编译原理与技术

词法分析程序

目录

2017-11-2

[裴子祥 计科七班 学号 2015211921]

[指导老师：刘辰]

一．实验内容与要求

题目：词法分析程序的设计与实现。

实验内容：设计并实现 C 语言的词法分析程序，要求如下：

1. 可以识别出用 C 语言编写的源程序中的每个单词符号，并以记号的形式输出每个单词符号。
2. 可以识别并读取源程序中的注释。
3. 可以统计源程序汇总的语句行数、单词个数和字符个数，其中标点和空格不计算为单词，并输出统计结果。
4. 检查源程序中存在的错误，并可以报告错误所在的行列位置。
5. 发现源程序中存在的错误后，进行适当的恢复，使词法分析可以继续进行，通过一次词法分析处理，可以检查并报告源程序中存在的所有错误。

方法：采用 C/C++作为实现语言，手工编写词法分析程序。

实验环境：

MICSOFT WINDOW 10

Visual Studio 2015

二． 程序设计与实现

1． 主要步骤

首先，描述 C 语言各种单词符号的词法规则；其次，构造状态转换图；然后，构造词法分析程序。

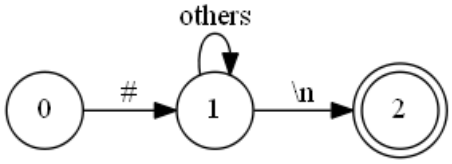
2． 词法规则

C 语言定义记号及单词：

(1) 预编译命令
(2) 标识符
(3) 关键字
(4) 无符号数
(5) 字符常量
(6) 字符串常量
(7) 运算符与标号
(8) 注释，行注释、块注释
(9) 错误处理

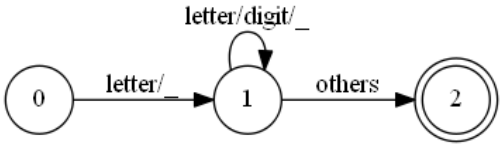
3． 状态转移图 DFA

(1) 预编译命令



预编译命令在词法分析编译前就已经分析了，但这里，还是读出显示，并不计算为单词量。

(2) 标识符



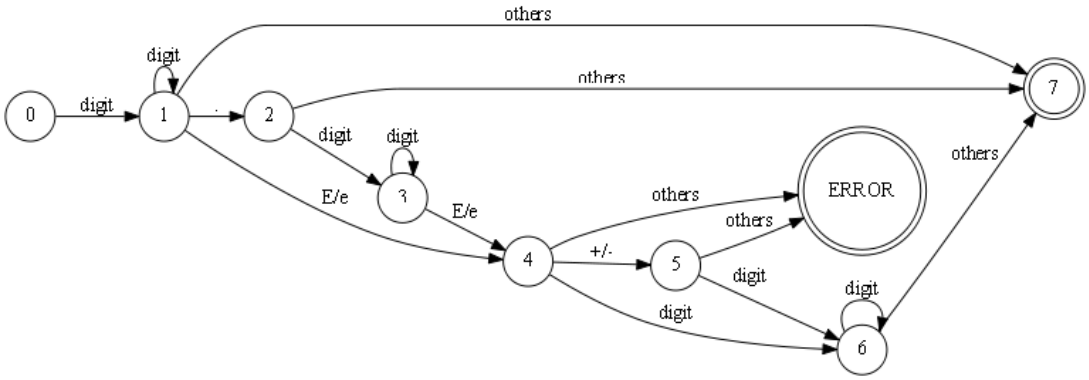
ID 结构体记录标识符

```
typedef struct id {
    string name;        //标识符名字
    int pos;            //标识符入口地址，id数组中的偏移位置，即下标
}ID;
```

(3) 关键字

void	char	int	short	float	double
long	unsigned	struct	union	enum	typedef
sizeof	auto	static	register	extern	const
volatile	return	continue	break	goto	if
else	switch	case	default	for	do
while					

