# C语言复习指南

---

title: C语言复习指南
tags: [C, 编程]
created: 2026-01-02

---

# C语言基础语法

## printf函数

### 基本用法

```
printf("格式字符串", 参数1, 参数2, ...);
```

### 常用格式说明符

| 格式符 | 说明 |
| --- | --- |
| `%d` | 十进制整数 |
| `%f` | 浮点数（默认6位小数） |
| `%c` | 单个字符 |
| `%s` | 字符串 |
| `%lf` | double类型浮点数 |
| `%u` | 无符号十进制整数 |
| `%x` | 十六进制整数（小写） |
| `%X` | 十六进制整数（大写） |
| `%o` | 八进制整数 |
| `%p` | 指针地址 |
| `%%` | 输出百分号 |

### 控制格式

```
// 宽度和精度控制
printf("%5d", 10);      // "   10"（右对齐，宽度5）
printf("%-5d", 10);     // "10   "（左对齐，宽度5）
```

```c
printf("%05d", 10);      // "00010"（用0填充）
printf("%.2f", 3.1415); // "3.14"（保留2位小数）
printf("%8.2f", 3.1415); // "    3.14"（宽度8，保留2位小数）
```

# scanf函数

## 基本用法

```c
scanf("格式字符串", &变量1, &变量2, ...);
```

**注意**：除了字符串，其他类型变量前必须加 `&` 取地址符！

## 常用格式说明符

```c
// 读取单个值
int num;
scanf("%d", &num);

// 读取多个值
int a, b;
scanf("%d%d", &a, &b);  // 输入时可以用空格、Tab或换行分隔

// 读取字符串
char name[20];
scanf("%s", name);  // 注意：数组名本身是地址，不需要&

// 读取字符
char ch;
scanf(" %c", &ch);  // 前面的空格可以跳过空白字符
```

# getchar 和 putchar

## getchar

从标准输入读取一个字符（无参数，返回int类型字符或EOF）

- 从标准输入读取单个字符
- 返回类型为 `int` ,即读取到的字符

## putchar

向标准输出写入一个字符

```c
int putchar(int c);
```

- 将字符 `c` 输出到标准输出（通常是屏幕）

- 返回输出的字符

例：

```
c = getchar();
putchar(c);
putchar('\n');
```

## 转义字符

转义字符是以反斜杠 \ 开头的特殊字符序列，用于表示一些无法直接输入的字符或有特殊意义的字符。

| 转义字符 | 含义 | ASCII码 | 示例 |
| --- | --- | --- | --- |
| `\n` | 换行符 | 10 | `printf("Hello\nWorld");` |
| `\\` | 反斜杠 | 92 | `printf("C:\\Windows");` |
| `\'` | 单引号 | 39 | `char ch = '\'';` |
| `\"` | 双引号 | 34 | `printf("\"Hello\"");` |
| `\0` | 空字符（字符串结束标志） | 0 | `char str[] = "Hello\0World";` |
| `\ooo` | 八进制表示，ooo是1-3位八进制数字 | | `char c1 = '\101' // 八进制，相当于'A'` |
| \xhh | \xhh　// 十六进制表示，hh是1-2位十六进制数字 | | `char c2 = '\x41'; // 十六进制，相当于'A'` |

```
#include <stdio.h>
#include <string.h>
int main(){
    char a[20]="6\054321\0", b[20]="123\08" ; // \054 被转义为 ','
    printf("length: %d %d\n", strlen(a), strlen(b));
    printf("%s\n%s", a, b);
    return 0;
}
//输出为:
//length: 5 3
//6,321 123
```

## 基本数据类型

### 位和字节

- **位（bit）**：二进制的基本单位，值为0或1
- **字节（byte）**：通常8位组成1个字节

- 不同变量类型占用的字节数不同：

## 整型

用于存储整数。

| 类型 | 字节数 | 格式说明符 | 范围（常见） |
| --- | --- | --- | --- |
| `int` | 4字节 | `%d` | -2,147,483,648 ~ 2,147,483,647 |
| `short` | 2字节 | `%hd` | -32,768 ~ 32,767 |
| `long` | 4或8字节 | `%ld` | 范围通常更大 |
| `char` | 1字节 | `%c`（字符）或 `%d`（ASCII值） | -128 ~ 127 |

## 浮点型

用于存储带小数点的实数。

| 类型 | 字节数 | 格式说明符 | 有效数字 |
| --- | --- | --- | --- |
| `float` | 4字节 | `%f` | 约6-7位 |
| `double` | 8字节 | `%lf` | 约15-16位 |

# 整型常量

C语言中表示整数值的常量。

## 整型常量的类型

1. 十进制整数常量：

```
int a = 123;
int b = -456;
int c = 0;
```

2. 八进制整数常量
   以 `0` 开头的整数：

```
int a = 0123;      // 八进制123，相当于十进制的83
int b = 0777;      // 八进制777，相当于十进制的511
int c = 0;         // 这也是八进制，值为0
```

3. 十六进制整数常量
   以 `0x` 或 `0X` 开头的整数

```
int a = 0x123;     // 十六进制123，相当于十进制的291
int b = 0xFF;      // 十六进制FF，相当于十进制的255
int c = 0XABCD;    // 十六进制ABCD
```

4. 二进制整数常量
   以 `0b` 或 `0B` 开头的整数（需要C99或更高版本支持）

## 整型变量的后缀

| 后缀 | 含义 | 示例 |
| --- | --- | --- |
| `u` 或 `U` | 无符号整型 | `123U`，`0xFFU` |
| `l` 或 `L` | 长整型 | `123L`，`0x123L` |
| `ll` 或 `LL` | 长长整型 | `123LL`，`0x123LL` |
| `ul` 或 `UL` | 无符号长整型 | `123UL` |
| `ull` 或 `ULL` | 无符号长长整型 | `123ULL` |

# 有符号整型变量的存储

三种**二进制表示方法**

1. 原码：

- 最高位表示符号位：**0表示正数，1表示负数**
- 其余位表示数值的绝对值

2. 反码

- **正数**的反码与原码相同
- **负数**的反码：符号位不变，其余位取反

3. 补码 **（现代计算机采用）**

- **正数**的补码与原码相同
- **负数**的补码：反码 + 1

示例：

```
+5 的原码：00000101
-5 的原码：10000101
+0 的原码：00000000
-0 的原码：10000000

+5 的反码：00000101  （与原码相同）
-5 的反码：11111010  （符号位不变，其余取反）
```

```
+0 的反码：00000000
-0 的反码：11111111

+5 的补码：00000101
-5 的补码：11111011
-5 的补码计算：
原码10000101 → 取反11111010 → 加1 → 11111011
0的补码：00000000
```

补码的**优势**：

- 解决了0的唯一性问题
- 减法可直接转化为加法

```
计算 7 - 5：
7 - 5 = 7 + (-5)

  7的补码：00000111
 -5的补码：11111011
    + ------------
         100000010  (9位，舍弃最高位)
          00000010 = 2
```

# char与ASCII码

## 什么是ASCII码

- **ASCII**（American Standard Code for Information Interchange）
- 美国信息交换标准代码，用于字符编码
- 使用7位二进制数（0-127）表示128个字符
- C语言中，char使用完整的8位，但标准ASCII只占用0-127

## ASCII码主要分区

| 范围 | 类别 | 示例 |
| --- | --- | --- |
| 0-31 | 控制字符 | '\n'(换行), '\t'(制表符) |
| 32-126 | 可打印字符 | 字母、数字、标点 |
| 127 | 控制字符 | DEL(删除) |
| 128-255 | 扩展ASCII | 非标准，依系统而定 |

## 重要ASCII值

```
十进制   字符      说明
0       '\0'    空字符（字符串结束标志）
10      \n      换行符
32      空格      空格字符
48      '0'     数字0
65      'A'     大写字母A
97      'a'     小写字母a
```

### char与ASCII的关联

- 在C语言中，`char`实际上存储的是字符对应的**ASCII码整数值**
- 字符常量用单引号括起：`'A'`等价于整数`65`

```c
char c = 'A';
printf("字符: %c\n", c);   // 输出: A
printf("ASCII值: %d\n", c); // 输出: 65
```

## 变量、命名、声明与作用域

### 命名规则

- 由字母、数字、下划线(`_`)组成
- 不能以数字开头
- 不能使用C语言的关键字（如`int`，`if`，`return`等）
- 严格区分大小写

### 声明与初始化

```c
// 先定义，后使用
int score;                      // 声明
int height = 175;               // 声明并初始化
```

**注意**：未初始化的局部变量值是随机的，直接使用可能导致错误。

### 作用域

1. **局部变量（块作用域）**：在函数或`{}`块内定义，仅在其内部可见

```c
for(int i = 0; i < 10; i++) {
        // i在此处可见
}
// i在此处不可访问
```

2. **全局变量（文件作用域）**：在函数外定义，从定义处到文件末尾都可见

## 示例代码

```c
#include<stdio.h>

int test1 = 1;                  // 全局变量 test1，值为1

void f(int test2) {             // 函数f，参数test2（局部于函数f）
    printf("test1:%d\n", test1);  // 此处test1指向全局变量（值为1）
    printf("test2:%d\n", test2);  // 此处test2指向函数参数
}

int main() {
    int test1 = 2, test2 = 3;  // 局部变量test1(2), test2(3)，屏蔽了全局test1
    printf("test1:%d\n", test1); // 输出局部test1: 2

    if(1) {                        // if块开启一个新的作用域
        int test2 = 4;             // 局部变量test2(4)，屏蔽了外层main的test2(3)
        printf("test2:%d\n", test2); // 输出if块内的test2: 4
        f(5);                      // 调用f(5)，参数test2为5
    }

    return 0;
}
```

**程序结构分析：**

- 全局变量：`int test1 = 1;` 定义在函数外部，作用域为整个程序
- 函数f：接受一个参数test2，打印全局变量test1和其自身的参数test2
- 主函数main：内部定义了局部变量，并进行了多层作用域的嵌套
- 局部变量屏蔽全局变量：在main函数内部，局部变量test1（值为2）屏蔽了同名的全局变量test1（值为1）
- 函数形参作为局部变量：函数f的参数test2是函数f的局部变量

**输出结果：**

```
test1:2
test2:4
test1:1
test2:5
```

# 静态变量和 `static` 关键字

## 静态变量的声明

```
static 变量类型 变量名
```

## 局部静态变量（在函数内部定义）

```c
void counter() {
    static int count = 0;  // 只初始化一次
    count++;
    printf("调用次数: %d\n", count);
}
int main() {
    counter();  // 输出: 调用次数: 1
    counter();  // 输出: 调用次数: 2
    counter();  // 输出: 调用次数: 3
    return 0;
}
```

**特点：**

- 生命周期贯穿整个程序运行期间
- 只初始化一次
- 作用域仍然仅限于函数内部(如无法从主函数中直接调用)

## 全局静态变量（在函数外定义）

**特点：**

- 只能在定义它的源文件中访问
- 具有文件作用域，对其他文件不可见

# 数据类型转换

## 隐式转换（自动转换）

由编译器自动按规则进行。

**核心规则**：参与运算时，值域小的类型自动向值域大的类型转换，以保证精度。

**通常顺序**：`char` / `short` → `int` → `long` → `double`

**整型提升**：`char`、`short` 等短整型在参与运算时，会先被提升为 `int` 类型。

**赋值时**：等号右边类型自动转换为左边变量的类型。

## 强制转换（显式转换）

由程序员手动指定。

**语法**：`(目标类型)表达式` 或 `(目标类型)(表达式)`

**示例：**

```c
int a = 5, b = 2;
double result = (double)a / b; // 将a强制转为double，使除法结果为浮点数2.5
```

**注意**：强转只是临时转换，不改变原变量本身的类型和值。

# 基本算术运算符

C语言提供了五种基本的算术运算符：`+`、`-`、`*`（乘法）、`/`（除法）、`%`（取模）

## 整数除法的特殊性

当`/`运算符两边的操作数都是整数时，执行的是**整除**，结果会舍弃小数部分，只保留整数部分。

```c
int result = 7 / 2;          // result 的值是 3，不是 3.5
float result2 = 7 / 2;       // 先进行整数除法得到3，再赋值给float，result2是3.0
```

要得到浮点数结果，至少需要一个操作数是浮点数：

```c
float result = 7.0 / 2;      // 正确：result = 3.5
float result = (float)7 / 2; // 正确：通过强制类型转换
```

