printf("\n");
    return 0;
}
```

常见错误

- `&` 是按位与,`&&` 是逻辑与 —— 别混淆。
- 左移 `<< 1` 让值翻倍; 右移 `>> 1` 让值减半。
- 第 `n` 位的值是 `1 << n`(C 没有 `**` 幂运算符); 用 `(x >> n) & 1` 检查某一位。

14 预处理器与限定符

14.1 预处理器、const、static、enum

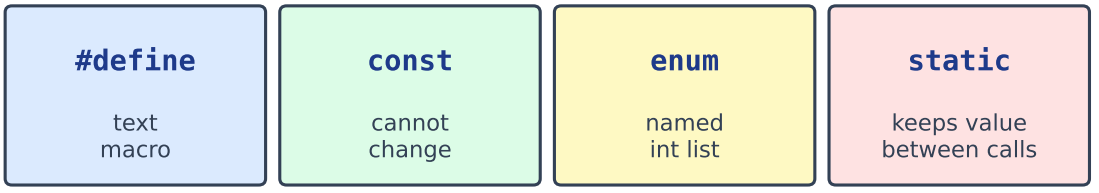
预处理器 (preprocessor) 在编译之前运行。`#define` 创建一个宏 (macro)—— 在所有地方被替换的纯文本。`const` 创建一个不能改变的常量 (constant)。函数内部的 `static` 让变量在多次调用之间保留它的值。`enum`(枚举,enumeration) 给一组从 0 开始的整数起名字。

```c
#include <stdio.h>

#define MAX 100                      // 宏: 编译前被替换的文本

enum Color { RED, GREEN, BLUE };     // 有名字的整数 0、1、2

int next_id(void) {
    static int count = 0;            // 在多次调用之间保留值
    count++;
    return count;
}

int main(void) {
    const double PI = 3.14;          // 不能被改变
    printf("%d %.2f\n", MAX, PI);            // 100 3.14
    printf("%d %d %d\n", RED, GREEN, BLUE);  // 0 1 2
    int a = next_id();
    int b = next_id();
    printf("%d %d\n", a, b);                 // 1 2
    return 0;
}
```

更大的程序会拆成多个文件: 声明放进头文件 (header)`mine.h`, 再用 `#include "mine.h"` 引入 —— 引号表示你自己的文件,`< >` 表示标准库。包含保护 (`#ifndef MINE_H` / `#define MINE_H` / `#endif`) 防止同一个头文件被引入两次。

常见错误

- `#define` 只做纯文本替换, 没有类型检查; 宏体和参数都要用括号包起来。
- `const` 值一旦设定就不能再改。
- `static` 局部变量在两次函数调用之间会保留它的值。



Exam tip: preprocessor runs first; enum values start at 0 unless set

#define, const, enum, and static keep code clear