

Lesson 34

Objectives: -

- Write down program of maze solver

Maze Solver

Function

This program reads the representation of a maze from a text file, and finds a path through it.

Programming Issues

The first problem is to read in the maze itself. This is done in the following steps:

1. Retrieve the name of the text file from the command line;
2. Open the file;
3. Skip through the file once to determine the number of lines;
4. Dynamically allocate an array of pointer to pointer to char, with the same number of elements as the number of lines in the file;
5. Skip through the file again, and allocate a char array for each line, and point one of our char ** pointers at it;
6. As we go through, check that both an entrance and an exit are present, and store the location of the entrance;

We use a struct to store all the information about the maze.

The algorithm used to solve the maze is simple - stick to the left hand wall of the maze, and you will find the exit. Note this only works if both the entrance and the exit are on the outer wall of the maze (as would be usual). If the exit is in the centre, for instance, it might not be solved. We show an example of this type in our code.

A recursive function is used to solve the maze. The function knows the current position, and the current direction. From here, it tries turning 90 degrees left, going straight on, and turning 90 degree right. Having chosen the direction, it calls itself. The function terminates when it comes to a dead end, or it finds the exit, or it finds the entrance again.

Usage

Specify the name of the text file containing the maze on the command line.

For instance, if you have a file called "maze1.maz" that looks like this:

```
XXXXXXXXXXXXXXXXXXXXIXXX
X                               X
X XXXXXXXXXXXXXXXXXXXXXXX
X                               X   X
X XXX XXXXX X X X X
X  X X   X X X X X
X X X XXX X X X X X
X X X   X X   X X
XXXXXXXXXXXXOXXXXXXXX
```

Call the program as follows:

```
./maze maze1.maz
```

and the program will produce the following output:

```
[paul@hermes maze]$ ./maze maze1.maz
Found exit!
XXXXXXXXXXXXXXXXXXXXIXXX
X@XXXXXXXXXXXXXXXXXXXX
X@XXXXXXXXXXXXXXXXXXXX
X@XXXXXXXXXXXXXXXXXXXX
X XXX XXXXX@X X X X
X  X X   X@X X X X
X X X XXX X@X X X X
```

```
X X X      X@X  X X
XXXXXXXXXXXXOXXXXXX
[paul@hermes maze]$
```

The format of the text file should be:

- Walls are denoted by a capital 'X';
- Paths through the maze are denoted by spaces;
- The entrance is denoted by a capital 'I'; and
- The exit is denoted by a capital 'O'

The program uses the '@' character to show the path through the maze.

Try testing the program out by designing your own mazes.

main.c

```
/*
  MAIN.C
  =====
  (c) Copyright Paul Griffiths 2002
  Email: mail@paulgriffiths.net

  Main function for maze solver
*/

#include <stdio.h>
#include <stdlib.h>

#include "maze.h"
#include "solve.h"

int main(int argc, char *argv[]) {
    struct maze maze;

    if ( argc < 2 ) {
        puts("You must specify the filename of
your maze");
        return EXIT_FAILURE;
    }
    else if ( argc > 2 ) {
        puts("Too many command line arguments");
        return EXIT_FAILURE;
    }
}
```

```
    GetMazeFromFile(argv[1], &maze);

    if ( solve(&maze) == MAZE_FOUNDEXIT )
        puts("Found exit!");
    else
        puts("Can't reach exit!");

    PrintMaze(&maze);
    FreeMaze(&maze);

    return EXIT_SUCCESS;
}
```

maze.h

```
/*
MAZE.H
=====
(c) Copyright Paul Griffiths 2002
Email: mail@paulgriffiths.net

Interface to maze functions
*/

#ifndef PG_MAZE_H
#define PG_MAZE_H

/* Structure definitions */

struct maze {
    char ** map;
    int startx, starty;
    int numrows;
    int initdir;
};

struct pos {
    int x, y, dir;
};

/* Macros */

#define BUFFERSIZE (1000)

#define MAZE_ENTRANCE 'I'
#define MAZE_EXIT      'O'
#define MAZE_WALL      'X'
#define MAZE_PATH      ' '
#define MAZE_TRAIL     '@'
```

```

#define MOVE_LEFT    (0)
#define MOVE_UP      (1)
#define MOVE_RIGHT   (2)
#define MOVE_DOWN    (3)

#define MAZE_NOWAY   (0)
#define MAZE_FOUNDEXIT (1)
#define MAZE_NOEXIT  (-1)

/* Function declarations */

void GetMazeFromFile(char * filename, struct maze
* maze);
void FreeMaze(struct maze * maze);
void PrintMaze(struct maze * maze);

#endif /* PG_MAZE_H */