## 取模运算符 `%` 的限制

- `%` 运算符只能用于整数类型（`int`，`char`，`long` 等）。不能对浮点数（`float`，`double`）使用
- 取模运算符的结果：符号与被除数相同，绝对值等于两数绝对值取模

```c
int remainder = 7 % 2;       // 正确：remainder = 1
// double error = 7.5 % 2;   // 错误！编译不通过

printf("%d\n", 7 % 3);       // 输出 1
printf("%d\n", -7 % 3);      // 输出 -1
printf("%d\n", 7 % -3);      // 输出 1 (商为-2，余数 = 7 - (-2)*(-3) = 1)
printf("%d\n", -7 % -3);     // 输出 -1
```

# 自增与自减运算符

- `++`：自增，变量值加一
- `--`：自减，变量值减一

## 前缀与后缀的区别

- **前缀形式**（`++a`，`--a`）：先自增/减，然后将新值用于表达式
- **后缀形式**（`a++`，`a--`）：先将原值用于表达式，然后再自增/减

```c
int a = 5, b, c;

b = ++a; // 步骤：a先加1变为6，再将6赋值给b。结果：a=6, b=6
printf("a=%d, b=%d\n", a, b); // 输出 a=6, b=6

a = 5; // 重置a
c = a++; // 步骤：先将a的当前值5赋值给c，然后a再加1变为6。结果：a=6, c=5
printf("a=%d, c=%d\n", a, c); // 输出 a=6, c=5
```

# 赋值运算符

## 基本赋值运算符 `=`

**功能**：将`=`右边表达式的值计算出来，然后存入左边变量所代表的内存空间

**格式**：`变量 = 表达式;`

**关键特性**：赋值操作本身也是一个表达式，它有自己的值，其值就是被赋的值

```c
#include<stdio.h>

int main() {
    int a, b, c;
    c = 0;
    a = b = c++;  // 相当于：b = c++; a = b;
    printf("a : %d, b : %d, c : %d", a, b, c);
    return 0;
}
// 输出结果：a : 0, b : 0, c : 1
```

## 复合赋值运算符

为了简化"先运算，再赋值"的常见操作，C语言提供了复合赋值运算符。

若`int a = 5;`：

| 运算符 | 等效于 | 示例 | 结果 |
| --- | --- | --- | --- |
| `+=` | `a = a + expr` | `a += 3;` | `a = 8` |
| `-=` | `a = a - expr` | `a -= 2;` | `a = 3` |
| `*=` | `a = a * expr` | `a *= 4;` | `a = 20` |
| `/=` | `a = a / expr` | `a /= 2;` | `a = 2` (整数除法) |
| `%=` | `a = a % expr` | `a %= 3;` | `a = 2` (5%3的余数) |

# 关系运算符

关系运算符用于比较两个表达式的值，C语言提供了6种关系运算符：

- `==` 等于
- `!=` 不等于
- `>` 大于
- `<` 小于
- `>=` 大于等于
- `<=` 小于等于

## 特点

1. **结果为整数：** `int` 类型，只有两个可能值
   - 真（True）：用整数 `1` 表示
   - 假（False）：用整数 `0` 表示

```c
printf("%d\n", 10 > 5);   // 输出 1
printf("%d\n", 10 == 5);  // 输出 0
```

2. **可参与任何整数运算**：由于结果就是0或1，它可以被用在任何需要整数的表达式中

```c
int count = (a > b) + (c > d);  // 计算有几个条件为真
```

## 逻辑运算符

逻辑运算符用于将多个条件表达式组合起来，形成更复杂的逻辑判断。C语言提供了三种逻辑运算符：

| 运算符 | 名称 | 示例 | 说明 |
| --- | --- | --- | --- |
| `&&` | 逻辑与（AND） | `a && b` | 当a和b都为真时，结果为真 |
| \|\| | 逻辑或（OR） | `a \|\| b` | 当a或b至少有一个为真时，结果为真 |
| `!` | 逻辑非（NOT） | `!a` | 对a的真假值取反 |

**在C语言中，逻辑运算符将 *非零值视为真* ，零值视为假**
`3 && (-2)` 的值为1
`! ( -1 )` 的值为0

## 逻辑运算的真值表

### 逻辑与 (&&)

| A | B | A && B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

### 逻辑或 (||)

| A | B | A \|\| B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

### 逻辑非 (!)

| A | !A |
|---|---|
| 0 | 1 |
| 1 | 0 |

## 短路求值

### 逻辑与 (&&) 的短路特性

- 如果第一个操作数为假（0），则不会计算第二个操作数
- 因为无论第二个操作数是什么，结果都是假

```c
#include <stdio.h>

int main() {
    int a = 0, b = 5;

    // 短路示例1
    printf("a && (b++): ");
    if (a && (b++)) {
        printf("条件为真\n");
    } else {
        printf("条件为假\n");
    }
```

```
    printf("b的值: %d\n", b);  // b仍然是5，因为b++没有被执行

    // 短路示例2
    int x = 5, y = 0;
    if (x != 0 && y != 0 && (x / y) > 2) {
        printf("不会执行到这里\n");
    } else {
        printf("避免了除以零的错误\n");
    }

    return 0;
}
```

### 逻辑或 (||) 的短路特性

- 如果第一个操作数为真（非0），则不会计算第二个操作数
- 因为无论第二个操作数是什么，结果都是真

```
#include <stdio.h>

int main() {
    int a = 5, b = 10;

    // 短路示例
    printf("a > 0 || (b++): ");
    if (a > 0 || (b++)) {
        printf("条件为真\n");
    } else {
        printf("条件为假\n");
    }
    printf("b的值: %d\n", b);  // b仍然是10，因为b++没有被执行

    // 实际应用：提供默认值
    int input = 0;  // 假设用户没有输入
    int value = input || 100;  // 如果input为0，使用默认值100
    printf("value: %d\n", value);  // 输出: 100

    return 0;
}
```

## 优先级顺序（从高到低）

1. `!`（逻辑非）
2. `&&`（逻辑与）
3. `||`（逻辑或）

# 条件运算符

条件运算符是编程中一个非常常用且简洁的工具，它允许我们根据条件在单行代码中选择两个值中的一个。它本质上是 `if-else` 语句的简化表达式形式。

条件运算符是C语言中**唯一的三目运算符**（需要三个操作数），语法如下：

```
条件表达式 ? 表达式1 : 表达式2
```

**执行逻辑：**

1. 先计算 `条件表达式` 的值
2. 如果条件为**真（非0）**，执行 `表达式1` 并作为结果
3. 如果条件为**假（0）**，执行 `表达式2` 并作为结果

## 基本用法示例

```c
#include <stdio.h>

int main() {
    int a = 10, b = 20;

    // 基础用法：找出较大值
    int max = (a > b) ? a : b;
    printf("较大值：%d\n", max);  // 输出：20

    // 判断奇偶
    int num = 7;
    printf("%d是%s\n", num, (num % 2 == 0) ? "偶数" : "奇数");
    // 输出：7是奇数

    return 0;
}
```

条件运算符运行优先级**非常低**，为方便阅读应多使用小括号。

# 逗号运算符

## 基本概念

**逗号运算符**是C语言中特殊的运算符，它允许在单个表达式中**按顺序执行多个表达式**，并返回**最后一个表达式的值**。

**基本语法：**

```
表达式1, 表达式2, 表达式3, ..., 表达式n
```

1. 执行顺序：从左到右
2. 返回值：**最后一个表达式**
3. 结合性：从左到右

## 基本示例

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

    // 基本用法
    a = (b = 3, c = 5, b + c);  // b=3, c=5, 返回b+c=8
    printf("a = %d\n", a);  // 输出：8

    // 逗号表达式的值
    int x = (10, 20, 30);  // 返回最后一个值30
    printf("x = %d\n", x);  // 输出：30

    return 0;
}
```

## 位运算（不考）

| 运算符 | 名称 | 示例 | 说明 |
| --- | --- | --- | --- |
| `&` | 按位与 | `a & b` | 两位都为1时结果为1 |
| `\|` | 按位或 | `a \| b` | 有一位为1时结果为1 |
| `^` | 按位异或 | `a ^ b` | 两位不同时结果为1 |
| `~` | 按位取反 | `~a` | 0变1，1变0 |
| `<<` | 左移 | `a << n` | 所有位向左移动n位 |
| `>>` | 右移 | `a >> n` | 所有位向右移动n位 |

## 运算优先级

**最高优先级 → 最低优先级**

**第1级：括号、数组、结构体**

```
()      函数调用
[]      数组下标
```

```
.       结构体成员
->      指针结构体成员
```

**第2级：单目运算符（从右向左结合）**

```
++ --      后置自增自减
(type)     强制类型转换
sizeof     求字节大小
++ --      前置自增自减
+ -        正负号（一元）
! ~        逻辑非、按位取反
& *        取地址、解引用
```

**注意**：从右向左结合！！！如`*p++`，结合顺序为`*(p++)`，表达式返回p所指向的元素，再使p指向下一个元素

**第3级：算术运算符**

```
先：  * / %      乘、除、取模
后：  + -        加、减
```

**第4级：移位运算符**

```
<< >>      左移、右移
```

**第5级：关系运算符**

```
先：< <= > >= 比较大小
后：== !=     相等性判断
```

**第6级：位运算符**

```
先：&          按位与
再：^          按位异或
后：|          按位或
```

**第7级：逻辑运算符**

```
先：&&         逻辑与
后：||         逻辑或
```

**第8级：条件运算符（从右向左结合）**

```
? :          三目运算符
```

**第9级：赋值运算符（从右向左结合）**

```
= += -= *= /= %= &= ^= |= <<= >>=
```

**第10级：逗号运算符**

```
,            逗号表达式
```

# 分支结构

## if-else 结构

### 基础结构

分支结构是程序设计中用于根据不同条件执行不同代码块的结构。在C语言中，if-else语句是实现分支控制的主要方式。

```c
// 分支结构的基本思想：
// 程序根据条件表达式的真假值（true/false）选择执行不同的代码路径
// C语言中，0表示假(false)，非0表示真(true)

if (表达式1) {
    语句块;
}
else {
    语句块;
}
```

`if`可以省略`else`配对，即表达式为真则运行语句块，表达式为假则不运行语句块
if-else 结构若语句块中仅有**单条语句**可省略花括号`{}`

## 二分支结构

```c
#include <stdio.h>

int main() {
    int num = 10;
    if (num % 2 == 0) {
        // 条件为真：num是偶数
        printf("%d是偶数\n", num);
    } else {
        // 条件为假：num是奇数
        printf("%d是奇数\n", num);
```

```
    }
    return 0;
}
```

## if-else嵌套

```
#include <stdio.h>
int main() {
    int score;
    scanf("%d", &score)
    // if-else嵌套：在一个if或else代码块中再包含if-else语句
    if (score >= 60) {
        if (score >= 90) {
            printf("优秀\n");
        } else {
            printf("及格\n");
        }
    } else {
        printf("不及格\n");
    }
    return 0;
}
```

## else-if 结构与多分支结构

else-if多分支结构中，从上到下一次检查条件，执行第一个为真对应的代码块。若均不满足，则执行最后一个else代码块（若存在）

结构：

```
if (表达式1) {
    语句块1;
} else if (表达式2) {
    语句块2;
} else if (表达式3) {
    语句块3;
} else {
    语句块4;
}
```

样例:

```
#include <stdio.h>

int main() {
    int score;
    scanf("%d", &score);
```

```
    char grade;

    // 更简洁的else-if结构
    // 注意条件的顺序很重要，应该从大到小或从小到大排列
    if (score >= 90) {
        grade = 'A';
    } else if (score >= 80) {  // 隐含了 score < 90
        grade = 'B';
    } else if (score >= 70) {  // 隐含了 score < 80
        grade = 'C';
    } else if (score >= 60) {  // 隐含了 score < 70
        grade = 'D';
    } else {
        grade = 'E';
    }

    printf("分数：%d，等级：%c\n", score, grade);

    return 0;
}
```

## if-else的配对

在C语言中，else总是与最近的、未配对的if配对

示例：

```
#include <stdio.h>

int main() {
    int a = 10, b = 20;

    if (a > 5)
        if (b > 15)
            printf("a>5且b>15\n");
    else  // 这个else属于哪个if？
        printf("这条语句何时执行？\n");

    // 上面的代码等价于：
    if (a > 5) {
        if (b > 15) {
            printf("a>5且b>15\n");
        } else {
            printf("这条语句何时执行？\n");
        }
    }
```

```
    return 0;
}
```

## switch语句

switch语句是C语言中另一种实现多分支选择的结构
与 `if-else` if阶梯相比， `switch` 更适合单个整形表达式的等值比较

特点：

1. 基于单个表达式的多路分支
2. 只能用于整型变量（参照C语言复习指南 > 整型），不能是浮点型、字符串或其它非整型类型
3. 每个分支用 `case` 标签标记
4. 与 `break` 配合使用

## 基本语法结构

