## **青岛科技大学实验报告**

姓名： 刘洋 专业： 计算机科学与技术 班级： 计算213

学号： 2108010316 课程： 编译原理 时间： 2024.05.19

实验项目： 实验1 词法分析器的设计与实现

**实验名称：**词法分析

实验目的和要求

实现一个简单的词法分析器，从文件中读取C语言代码，识别并区分出不同类型的单词（tokens），如标识符、关键字、数字、运算符、分隔符、注释、字符串和字符常量等。

实验内容和步骤：

**一、实验内容**

本次实验的目的是实现一个简单的词法分析器，从文件中读取C语言代码，识别并区分出不同类型的单词（tokens），如标识符、关键字、数字、运算符、分隔符、注释、字符串和字符常量。具体内容包括：

1.从文件中读取内容。2.识别并区分C语言中的不同词法单元。3.输出每个词法单元的类型和值。

**二、实验步骤**

1.模块构建：定义并实现三个主要模块：主函数、扫描函数、拆分字符串函数。关键步骤包括定义关键字列表、Token类型和结构，设计用于识别Token的函数，以及实现从文件读取源代码的函数。

2.模块实现：2.1 定义关键字和Token结构：

列出所有的C语言关键字。 定义Token类型的枚举和Token结构体。2.2扫描函数（getNextToken）：实现用于从输入字符串中识别并提取下一个Token的函数。识别并分类标识符、关键字、数字、运算符、分隔符、注释、字符串、字符常量等。2.3文件读取函数（readFile）：实现用于从文件读取内容并返回字符串的函数。

3.调用模块：在主函数中，调用文件读取函数读取源代码。使用扫描函数逐个提取Token并输出Token的类型和值

**三、模块设计**

1.定义关键字列表和Token结构：

1.1使用一个字符串数组存储所有C语言的关键字。

1.2使用枚举定义Token的类型。

1.3使用结构体定义Token，包含类型和值两个字段。

2.扫描函数实现：

2.1实现isKeyword函数用于判断一个字符串是否为关键字。

2.2实现getNextToken函数用于从输入字符串中提取下一个Token。根据不同的字符类型，识别并分类不同类型的Token。

2.3在getNextToken中，分别处理标识符/关键字、数字、注释、字符串、字符常量、运算符和分隔符。

3.文件读取函数实现：

实现readFile函数用于从文件读取内容，并返回以NULL结尾的字符串。

4.主函数实现：

4.1在主函数中调用readFile函数读取源代码。

4.2使用getNextToken函数逐个提取Token，并输出Token的类型和值。

4.3释放动态分配的内存。

实验代码如下：

#include <stdio.h>

#include <ctype.h>

#include <string.h>

#include <stdlib.h>

// 关键字列表

const char \*keywords[] = {

"auto", "break", "case", "char", "const", "continue", "default",

"do", "double", "else", "enum", "extern", "float", "for", "goto",

"if", "int", "long", "register", "return", "short", "signed",

"sizeof", "static", "struct", "switch", "typedef", "union",

"unsigned", "void", "volatile", "while"

};

#define NUM\_KEYWORDS (sizeof(keywords) / sizeof(keywords[0]))

// Token类型

typedef enum {

TOKEN\_IDENTIFIER,

TOKEN\_KEYWORD,

TOKEN\_NUMBER,

TOKEN\_OPERATOR,

TOKEN\_SEPARATOR,

TOKEN\_COMMENT,

TOKEN\_STRING,

TOKEN\_CHAR,

TOKEN\_PREPROCESSOR,

TOKEN\_UNKNOWN,

TOKEN\_END

} TokenType;

// Token结构

typedef struct {

TokenType type;

char value[256];

} Token;

// 检查是否为关键字

int isKeyword(const char \*str) {

int i;

for (i = 0; i < NUM\_KEYWORDS; i++) {

if (strcmp(str, keywords[i]) == 0) {

return 1;

}

}

return 0;

}

// 获取下一个Token

Token getNextToken(const char \*\*input) {

Token token;

token.type = TOKEN\_UNKNOWN;

token.value[0] = '\0';

const char \*ptr = \*input;

while (isspace(\*ptr)) {

ptr++;

}

if (\*ptr == '\0') {

token.type = TOKEN\_END;

strcpy(token.value, "EOF");

\*input = ptr;

return token;

}

// 识别标识符或关键字

if (isalpha(\*ptr) || \*ptr == '\_') {

int len = 0;

while (isalnum(\*ptr) || \*ptr == '\_') {

token.value[len++] = \*ptr++;

}

token.value[len] = '\0';

token.type = isKeyword(token.value) ? TOKEN\_KEYWORD : TOKEN\_IDENTIFIER;

\*input = ptr;

return token;

}

// 识别头文件

if (\*ptr == '<') {

int len = 0;

token.value[len++] = \*ptr++; // 将 '<' 添加到 token 值中

int isHeaderFile = 1; // 默认为头文件

while (\*ptr && \*ptr != '>') {

if (!isspace(\*ptr) && (\*(ptr + 1) == '\n' || \*(ptr + 1) == '\r' || \*(ptr + 1) == '\0')) {

isHeaderFile = 0; // 如果下一个字符是换行符或者文件结束符，认为不是头文件

}

token.value[len++] = \*ptr++; // 将头文件名添加到 token 值中

}

if (\*ptr == '>') {

token.value[len++] = \*ptr++; // 将 '>' 添加到 token 值中

}

token.value[len] = '\0';

token.type = isHeaderFile ? TOKEN\_STRING : TOKEN\_OPERATOR; // 根据是否头文件来设置类型

if(token.type==TOKEN\_OPERATOR){

while(len--)

{

ptr--;

}

ptr++;

token.value[0]='<';

token.value[1]='\0';

}

\*input = ptr;

return token;

}

// 识别数字

if (isdigit(\*ptr) || \*ptr == '.') {

int len = 0;

int hasDot = 0;

while (isdigit(\*ptr) || (\*ptr == '.' && !hasDot)) {

if (\*ptr == '.') {

hasDot = 1;