```

maze.c

```

/*
MAZE.C
=====
(c) Copyright Paul Griffiths 2002
Email: mail@paulgriffiths.net

Implementation of maze functions
*/

#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <string.h>

#include "maze.h"

/* Creates a maze from a file */

void GetMazeFromFile(char * filename, struct maze
* maze) {
    char buffer[BUFFERSIZE];
    FILE * fp;
    char ** map;
    int n = 0, foundentrance = 0, foundexit = 0;

    /* Open file */

    if ( !(fp = fopen(filename, "r")) ) {

```

```

        sprintf(buffer, "Couldn't open file %s",
filename);
        perror(buffer);
        exit(EXIT_FAILURE);
    }

    /* Determine number of rows in maze */

    while ( fgets(buffer, BUFFERSIZE, fp) )
        ++n;

    /* Allocate correct number of rows */

    if ( !(map = malloc(n * sizeof *map)) ) {
        fputs("Couldn't allocate memory for
map\n", stderr);
        exit(EXIT_FAILURE);
    }

    /* Reset file */

    rewind(fp);
    n = 0;

    /* Store each row */

    while ( fgets(buffer, BUFFERSIZE, fp) ) {
        int i;

        if ( !(map[n] = malloc((strlen(buffer)+1
* sizeof map[n])) ) ) {
            fputs("Couldn't allocate memory for
map\n", stderr);

            for ( i = 0; i < n; ++i )
                free(map[i]);

            free(map);

            exit(EXIT_FAILURE);
        }

        strcpy(map[n], buffer);

        /* Remove trailing whitespace */

        for ( i = strlen(map[n]) - 1;
isspace(map[n][i]); --i )
            map[n][i] = 0;
    }

```

```

        /* Check for entrance and store position
if found */

        if ( !foundentrance ) {
            i = 0;
            while ( map[n][i] != 'I' &&
map[n][i++] );
            if ( map[n][i] == MAZE_ENTRANCE ) {
                maze->startx = i;
                maze->starty = n;
                foundentrance = 1;
            }
        }

        /* Check for exit */

        if ( !foundexit ) {
            if ( strchr(map[n], MAZE_EXIT) )
                foundexit = 1;
        }
        ++n;
    }

    /* Fill in maze structure */

    maze->map = map;
    maze->numrows = n;

    /* Exit if there is no entrance or no exit
*/

    if ( !foundentrance ) {
        fputs("Maze has no entrance!\n", stderr);
        FreeMaze(maze);
        exit(EXIT_FAILURE);
    }

    if ( !foundexit ) {
        fputs("Maze has no exit!\n", stderr);
        FreeMaze(maze);
        exit(EXIT_FAILURE);
    }

    /* Check for initial direction into the maze
*/

    if ( maze->startx < strlen(maze->map[maze-
>starty]) - 1 &&
        maze->map[maze->starty][maze->startx+1]
== MAZE_PATH )
        maze->initdir = MOVE_RIGHT;
    else if ( maze->startx > 0 &&
        maze->map[maze->starty][maze-
>startx-1] == MAZE_PATH )

```

```

        maze->initdir = MOVE_LEFT;
    else if ( maze->starty > 0 &&
             maze->map[maze->starty-1][maze->startx]
== MAZE_PATH )
        maze->initdir = MOVE_UP;
    else if ( maze->starty < (maze->numrows-1) &&
             maze->map[maze->starty+1][maze-
>startx] == MAZE_PATH )
        maze->initdir = MOVE_DOWN;

    /* Close file and return */

    if ( fclose(fp) ) {
        perror("Couldn't close file");
        FreeMaze(maze);
        exit(EXIT_FAILURE);
    }
}

/* Frees memory used by a maze */

void FreeMaze(struct maze * maze) {
    int n;

    for ( n = 0; n < maze->numrows; ++n )
        free(maze->map[n]);

    free(maze->map);
}

/* Outputs a maze */

void PrintMaze(struct maze * maze) {
    int n;

    for ( n = 0; n < maze->numrows; ++n )
        puts(maze->map[n]);
}

```

solve.h

```

/*
SOLVE.H
=====
(c) Copyright Paul Griffiths 2002
Email: mail@paulgriffiths.net

Interface to maze solving operation
*/

