C 语言高级特性手册

January 9, 2026

1 动态内存分配

1.1 动态内存分配概述

在 C 语言中，动态内存分配是在程序运行时分配内存的方法，与静态内存分配（编译时确定大小）相对。

1.1.1 为什么需要动态内存分配

- **未知数据大小**：程序运行时才能确定需要多少内存
- **内存高效利用**：按需分配，避免浪费
- **数据结构灵活性**：链表、树等动态数据结构的需要
- **生命周期控制**：手动控制内存的分配和释放

1.2 内存分配函数

1.2.1 malloc 函数

```c
void* malloc(size_t size);
```

Listing 1: malloc 函数原型

- **功能**：分配指定字节数的内存块
- **参数**：需要分配的字节数
- **返回值**：成功返回指向分配内存的指针，失败返回 NULL
- **特点**：分配的内存不会初始化，内容是随机的

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

int main() {
    // 分配10个整数的空间
    int *arr = (int*)malloc(10 * sizeof(int));

    if(arr == NULL) {
        printf("内存分配失败!\n");
        return 1;
    }

    // 使用分配的内存
    for(int i = 0; i < 10; i++) {
        arr[i] = i * 10;
    }

    // 输出
    for(int i = 0; i < 10; i++) {
```

```
        printf("arr[%d] = %d\n", i, arr[i]);
    }

    // 释放内存
    free(arr);

    return 0;
}
```

Listing 2: malloc 使用示例

1.2.2 calloc 函数

```
void* calloc(size_t num, size_t size);
```

Listing 3: calloc 函数原型

- **功能**：分配指定数量、指定大小的内存块
- **参数**：
  - num：元素数量
  - size：每个元素的大小
- **返回值**：成功返回指向分配内存的指针，失败返回 NULL
- **特点**：分配的内存会初始化为 0

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

int main() {
    // 分配10个整数的空间，并初始化为0
    int *arr = (int*)calloc(10, sizeof(int));

    if(arr == NULL) {
        printf("内存分配失败!\n");
        return 1;
    }

    printf("calloc分配的内存已初始化为0:\n");
    for(int i = 0; i < 10; i++) {
        printf("arr[%d] = %d\n", i, arr[i]);
    }

    free(arr);
    return 0;
}
```

Listing 4: calloc 使用示例

1.2.3 realloc 函数

```c
void* realloc(void* ptr, size_t new_size);
```

Listing 5: realloc 函数原型

- **功能**：重新分配内存块的大小
- **参数**：
  - `ptr`：指向之前分配的内存块的指针
  - `new_size`：新的内存块大小
- **返回值**：成功返回指向新内存块的指针，失败返回 NULL
- **特点**：
  - 如果新大小小于原大小，多余部分被丢弃
  - 如果新大小大于原大小，新增部分不会被初始化
  - 可能返回一个新的指针地址

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

int main() {
    // 初始分配5个整数的空间
    int *arr = (int*)malloc(5 * sizeof(int));

    if(arr == NULL) {
        printf("内存分配失败!\n");
        return 1;
    }

    // 初始化
    for(int i = 0; i < 5; i++) {
        arr[i] = i + 1;
    }

    // 扩展到10个整数
    int *new_arr = (int*)realloc(arr, 10 * sizeof(int));

    if(new_arr == NULL) {
        printf("内存重新分配失败!\n");
        free(arr);  // 释放原始内存
        return 1;
    }

    arr = new_arr;  // 使用新的指针

    // 初始化新增的部分
    for(int i = 5; i < 10; i++) {
```

```
        arr[i] = i + 1;
    }

    // 输出所有元素
    for(int i = 0; i < 10; i++) {
        printf("arr[%d] = %d\n", i, arr[i]);
    }

    free(arr);
    return 0;
}
```

Listing 6: realloc 使用示例

1.2.4 free 函数

```
void free(void* ptr);
```

Listing 7: free 函数原型

- **功能**：释放之前分配的内存
- **参数**：指向要释放的内存块的指针
- **注意事项**：
  - 只能释放由 malloc、calloc、realloc 分配的内存
  - 不能重复释放同一块内存
  - 释放后应将指针设为 NULL，避免悬空指针

1.3 malloc vs calloc vs realloc 对比

malloc	calloc	realloc
分配未初始化内存 参数：字节数	分配并初始化为 0 参数：数量 × 大小	调整已分配内存大小 参数：原指针 + 新大小
内存内容随机 效率较高	内存内容全为 0 效率较低（需初始化）	新内存内容可能改变 可能需要内存复制

Table 1: 内存分配函数对比

1.4 动态内存分配的最佳实践

1.4.1 正确的内存管理流程

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

int main() {
    // 1. 分配内存
    char *str = (char*)malloc(100 * sizeof(char));

    // 2. 检查是否分配成功
    if(str == NULL) {
        fprintf(stderr, "内存分配失败!\n");
        return 1;
    }

    // 3. 使用内存
    strcpy(str, "Hello, Dynamic Memory!");
    printf("%s\n", str);

    // 4. 释放内存
    free(str);

    // 5. 将指针设为NULL（避免悬空指针）
    str = NULL;

    return 0;
}
```

Listing 8: 内存管理最佳实践

1.4.2 常见错误与避免方法

```c
// 错误1：使用未初始化的指针
int *ptr;
*ptr = 10;  // 错误：ptr未指向有效内存

// 正确做法：
int *ptr = (int*)malloc(sizeof(int));
if(ptr != NULL) {
    *ptr = 10;
}

// 错误2：内存泄漏
void func() {
    int *ptr = malloc(100 * sizeof(int));
    // 使用ptr...
    // 忘记 free(ptr);  // 内存泄漏!
}

// 正确做法：
void func() {
```

```c
    int *ptr = malloc(100 * sizeof(int));
    if(ptr != NULL) {
        // 使用ptr...
        free(ptr);
    }
}

// 错误3: 重复释放
int *ptr = malloc(sizeof(int));
free(ptr);
free(ptr);  // 错误: 重复释放!

// 正确做法:
int *ptr = malloc(sizeof(int));
if(ptr != NULL) {
    free(ptr);
    ptr = NULL;  // 设为NULL
}
// 再次释放时检查
if(ptr != NULL) {
    free(ptr);
}

// 错误4: 使用已释放的内存
int *ptr = malloc(sizeof(int));
free(ptr);
*ptr = 20;  // 错误: ptr指向的内存已释放!

// 正确做法:
int *ptr = malloc(sizeof(int));
if(ptr != NULL) {
    *ptr = 20;
    // 使用...
    free(ptr);
    ptr = NULL;  // 设为NULL, 避免误用
}
```

Listing 9: 常见内存错误示例

2 结构体（Struct）

2.1 结构体的定义与声明

2.1.1 什么是结构体

结构体是 C 语言中一种用户自定义的数据类型，它允许我们将不同类型的数据组合成一个整体。

2.1.2 定义结构体

```c
// 定义结构体类型
struct Student {
    char name[50];      // 姓名
    int age;            // 年龄
    float score;        // 成绩
    char gender;        // 性别
};  // 注意：这里需要分号

// 定义结构体的同时声明变量
struct Point {
    int x;
    int y;
} p1, p2;  // 同时声明两个Point类型的变量

// 使用typedef创建别名
typedef struct Date {
    int year;
    int month;
    int day;
} Date;  // 现在可以直接使用Date作为类型名
```