}

token.value[len++] = \*ptr++;

}

token.value[len] = '\0';

token.type = TOKEN\_NUMBER;

\*input = ptr;

return token;

}

// 识别注释

if (\*ptr == '/' && \*(ptr + 1) == '/') {

int len = 0;

while (\*ptr && \*ptr != '\n') {

token.value[len++] = \*ptr++;

}

token.value[len] = '\0';

token.type = TOKEN\_COMMENT;

\*input = ptr;

return token;

}

if (\*ptr == '/' && \*(ptr + 1) == '\*') {

int len = 0;

ptr += 2; // 跳过 /\*

while (\*ptr && !(\*ptr == '\*' && \*(ptr + 1) == '/')) {

token.value[len++] = \*ptr++;

}

if (\*ptr == '\*' && \*(ptr + 1) == '/') {

ptr += 2; // 跳过 \*/

}

token.value[len] = '\0';

token.type = TOKEN\_COMMENT;

\*input = ptr;

return token;

}

// 识别字符串

if (\*ptr == '"') {

int len = 0;

token.value[len++] = \*ptr++;

while (\*ptr && \*ptr != '"') {

token.value[len++] = \*ptr++;

}

if (\*ptr == '"') {

token.value[len++] = \*ptr++;

}

token.value[len] = '\0';

token.type = TOKEN\_STRING;

\*input = ptr;

return token;

}

// 识别字符常量

if (\*ptr == '\'') {

int len = 0;

token.value[len++] = \*ptr++;

while (\*ptr && \*ptr != '\'') {

token.value[len++] = \*ptr++;

}

if (\*ptr == '\'') {

token.value[len++] = \*ptr++;

}

token.value[len] = '\0';

token.type = TOKEN\_CHAR;

\*input = ptr;

return token;

}

// 识别运算符和分隔符

const char \*operators[] = { "==", "!=", "<=", ">=", "&&", "||", "++", "--", "->", "+=", "-=", "\*=", "/=", "%=", "&=", "|=", "^=", "<<", ">>" };

int matched = 0;

int i;

for (i = 0; i < sizeof(operators) / sizeof(operators[0]); i++) {

int len = strlen(operators[i]);

if (strncmp(ptr, operators[i], len) == 0) {

strncpy(token.value, ptr, len);

token.value[len] = '\0';

token.type = TOKEN\_OPERATOR;

ptr += len;

matched = 1;

break;

}

}

if (!matched) {

if (ispunct(\*ptr)) {

token.value[0] = \*ptr++;

token.value[1] = '\0';

token.type = TOKEN\_SEPARATOR;

} else {

token.value[0] = \*ptr++;

token.value[1] = '\0';

token.type = TOKEN\_UNKNOWN;

}

}

\*input = ptr;

return token;

}

// 获取Token类型的字符串表示

const char\* getTokenTypeString(TokenType type) {

switch (type) {

case TOKEN\_IDENTIFIER: return "Identifier";

case TOKEN\_KEYWORD: return "Keyword";

case TOKEN\_NUMBER: return "Number";

case TOKEN\_OPERATOR: return "Operator";

case TOKEN\_SEPARATOR: return "Separator";

case TOKEN\_COMMENT: return "Comment";

case TOKEN\_STRING: return "String";

case TOKEN\_CHAR: return "Char";

case TOKEN\_PREPROCESSOR: return "Preprocessor";

case TOKEN\_UNKNOWN: return "Unknown";

case TOKEN\_END: return "End";

default: return "Unknown";

}

}

// 从文件中读取内容

char\* readFile(const char \*filename) {

FILE \*file = fopen(filename, "rb");

if (!file) {

perror("Unable to open file");

return NULL;

}

fseek(file, 0, SEEK\_END);

long length = ftell(file);

fseek(file, 0, SEEK\_SET);

char \*buffer = (char \*)malloc(length + 1);

if (!buffer) {

perror("Unable to allocate buffer");

fclose(file);

return NULL;

}

fread(buffer, 1, length, file);

buffer[length] = '\0';

fclose(file);

return buffer;

}

int main() {

const char \*filename = "example.c";

char \*sourceCode = readFile(filename);

if (!sourceCode) {

return 1;

}

const char \*input = sourceCode;

Token token;

while ((token = getNextToken(&input)).type != TOKEN\_END) {

printf("Token Type: %s, Token Value: %s\n", getTokenTypeString(token.type), token.value);

}

free(sourceCode);

return 0;

}

**三、实验过程记录：**

1、在我刚开始代码版本中，在识别数字时不能够**识别小数**，改进之后可以识别。例如： double c=1.4;

改进前识别数字代码：

// 识别数字

if (isdigit(\*ptr)) {

int len = 0;

while (isdigit(\*ptr)) {

token.value[len++] = \*ptr++;

}

token.value[len] = '\0';

token.type = TOKEN\_NUMBER;

\*input = ptr;

return token;

}

改进后识别数字代码：

// 识别数字

if (isdigit(\*ptr) || \*ptr == '.') {

int len = 0;

int hasDot = 0;

while (isdigit(\*ptr) || (\*ptr == '.' && !hasDot)) {

if (\*ptr == '.') {

hasDot = 1;

}

token.value[len++] = \*ptr++;

}

token.value[len] = '\0';

token.type = TOKEN\_NUMBER;

\*input = ptr;

return token;

