COMP0002 Principles of Programming

C Coursework

Submission: The coursework must be submitted online using Moodle by 4pm Wednesday 9th November 2022.

Aim: To design and implement a longer C program of up to several hundred lines of source code in length.

Feedback: The coursework will be marked by 9th December 2022.

The coursework is worth 5% of the overall module mark, and will be marked according to the UCL Computer Science Marking Criteria and Grade Descriptors.

On this scheme a mark in the range 50-59 is considered to be satisfactory, which means the code compiles and runs, does more or less the right things and has a reasonable design using functions. Marks in the ranges 60-69 and 70-79 represent better and very good programs, while the range 40-49 denotes a less good program that shows some serious problems in execution and/or design. A mark of 80-89 means a really outstanding program, while 90+ is reserved for something exceptional. A mark below 40 means a failure to submit work of sufficient merit.

To get a mark of 70 or better you need to submit a really very good program. Credit will be given for using the C language properly, novelty, as well as quality.

Getting a good mark: Marking will take into account the quality of the code you write. In particular pay attention to the following:

- Proper declaration, definition and use of functions, variables and data structures.
- The layout and presentation of the source code.
- Appropriate selection of variable and function names.
- Appropriate use of comments. Comments should add information to the source code, not duplicate what the code already says (i.e., no comments like "This is a variable"!).
- As much as possible your code should be fully readable without having to add comments.
- Selection of a suitable design to provide an effective solution to the problem in question.

Clean straightforward and working code, making good use of functions, is considered better than longer and more complex but poorly organised code.

Development Advice:

- Keep things straightforward!
- Keep things straightforward! (Very important so it is repeated!)

- Straightforward does not mean trivial.
- First brainstorm/doodle/sketch to get a feel for the program you need to write and what it should do.
- Don't rush into writing C code if you don't fully understand what variables or functions are needed. Don't let the detail of writing code confuse your design thinking.
- How is the behaviour of the program implemented in terms of functions calling each other?
- Role play or talk through the sequence of function calls to make sure everything makes sense.
- Are your functions short and cohesive?
- Can't get started? Do a subset of the problem or invent a simpler version, and work on that to see how it goes. Then return to the more complex problem.

What to Submit

Your coursework should be submitted on Moodle, via the upload link for the C Coursework. The upload will permit a single file to be uploaded, so you should create a zip archive file containing all the files you intend to submit and upload that. Please use the standard .zip file format only, *don't* use any other variant or file compression system.

The zipfile should be named COMP0002CW1.zip.

The zipfile should contain:

- The C source code file(s).
- A readMe file (see below).
- Any data files or image files needed to run the program.

Submit source code files, data or image files, and the readMe file only, *don't submit compiled code* (binary code) such as .o files or executable programs (.out or .exe). Also don't submit the drawing app files (drawapp-2.0.jar, graphics.h, graphics.c).

The readMe should include the following:

- A concise description of what the program does. You might use one or two (small) screenshots to help explain your program.
- The command(s) needed to compile and run the program.

This should be 1 page at the very most.

Note that anonymous marking of coursework is used, don't include your name or student number in the files submitted. The Moodle submission details are used by the Teaching and Learning team to determine identities after marking is completed.

Plagiarism

This is an individual coursework, and the work submitted must be the results of your own efforts. You can ask questions and get help at the Lab sessions in Week 5.

Using comments in your source code, you should clearly reference any code you copy and paste from other sources, or any non-trivial algorithms or data structures you use.

See the UCL guidelines at https://www.ucl.ac.uk/ioe-writing-centre/reference-effectively-avoid-plagiarism/plagiarism-guidelines.

