Chap1 程序设计概述

核心结论：本章围绕程序设计的基础框架展开，介绍了程序设计语言的发展与特点、程序实现的软硬件环境、C程序的加工执行流程，以及程序设计的核心要素（数据结构与算法），为后续编程学习奠定理论基础。

1.1 程序设计语言

- 发展历程：从面向机器的机器语言（0/1序列）、汇编语言（助记符），演进到面向过程（如C、Pascal）和面向对象（如Java、C++）的高级语言。
- C语言特点：入门门槛低、功能丰富、支持底层操作、目标代码效率高、可移植性强，兼具高级语言易用性和低级语言灵活性，被誉为“中级语言”。
- 核心地位：许多新语言借鉴C语言特性，其设计者获图灵奖，广泛应用于底层开发和复杂程序设计。

1.2 程序的实现环境

- 硬件环境：由输入设备（键盘、鼠标）、输出设备（显示器、打印机）、存储设备（内存、硬盘）和CPU（运算器+控制器）组成，是程序运行的物理基础。
- 软件环境：包含操作系统（Windows、Linux）、编辑程序（记事本、VS Code）、编译程序、连接程序，常用集成开发环境（IDE）有Visual Studio、Dev-C++、VS Code，简化编程、编译、调试流程。

1.3 C程序的加工和执行

- 加工步骤：
  1. 编译：编译程序将源程序（.c）转换为机器语言目标程序（.obj），检查语法错误。
  2. 连接：连接程序将目标程序与运行系统、库模块组合，生成可执行程序（.exe）。
- 执行规则：C程序必须包含且仅包含一个`main`函数，程序从`main`函数开始执行，与`main`函数在代码中的位置无关。
- 示例：简单C程序（helloworld.c）

```c
/*程序名：helloworld.c */
/*功能：在屏幕上输出一行文本*/
#include <stdio.h>              /*文件包含   */
int main( )                    /*主函数     */
{                               /*函数体开始*/
    printf("Hello world!\n");   /*输出语句   */
    return 0;
}                              /*函数体结束*/
```

1.4 程序设计基本概念

- 程序：解决特定问题的指令序列，核心包含两部分——数据描述（数据结构）和操作描述（算法）。
- 算法：解决问题的有限步骤集合，需满足有穷性、确定性、可行性，示例：
  1. 求5!：步骤1→1×2，步骤2→结果×3，步骤3→结果×4，步骤4→结果×5，最终得120。
  2. 找出三个数的最大值：S1输入a、b、c；S2比较a和b，取较大值存为MAX；S3比较MAX和c，更新MAX；S4输出MAX。
- 数据结构：数据的组织形式，描述数据的存储和关联方式（如整型、数组等）。
- 程序设计步骤：分析问题→确定算法→编程→运行调试→总结。

Chap2 程序设计初步

核心结论：本章聚焦C程序的基础编写能力，涵盖程序结构规范、输入输出方法、数据类型体系、常量变量定义、运算符表达式及类型转换，是编写简单C程序的核心基础。

2.1 程序的基本结构

- 核心组成：
  1. `main`函数：程序入口，必须包含且仅一个，格式为`int main()`（需`return 0`）或`void main()`（无返回值）。
  2. 注释：`/* 多行注释 */`或`// 单行注释`，不参与编译，增强代码可读性。
  3. 预编译命令：以`#`开头，如`#include <stdio.h>`（引用标准输入输出库），需放在程序开头。
- 结构规则：函数体用`{}`包裹，语句以`;`结束，使用标准库函数需包含对应头文件。
- 示例：计算两数之和

```c
/*程序名：2_1_2.cpp*/
/*功能：计算两个整数的和并输出结果*/
#include <stdio.h>

int add(int x, int y)  // 自定义函数
{
    int z;
    z = x + y;
    return z;
}

int main()
{
   int i1, i2, sum;
   printf("请输入两个整数：");
   scanf("%d,%d", &i1, &i2);
   sum = add(i1, i2);
   printf("sum=%d\n", sum);
   return 0;
}
```

2.2 数据的输入和输出

- 格式输出函数`printf`：
  - 格式：`printf("格式控制串", 输出表列)`，格式控制串包含格式说明（`%d`整型、`%f`浮点型、`%c`字符型、`%s`字符串）、普通字符、转义字符（`\n`换行、`\t`制表符）。
  - 示例：

```
int a = -1, b = 25;
float c = 12.35;
printf("%d,%5d\n", a, b);  // 输出：-1,   25
printf("a=%7.2f\n", c);   // 输出：a=  12.35
```

- 格式输入函数`scanf`：
  - 格式：`scanf("格式控制串", 地址表列)`，变量需加`&`（地址符），格式说明与`printf`对应（`%lf`对应`double`）。
  - 分隔规则：输入多个数据时，可用空格、换行或显式分隔符（如逗号）。
  - 示例：

```
int n;
double x;
scanf("%d,%lf", &n, &x);  // 输入：23,2.345
```

- 字符输入输出函数：
  - `getchar()`：读单个字符，格式`char ch = getchar();`。
  - `putchar(ch)`：输单个字符，格式`putchar(ch);`。
- 注意：`scanf`无法读取带空格的字符串，需用`fgets`或`scanf("%[^\n]", str)`。
- 转义字符对照表：

转义字符	含义	ASCII 码值（十进制）	详细解释
\a	警报（响铃）	7	发出系统提示音，通常用于引起用户注意。在某些系统中可能没有明显效果。
\b	退格	8	将光标位置向左移动一个字符。如果光标已经在行首，则行为可能因系统而异。
\f	换页	12	将光标移动到下一页的开头。在打印输出时很有用，但在屏幕输出中可能显示为空白字符。
\n	换行	10	将光标移动到下一行的开头。这是最常用的行结束符，用于文本的换行。
\r	回车	13	将光标移动到当前行的开头。通常与 \n 配合使用，在 Windows 系统中表示换行。
\t	水平制表符	9	将光标移动到下一个制表位置。通常用于对齐文本或创建缩进效果。

转义字符	含义	ASCII 码值（十进制）	详细解释
\v	垂直制表符	11	在支持的设备上，将光标向下移动到下一个垂直制表位置。在大多数现代终端中，效果可能与 \n 相同。
\	反斜杠	92	用于在字符串中表示一个字面意义上的反斜杠字符。
'	单引号	39	用于在字符常量中表示一个字面意义上的单引号字符。
"	双引号	34	用于在字符串常量中表示一个字面意义上的双引号字符。
\?	问号	63	用于表示字面意义上的问号，主要用于避免触发三字符序列。
\0	空字符（NULL）	0	表示字符串的结束。在 C 语言中，字符串以空字符结尾。
\xhh	十六进制值	-	表示一个十六进制值，其中 hh 是一个或多个十六进制数字。例如，\x41 表示字符 'A'。
\ooo	八进制值	-	表示一个八进制值，其中 ooo 是一到三个八进制数字。例如，\101 表示字符 'A'。

2.3 基本概念

- 基本字符：数字（0-9）、字母（a-z/A-Z）、特殊字符、空白字符（空格、换行、制表符），非基本字符（双引号内除外）为非法字符。
- 标识符：变量/函数名，规则为字母/下划线开头，由字母、数字、下划线组成，区分大小写（如`a`和`A`不同）。
- 关键字：32个ANSI标准关键字（如`int`、`if`、`for`），不可用作标识符。

2.4 数据类型

- 基本数据类型：

类型	细分类型	占用字节（常见）	取值范围	特点
整型	int	4	-2^{31}~2^{31}-1	存储整数
	short	2	-32768~32767	短整型
	long	4	-2^{31}~2^{31}-1	长整型
实型（浮点型）	float	4	±(3.4×10^{-38}~3.4×10^{38})	单精度，有效数字7-8位

类型	细分类型	占用字节（常见）	取值范围	特点
	double	8	$\pm(1.7\times10^{-308}\sim1.7\times10^{308})$	双精度，有效数字15-16位
字符型	char	1	-128~127（signed char）	存储ASCII码，可与整型通用
	unsigned char	1	0~255	无符号字符

- 扩展类型：`unsigned`（无符号）修饰整型/字符型，`long long`（64位整型）。
- 格式说明符：

格式说明符	类型
%c	字符
%d	有符号整数
%e 或 %E	浮点数的科学计数法
%f	浮点值
%g 或 %G	类似于 %e 或%E
%hi	有符号整数(短整型)
%hu	无符号整数(短整型)
%i	无符号整数
%l 或 %ld 或 %li	长整型
%lf	双精度型
%Lf	长双精度型
%lu	无符号整数或无符号long
%lli 或 %lld	Long long
%llu	无符号 long long
%o	八进制表示
%p	指针
%s	字符串
%u	无符号 int
%x 或 %X	十六进制表示

2.5 常量和变量

- 常量：程序运行中值不变的量，分类如下：
  1. 整型常量：十进制（123）、八进制（0123）、十六进制（0x12）。
  2. 实型常量：小数形式（12.3）、指数形式（1.23e2），后缀`f`表示`float`（否则为`double`）。
  3. 字符常量：单引号包裹（`'a'`），本质是ASCII码（`'A'=65`、`'0'=48`）。
  4. 转义字符：以`\`开头，如`\n`（换行）、`\t`（制表符）、`\0`（字符串结束符）、`\x41`（十六进制ASCII码）。
  5. 字符串常量：双引号包裹（`"Good"`），自动添加`\0`结束符，占字节数=字符数+1。
  6. 符号常量：`#define 符号常量名 常量`（无分号），示例：

```
#define PI 3.14159
int main() {
    float r = 2.0;
    float s = PI * r * r;
    printf("面积：%f\n", s);
    return 0;
}
```

- 变量：程序运行中值可变的量：
  - 定义格式：`数据类型 变量名1, 变量名2;`（如`int a, b;`）。
  - 初始化：定义时赋值（`int a=5;`），区别于赋值语句（运行时赋值），未初始化的自动变量值随机。
  - 内存特性：不同类型占用不同字节，变量名对应内存地址，通过地址访问值。

2.6 运算符与表达式

- 算术运算符：`+`、`-`、`*`、`/`、`%`（取模，仅整型）、`++`（自增）、`--`（自减）：
  - `++i`（先运算后取值）、`i++`（先取值后运算）。
  - 示例：`8%5=3`、`5/2=2`（整数除法舍弃小数）、`5.0/2=2.5`（实型除法）。
- 赋值运算符：`=`、`+=`、`-=`、`*=`、`/=`、`%=`，结合性自右向左（`a=b=c`等价于`a=(b=c)`）。
- 关系运算符：`<`、`<=`、`>`、`>=`、`==`（等于）、`!=`（不等于），返回值为1（真）或0（假），优先级低于算术运算符。
- 逻辑运算符：`!`（非）、`&&`（与，短路特性）、`||`（或，短路特性），优先级：`!`>算术>关系>`&&`>`||`>赋值。
- 逗号运算符：`表达式1,表达式2`，结果为表达式2的值，优先级最低。
- 表达式：由运算符和运算对象组成，示例：
  - 算术表达式：`(a+b+c)/3`、`sqrt(s*(s-a)*(s-b)*(s-c))`。
  - 关系表达式：`a>b`、`x>=3&&x<=5`。
  - 逻辑表达式：`!a||(a&&b)`。

2.7 数据类型的转换

- 自动转换（隐式转换）：
  - 运算转换：“向高看齐”，低精度→高精度（`char`→`int`→`long`→`float`→`double`）。
  - 赋值转换：右值→左值类型（实型→整型舍弃小数，整型→实型补0，长字节→短字节可能溢出）。
- 强制转换（显式转换）：格式`(类型标识符)表达式`，不改变原变量类型，示例：

```
float x = 3.6;
int i = (int)x;  // i=3（舍弃小数，非四舍五入）
```

2.8 数学函数

- 需包含头文件`<math.h>`，常用函数：

函数名	功能	示例
`fabs(x)`	求x的绝对值（实型）	`fabs(-3.5)=3.5`
`sqrt(x)`	求x的非负平方根	`sqrt(16.0)=4.0`
`pow(x,y)`	求x^y	`pow(2.0,3.0)=8.0`
`sin(x)`	正弦函数（x为弧度）	`sin(0.0)=0.0`
`log(x)`	自然对数lnx	`log(2.71828)=1.0`

- 示例：计算三角形面积

```
#include <stdio.h>
#include <math.h>
int main() {
    double a, b, c, s, area;
    printf("请输入三角形三边：");
    scanf("%lf,%lf,%lf", &a, &b, &c);
    s = (a + b + c) / 2.0;
    area = sqrt(s * (s - a) * (s - b) * (s - c));
    printf("面积：%7.2f\n", area);
    return 0;
}
```

Chap3 程序控制结构

核心结论：本章围绕程序的三种基本控制结构（顺序、选择、循环）展开，详解关系/逻辑/条件表达式的用法，掌握if、switch选择语句和while、do-while、for循环语句的实现，结合循环控制语句（break/continue）和嵌套结构，解决实际编程问题。

3.1 关系表达式、逻辑表达式、条件表达式

这三类表达式是控制结构的核心判断依据，用于描述程序执行的条件。

3.1.1 关系表达式

- 关系运算符：共6个，分为两组优先级（高：`<` `<=` `>` `>=`；低：`==` `!=`），结合性自左向右。
- 表达式定义：用关系运算符连接两个表达式（算术、关系等），结果为逻辑值（1代表真，0代表假）。
- 示例：

```
int a=0, b=1, c=2;
printf("%d\n", a<=c);     // 输出1（真）
printf("%d\n", a<=c<=b); // 输出1（等价于(a<=c)<=b，即1<=1为真）
printf("%d\n", 5>2>7);    // 输出0（等价于(5>2)>7，即1>7为假）
```

- 注意：避免浮点型用`==`/`!=`比较，需通过差值绝对值判断（如`fabs(x-1.0)<=1e-6`）。

3.1.2 逻辑表达式

- 逻辑运算符：`!`（非，单目）、`&&`（与）、`||`（或），优先级：`!`>算术>关系>`&&`>`||`>赋值，结合性自左向右。
- 短路特性：
  - `a&&b`：a为假时，不计算b（直接返回假）。
  - `a||b`：a为真时，不计算b（直接返回真）。
- 真值表：

a（非0为真）	b（非0为真）	!a	!b	a	b
真	真	0	0	1	1
真	假	0	1	0	1
假	真	1	0	0	1
假	假	1	1	0	0

- 示例：

```
int a=1, b=2, m=1, n=1;
(m=a>b)&&(n=c>d); // a>b为0，m=0，n不计算仍为1
printf("m=%d, n=%d\n", m, n); // 输出m=0, n=1
```

3.1.3 条件表达式

- 格式：`表达式1?表达式2:表达式3`，结合性自右向左。
- 执行流程：表达式1为真则取表达式2的值，否则取表达式3的值。
- 示例：

```
int max = a>b?a:b; // 求两数最大值
int sign = x>0?1:(x==0?0:-1); // 符号函数
```

- 注意：优先级高于赋值，低于关系/逻辑运算符，复杂条件需加括号明确优先级。

3.2 顺序结构程序设计

- 定义：语句按排列顺序逐行执行，无跳转，是最基础的程序结构。
- 组成语句：赋值语句、复合语句（`{}`包裹）、函数调用语句。
- 示例：计算工业产值（年增长率7.4%，20年后产值）

```
#include <stdio.h>
#include <math.h>
int main() {
    int n;
    double p0, p1, r;
    scanf("%d,%lf,%lf", &n, &p0, &r);
    p1 = p0 * pow(1 + r, n);
    printf("20年后产值：%10.4f\n", p1);
    return 0;
}
```

