

第一章

1. 单选题

(1) C. 机器语言

解析：机器语言由二进制指令构成，计算机硬件可直接执行，无需翻译。高级语言需编译或解释，汇编语言需汇编器转换为机器码。

(2) A. 丹尼斯·里奇，贝尔实验室

解析：C 语言由丹尼斯·里奇于 1972 年在贝尔实验室开发，最初用于 Unix 系统的开发。

(3) B. main

解析：C 程序必须包含且仅包含一个 main 函数，它是程序执行的入口点。

(4) C. #include <stdio.h>

解析：以#开头的指令是预处理指令，#include 用于引入标准库头文件。

(5) B. _score

解析：C 标识符必须以字母或下划线开头，不能包含特殊符号或数字开头，且不能是关键字（如 int）。

(6) C. 解释

解析：C 程序的运行流程包括编辑、编译、链接和运行，不包含解释阶段（解释是 Python 等语言的特性）。

(7) C. 可读性和可移植性强

解析：高级语言（如 C）用自然语言语法，代码易读，且通过编译器可跨平台编译，无需针对不同硬件重写。

(8) B. 32 个

解析：ANSI C 标准（C89）定义了 32 个关键字，后续版本（如 C99、C11）有所扩展，但题目默认考察基础标准。

(9) B. .obj

解析：编译器将.c 文件编译为目标文件（Windows 下为.obj，Linux 下为.o），链接后生成可执行文件（.exe）。

(10) B. 单步调试

解析：在 VS Code 中，按 F11（单步调试）可进入函数内部逐行执行，F10（单步跳过）则执行当前行但不进入函数。

2. 简答题

(1) 程序设计的四个核心阶段

问题分析：明确需求，例如“计算学生成绩平均分”。

算法设计：设计解决问题的步骤（如排序、遍历），核心目标是找到高效且正确的逻辑流程。

代码实现：将算法转化为 C 语言代码，例如用 for 循环遍历数组。

测试调试：检查代码是否满足需求，修正逻辑或语法错误。

示例：在算法设计阶段，若需求是“找出最大值”，需设计比较逻辑（如逐个比较或使用数学函数）。

(2) C 语言的三个核心特点

高效性：生成的机器码接近汇编语言，执行速度快。

可移植性：通过标准库和编译器适配，代码可在不同平台运行。

结构化编程：支持函数、条件语句、循环等，便于代码组织。

兼顾高级抽象与底层控制：

C 语言提供 `if-else`、`for` 等高级结构提升可读性，同时允许通过指针直接操作内存地址，适用于系统开发。

（3）C 程序从源代码到可执行程序的流程

编辑：输入代码保存为 `.c` 文件（输入：文本编辑器；输出：`.c` 文件）。

预处理：处理 `#include`、`#define` 等指令，生成 `.i` 文件（输入：`.c`；输出：`.i`）。

编译：将 `.i` 文件转换为汇编代码（`.s`），检查语法错误（输入：`.i`；输出：`.s`）。

汇编：将汇编代码转换为二进制目标文件（`.obj`），生成机器指令（输入：`.s`；输出：`.obj`）。

链接：将多个 `.obj` 文件和库文件合并为可执行文件（`.exe`），解析外部符号（如 `printf` 的实现）。

3. 编程题

（1）输出欢迎信息

```
#include <stdio.h>
int main() {
    printf("欢迎来到 C 语言的世界！\n");
    printf("这是我的第一个 C 程序。\\n");
    return 0;
}
```

解析：使用 `printf` 函数输出多行文本，`\\n` 表示换行。

（2）计算整数的平方

```
#include <stdio.h>
int main() {
    int num, square;
    printf("请输入一个整数：");
    scanf("%d", &num);
    square = num * num;
    printf("该数的平方是： %d\\n", square);
    return 0;
}
```

解析：`scanf` 读取用户输入，`%d` 指定整数格式，计算平方后通过 `printf` 输出结果。

（3）找出三个整数中的最大值

```
#include <stdio.h>
int main() {
    int a, b, c, max;
    printf("请输入第一个整数：");
    scanf("%d", &a);
    printf("请输入第二个整数：");
    scanf("%d", &b);
    printf("请输入第三个整数：");
    scanf("%d", &c);
    if (a > b) {
        max = a;
    } else {
```

```

        max = b;
    }
    if (c > max) {
        max = c;
    }
    printf("最大值是: %d\n", max);
    return 0;
}

```

解析：使用两次 if-else 比较：先比较前两个数，再将结果与第三个数比较，最终输出最大值。

第二章

1. 单选题

- (1) B 解析：算法是解决特定问题的一系列明确且有序的步骤，与硬件无关，需有输出，可用多种方式表示。
- (2) A 解析：有穷性指算法必须在有限步骤后结束，与输入 / 输出数量、执行时间无直接关联。
- (3) C 解析：流程图中菱形表示判断条件，椭圆表示起止，平行四边形表示输入输出，长方形表示处理步骤。
- (4) B 解析：N-S 图取消流程线，通过嵌套结构表示逻辑，支持所有基本结构，可用中英文描述。
- (5) D 解析：结构化程序设计的三种基本结构为顺序、选择、循环，跳转结构（如 goto）不符合结构化思想。
- (6) C 解析：伪代码灵活易懂，无需严格语法，不能直接执行，可中英文混合，保留 IF、WHILE 等关键词。
- (7) D 解析：结构化程序设计反对使用 goto 语句，核心原则包括自顶向下、逐步细化、模块化设计、结构化编码。
- (8) B 解析：算法必须有一个或多个输出，输出形式可以是数字、文字等。
- (9) B 解析：当型循环先判断条件，条件成立则执行循环体；直到型循环先执行循环体，再判断条件，条件成立则退出。
- (10) B 解析：模块化设计中模块独立，通过接口（如函数参数、返回值）通信，与内部变量、全局变量无关。

2. 填空题

- (1) 输入；输出 解析：算法可接收零个或多个输入，但必须有至少一个输出。
- (2) 选择结构；循环结构 解析：三种基本结构是构成结构化算法的核心。
- (3) 椭圆；平行四边形 解析：流程图符号的标准定义。
- (4) Nassi；Shneiderman 解析：N-S 图名称来源于两位提出者的姓氏。
- (5) IF；WHILE（或 FOR） 解析：伪代码常用结构化关键词描述逻辑。
- (6) 模块化设计；结构化编码 解析：结构化程序设计的四大核心原则。
- (7) 当；条件 解析：当型循环“先判断后执行”，直到型循环“先执行后判断”。
- (8) N-S 图；编程语言 解析：算法的五种主要表示方法。
- (9) 特定；不可见 解析：模块化设计要求模块功能单一，内部细节封装（不可见）。

(10) goto 解析: goto 语句会破坏程序结构化逻辑, 应避免使用。

3. 编程题

(1) 自然语言描述:

- 输入: 提示用户输入两个整数 a 和 b;
- 计算: 计算 $sum = a + b$;
- 输出: 显示 “两个数的和为: sum”。

流程图省略

(2)

```
#include <stdio.h>
int main() {
    int num;
    printf("请输入一个整数: ");
    scanf("%d", &num);
    if (num % 3 == 0) {
        printf("%d 是 3 的倍数\n", num);
    } else {
        printf("%d 不是 3 的倍数\n", num);
    }
    return 0;
}
```