(4) 无符号数



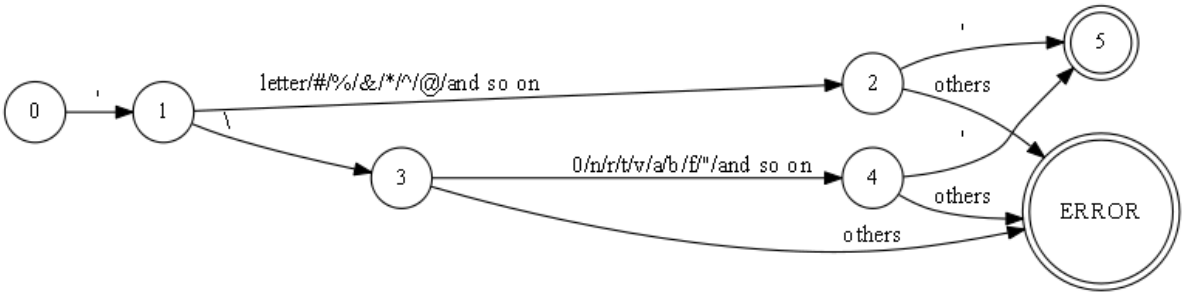
错误处理：

不完整数字常量错误。

E 后面没有数字，+/-号后面没有数字。

如:1.234E、25E+、是错误的，需要对其报错。

(5) 字符常量

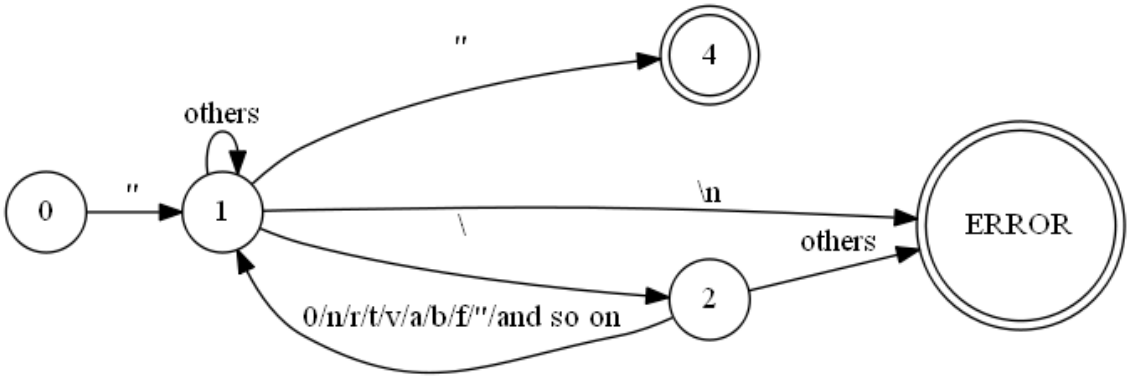


错误处理：

单引号没有闭合、引号内字符数量不只一个、转义字符出错。

如：'az'、'a;、'\q'、是错误的，需要报错。

(6) 字符串常量



错误处理：

不完字符串常量错误。

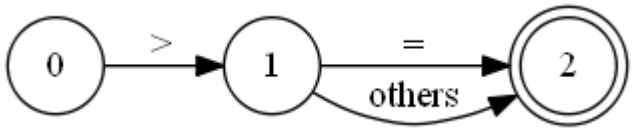
会出现字符串中间出现回车隔断、双引号不闭合。两者可归为双引号不闭合。

如:"abcd

efgh"、"abcdefg;、是错误的，需要对其报错。

(7) 运算符与标号

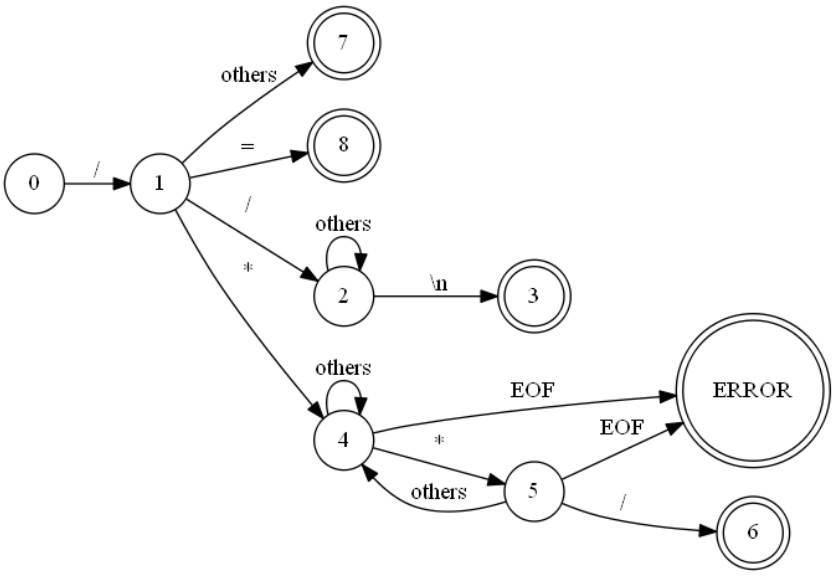
比较简单，有需要超前识别的情况，例如>、>=



bound_op	界符	, ( ) [ ] { } ;
Assign_op	赋值运算符	= += -= /= &= *= |= ^=

Logic_op	逻辑运算符	&& || !
Relop	关系运算符	> < == != >= <=
Arith_op	算术运算符	+ ++ - -- * / %
Bitwise_op	位运算符	& | ~ ^ >> <<
Other_op	其它符号	? : . ->

（8） 注释，行注释、块注释、除号、赋值运算



错误处理：

块注释不完整错误。

如:/*这是一条注释、/*这是第二条注释*、是错误的，需要对其报错。

（9） 错误处理

ERROR 结构体记录错误

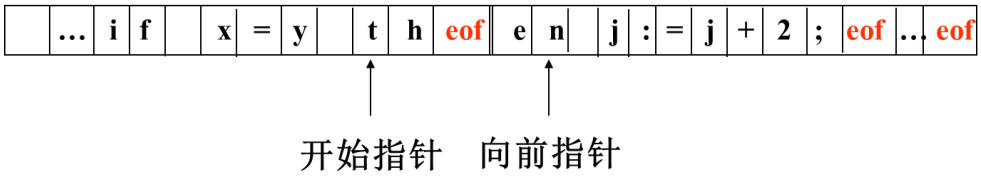
```
typedef struct error {
    int lineNum;        //错误发生行数
    int kind;           //错误类型
    /**
    *kind=0;识别到非法符号,如$、/等
    *kind=1;识别到无符号数的错误,
    *kind=2;识别字符常量的错误
    *kind=3,识别字符串常量的错误
    *kind=4,识别注释快不闭包的错误
    */
    string err;         //错误输出
}ERROR;
```

4． 程序构造

(1) 输入文件

C 语言源程序存放在 test.txt 文件中。

增加结束标记的配对输入缓冲区的应用。




```
const int bufferSize = 1000;//半个缓冲区大小
char bufferL[bufferSize];//配对缓冲区左半区
char bufferR[bufferSize];//配对缓冲区右半区
```

(2) 输出文件形式部分举例

输出重定向，将标准输出 stdout。输出至“output.txt”文件中，以便查阅

正规表达式	记号	属性
Reserve（关键字）	Reserve	-
ID	id	符号表入口指针（数组偏移量）
Num	NUM	常数值
<	Relop	LT
<=	Relop	LE
=	Assign_op	-
^=	Assign_op	-
--	Arith_op	-
%	Arith_op	-
(	Bound_op	-
;	Bound_op	-
!	Logic_op	NOT
&&	Logic_op	AND
||	Logic_op	OR

&	Bitwise_op	-
>>	Bitwise_op	-
?	Other_op	?
:	Other_op	:
“…”	Const string	…
‘\t’	Const char	\t
‘a’	Const char	a

————————————————- Statistics:

总行数、总字符数、总单词数、各种类运算符数、各种类单词数

————————————————- Errors:

错误行数、错误单词、错误原因种类

(3) 全局变量

```
int state = 0;                  //自动机状态,初始态
char C = ' ';                       //当前读入字符
int iskey;                  //值为-1为自定义标识符,否则是关键字
string token = "";              //当前正在识别的单词，初始为空
char *lexemebegin;          //字符指针指向输入缓冲区中当前单词的开始位置
char *forwardptr;           //向前扫描指针
char bufferL[bufferSize];//配对缓冲区左半区
char bufferR[bufferSize];//配对缓冲区右半区

int lineCur = 1;                //统计当前行数，初始为0
int charSum = 0;                //字符总数

int relopCount = 0;             //关系运算符个数
int assign_opCount = 0;         //赋值运算符个数
int logic_opCount = 0;          //逻辑运算符个数
int bound_opCount = 0;          //界符
int arith_opCount = 0;          //算术运算符
int bitwise_opCount = 0;            //位运算符
int other_opCount = 0;          //其它运算符

int keywordCount = 0;           //关键字个数
int numCount = 0;               //常数个数
int stringCount = 0;            //字符串常量个数
int constcharCount = 0;         //字符常量字符个数
int idCount = 0;                //标识符个数
int wordCount = 0;              //单词个数
```

```
vector<ID>  idTable;    //标识符表
vector<ERROR> errorTable;//报错信息表
```

