山东大学 计算机科学与技术 学院

数据结构与算法 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：201700140056 | 姓名：李港 | | 班级：跟18.2（17.4） |
| 实验题目：实验六 栈 | | | |
| 实验学时：2h | | 实验日期：2019.10.25 | |
| 实验目的：  1、掌握栈结构的定义与实现；  2、掌握栈结构的使用。 | | | |
| 软件开发工具：  Virtual Studio 2019 | | | |
| 1. **实验内容**    1. 创建栈类，采用数组描述；    2. 计算数学表达式的值：输入数学表达式，输出表达式的计算结果。数学表达式由单个数字和运算符“+”、“-”、“\*”、“/”、“(”、“) ”构成，例如 2+3\*(4+5)–6/4。假定表达式输入格式合法。    3. 以一个m\*n的长方阵表示迷宫，0和1分别表示迷宫中的通路和障碍。设计一个程序，对任意设定的迷宫，求出一条从入口到出口的通路，或得出没有通路的结论。   迷宫根据一个迷宫数据文件建立。迷宫数据文件由一个包含0、1的矩阵组成。迷宫的通路可以使用通路上各点的坐标序列进行展示(使用图形展示最佳)。   1. **数据结构与算法描述（整体思路描述，所需要的数据结构与算法）**   **总体思路：**   * 1. 使用原生数组存储元素。   2. 提供缓冲区的动态扩大与缩小以节约内存资源。   3. 表达式计算思路：通过栈计算后缀表达式，通过后缀表达式计算最终结果，支持小数。   4. 走迷宫思路：通过栈存储路径，可区分此路不同，此路通，此路未尝试等信息，并支持通过字符阵列显示图像。   **数据结构：**   * 1. 采用原生数组作为底层数据结构   2. 提供缓冲区的动态扩容与收缩      1. 当缓冲区满时扩容到原来的二倍。      2. 当栈长度小于缓冲区的四分之一时，将缓冲区收缩为原来的一半。   **算法：**   * 1. 表达式计算思路：通过栈计算后缀表达式      1. 先生成后缀表达式：   中缀转后缀需要一个中缀表达式，一个存放后缀表达式的缓冲区，一个符号栈，对于中缀表达式从左到右每个元素   * + - 1. 如果是数字，直接进入后缀表达式       2. 如果是操作符   ->循环判断开始   * + - * 1. 如果 符号栈为空 或 栈顶为左括号，则直接入栈         2. 如果 当前操作符比栈顶符号优先级高，也直接入栈         3. 如果 当前操作符比栈顶符号优先级低或相等，则栈顶元素弹出到后缀表达式，并重新进行此判断（强调“重新进行此的判断”是为了优化程序结构，这样表述就只需要写一个while循环）   <-循环判断结束   * + - 1. 如果是左右括号          1. 如果是左括号，则入栈          2. 如果是右括号，则栈弹出到后缀表达式，直到左括号，并抛弃左括号     1. 通过后缀表达式计算结果   后缀表达式的计算要比比中缀转后缀简单，首先准备一个后缀表达式与一个数字栈，对于后缀表达式从左到右每个元素。   * + - 1. 如果是数字，则压入数字栈       2. 如果是操作符，则弹出数字栈中的两个，以第一次弹出作为右值，第二次弹出作为左值，进行相应计算并入数字栈（因为/与-与数字顺序相关）直到后缀表达式结束，返回栈顶数字       3. 本函数能够识别小数点，提供了对浮点数的转化功能   1. 走迷宫思路：通过栈存储路径      1. 首先将迷宫数组的边界进行扩展，这样就不必撰写额外的边界判断逻辑      2. 然后通过深度优先或广度优先方法进行线路搜索。  1. **测试结果（测试输入，测试输出）**    1. 验收展示：       1. 迷宫展示： 输入字符：  1：不可行走的墙壁  0：可以行走的道路 输出字符：  1：不可行走的墙壁  0：可以行走但没有走过的道路  +：可以行走但到达不了终点的道路  \*：可以行走且可以到达终点的道路       2. 算术表达式测试：    2. 平台提交      1. **分析与探讨（结果分析，若存在问题，探讨解决问题的途径）**   本实验最终结果正确，在实验过程中有以下问题或心得：   * 1. ！！！！有几个bug，首先是getposbykey调用双重错误，后来发现delete中并没有删除delete项目，导致后续null情况下出错，将条件倒转 正确   2. 却又忘记delete中后项前移   3. 却又将qianyixiang忘记判断标准位置   4. 这是因为没时间了解前提注意事项就上手干   5. 修复getpos后出现新的错误，修复新错误后delete又出错   6. 使用宏进行调试信息打印，不需要打印调试信息的时候将宏切换为为注释即可。   #ifdef DEBUG  #define DEBUG\_PRINT printf  #else  #define DEBUG\_PRINT /\  /printf  #endif  int a = 0;  DEBUG\_PRINT ( "%d",a );   * 1. 中缀表达式与后缀表达式可以理解为树的中序遍历与后序遍历，数字是叶子，运算符是父节点。   2. 良好的注释可以为以后的使用省去很多麻烦，比如我在这次在写代码时写了很多注释，在写实验报告的时候这些注释就派上了用场。   3. 实验过程中参考了很多他人的代码，但这些代码或多或少都存在一些不足，比如有些转后缀表达式的算法需要通过井号确定表达式头尾，有的代码结构混乱等。为了避免这些问题，我将计算表达式的函数拆分并封装成无状态的工具类，使代码更加整洁，也降低了出错几率。  