(3)

```
#include <stdio.h>
int main() {
    int n, sum = 0;
    printf("请输入正整数 n: ");
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) {
        if (i % 2 == 0) { // 判断是否为偶数
            sum += i;
        }
    }
    printf("1 到%d 的偶数和为: %d\n", n, sum);
    return 0;
}
```

第三章

1. 单选题

1-5 BDBCC

2. 编程题

使用 AI 大模型优化后的结果如下:

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

```

int main() {
    int a, b;
    int result = scanf("%d %d", &a, &b);
    if (result != 2) {
        printf("输入错误\n");
        return 1;
    }
    printf("和: %d\n 差: %d", a + b, a - b);
    return 0;
}

```

优化后的代码判断了输入数据是否正确，提供了程序的健壮性。

第四章

1. 单选题

- (1) C 解析: struct 属于构造数据类型, int、float、char 是基本数据类型。
- (2) B 解析: const 定义的常量有类型信息, 可进行类型检查; #define 是预处理替换, 无类型。
- (3) B 解析: 字符串 "abc" 包含 'a'、'b'、'c' 和结束符 '\0', 共 4 个字节。
- (4) B 解析: int 与 float 运算时, int 自动转换为 float, 结果为 float 类型。
- (5) D 解析: %s 用于输出字符串, %c 用于输出单个字符。
- (6) B 解析: scanf 需要变量地址, 通过 & 运算符获取; printf 用于输出, getchar 用于输入单个字符。
- (7) B 解析: 整数除法会舍弃小数部分, 10/3 结果为 3。
- (8) B 解析: 乘法 (*) 优先级高于加法 (+)、赋值 (=) 和关系运算符 (>)。
- (9) A 解析: 5 的二进制为 0101, 3 为 0011, 按位与后为 0001, 即 1。
- (10) B 解析: 字符数据在内存中存储 ASCII 码值, 占用 1 个字节, 可与整型互相转换。

2. 填空题

- (1) 指针类型; 空类型 (void) 解析: C 语言数据类型的四大分类。
- (2) #define; const 解析: 两种定义常量的方式, #define 是预处理指令, const 是关键字。
- (3) '\0'; n+1 解析: 字符串结束标志为 '\0', 额外占用 1 个字节。
- (4) int; float 解析: 自动转换向精度更高的类型转换。
- (5) %d; %f 解析: printf 的基本格式符定义。
- (6) &; 地址 解析: scanf 需要变量地址才能存储输入值。
- (7) 先自增后取值; 先取值后自增 解析: 自增运算符的两种形式区别。
- (8) 0 (假); 1 (真) 解析: 关系运算结果为逻辑假 (0) 或真 (1)。
- (9) 二进制; ASCII 解析: 计算机底层存储形式及字符存储本质。
- (10) 按位与; 按位或; 按位异或 解析: 位运算符的基本功能。

3. 编程题

(1)

```

#include <stdio.h>
#define PI 3.14159 // 定义圆周率常量
int main() {

```

```

float radius, perimeter, area;
printf("请输入圆的半径: ");
scanf("%f", &radius);

perimeter = 2 * PI * radius; // 计算周长
area = PI * radius * radius; // 计算面积

printf("圆的周长: %.2f, 圆的面积: %.2f\n", perimeter, area);
return 0;
}

```

(2)

```

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

    printf("请输入第一个整数: ");
    scanf("%d", &a);
    printf("请输入第二个整数: ");
    scanf("%d", &b);
    printf("请输入第三个整数: ");
    scanf("%d", &c);

    sum = a + b + c; // 计算总和
    avg = sum / 3.0f; // 计算平均值 (用 3.0 确保浮点除法)

    printf("总和: %d, 平均值: %.1f\n", sum, avg);
    return 0;
}

```

(3)

```

#include <stdio.h>
int main() {
    float base, height, area;

    printf("请输入三角形的底: ");
    scanf("%f", &base);
    printf("请输入三角形的高: ");
    scanf("%f", &height);

    area = base * height / 2; // 计算面积

    printf("三角形的面积为: %.1f\n", area);
    return 0;
}

```

第五章

1. 单选题

- (1) B 解析: if 语句的语法为 if (表达式) 语句, 表达式必须用圆括号包裹。
- (2) C 解析: 条件表达式的结果为逻辑真 (非 0, 通常用 1 表示) 或假 (0)。
- (3) B 解析: a=5 满足 a>3, 执行第二个分支, 输出 B。
- (4) C 解析: switch 的表达式必须是整数类型 (int、char、enum 等), 浮点型 (float) 不允许。
- (5) B 解析: break 用于终止当前 case 的执行, 并跳出 switch 语句。
- (6) B 解析: score=85 满足 score>=60 但不满足 score>=90, 执行内层 else, 输出合格。
- (7) A 解析: case 后无 break 时, 程序会继续执行下一个 case 的代码, 即穿透现象。
- (8) B 解析: 星期几是离散的整数 (1-7), 适合用 switch 处理; A、C 适合用 if-else, D 需结合运算符判断。
- (9) B 解析: x=3 匹配 case 3, 无 break, 继续执行 case 4, 遇到 break 跳出, 输出 34。
- (10) B 解析: 嵌套 if 中, 必须先满足外层 if 条件, 才会进入内层 if 的判断。

2. 填空题

- (1) 条件 解析: 分支结构的核心是根据条件真假选择执行路径。
- (2) 非 0 (真); 0 (假) 解析: if 语句通过表达式的真假 (非 0/0) 决定是否执行语句块。
- (3) 2; 3 解析: if-else 处理二选一逻辑, if-else if-else 处理多分支。
- (4) 外层 if 解析: 嵌套 if 的内层条件依赖外层条件的成立。
- (5) 整数; 常量 解析: switch 表达式必须是整数类型, case 后必须是常量值。
- (6) default; 可选 解析: default 处理未匹配的情况, 可省略但建议添加。
- (7) 穿透 解析: 无 break 时的穿透现象会导致后续 case 代码被执行。
- (8) 1 (真); 0 (假) 解析: 关系和逻辑表达式的结果用 1 和 0 表示真和假。
- (9) 能被 400 整除; (year%4==0 && year%100!=0) || (year%400==0) 解析: 闰年的判断条件。
- (10) if; switch 解析: C 语言分支结构的两种主要实现方式。

3. 编程题

(1)

```
#include <stdio.h>

int main() {
    int num;
    printf("请输入一个整数: ");
    scanf("%d", &num);

    if (num > 0) {
        printf("%d 是正数\n", num);
    } else if (num < 0) {
        printf("%d 是负数\n", num);
    } else {
        printf("%d 是零\n", num);
    }
}
```

```

    }

    return 0;
}