```
switch (表达式) {
    case 常量1:
        语句序列1;
        break;
    case 常量2:
        语句序列2;
        break;
    //...
    default:
        默认语句序列;
 }
```

样例

```
#include <stdio.h>

int main() {
    int day = 3;

    // 基本的switch语句
    // 表达式day的值会与各个case后的常量进行比较
    switch (day) {
        case 1:  // 如果day == 1
            printf("星期一\n");
            break;  // 跳出switch语句
        case 2:  // 如果day == 2
            printf("星期二\n");
            break;
        case 3:  // 如果day == 3
```

```
            printf("星期三\n");
            break;
        case 4:
            printf("星期四\n");
            break;
        case 5:
            printf("星期五\n");
            break;
        default:  // 如果以上都不匹配
            printf("周末\n");
            // default语句通常不需要break，但可以加上
    }

    return 0;
}
```

## case穿透

每个case后面都应该有break，除非有意使用case穿透

在没有 `break` 的switch语句中
程序会从匹配到的case开始执行，一直运行到break或switch语句结束

样例：

```
#include <stdio.h>

int main() {
    int option = 2;

    printf("选项 %d 的结果：\n", option);
    switch (option) {
        case 1:
            printf("执行case 1\n");  // 没有break
        case 2:
            printf("执行case 2\n");  // 没有break
        case 3:
            printf("执行case 3\n");  // 没有break
        case 4:
            printf("执行case 4\n");  // 没有break
        default:
            printf("执行default\n");
    }

    // 输出结果：
    // 选项 2 的结果：
    // 执行case 2
    // 执行case 3
    // 执行case 4
```

```
    // 执行default

    // 注意：因为没有break，程序从case 2开始，一直执行到switch结束

    return 0;
}
```

有意识使用case穿透可能可以简短代码

```
#include <stdio.h>

int main() {
    int month;
    printf("请输入月份：");
    scanf("%d", &month);

    switch (month) {
        case 12: case 1: case 2:
            printf("%d月是冬季\n", month);
            break;
        case 3: case 4: case 5:
            printf("%d月是春季\n", month);
            break;
        case 6: case 7: case 8:
            printf("%d月是夏季\n", month);
            break;
        case 9: case 10: case 11:
            printf("%d月是秋季\n", month);
            break;
        default:
            printf("无效的月份\n");
    }
    return 0;
}
```

### default的位置

注意：无论default放在哪里，它总是在所有case都不匹配时执行
default可以放在任何位置，但会影响执行顺序
通常放在最后，但有时可能有特殊需求（利用case穿透）

# 循环结构

## 基本概念

循环结构是程序设计中重复执行相同或相似任务的基本结构

作用：

1. 避免代码重复，处理重复性计算
2. 遍历数据结构（如数组）

# for循环

## 基本语法

```
for (初始化表达式; 循环条件; 更新表达式) {
    循环体;
}
```

执行顺序：

1. 执行初始化表达式（1次）
2. 判断条件是否成立
3. 如果条件成立，执行循环体；如果不成立，跳出循环
4. 执行更新表达式
5. 回到第二步

初始表达式、循环条件、更新表达式均可省略（**循环条件省略式默认为真**）

可同时对多个变量初始化

```
for (int a = 0, b = 10; a < b; a++, b--) {
    printf("a=%d, b=%d\n", a, b);
}
```

## 求从A年到B年所有的闰年

```
#include <stdio.h>
int main() {
    int startYear, endYear, count = 0;
    scanf("%d", &startYear);
    scanf("%d", &endYear);
    if (startYear > endYear) {
        int temp = startYear;
        startYear = endYear;
        endYear = temp;
    }
    // 使用for循环遍历每一年
    for (int year = startYear; year <= endYear; year++) {
        if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)) {
            printf("%d ", year);
            count++;
            if (count % 5 == 0) {
                printf("\n");
```

```
            }
        }
    }
    printf("\n\n总共%d个闰年\n", count);
    return 0;
}
```

## 判断素数

```
#include <stdio.h>
#include <math.h>  // 为了使用sqrt函数

int main() {
    int num;
    int isPrime = 1;  // 1表示是素数，0表示不是素数
    scanf("%d", &num);
    if (num <= 1) {
        isPrime = 0;  // 1及以下的数不是素数
    } else if (num == 2) {
        isPrime = 1;  // 2是素数
    } else {
        // 优化判断（从2到sqrt(num)）
        // 因为如果一个数不是素数，它必然有一个因子小于等于它的平方根
        int limit = sqrt(num);
        for (int i = 2; i <= limit; i++) {
            if (num % i == 0) {
                isPrime = 0;
                break;
            }
        }
    // 输出结果
    if (isPrime) {
        printf("%d 是素数\n", num);
    } else {
        printf("%d 不是素数\n", num);
    }
    return 0;
}
```

# while循环

## 基本语法

```
while (循环条件) {
    循环体;
}
```

执行顺序：

1. 判断循环条件是否成立
2. 如果条件成立，执行循环体；如果不成立，跳出循环
3. 回到第一步

```
do {
    循环体;
} while (循环条件);
```

执行顺序：

1. 执行循环体
2. 判断循环条件是否成立
3. 如果条件成立，回到第1步

## 求最大公约数和最小公倍数

利用辗转相除法求最大公约数：
要点or原理：`gcd(a, b)=acd(b, a % b)`

```
#include <stdio.h>
int main() {
    int num1, num2;
    printf("请输入两个正整数：");
    scanf("%d %d", &num1, &num2);
    int original1 = num1, original2 = num2;
    if (num1 < num2) {
        int temp = num1;
        num1 = num2;
        num2 = temp;
    }
    int remainder;
    while (num2 != 0) {
        remainder = num1 % num2;  // 求余数
        num1 = num2;              // 除数变成被除数
        num2 = remainder;         // 余数变成除数
    }
    printf("最大公约数是：%d\n\n", num1);
    // 扩展：求最小公倍数
    // 公式：最小公倍数 = 两数乘积 / 最大公约数
    int lcm = (original1 * original2) / num1;
    printf("最小公倍数是：%d\n", lcm);

    return 0;
}
```

## 十进制转二进制

```c
#include <stdio.h>
int main() {
    int decimal;
    scanf("%d", &decimal);

    // 使用do-while循环
    do {
        binary[index] = decimal % 2; // 储存二进制余数
        decimal = decimal / 2;
        index++;
    } while (decimal > 0); // 使用do-while避免对于0的特判

    // 逆序输出
    for (int i = index - 1; i >= 0; i--) {
        printf("%d", binary[i]);
    }

    return 0;
}
```

## 求质因数

```c
#include <stdio.h>

int main() {
    int n, fac = 2;  // 定义变量：n为输入的数，t为临时变量，fac为质因数起始值
    scanf("%d", &n);
    // 循环分解质因数
    while (t >= fac) {
        if (t % fac == 0) {  // 如果t能被fac整除
            t /= fac;         // 将t除以该质因数
            printf("%d\n", fac);  // 输出该质因数
        } else {
            fac++;  // 如果不能整除，尝试下一个数
        }
    }
    return 0;
}
```

## break与continue

`break`：在循环语句中，用于立即终止循环
详见判断素数，遇到可被整除的因数后直接判断为合数跳过整个循环

`continue`：在循环语句中，跳出本次循环
例：输出从a到b所有不被6整除的数

```c
#include <stdio.h>

int main(){
    int a, b;
    scanf("%d%d", &a, &b);

    for(int i = a; i <= b; i++){
        if(i % 6 == 0) // 判断是否为6的倍数
            continue; // 跳过本次循环体部分，进入下一个循环
        printf("%d\n", i);
    }

    return 0;
}
```

## 多重循环

一个循环内部包含另一个循环
常见应用：

1. 打印二维图形
2. 处理二维数据
3. 排列组合问题

```c
#include<stdio.h>

int main(){
    for (int i = 1; i <= 9; i++) {           // 外层循环：行
        for (int j = 1; j <= i; j++) {       // 内层循环：列
            printf("%d×%d=%2d  ", j, i, i * j);
        }
        printf("\n");
    }
    return 0;
}
```

## 换硬币

问题：将一笔零钱换成5分、2分和1分的硬币，要求：每种硬币至少有一枚；问：有几种不同的换法?

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

```c
int main() {
    int money;
    scanf("%d", &money);
    int count = 0;
    // 5分硬币数量：从1到最多可能数量
    for (int fen5 = 1; fen5 <= money / 5; fen5++) {
        // 2分硬币数量：从1到剩余钱数允许的数量
        for (int fen2 = 1; fen2 <= (money - 5 * fen5) / 2; fen2++) {
            // 1分硬币数量：剩余的钱数
            int fen1 = money - 5 * fen5 - 2 * fen2;

            // 检查1分硬币是否至少有1枚
            if (fen1 >= 1) {
                count++;
                printf("方法%2d：5分×%d + 2分×%d + 1分×%d = %d分\n",
                       count, fen5, fen2, fen1, fen5*5 + fen2*2 + fen1);
            }
        }
    }

    printf("总共%d种方法\n\n", count);
    return 0;
}
```

# 数组

## 基本概念

为便于批量存储多个变量，避免重复声明，可使用数组

特点：

1. 元素类型相同
2. 在内存中连续存储（指针、地址相关
3. 长度固定（定义后不可改变）
4. 可通过下标访问元素，**下标从0开始**‘

## 一维数组

### 数组的声明

```c
int num[10]; //声明一个包含十个整数的数组
float pri[20]; // 声明一个包含20个浮点数的数组
char name[20]; // 声明一个包含20个字符的数组
```

数组长度必须是**整数常量或常量表达式**

## 数组的初始化

概念：在声明数组的同时给数组元素赋初值

```
int numbers1[5] = {10, 20, 30, 40, 50}; // 给出每个元素的值
// 只给出部分元素的值，其余自动为0
int numbers2[5] = {10, 20, 30};  // numbers2[0]=10, [1]=20, [2]=30, [3]=0, [4]=0
float prices[5] = {12.5, 15.8};  // prices[0]=12.5, [1]=15.8, 其余为0.0
// 未指定大小初始化（自动确定数组大小）
int numbers3[] = {1, 2, 3, 4, 5};  // 数组大小自动确定为5
// 方法4：指定位置的初始化（C99标准）
int numbers4[10] = {[2] = 30, [5] = 50, [9] = 90};
// 结果是：numbers4[2]=30, numbers4[5]=50, numbers4[9]=90, 其余为0
```

如果不初始化数组，数组元素的初值是不确定的

## 数组的大小和sizeof运算符

`sizeof` 是C语言中的一个**运算符**（不是函数），用于计算数据类型或表达式在内存中所占的字节数。
`sizeof` 在编译阶段就确定结果，不会在运行时计算
数组的大小 = 变量类型字节数 * 元素个数

可用于计算数据类型的大小，如

```
printf("int 大小: %zu 字节\n", sizeof(int));        // 通常为4字节
printf("char 大小: %zu 字节\n", sizeof(char));      // 总是1字节
printf("double 大小: %zu 字节\n", sizeof(double)); // 通常为8字节
struct Student {
    char name[20];
    int age;
    float score;
};
printf("结构体大小: %zu\n", sizeof(struct Student)); // 20*1+4+4=28
```

也可用于求变量/数组的大小，如

```
int num[20] = {20,10,15,10};
float f = 3.14;
printf("num 大小: %zu\n", sizeof(num)); // 通常为80字节
printf("f 大小: %zu\n", sizeof(f)); // 通常为4字节
```

可利用sizeof求数组的元素个数

## 数组元素的访问

方式：`数组名[下标]`

注意：

1. 下标从0开始
2. 下标从0到数组长度-1
3. 可读取或修改元素值

### 数组越界

访问数组时使用了超出有效范围的下标，可能读取到其他变量的值或修改其他变量的值，导致程序崩溃

```c
#include <stdio.h>

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    printf("arr[-1] = %d\n", arr[-1]);  // 访问数组前面的内存
    printf("arr[5] = %d\n", arr[5]);    // 访问数组后面的内存
    printf("arr[100] = %d\n", arr[100]);  // 访问很远的地址
    return 0;
}
```

### 数组名

1. 数组名是数组首元素的地址（常量指针）
2. 数组名不是普通的变量，而是一个**地址常量**（不能被重新赋值）
3. 在大多数情况下（如数组名作为参数传递时），数组名会退化为**指向数组首元素的指针**，*因此在函数内部无法通过sizeof获取数组的实际大小*

## 二维数组

### 基本概念

数组的数组，可看作一个表格，分为行、列
声明：`数组类型 数组名[行数][列数]`
内存中的存储：按行优先顺序存储，先存储第一行，再存储第二行，依此类推
可视为连续的一维数组
访问元素方式：`数组名[行下标][列下标]`

### 二维数组的初始化

方法1：**给出所有元素的初值**