Listing 10: 结构体定义

2.1.3 结构体变量声明

```c
// 方法1：先定义类型，再声明变量
struct Student stu1, stu2;

// 方法2：定义类型的同时声明变量
struct Employee {
    int id;
    char name[50];
    float salary;
} emp1, emp2;

// 方法3：使用typedef别名
Date today, tomorrow;
```

Listing 11: 结构体变量声明

2.2 结构体的初始化与访问

2.2.1 结构体初始化

```c
// 方法1：声明时初始化
struct Student stu1 = {"张三", 20, 85.5, 'M'};

// 方法2：部分初始化（未指定的成员自动初始化为0或空）
struct Student stu2 = {"李四"};  // age=0, score=0.0, gender='\0'

```

```c
// 方法3: 指定成员初始化 (C99标准)
struct Student stu3 = {
    .name = "王五",
    .age = 22,
    .score = 90.5
};

// 方法4: 数组初始化
struct Student class[3] = {
    {"张三", 20, 85.5, 'M'},
    {"李四", 21, 78.0, 'F'},
    {"王五", 19, 92.0, 'M'}
};
```

Listing 12: 结构体初始化

2.2.2 结构体成员访问

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

struct Student {
    char name[50];
    int age;
    float score;
};

int main() {
    struct Student stu;

    // 使用点运算符(.)访问结构体成员
    strcpy(stu.name, "张三");
    stu.age = 20;
    stu.score = 85.5;

    // 读取结构体成员
    printf("姓名: %s\n", stu.name);
    printf("年龄: %d\n", stu.age);
    printf("成绩: %.1f\n", stu.score);

    // 结构体数组
    struct Student class[3];
    for(int i = 0; i < 3; i++) {
        printf("请输入第%d个学生的信息:\n", i+1);
        printf("姓名: ");
        scanf("%s", class[i].name);
        printf("年龄: ");
        scanf("%d", &class[i].age);
        printf("成绩: ");
        scanf("%f", &class[i].score);
    }
```

```
34
35      return 0;
36  }
```

Listing 13: 结构体成员访问

2.3 结构体指针

2.3.1 指向结构体的指针

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  typedef struct Student {
6      char name[50];
7      int age;
8      float score;
9  } Student;
10
11 int main() {
12     Student stu = {"张三", 20, 85.5};
13     Student *p = &stu;  // 指向结构体的指针
14
15     // 通过指针访问结构体成员（两种方法）
16     printf("方法1 - 使用(*p).成员: %s\n", (*p).name);
17     printf("方法2 - 使用p->成员: %d\n", p->age);
18
19     // 动态分配结构体内存
20     Student *p2 = (Student*)malloc(sizeof(Student));
21     if(p2 != NULL) {
22         strcpy(p2->name, "李四");
23         p2->age = 21;
24         p2->score = 78.0;
25
26         printf("动态分配: %s, %d, %.1f\n",
27                p2->name, p2->age, p2->score);
28
29         free(p2);  // 释放内存
30     }
31
32     return 0;
33 }
```

Listing 14: 结构体指针

2.3.2 结构体作为函数参数

```
1  #include <stdio.h>
2  #include <string.h>
3
```

```c
typedef struct {
    char name[50];
    int age;
} Person;

// 值传递（复制整个结构体）
void printPersonByValue(Person p) {
    printf("姓名：%s，年龄：%d\n", p.name, p.age);
}

// 指针传递（更高效）
void printPersonByPointer(const Person *p) {
    printf("姓名：%s，年龄：%d\n", p->name, p->age);
}

// 修改结构体内容
void updatePerson(Person *p, const char *name, int age) {
    strcpy(p->name, name);
    p->age = age;
}

int main() {
    Person p1 = {"张三", 20};

    printPersonByValue(p1);     // 值传递
    printPersonByPointer(&p1);  // 指针传递

    updatePerson(&p1, "李四", 25);
    printPersonByPointer(&p1);

    return 0;
}
```

Listing 15: 结构体作为函数参数

2.4 结构体嵌套

```c
#include <stdio.h>

// 定义日期结构体
typedef struct {
    int year;
    int month;
    int day;
} Date;

// 定义学生结构体，嵌套日期结构体
typedef struct {
    char name[50];
    Date birthday;  // 嵌套结构体
    float score;
```

```c
} Student;

// 定义班级结构体，包含学生数组
typedef struct {
    Student students[50];
    int count;
    char className[20];
} Class;

int main() {
    // 创建学生
    Student stu = {
        "张三",
        {2000, 5, 15},  // 初始化嵌套结构体
        85.5
    };

    printf("学生: %s\n", stu.name);
    printf("生日: %d年%d月%d日\n",
           stu.birthday.year,
           stu.birthday.month,
           stu.birthday.day);

    // 创建班级
    Class csClass;
    strcpy(csClass.className, "计算机1班");
    csClass.count = 0;

    // 添加学生
    strcpy(csClass.students[0].name, "李四");
    csClass.students[0].birthday.year = 2001;
    csClass.students[0].birthday.month = 8;
    csClass.students[0].birthday.day = 20;
    csClass.students[0].score = 90.0;
    csClass.count++;

    return 0;
}
```

Listing 16: 结构体嵌套

2.5 结构体大小与对齐

```c
#include <stdio.h>

// 结构体大小受内存对齐影响
struct Example1 {
    char a;     // 1字节
    int b;      // 4字节
    char c;     // 1字节
};  // 总大小可能是12字节（取决于编译器对齐）
```

```c

struct Example2 {
    int b;      // 4字节
    char a;     // 1字节
    char c;     // 1字节
};  // 总大小可能是8字节（内存对齐后）

// 使用#pragma pack改变对齐方式
#pragma pack(push, 1)  // 设置为1字节对齐
struct PackedStruct {
    char a;
    int b;
    char c;
};  // 总大小为6字节
#pragma pack(pop)      // 恢复默认对齐

int main() {
    printf("Example1大小: %lu字节\n", sizeof(struct Example1));
    printf("Example2大小: %lu字节\n", sizeof(struct Example2));
    printf("PackedStruct大小: %lu字节\n", sizeof(struct PackedStruct)
        );

    return 0;
}
```

Listing 17: 结构体大小与内存对齐

3 链表（Linked List）

3.1 链表的基本概念

3.1.1 什么是链表

链表是一种动态数据结构，由一系列节点组成，每个节点包含数据和指向下一个节点的指针。

```
// 链表节点结构
+-------------+-------------+
|    数据     |    指针     |
+-------------+-------------+
      |             |
   数据域        指向下一个节点
```

Figure 1: 链表节点结构示意图

3.1.2 链表与数组的比较

3.2 单向链表的实现

3.2.1 链表节点定义

链表	数组
动态分配内存，大小可变	静态分配内存，大小固定
插入删除效率高（O(1)）	插入删除效率低（O(n)）
内存不连续	内存连续
访问元素效率低（O(n)）	随机访问效率高（O(1)）
需要额外空间存储指针	不需要额外指针空间