```

(2)

```

#include <stdio.h>

int main() {
    int month;
    printf("请输入月份（1-12）： ");
    scanf("%d", &month);

    if (month < 1 || month > 12) {
        printf("输入错误！\n");
    } else if (month >= 3 && month <= 5) {
        printf("%d 月是春季\n", month);
    } else if (month >= 6 && month <= 8) {
        printf("%d 月是夏季\n", month);
    } else if (month >= 9 && month <= 11) {
        printf("%d 月是秋季\n", month);
    } else { // 12、1、2 月
        printf("%d 月是冬季\n", month);
    }

    return 0;
}

```

(3)

```

#include <stdio.h>

int main() {
    int level;
    float amount, discount_amount, pay;

    printf("请输入会员等级（1：普通，2：银卡，3：金卡）： ");
    scanf("%d", &level);
    printf("请输入消费金额： ");
    scanf("%f", &amount);

    // 判断会员等级并计算折扣后金额
    if (level == 1) {
        discount_amount = amount; // 普通会员无折扣
    } else if (level == 2) {
        discount_amount = amount * 0.9; // 银卡 9 折
    } else if (level == 3) {
        discount_amount = amount * 0.8; // 金卡 8 折
    } else {
        printf("会员等级无效！\n");
        return 0; // 退出程序
    }
}

```



```

// 计算满减
if(discount_amount >= 1000) {
    pay = discount_amount - 120;
} else if (discount_amount >= 500) {
    pay = discount_amount - 50;
} else {
    pay = discount_amount; // 不满足满减
}

printf("实际支付金额: %.1f 元\n", pay);
return 0;
}

```

第六章

1. 单选题

- (1) B 解析: **while** 循环先判断条件, 条件为假则不执行循环体; **do-while** 循环先执行一次循环体, 再判断条件, 因此至少执行一次。
- (2) D 解析: **for** 循环的三个部分为初始化、条件判断、循环后操作, 不包含循环体返回值。
- (3) A 解析: 循环计算 $1+2+3+4+5=15$, 输出 15。
- (4) B 解析: **break** 用于立即终止当前循环或 **switch** 语句; **continue** 跳过本次循环剩余部分; **return** 终止函数; **goto** 跳转至标签。
- (5) A 解析: 当 *i* 为 3、6、9 时执行 **continue**, 跳过输出, 因此输出 1 2 4 5 7 8 10。
- (6) B 解析: 循环嵌套中, 外层循环每执行一次, 内层循环完整执行一轮 (满足内层条件的所有次数)。
- (7) B 解析: **do-while** 循环至少执行一次, 适合菜单选择等需至少显示一次的场景; A、C、D 适合 **for** 或 **while** 循环。
- (8) A 解析: *i* 从 0 开始, **do-while** 循环先执行 $i++$ ($i=1$), 输出 1; $i=2$ 时输出 2; $i=3$ 时 **break**, 循环终止, 最终输出 1 2。
- (9) C 解析: **break** 在嵌套循环中仅终止当前所在的循环 (如内层循环的 **break** 不影响外层)。
- (10) B 解析: 死循环的原因是循环条件始终为真, 因此必须保证循环条件最终会变为假。

2. 填空题

- (1) 循环条件判断; 循环变量更新 解析: 循环三要素确保循环可控执行有限次。
- (2) 判断条件; 判断条件 解析: **while** 先判断后执行, **do-while** 先执行后判断。
- (3) 条件表达式; 循环后操作表达式 解析: **for** 循环的语法格式为 **for**(初始化; 条件; 更新)。
- (4) 循环体; 循环结构 解析: 循环嵌套是在一个循环内部包含另一个完整循环。
- (5) 终止; 跳过 解析: **break** 终止整个循环, **continue** 跳过本次剩余部分。
- (6) 死循环 解析: 循环条件始终为真会导致无限循环 (死循环)。
- (7) 1 解析: **do-while** 循环至少执行一次循环体。
- (8) 1; 一 解析: 外层循环每执行一次, 内层循环完整执行一轮。
- (9) 行数; 列数 解析: 九九乘法表外层控制行数, 内层控制每行的列数。

(10) 标签：退出 解析：goto 通过标签跳转，偶尔用于退出多层嵌套循环，但不推荐频繁使用。

3. 编程题

(1)

```
#include <stdio.h>

int main() {
    int n, sum = 0;
    printf("请输入正整数 n: ");
    scanf("%d", &n);

    for (int i = 1; i <= n; i++) {
        // 判断是否能被 3 或 5 整除
        if (i % 3 == 0 || i % 5 == 0) {
            sum += i;
        }
    }

    printf("1 到%d 中能被 3 或 5 整除的数的和为: %d\n", n, sum);
    return 0;
}
```

(2)

```
#include <stdio.h>

int main() {
    int n;
    printf("请输入直角三角形的行数 n: ");
    scanf("%d", &n);

    // 外层循环控制行数
    for (int i = 1; i <= n; i++) {
        // 内层循环控制每行星号数量（第 i 行 i 个星号）
        for (int j = 1; j <= i; j++) {
            printf("*");
        }
        printf("\n"); // 每行结束换行
    }

    return 0;
}
```

(3)

```
#include <stdio.h>
#include <math.h> // 用于 sqrt 函数

int main() {
    int m;
    printf("请输入正整数 m: ");
    scanf("%d", &m);

    printf("1 到%d 之间的素数有: ", m);
}
```

```

// 遍历 1 到 m 的所有数
for (int num = 2; num <= m; num++) { // 素数从 2 开始
    int is_prime = 1; // 标记是否为素数
    // 检查是否有因数（优化：只需检查到 sqrt(num)）
    for (int i = 2; i <= sqrt(num); i++) {
        if (num % i == 0) { // 若能被 i 整除，则不是素数
            is_prime = 0;
            break; // 找到因数，提前退出内层循环
        }
    }
    if (is_prime) {
        printf("%d ", num); // 输出素数
    }
}
printf("\n");

return 0;
}

```

第七章

1. 单选题

- (1) B 解析：一维数组下标从 0 开始，依次递增。
- (2) D 解析：二维数组元素个数 = 行数 × 列数， $3 \times 4 = 12$ 。
- (3) C 解析：字符数组以 '\0' 作为字符串结束标志，系统自动添加。
- (4) B 解析：数组部分初始化时，未赋值元素自动为 0，故 arr[3]=0。
- (5) B 解析：数组越界编译时不报错，但运行时可能导致程序崩溃或数据错误。
- (6) B 解析："Hello" 包含 5 个字符，系统添加 '\0'，故数组长度为 6。
- (7) A 解析：二维数组遍历需外层控制行（0~1），内层控制列（0~2）。
- (8) B 解析：数组作为参数时传递首地址，函数内修改会影响原数组。
- (9) B 解析：线性表元素间是一对一相邻关系，可通过数组或链表实现，长度可变。
- (10) C 解析：fgets 可指定最大读取长度，避免越界；scanf 和 gets 无长度限制，不安全。