```

```
#ifndef PG_SOLVE_H
#define PG_SOLVE_H

#include "maze.h"

/* Function declarations */

int solve(struct maze * maze);

#endif /* PG_SOLVE_H */
```

solve.c

```
/*
  SOLVE.C
  =====
  (c) Copyright Paul Griffiths 2002
  Email: mail@paulgriffiths.net

  Implementation of maze solving operation
  */

#include "solve.h"

/* Recursive function to find a path through a
  maze */

int look(struct maze * maze, struct pos pos) {
  int i, n;

  /* Set new position based on direction */

  switch ( pos.dir ) {
  case MOVE_UP:
    pos.y -= 1;
    break;

  case MOVE_DOWN:
    pos.y += 1;
    break;

  case MOVE_LEFT:
    pos.x -= 1;
    break;

  case MOVE_RIGHT:
    pos.x += 1;
    break;
```

```

default:
    break;
}

/* Check new position */

if ( pos.y < 0 || pos.y >= maze->numrows )
    return MAZE_NOWAY;
else if ( pos.x < 0 || pos.x >= strlen(maze-
>map[pos.y]) )
    return MAZE_NOWAY;
else if ( maze->map[pos.y][pos.x] ==
MAZE_WALL )
    return MAZE_NOWAY;
else if ( maze->map[pos.y][pos.x] ==
MAZE_EXIT )
    return MAZE_FOUNDEXIT;
else if ( maze->map[pos.y][pos.x] ==
MAZE_ENTRANCE )
    return MAZE_NOEXIT;

/* Try all the three directions other than
backwards */

pos.dir -= 1;
if ( pos.dir < 0 )
    pos.dir += 4;

for ( i = 0; i < 3; ++i ) {
    maze->map[pos.y][pos.x] = MAZE_TRAIL;
/* Leave trail through maze */

    n = look(maze, pos);
    if ( n ) {
        if ( n == MAZE_NOEXIT )
            maze->map[pos.y][pos.x] =
MAZE_PATH; /* No exit, so clear trail */
        return n;
    }

    pos.dir += 1;
    if ( pos.dir > 3 )
        pos.dir -= 4;
}

/* Dead end, so clear trail and return */
maze->map[pos.y][pos.x] = MAZE_PATH;

return 0;
}

/* Function to find a path through a maze */

```

```

int solve(struct maze * maze) {
    struct pos pos;

    pos.x = maze->startx;
    pos.y = maze->starty;
    pos.dir = maze->initdir;

    return look(maze, pos);
}

```

maze1.maz

```

XXXXXXXXXXXXXXXXXXXXIXXX
X                    X
X XXXXXXXXXXXXXXXXXXXX
X                    X  X
X XXX XXXXX X X X X
X  X X  X X X X X
X X X XXX X X X X
X X X    X X  X X
XXXXXXXXXXXXOXXXXXXXX

```

maze2.maz

```

XXXXXXXXXXXXXXXXXXXXX
X X      X      X
X X XXXX  XXX XXXXXXXX
X X X                    X
X X X XXXXX XXX X XX
X XXX X  X X X X XX
I      X  X X XXXXXXX
X XXXXX X X      X
X X      X  X  XX O
XXXXXXXXXXXXXXXXXXXXX

```

nowayout.maz

```

XXXXXXXXXXIXXXXXXXXX
X  X XX  XX X  X
X X  X    X  X X
X XXXXX X X XXXXX X
X          O      X
X XXXXX X X XXXXX X
X X  X    X  X X
X  X XX  XX X  X
XXXXXXXXXXXXXXXXXXXXX

```

"Hello, World!", X Window System style

Function

This is another good old "Hello, World!" application, this time using the Xlib interface to the X Window System. Because an X application is event driven (i.e. program execution is determined by the user, rather than by the application itself), there is a certain amount of overhead involved. Most of the code shown here will be common to all X applications.

Programming Issues

There are a number of basic steps involved in creating an X application:

1. Connect to the X server
2. Create a window
3. Set some "hints" for the window manager, to suggest how the window manager should display the window (e.g. in terms of size, initial state)
4. Decide which events we want the window to receive
5. Create a graphics context for output
6. "Map", or show the window
7. Enter the events loop, which is where the application "does things"