Table 2: 链表与数组比较

```c
typedef struct Node {
    int data;           // 数据域
    struct Node *next;  // 指针域，指向下一个节点
} Node;
```

Listing 18: 链表节点定义

3.2.2 链表基本操作

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

// 链表节点定义
typedef struct Node {
    int data;
    struct Node *next;
} Node;

// 创建新节点
Node* createNode(int data) {
    Node *newNode = (Node*)malloc(sizeof(Node));
    if(newNode == NULL) {
        printf("内存分配失败!\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// 在链表头部插入节点
Node* insertAtHead(Node *head, int data) {
    Node *newNode = createNode(data);
    newNode->next = head;
    return newNode;  // 新节点成为新的头节点
}

// 在链表尾部插入节点
Node* insertAtTail(Node *head, int data) {
    Node *newNode = createNode(data);
```

```
32
33      if(head == NULL) {
34          return newNode;
35      }
36
37      Node *current = head;
38      while(current->next != NULL) {
39          current = current->next;
40      }
41      current->next = newNode;
42
43      return head;
44  }
45
46  // 查找节点
47  Node* findNode(Node *head, int data) {
48      Node *current = head;
49      while(current != NULL) {
50          if(current->data == data) {
51              return current;
52          }
53          current = current->next;
54      }
55      return NULL;
56  }
57
58  // 删除节点
59  Node* deleteNode(Node *head, int data) {
60      if(head == NULL) {
61          return NULL;
62      }
63
64      // 如果要删除的是头节点
65      if(head->data == data) {
66          Node *temp = head;
67          head = head->next;
68          free(temp);
69          return head;
70      }
71
72      // 查找要删除的节点
73      Node *current = head;
74      while(current->next != NULL && current->next->data != data) {
75          current = current->next;
76      }
77
78      if(current->next != NULL) {
79          Node *temp = current->next;
80          current->next = current->next->next;
81          free(temp);
82      }
```

```c

    return head;
}

// 遍历链表
void printList(Node *head) {
    Node *current = head;
    printf("链表: ");
    while(current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}

// 获取链表长度
int getLength(Node *head) {
    int length = 0;
    Node *current = head;
    while(current != NULL) {
        length++;
        current = current->next;
    }
    return length;
}

// 释放链表内存
void freeList(Node *head) {
    Node *current = head;
    while(current != NULL) {
        Node *temp = current;
        current = current->next;
        free(temp);
    }
}

int main() {
    Node *head = NULL;

    // 插入测试
    head = insertAtHead(head, 10);
    head = insertAtHead(head, 20);
    head = insertAtTail(head, 30);
    head = insertAtTail(head, 40);

    printList(head);
    printf("链表长度: %d\n", getLength(head));

    // 查找测试
    Node *found = findNode(head, 30);
    if(found != NULL) {
```

```
        printf("找到节点: %d\n", found->data);
    }

    // 删除测试
    head = deleteNode(head, 20);
    printList(head);

    // 释放内存
    freeList(head);

    return 0;
}
```

Listing 19: 链表基本操作

3.2.3 有序链表插入

```
// 在有序链表中插入节点（保持升序）
Node* insertInOrder(Node *head, int data) {
    Node *newNode = createNode(data);

    // 如果链表为空或新节点应插入头部
    if(head == NULL || data < head->data) {
        newNode->next = head;
        return newNode;
    }

    // 查找插入位置
    Node *current = head;
    while(current->next != NULL && current->next->data < data) {
        current = current->next;
    }

    // 插入节点
    newNode->next = current->next;
    current->next = newNode;

    return head;
}
```

Listing 20: 有序链表插入

3.3 双向链表

3.3.1 双向链表节点定义

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

// 双向链表节点
typedef struct DNode {
```

```c
    int data;
    struct DNode *prev;  // 指向前一个节点
    struct DNode *next;  // 指向后一个节点
} DNode;

// 创建双向链表节点
DNode* createDNode(int data) {
    DNode *newNode = (DNode*)malloc(sizeof(DNode));
    if(newNode == NULL) {
        printf("内存分配失败!\n");
        exit(1);
    }
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}

// 在双向链表尾部插入
DNode* insertAtTail(DNode *head, int data) {
    DNode *newNode = createDNode(data);

    if(head == NULL) {
        return newNode;
    }

    DNode *current = head;
    while(current->next != NULL) {
        current = current->next;
    }

    current->next = newNode;
    newNode->prev = current;

    return head;
}

// 遍历双向链表（向前）
void printForward(DNode *head) {
    DNode *current = head;
    printf("向前遍历: ");
    while(current != NULL) {
        printf("%d <-> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}

// 遍历双向链表（向后）
void printBackward(DNode *tail) {
    DNode *current = tail;
```

```c
    printf("向后遍历: ");
    while(current != NULL) {
        printf("%d <-> ", current->data);
        current = current->prev;
    }
    printf("NULL\n");
}

// 查找尾节点
DNode* findTail(DNode *head) {
    if(head == NULL) return NULL;

    DNode *current = head;
    while(current->next != NULL) {
        current = current->next;
    }
    return current;
}

int main() {
    DNode *head = NULL;

    head = insertAtTail(head, 10);
    head = insertAtTail(head, 20);
    head = insertAtTail(head, 30);

    printForward(head);

    DNode *tail = findTail(head);
    printBackward(tail);

    return 0;
}
```

Listing 21: 双向链表

3.4 循环链表

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

// 循环链表节点
typedef struct CNode {
    int data;
    struct CNode *next;
} CNode;

// 创建循环链表节点
CNode* createCNode(int data) {
    CNode *newNode = (CNode*)malloc(sizeof(CNode));
    if(newNode == NULL) {
```

```
        printf("内存分配失败!\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = newNode;  // 指向自己，形成循环
    return newNode;
}

// 在循环链表尾部插入
CNode* insertAtTail(CNode *head, int data) {
    CNode *newNode = createCNode(data);

    if(head == NULL) {
        return newNode;
    }

    CNode *current = head;
    while(current->next != head) {
        current = current->next;
    }

    current->next = newNode;
    newNode->next = head;

    return head;
}

// 遍历循环链表
void printCircularList(CNode *head) {
    if(head == NULL) {
        printf("空链表\n");
        return;
    }

    CNode *current = head;
    printf("循环链表: ");
    do {
        printf("%d -> ", current->data);
        current = current->next;
    } while(current != head);
    printf("(回到头部)\n");
}

int main() {
    CNode *head = NULL;

    head = insertAtTail(head, 10);
    head = insertAtTail(head, 20);
    head = insertAtTail(head, 30);

    printCircularList(head);
```

```
65
66      return 0;
67  }
```

Listing 22: 循环链表

4 文件指针（File Pointer）

4.1 文件操作基础

4.1.1 文件指针概念

```c
#include <stdio.h>

int main() {
    // 声明文件指针
    FILE *fp;

