重学标准IO

操作系统编程

宿船长

Press Space for next page →

什么是标准IO

> 标准I/O是C语言中最基本的输入输出操作机制。

```
 7
 8 #include<stdio.h>
 9 int main() {
10
11     print("Hello World!\n");
12     return 0;
13 }
```

stdio.h头文件包含什么？

> C语言的stdio.h中主要包涵以下几类操作函数

- ✨ 标准输入输出函数（printf、scanf）
- 📁 文件操作函数（fopen、fclose）
- 🔄 缓冲区控制函数（setbuf、fflush）
- ⚠ 错误处理函数（ferror、clearerr）

标准I/O的特点

> 提供缓冲机制，在用户程序和操作系统之间建立缓冲层，显著提升I/O效率

标准I/O与系统调用的关系

标准I/O库是构建在系统调用之上的高级接口，它为程序员提供了更便捷的I/O操作方式。

> 🔍 标准I/O库在底层仍然使用read()、write()等系统调用来完成实际的I/O操作

主要区别：

- 📦 标准I/O提供缓冲机制，而系统调用是无缓冲的直接操作
- 🚀 标准I/O效率更高，因为它减少了系统调用的次数
- ⚙️ 系统调用直接与内核交互，而标准I/O通过库函数间接调用

工作流程：

```
用户程序 -> 标准I/O库函数 -> 系统调用 -> 操作系统内核
```

这种分层设计既保证了效率，又提供了良好的可移植性和易用性。

常见的标准文件流

> C语言定义了三个标准文件流，它们在程序启动时自动打开

- 📥 标准输入流（stdin）- 默认与键盘关联- 用于从键盘读取输入数据- 通过scanf、getchar等函数使用
- 📤 标准输出流（stdout）- 默认与显示器关联- 用于向屏幕输出数据- 通过printf、putchar等函数使用
- ⚠️ 标准错误流（stderr）- 默认与显示器关联- 用于输出错误信息和诊断信息- 不使用缓冲区，直接输出

重定向操作：

标准流可以通过重定向机制改变其输入输出目标：

```
./program < input.txt  # 输入重定向
./program > output.txt # 输出重定向
./program 2> error.txt # 错误重定向
```

fprintf函数的使用

> fprintf是一个格式化输出函数，可以将格式化的数据写入指定的文件流中

函数原型：

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

参数说明：

- 📄 stream：指向FILE结构的文件指针
- 📝 format：格式化字符串
- ...：可变参数列表

使用示例：

```
FILE *fp = fopen("test.txt", "w");
if (fp != NULL) {
    fprintf(fp, "Hello, %s!\\n", "World");
    fprintf(fp, "Number: %d\\n", 42);
    fclose(fp);
}
```

什么是缓冲?

> 缓冲是一种临时存储机制，在数据传输过程中起到"中转站"的作用

缓冲的基本概念:

- 💾 缓冲区是内存中的一块区域，用于临时存储数据
- 🔄 数据传输过程:
  - 程序 ↔ 缓冲区 ↔ 外部设备
- ⚡ 主要作用:
  - 减少实际I/O操作的次数
  - 提高数据传输效率
  - 协调速度不匹配的设备

举个例子：想象一个水桶接水的过程

- 🚰 水龙头代表数据源（如键盘输入）
- 🪣 水桶代表缓冲区
- 🌱 植物代表最终的数据使用者（如文件）

我们不会对每滴水都进行单独处理，而是等水桶装满后，一次性浇灌。这就是缓冲的基本原理。

缓冲I/O机制

> 缓冲I/O是标准I/O的一个重要特性，它通过在用户空间设置缓冲区来提高I/O操作的效率

缓冲区的类型：

- 🔄 全缓冲：缓冲区满时才进行实际的I/O操作
  - 典型例子：文件的读写操作
- 📝 行缓冲：遇到换行符时进行I/O操作
  - 典型例子：标准输出（stdout）
- ❌ 无缓冲：直接进行I/O操作，不使用缓冲区
  - 典型例子：标准错误（stderr）

缓冲区操作函数：

```c
// 设置流的缓冲区
void setbuf(FILE *stream, char *buf);

// 设置流的缓冲区，可以指定缓冲区大小
int setvbuf(FILE *stream, char *buf, int mode, size_t size);

// 刷新缓冲区
int fflush(FILE *stream);
```