3.3 选择结构程序设计

选择结构根据条件判断执行不同分支，核心语句为`if`和`switch`。

3.3.1 if语句

- 三种形式：
  1. 单分支：`if(表达式) 语句;`
  2. 双分支：`if(表达式) 语句1; else 语句2;`
  3. 多分支：`if(表达式1) 语句1; else if(表达式2) 语句2; ... else 语句n;`
- 嵌套规则：else与最近的未匹配if配对，建议用`{}`明确作用域。
- 示例1：判断闰年

```
#include <stdio.h>
int main() {
    int year;
    printf("请输入年号：");
    scanf("%d", &year);
    if ((year%4==0&&year%100!=0) || year%400==0)
        printf("%d是闰年\n", year);
    else
        printf("%d不是闰年\n", year);
    return 0;
}
```

- 示例2：求三个数的最大值

```c
#include <stdio.h>
int main() {
    int a, b, c, max;
    printf("请输入三个整数：");
    scanf("%d%d%d", &a, &b, &c);
    if (a > b) max = a;
    else max = b;
    if (c > max) max = c;
    printf("最大值：%d\n", max);
    return 0;
}
```

3.3.2 switch语句

- 格式：

```c
switch(算术表达式) {
    case 常量表达式1: 语句组1; break;
    case 常量表达式2: 语句组2; break;
    ...
    default: 语句组n+1; // 可省略
}
```

- 说明：
  - 表达式为整型/字符型/枚举型，case常量互不相等。
  - `break`用于跳出switch，否则执行后续case（穿透特性）。
  - default可在任意位置，不影响执行逻辑。
- 示例：成绩等级评定

```c
#include <stdio.h>
int main() {
    int score, grade;
    printf("输入成绩（0~100）：");
    scanf("%d", &score);
    grade = score / 10;
    switch(grade) {
        case 10:
        case 9: printf("等级A\n"); break;
        case 8: printf("等级B\n"); break;
        case 7: printf("等级C\n"); break;
        case 6: printf("等级D\n"); break;
        default: printf("等级E\n");
    }
    return 0;
}
```

3.3.3 应用举例：求解一元二次方程$ax^2+bx+c=0$

```c
#include <stdio.h>
#include <math.h>
int main() {
    double a, b, c, delta, k;
```

```
    scanf("%lf,%lf,%lf", &a, &b, &c);
    if (a == 0) {
        if (b != 0)
            printf("唯一根：%f\n", -c/b);
        else if (c == 0)
            printf("无穷多根\n");
        else
            printf("无解\n");
    } else {
        delta = b*b - 4*a*c;
        if (delta == 0)
            printf("两个相同实根：%f\n", -b/(2*a));
        else if (delta > 0)
            printf("两个不同实根：%f, %f\n",
                   (-b+sqrt(delta))/(2*a), (-b-sqrt(delta))/(2*a));
        else {
            k = sqrt(-delta)/(2*a);
            printf("两个虚根：%f+%fi, %f-%fi\n", -b/(2*a), k, -b/(2*a), k);
        }
    }
    return 0;
}
```

3.4 循环结构程序设计

循环结构重复执行某段语句，核心语句为`while`、`do-while`、`for`，用于处理重复任务。

3.4.1 while语句（当型循环）

- 格式：`while(表达式) 语句;`
- 执行流程：先判断表达式，真则执行循环体，需包含使表达式变化的语句（避免死循环）。
- 特点：可能一次不执行循环体。
- 示例：计算1+2+...+10000

```
#include <stdio.h>
int main() {
    int i=1, sum=0;
    while (i <= 10000) {
        sum += i;
        i++;
    }
    printf("和：%d\n", sum);
    return 0;
}
```

3.4.2 do-while语句（直到型循环）

- 格式：`do { 语句; } while(表达式);`
- 执行流程：先执行循环体，再判断表达式，至少执行一次。
- 示例：求满足`1+2+...+n<500`的最大n

```c
#include <stdio.h>
int main() {
    int n=0, sum=0;
    do {
        n++;
        sum += n;
    } while (sum < 500);
    printf("最大n：%d，和：%d\n", n-1, sum-n);
    return 0;
}
```

3.4.3 for语句

- 格式：`for(表达式1;表达式2;表达式3) 语句;`
  - 表达式1：初始化循环变量（可多个，逗号分隔）。
  - 表达式2：循环条件（省略则为真，死循环）。
  - 表达式3：更新循环变量（可多个，逗号分隔）。
- 灵活用法：可省略任意表达式（如`for(;;)`为死循环）。
- 示例：打印九九乘法表

```c
#include <stdio.h>
int main() {
    int i, j, m;
    printf("*");
    for (j=1; j<=9; j++)
        printf("%4d", j);
    printf("\n");
    for (i=1; i<=9; i++) {
        printf("%d", i);
        for (j=1; j<=i; j++) {
            m = i*j;
            printf("%4d", m);
        }
        printf("\n");
    }
    return 0;
}
```

3.4.4 循环控制语句（break/continue）

- `break`：跳出当前循环或switch语句，终止整个循环。
- `continue`：跳过本次循环剩余语句，进入下一次循环。
- 示例：打印100-200之间能被7整除的数

```c
#include <stdio.h>
int main() {
    for (int n=100; n<=200; n++) {
        if (n%7 != 0)
            continue; // 跳过非7的倍数，进入下一次循环
        printf("%5d", n);
    }
    printf("\n");
    return 0;
}
```

3.4.5 循环嵌套

- 定义：循环体内包含另一个循环，用于处理多维数据（如矩阵、表格）。
- 示例：斐波那契数列前12项（每行6项）

```c
#include <stdio.h>
int main() {
    int f2=1, f1=1, f, n;
    printf("%-8d%-8d", f2, f1);
    for (n=3; n<=12; n++) {
        f = f1 + f2;
        printf("%-8d", f);
        if (n%6 == 0)
            printf("\n");
        f2 = f1;
        f1 = f;
    }
    return 0;
}
```

3.4.6 应用举例

1. 穷举法：搬砖问题（36块砖36人搬，男搬4、女搬3、两小孩搬1，一次搬完）

```c
#include <stdio.h>
int main() {
    int men, women, children;
    for (men=0; men<=8; men++) { // 男最多搬32块，men<=8
        for (women=0; women<=11; women++) { // 女最多搬33块，women<=11
            children = 36 - men - women;
            if (4*men + 3*women + children/2 == 36 && children%2 == 0) {
                printf("男：%d，女：%d，小孩：%d\n", men, women, children);
            }
        }
    }
    return 0;
}
```

2. 迭代法：求斐波那契数列前12项（循环实现）

```c
#include <stdio.h>
int main() {
    int f[12] = {1, 1}; // 前两项为1
    for (int i=2; i<12; i++) {
        f[i] = f[i-1] + f[i-2];
    }
    for (int i=0; i<12; i++) {
        printf("%-6d", f[i]);
        if ((i+1)%6 == 0)
            printf("\n");
    }
    return 0;
}
```

Chap4 模块化程序设计——函数

核心结论：本章围绕模块化程序设计展开，详解函数的定义、调用、参数传递、多级调用（嵌套与递归），明确变量的作用域与存储类别，介绍编译预处理命令，实现复杂问题的拆分与高效解决，是C语言结构化编程的核心。

4.1 函数概述

函数是完成特定功能的独立程序模块，是C语言模块化编程的核心，可将复杂问题拆分为多个子问题逐一解决。

4.1.1 函数的分类

- 库函数：由C语言系统提供，如`printf`、`scanf`，使用时需包含对应头文件（如`#include <stdio.h>`）。
- 自定义函数：用户根据需求自行定义，用于实现特定功能（如求面积、排序等）。

4.1.2 函数的核心规则

- 程序入口：一个完整C程序必须且仅包含一个`main`函数，程序从`main`函数开始执行。
- 不能嵌套定义：函数定义不能嵌套在另一个函数内部，需单独定义。
- 功能独立：每个函数应完成单一、明确的功能，增强代码可读性和复用性。

4.1.3 函数的作用

- 简化代码：避免重复编写相同逻辑，减少冗余。
- 便于维护：单个功能模块修改不影响其他部分。
- 拆分复杂问题：将大问题分解为小模块，降低编程难度。

4.2 函数的定义和调用

函数的使用需遵循“先定义后调用”或“先声明后调用”的原则，核心包括定义格式、调用形式、参数传递和返回值处理。

4.2.1 函数的定义

函数定义分为“有返回值”和“无返回值”两种形式：

1. 有返回值函数（用于返回运算结果）：

```
函数类型 函数名(形参表) {
    函数体（变量定义、执行语句）
    return 表达式; // 返回值类型与函数类型一致
}
```

示例：求圆盘面积

```
#include <stdio.h>
double c_area(double r) { // 函数类型为double，形参r为double型
    return r * r * 3.1416; // 返回面积
}
```

2. 无返回值函数（仅执行操作，无返回结果）：

```
void 函数名(形参表) {
    函数体
    return; // 可省略，仅用于结束函数
}
```

示例：打印圆盘面积

```
void pc_area(double r) { // void表示无返回值
    printf("Radius:%f, area:%f\n", r, c_area(r));
}
```

4.2.2 函数的调用

1. 调用形式：
   - 作为语句：无返回值函数的调用（如`pc_area(3.24);`）。
   - 作为表达式：有返回值函数的调用（如`double s = c_area(2.13);`）。
   - 作为实参：函数调用作为其他函数的参数（如`printf("%f", c_area(0.865));`）。
2. 参数传递：
   - 形参：函数定义时的参数（如`c_area(double r)`中的`r`），仅在函数内有效。
   - 实参：函数调用时的参数（如`c_area(3.24)`中的`3.24`），需与形参数量、类型、顺序一致。
   - 传递规则：C语言采用“值传递”，实参值单向传递给形参，形参修改不影响实参。
     示例：参数传递的单向性

```c
#include <stdio.h>
void exch(int x, int y) { // 形参x、y
    int t = x; x = y; y = t; // 仅修改形参
}
int main() {
    int a=2, b=5;
    exch(a, b); // 实参a、b
    printf("a=%d, b=%d\n", a, b); // 输出a=2, b=5，实参未变
    return 0;
}
```

3. 函数返回值：
   - `return`语句用于返回结果，一个函数最多返回一个值。
   - 主函数`main`的返回值：默认类型为`int`，返回0表示程序正常结束，非0表示出错。
4. 函数原型声明：
   - 若自定义函数在主调函数之后定义，需在调用前声明函数原型，格式：

     ```c
     函数类型 函数名(形参类型1, 形参类型2, ...); // 省略形参名也可
     ```

     示例：函数原型声明

```c
#include <stdio.h>
double c_area(double); // 函数原型声明（省略形参名）
int main() {
    printf("Area:%f\n", c_area(3.24)); // 调用前已声明
    return 0;
}
double c_area(double r) { // 函数定义在主函数之后
    return r * r * 3.1416;
}
```

4.3 函数的多级调用

函数的多级调用包括“嵌套调用”（函数内调用其他函数）和“递归调用”（函数调用自身），用于解决复杂逻辑问题。

4.3.1 嵌套调用

- 定义：一个函数调用另一个函数，被调用函数再调用第三个函数，形成层级调用。
- 示例：弦截法求方程根（`main`调用`root`，`root`调用`xpoint`，`xpoint`调用`f`）

```c
#include <stdio.h>
#include <math.h>
double f(double x) { // 计算方程值
    return x*x*x - 5*x*x + 16*x - 80;
}
double xpoint(double x1, double x2) { // 求弦与x轴交点
    return (x1*f(x2) - x2*f(x1))/(f(x2)-f(x1));
}
```

```c
double root(double x1, double x2) { // 求方程根
    double x = xpoint(x1, x2);
    while (fabs(f(x)) >= 1e-6) {
        x = xpoint(x, x2);
    }
    return x;
}
int main() {
    double x1=2, x2=5;
    printf("方程的根：%f\n", root(x1, x2));
    return 0;
}
```

4.3.2 递归调用

- 定义：函数直接或间接调用自身，需满足“递归公式”和“终止条件”（避免死循环）。
- 核心条件：1. 递归公式（n与n-1的关系）；2. 终止条件（n=1或n=0时的基例）。

1. 递归示例1：求n!（阶乘）

```c
#include <stdio.h>
unsigned long fact(int n) {
    if (n == 1) return 1; // 终止条件
    else return n * fact(n-1); // 递归公式：n! = n*(n-1)!
}
int main() {
    int x;
    printf("请输入正整数：");
    scanf("%d", &x);
    printf("%d! = %ld\n", x, fact(x));
    return 0;
}
```

2. 递归示例2：斐波那契数列第n项

```c
#include <stdio.h>
long fibo(int n) {
    if (n == 1 || n == 2) return 1; // 终止条件
    else return fibo(n-1) + fibo(n-2); // 递归公式：f(n)=f(n-1)+f(n-2)
}
int main() {
    int n;
    scanf("%d", &n);
    printf("第%d项：%ld\n", n, fibo(n));
    return 0;
}
```

3. 递归示例3：汉诺塔问题

```c
#include <stdio.h>
void moveone(char from, char to) { // 移动一个金盘
    printf("%c -> %c\n", from, to);
}
void hanoi(int n, char from, char by, char to) {
```

```
    if (n == 1) moveone(from, to); // 终止条件：1个金盘直接移动
    else {
        hanoi(n-1, from, to, by); // 把n-1个从from移到by
        moveone(from, to); // 把第n个从from移到to
        hanoi(n-1, by, from, to); // 把n-1个从by移到to
    }
}
int main() {
    hanoi(3, 'A', 'B', 'C'); // 3个金盘从A移到C，B为中间柱
    return 0;
}
```

4.4 变量与函数

变量按“作用域”和“存储类别”分类，核心是明确变量的可用范围和生命周期。

4.4.1 按作用域分类

1. 局部变量：
   - 定义位置：函数内、复合语句内（包括形参）。
   - 作用域：仅在定义它的函数或复合语句内有效。
   - 特点：函数调用时分配内存，调用结束释放；未初始化时值随机。
     示例：

```
#include <stdio.h>
void func() {
    int a = 10; // 局部变量，仅func内有效
    printf("func内a：%d\n", a);
}
int main() {
    func();
    // printf("%d", a); // 错误：a是func的局部变量，主函数不可用
    return 0;
}
```

2. 全局变量：
   - 定义位置：函数外部，不属于任一函数。
   - 作用域：从定义处到源文件结束，可被多个函数访问。
   - 特点：程序运行期间始终占用内存；未初始化时自动赋0；若与局部变量同名，局部变量优先。
     示例：

```c
#include <stdio.h>
int a = 2; // 全局变量
void exch() {
    a = 5; // 访问全局变量
}
int main() {
    printf("修改前a: %d\n", a);
    exch();
    printf("修改后a: %d\n", a); // 输出5，全局变量被修改
    return 0;
}
```

4.4.2 按存储类别分类

存储类别	关键字	存储区域	生命周期	特点
自动变量	auto	动态存储区	函数调用期间	局部变量默认类型，未初始化值随机
静态变量	static	静态存储区	程序运行期间	局部静态变量初始化仅一次，值保留；全局静态变量仅本源文件可用
寄存器变量	register	寄存器	函数调用期间	频繁使用的局部变量，提高效率，不能取地址
外部变量	extern	静态存储区	程序运行期间	引用其他文件的全局变量

示例：静态局部变量

```c
#include <stdio.h>
void test() {
    static int y = 20; // 静态局部变量，初始化仅一次
    int x; // 自动变量，每次调用重新初始化
    if (1) x = 10;
    printf("auto:x=%d, static:y=%d\n", x, y++);
}
int main() {
    test(); // 输出：auto:x=10, static:y=20
    test(); // 输出：auto:x=随机值, static:y=21
    return 0;
}
```

