|  |
| --- |
| MAZE.H  ~~// Constants~~ |
|  |  |
|  | ~~#ifndef Maze\_h~~ |
|  | ~~#define Maze\_h~~ |
|  |  |
|  |  |
|  | ~~#ifndef TRUE~~ |
|  | #define TRUE 1 |
|  | ~~#endif~~ |
|  |  |
|  | ~~#ifndef FALSE~~ |
|  | #define FALSE 0 |
|  | ~~#endif~~ |
|  |  |
|  | #define SIZE 16 // Size of one dimention of Map |
|  |  |
|  |  |
|  | // Directions |
|  | #define NORTH 0 |
|  | #define EAST 1 |
|  | #define SOUTH 2 |
|  | #define WEST 3 |
|  |  |
|  | // Shortcut Constants |
|  | #define MAPIJ this\_maze->map[i][j] |
|  | #define MAP this\_maze->map |
|  | #define FLOODVAL this\_node->floodval |
|  | #define ROW this\_node->row |
|  | #define COL this\_node->column |
|  | #define VISITED this\_node->visited |
|  | #define LEFT this\_node->left |
|  | #define RIGHT this\_node->right |
|  | #define UP this\_node->up |
|  | #define DOWN this\_node->down |
|  |  |
|  | // Stack Constants |
|  | ~~#define SPI 1 // Stack Pointer Index~~ |
|  | ~~#define SSI 0 // Stack Size Index~~ |
|  | ~~#define STACK\_OFFSET 2~~ |
|  | #define STACKSIZE 256 |
|  |  |
|  | // Solver Constants - will be used on mouse |
|  | #define START\_X 0 |
|  | #define START\_Y 0 |
|  | #define LARGEVAL 301 |
|  |  |
|  | ~~// Solver Constants - for command line simulation only~~ |
|  | ~~#define NEWLINE 13~~ |
|  | ~~#define YES 'y'~~ |
|  | ~~#define NO 'n'~~ |
|  |  |
|  | ~~/\* Main template constants \*/~~ |
|  | ~~#define ONECELL 61~~ |
|  | ~~#define LEFT\_WALL\_SENSED 1700~~ |
|  | ~~#define FRONT\_WALL\_SENSED 2200~~ |
|  | ~~#define RIGHT\_WALL\_SENSED 1700~~ |
|  | ~~#define LEFT\_BASE\_SPEED 23000~~ |
|  | ~~#define RIGHT\_BASE\_SPEED 23000~~ |
|  | ~~#define P\_VAL 7~~ |
|  | ~~#define D\_VAL 0~~ |
|  | ~~#define TURN\_LEFT\_COUNT 17~~ |
|  | ~~#define TURN\_RIGHT\_COUNT 18~~ |
|  | ~~#define ABOUT\_FACE\_COUNT 39~~ |
|  | ~~#define CENTER 2000~~ |
|  | ~~#define LEFT\_TWO\_AWAY 570~~ |
|  | ~~#define RIGHT\_TWO\_AWAY 630~~ |
|  |  |
|  |  |
|  | typedef struct Node { |
|  |  |
|  | /\* data fields \*/ |
|  | short floodval; |
|  | short row; |
|  | short column; |
|  | short visited; |
|  |  |
|  | /\* pointers to neighbors \*/ |
|  | struct Node \* left; |
|  | struct Node \* right; |
|  | struct Node \* up; |
|  | struct Node \* down; |
|  |  |
|  | } Node; |
|  |  |
|  | typedef struct Maze { |
|  |  |
|  | Node \* map [SIZE][SIZE]; |
|  |  |
|  | } Maze; |
|  |  |
|  | typedef struct Stack { |
|  |  |
|  | short properties [STACK\_OFFSET]; |
|  | Node \* the\_stack [STACKSIZE]; |
|  |  |
|  | } Stack; |
|  |  |
|  |  |
|  | // Node Functions |
|  | Node \* new\_Node (const short i, const short j); |
|  | void delete\_Node (Node \*\* npp); |
|  | void flood\_fill (Node \* this\_node, Stack \* this\_stack, const short reflood\_flag); |
|  | void set\_wall (Node \* this\_node, const short dir); |
|  | void set\_value (Node \* this\_node, const short value); |
|  | void set\_visited (Node \* this\_node);  short get\_smallest\_neighbor\_dir (Node \* this\_node) |
|  | ~~short get\_smallest\_neighbor\_dir (Node \* this\_node, const short preferred\_dir);~~ |
|  |  |
|  | // Floodfill Helper Functions |
|  | short get\_smallest\_neighbor (Node \* this\_node); |
|  | short floodval\_check(Node \* this\_node) ; |
|  | void update\_floodval (Node \* this\_node); |
|  | void push\_open\_neighbors (Node \* this\_node, Stack \* this\_stack); |
|  |  |
|  |  |
|  | // Maze Functions |
|  | Maze \* new\_Maze (); |
|  | void delete\_Maze (Maze \*\* mpp); |
|  | void print\_map (const Maze \* this\_maze); |
|  |  |
|  |  |
|  | // Stack Functions |
|  | Stack \* new\_Stack(); |
|  | void delete\_Stack (Stack \*\* spp); |
|  | int is\_empty\_Stack (Stack \* this\_stack); |
|  | void pop (Stack \* this\_stack, Node \*\* npp); |
|  | void push (Stack \* this\_stack, Node \* this\_node); |
|  |  |
|  | ~~// Debug On, Off~~ |
|  | ~~void set\_debug\_on ();~~ |
|  | ~~void set\_debug\_off ();~~ |
|  | ~~int get\_debug\_mode ();~~ |
|  |  |
|  | ~~#endif~~ |