缓冲区模式说明：

- _IOFBF：全缓冲模式
- _IOLBF：行缓冲模式
- _IONBF：无缓冲模式

使用示例：

```c
FILE *fp = fopen("test.txt", "w");
if (fp != NULL) {
    // 设置无缓冲模式
    setvbuf(fp, NULL, _IONBF, 0);

    // 或设置自定义缓冲区
    char buf[1024];
    setvbuf(fp, buf, _IOFBF, sizeof(buf));

    // 强制刷新缓冲区
    fflush(fp);

    fclose(fp);
}
```

缓冲区的优势：

- 📈 减少系统调用次数，提高性能
- 🔋 降低系统资源消耗
- ⚡ 提供更高效的数据传输

注意事项：

- ⚠ 程序异常终止时缓冲区数据可能丢失
- 🔍 需要及时刷新重要数据
- 📊 合理选择缓冲区大小很重要

缓冲IO的效率验证

1. 在Linux主机上，使用open，write系统调用来实现一个程序
2. 分别模拟有缓冲，缓冲大小设置为4096，和没有缓冲（一次写入一个字符）的情况下
3. 将等量的数据写入同一个文件，查看效率差异

```c
//请编码实现
void buffered_write(int fd, const char *data, size_t count);
void test_buffered_io();
void test_unbuffered_io();
int main(int argc, char *argv[]) {
    int use_buffer = 0;
    int opt;
    //获取选项
    if (use_buffer) {
        printf("执行缓冲I/O测试...\n");
        test_buffered_io();
    } else {
        printf("执行无缓冲I/O测试...\n");
        test_unbuffered_io();
    }
    return 0;
}
```

预期结果：

- 🚀 有缓冲I/O的执行时间显著少于无缓冲I/O
- 📊 两种方式生成的文件大小应该相同
- 💡 差异原因：缓冲I/O减少了实际的系统调用次数

文件的打开和关闭

> 标准I/O库提供了三个主要的文件打开和关闭函数：fopen、fclose和freopen

- FILE *fopen(const char* filename, const char *mode) - 打开文件
- int fclose(FILE *stream) - 关闭文件
- FILE *freopen(const char* filename, const char *mode, FILE* stream) - 重新打开文件

1. fopen 函数

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

- 🔑 功能：打开一个文件并创建文件流
- 📝 参数说明：
  - filename：要打开的文件名
  - mode：文件打开模式（"r"读取，"w"写入，"a"追加等）
- ↩️ 返回值：成功返回FILE指针，失败返回NULL

常用文件打开模式：

模式	描述
"r"	只读模式打开文件
"w"	只写模式打开文件（会清空原内容）
"a"	追加模式打开文件
"r+"	读写模式打开文件
"w+"	读写模式打开文件（会清空原内容）
"a+"	读写模式打开文件（追加）

2. fclose 函数

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

- 🔒 功能：关闭文件流并刷新缓冲区
- 📝 参数说明：stream为要关闭的文件流指针
- ↩️ 返回值：成功返回0，失败返回EOF

3. freopen 函数

```
FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

- 🔄 功能：重新打开一个文件流
- 📝 参数说明：
  - filename：新文件名
  - mode：打开模式
  - stream：要重新打开的文件流
- ↩️ 返回值：成功返回新的FILE指针，失败返回NULL

小练习：标准I/O重定向综合实验

> 这个练习将帮助你理解如何使用freopen函数同时处理输入和输出重定向。

实验目标：创建一个程序，从文件读取成绩数据，计算平均分，并将结果写入新文件。

程序要求：

1. 准备输入文件"grades.txt"，每行包含一个学生成绩（0-100的整数）
2. 程序需要完成：
   - 读取所有成绩
   - 计算平均分
   - 输出结果到"results.txt"文件

文件读写

1. fgetc 和 fputc

```
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
```

- 🔍 功能：单字符的读写操作
- 📝 参数说明：
  - stream：文件流指针
  - c：要写入的字符
- ↩️ 返回值：
  - fgetc：成功返回读取的字符，失败或到达文件末尾返回EOF
  - fputc：成功返回写入的字符，失败返回EOF

文件读写

2. fgets 和 fputs

```
char *fgets(char *str, int n, FILE *stream);
int fputs(const char *str, FILE *stream);
```

- 📝 功能：行级别的读写操作
- 参数说明：
  - str：字符串缓冲区
  - n：最大读取字符数（包含空字符）
  - stream：文件流指针

文件读写

3. fread 和 fwrite

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

- 🔢 功能：二进制数据的块级读写
- 📝 参数说明：
  - ptr：数据缓冲区指针
  - size：每个元素的大小
  - nmemb：元素个数
  - stream：文件流指针

vfprintf 的高级应用