4.5 编译预处理

编译预处理是编译前的预处理操作，以`#`开头，用于简化编程、提高代码通用性，包括文件包含、宏定义、条件编译。

4.5.1 文件包含（#include）

- 功能：将指定文件内容嵌入当前源文件。
- 两种格式：
  1. `#include <文件名>`：用于系统头文件（如`#include <stdio.h>`），从系统目录查找。
  2. `#include "文件名"`：用于自定义文件（如`#include "abc.h"`），先从当前目录查找，再查系统目录。
- 示例：自定义头文件引用

```
// abc.h（头文件）
#define PI 3.14159
double c_area(double r); // 函数声明
```

```
// abc.c（源文件）
#include "abc.h"
double c_area(double r) {
    return PI * r * r; // 使用头文件中的宏定义
}
```

4.5.2 宏定义（#define）

- 功能：定义宏名替代正文，实现文本替换（预处理阶段完成）。

1. 简单宏定义：

```
#define 宏名 替代正文 // 无分号
```

示例：

```
#include <stdio.h>
#define NUM 30 // 宏定义：NUM替代30
int main() {
    printf("%d\n", NUM); // 预处理后替换为printf("%d\n", 30);
    return 0;
}
```

2. 带参数宏定义：

```
#define 宏名(参数表) 替代正文 // 宏名与括号间无空格
```

示例：求两数最小值

```
#include <stdio.h>
#define min(A,B) ((A)<(B)?(A):(B)) // 加括号避免优先级问题
int main() {
    int x=5, y=3;
    printf("最小值：%d\n", min(x+y, x*y)); // 替换为((5+3)<(5*3)?(5+3):(5*3))
    return 0;
}
```

注意：带参数宏是文本替换，非函数调用，需避免优先级陷阱（加括号）。

4.5.3 条件编译（#if...#else...#endif）

- 功能：按条件选择性编译代码，用于程序移植、调试。
- 格式：

```
#if 条件表达式
    代码段1 // 条件为真时编译
#else
    代码段2 // 条件为假时编译
#endif
```

示例：

```
#include <stdio.h>
#define DEBUG 1 // 调试模式开启
int main() {
    int a=10;
#if DEBUG
    printf("调试信息：a=%d\n", a); // 调试模式下编译
#else
    printf("a的值\n"); // 非调试模式下编译
#endif
    return 0;
}
```

Chap5 构造数据类型

核心结论：本章聚焦C语言构造数据类型的应用，涵盖数组（一维、二维、字符数组）和结构体类型两大核心，详解其定义、初始化、引用规则、内存存储特性及实际应用（排序、矩阵操作、字符串处理、数据封装），是处理批量数据和复杂结构化数据的核心基础。

5.1 数组概述

数组是由**相同数据类型**的元素按**有序排列**组成的构造类型，用于存储批量同类数据，解决单个变量无法高效处理多数据的问题。

核心特点

- 元素类型一致：所有元素的数据类型相同，占用内存大小一致。
- 存储连续：元素在内存中占用连续存储空间，数组名代表首元素地址（地址常量）。
- 下标访问：通过“数组名[下标]”访问元素，下标从0开始计数。
- 用途：存储批量数据（如学生成绩、矩阵元素、字符串），方便批量处理。

数组分类

- 按维度：一维数组（线性数据）、二维数组（矩阵数据）、多维数组。
- 按元素类型：数值数组（int、float）、字符数组（char）、结构体数组等。

5.2 一维数组

一维数组是最基础的数组形式，用于存储线性排列的批量数据。

5.2.1 定义格式

```
类型说明符 数组名[常量表达式];
```

- 类型说明符：指定数组元素的数据类型（int、float、char等）。
- 数组名：符合标识符规则，代表数组首地址。
- 常量表达式：指定数组元素个数（必须为非负常量，不能是变量）。
- 示例：

```
int a[5];      // 5个int型元素的一维数组
char c[20];    // 20个char型元素的一维数组
float x[10];    // 10个float型元素的一维数组
```

5.2.2 初始化方式

一维数组初始化有4种常见形式，未初始化的自动变量元素值随机，静态数组自动赋0：

1. 全量初始化：指定所有元素值

```
int a[5] = {1, 2, 3, 4, 5}; // a[0]=1, a[1]=2, ..., a[4]=5
```

2. 部分初始化：未指定的元素自动赋0

```
int a[5] = {1, 0, 1}; // a[0]=1, a[1]=0, a[2]=1, a[3]=0, a[4]=0
```

3. 省略数组长度：由初始化元素个数自动确定长度

```
int b[] = {1, 2, 3}; // 数组长度为3，b[0]=1, b[1]=2, b[2]=3
```

4. 静态数组初始化：静态存储区数组自动赋0

```
static int a[5]; // 所有元素均为0
```

5.2.3 元素引用规则

- 引用格式：`数组名[下标]`，下标范围为`0 ~ 数组长度-1`（超出范围会数组越界）。
- 示例：

```
int a[5] = {1,2,3,4,5};
printf("%d", a[2]); // 输出3（引用第3个元素）
a[3] = 10;          // 修改第4个元素值为10
```

- 注意：不能直接引用整个数组（如`printf("%d", a)`），需逐个元素操作。

5.2.4 输入输出

通过循环逐个读写元素，示例：

```c
#include <stdio.h>
int main() {
    int a[10], i;
    // 输入
    printf("请输入10个整数：");
    for (i = 0; i < 10; i++) {
        scanf("%d", &a[i]); // 注意取地址符&
    }
    // 输出
    printf("输出结果：");
    for (i = 0; i < 10; i++) {
        printf("%4d", a[i]);
    }
    printf("\n");
    return 0;
}
```

5.2.5 应用举例

1. 冒泡排序（从小到大）

```c
#include <stdio.h>
#define N 5
int main() {
    int a[N], i, j, med;
    // 输入数据
    printf("请输入%d个整数：", N);
    for (i = 0; i < N; i++) {
        scanf("%d", &a[i]);
    }
    // 冒泡排序
    for (i = 1; i < N; i++) { // 控制轮数（N-1轮）
        for (j = 0; j < N - i; j++) { // 每轮比较次数
            if (a[j] > a[j+1]) { // 相邻元素比较，大的后移
                med = a[j];
                a[j] = a[j+1];
                a[j+1] = med;
            }
        }
    }
    // 输出结果
    printf("排序后：");
    for (i = 0; i < N; i++) {
        printf("%4d", a[i]);
    }
    return 0;
}
```

2. 选择排序（从小到大）

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

```c
#define N 7
int main() {
    int a[N], i, j, med;
    printf("请输入%d个整数: ", N);
    for (i = 0; i < N; i++) {
        scanf("%d", &a[i]);
    }
    // 选择排序
    for (i = 0; i < N - 1; i++) { // 控制选择轮数（N-1轮）
        for (j = i + 1; j < N; j++) { // 从剩余元素中找最小值
            if (a[j] < a[i]) { // 最小值交换到当前位置
                med = a[i];
                a[i] = a[j];
                a[j] = med;
            }
        }
    }
    printf("排序后: ");
    for (i = 0; i < N; i++) {
        printf("%6d", a[i]);
    }
    return 0;
}
```

5.3 二维数组

二维数组用于存储矩阵形式的数据（行×列），本质是“数组的数组”（每个元素是一维数组）。

5.3.1 定义格式

```
类型说明符 数组名[行数][列数];
```

- 行数：二维数组的行数（可省略，由初始化自动推导）。
- 列数：二维数组的列数（不可省略）。
- 示例：

```c
float a[5][4];  // 5行4列的float型二维数组
int b[3][6];    // 3行6列的int型二维数组
char str[30][20];// 30行20列的char型二维数组（存储30个字符串）
```

5.3.2 内存存储规则

- 行优先存储：元素按“先存第0行，再存第1行……”的顺序连续存储。
- 示例（`int a[2][3]`）：
  内存顺序为：a[0][0] → a[0][1] → a[0][2] → a[1][0] → a[1][1] → a[1][2]
- 地址表示：`a[i][j]` 等价于 `*(a[i] + j)` 或 `*(*(a + i) + j)`（a为行地址，a[i]为首元素地址）。

5.3.3 初始化方式

1. 分行初始化（推荐，清晰直观）

```
int a[2][3] = {{1,2,3}, {4,5,6}}; // 全量初始化
int b[2][3] = {{1,2}, {4}};       // 部分初始化，未赋值元素为0
int c[][3] = {{1}, {4,5}};        // 省略行数，自动推导为2行
```

2. 按存储顺序初始化（不推荐，可读性差）

```
int a[2][3] = {1,2,3,4,5,6};     // 等价于分行初始化
int b[][3] = {1,2,3,4,5};       // 自动推导为2行，b[1][2]=0
```

5.3.4 元素引用

- 格式：`数组名[行下标][列下标]`，行、列下标均从0开始。
- 示例：

```
int a[3][4] = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
printf("%d", a[1][2]); // 输出7（第2行第3列元素）
a[2][3] = 15;          // 修改第3行第4列元素为15
```

5.3.5 输入输出

通过双重循环实现，外循环控制行，内循环控制列：

```
#include <stdio.h>
int main() {
    int b[3][2], i, j;
    // 输入
    printf("请输入3行2列数据：\n");
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 2; j++) {
            scanf("%d", &b[i][j]);
        }
    }
    // 输出
    printf("输出矩阵：\n");
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 2; j++) {
            printf("b[%d][%d] = %d ", i, j, b[i][j]);
        }
        printf("\n"); // 换行
    }
    return 0;
}
```

5.3.6 应用举例

1. 两3×4矩阵求和

```
#include <stdio.h>
int main() {
    int a[3][4], b[3][4], c[3][4], i, j;
```

```
    // 输入矩阵a
    printf("请输入a矩阵元素：\n");
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 4; j++) {
            scanf("%d", &a[i][j]);
        }
    }
    // 输入矩阵b
    printf("请输入b矩阵元素：\n");
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 4; j++) {
            scanf("%d", &b[i][j]);
        }
    }
    // 矩阵求和
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 4; j++) {
            c[i][j] = a[i][j] + b[i][j];
        }
    }
    // 输出结果
    printf("a矩阵：\n");
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 4; j++) {
            printf("%5d", a[i][j]);
        }
        printf("\n");
    }
    printf("b矩阵：\n");
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 4; j++) {
            printf("%5d", b[i][j]);
        }
        printf("\n");
    }
    printf("和矩阵：\n");
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 4; j++) {
            printf("%5d", c[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

2. 4×3矩阵转置（转为3×4矩阵）

```
#include <stdio.h>
int main() {
    int a[4][3], b[3][4], i, j;
    // 输入原矩阵a
    printf("请输入4行3列矩阵A：\n");
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 3; j++) {
            scanf("%d", &a[i][j]);
        }
```

```
    }
    // 转置：b[j][i] = a[i][j]
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 3; j++) {
            b[j][i] = a[i][j];
        }
    }
    // 输出原矩阵
    printf("原矩阵A：\n");
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 3; j++) {
            printf("%6d", a[i][j]);
        }
        printf("\n");
    }
    // 输出转置矩阵
    printf("转置矩阵B：\n");
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 4; j++) {
            printf("%6d", b[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

3. 求3×4矩阵最大值（函数实现）

```
#include <stdio.h>
float largest(float a[][4]) { // 列数不可省略
    float max = a[0][0];
    int i, j;
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 4; j++) {
            if (a[i][j] > max) {
                max = a[i][j];
            }
        }
    }
    return max;
}
int main() {
    float a[3][4], max;
    int i, j;
    printf("请输入3行4列矩阵：\n");
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 4; j++) {
            scanf("%f", &a[i][j]);
        }
    }
    max = largest(a);
    printf("最大值：%f\n", max);
    return 0;
}
```

5.4 字符数组

字符数组用于存储字符或字符串，字符串以`'\0'`（ASCII码0）作为结束标志。

5.4.1 定义格式

```
char 数组名[长度]; // 一维字符数组
char 数组名[行数][列数]; // 二维字符数组（存储多个字符串）
```

- 示例：

```
char str[20];      // 存储单个字符串（最多19个有效字符+1个'\0'）
char name[30][20]; // 存储30个字符串，每个字符串最多19个字符
```

5.4.2 初始化方式

1. 逐个字符初始化（无`'\0'`，需手动添加）

```
char str[4] = {'w', 'e', 'i', 'j'}; // 无结束符，不是字符串
char str[5] = {'G', 'o', 'o', 'd', '\0'}; // 手动添加结束符
```

2. 字符串常量初始化（自动添加`'\0'`）

```
char str[] = "Good"; // 长度自动为5（4个有效字符+1个'\0'）
char str[8] = "welcome"; // 等价于{'w','e','l','c','o','m','e','\0'}
```

5.4.3 字符串的输入输出

1. 逐个字符读写（用`%c`）

```
#include <stdio.h>
int main() {
    char a[3];
    int i;
    // 输入
    for (i = 0; i < 3; i++) {
        scanf("%c", &a[i]);
    }
    // 输出
    for (i = 0; i < 3; i++) {
        printf("%c", a[i]);
    }
    return 0;
}
```

2. 整体读写（用`%s`，自动识别`'\0'`）

```c
#include <stdio.h>
int main() {
    char c[10];
    scanf("%s", c); // 输入时遇空格、换行结束，无需&（数组名是地址）
    printf("%s", c); // 输出时遇'\0'结束
    return 0;
}
```

- 注意：`scanf("%s")`无法读取带空格的字符串，需用`fgets`或`scanf("%[^\n]", str)`。

5.4.4 常用字符串处理函数（需包含`<string.h>`）

函数名	功能描述	示例
`strlen(s)`	求字符串长度（不含`'\0'`）	`strlen("Good") → 4`
`strcpy(s1,s2)`	将s2复制到s1（s1需足够大）	`strcpy(str1, "Hello")`
`strcmp(s1,s2)`	比较s1和s2（ASCII码顺序），返回0（相等）、正数（s1大）、负数（s2大）	`strcmp("abc", "abd") → -1`
`strcat(s1,s2)`	将s2连接到s1末尾（s1需足够大）	`strcat(str1, "World")`
`strlwr(s)`	将s中大写字母转为小写	`strlwr("ABC") → "abc"`
`strupr(s)`	将s中小写字母转为大写	`strupr("abc") → "ABC"`

- 示例：字符串比较与连接

```c
#include <stdio.h>
#include <string.h>
int main() {
    char str1[20] = "Hello", str2[10] = "World";
    printf("长度：%d\n", strlen(str1)); // 输出5
    strcat(str1, str2);
    printf("连接后：%s\n", str1); // 输出HelloWorld
    if (strcmp(str1, "HelloWorld") == 0) {
        printf("相等\n");
    }
    return 0;
}
```

5.4.5 应用举例：统计字符串中字母、数字、空格个数

```c
#include <stdio.h>
#include <string.h>
int main() {
    char str[80];
    int letter = 0, digit = 0, blank = 0, others = 0, i;
    printf("请输入一行字符：\n");
```

```
    gets(str); // 读取带空格的字符串（或用fgets(str,80,stdin)）
    for (i = 0; str[i] != '\0'; i++) {
        if ((str[i] >= 'A' && str[i] <= 'Z') || (str[i] >= 'a' && str[i] <= 'z'))
{
            letter++;
        } else if (str[i] >= '0' && str[i] <= '9') {
            digit++;
        } else if (str[i] == ' ') {
            blank++;
        } else {
            others++;
        }
    }
    printf("字母：%d，数字：%d，空格：%d，其他：%d\n", letter, digit, blank, others);
    return 0;
}
```