Constraints

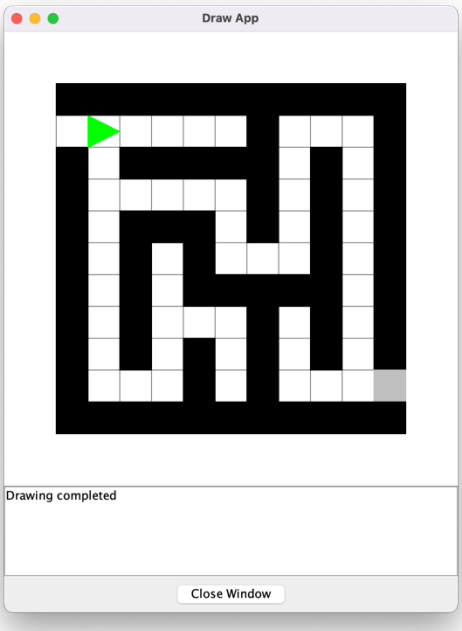
- Your code must compile with the gcc compiler and should only use the standard C libraries that come with gcc, plus the graphics.h and graphics.c files that come with the drawing app.
- You cannot use any other libraries or addons.
- Do not use any platform specific headers files or libraries (e.g., sys/windows.h).
- You do not need to worry about which specific version of C, ISO C, C99, etc., just write C code that can be compiled by gcc.
- Do not modify the graphics.h or graphics.c files. Your code must work with the versions supplied.

General Specification

You should remember or review the video about the simple robot abstraction (Week 2 on Moodle). The coursework is to implement a drawing program that displays and animates a robot moving around a rectangular maze, starting at the entrance and finishing at the exit.

The program should use version 2 of the drawing app that supports foreground and background layers, and can draw polygon shapes (such as a triangle).

As an example, a maze in the drawing window might look like this (this is an example, you don't have to use this maze):



Here the background layer displays the maze, with the filled squares representing the internal maze walls, while the grey filled square is a reflective marker denoting the exit. The robot is on the foreground layer at the maze entrance and is represented by the green triangle pointing in the direction of movement (to the right, or east in this case).

When the program runs the robot moves through the maze to try and find the marker at the exit.

The maze and the exit marker are drawn once at the start of the program and remain unchanged on the background layer. The robot is moved around the foreground layer, representing the animation of the robot running through the maze.

The robot is simple and only these functions should be provided to control the robot:

- void forward() – move forward to the next square in the direction the robot is facing. If there is a maze wall in front of the robot it does not move.
- void left() – turn the robot direction left (anti-clockwise) by 90 degrees, while remaining in the same position.
- void right() – turn the robot direction right (clockwise) by 90 degrees, while remaining in the same position.
- int atMarker() – return true if the robot is on the exit marker, otherwise false.
- int canMoveForward() – return true if the robot can move forward as there is no maze wall in front of it. Return false otherwise.

The robot cannot move diagonally or turn through any other angle than 90 degrees. You can additionally add parameters to these functions, to pass in the information about the robot position and direction.

The Program to Write

The program can be written in stages as suggested in the list below. Go as far as you can, but being able to complete the basic stages will be good enough to get a pass mark if you write reasonable code.

Basic Stage 1: Display a maze in the drawing window. The maze can be one that you have designed, it doesn't need to be generated by code. You might read the maze structure from a data file.

Basic Stage 2: Display a robot and animate it so that it moves through the maze to the exit marker. The robot can follow a path that you have pre-determined, it does not need to use a maze path finding algorithm.

Stage 3: Implement a basic maze search algorithm to move the robot from the entrance to the exit marker. Start with something simple, for example a maze where the robot can always turn right to get to the exit. Then progressively try more complicated mazes, where the search algorithm gets more sophisticated. Think about using recursion.

Stage 4: Implement an algorithm to generate mazes, with a single entrance and single exit. (challenging)

Stage 5: Improve your robot search algorithm to work with any maze you can generate. (challenging)

Hints:

- The robot can be represented by its (x,y) position in the grid and direction. Think about using a struct to represent the robot, and pass the struct as a parameter to the forward, left, right, etc. functions, so that each function can use the struct.
- The maze can be stored as a 2D array in the program, where each array element represents a maze square and holds a number denoting whether the square is empty or contains a wall or the exit marker.
- Use the sleep function call between each move of the robot, otherwise it will zoom round too fast to see!
- Here is a bit of example code to show what it might look like to control the robot:

```
while(…)
 {
   if (canMoveForward(aRobot)
     forward(aRobot);
   right(aRobot);
   sleep(500);
 }
```

  The functions canMoveForward and forward are passed a robot struct. The sleep at the end of the loop slows movement down so it can be seen. Experiment with the delay so you can see what is going on as the robot moves, but not so slow it takes too long to wait for the program to finish!
- How do you enter the starting position for the robot or parameters like the size of the maze?
  A drawing program can't conveniently do input from the keyboard as it cannot display input prompts that the user can see. All output to stdout is redirected to the drawapp program, so prompts will be redirected as well and the drawapp program will treat them as invalid input.

