C Handout

English edition

Contents

1 C basics

1.1 main, printf & functions

Every C program starts in `main`. `#include <stdio.h>` pulls in a header 头文件 so you can use `printf` to print. A function 函数 is a named block you can call; `\n` starts a new line. `main` returns `0` to mean "success".

```c
#include <stdio.h>

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

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

1.2 Variables, types & arithmetic

C needs a type for every variable: `int` (integer 整数), `double` (floating-point 浮点数), `char` (one character). `printf` uses a format specifier 格式说明符—`%d`, `%f`, `%c` —for each value. `/` between two ints is integer division; `%` is the remainder 余数.

```c
#include <stdio.h>

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;
}
```

1.3 Comments & style

A comment 注释 is a note for humans; the compiler ignores it. Use `//` for one line and `/* ... */` for a block. Good indentation 缩进 and clear names make code easy to read.

```c
#include <stdio.h>

int main(void) {
    // a single-line comment
    /* a block
       comment */
    int total = 3 + 4;       // clear names help
```

```
    printf("%d\n", total);   // 7
    return 0;
}
```

2 Selection

2.1 if / else

An `if` runs a block when a condition 条件 is true. C has no real boolean 布尔值 type by default: `0` is false and any non-zero value is true. Chain choices with `else if` and `else`. Compare with `==`, `!=`, `<`, `>`, `<=`, `>=`.

```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;
}
```

2.2 switch

A `switch` picks one `case` by an integer value. Each `case` needs a `break` to jump out 跳出—without it, C "falls through" into the next case. `default` runs when nothing matches.

```c
#include <stdio.h>

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;
}
```

3 Loops

3.1 while & for loops

A loop 循环 repeats a block. A `while` loop runs as long as a condition is true. A `for` loop packs the start, the test, and the increment 自增 (`i++`) into one line —best when you know how many times to repeat.

```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;
}
```

3.2 Accumulation

A common pattern: start an accumulator 累加器 at `0`, then add to it inside a loop. The same idea counts items or finds a running total.

```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 Nested loops & patterns

A loop inside another loop is a nested 嵌套 loop. The inner loop finishes fully for each step of the outer loop —perfect for grids and patterns.

```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;
}
```

4 Functions

4.1 Parameters & return values

A function takes parameters 参数 (inputs) and gives back a return value 返回值. The return type comes first (`int`, `double`, ⋯); `void` means it returns nothing.

```c
#include <stdio.h>

int square(int x) {        // x is a parameter
    return x * x;          // hand back a value
}

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

4.2 Prototypes & scope

C reads top to bottom, so a function must be known before it is called. A prototype 函数原型—the header line plus `;` —declares it early so you can keep `main` first. A variable's scope 作用域 is the block it lives in: it is local 局部 and disappears when the block ends.

```c
#include <stdio.h>

int add(int a, int b);     // prototype: declared before use

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

int add(int a, int b) {    // definition comes later
    int sum = a + b;       // sum is local to add
    return sum;
}
```

5 Arrays

5.1 1-D arrays

An array 数组 holds several values of one type. Index 索引 from `0`. C does **not** store an array's length, so a common trick computes the element 元素 count: `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;
}
```

5.2 Array algorithms

Traverse 遍历 an array with a `for` loop to find a max, a total, or a count. Always loop from `0` up to `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 2-D arrays

A 2-D array is a grid of rows 行 and columns 列: `grid[row][col]`. Use two nested loops to visit every cell.

```c
#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;
}
```

6 Pointers

6.1 &, * and pass-by-pointer

A pointer 指针 stores the address 地址 of a variable. `&x` gives the address of `x`; `*p` dereferences 解引用 the pointer —it reads or writes the value stored there. Passing a pointer lets a function change the caller's variable (pass-by-pointer).

```c
#include <stdio.h>

void addOne(int *p) {      // p holds an address
    *p = *p + 1;           // change the value at that address
}

int main(void) {
    int x = 10;
    int *ptr = &x;         // & takes the address of x
    printf("%d\n", *ptr); // * reads the value: 10
    addOne(&x);
    printf("%d\n", x);     // 11 — changed through the pointer
    return 0;
}
```

7 Strings

7.1 Char arrays & the null terminator

A C string 字符串 is an array of `char`. Every string ends with a hidden null terminator 空终止符 `'\0'`, which marks where it stops. `strlen` (from `<string.h>`) counts characters up to that `'\0'`. Print a whole string with `%s` and one character 字符 with `%c`.

```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 (stops at '\0')
    printf("%c\n", name[0]);        // M
    return 0;
}
```

You can traverse a string by looping until you reach `'\0'`.

```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;
}
```