1. **附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释）**   文件1 stack.h 栈头文件  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* stack.h \*  \* Copyright (C) 2019.10.25 TriAlley lg139@139.com \*  \* @brief 栈 \*  \* @license GNU General Public License (GPL) \*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #pragma once  #include<cstring>//memcpy  #include<cstdlib>//min  using namespace std;  /\*stack类  public:  enum stack\_err; 常见错误  protected:  void \_exLength (); 扩展缓冲区长度  void \_shLength (); 压缩缓冲区长度  int stackTop; 栈顶下标  int bufferLength; 缓冲区长度  T\* head; 栈数组头部  public:  stack ( int initialCapacity = 10 ); 构造函数，默认长度10  ~stack () { delete[] head; } 析构函数，释放缓冲区  bool empty () const { return stackTop == -1; } 返回是否空  int size () const { return stackTop + 1; } 返回长度  T top (); 返回栈顶元素  void pop (); 弹出栈顶元素，一般不会返回值  void push ( const T theElement ); 压入元素  \*/  template<class T>  class stack {  public:  typedef enum { pointer\_is\_null, newLength\_less\_than\_zero, stack\_empty } stack\_err;  protected:  void \_exLength () {  T\* temp = new T[bufferLength \* 2];  memcpy ( temp, head, bufferLength \* sizeof ( T ) );  delete[] head;  head = temp;  bufferLength = bufferLength \* 2;  }  void \_shLength () {  T\* temp = new T[bufferLength / 2];  memcpy ( temp, head, bufferLength / 2 \* sizeof ( T ) );  delete[] head;  head = temp;  bufferLength = bufferLength / 2;  }  int stackTop;  int bufferLength;  T\* head;  public:  stack ( int initialCapacity = 10 ) {  bufferLength = initialCapacity;  head = new T[bufferLength];  stackTop = -1;  }  ~stack () { delete[] head; }  bool empty () const { return stackTop == -1; }  int size () const { return stackTop + 1; }  T top () {  //判断是否为空  //有的实现版本中top输入引用，栈空则返回原数值，这样很不符合职责单一原则  if ( stackTop == -1 ) {  throw stack\_empty;  }  return head[stackTop];  }  void pop () {  if ( stackTop == -1 ) {  throw stack\_empty;  }  head[stackTop--].~T ();  /\*用于缩小缓冲区的代码\*/  if ( size () < bufferLength / 4 ) {  \_shLength ();  }  }  void push ( const T theElement ) {  //判断缓冲区长度并扩大缓冲区  if ( stackTop == bufferLength - 1 ) {  \_exLength ();  }  head[++stackTop] = theElement;  }  };  文件2 cal.h头文件  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* cal.h \*  \* Copyright (C) 2019.11.24 TriAlley lg139@139.com \*  \* @brief 对表达式计算所需函数的封装 \*  \* @license GNU General Public License (GPL) \*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #include"stack.h"  using namespace std;  /\*  \*中缀转后缀，后缀再计算  \*这些函数整理得比较规整，集合成一个工具类，不保存任何状态。  \*注释详实，不再进行过多解释。  \*/  /\*template<class T>  class calExpression  public:  typedef enum { wrong\_char, no\_such\_operator }cal\_err; //常见的错误项  private:  inline bool isNumber ( char c ) ; //判断字符是否是数字  inline bool isOperator ( char c ); //判断字符是否是操作符  inline T calOperator ( const char c, const T& a, const T& b );  inline int priority ( char c ); //返回操作符的优先级，优先级越高数值越大  public:  T calPostfix ( char\* postfix ); //计算后缀表达式，内部注释详实  void infixToPostfix ( char\* infix, char\* postfix ); //中缀转后缀，内部注释详实  \*/  template<class T>  class calExpression {  public:  typedef enum { wrong\_char, no\_such\_operator }cal\_err;  private:  inline bool isNumber ( char c ) {  if ( c >= '0' && c <= '9' ) {  return true;  } else {  return false;  }  }  inline bool isOperator ( char c ) {  switch ( c ) {  case '\*':  case '/':  case '+':  case '-':  return true;  default:  return false;  }  }  /\*第一次弹出a作为右值，第二次弹出b作为左值 eg b/a\*/  inline T calOperator (const char c,const T& a, const T& b ) {  switch ( c ) {  case '\*':  return b \* a;  case '/':  return b / a;  case '+':  return b + a;  case '-':  return b - a;  default:  return a;  }  }  inline int priority ( char c ) {  switch ( c ) {  case '\*':return 5; break;  case '/':return 5; break;  case '+':return 4; break;  case '-':return 4; break;  default:throw no\_such\_operator;  }  }  public:  /\*计算后缀表达式\*/  T calPostfix ( char\* postfix ) {  stack<T> result;  /\*  计算后缀表达式比中缀转后缀简单  首先准备一个后缀表达式与一个数字栈  对于后缀表达式从左到右每个元素  如果是数字，则压入数字栈  如果是操作符，则弹出数字栈中的两个，以第一次弹出作为右值，第二次弹出作为左值，进行相应计算并入数字栈（因为/与-与数字顺序相关）  直到后缀表达式结束，返回栈顶数字  \*/  for ( int i = 0;;) {  if ( isNumber ( postfix[i] ) || postfix[i] == '.' ) {  /\*这里将浮点字符串转化为浮点数，有时间可以抽离成单独的函数\*/  double temp = 0;  bool is\_int = true;  while ( isNumber ( postfix[i] ) || postfix[i] == '.' ) {  if ( postfix[i] == '.' ) {  is\_int = false;  } else {  if ( is\_int ) {  temp = ( postfix[i] - '0' ) + temp \* 10;  } else {  temp = ( postfix[i] - '0' ) \* 0.1 + temp;  }  }  i++;  }  i--;  result.push ( temp );  } else if ( postfix[i] == ' ' ) {  } else if ( isOperator ( postfix[i] ) ) {  double a = result.top ();  result.pop ();  double b = result.top ();  result.pop ();  result.push ( calOperator ( postfix[i], a, b ) );  } else if ( postfix[i] == '\0' ) {  return result.top ();  } else {  throw wrong\_char;  return 0;  }  i++;  }  }  /\*中缀转后缀，注释详实\*/  void infixToPostfix ( char\* infix, char\* postfix ) {  int j = 0;//目的下标  int i = 0;//源下标  stack<char> op\_stack;  /\*  中缀转后缀需要一个中缀表达式，一个存放后缀表达式的缓冲区，一个符号栈  对于中缀表达式从左到右每个元素  1.如果是数字，直接进入后缀表达式  2.如果是操作符  ->循环判断开始  1.如果 符号栈为空 或 栈顶为左括号，则直接入栈  2.如果 当前操作符比栈顶符号优先级高，也直接入栈  3.