```c
int matrix1[3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

方法2：**按行初始化**（用花括号括住每一行）

```
int matrix2[3][4] = {
    {1, 2, 3, 4},      // 第1行
    {5, 6, 7, 8},      // 第2行
    {9, 10, 11, 12}    // 第3行
};
```

方法:3：部分初始化，未初始化的元素自动为**0**

```
int matrix3[3][4] = {
    {1, 2},         // 第1行：前2个元素为1和2，后2个为0
    {5, 6, 7},      // 第2行：前3个元素，第4个为0
    {9}             // 第3行：第1个元素为9，其余为0
    };
int matrix3_2[3][4] = {1, 2, 3, 4, 5};
// [0][0]为1，[0][1]为2，[0][2]为3，[0][3]为4
// [1][0]为5，其余元素均为0
```

方法4：**省略行数初始化（编译器自动计算行数）**
注意：
只可以省略行数，但不能省略列数

```
int matrix4[][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12},
    {13, 14, 15, 16}  // 编译器会自动计算为4行
};
```

## 二维数组的元素访问与遍历

正确方式：`数组名[行下标][列下标]`
规则：

1. 行下标范围：0 到 (行数-1)
2. 列下标范围：0 到 (列数-1)

样例：

```
#include <stdio.h>

int main() {
    int matrix[3][4] = {
        {1, 2, 3, 4},     // 第0行
        {5, 6, 7, 8},     // 第1行
```

```
        {9, 10, 11, 12}   // 第2行
    };
    // 遍历所有元素
    printf("\n遍历整个二维数组：\n");
    for (int i = 0; i < 3; i++) {          // i: 行下标 (0, 1, 2)
        for (int j = 0; j < 4; j++) {      // j: 列下标 (0, 1, 2, 3)
            printf("matrix[%d][%d] = %2d  ", i, j, matrix[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

**越界访问**

`int matrix[3][4];` 表示一个3行4列的数组
若访问 `matrix[0][4]` ，试图访问第0行的第5个元素，实际上就是第1行的 `matrix[1][0]` （二维数组可视为连续储存的一维数组）

## 打印杨辉三角

```
#include <stdio.h>

int main() {
    int rows;
    scanf("%d", &rows);
    int triangle[20][20];  // 使用二维数组存储
    // 初始化第一列和对角线为1
    for (int i = 0; i < rows; i++) {
        triangle[i][0] = 1;       // 第一列都是1
        triangle[i][i] = 1;       // 对角线都是1
    }
    // 计算其他元素：triangle[i][j] = triangle[i-1][j-1] + triangle[i-1][j]
    for (int i = 2; i < rows; i++) {
        for (int j = 1; j < i; j++) {
            triangle[i][j] = triangle[i-1][j-1] + triangle[i-1][j];
        }
    }
    // 打印三角形（居中对齐）
    for (int i = 0; i < rows; i++) {
        // 打印前导空格，使三角形居中
        for (int space = 0; space < rows - i - 1; space++) {
            printf("   ");
        }

        // 打印当前行的数字
        for (int j = 0; j <= i; j++) {
            printf("%6d", triangle[i][j]);
        }
```

```
            printf("\n");
        }
        return 0;
    }
```

## 螺旋矩阵

输入：N
输出N*N的螺旋方阵
示例：
输入：4
输出：

```
 1  2  3  4
12 13 14  5
11 16 15  6
10  9  8  7
```

思路：
**模拟**旋转填充的过程
按照右→下→左→上的顺序填充
每次无法填充，顺时针旋转90度，以下一个方向继续填充

```
#incldue <stdio.h>

int main(){
    int matrix[10][10] = {};// 初始化矩阵为0（表示未填充）
    // 方向数组：右(0,1)，下(1,0)，左(0,-1)，上(-1,0)
    int dirs[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
    int dir = 0;  // 当前方向

    int row = 0, col = 0;  // 当前位置
    int num = 1;           // 要填充的数字
    int N;
    scanf("%d", &N);
    for (int i = 0; i < N * N; i++) {
        matrix[row][col] = num++;

        // 计算下一个位置
        int nextRow = row + dirs[dir][0];
        int nextCol = col + dirs[dir][1];

        // 如果下一个位置越界或已填充，改变方向
        if (nextRow < 0 || nextRow >= N || nextCol < 0 || nextCol >= N ||
            matrix[nextRow][nextCol] != 0) {
            dir = (dir + 1) % 4;  // 改变方向
            nextRow = row + dirs[dir][0];
```

```
            nextCol = col + dirs[dir][1];
        }

        row = nextRow;
        col = nextCol;
    }
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%3d ", matrix[i][j]);
        }
        printf("\n");
    }
}
```

# 函数

## 函数的作用

1. 函数允许我们将常用的代码块封装起来，在需要时重复调用，避免重复编写相同的代码。
2. 将复杂问题分解为小问题
3. 提供清晰的接口，降低程序复杂度

## 函数的声明与定义

### 函数的基本结构

```
返回类型 函数名(参数列表) {
    // 函数体
    return 返回值; // 如果返回类型不是void
}
```

### 函数的声明

```

// 声明（告诉编译器函数的存在）
int add(int a, int b);
double calculateArea(double radius);
void printMessage(void);

// 实际定义
int add(int a, int b) {
    return a + b;
}
// 可先声明，后定义，甚至在main函数之后
```

### 返回值类型

- `void`: 无返回值
- `int`, `float`, `double`, `char`等: 返回对应类型
- 自定义类型: 结构体、指针等

# 函数的调用

## 基本调用方式

`函数名(参数1, 参数2, … )`

## 函数的调用过程

1. 保存当前执行状态
2. 传递参数
3. 跳转到函数代码
4. 执行函数体
5. 返回结果
6. 恢复之前状态继续执行

# 形参与实参

- **形参(形式参数)**: 函数定义时声明的参数
- **实参(实际参数)**: 函数调用时传递的实际值

```c
#include <stdio.h>

void swap(int a, int b) {// a和b是形参
    int temp = a;
    a = b;
    b = temp;
    printf("函数内交换后: a=%d, b=%d\n", a, b);
}

int main() {
    int x = 5, y = 10;
    printf("交换前: x=%d, y=%d\n", x, y); // 5 10
    swap(x, y); // x和y是实参 输出: 10 5
    printf("交换后: x=%d, y=%d\n", x, y); // 5 10
    // 注意: x和y的值没有改变! 因为C语言默认是值传递
    return 0;
}
```

C语言中函数参数只有值传递，需要通过指针实现引用传递（见指针作为函数的参数）

# 函数的嵌套

函数定义时，函数体中**可包含其他函数**
如：

```
double distance(double x1, double y1, double x2, double y2) {
    double dx = x2 - x1;
    double dy = y2 - y1;
    return sqrt(dx*dx + dy*dy);  // 嵌套调用sqrt函数
}
```

# 递归

## 基本概念

- 函数调用自身
- 必须包含递归终止条件
- 每次递归调用问题规模应减小

## 阶乘示例

```
#include <stdio.h>

// 递归计算阶乘
long long factorial(int n) {
    // 终止条件
    if (n <= 1) {
        return 1;
    }
    // 递归调用
    return n * factorial(n - 1);
}

int main() {
    int num = 5;
    printf("%d! = %lld\n", num, factorial(num));
    return 0;
}
```

## 递归法求最大公约数

与求最大公约数和最小公倍数原理相同，利用辗转相除法，可简洁的写出返回两个正整数最大公约数的函数

```
int gcd(int a, int b) {
    if (b == 0) {
        return a;
    }
```

```
    return gcd(b, a % b);
}
```

## 汉诺塔问题

### 问题描述

有三根柱子A、B、C，A柱上有n个大小不同的圆盘，从小到大叠放。要求将A柱的圆盘全部移到C柱，移动规则：

1. 每次只能移动一个圆盘
2. 大圆盘不能放在小圆盘上面
3. 只能借助B柱进行移动

### 递归解法思路

1. 将n-1个盘子从A移到B（借助C）
2. 将第n个盘子从A移到C
3. 将n-1个盘子从B移到C（借助A）

### 代码

```c
#include <stdio.h>

// 汉诺塔递归函数
void hanoi(int n, char from, char to, char aux) {
    // 终止条件：只有一个盘子
    if (n == 1) {
        printf("移动盘子 1 从 %c 到 %c\n", from, to);
        return;
    }

    // 递归步骤
    hanoi(n - 1, from, aux, to);  // 将n-1个盘子从from移到aux
    printf("移动盘子 %d 从 %c 到 %c\n", n, from, to);  // 移动第n个盘子
    hanoi(n - 1, aux, to, from);  // 将n-1个盘子从aux移到to
}

int main() {
    int n;
    scanf("%d", &n);
    printf("汉诺塔问题（%d个盘子）的解法：\n", n);
    hanoi(n, 'A', 'C', 'B');

    return 0;
}
```

### 输出结果

```
汉诺塔问题（3个盘子）的解法：
移动盘子 1 从 A 到 C
移动盘子 2 从 A 到 B
移动盘子 1 从 C 到 B
移动盘子 3 从 A 到 C
移动盘子 1 从 B 到 A
移动盘子 2 从 B 到 C
移动盘子 1 从 A 到 C
```

## 快速排序

### 算法思想

策略：

1. **选择基准**：从数组中选择一个元素作为基准
2. **分区**：将数组分为两部分，小于基准的在左，大于基准的在右
3. **递归排序**：对左右两部分递归进行快速排序

### 对数组的排序过程示例

### 第一次分区（选择第一个元素5作为基准）：

```
原始：[5, 3, 8, 1, 2]
分区后：[3, 1, 2, 5, 8]
左部分：[3, 1, 2]  右部分：[8]
```

### 递归排序左部分 `[3, 1, 2]`：

```
选择3作为基准
分区后：[1, 2, 3]
左部分：[1, 2]  右部分：[]
```

### 递归排序 `[1, 2]`：

```
选择1作为基准
分区后：[1, 2]
左部分：[]  右部分：[2]
```

### 最终结果：

`[1, 2, 3, 5, 8]`

### 代码

```c
#include <stdio.h>
// 交换两个元素
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
// 分区函数
int partition(int arr[], int low, int high) {
    int pivot = arr[low];  // 选择第一个元素作为基准
    int i = low + 1;
    int j = high;

    while (1) {
        // 从左向右找第一个大于基准的元素
        while (i <= j && arr[i] <= pivot) {
            i++;
        }
        // 从右向左找第一个小于基准的元素
        while (i <= j && arr[j] > pivot) {
            j--;
        }

        if (i >= j) {
            break;
        }

        // 交换这两个元素
        swap(&arr[i], &arr[j]);
    }

    // 将基准放到正确位置
    swap(&arr[low], &arr[j]);

    return j;  // 返回基准的最终位置
}

// 快速排序递归函数
void quickSortRecursive(int arr[], int low, int high) {
    if (low < high) {
        // 分区，获取基准位置
        int pi = partition(arr, low, high);

        // 递归排序左半部分
        quickSortRecursive(arr, low, pi - 1);

        // 递归排序右半部分
        quickSortRecursive(arr, pi + 1, high);
    }
```

```c
}

// 快速排序包装函数
void quickSort(int arr[], int n) {
    quickSortRecursive(arr, 0, n - 1);
}

// 打印数组
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int n, arr[200];
    scanf("%d", &n);
    for(int i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    quickSort(arr, n);

    printArray(arr, n);

    return 0;
}
```

# 宏定义

## 宏基本定义

### 基本语法

```c
#define 宏名 替换文本
```

**示例:**

```c
#define PI 3.14159
#define MAX_SIZE 100
#define NEWLINE '\n'

float circle_area = PI * radius * radius;
int array[MAX_SIZE];
```

### 特点

- 编译前进行文本替换
- 宏名通常用大写字母（约定）
- 行末**不要**加分号（除非需要）

# 带参数的宏定义

## 基本语法

```
#define 宏名(参数列表) 替换文本
```

**示例:**

```
// 带参数的宏
#define SQUARE(x) ((x) * (x))
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define PRINT_INT(n) printf("%d", n)

// 使用示例
int result = SQUARE(5);       // 展开为: ((5) * (5))
int max_val = MAX(3, 7);      // 展开为: ((3) > (7) ? (3) : (7))
PRINT_INT(42);                // 展开为: printf("%d", 42)
```

## 注意事项

宏定义只是在预处理阶段进行**文本替换**，为保证宏定义的正确使用，在必要处加括号保证运算顺序