2. 填空题

- (1) 数据类型；数组长度 解析：一维数组定义需指定数据类型和长度。
- (2) 20；0~3 解析：4 行 5 列共 20 个元素，行下标从 0 到 3（4-1）。
- (3) '\0'；1 解析：字符串结束符占 1 字节，数组长度需至少为字符串长度 + 1。
- (4) 初始化列表的元素个数；3 解析：编译器根据初始化元素个数推断长度，此处为 3。
- (5) 0~ 数组长度 - 1；越界 解析：下标超出范围会导致数组越界。
- (6) 行；列 解析：二维数组嵌套循环外层控制行，内层控制列。
- (7) '\0'；长度 解析：字符串长度是到 '\0' 的字符个数，不含结束符。
- (8) 删除；查找 解析：线性表的基本操作包括插入、删除、查找等。
- (9) n-1-i；数组中间 解析：反转需交换对称位置元素，循环到中间即可。

(10) 空格；数组越界 解析：scanf 遇空格停止，输入过长会导致越界。

3. 编程题

(1)

```
#include <stdio.h>

int main() {
    int arr[10];
    int sum = 0;
    float avg;

    printf("请输入 10 个整数: ");
    for (int i = 0; i < 10; i++) {
        scanf("%d", &arr[i]);
        sum += arr[i]; // 累加元素值
    }

    avg = sum / 10.0f; // 计算平均值 (用 10.0 确保浮点除法)
    printf("数组的平均值为: %.1fn", avg);

    return 0;
}
```

(2)

```
#include <stdio.h>

int main() {
    int n;
    printf("请输入杨辉三角的阶数 n: ");
    scanf("%d", &n);

    int yanghui[n][n]; // 定义 n 阶杨辉三角数组

    // 初始化杨辉三角
    for (int i = 0; i < n; i++) {
        yanghui[i][0] = 1; // 每行首元素为 1
        yanghui[i][i] = 1; // 每行尾元素为 1

        // 计算中间元素 (从第 2 行开始)
        for (int j = 1; j < i; j++) {
            yanghui[i][j] = yanghui[i-1][j-1] + yanghui[i-1][j];
        }
    }

    // 打印杨辉三角
    for (int i = 0; i < n; i++) {
        for (int j = 0; j <= i; j++) {
            printf("%d ", yanghui[i][j]);
        }
        printf("\n"); // 每行结束换行
    }
}
```

```

    }

    return 0;
}

```

(3)

```

#include <stdio.h>
#include <ctype.h> // 用于 tolower 函数

int main() {
    char str[100];
    int count = 0;

    printf("请输入一个字符串: ");
    fgets(str, 100, stdin); // 安全读取字符串（含空格）

    // 遍历字符串统计元音字母
    for (int i = 0; str[i] != '\0'; i++) {
        char c = tolower(str[i]); // 转换为小写，统一判断
        if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u') {
            count++;
        }
    }

    printf("元音字母的个数为: %d\n", count);
    return 0;
}

```

第八章

1. 单选题

- (1) B 解析：函数定义的语法为“返回类型 函数名 (参数列表){ 函数体 }”。
- (2) A 解析：在 C 语言中，void 是专门用于声明无返回值函数的返回类型，这类函数不需要使用 return 语句返回结果（或仅用 return; 表示结束）。
- (3) C 解析：递归函数必须包含基线条件（终止）和递归条件（自我调用），否则会无限递归。
- (4) B 解析：冒泡排序通过相邻元素比较交换，使最大元素逐轮“冒泡”到末尾。
- (5) C 解析：二维数组传参需指定第二维（列数），第一维（行数）可省略。
- (6) C 解析：函数参数可设置默认值（C99 及以上），并非必须传递所有参数。
- (7) B 解析：缺失基线条件会导致无限递归，耗尽栈内存引发栈溢出。
- (8) B 解析：选择排序每轮选择最小（或最大）元素，与未排序部分首位交换。
- (9) B 解析：计算平均值需数组首地址和长度，形参顺序不影响功能，但通常数组名在前。
- (10) C 解析：快速排序核心是选择基准元素，将数组分为“小于基准”和“大于基准”两部分。

2. 填空题

- (1) 函数名; 参数列表 解析: 函数定义的四要素为返回类型、函数名、参数列表和函数体。
- (2) 数组长度 解析: 数组名仅传递地址, 函数需通过额外参数获取长度避免越界。
- (3) 基线条件; 递归条件 解析: 基线条件终止递归, 递归条件实现自我调用。
- (4) 第二维 (列数); 地址偏移 解析: 编译器通过列数计算行内元素的地址偏移。
- (5) $n-1$; $n-1$ - 当前轮次 解析: 冒泡排序 n 个元素需 $n-1$ 轮, 每轮比较次数递减。
- (6) 类型; 顺序 解析: 实参与形参需类型匹配、顺序一致, 否则编译报错。
- (7) 已排序部分; 有序 解析: 插入排序通过逐步插入构建有序数组。
- (8) 可读性强; 易于维护 解析: 模块化编程的核心优势包括复用、可读性和可维护性。
- (9) 分区; 递归 解析: 快速排序通过分区操作拆分问题, 再递归解决子问题。
- (10) `return`; `void` 解析: `return` 语句返回值, `void` 表示函数无返回值。

3. 编程题

(1)

```
#include <stdio.h>

// 计算数组元素的和
int sumArray(int arr[], int len) {
    int sum = 0;
    for (int i = 0; i < len; i++) {
        sum += arr[i];
    }
    return sum;
}

int main() {
    int arr[5];
    printf("请输入 5 个整数: ");
    for (int i = 0; i < 5; i++) {
        scanf("%d", &arr[i]);
    }
    int total = sumArray(arr, 5);
    printf("数组元素的和为: %d\n", total);
    return 0;
}
```

(2)

```
#include <stdio.h>

// 查找数组的最大值
int findMax(int arr[], int len) {
    int max = arr[0];
    for (int i = 1; i < len; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}
```

```

// 查找数组的最小值
int findMin(int arr[], int len) {
    int min = arr[0];
    for (int i = 1; i < len; i++) {
        if (arr[i] < min) {
            min = arr[i];
        }
    }
    return min;
}

int main() {
    int arr[8];
    printf("请输入 8 个整数: ");
    for (int i = 0; i < 8; i++) {
        scanf("%d", &arr[i]);
    }
    int max = findMax(arr, 8);
    int min = findMin(arr, 8);
    printf("最大值: %d, 最小值: %d\n", max, min);
    return 0;
}

```

(3)

```

#include <stdio.h>

// 递归计算斐波那契数列第 n 项
int fibonacci(int n) {
    // 基线条件
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    // 递归条件
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
    int n;
    printf("请输入斐波那契数列的项数 n: ");
    scanf("%d", &n);
    int result = fibonacci(n);
    printf("斐波那契数列第%d 项为: %d\n", n, result);
    return 0;
}

```

第九章

1. 单选题

- (1) B 解析：预处理是编译过程的第一个阶段，在正式编译之前对源代码进行文本处理。
- (2) C 解析：无参宏定义的语法为**#define** 宏名 替换文本，无需等号或括号。
- (3) B 解析：带参宏会严格替换参数，**SQUARE(3+2)**会替换为**((3+2)*(3+2))**，括号确保运算优先级正确。
- (4) A 解析：**#ifndef** 宏名表示“如果宏未定义，则执行后续代码”，常用于头文件保护。
- (5) C 解析：**<>**优先从系统目录查找，**"**优先从当前目录查找，找不到再查系统目录。
- (6) B 解析：**__LINE__**是预定义宏，替换为当前代码的行号（整数）。
- (7) C 解析：**#undef** 宏名用于取消已定义的宏，使其在后续代码中不再生效。
- (8) B 解析：**#if** 后的表达式必须是常量表达式（预处理阶段可求值），不能包含变量。
- (9) B 解析：带参宏是文本替换，无调用开销；函数是代码块调用，有参数传递和返回过程。
- (10) B 解析：头文件保护通过判断宏是否定义，避免同一头文件被多次包含导致的函数 / 变量重复定义。