    // 打开文件
    fp = fopen("test.txt", "w");  // 以写入模式打开

    if(fp == NULL) {
        printf("无法打开文件!\n");
        return 1;
    }

    // 写入数据
    fprintf(fp, "Hello, World!\n");
    fprintf(fp, "这是C语言文件操作示例\n");

    // 关闭文件
    fclose(fp);
    printf("文件写入成功!\n");

    return 0;
}
```

Listing 23: 文件指针基础

4.1.2 文件打开模式

4.2 文件读写操作

4.2.1 字符读写

```c
#include <stdio.h>

int main() {
    FILE *fp;
    char ch;
```

模式	描述
"r"	只读方式打开文本文件
"w"	只写方式创建/清空文本文件
"a"	追加方式打开文本文件
"rb"	只读方式打开二进制文件
"wb"	只写方式创建/清空二进制文件
"ab"	追加方式打开二进制文件
"r+"	读写方式打开文本文件
"w+"	读写方式创建/清空文本文件
"a+"	读写方式打开文本文件（追加）
"rb+"	读写方式打开二进制文件
"wb+"	读写方式创建/清空二进制文件
"ab+"	读写方式打开二进制文件（追加）

Table 3: 文件打开模式

```
6
7       // 写入字符
8       fp = fopen("char.txt", "w");
9       if(fp != NULL) {
10          fputc('A', fp);
11          fputc('B', fp);
12          fputc('C', fp);
13          fclose(fp);
14      }
15
16      // 读取字符
17      fp = fopen("char.txt", "r");
18      if(fp != NULL) {
19          printf("文件内容: ");
20          while((ch = fgetc(fp)) != EOF) {
21              printf("%c ", ch);
22          }
23          fclose(fp);
24          printf("\n");
25      }
26
27      return 0;
28  }
```

Listing 24: 字符读写

4.2.2 字符串读写

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main() {
5      FILE *fp;
```

```
    char buffer[100];

    // 写入字符串
    fp = fopen("string.txt", "w");
    if(fp != NULL) {
        fputs("第一行\n", fp);
        fputs("第二行\n", fp);
        fputs("第三行\n", fp);
        fclose(fp);
    }

    // 读取字符串
    fp = fopen("string.txt", "r");
    if(fp != NULL) {
        printf("文件内容:\n");
        while(fgets(buffer, sizeof(buffer), fp) != NULL) {
            printf("%s", buffer);
        }
        fclose(fp);
    }

    return 0;
}
```

Listing 25: 字符串读写

4.2.3 格式化读写

```
#include <stdio.h>

typedef struct {
    char name[50];
    int age;
    float score;
} Student;

int main() {
    FILE *fp;
    Student stu1 = {"张三", 20, 85.5};
    Student stu2;

    // 格式化写入
    fp = fopen("student.txt", "w");
    if(fp != NULL) {
        fprintf(fp, "%s %d %.1f\n", stu1.name, stu1.age, stu1.score);
        fclose(fp);
    }

    // 格式化读取
    fp = fopen("student.txt", "r");
    if(fp != NULL) {
```

```
        fscanf(fp, "%s %d %f", stu2.name, &stu2.age, &stu2.score);
        printf("读取: %s, %d, %.1f\n", stu2.name, stu2.age, stu2.
            score);
        fclose(fp);
    }

    return 0;
}
```

Listing 26: 格式化读写

4.3 二进制文件操作

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

typedef struct {
    char name[50];
    int age;
    float score;
} Student;

int main() {
    FILE *fp;
    Student stu1 = {"张三", 20, 85.5};
    Student stu2;

    // 二进制写入
    fp = fopen("student.dat", "wb");
    if(fp != NULL) {
        fwrite(&stu1, sizeof(Student), 1, fp);
        fclose(fp);
    }

    // 二进制读取
    fp = fopen("student.dat", "rb");
    if(fp != NULL) {
        fread(&stu2, sizeof(Student), 1, fp);
        printf("读取: %s, %d, %.1f\n", stu2.name, stu2.age, stu2.
            score);
        fclose(fp);
    }

    return 0;
}
```

Listing 27: 二进制文件操作

4.4 文件定位函数

```c
#include <stdio.h>

int main() {
    FILE *fp;
    char buffer[100];

    fp = fopen("test.txt", "w+");
    if(fp != NULL) {
        // 写入数据
        fprintf(fp, "这是第一行\n");
        fprintf(fp, "这是第二行\n");
        fprintf(fp, "这是第三行\n");

        // 回到文件开头
        rewind(fp);

        // 读取并显示
        printf("文件内容:\n");
        while(fgets(buffer, sizeof(buffer), fp) != NULL) {
            printf("%s", buffer);
        }

        // 使用fseek定位
        fseek(fp, 0, SEEK_SET);  // 定位到文件开头
        fseek(fp, 10, SEEK_CUR); // 从当前位置向前移动10字节
        fseek(fp, -20, SEEK_END); // 从文件末尾向前移动20字节

        // 获取当前位置
        long position = ftell(fp);
        printf("当前位置: %ld\n", position);

        fclose(fp);
    }

    return 0;
}
```

Listing 28: 文件定位

4.5 错误处理与文件状态

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

int main() {
    FILE *fp;
    char ch;

    fp = fopen("nonexistent.txt", "r");

```

```c
    if(fp == NULL) {
        perror("打开文件失败");  // 输出错误信息
        return 1;
    }

    // 读取文件直到结束
    while(1) {
        ch = fgetc(fp);

        if(feof(fp)) {  // 检查是否到达文件末尾
            printf("\n到达文件末尾\n");
            break;
        }

        if(ferror(fp)) {  // 检查是否发生错误
            printf("\n读取文件时发生错误\n");
            break;
        }

        printf("%c", ch);
    }

    // 清除错误标志
    clearerr(fp);

    fclose(fp);
    return 0;
}
```

Listing 29: 文件错误处理

4.6 文件操作综合示例

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

#define MAX_STUDENTS 100
#define FILENAME "students.dat"

typedef struct {
    int id;
    char name[50];
    int age;
    float score;
} Student;

Student students[MAX_STUDENTS];
int studentCount = 0;

// 保存数据到文件
```

```c
void saveToFile() {
    FILE *fp = fopen(FILENAME, "wb");
    if(fp == NULL) {
        printf("无法打开文件进行保存!\n");
        return;
    }

    // 写入学生数量
    fwrite(&studentCount, sizeof(int), 1, fp);

    // 写入所有学生数据
    fwrite(students, sizeof(Student), studentCount, fp);

    fclose(fp);
    printf("数据已保存到文件!\n");
}

// 从文件加载数据
void loadFromFile() {
    FILE *fp = fopen(FILENAME, "rb");
    if(fp == NULL) {
        printf("文件不存在或无法打开!\n");
        return;
    }

    // 读取学生数量
    fread(&studentCount, sizeof(int), 1, fp);

    // 读取学生数据
    fread(students, sizeof(Student), studentCount, fp);

    fclose(fp);
    printf("数据已从文件加载!\n");
}

// 添加学生
void addStudent() {
    if(studentCount >= MAX_STUDENTS) {
        printf("学生数量已达上限!\n");
        return;
    }