```
int vfprintf(FILE *stream, const char *format, va_list arg);
```

- 🔍 主要用途：
  - 实现自定义的格式化输出函数
  - 处理可变参数的日志记录
  - 封装通用的输出接口

文件指针操作

> 文件指针操作允许在文件中随机访问数据，提供了灵活的文件处理方式

1. fseek 函数

```c
int fseek(FILE *stream, long offset, int whence);
```

- 🎯 功能：移动文件指针到指定位置
- 📝 参数说明：
  - stream：文件流指针
  - offset：偏移量（字节数）
  - whence：起始位置（SEEK_SET, SEEK_CUR, SEEK_END）

文件指针操作

2. ftell 函数

```
long ftell(FILE *stream);
```

- 📍 功能：获取当前文件指针位置
- ↩️ 返回值：返回当前位置距文件开始的字节数

文件指针操作

3. rewind 函数

```
void rewind(FILE *stream);
```

- 🔄 功能：将文件指针重置到文件开头
- 📝 特点：
  - 等效于 fseek(stream, 0L, SEEK_SET)
  - 同时清除文件的错误标志
  - 不返回任何值

临时文件

> 临时文件函数提供了创建和管理临时文件的便捷方式，适用于需要临时存储数据的场景

1. tmpfile 函数

```
FILE *tmpfile(void);
```

- 🔍 功能：创建临时二进制文件
- 📝 特点：
  - 自动在程序结束时删除
  - 以 "wb+" 模式打开
  - 返回文件指针

临时文件

2. tmpnam 函数

```
char *tmpnam(char *str);
```

- 🔍 功能：生成唯一的临时文件名
- 📝 参数说明：
  - str：存储文件名的缓冲区，如果为NULL则使用静态缓冲区

📝多级日志系统实现 - 任务书

项目目标：

实现一个基于C语言的多级日志系统，支持不同级别的日志记录功能。

功能需求：

- 实现三个日志级别：INFO、WARNING、ERROR
- 支持日志文件的创建、写入和关闭操作
- 每条日志需包含时间戳和日志级别标识
- 支持格式化日志消息输出

技术规范：

- 使用标准I/O函数进行文件操作
- 采用可变参数实现格式化输出
- 确保线程安全的日志写入
- 实现文件缓冲区自动刷新

参考代码

```
typedef enum {
    LOG_INFO,
    LOG_WARNING,
    LOG_ERROR
} LogLevel;
int init_logger(const char *filename);
void close_logger();
const char* get_level_str(LogLevel level);
void log_message(LogLevel level, const char *format, ...);
int main() {
    // 初始化日志系统
    if (!init_logger("app.log")) {
        printf("无法创建日志文件!\n");
        return 1;
    }

    // 记录不同级别的日志
    log_message(LOG_INFO, "系统启动成功");
    log_message(LOG_WARNING, "磁盘空间不足: %d%%", 15);
    log_message(LOG_ERROR, "数据库连接失败: %s", "Connection timeout");

    // 关闭日志系统
    close_logger();
    return 0;
}
```

📝 文本文件倒排索引实现 - 任务书

任务描述：

本任务要求实现一个基于C语言的文本文件倒排索引系统。该系统能够建立单词到文档的映射关系，实现文本的快速检索功能。

主要功能要求：

- 实现索引数据结构，包括单词存储和行号记录
- 完成单词添加到索引的功能
- 实现文本文件的索引构建过程
- 开发单词查询功能

技术规格：

- 最大单词长度：50字符
- 最大行长度：1024字符
- 最大单词数量：1000个

参考代码

```
#define MAX_WORD_LEN 50
#define MAX_LINE_LEN 1024
#define MAX_WORDS 1000
typedef struct {
    char word[MAX_WORD_LEN];
    int line_numbers[MAX_LINE_LEN];
    int count;
} IndexItem; // 索引项结构
typedef struct {
    IndexItem items[MAX_WORDS];
    int size;
} IndexDB; // 索引数据库
void init_index(IndexDB *db);
void add_to_index(IndexDB *db, const char *word, int line_num);
void build_index( *db, const char *filename);
void search_word(const IndexDB *db, const char *word);
int main() {
    IndexDB db;
    init_index(&db);
    build_index(&db, "sample.txt");
    search_word(&db, "hello");
    search_word(&db, "world");
    return 0;
}
```