(4) 函数与过程

```
//读字符过程,每调用一次,从输入缓冲区中读一个字符,放入全局变量C中,forward指向下一个指针
void get_char();

//过程,检查C是否为空字符,若是，则反复调用，直到C为非空字符
void get_nbc();

//过程,把C中字符与token连接起来
void cat();

//布尔函数,判断C是否为字母,是返回true
bool isletter();

//布尔函数,判断C是否为数字,是返回true
bool isdigit();

//过程,向前扫描指针forward后退一个字符
void retract();

//查关键字表,若此函数返回值为0,表示token是标识符,否则是关键字
int reserve();

//将标识符插入符号表,返回插入位置
int idTable_insert();

//错误处理
void error(int kind);

//自动机词法分析主体
void DFA();

//打印输出词法分析结果
void printRes();
```

5． 样例结果

(1) 示例，为了体现词法分析器的效果，手动加入错误，与符号标识，输入 C 语言文件没有逻辑，test.txt 中 C 语言源程序如下:

```
(1)  #include<stdio.h>
```

```
(2)  struct p{
(3)  int a;
(4)  int b;
(5)  }P
(6)  int main()
(7)  {
(8)  int a=0;
(9)  a=$+1;
(10) char b='a';
(11) b='az;
(12) b='\z';
(13) int ++a;
(14) char[21] c="aad
(15) a";
(16) int c;
(17) c^=b;
(18) b=c^a;
(19) //hahahahaa
(20) /*
(21) 这是一个注释
(22) */
(23) int p[10];
(24) a>>2;
(25) a+=2;
(26) if(a<=2)
(27) printf("sad%cd",b);
(28) b--;
(29) char* ptr;
(30) P->b;
(31) int m=(a==2)?b:c;
(32) string="sadad";
(33) char d='s';
(34) d='\t';
(35) return 0;
(36) }
(37) /*sss
```

(2) 结果：输出至 output.txt 文件中，内容如下：

1	#include<stdio.h> <预编译命令,->
2	struct <struct,->
3	p <ID,0>
4	{ <bound_op,{>
5	int <int,->

6	a <ID,1>
7	; <bound_op,;>
8	int <int,->
9	b <ID,2>
10	; <bound_op,;>
11	} <bound_op,}>
12	P <ID,3>
13	int <int,->
14	main<ID,4>
15	(<bound_op,(>
16	) <bound_op,)>
17	{ <bound_op,{>
18	int <int,->
19	a <ID,1>
20	= <assign_op,=>
21	0 <NUM,0>
22	; <bound_op,;>
23	a <ID,1>
24	= <assign_op,=>
25	+ <arith_op,+>
26	1 <NUM,1>
27	; <bound_op,;>
28	char<char,->
29	b <ID,2>
30	= <assign_op,=>
31	'a' <CONST CHAR,'a'>
32	; <bound_op,;>
33	b <ID,2>
34	= <assign_op,=>
35	b <ID,2>
36	= <assign_op,=>
37	; <bound_op,;>
38	int <int,->
39	++ <arith_op,++>
40	a <ID,1>
41	; <bound_op,;>
42	char<char,->
43	[<bound_op,[>
44	21 <NUM,21>
45	] <bound_op,]>
46	c <ID,5>
47	= <assign_op,=>
48	a <ID,1>
49	int <int,->

50	c <ID,5>
51	; <bound_op,;>
52	c <ID,5>
53	^= <assign_op,^=>
54	b <ID,2>
55	; <bound_op,;>
56	b <ID,2>
57	= <assign_op,=>
58	c <ID,5>
59	^ <bitwise_op,^>
60	a <ID,1>
61	; <bound_op,;>
62	int <int,->
63	p <ID,0>
64	[<bound_op,[>
65	10 <NUM,10>
66	] <bound_op,]>
67	; <bound_op,;>
68	a <ID,1>
69	>> <bitwise_op,>>>
70	2 <NUM,2>
71	; <bound_op,;>
72	a <ID,1>
73	+= <assign_op,+=>
74	2 <NUM,2>
75	; <bound_op,;>
76	if <if,->
77	(<bound_op,(>
78	a <ID,1>
79	<= <relop,LE>
80	2 <NUM,2>
81	) <bound_op,)>
82	printf <ID,6>
83	(<bound_op,(>
84	"sad%cd" <CONST STRING,"sad%cd">
85	, <bound_op,,>
86	b <ID,2>
87	) <bound_op,)>
88	; <bound_op,;>
89	b <ID,2>
90	-- <arith_op,-->
91	; <bound_op,;>
92	char <char,->
93	* <arith_op,*>

94	ptr <ID,7>
95	; <bound_op,;>
96	P <ID,3>
97	-> <other_op,->>
98	b <ID,2>
99	; <bound_op,;>
100	int <int,->
101	m <ID,8>
102	= <assign_op,=>
103	(<bound_op,(>
104	a <ID,1>
105	== <relop_op,EQ>
106	2 <NUM,2>
107	) <bound_op,)>
108	? <other_op,?>
109	b <ID,2>
110	: <other_op,:>
111	c <ID,5>
112	; <bound_op,;>
113	string <ID,9>
114	= <assign_op,=>
115	"sadad" <CONST STRING,"sadad">
116	; <bound_op,;>
117	char<char,->
118	d <ID,10>
119	= <assign_op,=>
120	's' <CONST CHAR,'s'>
121	; <bound_op,;>
122	d <ID,10>
123	= <assign_op,=>
124	'\t'<CONST CHAR,'\t'>
125	; <bound_op,;>
126	return <return,->
127	0 <NUM,0>
128	; <bound_op,;>
129	} <bound_op,}>
130	----------
131	- Statistics:
132	totoal line numbers: 37
133	total char Sum:359
134	total word Sum:86
135	---
136	relation operator: 2
137	assign operator:13

138	logic operator:0
139	bound operator:39
140	arith operator:4
141	bitwise operator:2
142	other operator:3
143	---
144	keyword Sum:15
145	const numbers: 9
146	const strings: 2
147	ids:36
148	kinds of ids:11
149	----------
150	- Errors:
151	line Error
152	9 [Error]$ Invalid character
153	11 [Error]'az; The const char is wrong!
154	12 [Error]'\z' The const char is wrong!
155	14 [Error]"aad The const string lose a " wrongly!
156	15 [Error]"; The const string lose a " wrongly!
157	37 [Error]/ The last line is wrong.