    Student *s = &students[studentCount];

    printf("请输入学号: ");
    scanf("%d", &s->id);

    printf("请输入姓名: ");
    scanf("%s", s->name);

    printf("请输入年龄: ");
```

```
    scanf("%d", &s->age);

    printf("请输入成绩: ");
    scanf("%f", &s->score);

    studentCount++;
    printf("学生添加成功!\n");
}

// 显示所有学生
void displayStudents() {
    if(studentCount == 0) {
        printf("没有学生信息!\n");
        return;
    }

    printf("\n%-10s %-20s %-10s %-10s\n",
           "学号", "姓名", "年龄", "成绩");
    printf("-----------------------------------------------\n");

    for(int i = 0; i < studentCount; i++) {
        printf("%-10d %-20s %-10d %-10.1f\n",
               students[i].id,
               students[i].name,
               students[i].age,
               students[i].score);
    }
}

// 菜单
void menu() {
    int choice;

    do {
        printf("\n=== 学生信息管理系统 ===\n");
        printf("1. 添加学生\n");
        printf("2. 显示所有学生\n");
        printf("3. 保存到文件\n");
        printf("4. 从文件加载\n");
        printf("0. 退出\n");
        printf("请选择: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                addStudent();
                break;
            case 2:
                displayStudents();
                break;
            case 3:
```

```
                saveToFile();
                break;
            case 4:
                loadFromFile();
                break;
            case 0:
                printf("感谢使用!\n");
                break;
            default:
                printf("无效选择!\n");
        }
    } while(choice != 0);
}

int main() {
    menu();
    return 0;
}
```

Listing 30: 学生信息管理系统

5 综合实例：通讯录管理系统

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

#define FILENAME "contacts.dat"

typedef struct {
    char name[50];
    char phone[20];
    char email[50];
} Contact;

typedef struct Node {
    Contact contact;
    struct Node *next;
} Node;

Node *head = NULL;

// 创建新节点
Node* createNode(Contact contact) {
    Node *newNode = (Node*)malloc(sizeof(Node));
    if(newNode == NULL) {
        printf("内存分配失败!\n");
        exit(1);
    }
    newNode->contact = contact;
```

```
    newNode->next = NULL;
    return newNode;
}

// 添加联系人
void addContact() {
    Contact contact;

    printf("请输入姓名: ");
    scanf("%s", contact.name);

    printf("请输入电话: ");
    scanf("%s", contact.phone);

    printf("请输入邮箱: ");
    scanf("%s", contact.email);

    Node *newNode = createNode(contact);

    // 插入到链表头部
    newNode->next = head;
    head = newNode;

    printf("联系人添加成功!\n");
}

// 显示所有联系人
void displayContacts() {
    Node *current = head;
    int count = 0;

    printf("\n=== 通讯录 ===\n");
    printf("%-20s %-15s %-30s\n", "姓名", "电话", "邮箱");
    printf("
       ------------------------------------------------------------\
       n");

    while(current != NULL) {
        printf("%-20s %-15s %-30s\n",
               current->contact.name,
               current->contact.phone,
               current->contact.email);
        current = current->next;
        count++;
    }

    printf("总共 %d 个联系人\n", count);
}

// 查找联系人
void searchContact() {
```

```
    char name[50];
    printf("请输入要查找的姓名: ");
    scanf("%s", name);

    Node *current = head;
    int found = 0;

    while(current != NULL) {
        if(strcmp(current->contact.name, name) == 0) {
            printf("找到联系人:\n");
            printf("姓名: %s\n", current->contact.name);
            printf("电话: %s\n", current->contact.phone);
            printf("邮箱: %s\n", current->contact.email);
            found = 1;
            break;
        }
        current = current->next;
    }

    if(!found) {
        printf("未找到联系人: %s\n", name);
    }
}

// 保存通讯录到文件
void saveContacts() {
    FILE *fp = fopen(FILENAME, "wb");
    if(fp == NULL) {
        printf("无法打开文件!\n");
        return;
    }

    Node *current = head;
    while(current != NULL) {
        fwrite(&current->contact, sizeof(Contact), 1, fp);
        current = current->next;
    }

    fclose(fp);
    printf("通讯录已保存到文件!\n");
}

// 从文件加载通讯录
void loadContacts() {
    FILE *fp = fopen(FILENAME, "rb");
    if(fp == NULL) {
        printf("文件不存在或无法打开!\n");
        return;
    }

    // 清空当前链表
```

```
    Node *current = head;
    while(current != NULL) {
        Node *temp = current;
        current = current->next;
        free(temp);
    }
    head = NULL;

    // 从文件读取
    Contact contact;
    while(fread(&contact, sizeof(Contact), 1, fp) == 1) {
        Node *newNode = createNode(contact);
        newNode->next = head;
        head = newNode;
    }

    fclose(fp);
    printf("通讯录已从文件加载!\n");
}

// 释放链表内存
void freeList() {
    Node *current = head;
    while(current != NULL) {
        Node *temp = current;
        current = current->next;
        free(temp);
    }
}

// 菜单
void menu() {
    int choice;

    do {
        printf("\n=== 通讯录管理系统 ===\n");
        printf("1. 添加联系人\n");
        printf("2. 显示所有联系人\n");
        printf("3. 查找联系人\n");
        printf("4. 保存到文件\n");
        printf("5. 从文件加载\n");
        printf("0. 退出\n");
        printf("请选择: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                addContact();
                break;
            case 2:
                displayContacts();
```

```
                break;
            case 3:
                searchContact();
                break;
            case 4:
                saveContacts();
                break;
            case 5:
                loadContacts();
                break;
            case 0:
                printf("感谢使用!\n");
                break;
            default:
                printf("无效选择!\n");
        }
    } while(choice != 0);

    freeList();
}

int main() {
    menu();
    return 0;
}
```

Listing 31: 通讯录管理系统

6 字符串常用方法

6.1 字符串基础回顾

在 C 语言中，字符串是以空字符 ('\0') 结尾的字符数组。

6.1.1 字符串声明与初始化

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

int main() {
    // 方法1: 字符数组
    char str1[] = "Hello";

    // 方法2: 指定大小的字符数组
    char str2[20] = "World";

    // 方法3: 字符指针 (指向字符串常量)
    char *str3 = "Hello World";

    // 方法4: 逐个字符初始化
    char str4[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

    printf("str1: %s\n", str1);
    printf("str2: %s\n", str2);
    printf("str3: %s\n", str3);
    printf("str4: %s\n", str4);

    return 0;
}
```

Listing 32: 字符串声明与初始化

6.2 字符串输入输出函数

6.2.1 gets 与 fgets

gets()	fgets()
不安全，可能导致缓冲区溢出	安全，可以指定最大读取长度
不保留换行符	保留换行符（如果读取到）
已从 C11 标准中移除	推荐使用

Table 4: gets 与 fgets 比较

```c
#include <stdio.h>

int main() {
    char name[50];
```

```c

    printf("请输入您的姓名: ");
    fgets(name, sizeof(name), stdin);