如果 当前操作符比栈顶符号优先级低或相等，则栈顶元素弹出到后缀表达式，并重新进行此判断  （强调“重新进行此的判断”是为了优化程序结构，这样表述就只需要写一个while循环）  <-循环判断结束  3.如果是左右括号  1.如果是左括号，则入栈  2.如果是右括号，则栈弹出到后缀表达式，直到左括号，并抛弃左括号  \*/  /\*每次循环，i均指向当前循环需要被判断的位置\*/  for ( ;;) {  if ( isNumber ( infix[i] ) || infix[i] == '.' ) {  /\*数字字符直接输出到后缀表达式，直到下一位不是数字字符输出空格到后缀表达式并跳出循环 \*/  while ( isNumber ( infix[i] ) || infix[i] == '.' ) {  postfix[j++] = infix[i++];  }  postfix[j++] = ' ';  i--;  } else if ( isOperator ( infix[i] ) ) {  while ( true ) {  if ( op\_stack.empty () || op\_stack.top () == '(' ) {  op\_stack.push ( infix[i] );  break;  } else if ( priority ( infix[i] ) > priority ( op\_stack.top () ) ) {  op\_stack.push ( infix[i] );  break;  } else {  postfix[j++] = op\_stack.top ();  postfix[j++] = ' ';  op\_stack.pop ();  }  }  } else if ( infix[i] == ')' ) {  while ( op\_stack.top () != '(' ) {  postfix[j++] = op\_stack.top ();  postfix[j++] = ' ';  op\_stack.pop ();  }  op\_stack.pop ();  } else if ( infix[i] == '(' ) {  op\_stack.push ( infix[i] );  } else if ( infix[i] == '\0' ) {  break;  } else {  throw wrong\_char;  return;  }  i++;//循环末，i处于本次循环判断处，需要让其指向下次循环判断处。  }  while ( !op\_stack.empty () ) {  postfix[j++] = op\_stack.top ();  postfix[j++] = ' ';  op\_stack.pop ();  }  postfix[j++] = '\0';  return;  }  };  文件3 main.cpp  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* main.cpp \*  \* Copyright (C) 2019.10.25 TriAlley lg139@139.com \*  \* @brief 栈测试 \*  \* @license GNU General Public License (GPL) \*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #include"stack.h"  #include"cal.h"  #include<iostream>  #pragma warning(disable:4996)  using namespace std;  struct point{  int \_row;  int \_col;  point ( int row, int col ) :\_row ( row ), \_col ( col ) {}  point ():\_row ( 0 ), \_col ( 0 ) {}  };  template<class T>  bool searchMazePathBFS ( char\*\* maze, int m, int n, point entry, point finish, stack<T>& paths ) {  paths.push ( entry );  while ( !paths.empty () ) {  point current = paths.top ();  paths.pop ();  maze[current.\_row][current.\_col] = '\*';  //如果到了终点则返回真  if ( current.\_row == finish.\_row && current.\_col == finish.\_col ) {  return true;  }  //上  if ( maze[current.\_row - 1][current.\_col] == '0' ) {paths.push ( point ( current.\_row - 1, current.\_col ) );}  //下  if ( maze[current.\_row + 1][current.\_col] == '0' ) {paths.push ( point ( current.\_row + 1, current.\_col ) );}  //左  if ( maze[current.\_row][current.\_col - 1] == '0' ) {paths.push ( point ( current.\_row, current.\_col - 1 ) );}  //右  if ( maze[current.\_row][current.\_col + 1] == '0' ) {paths.push ( point ( current.