(3) 结果分析

前面部分为 test.txt 文件中 C 语言源代码的逐字分析，后面加粗部分是信息统计信息。

经过对比，该信息输出结果是正确的。

程序能识别出全部 C 语言的运算符，（或者是一个较大的子集，尽可能周期，但总怕有疏漏），能够识别出一些较为生僻的运算符如？、>>、->、.、^、:、等等。

总行数：37 与源文件对应

总字符数：359

总单词数：86 （这里单词定义为除了界符与空白字符的所有正则表达式识别出的，即自定义标识符，关键字，运算符，字符常量，数字常量，字符串常量的集合）

其它各种类单词数：详见上表。

一共识别出 5 种错误:

第 9 行，出现非法字符$

第 11 行，字符常量出错，单引号没有闭合

第 12 行，字符常量出错，错误的转义字符不能被识别

第 14、15 行，字符串常量出错，双引号没有闭合，或双引号之间中间存在换行符隔断。

第 37 行，块注释未闭合，导致一直使注释头包含到文件尾（最后一行）

6． 实验总结

这是编译原理与技术第一次程序设计实验，程序设计极大的参考并完善了课

本上所给出的有限状态自动机算法，还有输入输出形式，并对 C 语言的更多内容进行扩展添加，为了尽可能的周全地识别 C 语言中的单词（如许多不常用的运算符->、？、^等），最终的自动机状态数较为之前有很大的添加（共 30 个状态），代码行数也突破 1100 行，并尽可能使得代码具有良好的规范性与可读性，当程序能够成功运行并得出理想的结果，这是令人欢欣鼓舞的！！

实验过程中，使用 graphziv 与 dot 语言对有限状态自动机进行绘制，这也是实验的意外收获，极大地提高效率和美观。对于词法分析的错误处理，一开始是拿捏不准的，因为大多数错误都是在语法分析中实现的，词法分析能够找到的错误是有限的，我定义了 5 类错误在全局变量定义的 ERROR，分别是识别非法字符、无符号数、字符常量、字符串常量、块注释中发生错误的情况，也能够检许多错。

