HW2 Readme

2-1 思路

- 這題主要是利用 **stack** 能夠 **push and pop** 的特性，再經由 **while** 去判斷所有方向的可能性 ， 最後儲存在 **stack** 的答案轉換成地圖的格式做輸出。
- 接下來說明各個函式的功能。

CODE 分析

初始化函式

```
void Initialize( )
{
    for( int i = 0 ; i < dimension ; i++ )
    {
        for( int j = 0 ; j < dimension ; j++ )
        {
            mark[ i ][ j ] = 0 ; //標記是否走過
            maze[ i ][ j ] = 0 ; //地圖通不通
            ans[ i ][ j ] = 0 ; //最後由stack轉換的array
        }
    }
}
# 由於需要處理不只一個 query ， 故先做初始化 。
```

輸入處理函式

```
void ProcessInput(  )// read character store integer
{
    for( int i = 0 ; i < dimension ; i++ )
    {
        scanf("%s",str[ i ] ) ;
    }
    for( int i = 0 ; i < dimension ; i++ )
    {
        for( int j = 0 ; j < dimension ; j++ )
        {
            if( str[ i ][ j ] == '0' )
            {
                maze[ i  ][ j  ] = 0 ;
            }
            else maze[  i  ][ j   ] = 1 ;
        }
    } // read character store integer
    return ;
}

# 讀入字串再存為整數型態的輸入處理。
```

實作 stack

```
typedef struct{
    int row;
    int col;
    int dir;
}element ;
element stack[ MAX_STACK_SIZE ] ;  // global stack declaration

int full( )
{
    if( top == maxsize - 1 ) return 1 ;
    else return 0 ;
}

int empty( )
{
    if( top == -1 ) return 1 ;
    else return 0 ;
}

element push( element item ) // push item in stack
```

```
{
    if( full( ) == 0 )
    {
        stack[ ++ top ] = item ;
        return stack[ top ] ;
    }
}

element pop( ) // pop item in stack
{
    if( empty( ) == 0 ) return stack[ top -- ] ;
}

# 這題主要會用到這些 stack 的特性。
```

path函式

- 老鼠走迷宮的函式，其中我還有實作 **IsSafe** 來判斷邊界和走沒走過。

```
void path( void )
{
    int row , col , nxtrow , nxtcol , dir ,  found = FALSE ;
    top = 0 , stack[ 0 ].row = 0 ;
    stack[ 0 ].col = 0 , stack[ 0 ].dir = 0 ;
    mark[ 0 ][ 0 ] = 1 ;
    element position ; // postion of now

    while( top > -1 && !found )
    {
        position = pop( ) ;
        row = position.row , col = position.col , dir = position.dir
        while( dir < 4 &&  !found )
        {
            nxtrow = row + dirs[ dir ].vert ;
            nxtcol = col + dirs[ dir ].horiz ;
            if( nxtrow == dimension -1   &&  nxtcol == dimension -1
            else if( isSafe( nxtrow , nxtcol ) )
            {
                mark[ nxtrow ][ nxtcol ] = 1 ;
                position.row = row ;
                position.row = row ;
                position.col = col ;
                position.dir = ++dir ;
                push( position ) ; // push the path in stack
                row = nxtrow , col = nxtcol ;
                dir = 0 ; // keep the first direction : DOWN
            }   // current position has not been checked, place it
                // on the stack and continue
            else dir++ ;
        }
        // check all of the remaining
        // directions from the current position
    }

    for( int i = 0 ; i <= top ; i++ )
    {
        int stkrow = stack[ i ].row , stkcol = stack[ i ].col ;
        ans[ stkrow ][ stkcol ] = 1 ;
    } // put answer ( in stack ) to array for printing out

    ans[ row ][ col ] = 1 ;
    ans[ dimension - 1 ][ dimension - 1 ] = 1 ;

    for( int i = 0 ; i < dimension ; i++ )
    {
        for( int j = 0 ; j < dimension ; j++ )
        {
           printf("%d",ans[ i ][ j ]) ;
        }
        printf("\n") ;
    }

}

#  用while去做四個方向的判斷，
#  利用 stack push and pop 的特性將最終路徑儲存 。found == 1 時輸出。
```

```
int isSafe( int nxtrow , int nxtcol )
{
    if(  ( maze[ nxtrow ][ nxtcol ] != 1 ) ||
         ( mark[ nxtrow ][ nxtcol ] != 0 ) ||
           nxtrow >= dimension  ||  nxtcol >= dimension ||
           nxtrow < 0 ||  nxtcol < 0 )
    {
        return FALSE ;
    }
    else return  TRUE  ;
}
#     判斷邊界和這條路走過了沒，來做 push pop
```