5.5 结构体类型

结构体用于封装不同类型的数据（如学生信息：学号、姓名、成绩），形成一个整体。

5.5.1 结构体类型定义

三种定义方式：

1. 先定义类型，再定义变量

```
struct student { // 结构体类型名：student
    char number[10]; // 学号
    char name[20];   // 姓名
    char sex;        // 性别
    int age;         // 年龄
    float score[20]; // 成绩
    char addr[30];   // 地址
};
struct student stud1, stud2; // 定义结构体变量stud1、stud2
```

2. 定义类型的同时定义变量

```
struct student {
    char number[10];
    char name[20];
    char sex;
    int age;
    float score[20];
    char addr[30];
} stud1, stud2; // 直接定义变量
```

3. 无名结构体（仅能定义一次变量）

```
struct {
    char number[10];
    char name[20];
    char sex;
    int age;
    float score[20];
    char addr[30];
} stud1, stud2; // 无类型名，后续无法再定义该类型变量
```

5.5.2 结构体变量初始化

按成员顺序赋值，字符串需用`""`，数值直接赋值：

```
struct student {
    char number[10];
    char name[20];
    char sex;
    int age;
    float score;
} st1 = {"9708", "Liwei", 'F', 20, 95.5}; // 初始化
```

5.5.3 结构体成员引用

- 格式：`结构体变量名.成员名`（普通变量）、`结构体指针->成员名`（指针变量）
- 示例：

```
#include <stdio.h>
struct student {
    char name[20];
    int age;
    float score;
} st1 = {"Zhang San", 20, 89.0}, *p = &st1; // 指针p指向st1
int main() {
    printf("姓名：%s\n", st1.name);    // 直接引用：.
    printf("年龄：%d\n", p->age);     // 指针引用：->
    printf("成绩：%f\n", (*p).score); // 等价于p->score
    return 0;
}
```

5.5.4 结构体数组

用于存储多个结构体变量（如多个学生信息）：

```
#include <stdio.h>
struct student {
    long num;
    char name[20];
    float score;
} stud[3] = {
    {9701, "Li Ming", 98.0},
    {9702, "Wang Dan", 95.0},
    {9703, "Li Hui", 80.0}
};
int main() {
```

```
    int i;
    for (i = 0; i < 3; i++) {
        printf("学号：%ld，姓名：%s，成绩：%f\n", stud[i].num, stud[i].name,
stud[i].score);
    }
    return 0;
}
```

5.5.5 应用举例：结构体作为函数参数（求学生成绩平均分）

```
#include <stdio.h>
#define N 3
struct Score {
    char name[20];
    double chin, math, eng;
    double mean;
};
// 计算平均分
void getMean(struct Score *p) {
    p->mean = (p->chin + p->math + p->eng) / 3;
}
int main() {
    struct Score stu[N];
    int i;
    // 输入成绩
    for (i = 0; i < N; i++) {
        printf("请输入第%d个学生信息（姓名 语文 数学 英语）：\n", i+1);
        scanf("%s %lf %lf %lf", stu[i].name, &stu[i].chin, &stu[i].math,
&stu[i].eng);
        getMean(&stu[i]);
    }
    // 输出平均分≥80的学生
    printf("平均分≥80的学生：\n");
    for (i = 0; i < N; i++) {
        if (stu[i].mean >= 80) {
            printf("%s：%.2f\n", stu[i].name, stu[i].mean);
        }
    }
    return 0;
}
```

Chap6 指针

核心结论：本章聚焦C语言指针的核心概念与应用，涵盖指针变量的定义、初始化、算术运算，指针与数组、字符串、函数、结构体的关联，以及多级指针、动态内存分配等核心知识点，是实现高效内存操作、模块化编程的关键技术，也是C语言的核心难点与重点。

6.1 指针概述

指针是指向内存单元的变量，其核心价值在于直接操作内存地址，提高程序执行效率、灵活处理数组与结构体等复杂数据类型。

核心概念

- 内存地址：内存中每个字节的唯一编号（如0x0012FF7C），用于标识数据存储位置。
- 指针变量：存储内存地址的变量，通过该地址可间接访问对应内存中的数据。
- 指针与数据的关系：指针变量存储数据的地址，解引用操作（`*`）可通过地址获取/修改数据。

指针的作用

- 直接操作内存，提高数据访问效率（尤其适用于大数据量场景）。
- 灵活处理数组、字符串、结构体等复杂数据类型，简化代码。
- 实现函数间的数据双向传递（突破值传递的局限）。
- 支持动态内存分配，按需使用内存空间。

6.2 指针变量的定义与使用

指针变量的使用需遵循"定义→初始化→引用"的流程，核心是掌握指针的定义格式、地址操作与解引用。

6.2.1 指针变量的定义

格式：`数据类型 *指针变量名;`

- 数据类型：指针指向的目标数据的类型（决定解引用时访问的字节数）。
- `*`：标识该变量为指针类型。
- 示例：

```
int *p;       // 指向int型数据的指针变量
char *q;      // 指向char型数据的指针变量
double *r;    // 指向double型数据的指针变量
```

6.2.2 指针变量的初始化

指针变量需初始化后再使用（避免野指针），初始化方式有3种：

1. 指向已定义变量的地址：

```
int a = 10;
int *p = &a;  // &a获取变量a的地址，赋值给指针p
```

2. 指向数组元素：

```
int arr[5] = {1,2,3,4,5};
int *p = arr; // 数组名arr是首元素地址，等价于&arr[0]
```

3. 赋值为空指针（避免野指针）：

```c
int *p = NULL; // NULL是系统定义的空指针常量（值为0）
```

6.2.3 指针变量的引用

- 解引用运算符（`*`）：通过指针地址访问对应数据。
- 取地址运算符（`&`）：获取变量的内存地址。
- 示例：

```c
#include <stdio.h>
int main() {
    int a = 20;
    int *p = &a;  // 初始化指针p，指向a的地址

    printf("a的地址：%p\n", &a);    // 输出a的地址，%p用于打印地址
    printf("p存储的地址：%p\n", p); // 输出p的值（即a的地址）
    printf("通过p访问a的值：%d\n", *p); // 解引用，输出20

    *p = 30; // 通过指针修改a的值
    printf("修改后a的值：%d\n", a); // 输出30
    return 0;
}
```

6.2.4 指针的算术运算

指针的算术运算与普通变量不同，运算结果取决于指向的数据类型（步长 = 数据类型字节数）：

1. 指针自增（`p++`）/自减（`p--`）：指向相邻的下一个/上一个同类型数据。
2. 指针加减整数（`p + n`/`p - n`）：指向偏移n个步长的同类型数据。
3. 指针相减（`p1 - p2`）：仅适用于指向同一数组的指针，结果为两指针间的元素个数。

- 示例：

```c
#include <stdio.h>
int main() {
    int arr[3] = {10,20,30};
    int *p = arr; // p指向arr[0]

    printf("*p = %d\n", *p);   // 10（arr[0]）
    p++; // 指向arr[1]，步长4字节（int占4字节）
    printf("*p = %d\n", *p);   // 20（arr[1]）
    p += 1; // 指向arr[2]
    printf("*p = %d\n", *p);   // 30（arr[2]）

    int *p1 = &arr[0], *p2 = &arr[2];
    printf("p2 - p1 = %d\n", p2 - p1); // 2（两指针间有2个元素）
    return 0;
}
```

6.2.5 指针的比较运算

指针可进行`==`、`!=`、`<`、`>`等比较运算，仅适用于指向同一连续内存区域（如同一数组）的指针：

```
if (p1 == p2) {
    printf("两指针指向同一地址\n");
}
if (p1 < p2) {
    printf("p1指向的地址在p2之前\n");
}
```

6.3 指针与数组

数组与指针密切相关，数组名本质是指向数组首元素的**指针常量**（不可修改地址），指针可灵活访问数组元素。

6.3.1 数组名与指针的关系

- 数组名`arr`等价于`&arr[0]`（首元素地址），是指针常量（不能进行`arr++`等修改地址的操作）。
- 指针变量可指向数组首元素，通过指针运算访问任意元素。
- 核心等价关系：`arr[i]` ≡ `*(arr + i)` ≡ `*(p + i)` ≡ `p[i]`（p为指向数组首元素的指针变量）。

6.3.2 指针访问数组元素

两种方式：下标法（`p[i]`）和指针法（`*(p + i)`），示例：

```
#include <stdio.h>
int main() {
    int arr[5] = {1,2,3,4,5};
    int *p = arr; // 指针指向数组首元素

    // 指针法访问
    for (int i = 0; i < 5; i++) {
        printf("%d ", *(p + i)); // 输出1 2 3 4 5
    }
    printf("\n");

    // 下标法访问（指针变量支持下标运算）
    for (int i = 0; i < 5; i++) {
        printf("%d ", p[i]); // 输出1 2 3 4 5
    }
    return 0;
}
```

6.3.3 指针与二维数组

二维数组本质是“数组的数组”，数组名`arr`是指向首行（一维数组）的指针（行指针）。

- 核心关系：
    - `arr` ≡ `&arr[0]`（首行地址，行指针）。

    - `arr[i]` ≡ `*(arr + i)`（第i行首元素地址，列指针）。
    - `arr[i][j]` ≡ `*(arr[i] + j)` ≡ `*(*(arr + i) + j)`（第i行第j列元素）。
- 示例：

```
#include <stdio.h>
int main() {
    int arr[2][3] = {{1,2,3}, {4,5,6}};
    int (*p)[3] = arr; // 行指针，指向含3个int元素的一维数组

    // 访问arr[1][2]（值为6）
    printf("%d\n", arr[1][2]);
    printf("%d\n", *(arr[1] + 2));
    printf("%d\n", *(*(arr + 1) + 2));
    printf("%d\n", p[1][2]);
    return 0;
}
```

6.3.4 指针数组

指针数组是“元素为指针的数组”，格式：`数据类型 *数组名[常量表达式];`

- 用途：存储多个字符串（避免二维字符数组的空间浪费）、多个数组的地址等。
- 示例：指针数组存储字符串

```
#include <stdio.h>
int main() {
    char *strArr[] = {"Beijing", "Shanghai", "Guangzhou"}; // 指针数组，每个元素指向字符串常量
    int n = sizeof(strArr) / sizeof(strArr[0]); // 数组长度3

    for (int i = 0; i < n; i++) {
        printf("%s\n", strArr[i]); // 输出每个字符串
    }
    return 0;
}
```

6.4 指针与字符串

字符串的本质是字符数组，指针可更灵活地操作字符串（比数组名更灵活，可修改指向）。

6.4.1 指针指向字符串常量

- 格式：`char *p = "string";`（p指向字符串首字符地址，字符串常量存储在只读区，不可修改）。
- 示例：

```c
#include <stdio.h>
int main() {
    char *p = "Hello World";
    printf("%s\n", p); // 输出Hello World（printf通过p指向的地址遍历到'\0'）
    printf("%c\n", *(p + 6)); // 输出W（访问第7个字符）

    // p[0] = 'h'; // 错误：字符串常量不可修改
    return 0;
}
```

6.4.2 指针操作字符数组（可修改字符串）

- 指针指向字符数组首元素，可通过指针修改数组元素（字符串内容）。
- 示例：指针修改字符串

```c
#include <stdio.h>
int main() {
    char str[] = "Hello";
    char *p = str;

    // 将字符串改为"Hi"
    *p = 'H';
    *(p + 1) = 'i';
    *(p + 2) = '\0'; // 手动添加结束符

    printf("%s\n", str); // 输出Hi
    return 0;
}
```

6.4.3 指针与字符串处理函数

字符串处理函数（如`strlen`、`strcpy`、`strcmp`）的参数本质是字符指针，示例：

```c
#include <stdio.h>
#include <string.h>
int main() {
    char str1[20], str2[] = "Hello";
    char *p1 = str1, *p2 = str2;

    strcpy(p1, p2); // 复制p2指向的字符串到p1指向的数组
    printf("str1: %s\n", p1); // 输出Hello
    printf("长度: %d\n", strlen(p1)); // 输出5（不含'\0'）

    if (strcmp(p1, p2) == 0) {
        printf("两字符串相等\n");
    }
    return 0;
}
```

6.5 指针与函数

指针作为函数参数可实现数据双向传递，返回指针的函数可返回动态分配的内存或数组地址。

6.5.1 指针作为函数参数

- 普通变量指针：实现函数间数据双向传递（突破值传递局限）。
- 数组指针：传递数组时，本质是传递数组首地址（节省内存，提高效率）。
- 示例1：指针作为参数交换两数

```
#include <stdio.h>
void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}
int main() {
    int a = 10, b = 20;
    swap(&a, &b); // 传递变量地址
    printf("a = %d, b = %d\n", a, b); // 输出a=20, b=10
    return 0;
}
```

- 示例2：指针传递数组（求数组和）

```
#include <stdio.h>
int sumArr(int *arr, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += *(arr + i); // 指针访问数组元素
    }
    return sum;
}
int main() {
    int arr[] = {1,2,3,4,5};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("数组和：%d\n", sumArr(arr, n)); // 传递数组名（首地址）
    return 0;
}
```

6.5.2 返回指针的函数

- 格式：`数据类型 *函数名(参数表);`
- 注意：返回的指针不能指向函数内的局部变量（函数调用结束后局部变量内存释放），可返回全局变量、静态变量或动态分配的内存地址。
- 示例：返回动态分配的数组地址

```
#include <stdio.h>
#include <stdlib.h>
int *createArr(int n) {
    int *p = (int *)malloc(n * sizeof(int)); // 动态分配内存
    if (p == NULL) {
        printf("内存分配失败\n");
```

```
        exit(1);
    }
    // 初始化数组
    for (int i = 0; i < n; i++) {
        *(p + i) = i + 1;
    }
    return p;
}
int main() {
    int *arr = createArr(5);
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]); // 输出1 2 3 4 5
    }
    free(arr); // 释放动态内存
    arr = NULL; // 避免野指针
    return 0;
}
```

6.5.3 函数指针

- 函数指针是指向函数的指针，存储函数的入口地址。
- 格式：`返回值类型 (*指针名)(参数类型表);`
- 用途：实现函数的动态调用（如回调函数）。
- 示例：函数指针调用函数

```
#include <stdio.h>
int add(int a, int b) {
    return a + b;
}
int sub(int a, int b) {
    return a - b;
}
int main() {
    int (*funcPtr)(int, int); // 定义函数指针
    funcPtr = add; // 指向add函数
    printf("3 + 5 = %d\n", funcPtr(3, 5)); // 调用add，输出8

    funcPtr = sub; // 指向sub函数
    printf("3 - 5 = %d\n", funcPtr(3, 5)); // 调用sub，输出-2
    return 0;
}
```

6.6 指针与结构体

结构体指针是指向结构体变量的指针，通过`->`运算符访问结构体成员，比直接使用结构体变量更高效（传递地址而非整个结构体）。

6.6.1 结构体指针的定义与初始化

- 格式：`struct 结构体名 *指针名;`
- 初始化：指针指向结构体变量的地址。
- 示例：

```
#include <stdio.h>
struct Student {
    char name[20];
    int age;
    float score;
};
int main() {
    struct Student stu = {"Zhang San", 20, 90.5};
    struct Student *p = &stu; // 结构体指针指向stu

    // 访问结构体成员：两种方式
    printf("姓名：%s\n", (*p).name); // (*p)等价于stu，用.访问
    printf("年龄：%d\n", p->age);   // 结构体指针用->访问（推荐）
    printf("成绩：%f\n", p->score);
    return 0;
}
```