  However, you can use command line arguments. For example:

```
./a.out 1 0 east | java –jar drawapp–2.0.jar
```

  Here the "1 0 east" are the command line arguments to the program, meaning the robot starts at row 1, column 0 in the maze, facing east.

  To access the command line arguments the main function has two parameters:

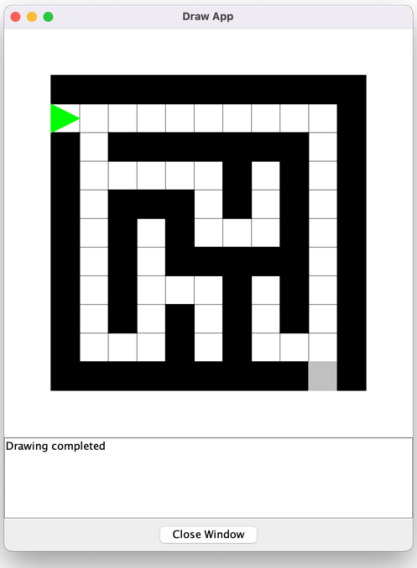
argc – the argument count, which is the number of command line arguments given *including* the name of the program. Hence, for './a.out 1 0 east' the value of argc will be 4.
argv – a pointer to an array of pointers to C Strings, hence of type char **. Each string holds one of the command line arguments and the size of the array of pointers will be 4, indexed 0-3. A command line argument is always stored as a string even if it represents a number.

This section of code illustrates how to use argc and argv:

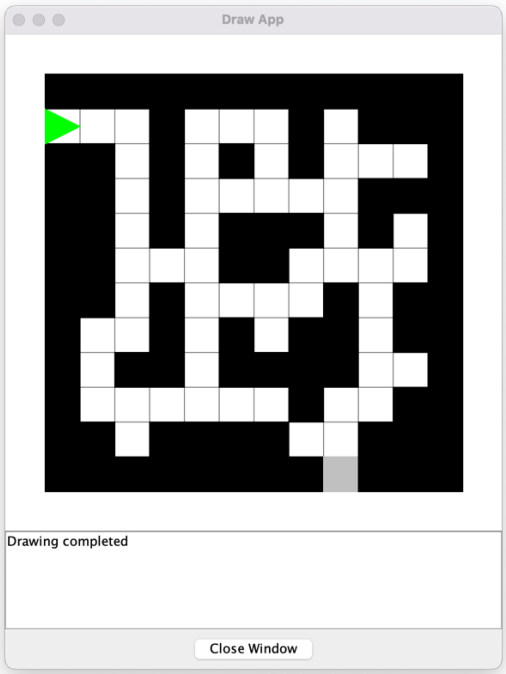
```
#include <stdlib.h> // Needed for the atoi function

int main(int argc, char **argv)
{
  // The default values if the command line arguments
  // are not given.
  int initialX = 0;
  int initialY = 0;
  char *initialDirection = "south";

  if (argc == 4) // Four arguments were typed
  {
    initialX = atoi(argv[1]); // Get x value
    initialY = atoi(argv[2]); // Get y value
    initialDirection = argv[3]; // Get direction
  }
  // Then continue with the rest of the code
```

The library function atoi (ascii to int) is used to convert an argument string into an int. The stdlib.h header file needs to be included to use atoi.

- A couple more example mazes.
  The first maze is a simple maze that can be solved by just turning right when a wall is reached:



This maze has several loops that a robot could get trapped in:



These are fairly small mazes that fit on the screen nicely but you can make bigger mazes if you like.