8 Structs

8.1 struct, typedef, . vs ->

A struct 结构体 groups related values into one type. Each value is a member 成员. `typedef` gives the struct a short name so you can write `Dog` instead of `struct Dog`. Use `.` on a struct value, but `->` on a pointer to a struct.

```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   (. on a value)

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

9 Dynamic memory

9.1 malloc, free & realloc

`malloc` allocates 分配 memory on the heap 堆 at run time and returns a pointer to it. `realloc` resizes that block; `free` gives it back. Every `malloc` needs a matching `free`, or you get a memory leak 内存泄漏. These live in `<stdlib.h>`.

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

int main(void) {
    int n = 3;
    int *a = malloc(n * sizeof(int));    // room for 3 ints
    for (int i = 0; i < n; i++) a[i] = i * 10;

    a = realloc(a, 4 * sizeof(int));     // grow to 4 ints
    a[3] = 30;

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

    free(a);                             // hand the memory back
    return 0;
}
```

10 Data structures

10.1 Linked lists

A linked list 链表 is a chain of nodes 节点. Each node holds a value and a pointer to the `next` node; the last one points to `NULL`. Unlike an array it grows one node at a time with `malloc`. Always `free` every node when done.

```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) {   // prepend 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) {                 // free the whole list
        Node *t = head;
        head = head->next;
        free(t);
    }
    return 0;
}
```

10.2 Stacks & queues

A stack 栈 is LIFO 后进先出 (last in, first out): push and pop at the same end. A queue 队列 is FIFO 先进先出 (first in, first out): add at the back, remove from the front. Both are easy to build on an array with index variables.

```c
#include <stdio.h>

int main(void) {
    int stack[10];
    int top = 0;                       // next free slot
    stack[top++] = 1;                  // push
    stack[top++] = 2;
```

```
    stack[top++] = 3;
    while (top > 0) {
        printf("%d ", stack[--top]);   // pop: 3 2 1
    }
    printf("\n");
    return 0;
}
```

```
#include <stdio.h>

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

11 Searching, sorting & recursion

11.1 Linear & binary search

Linear search 线性查找 checks each element in turn —it works on any array. Binary search 二分查找 is far faster but needs a sorted 已排序 array: it checks the middle and discards half each step. Both return the index, or `-1` if missing.

```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
}
```

11.2 Sorting

Bubble sort 冒泡排序 repeatedly compares neighbours and swaps 交换 any pair that is out of order. After each pass the largest value ”bubbles” to the end.

```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 Recursion

Recursion 递归 is a function that calls itself. It needs a base case 基准情形 to stop, and a recursive call that steps toward it. Without a base case it never ends.

```
#include <stdio.h>

int factorial(int n) {
    if (n <= 1) return 1;           // base case
    return n * factorial(n - 1);    // recursive call
}

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

12 Files & errors

12.1 Text files

`fopen` returns a file pointer 文件指针; the mode 模式 string says what to do —`"w"` write, `"r"` read, `"a"` append. Write with `fprintf`, read with `fscanf`, and always `fclose` when done.

```c
#include <stdio.h>

int main(void) {
    FILE *out = fopen("scores.txt", "w");   // open for writing
    fprintf(out, "Alice 80\nBob 95\n");
    fclose(out);

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

12.2 Error handling with return codes

C has no exceptions. Instead a function signals failure with a return code 返回码—by convention `0` means success and non-zero means an error. The caller checks the code before trusting the result. (`fopen` follows the same idea: it returns `NULL` when it fails.)

```c
#include <stdio.h>

// returns 0 on success, -1 if the divisor is 0
int safe_divide(int a, int b, int *result) {
    if (b == 0) return -1;          // error code
    *result = a / b;
    return 0;                       // success
}

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");   // error reported
    return 0;
}
```

13 Bits & data

13.1 Bits, binary & bitwise operators

A bit 位 is a single 0 or 1; numbers are stored in binary 二进制. Bitwise 按位 operators work on the bits directly: `&` AND, `|` OR, `^` XOR, `<<` shift left (×2 each step), `>>` shift right (÷2).

```c
#include <stdio.h>

int main(void) {
    int a = 12;              // 1100
    int b = 10;              // 1010
    printf("%d\n", a & b);   // 8   AND  -> 1000
    printf("%d\n", a | b);   // 14  OR   -> 1110
    printf("%d\n", a ^ b);   // 6   XOR  -> 0110
    printf("%d\n", a << 1);  // 24  left shift
    printf("%d\n", a >> 1);  // 6   right shift
    return 0;
}
```

13.2 Run-length encoding

Run-length encoding (RLE) is a simple compression 压缩 method: replace each run 游程 of repeated characters with a count and the character. It is lossless 无损—the original is fully recoverable.

```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;
}
```

14 Preprocessor & qualifiers

14.1 Preprocessor, const, static, enum

The preprocessor 预处理器 runs before compiling. `#define` makes a macro 宏—plain text replaced everywhere. `const` makes a constant 常量 that cannot change. `static` inside a function keeps a variable's value between calls. An `enum` (enumeration 枚举) names a set of integers starting at `0`.

```c
#include <stdio.h>

#define MAX 100                     // macro: text replaced before
↪  compiling

enum Color { RED, GREEN, BLUE };    // named integers 0, 1, 2

int next_id(void) {
    static int count = 0;           // keeps its value between calls
    count++;
    return count;
}

int main(void) {
    const double PI = 3.14;         // cannot be changed
    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;
}
```