这次实验获益匪浅，也极大地激励了学习编译原理的热情，对课本内容的不断理解与探索，对 debug 的一丝不苟，都是对后面的学习将是极大的鼓励。

7． 源码附件

```
/**
*程序使用与其它信息，
*详见 readme.txt 文件
*/

#include<iostream>
#include<vector>
#include<string>
#include<cstdlib>
#include<fstream>
using namespace std;

ifstream fptr("test.txt");//文件输入流

const int bufferSize = 1000;//半个缓冲区大小

typedef struct id {
    string name;        //标识符名字
    int pos;           //标识符入口地址
}ID;

typedef struct error {
    int lineNum;        //错误发生行数
    int kind;          //错误类型
    /**
    *kind=0;识别到非法符号,如$、/等
    *kind=1;识别到无符号数的错误，
    *kind=2;识别字符常量的错误
    *kind=3,识别字符串常量的错误
    *kind=4,识别注释快不闭包的错误
    */
    string err;         //错误输出
}ERROR;

int state = 0;             //自动机状态,初始态
char C = ' ';                 //当前读入字符
int iskey;             //值为-1 为自定义标识符,否则是关键字
string token = "";          //当前正在识别的单词，初始为空
char *lexemebegin;      //字符指针指向输入缓冲区中当前单词的开始位置
char *forwardptr;        //向前扫描指针
char bufferL[bufferSize];//配对缓冲区左半区
char bufferR[bufferSize];//配对缓冲区右半区
```

```
int lineCur = 1;           //统计当前行数，初始为 0
int charSum = 0;           //字符总数

int relopCount = 0;         //关系运算符个数
int assign_opCount = 0;     //赋值运算符个数
int logic_opCount = 0;      //逻辑运算符个数
int bound_opCount = 0;      //界符
int arith_opCount = 0;      //算术运算符
int bitwise_opCount = 0;        //位运算符
int other_opCount = 0;      //其它运算符

int keywordCount = 0;       //关键字个数
int numCount = 0;           //常数个数
int stringCount = 0;        //字符串常量个数
int constcharCount = 0;     //字符常量字符个数
int idCount = 0;            //标识符个数
int wordCount = 0;          //单词个数


vector<ID>  idTable;    //标识符表
vector<ERROR> errorTable;//报错信息表

//读字符过程,每调用一次,从输入缓冲区中读一个字符,放入全局变量 C 中,forward 指向下一个指针
void get_char();

//过程,检查 C 是否为空字符,若是，则反复调用，直到 C 为非空字符
void get_nbc();

//过程,把 C 中字符与 token 连接起来
void cat();

//布尔函数,判断 C 是否为字母,是返回 true
bool isletter();

//布尔函数,判断 C 是否为数字,是返回 true
bool isdigit();

//过程,向前扫描指针 forward 后退一个字符
void retract();

//查关键字表,若此函数返回值为 0,表示 token 是标识符,否则是关键字
int reserve();

//将标识符插入符号表,返回插入位置
int idTable_insert();
```

```
//错误处理
void error(int kind);

//自动机词法分析主体
void DFA();

//打印输出词法分析结果
void printRes();

int main()
{
    cout << "词法分析结果详见 output.txt 文件" << endl;
    FILE *outfptr;
    freopen_s(&outfptr, "output.txt", "w", stdout);//标准输出输出重定向到 output.txt 文件中

    bufferL[bufferSize - 1] = -1;//左半缓冲区末尾哨兵
    bufferR[bufferSize - 1] = -1;//右半缓冲区末尾哨兵
    lexemebegin = bufferL;
    forwardptr = bufferL;

    if (!fptr.is_open())
    {//未找到文件
        cout << "ERROR OPENING THE SOURCR FILE";
        exit(-1);
    }

    for (int i = 0; i < bufferSize - 1; ++i)
    {
        bufferL[i] = fptr.get();
        charSum++;
        //cout << bufferL[i];
        if (bufferL[i] == EOF)
        {
            break;
        }
    }

    while (C != EOF)//到文件结束
    {
        DFA();
    }

    printRes();//打印输出词法分析结果
```

```
    forwardptr=NULL;
    lexemebegin=NULL;
    fptr.close();
    system("pause");
    return 0;
}

//读字符过程,每调用一次,从输入缓冲区中读一个字符,放入全局变量 C 中,forward 指向下一个指针
void get_char()
{
    if ((*forwardptr) == -1)
    {
        if ((forwardptr == bufferL + bufferSize - 1))
        {
            for (int i = 0; i < bufferSize - 1; ++i)
            {
                bufferR[i] = fptr.get();//填入右半缓冲区
                charSum++;
                if (bufferR[i] == EOF)//读到文件尾
                    break;
            }
            forwardptr = bufferR;
        }
        else if (forwardptr == bufferR + bufferSize - 1)
        {
            for (int i = 0; i < bufferSize - 1; ++i)
            {
                bufferL[i] = fptr.get();//填入左半缓冲区
                charSum++;
                if (bufferL[i] == EOF)//读到文件尾
                    break;
            }
        }
        else
        {
            fptr.close();
        }
    }
    C = *forwardptr;
    forwardptr++;
}

//过程,检查 C 是否为空字符,若是，则反复调用，直到 C 为非空字符
void get_nbc()
{
```

```
    while (' ' == C || '\t' == C || '\n' == C)
    {
        if (C == '\n')
        {
            lineCur++;
        }
        get_char();
    }
}

//过程,把 C 中字符与 token 连接起来
void cat()
{
    token = token + C;
}

//布尔函数,判断 C 是否为字母,是返回 true
bool isletter()
{
    if ((C >= 'A'&&C <= 'Z') || (C >= 'a'&&C <= 'z'))
        return true;
    else
        return false;
}

//布尔函数,判断 C 是否为数字,是返回 true
bool isdigit()
{
    if (C >= '0'&&C <= '9')
        return true;
    else
        return false;
}

//过程,向前扫描指针 forward 后退一个字符
void retract()
{
    if (forwardptr == bufferL)
    {
        forwardptr = bufferR + bufferSize - 1;
    }
    else if (forwardptr == bufferR)
    {
        forwardptr = bufferL + bufferSize - 1;
    }
```

```
    else
    {
        forwardptr--;
    }
    //C = *forwardptr;
}

//查关键字表,若此函数返回值为 0,表示 token 是标识符,否则是关键字
int reserve()
{
    if (token == "void" || token == "char" || token == "int" || token == "short"
        || token == "float" || token == "double" || token == "long" || token == "signed"
        || token == "unsigned" || token == "struct" || token == "union" || token ==
"enum"
        || token == "typedef" || token == "sizeof" || token == "auto" || token ==
"static"
        || token == "register" || token == "extern" || token == "const" || token ==
"volatile"
        || token == "return" || token == "continue" || token == "break" || token ==
"goto"
        || token == "if" || token == "else" || token == "switch" || token == "case"
        || token == "default" || token == "for" || token == "do" || token == "while")
    {
        return 1;
    }
    else
    {
        return -1;
    }
}

//将标识符插入符号表,返回插入位置
int idTable_insert()
{
    int len = idTable.size();
    for (int i = 0; i < len; ++i)
    {
        if (idTable[i].name == token)
        {
            return i;                    //返回标识符入口地址
        }
    }
    ID newId;
    newId.name = token;
    newId.pos = len;
```

```
    idTable.push_back(newId);
    return len;
}

//错误处理
void error(int kind)
{
    ERROR newError;
    newError.lineNum = lineCur;
    newError.kind = kind;
    newError.err = token;
    errorTable.push_back(newError);
    lexemebegin = forwardptr;
}

void printRes()
{
    wordCount = idCount + keywordCount + stringCount + numCount + relopCount +
assign_opCount + logic_opCount + arith_opCount + bitwise_opCount + other_opCount;
    cout << "----------------------------\n- Statistics:" << endl;
    cout << "totoal line numbers: " << lineCur << endl;
    cout << "total char Sum:" << charSum << endl;
    cout << "total word Sum:" << wordCount << endl;
    cout << "---" << endl;
    cout << "relation operator: " << relopCount << endl;
    cout << "assign operator:" << assign_opCount << endl;
    cout << "logic operator:" << logic_opCount << endl;
    cout << "bound operator:" << bound_opCount << endl;
    cout << "arith operator:" << arith_opCount << endl;
    cout << "bitwise operator:" << bitwise_opCount << endl;
    cout << "other operator:" << other_opCount << endl;
    cout << "---" << endl;
    cout << "keyword Sum:" << keywordCount << endl;
    cout << "const numbers: " << numCount << endl;
    cout << "const strings: " << stringCount << endl;
    cout << "ids:" << idCount << endl;
    cout << "kinds of ids:" << idTable.size() << endl;


    if (errorTable.empty())
    {
        cout << "----------------------------\n- Errors: 0" << endl;
    }
    else
    {
```

```
        cout << "----------------------------\n- Errors:" << endl;
        cout << "line" << '\t' << "Error" << endl;
        int n = errorTable.size();