6.6.2 结构体指针作为函数参数

- 传递结构体指针，避免结构体变量的拷贝（节省内存），可修改结构体内容。
- 示例：

```
#include <stdio.h>
struct Student {
    char name[20];
    int age;
};
void updateAge(struct Student *p, int newAge) {
    p->age = newAge; // 修改结构体成员
}
int main() {
    struct Student stu = {"Li Si", 19};
    updateAge(&stu, 20);
    printf("修改后年龄：%d\n", stu.age); // 输出20
    return 0;
}
```

6.7 多级指针

多级指针是指向指针的指针，常用于处理指针数组、二维数组指针等场景，最常用的是二级指针。

6.7.1 二级指针的定义与使用

- 格式：`数据类型 **指针名;`（**表示二级指针）
- 核心关系：`**p` 访问最终指向的数据。
- 示例：

```
#include <stdio.h>
int main() {
    int a = 10;
    int *p = &a; // 一级指针，指向a
    int **pp = &p; // 二级指针，指向p

    printf("a的地址：%p\n", &a);
    printf("p存储的地址：%p\n", p);
    printf("pp存储的地址：%p\n", pp);
    printf("通过pp访问a：%d\n", **pp); // 输出10

    **pp = 20; // 通过二级指针修改a的值
    printf("修改后a：%d\n", a); // 输出20
    return 0;
}
```

6.7.2 二级指针与指针数组

二级指针常用于处理指针数组（如传递多个字符串），示例：

```
#include <stdio.h>
void printStrings(char **strArr, int n) {
    for (int i = 0; i < n; i++) {
        printf("%s\n", *(strArr + i)); // 二级指针访问指针数组元素
    }
}
int main() {
    char *strArr[] = {"Apple", "Banana", "Orange"};
    int n = sizeof(strArr) / sizeof(strArr[0]);
    printStrings(strArr, n); // 指针数组名是二级指针（指向指针的指针）
    return 0;
}
```

6.8 指针应用举例

6.8.1 动态内存分配（malloc/free）

- `malloc`：动态分配指定字节数的内存，返回void*指针（需强制类型转换）。
- `free`：释放动态分配的内存，避免内存泄漏。
- 示例：动态分配结构体数组

```
#include <stdio.h>
#include <stdlib.h>
struct Book {
```

```
    char title[50];
    float price;
};
int main() {
    int n;
    printf("请输入图书数量：");
    scanf("%d", &n);

    // 动态分配n个Book结构体的内存
    struct Book *books = (struct Book *)malloc(n * sizeof(struct Book));
    if (books == NULL) {
        printf("内存分配失败\n");
        return 1;
    }

    // 输入图书信息
    for (int i = 0; i < n; i++) {
        printf("请输入第%d本书的标题和价格：", i+1);
        scanf("%s %f", books[i].title, &books[i].price);
    }

    // 输出图书信息
    printf("\n图书列表：\n");
    for (int i = 0; i < n; i++) {
        printf("标题：%s，价格：%.2f\n", (books + i)->title, (books + i)->price);
    }

    free(books); // 释放内存
    books = NULL; // 避免野指针
    return 0;
}
```

6.8.2 指针实现字符串反转

```
#include <stdio.h>
#include <string.h>
void reverseStr(char *str) {
    char *left = str;
    char *right = str + strlen(str) - 1; // 指向字符串末尾（不含'\0'）
    char temp;

    while (left < right) {
        // 交换左右指针指向的字符
        temp = *left;
        *left = *right;
        *right = temp;
        left++;
        right--;
    }
}
int main() {
    char str[] = "Hello World";
    reverseStr(str);
    printf("反转后：%s\n", str); // 输出dlroW olleH
    return 0;
```

```
}
```

6.8.3 指针实现冒泡排序

```c
#include <stdio.h>
void bubbleSort(int *arr, int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (*(arr + j) > *(arr + j+1)) {
                // 交换相邻元素
                int temp = *(arr + j);
                *(arr + j) = *(arr + j+1);
                *(arr + j+1) = temp;
            }
        }
    }
}
int main() {
    int arr[] = {5,2,9,1,5,6};
    int n = sizeof(arr) / sizeof(arr[0]);
    bubbleSort(arr, n);

    printf("排序后：");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]); // 输出1 2 5 5 6 9
    }
    return 0;
}
```

Chap7 字符数组

核心结论：本章聚焦C语言字符数组与字符串的核心应用，涵盖字符数组的定义、初始化、输入输出，字符串与指针的关联，常用字符串处理函数的使用，以及字符数组、指针作为函数参数的实战场景，是处理文本数据、实现字符串操作的核心知识点，也是模块化编程中传递文本数据的关键技术。

7.1 字符数组

字符数组是存储字符数据的构造类型，核心用于存放字符串（以`'\0'`结尾的字符序列），其使用需掌握定义、初始化、输入输出及字符串处理函数的核心规则。

7.1.1 字符数组的定义

格式：`char 数组名[字符元素个数];`

- 数组名：符合标识符规则，代表字符数组的首地址（地址常量）。
- 字符元素个数：指定数组可存储的最大字符数（包含`'\0'`）。
- 示例：

```c
char str[20]; // 定义可存储20个字符的字符数组
char msg[10]; // 定义可存储10个字符的字符数组
```

7.1.2 字符数组的初始化

两种核心方式，初始化时需注意`'\0'`的自动添加规则：

1. 逐个字符赋值：

```
char str[4] = {'w', 'e', 'i', 'j'}; // 数组长度4，无自动添加'\0'
char str2[5] = {'h', 'e', 'l', 'l', 'o'}; // 无'\0'，非字符串形式
```

2. 字符串直接赋值（推荐）：

```
char str[] = "good morning"; // 自动在末尾添加'\0'，数组长度=字符串长度+1
char str2[8] = "welcome"; // 等价于{'w','e','l','c','o','m','e','\0'}
```

- 关键说明：字符串形式初始化会自动添加`'\0'`（字符串结束标志），数组长度默认包含该结束符；逐个字符赋值需手动添加`'\0'`才能作为字符串使用。

7.1.3 用字符数组存放字符串

- 字符串本质：以`'\0'`（ASCII码值0）结尾的字符序列，系统通过`'\0'`判断字符串结束。
- 存储示例：`char ch[] = "Good";`存储结构为`G o o d \0`，数组长度为5（而非4）。
- 字符串长度：有效字符个数（不含`'\0'`），需通过`strlen`函数获取。

7.1.4 字符数组的输入输出

支持逐个字符和整体（字符串）两种方式，需注意输入时的分隔符和结束标志：

1. 逐个字符输入输出：

```
#include <stdio.h>
void main() {
    char a[3];
    int i;
    // 输入
    for (i = 0; i < 3; i++)
        scanf("%c", &a[i]);
    // 输出
    for (i = 0; i < 3; i++)
        printf("%c", a[i]);
}
```

    - 说明：空格、回车会被当作输入字符，需注意输入格式。

2. 整体输入输出（字符串形式）：

```
#include <stdio.h>
void main() {
    char c[4];
    // 输入：遇空格、回车结束，自动添加'\0'
    scanf("%s", c);
    // 输出：遇'\0'结束，无需地址符&（数组名即首地址）
    printf("%s", c);
}
```

- 关键问题：`scanf("%s")`无法读取带空格的字符串，解决方案：
  - 使用`gets`函数（不推荐，存在缓冲区溢出风险）：

```
char str[13];
gets(str); // 可读取带空格的字符串，回车结束
```

  - 使用`fgets`函数（推荐，限制字符数）：

```
char name[20];
fgets(name, 20, stdin); // 最多读19个字符，保留换行符，自动加'\0'
```

  - 使用`scanf("%[^\n]", str)`：

```
char str[13];
scanf("%[^\n]", str); // 遇回车结束，可读取带空格的字符串
```

7.1.5 字符串处理标准函数

需包含头文件`#include <string.h>`（`puts`、`gets`除外），核心函数如下：

1. 字符串输出函数`puts()`
   - 格式：`puts(字符数组名/字符串常量)`
   - 功能：输出字符串至终端，遇`'\0'`结束，并自动换行（将`'\0'`转为`'\n'`）。
   - 示例：

```
#include <stdio.h>
void main() {
    char c[] = "BASIC\ndBASE";
    puts(c); // 可识别转义字符，输出时换行
}
```

2. 字符串输入函数`gets()`（已淘汰）
   - 格式：`gets(字符数组名)`
   - 功能：从终端读字符串至字符数组，回车结束，自动将`'\n'`转为`'\0'`。
   - 缺陷：无缓冲区大小检测，易导致内存溢出，推荐用`fgets`替代。
3. 字符串长度函数`strlen()`
   - 格式：`strlen(字符数组名/字符串常量)`
   - 功能：返回字符串有效字符个数（不含`'\0'`）。
   - 示例：

```
#include <stdio.h>
#include <string.h>
void main() {
    char ch[20] = "12$%ert \619\0pop";
    printf("长度：%d\n", strlen(ch)); // 输出10（'\0'前的有效字符）
    puts(ch); // 输出12$%ert 19（\61是'1'的八进制ASCII码）
}
```

4. 字符串复制函数`strcpy()`
   - 格式：`strcpy(目标字符数组, 源字符串)`
   - 功能：将源字符串（含`'\0'`）复制到目标字符数组，目标数组需足够大。
   - 示例：

```c
#include <stdio.h>
#include <string.h>
void main() {
    char st1[15], st2[] = "C Language";
    strcpy(st1, st2);
    puts(st1); // 输出C Language
}
```

5. 字符串比较函数`strcmp()`
   - 格式：`strcmp(字符串1, 字符串2)`
   - 功能：按ASCII码逐字符比较，返回值：
     - 0：两字符串相等；
     - 正数：字符串1 > 字符串2；
     - 负数：字符串1 < 字符串2。
   - 注意：不可用`==`直接比较字符串，需用该函数。
   - 示例：

```c
#include <stdio.h>
#include <string.h>
void main() {
    char str1[] = "abc", str2[] = "abd";
    if (strcmp(str1, str2) < 0)
        printf("str1 < str2\n"); // 输出该结果
}
```

6. 字符串大小写转换函数
   - `strlwr(字符数组)`：将大写字母转为小写；
   - `strupr(字符数组)`：将小写字母转为大写。
   - 示例：

```c
#include <stdio.h>
#include <string.h>
void main() {
    char str[] = "Hello World";
    strlwr(str);
    printf("%s\n", str); // 输出hello world
    strupr(str);
    printf("%s\n", str); // 输出HELLO WORLD
}
```

7. 字符串连接函数`strcat()`
   - 格式：`strcat(目标字符数组, 源字符串)`

- 功能：将源字符串连接到目标字符串末尾，目标数组需足够大（覆盖目标字符串的`'\0'`）。
- 示例：

```
#include <stdio.h>
#include <string.h>
void main() {
    char ch1[20] = "aaa", ch2[20] = "bbbb";
    strcat(ch1, ch2);
    printf("%s\n", ch1); // 输出aaabbbb
}
```

7.2 字符串与指针

C语言中字符串可通过字符数组或字符指针两种方式表示，指针操作字符串更灵活，但需注意字符串常量的只读特性。

7.2.1 字符串的两种表示形式

1. 字符数组形式：

```
char str[] = "programming"; // 数组名是首地址（常量），可修改数组元素
```

   - 特点：字符串存储在栈区，可修改字符内容；数组长度=字符串长度+1（含`'\0'`）。

2. 字符指针形式：

```
char *pstr = "programming"; // 指针指向字符串常量（只读区）
// 或分步赋值
char *pstr;
pstr = "programming";
```

   - 特点：指针是变量，可重新赋值指向其他字符串；字符串常量存储在只读区，不可修改字符内容。

7.2.2 两种表示形式的核心区别

特性	字符数组（`char str[]`）	字符指针（`char *pstr`）
存储位置	栈区（可修改）	只读数据区（不可修改）
变量性质	地址常量（不可赋值）	指针变量（可重新赋值）
内存占用	字符串长度+1字节	4字节（32位系统）/8字节（64位系统）
示例操作	`str[0] = 'P'`（合法）	`pstr[0] = 'P'`（非法）

7.2.3 指针操作字符串示例

1. 指针指向字符数组（可修改字符串）：

```c
#include <stdio.h>
void main() {
    char str[] = "Hello";
    char *p = str;
    *p = 'H';           // 修改首字符为'H'
    *(p + 1) = 'i';     // 修改第二个字符为'i'
    *(p + 2) = '\0';    // 手动添加结束符
    printf("%s\n", str); // 输出Hi
}
```

2. 指针指向字符串常量（不可修改）：

```c
#include <stdio.h>
void main() {
    char *p = "Hello World";
    printf("%s\n", p);       // 输出Hello World
    printf("%c\n", *(p + 6)); // 输出W
    // p[0] = 'h'; // 错误：字符串常量不可修改
}
```

3. 指针复制字符串：

```c
#include <stdio.h>
void main() {
    char a[80], b[80], *pa = a, *pb = b;
    scanf("%s", pa);
    while (*pa != '\0')
        *pb++ = *pa++; // 逐字符复制，指针后移
    *pb = '\0'; // 手动添加结束符
    printf("%s\n", b);
}
```

4. 指针输出字符串部分内容：

```c
#include <stdio.h>
void main() {
    char *s1 = "C language";
    s1 += 2; // 指针指向第3个字符（'l'）
    printf("%s\n", s1); // 输出language
}
```

7.3 字符数组、指针作为函数参数

字符数组名或字符指针可作为函数参数，实现字符串的传递与处理，核心是利用地址传递特性，避免字符串拷贝，提高效率。

7.3.1 字符数组作为函数参数

示例1：字符串反序（字符数组参数）

```
#include <stdio.h>
#include <string.h>
void fanxu(char chuan[]) {
    int i = 0, n = strlen(chuan);
    char med;
    for (i = 0; i < n / 2; i++) {
        med = chuan[i];
        chuan[i] = chuan[n - i - 1];
        chuan[n - i - 1] = med;
    }
}
void main() {
    char ch[20];
    printf("请输入一个字符串（长度不超过20）：\n");
    fgets(ch, 20, stdin);
    printf("原始字符串：\n");
    puts(ch);
    fanxu(ch);
    printf("反序后字符串：\n");
    puts(ch);
}
```

示例2：统计字符个数（字符数组参数）

```
#include <stdio.h>
void main() {
    char str[80];
    int letter = 0, digit = 0, blank = 0, others = 0, i;
    printf("请输入一行字符：\n");
    gets(str);
    for (i = 0; str[i] != '\0'; i++) {
        if ((str[i] >= 'A' && str[i] <= 'Z') || (str[i] >= 'a' && str[i] <= 'z'))
            letter++;
        else if (str[i] >= '0' && str[i] <= '9')
            digit++;
        else if (str[i] == ' ')
            blank++;
        else
            others++;
    }
    printf("字母：%d，数字：%d，空格：%d，其他：%d\n", letter, digit, blank, others);
}
```

7.3.2 字符指针作为函数参数

示例1：字符串反序（指针参数）

```
#include <stdio.h>
#include <string.h>
void fanxu(char *m) {
    int n = strlen(m);
```

```
    char med, *p = m, *q = m + n - 1;
    for (; p < m + n / 2; p++, q--) {
        med = *p;
        *p = *q;
        *q = med;
    }
}
void main() {
    char ch[20];
    printf("请输入一个字符串（长度不超过20）：\n");
    gets(ch);
    printf("原始字符串：\n");
    puts(ch);
    fanxu(ch);
    printf("反序后字符串：\n");
    puts(ch);
}
```

示例2：删除指定字符（指针参数）

```
#include <stdio.h>
void del_ch(char *p, char ch) {
    char *q = p;
    for (; *p != '\0'; p++) {
        if (*p != ch) {
            *q = *p;
            q++;
        }
    }
    *q = '\0'; // 手动添加结束符
}
void main() {
    char str[80], *pt, ch;
    printf("请输入一个字符串：\n");
    gets(str);
    pt = str;
    printf("请输入要删除的字符：\n");
    ch = getchar();
    del_ch(pt, ch);
    printf("新字符串：\n%s\n", str);
}
```