    // 移除可能的换行符
    int len = strlen(name);
    if(len > 0 && name[len-1] == '\n') {
        name[len-1] = '\0';
    }

    printf("您好, %s!\n", name);

    return 0;
}
```

Listing 33: fgets 使用示例

6.2.2 puts 与 fputs

```c
#include <stdio.h>

int main() {
    char str[] = "Hello World";

    // puts自动添加换行符
    puts(str);
    puts("This is a test");

    // fputs不添加换行符
    fputs(str, stdout);
    fputs("\n", stdout);

    // 写入文件
    FILE *fp = fopen("output.txt", "w");
    if(fp != NULL) {
        fputs(str, fp);
        fputs("\n", fp);
        fclose(fp);
    }

    return 0;
}
```

Listing 34: puts 与 fputs 使用

6.3 字符串处理函数

6.3.1 字符串长度 - strlen

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

```c

int main() {
    char str[] = "Hello World";
    char empty[] = "";

    printf("'%s' 的长度: %lu\n", str, strlen(str));
    printf("空字符串的长度: %lu\n", strlen(empty));

    // 注意: strlen不包括结尾的'\0'
    printf("sizeof(str): %lu\n", sizeof(str));  // 12 (包含'\0')
    printf("strlen(str): %lu\n", strlen(str));  // 11 (不包含'\0')

    return 0;
}
```

Listing 35: strlen 使用示例

6.3.2 字符串复制 - strcpy 与 strncpy

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

int main() {
    char src[] = "Hello World";
    char dest1[20];
    char dest2[20];
    char dest3[5];  // 目标缓冲区较小

    // strcpy - 不安全, 可能导致缓冲区溢出
    strcpy(dest1, src);
    printf("strcpy: %s\n", dest1);

    // strncpy - 安全, 指定最大复制长度
    strncpy(dest2, src, sizeof(dest2)-1);
    dest2[sizeof(dest2)-1] = '\0';  // 确保以'\0'结尾
    printf("strncpy: %s\n", dest2);

    // 处理目标缓冲区较小的情况
    strncpy(dest3, src, sizeof(dest3)-1);
    dest3[sizeof(dest3)-1] = '\0';
    printf("strncpy(截断): %s\n", dest3);

    return 0;
}
```

Listing 36: 字符串复制函数

6.3.3 字符串连接 - strcat 与 strncat

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

```c
#include <string.h>

int main() {
    char str1[50] = "Hello";
    char str2[] = " World";
    char str3[50] = "Hello";
    char str4[] = " Universe!";

    // strcat - 连接字符串
    strcat(str1, str2);
    printf("strcat: %s\n", str1);

    // strncat - 安全连接，指定最大添加长度
    strncat(str3, str4, 8);  // 只连接前8个字符
    printf("strncat: %s\n", str3);

    // 更安全的用法
    char buffer[20] = "Hello";
    strncat(buffer, " World and Everyone!",
            sizeof(buffer) - strlen(buffer) - 1);
    printf("安全连接: %s\n", buffer);

    return 0;
}
```

Listing 37: 字符串连接函数

6.3.4 字符串比较 - strcmp 与 strncmp

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

int main() {
    char str1[] = "apple";
    char str2[] = "apple";
    char str3[] = "banana";
    char str4[] = "APPLE";

    // strcmp - 完全比较
    printf("strcmp('%s', '%s') = %d\n", str1, str2, strcmp(str1, str2
        ));
    printf("strcmp('%s', '%s') = %d\n", str1, str3, strcmp(str1, str3
        ));
    printf("strcmp('%s', '%s') = %d\n", str3, str1, strcmp(str3, str1
        ));

    // 比较规则:
    // 返回0: 字符串相等
    // 返回<0: str1小于str2
    // 返回>0: str1大于str2

```

```
    // strncmp - 比较前n个字符
    printf("strncmp('%s', '%s', 3) = %d\n", str1, str4, strncmp(str1,
        str4, 3));

    // 不区分大小写的比较（非标准，Windows可用_stricmp）
    #ifdef _WIN32
    printf("_stricmp('%s', '%s') = %d\n", str1, str4, _stricmp(str1,
       str4));
    #endif

    return 0;
}
```

Listing 38: 字符串比较函数

6.3.5 字符串查找 - strchr 与 strstr

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

int main() {
    char str[] = "Hello World! Welcome to C programming.";

    // strchr - 查找字符第一次出现的位置
    char *p1 = strchr(str, 'W');
    if(p1 != NULL) {
        printf("找到 'W' 在位置: %ld\n", p1 - str);
        printf("从该位置开始的字符串: %s\n", p1);
    }

    // strchr - 查找字符最后一次出现的位置
    char *p2 = strrchr(str, 'o');
    if(p2 != NULL) {
        printf("最后一个 'o' 在位置: %ld\n", p2 - str);
    }

    // strstr - 查找子字符串
    char *p3 = strstr(str, "World");
    if(p3 != NULL) {
        printf("找到 'World' 在位置: %ld\n", p3 - str);
        printf("子字符串: %s\n", p3);
    }

    // 查找所有出现的字符
    printf("\n查找所有 'o' 的位置:\n");
    char *p = str;
    while((p = strchr(p, 'o')) != NULL) {
        printf("位置: %ld\n", p - str);
        p++;  // 移动到下一个位置继续查找
    }

```

```
    return 0;
}
```

Listing 39: 字符串查找函数

6.3.6 字符串分割 - strtok

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

int main() {
    char str[] = "apple,banana,orange,grape";
    char *token;

    printf("原始字符串: %s\n", str);
    printf("分割结果:\n");

    // 第一次调用，传入原始字符串
    token = strtok(str, ",");

    while(token != NULL) {
        printf("  %s\n", token);
        // 后续调用，传入NULL继续分割
        token = strtok(NULL, ",");
    }

    // 注意: strtok会修改原始字符串!
    printf("\n修改后的原始字符串: %s\n", str);

    return 0;
}
```

Listing 40: strtok 使用示例

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

int main() {
    char str[] = "John,Doe,25,New York";
    char backup[100];

    // 备份原始字符串
    strcpy(backup, str);

    char *tokens[10];
    int count = 0;

    // 使用strtok_r (可重入版本，更安全)
    char *saveptr;
    char *token = strtok_r(str, ",", &saveptr);
```

```c

    while(token != NULL && count < 10) {
        tokens[count] = token;
        count++;
        token = strtok_r(NULL, ",", &saveptr);
    }

    printf("CSV数据:\n");
    printf("原始字符串: %s\n", backup);
    printf("分割结果:\n");
    printf("  姓名: %s %s\n", tokens[0], tokens[1]);
    printf("  年龄: %s\n", tokens[2]);
    printf("  城市: %s\n", tokens[3]);

    return 0;
}
```

Listing 41: 更安全的分割方法

6.3.7 字符串转换函数

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

int main() {
    // atoi - 字符串转整数
    char str1[] = "123";
    char str2[] = "456.78";
    char str3[] = "789abc";

    int num1 = atoi(str1);
    int num2 = atoi(str2);
    int num3 = atoi(str3);

    printf("atoi('%s') = %d\n", str1, num1);
    printf("atoi('%s') = %d\n", str2, num2);
    printf("atoi('%s') = %d\n", str3, num3);