HelloX.c

```
/*  
  
HelloX.C  
=====  
(c) Copyright Paul Griffiths 1999  
Email: mail@paulgriffiths.net  
  
"Hello, World!", X Window System style.  
  
*/
```

```

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Xos.h>
#include <X11/Xatom.h>

#include <stdlib.h>
#include <string.h>
#include <stdio.h>

/* Global variables */

Display *      display;
int           screen_num;
static char * appname;

/* main() function */

int main( int argc, char * argv[] ) {

    /* Window variables */

    Window      win;
    int         x, y;
    unsigned int width, height;
    unsigned int border_width;
    char *      window_name = "Hello, X Window
System!";
    char *      icon_name   = "HelloX";

    /* Display variables */

    char *      display_name = NULL;
    unsigned int display_width, display_height;

    /* Miscellaneous X variables */

    XSizeHints * size_hints;
    XWMHints *   wm_hints;
    XClassHint * class_hints;
    XTextProperty windowName, iconName;
    XEvent        report;
    XFontStruct * font_info;
    XGCValues      values;
    GC             gc;

    appname = argv[0];

    /* Allocate memory for our structures */

    if ( !( size_hints = XAllocSizeHints() ) ||
        !( wm_hints   = XAllocWMHints()   ) ||

```

```

        !( class_hints = XAllocClassHint() ) ) {
        fprintf(stderr, "%s: couldn't allocate
memory.\n", appname);
        exit(EXIT_FAILURE);
        }

        /* Connect to X server */

        if ( (display = XOpenDisplay(display_name))
== NULL ) {
        fprintf(stderr, "%s: couldn't connect to X
server %s\n",
                appname, display_name);
        exit(EXIT_FAILURE);
        }

        /* Get screen size from display structure
macro */

        screen_num      = DefaultScreen(display);
        display_width   = DisplayWidth(display,
screen_num);
        display_height  = DisplayHeight(display,
screen_num);

        /* Set initial window size and position, and
create it */

        x = y = 0;
        width  = display_width / 3;
        height = display_width / 3;

        win = XCreateSimpleWindow(display,
RootWindow(display, screen_num),
                x, y, width, height,
border_width,
                BlackPixel(display,
screen_num),
                WhitePixel(display,
screen_num));

        /* Set hints for window manager before
mapping window */

        if ( XStringListToTextProperty(&window_name,
1, &windowName) == 0 ) {
        fprintf(stderr, "%s: structure allocation
for windowName failed.\n",
                appname);
        exit(EXIT_FAILURE);
        }

```

```

        if ( XStringListToTextProperty(&icon_name, 1,
&iconName) == 0 ) {
            fprintf(stderr, "%s: structure allocation
for iconName failed.\n",
                appname);
            exit(EXIT_FAILURE);
        }

        size_hints->flags          = PPosition | PSize |
PMinSize;
        size_hints->min_width     = 200;
        size_hints->min_height    = 100;

        wm_hints->flags           = StateHint |
InputHint;
        wm_hints->initial_state   = NormalState;
        wm_hints->input           = True;

        class_hints->res_name     = appname;
        class_hints->res_class    = "hellox";

        XSetWMProperties(display, win, &windowName,
&iconName, argv, argc,
                size_hints, wm_hints,
class_hints);

        /* Choose which events we want to handle */

        XSelectInput(display, win, ExposureMask |
KeyPressMask |
                ButtonPressMask |
StructureNotifyMask);

        /* Load a font called "9x15" */

        if ( (font_info = XLoadQueryFont(display,
"9x15")) == NULL ) {
            fprintf(stderr, "%s: cannot open 9x15
font.\n", appname);
            exit(EXIT_FAILURE);
        }

        /* Create graphics context */

        gc = XCreateGC(display, win, 0, &values);

        XSetFont(display, gc, font_info->fid);
        XSetForeground(display, gc,
BlackPixel(display, screen_num));

        /* Display Window */

        XMapWindow(display, win);

```

```

/* Enter event loop */

while ( 1 ) {
    static char * message = "Hello, X Window
System!";
    static int    length;
    static int    font_height;
    static int    msg_x, msg_y;

    XNextEvent(display, &report);

    switch ( report.type ) {

    case Expose:

        if ( report.xexpose.count != 0 )
            break;

        /* Output message centrally in window
*/

        length = XTextWidth(font_info, message,
strlen(message));
        msg_x = (width - length) / 2;

        font_height = font_info->ascent +
font_info->descent;
        msg_y = (height + font_height) / 2;

        XDrawString(display, win, gc, msg_x,
msg_y,
                    message, strlen(message));

        break;

    case ConfigureNotify:

        /* Store new window width & height */

        width = report.xconfigure.width;
        height = report.xconfigure.height;

        break;

    case ButtonPress:          /* Fall
through */
    case KeyPress:

        /* Clean up and exit */

        XUnloadFont(display, font_info->fid);
        XFreeGC(display, gc);
        XCloseDisplay(display);

```

```
        exit(EXIT_SUCCESS);  
    }  
}  
    return EXIT_SUCCESS; /* We shouldn't get  
here */  
}
```

Questions: -

1. Write down program of maze solver