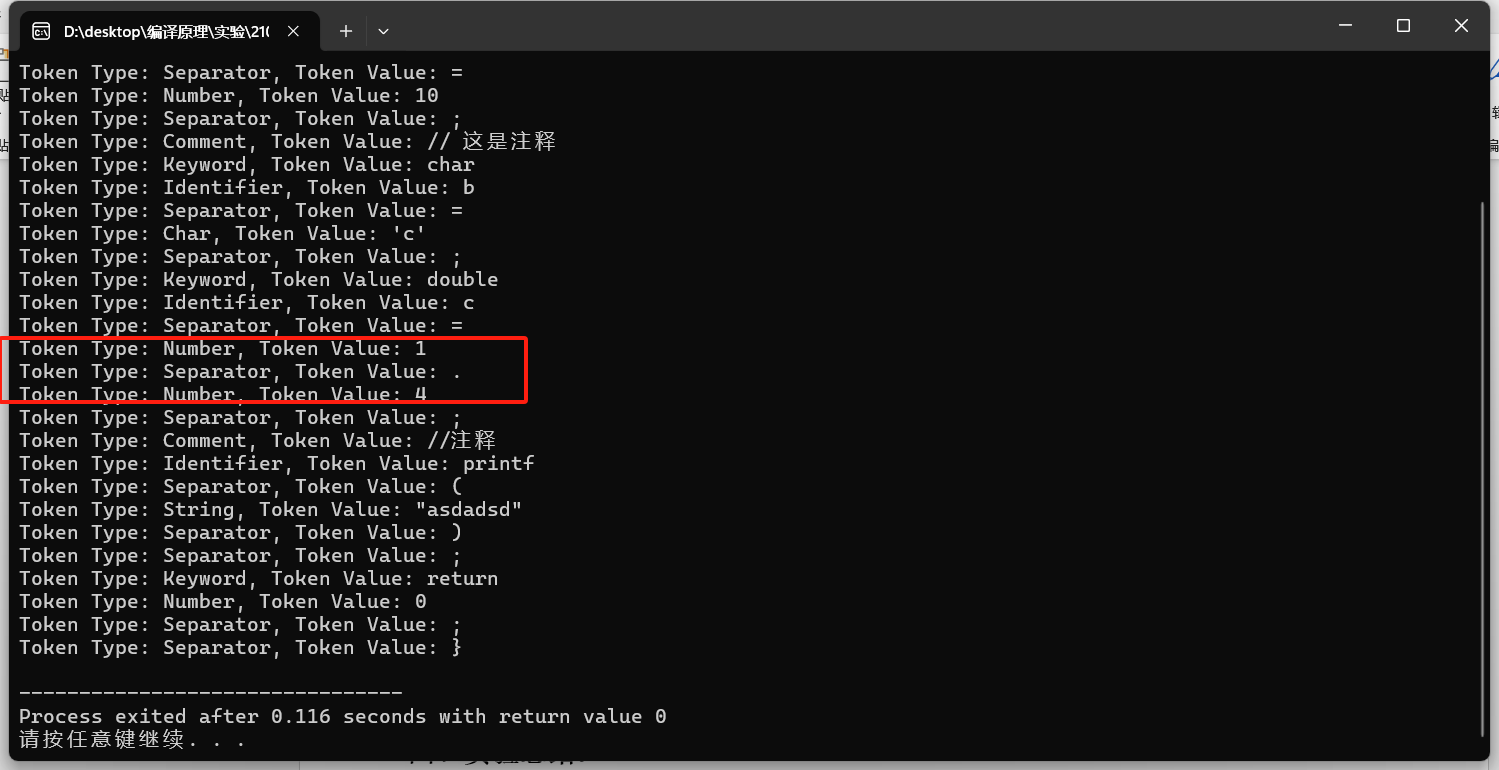
}

改进前后识别效果对比：

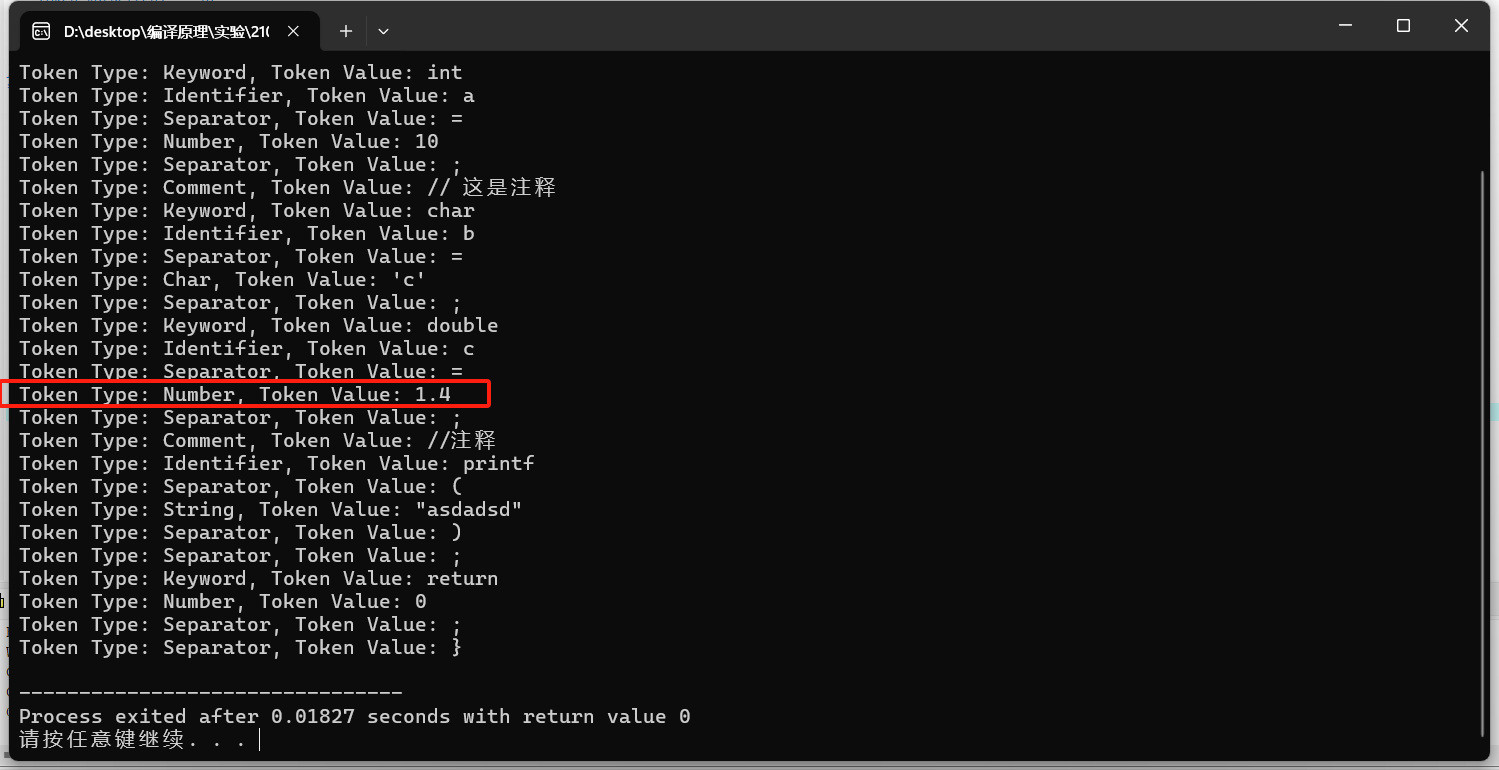
对于同一个句子：



改进前识别结果：把1.4拆分开了 识别失败！

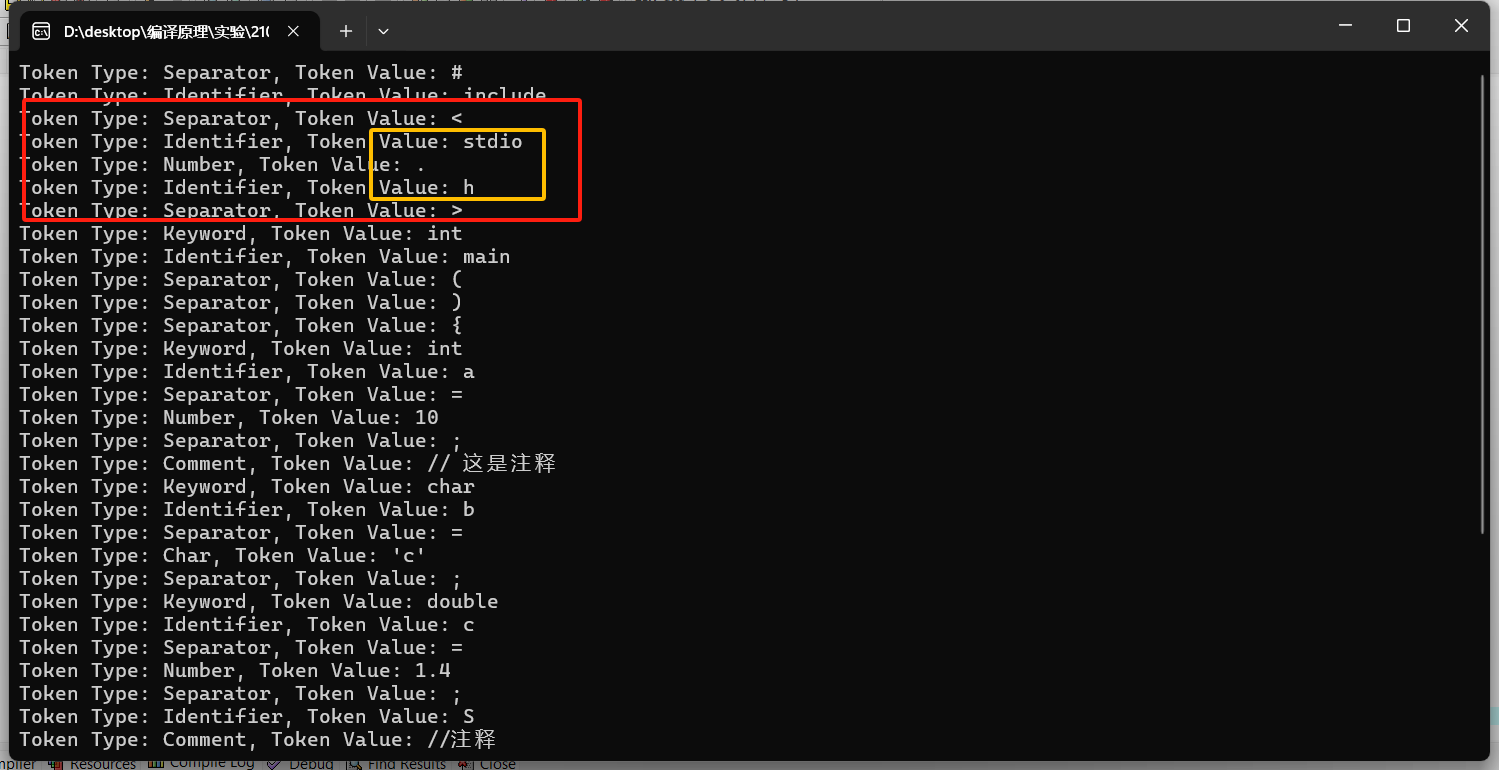


改进后：未拆分开，成功识别！



2、对于头文件的识别不准确的改进：

改进前：



改进后：增加新识别头文件的代码

// 识别头文件

if (\*ptr == '<') {

int len = 0;

token.value[len++] = \*ptr++; // 将 '<' 添加到 token 值中

while (\*ptr && \*ptr != '>') {

token.value[len++] = \*ptr++; // 将头文件名添加到 token 值中

}

if (\*ptr == '>') {

token.value[len++] = \*ptr++; // 将 '>' 添加到 token 值中

}

token.value[len] = '\0';

token.type = TOKEN\_STRING; // 头文件被视为字符串类型

\*input = ptr;

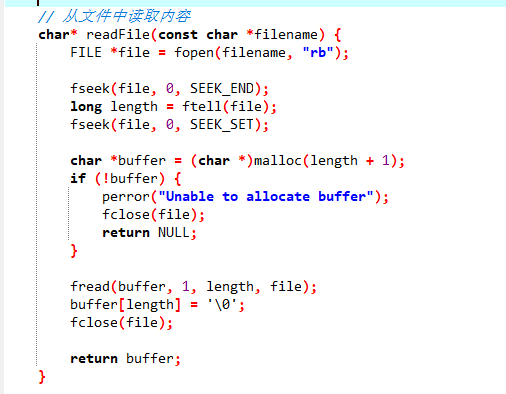
return token;