    // atof - 字符串转浮点数
    double f1 = atof(str2);
    printf("atof('%s') = %.2f\n", str2, f1);

    // strtol - 更安全的字符串转长整数
    char str4[] = "12345";
    char *endptr;
    long num4 = strtol(str4, &endptr, 10);  // 十进制

    if(*endptr == '\0') {
        printf("strtol('%s') = %ld (完全转换)\n", str4, num4);
```

```
    } else {
        printf("strtol('%s') = %ld (部分转换, 剩余: %s)\n", str4,
            num4, endptr);
    }

    // 大小写转换
    char str5[] = "Hello World";

    // 转换为小写
    for(int i = 0; str5[i]; i++) {
        str5[i] = tolower(str5[i]);
    }
    printf("小写: %s\n", str5);

    // 转换为大写
    for(int i = 0; str5[i]; i++) {
        str5[i] = toupper(str5[i]);
    }
    printf("大写: %s\n", str5);

    return 0;
}
```

Listing 42: 字符串转换示例

6.4 字符串格式化函数

6.4.1 sprintf 与 snprintf

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

int main() {
    char buffer[100];
    int age = 25;
    float salary = 5000.50;
    char name[] = "张三";

    // sprintf - 格式字符串写入缓冲区
    sprintf(buffer, "姓名: %s, 年龄: %d, 工资: %.2f",
            name, age, salary);
    printf("%s\n", buffer);

    // snprintf - 安全版本, 指定最大长度
    char small_buffer[20];
    int written = snprintf(small_buffer, sizeof(small_buffer),
                           "姓名: %s, 年龄: %d", name, age);

    if(written >= sizeof(small_buffer)) {
        printf("缓冲区不足! 需要 %d 字节, 只有 %lu 字节\n",
               written, sizeof(small_buffer));
```

```
    }
    printf("实际写入: %s\n", small_buffer);

    return 0;
}
```

Listing 43: sprintf 使用示例

6.4.2 sscanf

```c
#include <stdio.h>

int main() {
    char data[] = "John Doe 25 5000.50";
    char first[20], last[20];
    int age;
    float salary;

    // 从字符串中提取格式化数据
    int result = sscanf(data, "%s %s %d %f",
                        first, last, &age, &salary);

    if(result == 4) {
        printf("成功读取 %d 个字段:\n", result);
        printf("  姓: %s\n", last);
        printf("  名: %s\n", first);
        printf("  年龄: %d\n", age);
        printf("  工资: %.2f\n", salary);
    }

    // 提取特定格式的数据
    char date[] = "2024-01-15";
    int year, month, day;
    sscanf(date, "%d-%d-%d", &year, &month, &day);
    printf("\n日期: %d年%d月%d日\n", year, month, day);

    return 0;
}
```

Listing 44: sscanf 使用示例

6.5 自定义字符串函数

6.5.1 实现常用字符串函数

```c
#include <stdio.h>

// 自定义strlen函数
size_t my_strlen(const char *str) {
    size_t len = 0;
    while(str[len] != '\0') {
```

```
        len++;
    }
    return len;
}

// 自定义strcpy函数
char* my_strcpy(char *dest, const char *src) {
    char *original = dest;
    while((*dest++ = *src++) != '\0');
    return original;
}

// 自定义strcat函数
char* my_strcat(char *dest, const char *src) {
    char *original = dest;

    // 移动到dest的末尾
    while(*dest != '\0') {
        dest++;
    }

    // 复制src到dest末尾
    while((*dest++ = *src++) != '\0');

    return original;
}

// 自定义strcmp函数
int my_strcmp(const char *str1, const char *str2) {
    while(*str1 && *str2 && *str1 == *str2) {
        str1++;
        str2++;
    }
    return *(unsigned char*)str1 - *(unsigned char*)str2;
}

int main() {
    // 测试自定义函数
    char str1[50] = "Hello";
    char str2[] = " World";

    printf("my_strlen('%s') = %lu\n", str1, my_strlen(str1));

    my_strcat(str1, str2);
    printf("my_strcat: %s\n", str1);

    printf("my_strcmp('apple', 'apple') = %d\n",
           my_strcmp("apple", "apple"));
    printf("my_strcmp('apple', 'banana') = %d\n",
           my_strcmp("apple", "banana"));

```

```
    return 0;
}
```

Listing 45: 自定义字符串函数实现

6.6 字符串处理综合示例

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

// 去除字符串两端的空白字符
char* trim(char *str) {
    if(str == NULL) return NULL;

    // 去除开头的空白字符
    char *start = str;
    while(isspace(*start)) {
        start++;
    }

    // 去除结尾的空白字符
    char *end = str + strlen(str) - 1;
    while(end > start && isspace(*end)) {
        end--;
    }
    *(end + 1) = '\0';

    // 如果字符串开头有空白，需要移动字符串
    if(start != str) {
        memmove(str, start, strlen(start) + 1);
    }

    return str;
}

// 字符串反转
char* reverse_string(char *str) {
    if(str == NULL) return NULL;

    int len = strlen(str);
    for(int i = 0; i < len / 2; i++) {
        char temp = str[i];
        str[i] = str[len - i - 1];
        str[len - i - 1] = temp;
    }
    return str;
}

// 统计单词数量
```

```c
int count_words(const char *str) {
    if(str == NULL) return 0;

    int count = 0;
    int in_word = 0;

    for(int i = 0; str[i] != '\0'; i++) {
        if(!isspace(str[i])) {
            if(!in_word) {
                count++;
                in_word = 1;
            }
        } else {
            in_word = 0;
        }
    }

    return count;
}

// 判断是否为回文字符串
int is_palindrome(const char *str) {
    if(str == NULL) return 0;

    int len = strlen(str);
    for(int i = 0; i < len / 2; i++) {
        if(tolower(str[i]) != tolower(str[len - i - 1])) {
            return 0;
        }
    }
    return 1;
}

int main() {
    char text[] = "   Hello World! Welcome to C programming.   ";
    char palindrome[] = "A man a plan a canal Panama";

    printf("原始文本: '%s'\n", text);

    // 去除空白
    printf("去除空白后: '%s'\n", trim(text));

    // 统计单词
    printf("单词数量: %d\n", count_words(text));

    // 测试回文
    char test1[] = "racecar";
    char test2[] = "hello";

    printf("'%s' 是回文吗? %s\n", test1,
           is_palindrome(test1) ? "是" : "否");
```

```c
    printf("'%s' 是回文吗? %s\n", test2,
           is_palindrome(test2) ? "是" : "否");

    // 清理句子中的空格并测试回文
    char clean_palindrome[100];
    int j = 0;
    for(int i = 0; palindrome[i] != '\0'; i++) {
        if(!isspace(palindrome[i])) {
            clean_palindrome[j++] = tolower(palindrome[i]);
        }
    }
    clean_palindrome[j] = '\0';

    printf("清理后: '%s'\n", clean_palindrome);
    printf("是回文吗? %s\n",
           is_palindrome(clean_palindrome) ? "是" : "否");