```
#define SQUARE(x) x * x
int wrong = SQUARE(3 + 2);    // 展开为: 3 + 2 * 3 + 2 = 11（不是25）

#define SQUARE(x) ((x) * (x))
int correct = SQUARE(3 + 2); // 展开为: ((3 + 2) * (3 + 2)) = 25
```

# 宏定义的实质

## 本质：文本替换

- **不是函数调用**，没有参数传递和返回值概念
- **不是变量**，不占用内存空间
- **在预处理阶段**进行简单文本替换

## 预处理前后的对比

源代码:

```c
#define PI 3.14
#define CIRCLE_AREA(r) (PI * (r) * (r))
int main() {
    float area = CIRCLE_AREA(5.0);
    return 0;
}
```

预处理后：

```c
int main() {
    float area = (3.14 * (5.0) * (5.0));  // 直接文本替换
    return 0;
}
```

# 指针

## 基本概念

### 内存地址

- 计算机内存像一排连续的**房间**，每个房间有**唯一编号**（地址）
- 每个房间大小固定（以字节为单位）
- 内存地址即为变量储存在内存的特定位置

### 指针

指针是一个变量，它的值是**另一个变量的内存地址**
指针本身也是变量，跟其他变量一样储存在内存中

- 普通变量：**盒子里的内容**
- 指针变量：**写着另一个盒子地址的纸条**

## 指针变量的声明

**声明语法：** `数据类型 *指针变量名;`

示例：

```c
int *p;      // p是一个指向整型变量的指针
char *c;     // c是一个指向字符变量的指针
float *f;    // f是一个指向浮点型变量的指针
double *d;   // d是一个指向双精度浮点型变量的指针
```

## 指针的基本运算

### 取地址运算符 `&`

获取变量的内存地址

```
int a = 10;
int *p = &a;  // p现在保存了a的地址
```

## 解引用运算符 *

通过指针访问/修改该地址处的值

```
int a = 10;
int *p = &a;
printf("%d", *p);  // 输出10，*p获取a的值
*p = 20;           // 通过p修改a的值为20
printf("%d", a);   // 现在a的值是20
```

## 指针的自增和自减

移向下一个元素/上一个元素

```
int arr[5] = {1, 2, 3, 4, 5};
int *p = arr;      // p指向数组第一个元素

printf("%d\n", *p);  // 输出1
p++;                 // p现在指向第二个元素
printf("%d\n", *p);  // 输出2
p--;                 // p又指回第一个元素
```

## 指针的加法和减法

指：指针与整数的加减
指向后n个单位/前n个单位

```
int arr[5] = {10, 20, 30, 40, 50};
int *p = arr;

p = p + 2;  // p现在指向第三个元素(30)
p = p - 1;  // p现在指向第二个元素(20)
```

## 指针间的减法

计算元素之间的间隔

```
int arr[5] = {10, 20, 30, 40, 50};
int *p1 = &arr[0];  // 指向第一个元素
int *p2 = &arr[3];  // 指向第四个元素
```

```c
int diff = p2 - p1;  // diff = 3，表示p2在p1后面3个元素的位置
```

## 指针的初始化

与变量的初始化无差异

```c
int a = 10;
int *p = NULL;  // p不指向任何有效的内存地址
int *q = &a;
```

## 空指针NULL

表示指针不指向任何有效的内存地址

## 指针、数组、内存空间与地址的关系

### 1. 数组名可视为指针

见数组名

```c
int arr[5] = {1, 2, 3, 4, 5};
int *p = arr;  // arr本身就是第一个元素的地址

printf("%p\n", arr);      // 输出数组第一个元素的地址
printf("%p\n", &arr[0]); // 输出相同地址
printf("%d\n", arr[0]);  // 输出1
printf("%d\n", *arr);    // 同样输出1
```

### 2. 可通过指针访问数组中的元素

```c
int arr[5] = {1, 2, 3, 4, 5};
int *p = arr;

// 以下等价访问方式
arr[2] = 100;     // 传统数组访问
*(arr + 2) = 100; // 指针算术访问
*(p + 2) = 100;   // 通过指针变量访问
p[2] = 100;       // 指针可以像数组一样使用
```

### 3. 内存关系示意表

对于 `int arr[4] = {1, 2, 3, 4};`
有：

| 地址 | 内存内容 | 变量名 |
| --- | --- | --- |
| `0x1000` `&arr[0]` `arr` | 1 | `arr[0]` `*arr` |
| `0x1004` `&arr[1]` `arr+1` | 2 | `arr[1]` `*(arr+1)` |
| `0x1008` `&arr[2]` `arr+2` | 3 | `arr[2]` `*(arr+2)` |
| `0x100C` `&arr[3]` `arr+3` | 4 | `arr[3]` `*(arr+3)` |

## 指针作为函数的参数

普通参数传递属于**值传递**，将变量的值复制为形式参数，**无法改变原始变量的值**

指针参数属于**地址传递**

示例：

```
// 值传递：无法修改原始变量
void changeValue(int x) {
    x = 100;  // 只修改了副本
}

// 地址传递：可以修改原始变量
void changeValueByPointer(int *x) {
    *x = 100;  // 修改原始变量的值
}

int main() {
    int a = 10;
    changeValue(a);           // a还是10
    changeValueByPointer(&a); // a变成100
    return 0;
}
```

## swap函数

交换两个变量的值

```
void swap(int *a, int *b) {
    int temp = *a;  // 获取a指向的值
    *a = *b;        // 将b的值赋给a指向的位置
```

```
    *b = temp;      // 将temp的值赋给b指向的位置
}
```

# 冒泡排序

*以升序为例*
基本思路：

- 从数组的第一个元素开始，**依次比较相邻的两个元素**
- 如果前一个元素大于后一个元素，就**交换它们的位置**
- 这样，每一轮比较都会将**当前未排序部分的最大元素"冒泡"到最后正确的位置**
  时间复杂度为O(n²)

## 举例说明

假设要排序的数组为：`[5, 3, 8, 1, 2]`

**第一轮排序**（将最大的数8放到最后）：

```
比较[0]和[1]: 5 > 3? 是 → 交换 → [3, 5, 8, 1, 2]
比较[1]和[2]: 5 > 8? 否 → 不交换 → [3, 5, 8, 1, 2]
比较[2]和[3]: 8 > 1? 是 → 交换 → [3, 5, 1, 8, 2]
比较[3]和[4]: 8 > 2? 是 → 交换 → [3, 5, 1, 2, 8]
```

结果：8已就位（最后位置）

**第二轮排序**（将第二大的数5放到倒数第二）：

```
比较[0]和[1]: 3 > 5? 否 → 不交换 → [3, 5, 1, 2, 8]
比较[1]和[2]: 5 > 1? 是 → 交换 → [3, 1, 5, 2, 8]
比较[2]和[3]: 5 > 2? 是 → 交换 → [3, 1, 2, 5, 8]
```

结果：5已就位（倒数第二位置）

**第三轮排序**：

```
比较[0]和[1]: 3 > 1? 是 → 交换 → [1, 3, 2, 5, 8]
比较[1]和[2]: 3 > 2? 是 → 交换 → [1, 2, 3, 5, 8]
```

结果：3已就位

**第四轮排序**：

```
比较[0]和[1]: 1 > 2? 否 → 不交换 → [1, 2, 3, 5, 8]
```

结果：全部排序完成

## 详细实现代码

```c
#include <stdio.h>

void swap(int *x, int *y){
    int temp = *x;
    *x = *y;
    *y = temp;
}
void bubbleSort(int *arr, int n) {
    for (int i = 0; i < n-1; i++)
        for (int j = 0; j < n-i-1; j++)
            // 使用指针比较和交换
            if (*(arr + j) > *(arr + j + 1))
                swap(arr + j, arr + j + 1);// 交换两个元素
}

int main() {
    int arr[100], n;
    scanf("%d", n);
    for(int i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    bubbleSort(arr, n);

    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    return 0;
}
```

# 指针作为函数的返回值

当指针作为函数的返回值时，**不可返回局部变量的地址**
可以返回：
静态变量、全局变量、传入参数及其运算后的地址

```c
int* badFunction() {
    int localVar = 42;  // 局部变量，函数结束时销毁
    return &localVar;   // 错误！返回已释放内存的地址
}

int* goodFunction1() {
    static int staticVar = 42;  // 静态变量，程序运行期间一直存在
    return &staticVar;          // 安全
}
```

```c
int* findMax(int *a, int *b) {
    if (*a > *b) {
        return a;  // 安全：返回传入参数的地址
    } else {
        return b;  // 安全：返回传入参数的地址
    }
}
```

## 指针数组

数组的元素是指针
声明：`数据类型 *数组名[大小]`

## 举例说明

代码：

```c
#include <stdio.h>

int main() {
    // 声明一个指针数组，包含3个int指针
    int* arrp[3];
    // 声明并初始化一个包含6个整数的数组
    int arr[] = {1, 2, 3, 4, 5, 6};

    // 将指针数组的三个元素分别指向arr数组的不同元素
    arrp[0] = &arr[0];  // 指向第一个元素
    arrp[1] = &arr[2];  // 指向第三个元素
    arrp[2] = &arr[4];  // 指向第五个元素

    // 打印arr数组的所有元素及其地址
    for(int i=0; i<6; i++)
        printf("arr[%d]:%d &arr[%d]:%p\n", i, arr[i], i, &arr[i]);
    printf("\n");

    // 打印指针数组arrp及其偏移的地址
    printf("arrp:%p\n", arrp);       // arrp的起始地址
    printf("arrp+1:%p\n", arrp+1);  // 第二个指针的地址
    printf("arrp+2:%p\n", arrp+2);  // 第三个指针的地址
    printf("\n");

    // 打印指针数组arrp每个元素的地址
    for(int i=0; i < 3; i++)
        printf("&arrp[%d]:%p\n", i, &arrp[i]);
    printf("\n");

    // 打印指针数组arrp每个元素的值（即指向的地址）
    for(int i=0; i < 3; i++)
        printf("arrp[%d]:%p\n", i, arrp[i]);
```

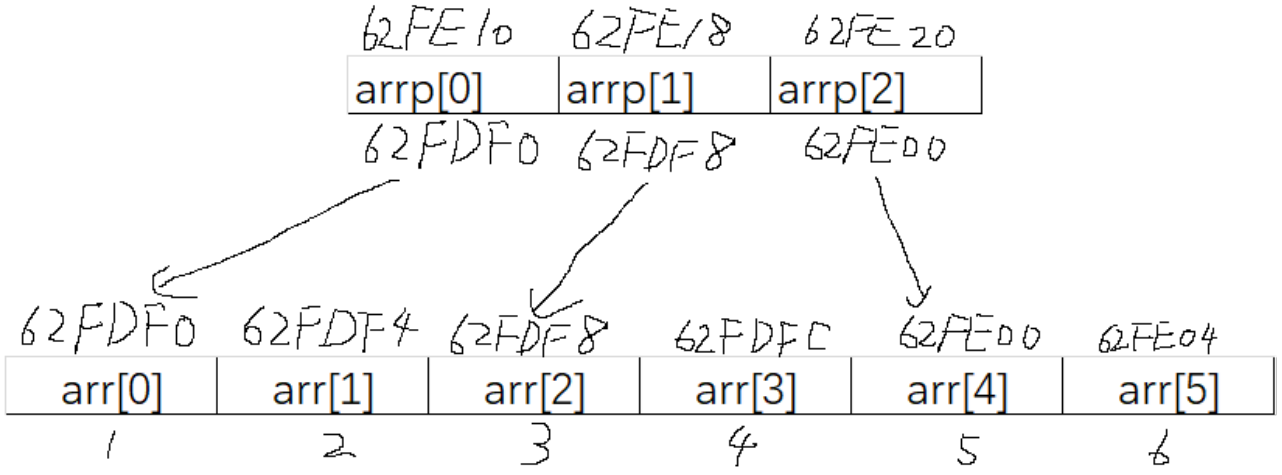
```
    printf("\n");

    // 打印指针数组arrp每个元素所指向的值
    for(int i=0; i < 3; i++)
        printf("*arrp[%d]:%d\n", i, *arrp[i]);
    printf("\n");

    // 打印第一个指针所指向的数组元素（从该位置开始的连续元素）
    for(int i=0; i < 6; i++)
        printf("(*arrp)[%d]:%d\n", i, (*arrp)[i]);
    printf("\n");

    // 打印第二个指针所指向的数组元素（从arr[2]开始的连续4个元素）
    for(int i=0; i < 4; i++)
        printf("(*(arrp+1))[%d]:%d\n", i, (*(arrp+1))[i]);
    printf("\n");