}

文本

描述已自动生成

3、空指针错误，未检查文件指针是否为空。



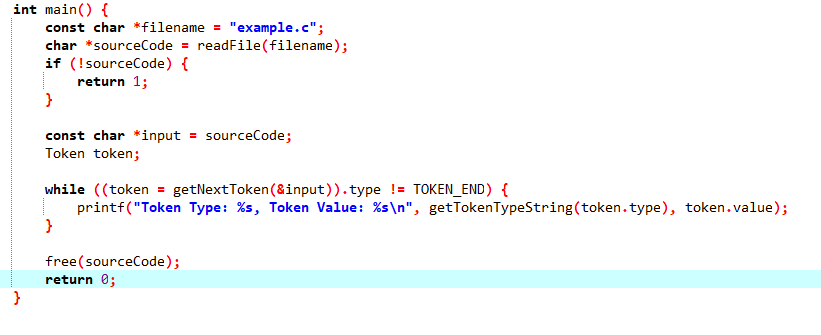
改进后：

文本, 白板

描述已自动生成

4、未释放资源：在分配内存后，函数返回前未释放内存和关闭文件，会导致内存泄漏和文件句柄泄漏。

改正后代码：

****

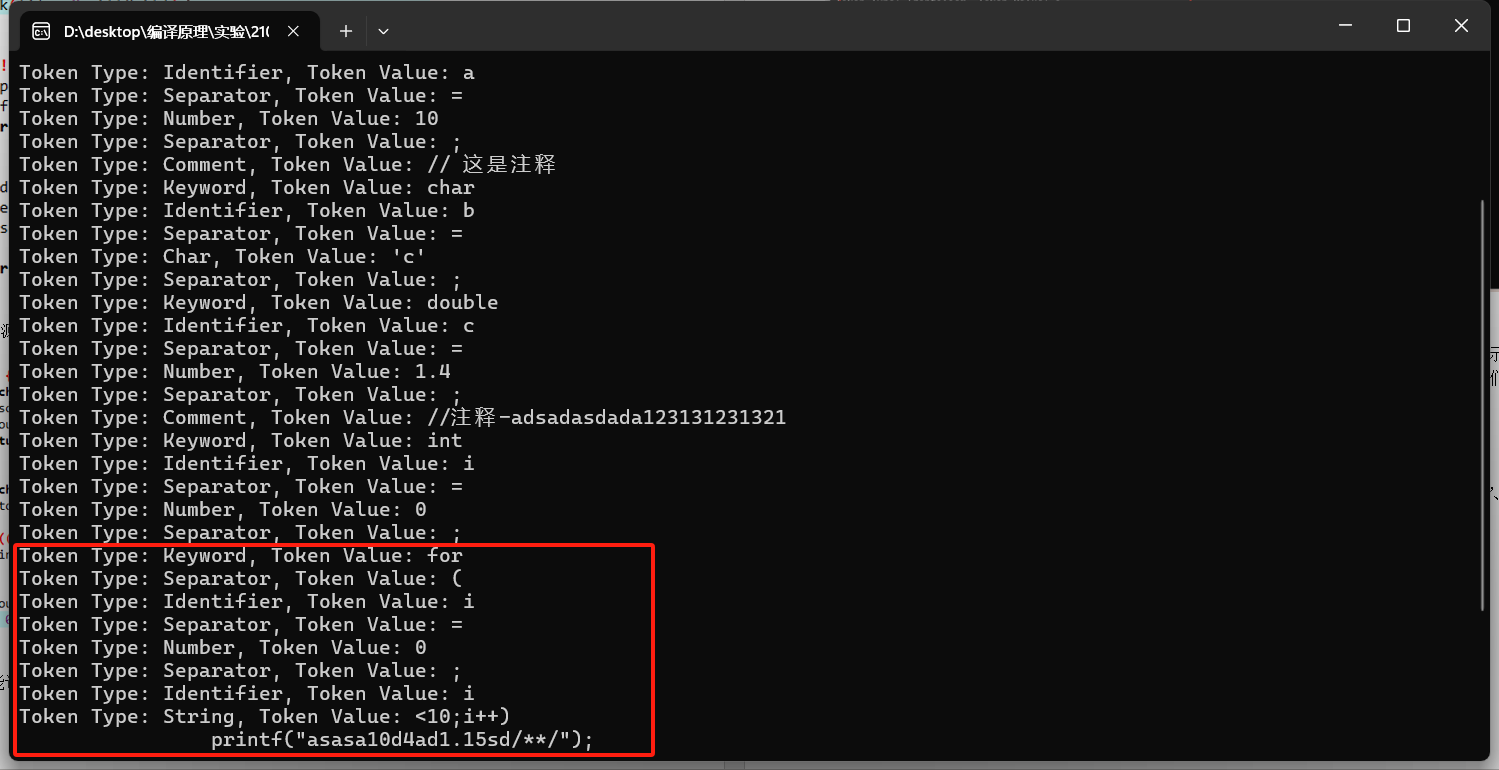
5.改进前不能识别循环，改进后可以识别循环

所要识别的代码：

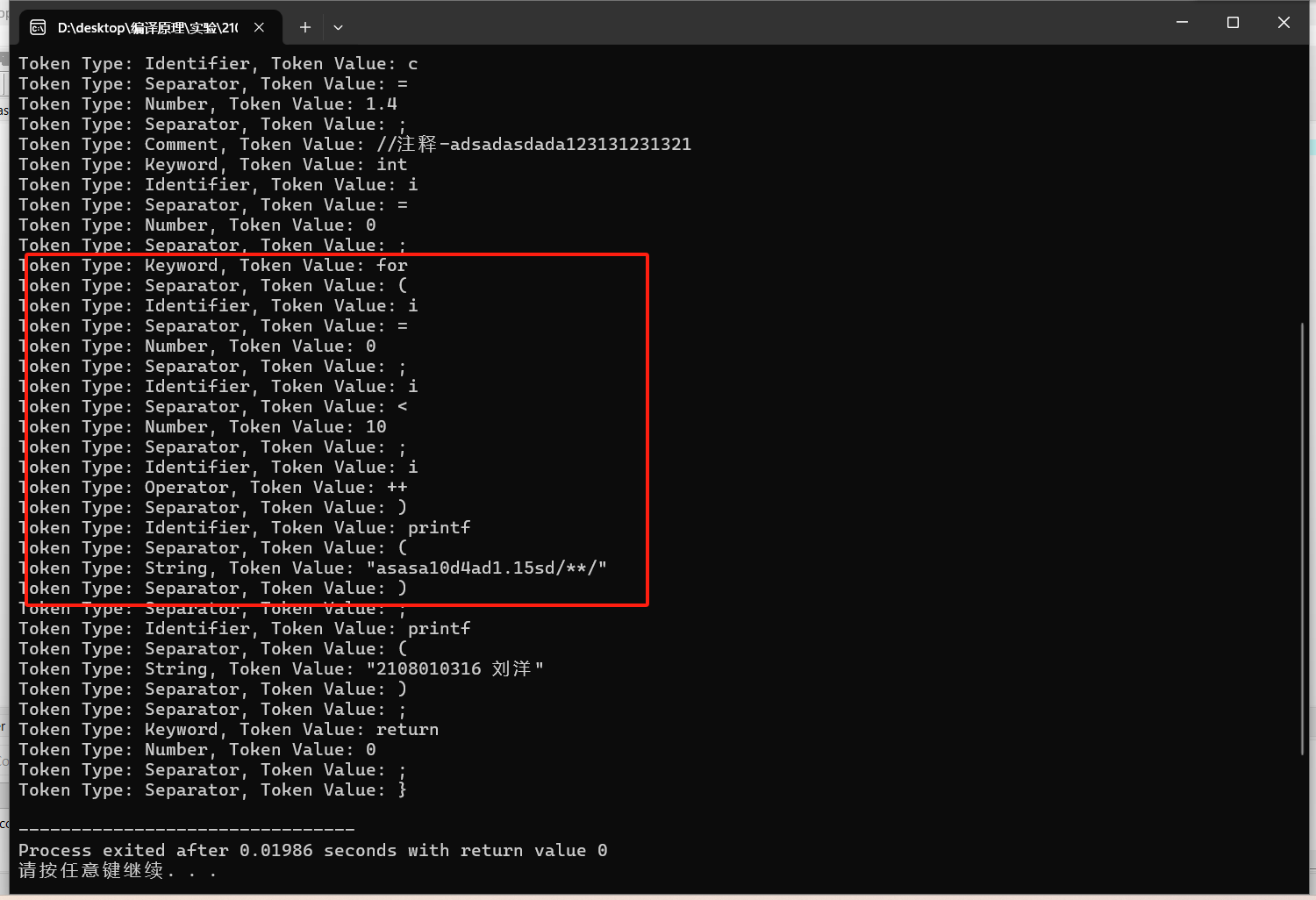
图形用户界面, 文本, 应用程序

描述已自动生成