        for (int i = 0; i < n; ++i)
        {
            switch (errorTable[i].kind)
            {
            case 0:                 //第一种错误，语句中出现非法字符，如$、\等
                cout << errorTable[i].lineNum << '\t' << "[Error]" << errorTable[i].err
<< "  Invalid character" << endl;
                break;
            case 1:                 //第二种错误，文件结束末尾出错，比如注释未闭合
                cout << errorTable[i].lineNum << '\t' << "[Error]" << errorTable[i].err
<< "  The last line is wrong." << endl;
                break;
            case 2:                 //第三种错误，无符号数出错，E 后面没跟数字或+、-号，或+、-号后面没
跟数字
                cout << errorTable[i].lineNum << '\t' << "[Error]" << errorTable[i].err
<< "  The const num is wrong." << endl;
                break;
            case 3:                 //第四种错误，字符常量出错，超过一个字符，或转义字符出错
                cout << errorTable[i].lineNum << '\t' << "[Error]" << errorTable[i].err
<< "  The const char is wrong!" << endl;
                break;
            case 4:                 //第五种错误，块注释符号不闭包的情况
                cout << errorTable[i].lineNum << '\t' << "[Error]" << errorTable[i].err
<< "  The const string lose a \" wrongly!" << endl;
                break;
            default:
                cout << errorTable[i].lineNum << '\t' << "[Error]" << "Unknown Error!" <<
endl;
            }

        }
    }
}

void DFA()
{
    switch (state)
    {
    case 0:
        get_char();
        get_nbc();
        if (isletter() || C == '_')     //标识符
```

```
        {
            state = 2;
        }
        else if (isdigit())
        {
            state = 3;
        }
        else
        {
            switch (C)
            {
            case '#':
                state = 1;
                break;
            case '\'':
                state = 9;
                break;
            case '"':
                state = 12;
                break;
            case '/':
                state = 14;
                break;
            case '<':
                state = 19;
                break;
            case '>':
                state = 20;
                break;
            case '!':
                state = 21;
                break;
            case '&':
                state = 22;
                break;
            case '|':
                state = 23;
                break;
            case '~':
                state = 24;
                break;
            case '^':
                state = 25;
                break;
            case '%':
```

```
            state = 26;
            break;
        case '*':
            state = 27;
            break;
        case '+':
            state = 28;
            break;
        case '-':
            state = 29;
            break;
        case '=':
            state = 30;
            break;
        case '?':
            state = 0;
            cout << '?' << '\t' << '<' << "other_op" << ',' << '?' << '>' << endl;
            other_opCount++;
            lexemebegin = forwardptr;
            break;
        case ':':
            state = 0;
            cout << ':' << '\t' << '<' << "other_op" << ',' << ':' << '>' << endl;
            other_opCount++;
            lexemebegin = forwardptr;
            break;
        case '.':
            state = 0;
            cout << '.' << '\t' << '<' << "other_op" << ',' << '.' << '>' << endl;
            other_opCount++;
            lexemebegin = forwardptr;
            break;
        case '(':
            state = 0;
            cout << '(' << '\t' << '<' << "bound_op" << ',' << '(' << '>' << endl;
            bound_opCount++;
            lexemebegin = forwardptr;
            break;
        case ')':
            state = 0;
            cout << ')' << '\t' << '<' << "bound_op" << ',' << ')' << '>' << endl;
            bound_opCount++;
            lexemebegin = forwardptr;
            break;
        case '{':
```

```
            state = 0;
            cout << '{' << '\t' << '<' << "bound_op" << ',' << '{' << '>' << endl;
            bound_opCount++;
            lexemebegin = forwardptr;
            break;
        case '}':
            state = 0;
            cout << '}' << '\t' << '<' << "bound_op" << ',' << '}' << '>' << endl;
            bound_opCount++;
            lexemebegin = forwardptr;
            break;
        case '[':
            state = 0;
            cout << '[' << '\t' << '<' << "bound_op" << ',' << '[' << '>' << endl;
            bound_opCount++;
            lexemebegin = forwardptr;
            break;
        case ']':
            state = 0;
            cout << ']' << '\t' << '<' << "bound_op" << ',' << ']' << '>' << endl;
            bound_opCount++;
            lexemebegin = forwardptr;
            break;
        case ',':
            state = 0;
            cout << ',' << '\t' << '<' << "bound_op" << ',' << ',' << '>' << endl;
            bound_opCount++;
            lexemebegin = forwardptr;
            break;
        case ';':
            state = 0;
            cout << ';' << '\t' << '<' << "bound_op" << ',' << ';' << '>' << endl;
            bound_opCount++;
            lexemebegin = forwardptr;
            break;
        default:
            if (C == EOF)
            {
                cout << "END OF C_FILE";
                if (state != 0)
                {
                    error(1);          //文件最后一行出现错误
                }
            }
            else    //错误字符
```

```
            {
                cat();
                error(0);          //非法字符$、\等
            }

            state = 0;
            break;
        }
    }
    break;

case 1:            //#预编译
    cat();
    get_char();
    if (C == '\n'|| C==EOF)
    {
        state = 0;
        cout << token << '\t' << '<' << "预编译命令" << ',' << '-' << '>' << endl;
        lineCur++;
        token.clear();
        lexemebegin = forwardptr;
    }
    else
    {
        state = 1;
    }
    break;

case 2:        //标识符
    cat();
    get_char();
    if (isletter() || isdigit() || C == '_')
    {
        state = 2;
    }
    else
    {
        //cout << *forwardptr;
        retract();
        //cout << *forwardptr;
        state = 0;
        iskey = reserve();
        if (iskey == 1)
        {
            cout << token << '\t' << '<' << token << ',' << '-' << '>' << endl;
```

```
            keywordCount++;
            token.clear();
            lexemebegin = forwardptr;

        }
        else
        {
            int loc = idTable_insert();//获得插入标识符表
            cout << token << '\t' << '<' << "ID" << ',' << loc << '>' << endl;
            idCount++;
            token.clear();
            lexemebegin = forwardptr;
        }
    }
    break;

case 3:
    cat();
    get_char();
    if (isdigit())
    {
        state = 3;
    }
    else if (C == '.')
    {
        state = 4;
    }
    else if (C == 'E')
    {
        state = 6;
    }
    else
    {
        retract();
        state = 0;
        cout << token << '\t' << '<' << "NUM" << ',' << atoi(token.c_str()) << '>' <<
endl;
        numCount++;
        token.clear();
        lexemebegin = forwardptr;
    }
    break;

case 4:
    cat();
```

```
        get_char();
        if (isdigit())
        {
            state = 5;

        }
        else
        {
            retract();
            state = 0;
            cout << token << '\t' << '<' << "NUM" << ',' << atof(token.c_str()) << '>' <<
endl;
            token.clear();
            lexemebegin = forwardptr;
        }
        break;

    case 5:
        cat();
        get_char();
        if (isdigit())
        {
            state = 5;
        }
        else if (C == 'E')
        {
            state = 6;
        }
        else
        {
            retract();
            state = 0;
            cout << token << '\t' << '<' << "NUM" << ',' << atof(token.c_str()) << '>' <<
endl;
            numCount++;
            token.clear();
            lexemebegin = forwardptr;
        }
        break;

    case 6:
        cat();
        get_char();
        if (isdigit())
        {
```

```
            state = 8;
        }
        else if (C == '+' || C == '-')
        {
            state = 7;
        }
        else
        {
            retract();
            error(2);          //无符号数出错E
            state = 0;
            token.clear();
        }
        break;

    case 7:
        cat();
        get_char();
        if (isdigit())
        {
            state = 8;
        }
        else
        {
            retract();
            error(2);          //无符号数出错+/-
            state = 0;
            token.clear();
        }
        break;

    case 8:
        cat();
        get_char();
        if (isdigit())
        {
            state = 8;
        }
        else
        {
            retract();
            state = 0;
            cout << token << '\t' << '<' << "NUM" << ',' << atof(token.c_str()) << '>' <<
endl;
            numCount++;
```

```
            token.clear();
            lexemebegin = forwardptr;
        }
        break;

    case 9:
        cat();
        get_char();
        if (C == '\\')
        {
            state = 11;         //转义字符
        }
        else
        {
            state = 10;         //正常字符
        }
        break;

    case 10:
        cat();
        get_char();
        if (C == '\'')          //字符常量中字符数超过 2
        {
            cat();
            state = 0;
            cout << token << '\t' << '<' << "CONST CHAR" << ',' << token << '>' << endl;
            constcharCount++;
            token.clear();
            lexemebegin = forwardptr;
        }
        else            //字符常量错误
        {
            cat();
            get_char();
            while (C != '\''&&C!='\n')
            {
                cat();
                get_char();
            }
            if (C == '\'')          //字符常量中不只有一个字符，如'az'
            {
                cat();
                error(3);
                state = 0;
                token.clear();
```

```
        }
        if (C == '\n')          //字符常量中没有单引号闭合，如'a;
        {
            retract();
            error(3);
            state = 0;
            token.clear();
        }
    }
    break;

case 11:
    cat();
    get_char();
    if ('0' == C || 'n' == C || 'a' == C || 'b' == C || 'f' == C || 'r' == C
        || 't' == C || 'v' == C || '\\' == C || '\'' == C )
    {
        state = 10;
    }
    else                    //非法转移字符如\w、\m
    {
        cat();
        get_char();
        while (C != '\''&&C != '\n')
        {
            cat();
            get_char();
        }
        if (C == '\'')          //非法转义字符\w、\m 等
        {
            cat();
            error(3);
            state = 0;
            token.clear();
        }
        if (C == '\n')          //字符常量中没有单引号闭合，如'\z;
        {
            retract();
            error(3);
            state = 0;
            token.clear();
        }
    }
    break;
```

```
    case 12:
        cat();
        get_char();
        if (C == '\\')    //转义字符
        {
            state = 13;
        }
        else if (C == '\n')
        {
            error(4);          //字符串中间换行隔断
            retract();
            state = 0;
            token.clear();
        }
        else if (C == '"')
        {
            cat();
            state = 0;
            cout << token << '\t' << '<' << "CONST STRING" << ',' << token << '>' <<
endl;
            stringCount++;
            token.clear();
            lexemebegin = forwardptr;//有问题
        }
        else
        {
            state = 12;
        }
        break;

    case 13:
        cat();
        get_char();
        if ('0' == C || 'n' == C || 'a' == C || 'b' == C || 'f' == C || 'r' == C
            || 't' == C || 'v' == C || '\\' == C || '\'' == C)
        {
            state = 12;
        }
        else
        {

            error(6);          //字符串中间转义字符错误
            state = 0;
            token.clear();
        }
```

```
        break;

    case 14:            //注释或除法运算
        cat();
        get_char();
        if (C == '/')           //单行注释
        {
            state = 15;
        }
        else if (C == '*')      //跳到块注释
        {
            state = 16;
        }
        else if (C == '=')      //赋值运算符
        {
            state = 18;
        }
        else
        {
            retract();
            state = 0;
            cout << '/' << '\t' << '<' << "arith_op" << ',' << '/' << '>' << endl;
            arith_opCount++;
            token.clear();
            lexemebegin = forwardptr;
        }
        break;

    case 15:
        get_char();
        if (C == '\n' || C == EOF)
        {
            retract();
            state = 0;
            token.clear();
            lexemebegin = forwardptr;
        }
        else
        {
            state = 15;
        }
        break;

    case 16:
```

```
        get_char();
        if (C == '*')
        {
            state = 17;
        }
        else
        {
            if (C == '\n')
                lineCur++;
            if (C == EOF)
            {
                error(1);// 注释到了末尾
            }
            state = 16;
        }
        break;

    case 17:
        get_char();
        if (C == '/')
        {
            state = 0;
            token.clear();
            lexemebegin = forwardptr;
        }
        else
        {
            if (C == EOF)
            {
                error(1);
            }
            state = 16;
        }
        break;

    case 18:
        cat();
        state = 0;
        cout << token << '\t' << '<' << "assign_op" << ',' << '-' << '>' << endl;
        assign_opCount++;
        token.clear();
        lexemebegin = forwardptr;

        break;
```

```
    case 19:
        cat();
        get_char();
        if (C == '=')   // <=
        {
            state = 0;
            cout << "<=" << '\t' << '<' << "relop" << ',' << "LE" << '>' << endl;
            relopCount++;
            lexemebegin = forwardptr;
            token.clear();
        }
        else if (C == '<')  // <<
        {
            state = 0;
            cout << "<<" << '\t' << '<' << "bitwise_op" << ',' << "<<" << '>' << endl;
            bitwise_opCount++;
            lexemebegin = forwardptr;
            token.clear();
        }
        else
        {             // <
            retract();
            state = 0;
            cout << '<' << '\t' << '<' << "relop" << ',' << "LT" << '>' << endl;
            relopCount++;
            lexemebegin = forwardptr;
            token.clear();
        }
        break;

    case 20:              // >
        cat();
        get_char();
        if (C == '=')   // >=
        {
            state = 0;
            cout << ">=" << '\t' << '<' << "relop" << ',' << "GE" << '>' << endl;
            relopCount++;
            lexemebegin = forwardptr;
            token.clear();
        }
        else if (C == '>')  // >>
        {
            state = 0;
            cout << ">>" << '\t' << '<' << "bitwise_op" << ',' << ">>" << '>' << endl;