    // 字符串反转
    char to_reverse[] = "Hello";
    printf("'%s' 反转后: '%s'\n",
           to_reverse, reverse_string(to_reverse));

    return 0;
}
```

Listing 46: 字符串处理综合应用

6.7 字符串安全注意事项

重要安全提醒:

- 避免使用不安全的函数: gets, strcpy, strcat 等
- 使用安全版本: fgets, strncpy, strncat 等
- 始终检查缓冲区边界
- 确保字符串以'\0' 结尾
- 使用 strncpy 时手动添加终止符
- 处理用户输入时要特别小心

6.7.1 安全字符串处理示例

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

// 安全字符串复制函数
int safe_strcpy(char *dest, size_t dest_size, const char *src) {
```

```c
    if(dest == NULL || src == NULL || dest_size == 0) {
        return -1;
    }

    size_t src_len = strlen(src);

    // 检查是否有足够空间（包括终止符）
    if(src_len >= dest_size) {
        // 复制尽可能多的字符
        strncpy(dest, src, dest_size - 1);
        dest[dest_size - 1] = '\0';
        return -2;  // 表示被截断
    }

    strcpy(dest, src);
    return 0;  // 成功
}

// 安全字符串连接函数
int safe_strcat(char *dest, size_t dest_size, const char *src) {
    if(dest == NULL || src == NULL || dest_size == 0) {
        return -1;
    }

    size_t dest_len = strlen(dest);
    size_t src_len = strlen(src);

    // 检查是否有足够空间
    if(dest_len + src_len + 1 > dest_size) {
        // 连接尽可能多的字符
        strncat(dest, src, dest_size - dest_len - 1);
        return -2;  // 表示被截断
    }

    strcat(dest, src);
    return 0;  // 成功
}

int main() {
    char buffer[20];

    // 安全复制
    int result = safe_strcpy(buffer, sizeof(buffer), "Hello World!");
    if(result == -2) {
        printf("警告：字符串被截断\n");
    }
    printf("缓冲区：%s\n", buffer);

    // 尝试复制过长的字符串
    result = safe_strcpy(buffer, sizeof(buffer),
                         "This is a very long string that won't fit");
```

```c
    if(result == -2) {
        printf("警告：字符串被截断\n");
    }
    printf("缓冲区：%s\n", buffer);

    // 安全连接
    safe_strcpy(buffer, sizeof(buffer), "Hello");
    result = safe_strcat(buffer, sizeof(buffer), " World!");
    if(result == 0) {
        printf("安全连接：%s\n", buffer);
    }

    return 0;
}
```

Listing 47: 安全字符串处理

7 最佳实践与常见错误

7.1 结构体最佳实践

1. **使用 typedef 创建别名**：提高代码可读性
2. **合理设计结构体布局**：考虑内存对齐，将相关数据放在一起
3. **使用 const 保护数据**：对于不需要修改的结构体参数，使用 const 修饰
4. **考虑结构体大小**：避免创建过大的结构体，考虑使用指针
5. **初始化所有成员**：防止未初始化错误

7.2 链表最佳实践

1. **检查内存分配**：每次 malloc 后检查返回值是否为 NULL
2. **及时释放内存**：使用 free 释放不再需要的节点
3. **处理边界情况**：考虑空链表、单个节点等情况
4. **避免内存泄漏**：确保每个 malloc 都有对应的 free
5. **使用哨兵节点**：简化插入和删除操作

7.3 文件操作最佳实践

1. **检查文件指针**：每次 fopen 后检查返回值是否为 NULL
2. **及时关闭文件**：使用 fclose 关闭不再需要的文件
3. **检查读写操作**：验证 fread/fwrite 等函数的返回值
4. **使用二进制模式处理结构体**：避免文本模式下的格式问题
5. **处理文件错误**：使用 feof 和 ferror 检查文件状态

7.4 常见错误

```c
// 错误1：忘记结构体末尾的分号
struct Student {  // 错误：缺少分号
    char name[50];
    int age;
}

// 错误2：访问未初始化的结构体指针
struct Student *p;
p->age = 20;  // 错误：p未初始化

// 错误3：链表节点未初始化next指针
Node *newNode = malloc(sizeof(Node));
newNode->data = 10;
// 忘记设置newNode->next = NULL;

// 错误4：文件未关闭
FILE *fp = fopen("test.txt", "w");
// ... 使用文件
// 忘记 fclose(fp);  // 内存泄漏

// 错误5：文件模式使用错误
fp = fopen("binary.dat", "r");  // 错误：应该用"rb"
```

Listing 48: 常见错误示例

8 补充

8.0.1 动态内存分配的综合应用

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

// 动态数组结构
typedef struct {
    int *data;      // 数组数据
    int size;       // 当前元素数量
    int capacity;   // 总容量
} DynamicArray;

// 初始化动态数组
DynamicArray* createDynamicArray(int initialCapacity) {
    DynamicArray *arr = (DynamicArray*)malloc(sizeof(DynamicArray));
    if(arr == NULL) return NULL;

    arr->data = (int*)malloc(initialCapacity * sizeof(int));
    if(arr->data == NULL) {
        free(arr);
        return NULL;
```

```
    }

    arr->size = 0;
    arr->capacity = initialCapacity;
    return arr;
}

// 添加元素
int addElement(DynamicArray *arr, int value) {
    if(arr->size >= arr->capacity) {
        // 容量不足，扩展
        int newCapacity = arr->capacity * 2;
        int *newData = (int*)realloc(arr->data, newCapacity * sizeof(
            int));

        if(newData == NULL) {
            return 0;  // 扩展失败
        }

        arr->data = newData;
        arr->capacity = newCapacity;
    }

    arr->data[arr->size] = value;
    arr->size++;
    return 1;
}

// 获取元素
int getElement(DynamicArray *arr, int index) {
    if(index < 0 || index >= arr->size) {
        printf("索引越界!\n");
        return -1;
    }
    return arr->data[index];
}

// 释放动态数组
void freeDynamicArray(DynamicArray *arr) {
    if(arr != NULL) {
        if(arr->data != NULL) {
            free(arr->data);
            arr->data = NULL;
        }
        free(arr);
    }
}

// 使用示例
int main() {
    DynamicArray *arr = createDynamicArray(5);
```

```c

    if(arr == NULL) {
        printf("创建动态数组失败!\n");
        return 1;
    }

    // 添加元素
    for(int i = 0; i < 10; i++) {
        if(!addElement(arr, i * 10)) {
            printf("添加元素失败!\n");
            break;
        }
    }

    // 输出元素
    printf("动态数组内容:\n");
    for(int i = 0; i < arr->size; i++) {
        printf("arr[%d] = %d\n", i, getElement(arr, i));
    }

    printf("数组大小: %d, 容量: %d\n", arr->size, arr->capacity);

    // 释放内存
    freeDynamicArray(arr);

    return 0;
}
```

Listing 49: 动态数组实现