    // 打印第三个指针所指向的数组元素（从arr[4]开始的连续2个元素）
    for(int i=0; i < 2; i++)
        printf("(*(arrp+2))[%d]:%d\n", i, (*(arrp+2))[i]);
    printf("\n");

    return 0;
}
```

输出结果：

```
arr[0]:1 &arr[0]:000000000062FDF0
arr[1]:2 &arr[1]:000000000062FDF4
arr[2]:3 &arr[2]:000000000062FDF8
arr[3]:4 &arr[3]:000000000062FDFC
arr[4]:5 &arr[4]:000000000062FE00
arr[5]:6 &arr[5]:000000000062FE04

arrp:000000000062FE10
arrp+1:000000000062FE18
arrp+2:000000000062FE20

&arrp[0]:000000000062FE10
&arrp[1]:000000000062FE18
&arrp[2]:000000000062FE20

arrp[0]:000000000062FDF0
arrp[1]:000000000062FDF8
arrp[2]:000000000062FE00

*arrp[0]:1
*arrp[1]:3
*arrp[2]:5
```

```
(*arrp)[0]:1
(*arrp)[1]:2
(*arrp)[2]:3
(*arrp)[3]:4
(*arrp)[4]:5
(*arrp)[5]:6

(*(arrp+1))[0]:3
(*(arrp+1))[1]:4
(*(arrp+1))[2]:5
(*(arrp+1))[3]:6

(*(arrp+2))[0]:5
(*(arrp+2))[1]:6
```

结构示意图：




## 多级指针

一级指针：指向变量的指针
二级指针：指向一级指针的指针
以此类推...

## 举例说明

```c
#include <stdio.h>

int main() {
    int* arrp[3];
    int arr[] = {1, 2, 3, 4, 5, 6};

    arrp[0] = &arr[0];
    arrp[1] = &arr[2];
    arrp[2] = &arr[4];
```

```c
    // 声明一个二级指针pp，指向指针数组arrp的第一个元素
    int **pp = arrp;  // 等价于 int **pp = &arrp[0];

    printf("二级指针pp的地址: %p\n", &pp);
    printf("二级指针pp存储的值: %p (即arrp[0]的地址)\n", pp);
    printf("\n");

    printf("通过二级指针访问一级指针:\n");
    printf("*pp = %p (这是arrp[0]的值，即&arr[0])\n", *pp);
    printf("*(pp + 1) = %p (这是arrp[1]的值，即&arr[2])\n", *(pp + 1));
    printf("*(pp + 2) = %p (这是arrp[2]的值，即&arr[4])\n", *(pp + 2));
    printf("\n");

    printf("通过二级指针访问原始数据:\n");
    printf("**pp = %d (即arr[0]的值)\n", **pp);
    printf("**(pp + 1) = %d (即arr[2]的值)\n", **(pp + 1));
    printf("**(pp + 2) = %d (即arr[4]的值)\n", **(pp + 2));
    printf("\n");

    printf("使用二级指针遍历指针数组:\n");
    for (int i = 0; i < 3; i++) {
        printf("pp + %d = %p, *(pp + %d) = %p, **(pp + %d) = %d\n",
               i, pp + i, i, *(pp + i), i, **(pp + i));
    }
    printf("\n");

    printf("二级指针的自增操作:\n");
    int **temp = pp;  // 保存原始值
    printf("初始: **temp = %d\n", **temp); // 输出1
    temp++;  // 移动到下一个指针数组元素
    printf("自增一次后: **temp = %d\n", **temp); // 输出3
    temp++;  // 再移动到下一个指针数组元素
    printf("自增两次后: **temp = %d\n", **temp); // 输出5
    printf("\n");

    return 0;
}
```

声明：

```c
数据类型 **指针变量名;  // 二级指针
数据类型 ***指针变量名; // 三级指针
```

## 指向函数的指针 *（不考）*

**函数指针**：指向函数的指针变量，存储的是函数的入口地址，而不是数据。

声明：

`int (*funcPtr)(int, int); // 指向接受两个int参数并返回int的函数`

**样例**

```c
#include <stdio.h>

// 一个普通函数
int add(int a, int b) {
    return a + b;
}

int main() {
    // 声明函数指针
    int (*funcPtr)(int, int);

    // 赋值：函数名本身就是地址
    funcPtr = add;  // 或者 funcPtr = &add;

    // 通过指针调用函数
    int result = funcPtr(3, 4);  // 等价于 add(3, 4)
    printf("3 + 4 = %d\n", result);  // 输出 7

    return 0;
}
```

# 字符串

## 基本概念

在C语言中，字符串并不是一种独立的数据类型，而是以字符数组的形式存在的。字符串的本质是：

- 一串连续的字符序列
- 以空字符'\0'作为结束标志
- 存储在连续的内存空间中

存储方式与其他数组相同

## 字符串的声明

```c
char str1[20]; / 声明一个可以存储19个字符+1个'\0'的字符串

// 不指定大小（由初始化决定）
char str2[] = "Hello"; // 编译器自动计算大小为6（5个字符+1个'\0'）
```

## 字符串的初始化

方法1：与其他数字相同，逐个初始化

```
char str3[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

方法2：直接用 = 赋值字符串常量

```
char str1[] = "Hello World"; // 自动在末尾添加'\0'并计算长度
char str2[20] = "Hello"; // 剩余部分自动填充'\0'
```

# 字符串的基本概念

**字符串结束的标志是** `'\0'`，`\0'` 是ASCII码为0的字符，也称为空字符或字符串结束符。

**字符串的长度不等于数组的大小**
字符串长度：从第一个字符到 `'\0'` 前的总字符数
字符数组大小：元素的总个数

# 字符串的输入

## scanf

读取规则：遇到空格停止

```
scanf("%s", str1); // 若输入"Hello World"，只会读取"Hello"
// str1本身即为地址，不需要取地址符&
```

# gets

语法：`gets(字符串名)`
读取规则：遇到换行停止（`'\n'` 不存入字符串中）

```
gets(str2);
```

# 逐个字符读取

自行设置读取规则，如遇到字符'#'则停止

```
char str[100];
char *p = str;
while((*p=getchar())!='#')
    p++;
p = ' ';
```

# 字符串的输出

### printf

例：

```
printf("%s", str);
```

## puts

语法：`puts(字符串名)`，会自动换行
例：

```
puts(str);
```

# string.h头文件与字符串常用函数

## 字符串长度函数 strlen

返回整数：计算字符串长度(不包括'\0')

```
char str[] = "Hello World";
int len = strlen(str);
printf("字符串长度: %d\n", len);  // 11
```

## 字符串复制函数 strcpy strncpy

```
char src[] = "Hello World";
char dest1[20];
char dest2[20];
// strcpy - 复制整个字符串
strcpy(dest1, src);
printf("strcpy复制: %s\n", dest1);

// strncpy - 复制指定长度
strncpy(dest2, src, 5);
dest2[5] = '\0';  // 需要手动添加结束符
printf("strncpy复制5个字符: %s\n", dest2);
```

## 字符串连接函数 strcat strncat

```
char str1[50] = "Hello";
char str2[] = " World";
char str3[50] = "Hello";

// strcat - 连接字符串
strcat(str1, str2);
```

```
printf("strcat连接: %s\n", str1);  // "Hello World"

// strncat - 连接指定长度
strncat(str3, str2, 3);  // 连接" Wo"
printf("strncat连接3个字符: %s\n", str3);  // "Hello Wo"
```

## 字符串比较函数 strcmp strncmp

strcmp：逐个比较字符串中的对应字符，知道字符不同或比较到串尾
strncmp：比较前n个字符

返回值:

- <0 (str1 < str2)
- 0 (相等)
- > 0 (str1 > str2)
- 不等时返回值为**不相等字符的差值**

# 字符串查找函数 strchr strrchr strstr

`strchr`：查找字符第一次出现的位置
`strrchr`：查找字符最后一次出现的位置

返回值：

- 找到：字符对应的**地址**
- 找不到：NULL

`strstr`：查找字符串第一次出现的位置

返回值：

- 找到：子串首字符对应的**地址**
- 找不到：NULL

# 样题

## 统计单词的长度

本题目要求编写程序，输入一行字符，统计每个单词的长度。所谓“单词”是指连续不含空格的字符串，各单词之间用空格分隔，空格数可以是多个。

```
#include <stdio.h>

int main() {
    int count = 0;       // 当前单词的字符计数器
```

```
    int flag = 1;        // 标志位：1表示尚未遇到单词，0表示已处理过单词
    char c;              // 当前读取的字符

    // 逐字符读取输入，直到遇到换行符
    while ((c = getchar()) != '\n') {
        if (c != ' ') {                  // 当前字符不是空格
            count++;                     // 单词长度加1
        } else if (count != 0) {         // 遇到空格且当前有未统计完的单词
            flag = 0;                    // 标记已遇到单词
            printf("%d ", count);        // 输出单词长度（后跟空格）
            count = 0;                   // 重置计数器准备下一个单词
        }
        // 注：连续空格时，第一个空格触发输出，后续空格被忽略
    }

    // 处理行末最后一个单词（如果存在）
    // 条件：1.最后一个单词长度>0  或  2.整行为空（flag仍为1）
    if (count != 0 || flag) {
        printf("%d ", count);            // 输出最后一个单词长度或0（空行）
    }

    return 0;
}
```

## 统计一行文本的单词个数

本题目要求编写程序统计一行字符中单词的个数。所谓“单词”是指连续不含空格的字符串，各单词之间用空格分隔，空格数可以是多个

```
#include <stdio.h>

int main() {
    int count = 0;       // 单词计数器
    int in_word = 0;     // 状态标志：0表示不在单词中，1表示在单词中
    char c;              // 当前读取的字符

    // 逐字符读取输入，直到遇到换行符
    while ((c = getchar()) != '\n') {
        if (c == ' ') {                  // 遇到空格字符
            in_word = 0;                 // 标记离开单词状态
        } else if (in_word == 0) {       // 非空格字符，且之前不在单词中
            in_word = 1;                 // 进入单词状态
            count++;                     // 发现新单词，计数器加1
        }
        // 注：连续的非空格字符不会重复计数，因为in_word保持为1
    }

    // 输出统计结果
    printf("%d\n", count);
```

```
    return 0;
}
```

## 求最长的字符串

本题要求编写程序，针对输入的N个字符串，输出其中最长的字符串。

```
#include <stdio.h>
#include <string.h>      // 用于strlen()和strcpy()函数

int main() {
    int n;               // 字符串个数
    int maxl = -1;       // 当前最长字符串长度，初始化为-1确保第一个字符串会被记录
    char maxstr[200] = "";  // 存储当前最长字符串，初始化为空字符串

    // 读取字符串个数，注意\n用于消耗换行符
    scanf("%d\n", &n);

    // 循环读取n个字符串
    for (int i = 1; i <= n; i++) {
        char str[200];    // 存储当前读取的字符串
        gets(str);

        // 计算当前字符串长度
        int l = strlen(str);

        // 如果当前字符串更长，则更新最长字符串记录
        if (maxl < l) {
            maxl = l;                // 更新最长长度
            strcpy(maxstr, str);     // 复制字符串到maxstr
        }
    }

    // 输出结果
    printf("The longest is: %s", maxstr);

    return 0;
}
```

# 字符指针

字符指针除了指向字符类型变量还有其他的用途：

1. 可指向字符串常量，*注意：不能修改字符串常量的内容，如：* `str1[0] = 'h'`

```
char *str1 = "Hello World";  // 字符串常量，存储在只读区域
```

2. 指向字符数组

```
char arr[] = "Hello World";
    char *str2 = arr; // 指向可修改的字符数组
    str2[0] = 'h'; // 正确！可以修改
    printf("str2: %s\n", str2); // 输出：hello World
    str2++;
    printf("str2: %s\n", str2); // 输出：ello World
```

## 字符串数组和字符指针数组

字符串数组本质上是**二维字符数组**，每个字符串有固定大小的空间

```
char names[3][20] = {
    "Alice",
    "Bob",
    "Charlie"
    };
printf("总内存: %zu字节\n", sizeof(names)); //60
```