```

```
            bitwise_opCount++;
            lexemebegin = forwardptr;
            token.clear();
        }
        else
        {               // >
            retract();
            state = 0;
            cout << '<' << '\t' << '<' << "relop" << ',' << "GT" << '>' << endl;
            relopCount++;
            lexemebegin = forwardptr;
            token.clear();
        }
        break;

    case 21:            //!
        cat();
        get_char();
        if (C == '=')   //!=
        {
            state = 0;
            cout << "!=" << '\t' << '<' << "relop" << ',' << "NE" << '>' << endl;
            relopCount++;
            token.clear();
            lexemebegin = forwardptr;
        }
        else            // !
        {
            retract();
            state = 0;
            cout << '!' << '\t' << '<' << "logic_op" << ',' << token << '>' << endl;
            logic_opCount++;
            token.clear();
            lexemebegin = forwardptr;
        }
        break;

    case 22:            //&
        cat();
        get_char();
        if (C == '&')
        {
            state = 0;
            cout << "&&" << '\t' << '<' << "logic_op" << ',' << "&&" << '>' << endl;
            logic_opCount++;
```

```
            token.clear();
            lexemebegin = forwardptr;
        }
        else if (C == '=')
        {
            state = 0;
            cout << "&=" << '\t' << '<' << "assign_op" << ',' << "&=" << '>' << endl;
            assign_opCount++;
            token.clear();
            lexemebegin = forwardptr;
        }
        else
        {
            retract();
            state = 0;
            cout << '&' << '\t' << '<' << "bitwise_op" << ',' << '&' << '>' << endl;
            bitwise_opCount++;
            token.clear();
            lexemebegin = forwardptr;
        }
        break;

    case 23:        //|
        cat();
        get_char();
        if (C == '|')
        {
            state = 0;
            cout << "||" << '\t' << '<' << "logic_op" << ',' << "||" << '>' << endl;
            logic_opCount++;
            token.clear();
            lexemebegin = forwardptr;
        }
        else if (C == '=')
        {
            state = 0;
            cout << "|=" << '\t' << '<' << "assign_op" << ',' << "|=" << '>' << endl;
            assign_opCount++;
            token.clear();
            lexemebegin = forwardptr;
        }
        else
        {
            retract();
            state = 0;
```

```
            cout << '|' << '\t' << '<' << "bitwise_op" << ',' << '|' << '>' << endl;
            bitwise_opCount++;
            token.clear();
            lexemebegin = forwardptr;
        }
        break;

    case 24:          //~
        state = 0;
        cout << '~' << '\t' << '<' << "bitwise_op" << ',' << '~' << '>' << endl;
        bitwise_opCount++;
        token.clear();
        lexemebegin = forwardptr;
        break;

    case 25:       //^
        cat();
        get_char();
        if (C == '=')   //异或
        {
            state = 0;
            cout << "^=" << '\t' << '<' << "assign_op" << ',' << "^=" << '>' << endl;
            assign_opCount++;
            token.clear();
            lexemebegin = forwardptr;
        }
        else          //异或
        {
            retract();
            state = 0;
            cout << '^' << '\t' << '<' << "bitwise_op" << ',' << '^' << '>' << endl;
            bitwise_opCount++;
            token.clear();
            lexemebegin = forwardptr;
        }
        break;

    case 26:          //%
        cat();
        get_char();
        if (C == '=')   //%=
        {
            state = 0;
            cout << "%=" << '\t' << '<' << "assign_op" << ',' << "%=" << '>' << endl;
            assign_opCount++;
```

```
            token.clear();
            lexemebegin = forwardptr;
        }
        else            //%取模
        {
            retract();
            state = 0;
            cout << '%' << '\t' << '<' << "arith_op" << ',' << '%' << '>' << endl;
            arith_opCount++;
            token.clear();
            lexemebegin = forwardptr;
        }
        break;

    case 27:            //*
        cat();
        get_char();
        if (C == '=')   //*=
        {
            state = 0;
            cout << "*=" << '\t' << '<' << "assign_op" << ',' << "*=" << '>' << endl;
            assign_opCount++;
            token.clear();
            lexemebegin = forwardptr;
        }
        else            //*
        {
            retract();
            state = 0;
            cout << '*' << '\t' << '<' << "arith_op" << ',' << '*' << '>' << endl;
            arith_opCount++;
            token.clear();
            lexemebegin = forwardptr;
        }
        break;

    case 28:            //+
        cat();
        get_char();
        if (C == '+')   //++
        {
            state = 0;
            cout << "++" << '\t' << '<' << "arith_op" << ',' << "++" << '>' << endl;
            arith_opCount++;
            token.clear();
```

```
            lexemebegin = forwardptr;
        }
        else if (C == '=')  //+=
        {
            state = 0;
            cout << "+=" << '\t' << '<' << "assign_op" << ',' << "+=" << '>' << endl;
            assign_opCount++;
            token.clear();
            lexemebegin = forwardptr;
        }
        else              //+
        {
            retract();
            state = 0;
            cout << '+' << '\t' << '<' << "arith_op" << ',' << '+' << '>' << endl;
            arith_opCount++;
            token.clear();
            lexemebegin = forwardptr;
        }
        break;

    case 29:          //-
        cat();
        get_char();
        if (C == '-')   //--
        {
            state = 0;
            cout << "--" << '\t' << '<' << "arith_op" << ',' << "--" << '>' << endl;
            arith_opCount++;
            token.clear();
            lexemebegin = forwardptr;
        }
        else if (C == '=')  //-=
        {
            state = 0;
            cout << "--" << '\t' << '<' << "assign_op" << ',' << "--" << '>' << endl;
            assign_opCount++;
            token.clear();
            lexemebegin = forwardptr;
        }
        else if (C == '>')      //->
        {
            state = 0;
            cout << "->" << '\t' << '<' << "other_op" << ',' << "->" << '>' << endl;
            other_opCount++;
```

```
            token.clear();
            lexemebegin = forwardptr;
        }
        else              //-
        {
            retract();
            state = 0;
            cout << '-' << '\t' << '<' << "arith_op" << ',' << '-' << '>' << endl;
            arith_opCount++;
            token.clear();
            lexemebegin = forwardptr;
        }
        break;

    case 30:           //=
        cat();
        get_char();
        if (C == '=')   //==
        {
            state = 0;
            cout << "==" << '\t' << '<' << "relop_op" << ',' << "EQ" << '>' << endl;
            relopCount++;
            token.clear();
            lexemebegin = forwardptr;
        }
        else          //=
        {
            retract();
            state = 0;
            cout << '=' << '\t' << '<' << "assign_op" << ',' << '=' << '>' << endl;
            assign_opCount++;
            token.clear();
            lexemebegin = forwardptr;
        }
        break;

    default:
        cout << "unknow error" << endl;
        exit(0);
    }
}
```