改进前：



改进后：



6.头文件<stdio.h>中的 < 与算术运算符小于号< 冲突，在解决时，未将是算术运算符的情况进行处理，导致<以及之后的内容均被当做运算符。

错误代码 ：

// 识别标识符或关键字

if (isalpha(\*ptr) || \*ptr == '\_') {

int len = 0;

while (isalnum(\*ptr) || \*ptr == '\_') {

token.value[len++] = \*ptr++;

}

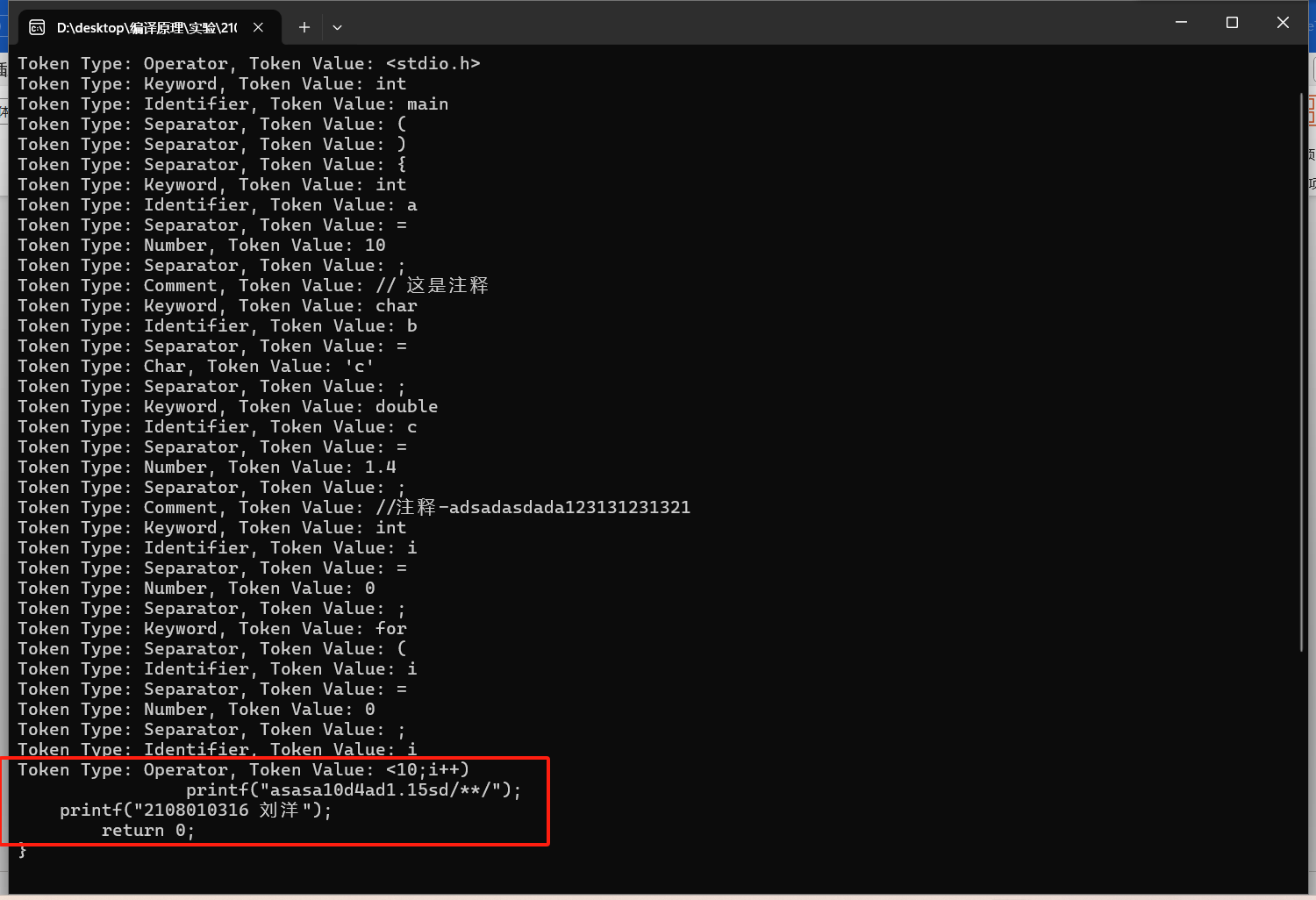
token.value[len] = '\0';