字符指针数组既可以是多个指向字符变量的指针，也可以是常量字符串数组 *（可按需分配内存）*

## 分析示例：

```
对于·char *str_arr[] = {"Hello", "World", "C", "Programming"};

    字符串 0 "Hello": [0]='H' [1]='e' [2]='l' [3]='l' [4]='o'
    字符串 1 "World": [0]='W' [1]='o' [2]='r' [3]='l' [4]='d'
    字符串 2 "C": [0]='C'
    字符串 3 "Programming": [0]='P' [1]='r' [2]='o' [3]='g' [4]='r' [5]='a'
[6]='m' [7]='m' [8]='i' [9]='n' [10]='g'

1. str_arr = 000000000062FDF0 (数组首地址)
   二维字符数组中names, names[0], &names[0][0]相等

2. str_arr[i] 表达式:
   str_arr[0] = 0000000000405000 (地址) -> "Hello" (内容)
   str_arr[1] = 0000000000405006 (地址) -> "World" (内容)
   str_arr[2] = 000000000040500C (地址) -> "C" (内容)
   str_arr[3] = 000000000040500E (地址) -> "Programming" (内容)

3. &str_arr[i] 表达式:
   &str_arr[0] = 000000000062FDF0 (数组中第0个指针的地址)
   &str_arr[1] = 000000000062FDF8 (数组中第1个指针的地址)
   &str_arr[2] = 000000000062FE00 (数组中第2个指针的地址)
   &str_arr[3] = 000000000062FE08 (数组中第3个指针的地址)
```

4. *str_arr[i] 表达式:
   *str_arr[0] = 'H' (字符串的第一个字符)
   *str_arr[1] = 'W' (字符串的第一个字符)
   *str_arr[2] = 'C' (字符串的第一个字符)
   *str_arr[3] = 'P' (字符串的第一个字符)

5. str_arr[i][j] 表达式:
   str_arr[0][0] = 'H'
   str_arr[3][0] = 'P'

6. *str_arr = 0000000000405000 -> "Hello"
7. **str_arr = 'H'

8. str_arr + i 表达式:
   str_arr + 0 = 000000000062FDF0
   str_arr + 1 = 000000000062FDF8
   str_arr + 2 = 000000000062FE00
   str_arr + 3 = 000000000062FE08

9. *(str_arr + i) 表达式:
   *(str_arr + 0) = 0000000000405000 -> "Hello"
   *(str_arr + 1) = 0000000000405006 -> "World"
   *(str_arr + 2) = 000000000040500C -> "C"
   *(str_arr + 3) = 000000000040500E -> "Programming"

10. *(*str_arr + i) 表达式:
   第一个字符串的字符遍历:
   *(*str_arr + 0) = 'H'
   *(*str_arr + 1) = 'e'
   *(*str_arr + 2) = 'l'
   *(*str_arr + 3) = 'l'
   *(*str_arr + 4) = 'o'

11. *(str_arr+i)+j 表达式分析:
   这是指向第i个字符串第j个字符的指针

   第0个字符串 "Hello":
   *(str_arr+0)+0 = 0000000000405000 (指向字符 'H' 的地址)
   *(str_arr+0)+1 = 0000000000405001 (指向字符 'e' 的地址)
   *(str_arr+0)+2 = 0000000000405002 (指向字符 'l' 的地址)
   *(str_arr+0)+3 = 0000000000405003 (指向字符 'l' 的地址)
   *(str_arr+0)+4 = 0000000000405004 (指向字符 'o' 的地址)

   第1个字符串 "World":
   *(str_arr+1)+0 = 0000000000405006 (指向字符 'W' 的地址)
   *(str_arr+1)+1 = 0000000000405007 (指向字符 'o' 的地址)
   *(str_arr+1)+2 = 0000000000405008 (指向字符 'r' 的地址)
   *(str_arr+1)+3 = 0000000000405009 (指向字符 'l' 的地址)
   *(str_arr+1)+4 = 000000000040500A (指向字符 'd' 的地址)

```
   第2个字符串 "C":
   *(str_arr+2)+0 = 000000000040500C (指向字符 'C' 的地址)

   第3个字符串 "Programming":
   *(str_arr+3)+0 = 000000000040500E (指向字符 'P' 的地址)
   *(str_arr+3)+1 = 000000000040500F (指向字符 'r' 的地址)
   *(str_arr+3)+2 = 0000000000405010 (指向字符 'o' 的地址)
   *(str_arr+3)+3 = 0000000000405011 (指向字符 'g' 的地址)
   *(str_arr+3)+4 = 0000000000405012 (指向字符 'r' 的地址)
   *(str_arr+3)+5 = 0000000000405013 (指向字符 'a' 的地址)
   *(str_arr+3)+6 = 0000000000405014 (指向字符 'm' 的地址)
   *(str_arr+3)+7 = 0000000000405015 (指向字符 'm' 的地址)
   *(str_arr+3)+8 = 0000000000405016 (指向字符 'i' 的地址)
   *(str_arr+3)+9 = 0000000000405017 (指向字符 'n' 的地址)
   *(str_arr+3)+10 = 0000000000405018 (指向字符 'g' 的地址)


12. *(*(str_arr+i)+j) 表达式分析:
   这是第i个字符串第j个字符的值
```

# 结构体

结构体将不同类型的数据组合成**一个整体**，提高代码可读性
结构体可作为一个整体进行传递，可以作为**函数的参数**，也可以作为**函数的返回值**

## 结构体的定义

```c
// 基本语法:
struct 结构体标签 {
    数据类型 成员1;
    数据类型 成员2;
    // ...
    数据类型 成员n;
};

// 示例：学生结构体
struct Student {
    int id;          // 学号
    char name[20];   // 姓名
    int age;         // 年龄
    float score;     // 成绩
};
```

## 结构体变量声明

方法1：**先定义，再声明**，语法：`struct 类型名称 变量名称`

样例：

```
#include <stdio.h>
// 定义结构体类型
struct Student {
    int id;
    char name[20];
    int age;
    float score;
};
int main() {
    // 方法1：先定义类型，再声明变量
    struct Student stu1;
    struct Student stu[10];
    return 0;
}
```

方法2：定义类型的同时声明变量

```
struct Point {
    int x;
    int y;
} p1, p2;  // 声明了两个Point变量
```

方法3：匿名结构体（只能使用一次）

```
struct {
    char title[50];
    char author[50];
    float price;
} book1, book2;
```

## 结构体的初始化

声明时用花括号将结构成员括起来
与数组初始化相同，未初始化的变量默认为0

```
struct Student stu1 = {1001, "Alex", 18, 90.5};
struct Student class2[3] = {
        {1001, "Adam", 18, 90.5},
        {1002, "Bob", 19, 88.0},
        {1003, "Cindy", 19, 92.5}
    };
```

注意，不可用 = 为整个结构体变量赋值，如 `stu1 = class2[0];` 需对结构体中的每个成员进行赋值

## 访问结构体成员

使用点运算符`.`访问结构体成员

```
printf("学号: %d\n", stu.id); // 读取成员值

stu.score = 95.0;
strcpy(stu.name, "Tim"); // 修改成员值
```

## 结构体嵌套

结构体的成员也可以是结构体

调用方式：`变量名.成员名(类型为结构体).成员名`

```
#include <stdio.h>
// 定义日期结构体
struct Date {
    int year;
    int month;
    int day;
};
// 定义学生结构体，包含日期结构体
struct Student {
    int id;
    char name[20];
    struct Date birthday;  // 结构体嵌套
    float score;
};
int main() {
    // 初始化嵌套结构体
    struct Student stu = {1001, "Adam", {2000, 5, 15}, 90.5 };
    // 修改嵌套结构体成员
    stu.birthday.year = 2001;
    stu.birthday.month = 6;
    printf("修改后生日: %d年%d月%d日\n",
           stu.birthday.year, stu.birthday.month, stu.birthday.day);

    return 0;
}
```

## 结构体的大小

使用sizeof运算符
其值=所有成员大小之和（见数组的大小和sizeof运算符）

## 结构体指针

声明：`struct 结构体标签 *指针变量名`
可进行初始化，与指针初始化方法相同

### 通过指针访问结构体成员

1. 使用点运算符和解引用

```
struct Student stu = {1001, "Adam", 18, 90.5}; // 声明结构体指针
struct Student *p = &stu;

(*p).id = 1000;
printf("姓名: %s\n", (*p).name);
```

2. 使用箭头运算符
   语法：`指针名->成员名`，与使用点运算符和解引用同义

```
printf("学号: %d\n", p->id);
p->score = 95.0;
strcpy(p->name, "jqbb");
```

# 文件操作

## 文件的概念

**什么是文件?**
在 C 语言中，**文件（File）** 是存储在外部介质（如硬盘、U盘）上的一组相关数据的集合。我们可以把文件想象成一个“数据容器”。

- **为什么需要文件?** 程序运行时，数据（变量、数组等）都存储在内存（RAM）中。内存的特点是**断电后数据丢失**。为了永久保存数据，我们必须将数据以文件的形式存储到硬盘等外部存储设备上。
- **文件的标识：** 每个文件通过一个唯一的 **“文件路径”** 来定位，例如 `C:\Users\Me\data.txt` 或 `/home/me/data.txt`。
- **操作系统管理：** 文件的创建、删除、读写等操作，最终都需要通过**操作系统**来完成。C 语言的标准库提供了一套函数，让我们能以统一、安全的方式来“请求”操作系统进行文件操作。

## 文本文件和二进制文件

根据数据的组织形式，文件分为两类：

| 特性 | 文本文件 | 二进制文件 |
| --- | --- | --- |
| **内容** | 由字符组成，每个字符对应一个 **ASCII/Unicode** 码。 | 由字节序列组成，是数据在内存中的原始**二进制形式**。 |

| 特性 | 文本文件 | 二进制文件 |
| --- | --- | --- |
| **可读性** | 可以用记事本、代码编辑器等文本工具直接打开和阅读。 | 用文本工具打开会看到乱码，必须用特定程序解读。 |
| **存储效率** | 相对较低。例如，整数 `12345` 在内存中占 4 个字节，但作为文本 `12345\0` 存储可能占 6 个字节（5个字符+结束符）。 | 高。数据以其内存映像形式存储。`12345` 作为 `int` 存储就是固定的 4 个字节。 |
| **常见例子** | `.txt`，`.c`，`.html`，`.csv` | `.exe`，`.jpg`，`.mp3`，`.dat`（自定义数据文件） |
| **C 库函数** | `fprintf`，`fscanf`，`fgets`，`fputs` 等 | `fread`，`fwrite` |

## 缓冲文件系统

为了提高数据读写的效率，C 语言采用了 **“缓冲文件系统”** 来处理文件。

- **什么是缓冲区？** 它是内存中开辟的一块临时存储区。
- **工作原理：**
  1. **写文件时：** 程序输出的数据并不直接写入硬盘，而是先放入**输出缓冲区**。当缓冲区**满了**，或者执行了**刷新**（`fflush`）、**关闭文件**（`fclose`）等操作时，系统才将缓冲区内的所有数据一次性写入硬盘。
  2. **读文件时：** 系统会从硬盘中一次性读入一大块数据到**输入缓冲区**。程序需要数据时，直接从缓冲区里取。缓冲区数据用完后，系统再自动读取下一块。
- **优点：**
  - **减少硬盘访问次数**。
  - 降低程序与操作系统的交互开销。

## typedef（定义与使用方法）

`typedef` 是 C 语言的关键字，用于为已有的数据类型**创建新的名称（别名）**。

**基本语法：**

```
typedef 已有类型 新类型名;
```

**示例：**

```
// 给 unsigned long long 起一个简单的新名字
typedef unsigned long long ull;
ull big_number = 18446744073709551615; // 使用新名字声明变量

// 给字符指针起新名字
typedef char* String;
String name = “Alice”; // 等价于 char* name = “Alice”;
```

```c
// 定义结构体
typedef struct {
    char name[20];
    int id;
    int age;
    float score;
} student;
```

在文件操作中，标准库使用 `typedef` 将复杂的结构体类型定义为 `FILE`。

## stdio.h 中的 FILE 结构类型

在 `stdio.h` 头文件中，定义了一个非常重要的结构体类型 `FILE` 。

```c
// 标准库中大致如下（具体实现因编译器而异，用户无需知道细节）
typedef struct {
    short level; // 缓冲区使用量
    unsigned flags; // 文件状态标志
    char fd; // 文件描述符
    short bsize; // 缓冲区大小
    unsigned char *buffer; // 文件缓冲区首地址
    unsigned char *curp; // 指向文件缓冲区的工作指针
    unsigned char hold;
    unsigned istemp;
    short token;
} FILE;
```

- `FILE` 是一个**结构体类型**。
- 这个结构体封装了管理一个文件流所需的所有信息
- 可以把 `FILE` 理解为一个 **“文件控制块”** 或 **“文件句柄”**，它是程序与操作系统之间关于某个文件进行通信的“联络官”或“遥控器”。