2-2 思路

- 依照題目需求，先用 **IsInfixValid** 判斷expression是否valid ，再利用 **postfix_stack** 存取 運算符號 進行 **prefix to postfix**

CODE分析

主函式

```
int main()
{
   int qry = 0 ;
    scanf("%d",&qry) ;
    while( qry-- )
    {
        scanf("%s",expr);
        int flag = IsInfixValid( expr ) ;
        if( flag == 1 )//means infix is valid
        {
            printf("1 ") ;
            postfix() ; // infix to postfix
            printf("\n") ;
        }
        else// otherwise
        {
            printf("0\n") ;
        }
    }
}
```

stack 函式 實作

```
precedence postfix_pop(int *top)
{
        return postfix_stack[(*top)--];
}
void postfix_push(int *top,precedence item)
{
        // add an item to the global stack
        if(*top >= MAX_STACK_SIZE-1){
                return;
        }
        postfix_stack[++*top] = item;
}
```

IsInfixValid 函式

```
int IsInfixValid( char* expr )
{
    int valid = 0 ;
    int len = strlen( expr ) ;
    int lepar = 0 , rtpar = 0 ; // leftparen rightparen ;
    int integer = 0  , opt = 0 ;
    // num_integer num_operator
    int keep_opt_flag = 0 , keep_int_flag = 0 ;
    // check consecutive operator or integer
    int flag_par = 0 ;
    for( int i = 0 ; i < len ; i++ )
    {
        if( keep_opt_flag == 2 || keep_int_flag == 2 ) break ;
        // for consecutive operator or integer
        if( expr[ i ] == '(' )
        {
            lepar++ ;
            flag_par++ ;
        }
        else if( expr[ i ] == ')' )
        {
```

```
        {
            rtpar++ ;
            flag_par-- ;
            if( flag_par < 0 ) break ;
        }
        else if( expr[ i ] >= '0' && expr[ i ] <= '9' )
        {
            integer ++ ;
            keep_int_flag ++ ;
            keep_opt_flag = 0 ;
        }
        else
        {
            opt++ ;
            keep_opt_flag ++ ;
            keep_int_flag = 0 ;
        }
    }
    if( lepar == rtpar ) valid = 1 ;
    // check num_leftpaten == num_rightparen
    if( keep_opt_flag == 2 || keep_int_flag == 2 ||
        flag_par < 0 )  valid = 0 ;
    // consecutive integer or operator
    if( opt >= integer  ) valid = 0 ;
    //too many operator : 1++2
    if( integer == 1 ) valid = 0 ;
    // only one integer is not allowed
    return valid ;
}
# 連續的數字，運算符號 valid == 0 ;
# 只有 integer || operator || left/right paren is not allowed.
```

postfix( ) 函式

- 遇到**數字**就**輸出**掉 ， 遇到 operator 判斷要不要放進**stack** 。
- 判斷的依據方式 : 自定義一個 **precedence** 的資料型別 ， 去確保丟進來的 operator 優先度必須 **大於** 前一個 operator。
- 這部分的code包含幾個子函式，下面接著說明。

```
void postfix(void)
{

    int isp[] = {0,19,12,12,13,13,13,0};
    // in stack presedence

    int icp[] = {25,19,12,12,13,13,13,0};
    // is coming presedence

    char symbol;
    precedence token;
    int n = 0 , i = 0;
    int top = 0;

    postfix_stack[0] = eos;
    for( token = get_token(&symbol , &n) ; token != eos ;
         token = get_token(&symbol , &n) )
    {
        if(token == operand)
        {
               printf("%c",symbol);
```

```
                printf("%c",symbol);
                expr[i++]=symbol;
        }// 有數字就輸出
        else if(token == rparen)
        {
            while (postfix_stack[top] != lparen)
                print_token(postfix_pop(&top) , &i);
            postfix_pop(&top);
        }// 遇到有右括號 pop 到 左括號
        else
        {
            while(isp[postfix_stack[top]] >= icp[token])
                print_token(postfix_pop(&top) , &i);
            postfix_push(&top , token);
        }
    }
    while (  (token=postfix_pop(&top)  ) != eos )
    print_token(token , &i);
    expr[i]='\0';
}
```