示例3：字符串复制（指针参数）

```c
#include <stdio.h>
void strcopy(char *from, char *to) {
    while (*from != '\0')
        *(to++) = *(from++);
    *to = '\0';
}
void main() {
    char a[80], b[80], *pa = a, *pb = b;
    scanf("%[^\n]", pa);
    strcopy(pa, pb);
    printf("%s\n", pb);
}
```

Chap8 结构体

核心结论：本章聚焦C语言结构体的核心应用，涵盖结构体类型的定义、变量初始化与引用，结构体数组的使用，typedef类型别名定义，以及指针与结构体的深度结合（结构体指针、指针作为函数参数），是处理多类型关联数据（如学生信息、商品数据）的核心工具，也是实现复杂数据组织与模块化编程的关键知识点。

8.1 结构体类型

结构体是由不同类型成员组成的构造数据类型，核心用于描述具有多个关联属性的复杂对象（如学生、商品），其使用需掌握定义、初始化、引用及输入输出的完整流程。

8.1.1 结构体概述

- 结构体的本质：将多个不同类型的数据（成员）封装为一个整体，描述“属性集合型”数据（如学生包含学号、姓名、成绩等属性）。
- 核心优势：解决单一基本类型无法描述复杂对象的问题，使数据组织更贴近实际场景。
- 结构体的一般格式：

```c
struct 结构体类型名 {
    类型标识符 成员名1;
    类型标识符 成员名2;
    // ... 其他成员
};
```

- 示例（学生信息结构体）：

```c
struct student {
    char number[10];  // 学号
    char name[20];    // 姓名
    char sex;         // 性别
    int age;          // 年龄
    float score[20];  // 成绩
    char addr[30];    // 地址
};
```

8.1.2 结构体类型与变量的定义

三种核心定义形式，按需选择使用：

1. 形式一：先定义结构体类型，再定义变量

```
// 先定义类型
struct student {
    char name[20];
    int age;
    float score;
};
// 再定义变量
struct student stud1, stud2;
```

2. 形式二：定义类型的同时定义变量

```
struct student {
    char name[20];
    int age;
    float score;
} stud1, stud2;  // 直接定义变量
```

3. 形式三：无名结构体类型（仅用于一次性变量）

```
struct {
    char name[20];
    int age;
    float score;
} stud1, stud2;  // 无类型名，无法复用
```

8.1.3 结构体变量的初始化

按成员顺序赋值，支持嵌套结构体初始化，未赋值成员自动为0（数值类型）或空（字符类型）：

```
// 基础初始化
struct stud {
    long num;
    char name[20];
    char sex;
    char addr[30];
} st1 = {9708, "Liwei", 'F', "44BeijingRoad"};

// 嵌套结构体初始化（日期+人员信息）
struct date {
    int month;
    int day;
    int year;
};
struct person {
    char name[20];
    struct date birthday;
    int age;
} p1 = {"Zhang San", {10, 1, 2000}, 24};
```

8.1.4 结构体变量的引用

通过"成员运算符（.）"访问成员，嵌套结构体需逐级访问：

```
#include <stdio.h>
void main() {
    struct student {
        char name[20];
        int age;
        float score;
    } stu = {"Li Si", 19, 89.5};

    // 访问普通成员
    printf("姓名：%s\n", stu.name);
    printf("年龄：%d\n", stu.age);
    stu.score = 92.0;  // 修改成员值
    printf("成绩：%.1f\n", stu.score);

    // 嵌套结构体访问
    struct date { int m, d, y; };
    struct person {
        char name[20];
        struct date birth;
    } p = {"Wang Wu", {5, 20, 2001}};
    printf("生日：%d年%d月%d日\n", p.birth.y, p.birth.m, p.birth.d);
}
```

8.1.5 结构体变量的输入输出

C语言不支持结构体整体输入输出，需逐个成员操作：

```
#include <stdio.h>
void main() {
    struct student {
        char name[20];
        int age;
        float score;
    } stu;

    // 输入
    printf("请输入姓名、年龄、成绩：\n");
    scanf("%s %d %f", stu.name, &stu.age, &stu.score);  // 字符数组无需&

    // 输出
    printf("姓名：%s\n年龄：%d\n成绩：%.1f\n", stu.name, stu.age, stu.score);
}
```

8.2 结构体数组

结构体数组是元素为结构体类型的数组，核心用于存储多个同类型复杂对象（如多个学生、多个商品）。

8.2.1 结构体数组的定义

三种定义形式，与结构体变量定义逻辑一致：

1. 先定义类型，再定义数组

```
struct student {
    long num;
    char name[20];
    float score;
};
struct student stud[100];  // 100个学生的数组
```

2. 定义类型同时定义数组

```
struct student {
    long num;
    char name[20];
    float score;
} stud[100];
```

3. 无名类型直接定义数组

```
struct {
    long num;
    char name[20];
    float score;
} stud[100];
```

8.2.2 结构体数组的初始化

按数组元素顺序，逐个结构体赋值，支持部分成员初始化：

```
struct student {
    long num;
    char name[20];
    int age;
    float score;
} stud[3] = {
    {9701, "LiMing", 20, 98.0},    // 第1个元素
    {9702, "WangDan", 20, 95.0},   // 第2个元素
    {9703, "LiHui", 19, 80.0}     // 第3个元素
};
```

8.2.3 结构体数组的引用与输入输出

通过"数组下标+成员运算符"访问元素成员，输入输出需遍历数组：

```
#include <stdio.h>
void main() {
    struct student {
        long num;
        char name[20];
        float score;
    } stud[3] = {
        {9701, "LiMing", 98.0},
        {9702, "WangDan", 95.0},
        {9703, "LiHui", 80.0}
    };

    // 输出所有学生信息
    for (int i = 0; i < 3; i++) {
        printf("学号：%ld，姓名：%s，成绩：%.1f\n",
               stud[i].num, stud[i].name, stud[i].score);
    }

    // 输入新学生信息
    struct student new_stud[2];
    for (int i = 0; i < 2; i++) {
        printf("请输入第%d个学生的学号、姓名、成绩：\n", i+1);
        scanf("%ld %s %f", &new_stud[i].num, new_stud[i].name,
&new_stud[i].score);
    }
}
```

8.3 定义类型（typedef）

typedef用于为已有类型定义别名，简化复杂类型描述（如结构体、数组、指针），提升代码可读性和可移植性。

8.3.1 typedef的基本格式

```
typedef 原类型 新类型名;
```

8.3.2 常见应用场景

1. 基本类型别名

   ```
   typedef unsigned long int ULI;
   ULI x = 123456789;  // 等价于 unsigned long int x;
   ```

2. 数组类型别名

```
typedef double VECT4[4];  // 定义“4个double元素的数组”类型
VECT4 v1, v2;  // 等价于 double v1[4], v2[4];

typedef double MAT[5][5];  // 5×5双精度数组类型
MAT a1, a2;  // 等价于 double a1[5][5], a2[5][5];
```

3. 指针类型别名

```
typedef int *IP;
IP p1, p2;  // 等价于 int *p1, *p2;（注意：不同于#define MIP int *，后者会导致p2为int类型）
```

4. 结构体类型别名（最常用）

```
// 方式1：先定义结构体，再typedef
struct student {
    char name[20];
    int age;
};
typedef struct student SD;

// 方式2：定义结构体同时typedef
typedef struct student {
    char name[20];
    int age;
} SD;

// 方式3：无名结构体typedef
typedef struct {
    char name[20];
    int age;
} SD;

SD stu1, stu2;  // 简化结构体变量定义，无需写struct
```

8.3.3 typedef与#define的区别

特性	typedef	#define
本质	类型别名定义（编译阶段处理）	文本替换（预处理阶段处理）
作用范围	局部或全局（遵循作用域规则）	全局（从定义处到文件结束）
指针类型处理	可正确定义多个指针	可能导致错误（如#define MIP int *; MIP p1,p2; 中p2为int类型）
数组类型处理	可定义数组类型别名	仅文本替换，无法定义数组类型

8.4 指针与结构体

结构体指针是指向结构体变量/数组的指针，通过“指向运算符（->）”访问成员，比直接使用结构体变量更高效（传递地址而非整个结构体）。

8.4.1 指向结构体变量的指针

1. 定义与初始化

```
struct student {
    char name[20];
    int age;
    float score;
} stu = {"Zhang San", 20, 90.5}, *p;
p = &stu;  // 指针指向结构体变量stu
```

2. 访问成员的两种方式

```
#include <stdio.h>
void main() {
    struct student {
        char name[20];
        int age;
        float score;
    } stu = {"Zhang San", 20, 90.5}, *p = &stu;

    // 方式1：(*p).成员名（括号不可省，.优先级高于*）
    printf("姓名：%s\n", (*p).name);

    // 方式2：p->成员名（结构体指针专用，推荐）
    printf("年龄：%d\n", p->age);
    printf("成绩：%.1f\n", p->score);

    // 修改成员值
    p->age = 21;
    (*p).score = 92.0;
    printf("修改后：年龄%d，成绩%.1f\n", p->age, p->score);
}
```

8.4.2 指向结构体数组的指针

结构体数组名是数组首元素地址，指针可通过算术运算遍历数组：

```
#include <stdio.h>
void main() {
    struct student {
        long num;
        char name[20];
        float score;
    } stud[3] = {
        {9701, "LiMing", 98.0},
        {9702, "WangDan", 95.0},
        {9703, "LiHui", 80.0}
    }, *p = stud;  // 指向数组首元素
```

```
    // 遍历数组（三种方式等价）
    for (int i = 0; i < 3; i++) {
        printf("学号：%ld，姓名：%s，成绩：%.1f\n",
               (p+i)->num,  // 指针偏移
               p[i].name,   // 数组下标
               (*(p+i)).score);  // 解引用+成员运算符
    }
}
```

8.4.3 结构体指针作为函数参数

传递结构体指针，避免结构体拷贝（节省内存），支持修改结构体内容，是结构体函数参数的首选方式。

1. 示例1：计算学生成绩总分与平均分

```
#include <stdio.h>
#define N 5
// 定义结构体类型
typedef struct stud {
    char name[10];
    int s[3];  // 三门课成绩
    int sum;
    float ave;
} STUD;

// 结构体指针作为参数
void count(STUD *p) {
    p->sum = 0;
    for (int j = 0; j < 3; j++) {
        p->sum += p->s[j];
    }
    p->ave = p->sum / 3.0;
}

void main() {
    STUD a[N];
    int i;
    // 输入学生信息
    for (i = 0; i < N; i++) {
        printf("请输入第%d个学生的姓名和三门课成绩：\n", i+1);
        scanf("%s %d %d %d", a[i].name, &a[i].s[0], &a[i].s[1], &a[i].s[2]);
        count(&a[i]);  // 传递结构体地址
    }
    // 输出结果
    for (i = 0; i < N; i++) {
        printf("姓名：%10s，成绩：%4d %4d %4d，总分：%6d，平均分：%.1f\n",
               a[i].name, a[i].s[0], a[i].s[1], a[i].s[2], a[i].sum, a[i].ave);
    }
}
```

2. 示例2：结构体数组指针遍历（打印学生信息）

```
#include <stdio.h>
typedef struct student {
```

```
    long num;
    char name[20];
    char sex;
    int age;
    double score;
    char addr[30];
} STU;

// 结构体数组指针作为参数
void printStu(STU *p, int n) {
    for (; p < p + n; p++) {
        printf("学号：%ld，姓名：%s，性别：%c，年龄：%d，成绩：%.2f，地址：%s\n",
               p->num, p->name, p->sex, p->age, p->score, p->addr);
    }
}

void main() {
    STU a[3] = {
        {99641, "Li Ping", 'M', 20, 56.0, "Tianjin Street"},
        {99341, "Zhang Fan", 'F', 21, 78.0, "Beijing Road"},
        {99441, "Ren Zhong", 'M', 19, 84.0, "Shenyang Road"}
    };
    printStu(a, 3);  // 传递数组名（首元素地址）
}
```

8.4.4 结构体指针的进阶应用（动态内存分配）

结合`malloc`动态分配结构体内存，灵活处理不确定数量的结构体数据：

```
#include <stdio.h>
#include <stdlib.h>
typedef struct BOOK {
    char Name[50];
    double Price;
    int Pages;
} BOOK;

// 动态分配n本图书内存
BOOK *createBooks(int n) {
    BOOK *p = (BOOK *)malloc(n * sizeof(BOOK));
    if (p == NULL) {
        printf("内存分配失败\n");
        exit(1);
    }
    return p;
}

// 释放动态内存
void freeBooks(BOOK *p) {
    free(p);
    p = NULL;
}

void main() {
    int n;
```

```
    printf("请输入图书数量：");
    scanf("%d", &n);
    BOOK *books = createBooks(n);

    // 输入图书信息
    for (int i = 0; i < n; i++) {
        printf("请输入第%d本书的书名、价格、页数：\n", i+1);
        scanf("%s %lf %d", books[i].Name, &books[i].Price, &books[i].Pages);
    }

    // 输出图书信息
    for (int i = 0; i < n; i++) {
        printf("书名：%s，价格：%.2f，页数：%d\n",
               (books+i)->Name, (books+i)->Price, (books+i)es);
    }

    freeBooks(books);
}
```

Chap9 指针高级应用与动态数据结构

核心结论：本章聚焦C语言指针的高级应用与动态数据管理，涵盖指针与函数的深度结合（返回指针的函数、函数指针、函数指针作为参数）、指针数组的定义与应用、命令行参数处理、复杂类型描述解读、二级指针，以及动态内存分配（malloc/calloc/realloc/free）与动态数据结构，是实现灵活模块化编程、动态数据管理和高效内存利用的核心知识点，为复杂程序开发提供关键技术支撑。

9.1 指针与函数

指针与函数的结合是模块化编程的核心技巧，包括返回指针的函数、指向函数的指针（函数指针），以及函数指针作为参数，实现函数的动态调用与灵活复用。

9.1.1 返回指针值的函数

函数的返回值可以是指向任意数据类型的指针，核心用于返回动态分配的内存、数组地址或复杂数据结构的地址。

- 定义格式：`数据类型 *函数名(参数表);`
  - 说明：`*` 与函数名绑定，函数返回指向指定类型的指针。
- 注意事项：不可返回函数内局部变量的指针（局部变量栈区内存会随函数结束释放），可返回全局变量、静态变量或动态分配的内存地址。
- 示例：返回数组指定位置的指针

```c
#include <stdio.h>
int *search(int *x, int n) {
    int *p;
    p = x + n; // 指向数组x的第n个元素（下标从0开始）
    return p;
}

void main() {
    int arr[] = {10, 20, 30, 40, 50};
    int *p = search(arr, 2); // 指向arr[2]
    printf("arr[2] = %d\n", *p); // 输出30
}
```

9.1.2 函数指针（指向函数的指针）

函数指针是指向函数入口地址的指针，可通过指针变量调用函数，实现函数的动态切换。

- 定义格式：`返回值类型 (*指针名)(参数类型表);`
  - 关键区别：`int (*p)()` 是函数指针（指向返回int的函数），`int *p()` 是返回int指针的函数。
- 函数调用形式：`(*指针名)(实参表列);` 或直接 `指针名(实参表列);`（等价）
- 示例：用函数指针调用求最小值函数

```c
#include <stdio.h>
int min(int x, int y) {
    return x < y ? x : y;
}

void main() {
    int (*funcPtr)(int, int); // 定义函数指针
    funcPtr = min; // 指向min函数（函数名即入口地址）
    int a = 15, b = 8;
    int m = (*funcPtr)(a, b); // 调用min函数
    printf("min(%d, %d) = %d\n", a, b, m); // 输出8
}
```