token.type = isKeyword(token.value) ? TOKEN\_KEYWORD : TOKEN\_IDENTIFIER;

\*input = ptr;

return token;

}



改正代码：

// 识别头文件

if (\*ptr == '<') {

int len = 0;

token.value[len++] = \*ptr++; // 将 '<' 添加到 token 值中

int isHeaderFile = 1; // 默认为头文件

while (\*ptr && \*ptr != '>') {

if (!isspace(\*ptr) && (\*(ptr + 1) == '\n' || \*(ptr + 1) == '\r' || \*(ptr + 1) == '\0')) {

isHeaderFile = 0; // 如果下一个字符是换行符或者文件结束符，认为不是头文件

}

token.value[len++] = \*ptr++; // 将头文件名添加到 token 值中

}

if (\*ptr == '>') {

token.value[len++] = \*ptr++; // 将 '>' 添加到 token 值中

}

token.value[len] = '\0';

token.type = isHeaderFile ? TOKEN\_STRING : TOKEN\_OPERATOR; // 根据是否头文件来设置类型

if(token.type==TOKEN\_OPERATOR){

while(len--)

{

ptr--;

}

ptr++;

token.value[0]='<';

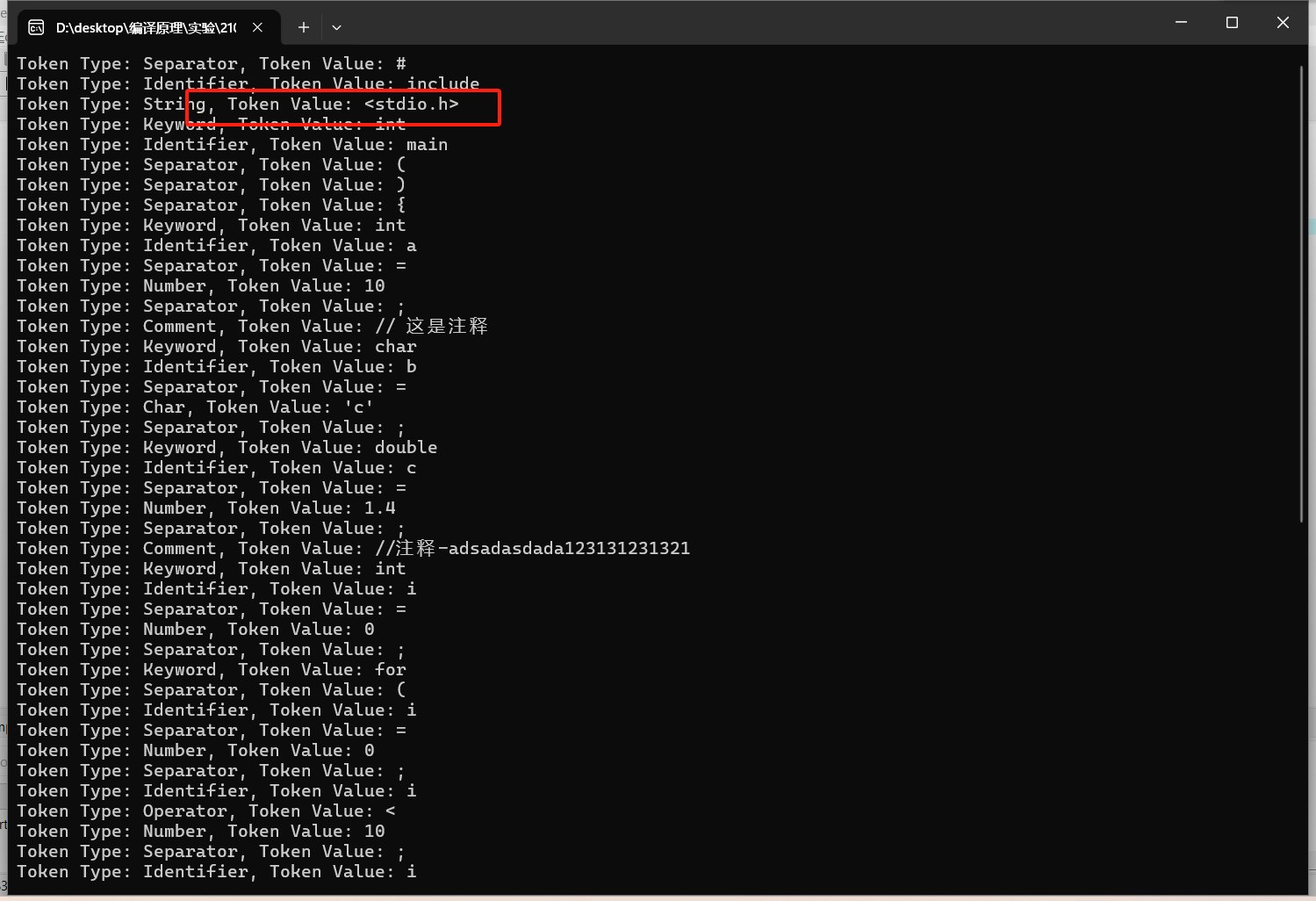
token.value[1]='\0';