建構 precedence 部分

```c
typedef enum {lparen,rparen,plus,minus,times,divide,
                mod,eos,operand} precedence;

precedence postfix_stack[MAX_STACK_SIZE];// 優先度判斷

int isp[] = {0,19,12,12,13,13,13,0}; // in stack presedence

int icp[] = {25,19,12,12,13,13,13,0}; // is coming presedence
```

輸入字串處理 部分

```
precedence get_token(char *symbol, int *n)
{

        *symbol = expr[(*n)++];
        switch(*symbol)
        {
        case '(': return lparen;
        case ')': return rparen;
        case '+': return plus;
        case '-': return minus;
        case '*': return times;
        case '/': return divide;
        case '\0': return eos;
        default: return operand;
        }
    // get token to build precedence
}
#  藉由分析讀入字串的operator去建構優先度
#  ensure in-stack precedence (ISP) is higher than or
   equal to the incoming precedence (ICP) of the new operator.
```

輸出字串處理 部分

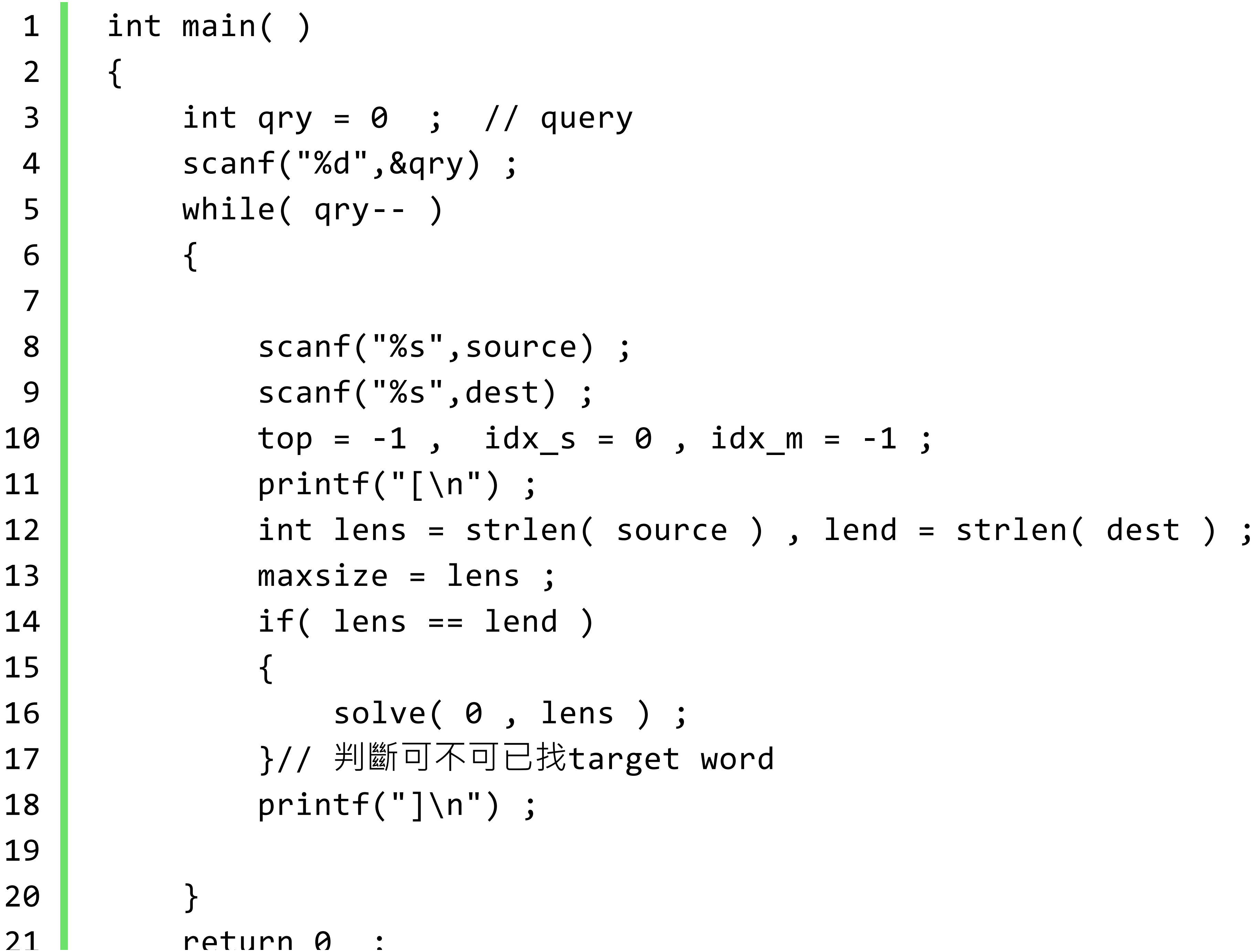
```
void print_token(precedence token,int *i)
{
        if( token == plus )
        {
            printf("+") ;
            expr[(*i)++]='+' ;
        }
        else if( token == minus )
        {
            printf("-") ;
            expr[(*i)++]='-' ;
        }
        else if( token == times )
        {
            printf("*") ;
            expr[(*i)++]='*' ;
        }
        else if( token == divide )
        {
            printf("/") ;
            expr[(*i)++]='/' ;
            expr[( 1)++]   ,    }
        }
        else return ;
}
```