\_row, current.\_col + 1 ) );}  }  return false;  }  template<class T>  bool searchMazePath ( char\*\* maze,int m, int n, point entry, point finish, stack<T>& paths ){  paths.push ( entry );  while ( !paths.empty () ) {  point current = paths.top ();  maze[current.\_row][current.\_col] = '\*';  //如果到了终点则返回真  if ( current.\_row == finish.\_row && current.\_col == finish.\_col ) {  return true;  }  //上  if ( maze[ current.\_row - 1][current.\_col] == '0' ) {//如果能通过，则入栈  paths.push ( point( current.\_row-1, current.\_col) );  continue;  }  //下  if ( maze[current.\_row +1][current.\_col] == '0' ) {  paths.push ( point ( current.\_row + 1, current.\_col ) );  continue;  }  //左  if ( maze[current.\_row][current.\_col-1] == '0') {  paths.push ( point ( current.\_row, current.\_col-1 ) );  continue;  }  //右  if ( maze[current.\_row][current.\_col + 1] == '0' ) {  paths.push ( point ( current.\_row, current.\_col + 1 ) );  continue;  }  paths.pop (); //若上下左右都不通，则回溯。  maze[current.\_row][current.\_col] = '+';  }  return false;  }  int main () {  freopen ( "input.txt", "r", stdin );  int rows, cols;  cin >> rows >> cols;  /\*------构建迷宫矩阵开始----------\*/  char\*\* maze = new char\*[rows+2];  for ( int r = 0; r < rows+2; r++ ) {  maze[r] = new char[cols + 2];  }  for ( int r = 0; r < rows; r++ ) {  for ( int c = 0; c < cols; c++ ) {cin >> maze[r + 1][c + 1];}  }  for ( int c = 0; c < cols + 2; c++ ) {maze[0][c] = '1';}  for ( int c = 0; c < cols + 2; c++ ) {maze[rows + 1][c] = '1';}  for ( int r = 0; r < rows + 2; r++ ) {maze[r][0] = '1';}  for ( int r = 0; r < rows + 2; r++ ) {maze[r][cols + 1] = '1';}  for ( int r = 0; r < rows + 2; r++ ) {  for ( int c = 0; c < cols + 2; c++ ) {cout << maze[r][c];}  cout << endl;  }  /\*------构建迷宫矩阵结束----------\*/  /\*使用栈计算路径\*/  //(1,1)是入口，(rows,cols)是出口，矩阵外围封闭  stack<point> paths;  if ( searchMazePath ( maze, rows, cols, point ( 1, 1 ), point ( rows, cols ), paths ) ) {  printf ( "寻找成功：\n" );  } else {  printf ( "寻找失败！\n" );  }  /\*----------------输出路径----------------\*/  printf ( "\n已经行走的一条道路：\n" );  for ( int r = 0; r < rows + 2; r++ ) {  for ( int c = 0; c < cols + 2; c++ ) {  cout << maze[r][c];  }  cout << endl;  }  calExpression<double> cal;  char str[100] = "12\*5-695+94.8/(44-(95/84+848))";  char\* postfix = new char[strlen ( str ) \* 2];  cal.infixToPostfix ( str, postfix );    cout << "中缀表达式：" << str << '\n';  cout <<"后缀表达式：" <<postfix << '\n';  cout <<"计算结果："<< cal.calPostfix ( postfix );  delete[] postfix;  return 0;  } | | | |