2. 填空题

- (1) **#**；编译之前 解析：预处理命令以 **#**开头，在编译前由预处理器处理。
- (2) **#define**；替换文本 解析：无参宏定义的格式为**#define** 宏名 替换文本。
- (3) 括号；运算符优先级 解析：括号可避免替换后因运算符优先级导致的逻辑错误。
- (4) **#endif**；**#if** 解析：**#endif**用于结束**#if**、**#ifdef**、**#ifndef**等条件块。
- (5) 系统；当前源文件所在 解析：两种文件包含的查找路径优先级不同。
- (6) **FILE**；**DATE** 解析：**__FILE__**表示当前文件名，**__DATE__**表示编译日期。
- (7) 取消；不再 解析：**#undef**取消宏定义，后续代码中该宏不再被替换。
- (8) 防止头文件被重复包含 解析：头文件保护的核心作用是避免重复定义错误。
- (9) 预处理；文本 解析：宏替换是预处理阶段的纯文本操作，不检查语法正确性。
- (10) 前面条件；代码块 解析：**#else**指定当前面条件不满足时执行的备选代码块。

3. 编程题

(1)

```
#include <stdio.h>
#define PI 3.14159 // 定义圆周率常量

int main() {
    float radius, perimeter, area;
    printf("请输入圆的半径: ");
    scanf("%f", &radius);

    perimeter = 2 * PI * radius; // 计算周长
    area = PI * radius * radius; // 计算面积

    printf("周长: %.2f, 面积: %.2f\n", perimeter, area);
    return 0;
}
```

(2)

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



```

#define ADD(a,b) ((a)+(b)) // 带参宏：两数之和
#define MUL(a,b) ((a)*(b)) // 带参宏：两数之积
#define DEBUG 1 // 定义 DEBUG 启用调试模式（注释此行关闭调试）

int main() {
    int x, y;
    printf("请输入两个整数: ");
    scanf("%d %d", &x, &y);

    // 计算和并输出
    int sum = ADD(x, y);
#ifdef DEBUG
    printf("%d + %d = %d\n", x, y, sum); // 调试模式输出过程
#else
    printf("和: %d\n", sum); // 非调试模式仅输出结果
#endif

    // 计算积并输出
    int product = MUL(x, y);
#ifdef DEBUG
    printf("%d * %d = %d\n", x, y, product); // 调试模式输出过程
#else
    printf("积: %d\n", product); // 非调试模式仅输出结果
#endif

    return 0;
}

```

(3)

■ 头文件 calc.h

```

#ifndef CALC_H // 头文件保护
#define CALC_H

// 函数声明
int add(int a, int b);
int mul(int a, int b);

#endif // CALC_H

```

■ 源文件 calc.c

```

#include "calc.h" // 包含头文件

// 函数实现：加法
int add(int a, int b) {
    return a + b;
}

// 函数实现：乘法
int mul(int a, int b) {
    return a * b;
}

```

(3) 主程序 main.c:

```
c
运行
#include <stdio.h>
#include "calc.h" // 包含自定义头文件

int main() {
    int a, b;
    printf("请输入两个整数: ");
    scanf("%d %d", &a, &b);

    printf("%d + %d = %d\n", a, b, add(a, b));
    printf("%d * %d = %d\n", a, b, mul(a, b));

    return 0;
}
```

(编译说明: 需将 calc.c 和 main.c 一起编译, 如 gcc main.c calc.c -o calc)

第十章

1. 单选题

(1) B 解析: 指针本质上是内存单元的地址, 通过地址可访问对应内存中的数据。

(2) B 解析: 指针变量定义格式为“数据类型 * 指针名”, int *p 表示定义指向 int 类型的指针 p。

(3) B 解析: p 存储 a 的地址, *p 通过地址访问 a 的值, 因此 *p = 20。

(4) B 解析: 在数据结构中, 链表节点通常通过结构体定义, 且通过指针连接节点 (如 struct Node *next)。访问结构体指针的成员时, C 语言规定需使用“->”运算符, 格式为“指针变量 -> 成员名”。

(5) A 解析: 字符串指针指向字符串首字符, 字符串常量不可修改, 结束标志为 '\0'。

(6) B 解析: 二级指针定义为“数据类型 ** 指针名”, int **p 表示指向 int 指针的指针。

(7) C 解析: p 指向数组首元素, p+3 指向第 4 个元素 (下标 3), 值为 7。

(8) B 解析: 函数指针存储函数入口地址, 可动态选择调用不同函数, 实现灵活编程。

(9) B 解析: 指针传递变量地址, 函数内通过解引用可修改外部变量, 实现双向数据传递。

(10) A 解析: 未初始化的指针可能指向随机地址, 导致野指针; 初始化和置空可避免。

2. 填空题

(1) 内存地址; 取地址 解析: 指针存储地址, & 用于获取变量地址。

(2) *; 变量 解析: *p 表示访问 p 所指向变量的值。

(3) 首; *(arr + i) 解析: 数组名是首地址, arr[i] 等价于指针偏移访问。

(4) 首字符 ('w'); '\0' 解析: 字符串指针指向首字符, 以 '\0' 结束。

(5) int; ** 解析: 二级指针定义格式为“int ** 指针名”。

(6) 参数 解析: 函数指针格式为“返回类型 (* 指针名)(参数列表)”。

(7) a 的地址 (&a); 15 解析: p 存地址, *p 访问 a 的值。

(8) 自增 (或 `p++`)：数据类型 解析：指针自增移动，步长由类型决定 (如 `int` 为 4 字节)。

(9) 字符串；参数 解析：`argv` 是字符串指针数组，存储命令行参数。

(10) 长度；计算 解析：指针偏移量需小于数组长度，长度通过 `sizeof` 计算。

3. 编程题

(1)

```
#include <stdio.h>

int main() {
    int num = 10; // 定义并初始化变量
    int *p = &num; // 指针 p 指向 num

    printf("修改前: %d\n", num);
    *p = 20; // 通过指针修改 num 的值
    printf("修改后: %d\n", num);

    return 0;
}
```

(2)

```
#include <stdio.h>

int main() {
    int arr[] = {2, 4, 6, 8, 10};
    int *p = arr; // 指针指向数组首元素
    int sum = 0;
    int len = sizeof(arr) / sizeof(arr[0]); // 计算数组长度

    printf("数组元素: ");
    for (int i = 0; i < len; i++) {
        printf("%d ", *(p + i)); // 指针偏移访问元素
        sum += *(p + i); // 累加元素值
    }
    printf("\n 总和: %d\n", sum);

    return 0;
}
```