9.1.3 函数指针作为函数参数

将函数指针作为参数传入其他函数，可在被调函数中动态调用不同功能的函数，提升代码灵活性。

- 应用场景：实现通用处理函数（如process函数可动态调用max/min/add）
- 示例：用函数指针参数实现多功能计算

```c
#include <stdio.h>
// 待传入的功能函数
int max(int x, int y) { printf("max="); return x > y ? x : y; }
int min(int x, int y) { printf("min="); return x < y ? x : y; }
int add(int x, int y) { printf("sum="); return x + y; }

// 接收函数指针参数的通用函数
void process(int x, int y, int (*fun)(int, int)) {
    int result = fun(x, y); // 调用传入的函数
    printf("%d\n", result);
```

```
}

void main() {
    int a = 10, b = 5;
    process(a, b, max);  // 调用max，输出max=10
    process(a, b, min);  // 调用min，输出min=5
    process(a, b, add);  // 调用add，输出sum=15
}
```

9.2 指针数组

指针数组是元素为指针的数组，核心用于存储多个字符串地址或数组地址，比二维数组更灵活（节省内存、便于修改指向）。

9.2.1 指针数组的定义

- 格式：`数据类型 *数组名[常量表达式];`
  - 说明：`[]` 优先级高于 `*`，数组的每个元素都是指向指定类型的指针。
- 与数组指针的区别：
  - 指针数组 `int *a[4]`：4个元素，每个是int指针（存储地址）；
  - 数组指针 `int (*a)[4]`：1个指针，指向含4个int元素的一维数组。

9.2.2 指针数组的初始化

- 核心场景：存储多个字符串（字符串常量地址）
- 示例：字符串指针数组初始化

```
#include <stdio.h>
void main() {
    // 指针数组存储5个字符串的地址
    char *strArr[] = {"Beijing", "Jiaotong", "University", "C", "Language"};
    printf("strArr[1] = %s\n", strArr[1]);    // 输出Jiaotong
    printf("*(strArr+2) = %s\n", *(strArr+2));// 输出University
    printf("*strArr[3] = %c\n", *strArr[3]);  // 输出C（取第3个指针指向的首字符）
}
```

9.2.3 指针数组的应用

- 典型场景：字符串排序（通过交换指针指向实现，无需移动字符串本身，效率高）
- 示例：指针数组实现字符串按字母顺序排序

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

// 排序函数：按ASCII码升序排序
void sort(char *name[], int n) {
    char *temp;
    int i, j, k;
    for (i = 0; i < n-1; i++) {
        k = i;
```

```
        for (j = i+1; j < n; j++) {
            // 比较两个字符串
            if (strcmp(name[k], name[j]) > 0) {
                k = j; // 记录当前最小字符串的下标
            }
        }
        // 交换指针指向（不移动字符串）
        if (k != i) {
            temp = name[i];
            name[i] = name[k];
            name[k] = temp;
        }
    }
}

// 打印函数
void print(char *name[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%s\n", name[i]);
    }
}

void main() {
    char *name[] = {"Follow me", "BASIC", "Great Wall", "FORTRAN",
"Computer"};
    int n = 5;
    sort(name, n);
    print(name, n);
    // 输出结果：BASIC、Computer、Follow me、FORTRAN、Great Wall
}
```

9.3 命令行参数

C程序可通过`main`函数的参数接收命令行输入的参数，实现程序的外部配置（如指定输入输出文件、运行模式）。

9.3.1 命令行参数的格式

- `main`函数的参数原型：`int main(int argc, char *argv[]);`
  - `argc`（argument count）：命令行参数的个数（包含程序名本身，最小为1）；
  - `argv`（argument vector）：字符指针数组，每个元素指向一个命令行参数字符串；
    - `argv[0]`：程序可执行文件名；
    - `argv[1]~argv[argc-1]`：用户输入的参数；
    - `argv[argc]`：NULL（结束标志）。

9.3.2 示例：输出所有命令行参数

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
    printf("参数个数 argc = %d\n", argc);
    for (int i = 0; i < argc; i++) {
        printf("argv[%d]: %s\n", i, argv[i]);
    }
    return 0;
}
```

- 运行命令：`./program.exe hello world 123`
- 输出结果：

```
参数个数 argc = 4
argv[0]: ./program.exe
argv[1]: hello
argv[2]: world
argv[3]: 123
```

9.3.3 注意事项

- 命令行参数以空格分隔；
- 若参数包含空格，需用双引号包裹（如`"hello world"`视为一个参数）；
- IDE中设置命令行参数：需在项目配置的“调试”选项中指定“程序参数”。

9.4 复杂类型描述与解读

C语言中复杂类型（如函数指针、指针数组、数组指针）的描述需遵循“右侧优先法则”，从标识符出发，先右后左解析。

9.4.1 解读原则

1. 找到被定义的标识符，从标识符向外解析；
2. 优先级：`[]`（数组）和`()`（函数）高于`*`（指针）；
3. 结合性：`[]`和`()`从左向右，`*`从右向左。

9.4.2 常见复杂类型示例

类型描述	解读结果
`int (*p)[4]`	p是指针，指向含4个int元素的一维数组
`int *p[4]`	p是数组，含4个int指针元素
`int (*p)(int)`	p是函数指针，指向返回int、参数为int的函数
`int *(*p)(int)`	p是函数指针，指向返回int指针、参数为int的函数
`char **argv`	argv是指针，指向char指针（二级指针）

9.5 指向指针的指针（二级指针）

二级指针是指向指针变量的指针，核心用于处理指针数组、动态二维数组等场景。

9.5.1 定义与初始化

- 格式：`数据类型 **指针名;`
- 核心关系：
  - `**pp`：访问最终指向的数据；
  - `*pp`：访问中间指针变量（一级指针）；
  - `pp`：二级指针本身（存储一级指针的地址）。
- 示例：二级指针访问int变量

```c
#include <stdio.h>
void main() {
    int a = 10;
    int *p = &a;    // 一级指针，指向a
    int **pp = &p;  // 二级指针，指向p

    printf("a = %d\n", a);
    printf("*p = %d\n", *p);        // 等价于a
    printf("**pp = %d\n", **pp);    // 等价于a
    printf("&p = %p, pp = %p\n", &p, pp); // 地址相同
}
```

9.5.2 应用场景：处理指针数组

```c
#include <stdio.h>
void main() {
    char *strArr[] = {"Apple", "Banana", "Orange"};
    char **pp = strArr; // 二级指针指向指针数组首元素（strArr[0]的地址）

    for (int i = 0; i < 3; i++) {
        printf("strArr[%d]: %s\n", i, *(pp + i)); // 等价于strArr[i]
    }
}
```

9.6 动态数据结构与动态内存分配

静态数据结构（如数组）大小固定，动态内存分配通过`malloc/calloc/realloc/free`函数在堆区分配内存，实现灵活的内存管理（按需分配、释放）。

9.6.1 动态内存分配概述

- 堆区：程序运行时系统分配的内存区域，大小不固定，需手动申请和释放；
- 核心函数：需包含头文件`#include <stdlib.h>`；
- 常见场景：动态数组、动态结构体数组、链表等数据结构。

9.6.2 动态内存分配函数

1. `malloc`：分配指定字节数的内存（未初始化）
   - 原型：`void *malloc(unsigned int size);`
   - 示例：动态分配10个int元素的数组

```
#include <stdio.h>
#include <stdlib.h>
void main() {
    int n = 10;
    int *arr = (int *)malloc(n * sizeof(int)); // 强制类型转换为int*
    if (arr == NULL) { // 检查分配是否成功
        printf("内存分配失败\n");
        exit(1);
    }
    // 初始化数组
    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;
    }
    // 输出
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]); // 输出1 2 ... 10
    }
    free(arr); // 释放内存
    arr = NULL; // 避免野指针
}
```

2. `calloc`：分配n个指定大小的内存（初始化为0）
   - 原型：`void *calloc(unsigned int n, unsigned int size);`
   - 示例：动态分配5个double元素的数组（初始为0）

```
double *dp = (double *)calloc(5, sizeof(double));
if (dp != NULL) {
    for (int i = 0; i < 5; i++) {
        printf("%lf ", dp[i]); // 输出0.000000 0.000000 ...
    }
}
free(dp);
dp = NULL;
```

3. `realloc`：重新分配已动态分配的内存（扩容/缩容）
   - 原型：`void *realloc(void *p, unsigned int new_size);`
   - 注意：
     - 若原内存后有足够空间，直接扩容；否则分配新内存并复制原数据，释放原内存；
     - 分配失败返回NULL，原内存不释放，需保存原指针。
   - 示例：将原数组扩容为15个int元素

```
int *arr = (int *)malloc(10 * sizeof(int));
int *new_arr = (int *)realloc(arr, 15 * sizeof(int));
if (new_arr != NULL) {
```

```
        arr = new_arr; // 指向新内存
        // 初始化新增元素
        for (int i = 10; i < 15; i++) {
            arr[i] = i + 1;
        }
    } else {
        printf("扩容失败\n");
        free(arr); // 释放原内存
    }
    free(arr);
    arr = NULL;
```

4. `free`：释放动态分配的内存
    - 原型：`void free(void *p);`
    - 注意：
        - 仅能释放堆区动态分配的内存，不可释放栈区变量（如局部数组）；
        - 释放后指针变为野指针，需置为NULL；
        - 不可重复释放同一内存。

9.6.3 动态内存分配的常见错误

- 内存泄漏：动态分配的内存未释放，程序运行期间持续占用；
- 野指针：释放内存后未置NULL，继续访问该指针；
- 越界访问：动态数组访问超出分配的大小；
- 重复释放：同一内存被多次调用`free`

Chap10 文件操作

核心结论：本章聚焦C语言文件操作的核心技术，基于ANSI C标准的**缓冲文件系统**，通过文件指针（FILE*）实现文件的打开-操作-关闭完整流程，涵盖文件分类、顺序读写（字符/字符串/数据块/格式化）、随机读写（位置定位）及文件错误检测，是实现数据持久化存储与读取的关键知识点，适用于文本文件和二进制文件的各类操作场景。

10.1 文件概述

文件是存储在外部介质（如磁盘）上的数据集合，操作系统以文件为单位管理数据。C语言通过库函数操作文件，核心依赖缓冲文件系统和文件指针。

10.1.1 文件分类

按不同标准可分为三类，核心区别如下：

分类标准	具体类别	特点/示例
按内容	程序文件（.c/.exe/.obj）	存储代码或可执行指令，如C源文件、可执行文件
	数据文件（.txt/.dat）	存储程序运行数据，如文本数据、二进制数据

分类标准	具体类别	特点/示例
按组织形式	顺序存取文件	只能按顺序读写，不可跳转（如磁带文件）
	随机存取文件	可通过指针定位任意位置读写（如磁盘文件）
按存储形式	ASCII码文件（文本文件）	字符以ASCII码存储，直观可读写，占空间大；如记事本文件
	二进制文件	按内存存储形式原样存储，占空间小、速度快，不可直接读写；如.exe/.obj文件

- 存储对比示例（十进制数12345）：
  - ASCII文件：占5字节，存储为`'1'('00110001')、'2'('00110010')、'3'('00110011')、'4'('00110100')、'5'('00110101')`
  - 二进制文件：占4字节（int型），存储为内存中整数12345的二进制形式

10.1.2 C语言文件处理方法

- 缓冲文件系统（ANSI C标准唯一支持）：
  - 系统自动为每个文件开辟**512字节缓冲区**，数据先写入缓冲区，装满后再写入磁盘；读取时先从磁盘读入缓冲区，再逐字节读取到变量。
  - 优势：减少磁盘I/O次数，提高效率，由操作系统自动管理缓冲区。
- 非缓冲文件系统：需手动设置缓冲区，ANSI C不推荐，仅部分系统支持。

10.1.3 文件类型指针（FILE*）

文件操作的核心是通过**文件指针**关联文件缓冲区，所有文件操作均通过指针实现。

1. FILE结构体（系统定义）：
   存储文件相关信息（文件号、缓冲区状态、读写位置等），定义如下：

```
typedef struct {
    int _fd;          // 文件号
    int _cleft;      // 缓冲区中剩余字符数
    int _mode;      // 文件操作方式
    char *_next;     // 文件当前读写位置
    char *_buff;     // 文件缓冲区地址
} FILE;
```

2. 文件指针定义：
   - 格式：`FILE *指针变量名;`
   - 示例：`FILE *fp;`（fp为文件指针，指向FILE结构体变量，关联文件缓冲区）
   - 多文件操作：每个文件对应独立缓冲区和文件指针，互不干扰。

10.2 文件的打开和关闭

文件操作必须遵循“**先打开，后操作，最后关闭**”的流程，核心函数为`fopen`（打开）和`fclose`（关闭）。

10.2.1 文件打开（fopen函数）

- 功能：申请文件缓冲区，建立文件与指针的关联，返回文件指针。
- 格式：`FILE *fopen(const char *文件名, const char *使用方式);`
  - 文件名：含路径（如`"d:\\file\\test.txt"`，双反斜杠转义），默认路径与程序同级。
  - 使用方式：指定文件操作类型（读/写/追加、文本/二进制），核心类型如下：

使用方式	含义	注意事项
"r"	只读打开文本文件	文件不存在则打开失败
"w"	只写创建文本文件	文件存在则覆盖原有内容，不存在则创建
"a"	追加打开文本文件	从文件末尾写入，不存在则创建
"rb"	只读打开二进制文件	-
"wb"	只写创建二进制文件	-
"ab"	追加打开二进制文件	-
"r+"	读写打开文本文件	文件必须存在
"w+"	读写创建文本文件	覆盖原有内容或创建新文件
"a+"	读写打开文本文件	从末尾写入，可读取全文
"rb+"	读写打开二进制文件	-
"wb+"	读写创建二进制文件	-
"ab+"	读写打开二进制文件	-

- 返回值：成功返回文件指针（非NULL），失败返回`NULL`（需判断打开状态）。
- 示例：打开文本文件（判断是否成功）

```c
#include <stdio.h>
#include <stdlib.h>
void main() {
    FILE *fp;
    // 打开当前目录下的f1.txt，只读方式
    if ((fp = fopen("f1.txt", "r")) == NULL) {
        printf("无法打开文件！\n");
        exit(1); // 终止程序，返回1表示出错
    }
    // 文件操作...
    fclose(fp); // 关闭文件
}
```

10.2.2 文件关闭（fclose函数）

- 功能：断开指针与文件的关联，释放缓冲区数据到磁盘，回收缓冲区内存。
- 格式：`int fclose(FILE *fp);`
- 返回值：成功返回`0`，失败返回`EOF`（-1）。
- 注意事项：
  - 必须关闭文件，否则缓冲区数据可能丢失。
  - 关闭后指针变为野指针，不可再操作。
- 示例：关闭文件

```
FILE *fp = fopen("test.txt", "w");
// 写入操作...
if (fclose(fp) != 0) {
    printf("文件关闭失败！\n");
}
```

10.2.3 标准文件（系统自动打开）

程序运行时，系统自动打开3个标准文件，无需手动fopen：

标准文件	文件指针	对应设备	用途
标准输入	stdin	键盘	输入数据
标准输出	stdout	显示器	输出数据
标准出错输出	stderr	显示器	输出错误信息

- 示例：使用stdin读取字符串

```
#include <stdio.h>
void main() {
    char str[100];
    fgets(str, 100, stdin); // 从键盘读取字符串
    printf("%s", str);
}
```