## 文件指针

既然 `FILE` 是一个类型，我们就可以用它来声明指针变量。

**文件型指针（常直接称为文件指针）**：指向 `FILE` 结构体变量的指针。

```c
FILE *fp; // 声明一个文件指针
```

**这个指针 `fp` 的作用：**
它是所有文件操作的**核心**
当你用 `fopen` 打开一个文件时，函数会动态创建一个 `FILE` 结构体变量（包含该文件的所有控制信息），并返回指向它的指针
所有针对该文件的操作函数（如 `fprintf`，`fgets`，`fclose`）都需要传入这个指针 `fp`

# 文件处理基本步骤

处理任何文件，都必须遵循以下四个标准步骤：

1. **定义文件指针**

```
FILE *fp;
```

2. **打开文件**

```
fp = fopen("文件名", "打开方式");
// 必须检查是否打开成功！
if (fp == NULL) {
    printf("打开文件失败");
    return 1; // 失败则通常终止或返回错误
}
```

3. **文件处理（读/写）**

```
// 使用各种读写函数，如 fputc, fgets, fread 等
// 所有函数都需要传入文件指针 fp
```

4. **关闭文件**

```
fclose(fp);
fp = NULL;
```

**为什么必须关闭文件？**

- 将缓冲区中最后残留的数据**强制写入硬盘**（刷新缓冲区）。
- 释放 `FILE` 结构体占用的内存和缓冲区。
- 释放文件描述符。*一个进程能同时打开的文件数是有限的，如果不关闭，会导致“文件描述符泄露”，最终无法再打开新文件。*
  **重要：** `fopen` 和 `fclose` 必须成对出现！

# 文件的打开 (fopen)

**函数原型：**

```
FILE *fopen(const char *filename, const char *mode);
```

**参数解释：**

- `filename`：字符串，表示要打开的文件路径（相对或绝对路径）。
- `mode`：字符串，表示文件的**打开方式**。

**返回值：**

- **成功**：返回一个指向 `FILE` 结构体的指针（即文件指针）。

- **失败**：返回 `NULL`。**失败原因可能是文件不存在、无权限、磁盘满等。所以必须检查返回值！**

## 文件打开方式 (mode)

| 模式字符串 | 含义 | 文件必须存在? | 文件不存在时 | 写入位置 |
| --- | --- | --- | --- | --- |
| `“r”` | 只读（read） | **是** | 返回 `NULL` | 从头开始 |
| `“w”` | 只写（write） | 否 | **创建新文件** | 从头开始。**若存在，内容被清空！** |
| `“a”` | 追加（append） | 否 | 创建新文件 | **自动定位到文件末尾** |
| `“r+”` | 读写（read+） | **是** | 返回 `NULL` | 从头开始 |
| `“w+”` | 读写（write+） | 否 | **创建新文件** | 从头开始。**若存在，内容被清空！** |
| `“a+”` | 读写（append… | 否 | 创建新文件 | **读从头开始，写自动追加** |

**对于二进制文件，在模式字符串后加一个 `”b“`**

**示例：**

```
// 以只读方式打开一个文本文件
FILE *fp1 = fopen(“data.txt”, “r”);

// 以二进制写入方式打开（创建或覆盖）
FILE *fp2 = fopen(“image.dat”, “wb”);

// 以追加文本方式打开（在末尾添加日志）
FILE *fp3 = fopen(“log.txt”, “a”);
```

## 文件打开的实质

当 `fopen(“test.txt”, “w”)` 成功执行时，发生了三件事：

1. **关联磁盘文件：** 在磁盘上找到或创建名为 `test.txt` 的文件。
2. **建立文件缓冲区：** 在内存中为这个文件分配一块输入/输出缓冲区，相关信息记录在 `FILE` 结构体中。
3. **确定操作方式：** 根据模式 `“w”`，设置 `FILE` 结构体中的状态标志，表明这是一个“只写”流，且初始写入位置在文件开头。

**允许打开多个文件吗？** 是的，可以同时打开多个文件，只需用不同的文件指针变量指向它们即可，例如 `FILE *fp1, *fp2, *fp3;`。但系统对一个进程同时打开的文件总数有限制。

# 关闭文件 (fclose)

**函数原型：**

```
int fclose(FILE *stream);
```

- `stream`：要关闭的文件指针。
- **返回值：** 成功返回 0；失败返回 `EOF`（通常是 -1），并设置错误标识。

**作用：** 如前所述，刷新并关闭流，释放所有相关资源。

## 检测文件结束 (feof)

**函数原型：**

```
int feof(FILE *stream);
```

- `stream`：文件指针。
- **返回值：** 如果已经到达了文件末尾，返回**非零值（真）**；否则返回 **0（假）**。

**重要：`feof` 的常见误用**
`feof` 不是在读取了最后一个字节后立刻返回“真”。它是在**尝试读取数据失败（因为已到结尾）后**，才将文件结束标志置位，之后调用 `feof` 才会返回真。

**错误用法：**

```
while (!feof(fp)) { // 这个判断是错误的！
    ch = fgetc(fp);
    putchar(ch);
}
// 上面的循环会导致最后一个字符被输出两次。
```

**正确用法：** 先读取，再判断是否成功，最后用 `feof` 判断是否是因为文件结束而失败的。

```
while ((ch = fgetc(fp)) != EOF) { // 这才是控制循环的正确条件
    putchar(ch);
}
// 循环结束后，如果想区分是文件结束还是其他错误，可以用 feof 或 ferror
if (feof(fp)) {
    printf("已到达文件末尾。\n");
} else if (ferror(fp)) {
    printf("读取文件时发生错误。\n");
}
```

# 文件的读写函数

## 按字符读写 (fgetc, fputc)

- `int fgetc(FILE *stream);`
  - 从 `stream` 指向的文件中读取**一个字符**。
  - 返回读取的字符（转换为 `unsigned char` 再转 `int`）。
  - 如果到达文件末尾或出错，返回 `EOF`（通常是 -1）。

```
int ch;
while ((ch = fgetc(fp)) != EOF) {
    putchar(ch); // 输出到屏幕
}
```

- `int fputc(int c, FILE *stream);`
  - 将字符 `c`（转换为 `unsigned char`）写入 `stream` 指向的文件。
  - 成功返回写入的字符，失败返回 `EOF`。

```
char str[] = "Hello";
for (int i = 0; str[i] != '\0'; i++) {
    fputc(str[i], fp); // 将字符串按字符写入文件
}
```

## 按字符串读写 (fgets, fputs)

- `char *fgets(char *str, int n, FILE *stream);`
  - 从 `stream` 中读取最多 `n-1` 个字符，放到字符数组 `str` 中。
  - 在读取 `n-1` 个字符、遇到换行符 `'\n'` 或到达文件末尾时停止。
  - 自动在读取的字符后添加 `'\0'` 终止符。
  - **换行符 `'\n'` 也会被读入字符串**。
  - 成功返回 `str`，失败或到达文件末尾返回 `NULL`。

```
char buffer[100];
while (fgets(buffer, 100, fp) != NULL) {
    printf("%s", buffer); // 一次读取一行（包含换行符）
}
```

- `int fputs(const char *str, FILE *stream);`
  - 将字符串 `str`（**不包含结尾的 `'\0'`**）写入 `stream`。
  - 成功返回非负整数，失败返回 `EOF`。
  - 它**不会自动在末尾添加换行符**，如果需要换行，字符串里要包含 `'\n'`。

```
fputs("Hello, World!\n", fp); // 写入一行
```

## 格式化读写 (fscanf, fprintf)

与 `scanf` 和 `printf` 用法几乎一样，只是第一个参数是文件指针。

- `int fprintf(FILE *stream, const char *format, ...);`

```
int age = 20;
float score = 95.5;
```

```
fprintf(fp, “Name: %s, Age: %d, Score: %.1f\n”, “Alice”, age, score);
// 文件内容：Name: Alice, Age: 20, Score: 95.5
```

- `int fscanf(FILE *stream, const char *format, ...);`

```
char name[20];
int age;
float score;
fscanf(fp, “Name: %s, Age: %d, Score: %f”, name, &age, &score);
// 从文件读取格式化数据，注意变量前要加 &
```

**注意：** `fscanf` 和 `scanf` 一样，对输入格式要求严格

### 按数据块读写 (fread, fwrite) *不考*

主要用于**二进制文件**，能高效地读写数组、结构体等连续内存块。

## 其他重要函数

### 文件定位 (rewind, fseek, ftell)

用于在文件中移动“读写位置指针”，实现随机访问。

- `void rewind(FILE *stream);`
  - 将文件的位置指针重新设置到文件的**开头**。等价于 `fseek(stream, 0L, SEEK_SET);`。
- `int fseek(FILE *stream, long offset, int whence);`
  - 将文件位置指针移动到特定位置。
  - `offset`：偏移量（长整型），可正可负。
  - `whence`：起始点，取以下值之一：
    - `SEEK_SET` (0)：文件开头。
    - `SEEK_CUR` (1)：当前位置。
    - `SEEK_END` (2)：文件末尾。

```
fseek(fp, 10L, SEEK_SET); // 移动到离开头10字节的位置
fseek(fp, -5L, SEEK_CUR); // 从当前位置向前（文件头方向）移动5字节
fseek(fp, -20L, SEEK_END); // 移动到离文件末尾20字节的位置
```

- `long ftell(FILE *stream);`
  - 返回文件位置指针的**当前值**（相对于文件开头的字节数）。
  - 可用于获取文件大小（结合 `fseek` 到文件尾）。

```
fseek(fp, 0L, SEEK_END); // 移动到文件尾
long file_size = ftell(fp); // 获取当前位置（即文件大小，字节数）
rewind(fp); // 回到文件头准备读取
```

### 错误处理 (ferror, clearerr)

- `int ferror(FILE *stream);`

    - 检查文件流是否发生了**读写错误**（非文件结束错误）。
    - 如果发生了错误，返回非零值。
- `void clearerr(FILE *stream);`
    - 清除文件流的**错误标志**和**文件结束标志**。
    - 在检测并处理了错误或文件尾后，如果想继续尝试操作，可以先调用此函数清除状态。

## 顺序存取与随机存取

- **顺序文件（顺序存取）：**
    - 读写操作必须从头到尾顺序进行。
    - 使用 `fgetc`，`fgets`，`fscanf` 等函数，如果不配合 `fseek`，就是顺序读写。
- **随机文件（随机存取）：**
    - 可以直接定位到文件的任意位置进行读写。就像 CD 或硬盘，可以直接跳转到任意扇区。
    - 通过 `fseek` 函数改变文件位置指针，然后进行读写，即可实现随机存取。
    - 常用于数据库、索引文件等场景。

**示例（随机存取）：**

```
struct Record {
    int id;
    char data[100];
};

// 假设每个Record结构体大小固定，要读取第3条记录（下标从0开始）
fseek(fp, 2 * sizeof(struct Record), SEEK_SET); // 移动到第3条记录开头
struct Record rec;
fread(&rec, sizeof(struct Record), 1, fp);
```

# `ctype.h` 常用函数

## 字符分类函数（返回非零值表示真）

### 字母数字相关

- `isalnum()` - 检查字符是否为字母或数字
- `isalpha()` - 检查字符是否为字母
- `isdigit()` - 检查字符是否为十进制数字（0-9）
- `isxdigit()` - 检查字符是否为十六进制数字（0-9, A-F, a-f）

### 大小写字母

- `islower()` - 检查字符是否为小写字母

- `isupper()` - 检查字符是否为大写字母

### 特殊字符

- `isspace()` - 检查字符是否为空白字符（空格、换行、制表符等）
- `ispunct()` - 检查字符是否为标点符号
- `iscntrl()` - 检查字符是否为控制字符
- `isgraph()` - 检查字符是否为图形字符（除空格外所有可打印字符）
- `isprint()` - 检查字符是否为可打印字符（包括空格）

## 字符转换函数

- `tolower()` - 将大写字母转换为小写字母
- `toupper()` - 将小写字母转换为大写字母

# `math.h` 常用函数

## 基本数学运算

- `fabs()` - 浮点数绝对值（参数为double）
- `sqrt()` - 平方根（参数须非负）
- `pow()` - 幂运算（计算x的y次方）

## 取整函数

- `ceil()` - 向上取整（返回≥参数的最小整数）
- `floor()` - 向下取整（返回≤参数的最大整数）

# `stdlib.h` 常用函数

## 整数运算

- `abs()` - 整数绝对值（参数为int）

## 字符串转换函数

- `atof()` - 字符串转浮点数（参数为字符串，返回double）
- `atoi()` - 字符串转整数（参数为字符串，返回int）
- `atol()` - 字符串转长整数（参数为字符串，返回long）