(3)

```
#include <stdio.h>

// 函数：通过指针交换两个整数的值
void swap(int *a, int *b) {
    int temp = *a; // 临时存储 a 指向的值
    *a = *b;       // 将 b 指向的值赋给 a 指向的变量
    *b = temp;     // 将临时值赋给 b 指向的变量
}

int main() {
    int a, b;
```

```

printf("请输入两个整数: ");
scanf("%d %d", &a, &b);

printf("交换前: a=%d, b=%d\n", a, b);
swap(&a, &b); // 传递地址给指针参数
printf("交换后: a=%d, b=%d\n", a, b);

return 0;
}

```

第十一章

1. 单选题

- (1) C 解析: 结构体通过 `struct` 关键字声明, `union` 用于定义共用体, `enum` 用于定义枚举, `typedef` 用于给已有类型起别名。
- (2) B 解析: 访问结构体变量的成员需使用成员运算符 “.”, 格式为 “结构体变量名.成员名”; 而 “->” 用于通过结构体指针访问成员。
- (3) B 解析: 结构体为每个成员分配独立内存, 总内存是成员内存之和, 可同时存储所有成员; 共用体所有成员共用一块内存, 总内存等于最大成员的内存, 同一时间只能存储一个成员。二者成员访问方式相同 (变量用 “.”, 指针用 “->”)。
- (4) B 解析: 枚举值默认从 0 开始编号, 如 `enum Weekday {Monday, Tuesday, ...}` 中, `Monday` 默认值为 0, `Tuesday` 为 1, 以此类推。
- (5) B 解析: 链表中的节点在内存中的地址是非连续的, 而数组的内存空间是连续的。A、C、D 选项均符合链表的特性。
- (6) B 解析: 通过结构体指针访问成员时需使用 “->” 运算符, 如 `p->name` 等价于 `(*p).name`; “.” 用于结构体变量直接访问成员。
- (7) B 解析: 栈区由编译器自动分配释放, 存放局部变量、形参等; 堆区需通过 `malloc()` 和 `free()` 手动管理; 全局区和常量区在程序启动时分配, 结束时释放。
- (8) B 解析: `typedef` 关键字用于给已有类型起别名, 如 `typedef int Integer` 后, `Integer` 即成为 `int` 的别名, 并非定义新类型或分配内存。
- (9) C 解析: 全局变量存储在全局 / 静态区, 在程序启动时分配, 结束时自动释放, 作用域为整个程序 (非仅定义它的函数)。栈区存放局部变量, 无需手动释放全局变量。
- (10) B 解析: 共用体适合处理 “互斥数据” (不会同时出现的数据)。A、C 需同时存储多个成员, 适合用结构体; D 是链表节点的典型结构, 需用结构体。B 中奖学金和助学贷款不会同时存在, 符合共用体的应用场景。

2. 填空题

- (1) 不同
- (2) 定义结构体
- (3) 数据域; 指针域
- (4) 共用; 一
- (5) `enum`
- (6) `struct`
- (7) 代码区

- (8) malloc; free
- (9) 另一个结构体
- (10) 插入 / 删除效率

3. 编程题

(1)

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

// 定义图书结构体
struct Book {
    char title[50]; // 书名
    char author[30]; // 作者
    float price; // 价格
};

int main() {
    // 声明结构体变量并赋值
    struct Book b;
    strcpy(b.title, "C 程序设计");
    strcpy(b.author, "张三");
    b.price = 59.9;

    // 输出图书信息
    printf("图书信息: \n");
    printf("书名: %s\n", b.title);
    printf("作者: %s\n", b.author);
    printf("价格: %.2f 元\n", b.price);
    return 0;
}
```

(2)

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

// 定义学生结构体
struct Student {
    int id; // 学号
    char name[20]; // 姓名
    float scores[3]; // 三门课程成绩
    float avg; // 平均分
};

int main() {
    // 初始化结构体数组
    struct Student stus[3] = {
        {1001, "张三", {85, 92, 78}, 0},
        {1002, "李四", {90, 88, 95}, 0},
        {1003, "王五", {76, 80, 85}, 0}
    };
}
```

```

// 计算平均分
for (int i = 0; i < 3; i++) {
    stus[i].avg = (stus[i].scores[0] + stus[i].scores[1] + stus[i].scores[2]) / 3;
}

// 输出学生信息
printf("学号\t 姓名\t 语文\t 数学\t 英语\t 平均分\n");
for (int i = 0; i < 3; i++) {
    printf("%d\t%s\t%.1f\t%.1f\t%.1f\t%.1f\n",
        stus[i].id, stus[i].name,
        stus[i].scores[0], stus[i].scores[1], stus[i].scores[2],
        stus[i].avg);
}
return 0;
}

```

(3)

```

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

// 定义学生结构体（链表节点）
struct Student {
    int id;           // 学号
    char name[20];    // 姓名
    struct Student *next; // 指针域
};

// 创建新节点
struct Student* createNode(int id, char *name) {
    struct Student *newNode = (struct Student*)malloc(sizeof(struct Student));
    newNode->id = id;
    strcpy(newNode->name, name);
    newNode->next = NULL;
    return newNode;
}

// 在链表尾部添加节点
void appendNode(struct Student **head, int id, char *name) {
    struct Student *newNode = createNode(id, name);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct Student *current = *head;
    while (current->next != NULL) {
        current = current->next;
    }
    current->next = newNode;
}

```

```

}

// 遍历输出链表
void displayList(struct Student *head) {
    struct Student *current = head;
    printf("链表学生信息: \n");
    while (current != NULL) {
        printf("学号: %d, 姓名: %s\n", current->id, current->name);
        current = current->next;
    }
}

int main() {
    struct Student *head = NULL; // 初始化头指针
    // 创建链表（添加 3 个节点）
    appendNode(&head, 1001, "张三");
    appendNode(&head, 1002, "李四");
    appendNode(&head, 1003, "王五");
    // 遍历输出
    displayList(head);
    return 0;
}

```

第十二章

1. 单选题

- (1) B 解析：文本文件以 ASCII 码存储，可直接用记事本阅读；二进制文件存储原始二进制数据，通常不可直接阅读。A、C、D 描述错误。
- (2) B 解析：fopen 用于打开文件，fclose 用于关闭文件，fread 和 fwrite 用于二进制读写。
- (3) B 解析："r" 表示只读方式打开文本文件，文件必须已存在；"w" 为只写创建 / 清空；"a" 为追加；"rb" 为二进制只读。
- (4) C 解析：一个文件指针只能关联一个文件，多文件操作需定义多个指针。A、B、D 描述正确。
- (5) C 解析：fclose 关闭成功返回 0，失败返回非 0（通常为 EOF）。
- (6) B 解析：fputc 用于写入单个字符，fgetc 用于读取单个字符；fgets/fputs 用于字符串读写。
- (7) C 解析："rb+" 表示二进制读写模式；"r+"/"w+"/"a+" 为文本文件读写方式。
- (8) B 解析：fseek 起始点中，0 表示文件开头，1 表示当前位置，2 表示文件末尾。
- (9) B 解析：缓冲文件系统通过内存缓冲区批量读写数据，减少磁盘操作次数，优化效率。
- (10) C 解析：rewind 函数的作用是将文件位置标记重置到文件开头；fseek 需指定参数，ftell 获取位置，fclose 关闭文件。