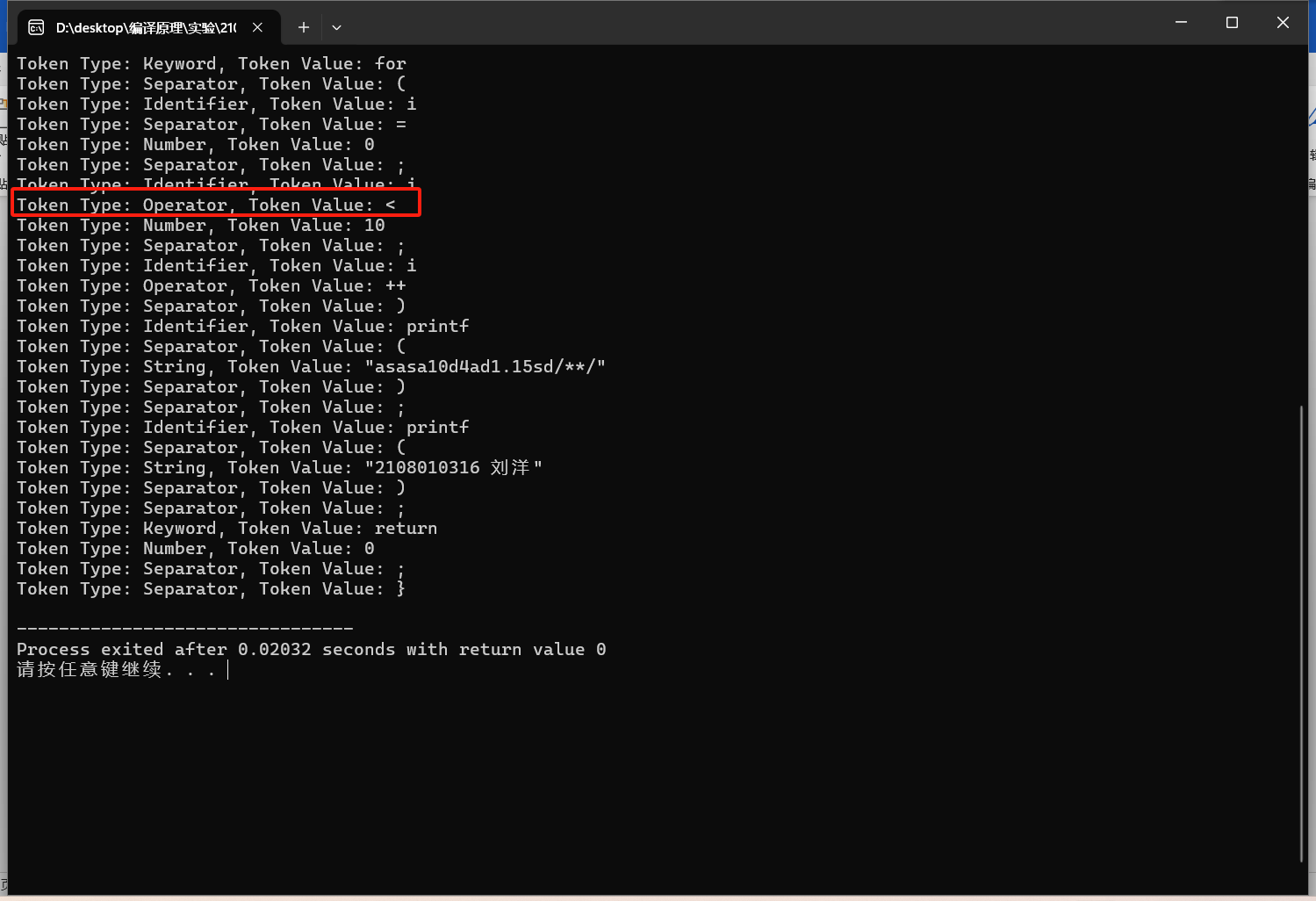
}

\*input = ptr;

return token;

}





**四、实验结果：**

通过运行词法分析器程序，我们能够正确识别并分类各种C语言的词法单元。实验中使用的示例源代码成功地被解析成一系列的Token，并在控制台输出了每个Token的类型和对应的值。这验证了我们实现的词法分析器的基本功能。

**五、实验总结：**

实验总结：

本次实验是实现一个简单的词法分析器，用于识别并区分C语言代码中的不同类型的单词（tokens），包括标识符、关键字、数字、运算符、分隔符、注释、字符串和字符常量等。在实验过程中，我们按照以下步骤进行了实验设计和代码实现：

1. 模块构建：定义并实现了主函数、扫描函数和拆分字符串函数等三个主要模块。在定义关键字列表、Token类型和结构等方面进行了设计。

2. 模块实现：实现了识别并分类标识符、关键字、数字、运算符、分隔符、注释、字符串和字符常量等的扫描函数，以及从文件读取源代码的函数。

3. 调用模块：在主函数中调用文件读取函数读取源代码，然后使用扫描函数逐个提取Token并输出Token的类型和值。

在实验过程中，我们遇到了一些问题，如对于小数的识别、头文件识别的不准确以及空指针错误等，但通过仔细分析和改进代码，成功解决了这些问题，并得到了正确的词法分析结果。

通过本次实验，我们深入理解了词法分析的基本原理和实现方法，提高了对C语言语法结构的理解和分析能力，加深了对编译原理的学习和掌握。同时，通过设计和实现词法分析器，也提升了我们的编程能力和问题解决能力。