STACK.H

|  |
| --- |
| // Stack Constructor |
|  | Stack \* new\_Stack() { |
|  |  |
|  | Stack \* this\_stack = (Stack \*) malloc(sizeof(Stack)); |
|  |  |
|  | this\_stack->properties[SPI] = 0; |
|  | this\_stack->properties[SSI] = STACKSIZE; |
|  |  |
|  | return this\_stack; |
|  | } |
|  |  |
|  | // Stack Destructor |
|  | void delete\_Stack (Stack \*\* spp) { |
|  |  |
|  | if (spp == 0 || \*spp == 0) { |
|  | fprintf(stderr, "NULL POINTER\n"); |
|  | return; |
|  | } |
|  |  |
|  | free(\*spp); |
|  |  |
|  | \*spp = 0; |
|  |  |
|  | } |
|  |  |
|  |  |
|  | // Checks if this\_stack is empty |
|  | int is\_empty\_Stack (Stack \* this\_stack) { |
|  |  |
|  | //printf("%d\n", this\_stack->properties[SPI]); |
|  |  |
|  | if (this\_stack->properties[SPI] == 0) |
|  | return 1; |
|  | else return 0; |
|  | } |
|  |  |
|  | // Pops the top element of this\_stack |
|  | void pop (Stack \* this\_stack, Node \*\* npp) { |
|  |  |
|  |  |
|  | short index; |
|  |  |
|  | index = this\_stack->properties[SPI] - 1; |
|  |  |
|  | \*npp = this\_stack->the\_stack[index]; |
|  |  |
|  | this\_stack->properties[SPI] -= 1; |
|  |  |
|  | } |
|  |  |
|  | // Pushes an element to the top of this\_stack |
|  | void push (Stack \* this\_stack, Node \* this\_node) { |
|  |  |
|  | short index; |
|  |  |
|  | index = this\_stack->properties[SPI]; |
|  |  |
|  | this\_stack->the\_stack[index] = this\_node; |
|  |  |
|  | this\_stack->properties[SPI] += 1; |
|  | } |

// Stack Functions

Stack \* new\_Stack();

void delete\_Stack (Stack \*\* spp);

int is\_empty\_Stack (Stack \* this\_stack);

void push (Stack \* this\_stack, Node \* this\_node);

void pop (Stack \* this\_stack, Node \*\* temp);

// Stack Constructor

Stack \* new\_Stack()

{

Stack \* this\_stack = (Stack \*) malloc(sizeof(Stack));

this\_stack->top = -1;

return this\_stack;

}

// Stack Destructor

void delete\_Stack (Stack \*\* spp)

{

if (spp == 0 || \*spp == 0)

{

return;

}

free(\*spp);

\*spp = 0;

}

// Checks if this\_stack is empty

int is\_empty\_Stack (Stack \* this\_stack) {

if (this\_stack->top == -1)

return 1;

else return 0;

}

// Pushes an element to the top of this\_stack

void push (Stack \* this\_stack, Node \* this\_node) {

short index;

index = this\_stack->top;

this\_stack->top += 1;

this\_stack->the\_stack[index] = this\_node;

}

// Pops the top element of this\_stack

void pop (Stack \* this\_stack, Node \*\* temp)

{

short index;

index = this\_stack->top;

\*temp = this\_stack->the\_stack[index];

this\_stack->top -= 1;

}