2. 填空题

- (1) 文本 解析：文件按编码方式分为文本文件（ASCII 码）和二进制文件（原始二进制

数据)。

(2) FILE 解析: FILE 是 C 语言定义的文件类型, 文件指针需声明为 FILE *类型。

(3) NULL 解析: fopen 打开成功返回 FILE 结构体地址, 失败返回 NULL。

(4) "a" 解析: "a" 表示以追加方式打开文本文件, 写入操作从文件末尾开始。

(5) fgets; fputs 解析: fgets 从文件读取字符串, fputs 向文件写入字符串。

(6) fprintf; fscanf 解析: fprintf 按格式写入文件, fscanf 按格式读取文件, 与 printf/scanf 类似但操作对象为文件。

(7) fwrite; fread 解析: fwrite 用于二进制写入数据块, fread 用于二进制读取数据块。

(8) 2 解析: fseek 起始点中, 0 = 文件开头, 1 = 当前位置, 2 = 文件末尾。

(9) 缓冲区 (内存缓冲区) 解析: 缓冲文件系统通过缓冲区暂存数据, 减少磁盘 IO 次数。

(10) ftell 解析: ftell 返回当前文件位置标记相对于文件开头的字节数, 失败返回 -1L。

3. 编程题

(1)

```
#include <stdio.h>

int main() {
    FILE *fp;
    int num;

    // 写入文件
    fp = fopen("numbers.txt", "w");
    if (fp == NULL) {
        printf("文件打开失败! \n");
        return 1;
    }
    for (num = 1; num <= 10; num++) {
        fprintf(fp, "%d\n", num); // 每个数占一行
    }
    fclose(fp);

    // 读取文件并显示
    fp = fopen("numbers.txt", "r");
    if (fp == NULL) {
        printf("文件打开失败! \n");
        return 1;
    }
    printf("文件内容: \n");
    while (fscanf(fp, "%d", &num) != EOF) { // 循环读取直到文件结束
        printf("%d ", num);
    }
    fclose(fp);
    return 0;
}
```

(2)

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



```

// 定义图书结构体
struct Book {
    char title[50]; // 书名
    char author[30]; // 作者
    float price; // 价格
};

int main() {
    FILE *fp;
    struct Book books[3] = {
        {"C 程序设计", "张三", 59.9},
        {"数据结构", "李四", 69.5},
        {"操作系统", "王五", 79.0}
    };
    struct Book temp; // 临时存储读取的图书信息

    // 写入文件
    fp = fopen("books.txt", "w");
    if (fp == NULL) {
        printf("文件打开失败! \n");
        return 1;
    }
    for (int i = 0; i < 3; i++) {
        fprintf(fp, "%s %s %.1f\n", books[i].title, books[i].author, books[i].price);
    }
    fclose(fp);

    // 读取文件并显示
    fp = fopen("books.txt", "r");
    if (fp == NULL) {
        printf("文件打开失败! \n");
        return 1;
    }
    printf("图书信息: \n");
    printf("书名\t作者\t价格\n");
    while (fscanf(fp, "%s %s %f", temp.title, temp.author, &temp.price) != EOF) {
        printf("%s\t%s\t%.1f\n", temp.title, temp.author, temp.price);
    }
    fclose(fp);
    return 0;
}

```

(3)

```

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

// 定义学生结构体
struct Student {
    int id; // 学号
    char name[20]; // 姓名

```

```

        float score;    // 成绩
    };

int main() {
    FILE *fp;
    struct Student students[3] = {
        {101, "张三", 90.5},
        {102, "李四", 85.0},
        {103, "王五", 92.5}
    };
    struct Student temp;
    int i;

    // 写入二进制文件
    fp = fopen("students.dat", "wb");
    if (fp == NULL) {
        printf("文件打开失败! \n");
        return 1;
    }
    fwrite(students, sizeof(struct Student), 3, fp);
    fclose(fp);

    // 修改第 2 名学生（下标 1）的成绩
    fp = fopen("students.dat", "rb+"); // 二进制读写模式
    if (fp == NULL) {
        printf("文件打开失败! \n");
        return 1;
    }
    // 定位到第 2 名学生
    fseek(fp, 1 * sizeof(struct Student), 0);
    // 读取原数据
    fread(&temp, sizeof(struct Student), 1, fp);
    // 修改成绩
    temp.score = 98.5;
    // 重新定位并写入修改后的数据
    fseek(fp, 1 * sizeof(struct Student), 0);
    fwrite(&temp, sizeof(struct Student), 1, fp);
    fclose(fp);

    // 验证修改结果
    fp = fopen("students.dat", "rb");
    if (fp == NULL) {
        printf("文件打开失败! \n");
        return 1;
    }
    printf("修改后的学生信息: \n");
    printf("学号\t 姓名\t 成绩\n");
    for (i = 0; i < 3; i++) {
        fread(&temp, sizeof(struct Student), 1, fp);

```

```

        printf("%d\t%s\t%.1f\n", temp.id, temp.name, temp.score);
    }
    fclose(fp);
    return 0;
}

```

第十三章

1. 单选题

(1) C 解析：线性表可采用顺序存储（数组）和链式存储（链表），A 错误；顺序存储插入需移动元素，B 错误；链式存储插入删除仅需修改指针，效率高，C 正确；线性表元素前后驱关系唯一，D 错误。

(2) B 解析：栈遵循“后进先出（LIFO）”原则，A 是队列特性，C 是数组特性，D 错误。

(3) B 解析：队列仅允许在队尾（rear）插入（入队），在队头（front）删除（出队）。

(4) B 解析：循环队列队满条件为 $(rear + 1) \% MAX_SIZE == front$ ，避免与队空条件 $front == rear$ 混淆。

(5) D 解析：邻接矩阵中，无直接边的顶点对通常用极大值（如 INF）标记，0 表示自身到自身的距离。

(6) C

解析：

选项 A 错误：通过计算 next 数组避免主串指针回溯、时间复杂度为 $O(n+m)$ 是 KMP 算法的特点，Brute-Force 算法无 next 数组，最坏时间复杂度为 $O(n \times m)$ ；

选项 B 错误：无需额外预处理、“逐位比较、回溯重试”、空间复杂度 $O(1)$ 是 Brute-Force 算法的特点，KMP 算法需预处理计算 next 数组，空间复杂度为 $O(m)$ ；

选项 C 正确：该场景下 Brute-Force 算法需频繁回溯主串指针，导致重复比较，而 KMP 算法通过 next 数组调整模式串指针，无主串指针回溯，比较次数更少；

选项 D 错误：两种算法的匹配结果一致（均能正确定位模式串在主串中的位置或判断匹配失败），且无“KMP 仅处理长串、Brute-Force 仅处理短串”的限制，仅在效率适用场景上有差异。