2-3 思路

- **深度優先搜索** 去組成 target word ，並且 配合 **backtracking** 中 需要 push and pop 的特性 利用 **stack** 實作這件事。

CODE 分析

主函式



```c
int main( )
{
    int qry = 0  ;  // query
    scanf("%d",&qry) ;
    while( qry-- )
    {

        scanf("%s",source) ;
        scanf("%s",dest) ;
        top = -1 ,  idx_s = 0 , idx_m = -1 ;
        printf("[\n") ;
        int lens = strlen( source ) , lend = strlen( dest ) ;
        maxsize = lens ;
        if( lens == lend )
        {
            solve( 0 , lens ) ;
        }// 判斷可不可已找target word
        printf("]\n") ;

    }
    return 0  ;
    return 0 ;
}
```

solve 函式：用backtracking找 i 和 o

```
void solve( int index , int lenth )
{
    if( idx_m == lenth - 1 )
    {
        for( int i = 0 ; i < index ; i++ )
        {
            printf("%c ",ans[ i ] ) ;
        }
        printf("\n") ;
    } // find word to print out it
    else
    {
        if( idx_s < lenth )
        {
            ans[ index ] = 'i' ;
            push( source[  idx_s ++ ]  ) ;
            //stack[ ++ top ] = source[  idx_s ++ ] ; // push i
            solve( index + 1 , lenth ) ;
            source[ -- idx_s ] = stack[ top  ] ; // pop i
            pop( ) ; // 原本push，之後pop掉
        } // the case of i
        if(  ( top >= 0 )  && (  idx_m < lenth - 1 ) &&
              dest[  idx_m + 1  ] == stack[ top ] )
        {

            ans[ index ] = 'o';
            move[ ++ idx_m ] = stack[ top  ] ;
            // pop answer in move
            pop( ) ;
            solve(  index + 1 , lenth ) ; // dfs
            push ( move[ idx_m -- ] ) ;
            //原本pop，之後要再push進來
        }// the case of pop
    }
}
# case : i  ，先 push >> dfs >> pop
# case : o , 先 pop  >> dfs >> push
# 利用 backtracking 的方式找出所有的解
# idx_s : index of source || idx_m : index of move

```

關於此題 stack 的 push pop

```
int empty( )
{
    if( top == -1 ) return 1 ;
    else return 0 ;
}

void push( char item ) // push item in stack
{
    if( full( ) == 0 )  stack[ ++ top ] = item ;
    return ;
}

void pop(  ) // pop item in stack
{
    if( empty( ) == 0 )  top --  ;
    return  ;
}

```