10.3 文件的操作

文件操作分为**顺序读写**（按指针顺序移动）和**随机读写**（手动定位指针），核心是通过各类读写函数操作数据。

10.3.1 顺序读写

按文件内部位置指针的自然顺序读写，指针自动向后移动，核心函数如下：

1. 字符读写（fgetc / fputc）

- 读字符：`int fgetc(FILE *fp);`
  - 功能：从fp指向的文件读取一个字符，返回字符ASCII码（unsigned char转换为int）。
  - 结束标志：读到文件末尾或出错返回`EOF`（-1）。
  - 示例：读文本文件并输出到屏幕

```
#include <stdio.h>
#include <stdlib.h>
void main() {
    FILE *fp;
    char ch;
    if ((fp = fopen("f1.txt", "r")) == NULL) {
        printf("无法打开文件！\n");
        exit(1);
    }
    // 循环读取字符，直到EOF
    while ((ch = fgetc(fp)) != EOF) {
        putchar(ch); // 输出到屏幕
    }
    printf("\n");
    fclose(fp);
}
```

- 写字符：`int fputc(int ch, FILE *fp);`
  - 功能：将字符ch（int型参数，实际取低8位）写入fp指向的文件。
  - 返回值：成功返回写入的字符，失败返回`EOF`。
  - 示例：从键盘输入字符，写入文件c2.txt

```
#include <stdio.h>
#include <stdlib.h>
void main() {
    FILE *fp;
    char ch;
    if ((fp = fopen("c2.txt", "w+")) == NULL) {
        printf("无法创建文件！\n");
        exit(1);
    }
    printf("输入字符串（回车结束）：\n");
    ch = getchar();
    while (ch != '\n') {
        fputc(ch, fp); // 写入文件
        ch = getchar();
    }
    fclose(fp);
}
```

2. 字符串读写（fgets / fputs）

- 读字符串：`char *fgets(char *buff, int n, FILE *fp);`
  - 功能：从fp读取`n-1`个字符到buff数组，遇换行或EOF提前结束，自动添加`'\0'`。
  - 返回值：成功返回buff首地址，失败或EOF返回`NULL`。
  - 示例：读文件中10个字符的字符串并输出

```
#include <stdio.h>
#include <stdlib.h>
void main() {
    FILE *fp;
    char str[11]; // 存10个有效字符+'\0'
    if ((fp = fopen("c3.txt", "r")) == NULL) {
        printf("无法打开文件！\n");
        exit(1);
    }
    fgets(str, 11, fp); // 最多读10个字符
    puts(str); // 输出字符串
    fclose(fp);
}
```

- 写字符串：`int fputs(const char *str, FILE *fp);`
  - 功能：将str指向的字符串（不含`'\0'`）写入fp指向的文件。
  - 返回值：成功返回`0`，失败返回`EOF`。
  - 示例：从键盘输入字符串，写入文件

```
#include <stdio.h>
#include <stdlib.h>
void main() {
    FILE *fp;
    char st[20];
    if ((fp = fopen("c3.txt", "w+")) == NULL) {
        printf("无法创建文件！\n");
        exit(1);
    }
    printf("输入字符串：\n");
    gets(st); // 读键盘输入
    fputs(st, fp); // 写入文件
    fclose(fp);
}
```

3. 数据块读写（fread / fwrite）

多用于**二进制文件**，按数据块批量读写，效率高。

- 读数据块：`size_t fread(void *buffer, size_t size, size_t count, FILE *fp);`
- 写数据块：`size_t fwrite(const void *buffer, size_t size, size_t count, FILE *fp);`
  - 参数：
    - buffer：数据存储地址（读时存数据，写时取数据）。

        - size：单个数据块的字节数（如`sizeof(int)`）。
        - count：数据块个数。
    - 返回值：成功读写的数据块个数（等于count为正常）。
- 示例1：将100个整数写入二进制文件，再读出

```
#include <stdio.h>
#include <stdlib.h>
void main() {
    int Iarr[100], myarr[100], i;
    FILE *fp;
    // 初始化数组
    for (i = 0; i < 100; i++) Iarr[i] = i;
    // 写入二进制文件
    if ((fp = fopen("test.dat", "wb")) == NULL) {
        printf("无法创建文件！\n");
        exit(1);
    }
    fwrite(Iarr, sizeof(int), 100, fp);
    fclose(fp);
    // 读出文件
    if ((fp = fopen("test.dat", "rb")) == NULL) {
        printf("无法打开文件！\n");
        exit(2);
    }
    if (fread(myarr, sizeof(int), 100, fp) != 100) {
        printf("数据读取不完整！\n");
        exit(3);
    }
    fclose(fp);
    // 输出结果
    for (i = 0; i < 100; i++) printf("%d ", myarr[i]);
}
```

- 示例2：结构体数组的二进制读写（商品信息）

```
#include <stdio.h>
#include <stdlib.h>
// 商品结构体
typedef struct GoodInfo {
    int nID;          // 商品ID
    char szName[20];  // 商品名称
    double dPrice;    // 商品价格
} GOODINFO;
// 商品列表结构体
typedef struct GoodList {
    GOODINFO Goods[100]; // 最多100种商品
    int nNumber;         // 实际商品数
} GOODLIST;

// 保存商品列表到文件
int SaveGoodListToFile(GOODLIST *gList, char *szFileName) {
    FILE *fp;
    if (gList->nNumber == 0) {
        printf("商品列表为空！\n");
```

```
        return -1;
    }
    fp = fopen(szFileName, "wb+");
    if (fp == NULL) {
        printf("无法创建文件！\n");
        return -2;
    }
    // 先写商品个数，再写商品数据
    fwrite(&gList->nNumber, sizeof(int), 1, fp);
    fwrite(gList->Goods, sizeof(GOODINFO), gList->nNumber, fp);
    fclose(fp);
    return 0;
}

// 从文件读取商品列表
int ReadGoodListFromFile(GOODLIST *gList, char *szFileName) {
    FILE *fp = fopen(szFileName, "rb");
    if (fp == NULL) return -1;
    int n;
    // 读商品个数
    if (fread(&n, sizeof(int), 1, fp) < 1) {
        printf("文件无数据！\n");
        fclose(fp);
        return -2;
    }
    gList->nNumber = n;
    // 读商品数据
    int nCount = fread(gList->Goods, sizeof(GOODINFO), n, fp);
    if (nCount < n) {
        printf("数据不完整！\n");
        fclose(fp);
        return -3;
    }
    fclose(fp);
    return 0;
}
```

4. 格式化读写（fscanf / fprintf）

按指定格式读写文件，类似`scanf`/`printf`，但操作对象是文件。

- 读格式：`int fscanf(FILE *fp, const char *format, ...);`
- 写格式：`int fprintf(FILE *fp, const char *format, ...);`
- 示例1：写入整数和字符串到文件

```c
#include <stdio.h>
#include <stdlib.h>
void main() {
    FILE *fp;
    int i = 617;
    char *s = "That's good news";
    if ((fp = fopen("test.txt", "w")) == NULL) {
        printf("无法创建文件！\n");
        exit(1);
    }
    fprintf(fp, "%d%s", i, s); // 格式化写入
    fclose(fp);
}
```

- 示例2：读写整数、实数、字符

```c
#include <stdio.h>
#include <stdlib.h>
void main() {
    int a, a1;
    float b, b1;
    char c, c1;
    FILE *fp;
    // 读写模式打开文件
    if ((fp = fopen("new.txt", "w+")) == NULL) {
        printf("无法创建文件！\n");
        exit(1);
    }
    // 从键盘输入并写入文件
    printf("请输入整数、实数、字符（用逗号分隔）：\n");
    scanf("%d,%f,%c", &a, &b, &c);
    fprintf(fp, "%d,%f,%c", a, b, c);
    fclose(fp);
    // 从文件读出并输出
    fp = fopen("new.txt", "r");
    fscanf(fp, "%d,%f,%c", &a1, &b1, &c1);
    printf("原输入：%d,%f,%c\n", a, b, c);
    printf("文件读出：%d,%f,%c\n", a1, b1, c1);
    fclose(fp);
}
```

- 示例3：学生成绩的格式化读写

```c
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
// 学生成绩结构体
typedef struct StudentScores {
    char szID[9];    // 学号
    char szName[20]; // 姓名
    double dScores[3]; // 三门课成绩
} STUDENTSCORES;
// 班级成绩结构体
typedef struct ClassScore {
    STUDENTSCORES *pStudents; // 学生数组指针
```

```
    int nCount;              // 学生个数
} CLASSSCORE;

// 从文件读入班级成绩
void ReadScoresFromFile(CLASSSCORE *pClass, char *szFileName) {
    FILE *fp = fopen(szFileName, "r");
    if (fp == NULL) return;
    // 读学生个数
    fscanf(fp, "%d", &pClass->nCount);
    // 动态分配学生数组内存
    pClass->pStudents = (STUDENTSCORES*)malloc(sizeof(STUDENTSCORES) *
pClass->nCount);
    // 逐行读学生数据
    for (int i = 0; i < pClass->nCount; i++) {
        fscanf(fp, "%s %s %lf %lf %lf",
               pClass->pStudents[i].szID,
               pClass->pStudents[i].szName,
               &pClass->pStudents[i].dScores[0],
               &pClass->pStudents[i].dScores[1],
               &pClass->pStudents[i].dScores[2]);
    }
    fclose(fp);
}

// 将班级成绩写入文件
void SaveScoresToFile(CLASSSCORE *pClass, char *szFileName) {
    FILE *fp = fopen(szFileName, "w");
    if (fp == NULL) return;
    // 写学生个数
    fprintf(fp, "%d\n", pClass->nCount);
    // 逐行写学生数据
    for (int i = 0; i < pClass->nCount; i++) {
        fprintf(fp, "%s %s %6.2lf %6.2lf %6.2lf\n",
               pClass->pStudents[i].szID,
               pClass->pStudents[i].szName,
               pClass->pStudents[i].dScores[0],
               pClass->pStudents[i].dScores[1],
               pClass->pStudents[i].dScores[2]);
    }
    fclose(fp);
}
```

10.3.2 随机读写

通过定位文件内部的**位置指针**，实现任意位置的读写，核心是文件定位函数。

1. 位置指针概述

- 文件内部有一个隐含的位置指针，指向当前读写位置：
  - 打开文件时，指针指向文件开头（“r”/“w”模式）或文件末尾（“a”模式）。
  - 顺序读写时，指针自动向后移动（移动字节数=读写数据的字节数）。
  - 随机读写时，通过函数手动调整指针位置。

2. 文件定位函数

- 指针复位：`void rewind(FILE *fp);`
  - 功能：将fp的位置指针移到文件开头。
- 指针偏移：`int fseek(FILE *fp, long offset, int base);`
  - 功能：将指针从base基准位置偏移offset字节。
  - 参数：
    - offset：偏移量（正数向后移，负数向前移）。
    - base：基准位置（宏定义）：

基准位置	宏名	数值
文件开头	SEEK_SET	0
当前指针位置	SEEK_CUR	1
文件末尾	SEEK_END	2

  - 返回值：成功返回`0`，失败返回`-1`。
- 示例：读取文件中第二个学生的数据

```
#include <stdio.h>
#include <stdlib.h>
// 学生结构体
struct stu {
    char name[10];
    int num;
    int age;
    char addr[15];
} stu1, *qq = &stu1;

void main() {
    FILE *fp;
    // 打开二进制文件
    if ((fp = fopen("stu.dat", "rb")) == NULL) {
        printf("无法打开文件！\n");
        exit(1);
    }
    rewind(fp); // 指针移到文件开头
    // 指针偏移一个学生结构体的字节数（指向第二个学生）
    fseek(fp, sizeof(struct stu), SEEK_SET);
    // 读取第二个学生数据
    fread(qq, sizeof(struct stu), 1, fp);
    // 输出结果
    printf("姓名：%s 学号：%d 年龄：%d 地址：%s\n",
           qq->name, qq->num, qq->age, qq->addr);
    fclose(fp);
}
```

10.4 文件检测

用于判断文件操作是否出错、是否到达文件末尾，核心函数为`ferror`、`clearerr`、`feof`。

10.4.1 错误检测（ferror函数）

- 格式：`int ferror(FILE *fp);`
- 功能：检测fp指向的文件操作是否出错。
- 返回值：`0`（未出错），非0（出错）。
- 示例：检测文件读写错误

```
#include <stdio.h>
void main() {
    FILE *fp;
    fp = fopen("DUMMY.FIL", "w"); // 只写模式打开
    fgetc(fp); // 尝试读文件（非法操作）
    if (ferror(fp)) { // 检测错误
        printf("文件操作出错！\n");
        clearerr(fp); // 清除错误标志
    }
    fclose(fp);
}
```

10.4.2 清除错误标志（clearerr函数）

- 格式：`void clearerr(FILE *fp);`
- 功能：将fp的“错误标志”和“文件结束标志”置为`0`。
- 注意：`ferror`检测到错误后，标志会保持，需手动清除。

10.4.3 文件结束检测（feof函数）

- 格式：`int feof(FILE *fp);`
- 功能：检测fp指向的文件是否到达末尾。
- 返回值：`0`（未结束），非0（已结束）。
- 注意：`fgetc`返回`EOF`可能是出错或文件结束，需用`feof`区分。
- 示例：区分文件结束和出错

```
#include <stdio.h>
void main() {
    FILE *fp;
    char ch;
    fp = fopen("test.txt", "r");
    while (1) {
        ch = fgetc(fp);
        if (feof(fp)) { // 检测是否文件结束
            printf("文件读取完毕！\n");
            break;
        }
        if (ferror(fp)) { // 检测是否出错
            printf("读取错误！\n");
            break;
        }
```

```
        putchar(ch);
    }
    fclose(fp);
}
```

10.5 总结与应用实例

10.5.1 常见错误

1. 打开/关闭方式不匹配（如“r”模式写文件）。
2. 混淆文件指针（FILE*）和位置指针（内部隐含指针）。
3. 随机读写时指针定位错误。
4. 忘记关闭文件，导致数据丢失。

10.5.2 常用文件函数汇总

分类	函数名	功能
打开关闭	fopen	打开文件
	fclose	关闭文件
定位	rewind	指针移到文件开头
	fseek	指针偏移指定字节
字符读写	fgetc	读一个字符
	fputc	写一个字符
字符串读写	fgets	读一个字符串
	fputs	写一个字符串
数据块读写	fread	读数据块（二进制）
	fwrite	写数据块（二进制）
格式化读写	fscanf	格式化读文件
	fprintf	格式化写文件
状态检测	ferror	检测操作错误
	clearerr	清除错误标志
	feof	检测文件结束

10.5.3 应用实例：文件复制并转换大小写

- 功能：将file1.txt的内容复制到file2.txt，同时将小写字母转为大写。

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

```c
void main() {
    FILE *fp1, *fp2;
    char ch;
    // 打开源文件（读）和目标文件（写）
    if ((fp1 = fopen("file1.txt", "r")) == NULL) {
        printf("无法打开源文件！\n");
        exit(1);
    }
    if ((fp2 = fopen("file2.txt", "w")) == NULL) {
        printf("无法创建目标文件！\n");
        fclose(fp1);
        exit(2);
    }
    // 循环读取并转换
    ch = fgetc(fp1);
    while (ch != EOF) {
        if (ch >= 'a' && ch <= 'z') {
            ch -= 32; // 小写转大写（ASCII码差32）
        }
        fputc(ch, fp2); // 写入目标文件
        ch = fgetc(fp1);
    }
    // 关闭文件
    fclose(fp1);
    fclose(fp2);
    printf("复制完成！\n");
}
```