(7) B 解析：逆波兰表达式计算通过栈存储操作数，遇到运算符时弹出操作数计算，核心结构为栈。

(8) B 解析：银行排队遵循“先进先出”原则，队列是最佳数据结构，栈适合“后进先出”场景。

(9) A 解析：深度优先搜索（DFS）通过递归或栈实现，优先深入路径；广度优先搜索（BFS）用队列实现。

(10) B 解析：空间复杂度衡量算法所需额外内存随输入规模的增长趋势，A 是时间复杂度的定义。

2. 填空题

(1) 顺序

解析：线性表的两种基本存储结构为顺序存储（连续内存）和链式存储（指针连接）。

(2) 入栈；出栈

解析：栈的插入操作为入栈（push），删除操作为出栈（pop）。

（3）队头；队尾

解析：队列中 front 指向队头元素，rear 指向队尾元素的下一个位置（空闲位置）。

（4）边；无向图

解析：图的定义为 $G=(V,E)$ ，V 是顶点集，E 是边集；按边的方向性分为有向图和无向图。

（5）大 O（O）

解析：时间复杂度用大 O 符号表示，如 $O(n)$ 、 $O(n^2)$ 。

（6）-1；满

解析：顺序栈初始化时 top = -1（空栈），top = MAX_SIZE - 1 时栈满。

（7）取余（%）

解析：循环队列通过 $rear = (rear + 1) \% MAX_SIZE$ 实现指针绕回，复用空闲空间。

（8）邻接表；邻接表

解析：图的存储方式包括邻接矩阵（适合稠密图）和邻接表（适合稀疏图，节省空间）。

（9）临时

解析：空间复杂度关注算法执行过程中所需的额外临时空间，如局部变量、栈空间等。

（10）单源；带权（或加权）

解析：迪杰斯特拉算法求解从单个起点到其他所有顶点的最短路径，适用于边权为非负的带权图。

3. 编程题

（1）

```
#include <stdio.h>
#define MAX_SIZE 100

typedef int ElemType;
typedef struct {
    ElemType data[MAX_SIZE];
    int length;
} SeqList;

// 初始化顺序表
void initList(SeqList *L) {
    L->length = 0;
}

// 向表尾添加元素
int appendElem(SeqList *L, ElemType elem) {
    if (L->length >= MAX_SIZE) return 0;
    L->data[L->length++] = elem;
    return 1;
}

// 在指定位置插入元素（位置从 1 开始）
int insertElem(SeqList *L, int pos, ElemType elem) {
    if (pos < 1 || pos > L->length + 1 || L->length >= MAX_SIZE) return 0;
    for (int i = L->length; i >= pos; i--) {
        L->data[i] = L->data[i - 1];
    }
}
```

```

        L->data[pos - 1] = elem;
        L->length++;
        return 1;
    }

// 遍历打印元素
void printList(SeqList *L) {
    for (int i = 0; i < L->length; i++) {
        printf("%d ", L->data[i]);
    }
    printf("\n");
}

int main() {
    SeqList L;
    initList(&L);
    // 添加初始元素 1-5
    for (int i = 1; i <= 5; i++) {
        appendElem(&L, i);
    }
    // 在第 3 个位置插入 99
    insertElem(&L, 3, 99);
    // 输出结果
    printf("顺序表元素: ");
    printList(&L); // 预期输出: 1 2 99 3 4 5
    return 0;
}

```

(2)

```

#include <stdio.h>
#include <string.h>
#define MAX_STACK_SIZE 100

typedef struct {
    char data[MAX_STACK_SIZE];
    int top;
} Stack;

void initStack(Stack *s) {
    s->top = -1;
}

int push(Stack *s, char c) {
    if (s->top >= MAX_STACK_SIZE - 1) return 0;
    s->data[++s->top] = c;
    return 1;
}

int pop(Stack *s, char *c) {
    if (s->top == -1) return 0;

```

```

        *c = s->data[s->top--];
        return 1;
    }

    int isEmpty(Stack *s) {
        return s->top == -1;
    }

    // 检查括号是否匹配
    int isMatch(char left, char right) {
        return (left == '(' && right == ')') ||
            (left == '{' && right == '}') ||
            (left == '[' && right == ']');
    }

    int checkBrackets(char *str) {
        Stack s;
        initStack(&s);
        for (int i = 0; i < strlen(str); i++) {
            char c = str[i];
            if (c == '(' || c == '{' || c == '[') {
                push(&s, c); // 左括号入栈
            } else if (c == ')' || c == '}' || c == ']') {
                if (isEmpty(&s)) return 0; // 右括号多余
                char top;
                pop(&s, &top);
                if (!isMatch(top, c)) return 0; // 不匹配
            }
        }
        return isEmpty(&s); // 栈空则所有括号匹配
    }

    int main() {
        char str[100];
        printf("请输入带括号的字符串: ");
        scanf("%s", str);
        if (checkBrackets(str)) {
            printf("匹配成功\n");
        } else {
            printf("匹配失败\n");
        }
        return 0;
    }

```

(3)

```

#include <stdio.h>
#define MAX_QUEUE_SIZE 10

// 顾客结构体
typedef struct {

```

```

        int id;           // 顾客 ID
        int arrivalTime; // 到达时间（秒）
        int goodsCount;  // 商品数量
    } Customer;

// 循环队列结构体
typedef struct {
    Customer data[MAX_QUEUE_SIZE];
    int front;
    int rear;
    int size;
} Queue;

// 初始化队列
void initQueue(Queue *q) {
    q->front = q->rear = q->size = 0;
}

// 入队
int enqueue(Queue *q, Customer c) {
    if (q->size >= MAX_QUEUE_SIZE) return 0;
    q->data[q->rear] = c;
    q->rear = (q->rear + 1) % MAX_QUEUE_SIZE;
    q->size++;
    return 1;
}

// 出队
int dequeue(Queue *q, Customer *c) {
    if (q->size == 0) return 0;
    *c = q->data[q->front];
    q->front = (q->front + 1) % MAX_QUEUE_SIZE;
    q->size--;
    return 1;
}

int main() {
    Queue q;
    initQueue(&q);
    // 初始化 5 名顾客（ID、到达时间、商品数量）
    Customer customers[5] = {
        {1, 0, 3}, // 服务时间 3×2=6 秒
        {2, 2, 2}, // 服务时间 2×2=4 秒
        {3, 5, 5}, // 服务时间 5×2=10 秒
        {4, 8, 1}, // 服务时间 1×2=2 秒
        {5, 10, 4} // 服务时间 4×2=8 秒
    };
    // 顾客入队
    for (int i = 0; i < 5; i++) {

```

```
        enqueue(&q, customers[i]);
    }
    // 模拟服务过程
    int currentTime = 0;
    Customer c;
    printf("收银台服务记录: \n");
    printf("顾客 ID\t 到达时间\t 开始时间\t 结束时间\n");
    while (dequeue(&q, &c)) {
        // 若顾客到达时间晚于当前时间，更新当前时间
        if (c.arrivalTime > currentTime) {
            currentTime = c.arrivalTime;
        }
        int serviceTime = c.goodsCount * 2;
        int endTime = currentTime + serviceTime;
        // 输出服务信息
        printf("%d\t%d\t%d\t%d\n", c.id, c.arrivalTime, currentTime, endTime);
        currentTime = endTime;
    }
    return 0;
}
```