COMP0002 Principles of Programming

C Coursework

Submission: The coursework must be submitted online using Moodle by 4pm Monday 11th November 2024. See the Assessment Tasks and Submission – Sitting: 2024, Main section for the upload link. If you have a SoRA or get an Extenuating Circumstances extension your submission deadline will be updated accordingly.

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 2024.

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, implements a reasonable subset of the requirements, has more or less the right functionality and has a reasonable design using functions.

Marks in the ranges 60-69 and 70-79 represent correspondingly better and very good programs, while the range 40-49 denotes a less good program that shows some serious deficiencies in the design and when it is run.

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 pass.

To get a mark of 70 or better you do 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 code 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 must 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-4.1.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, use plain text or markdown format.

You should submit the final version of the program you implement. You do not need to submit the intermediate versions based on the stages listed below.

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 actual 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. This includes getting sections of code from tools like CoPilot or Chatgpt.

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 or clang compilers and should only use the standard C libraries that come with gcc or clang, 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 or clang.
- Do not modify the graphics.h or graphics.c files. Your code must work with the versions on Moodle.

Your source code can, and really should, make use of multiple .h and .c files. You don't need to put all your code in a single source file!

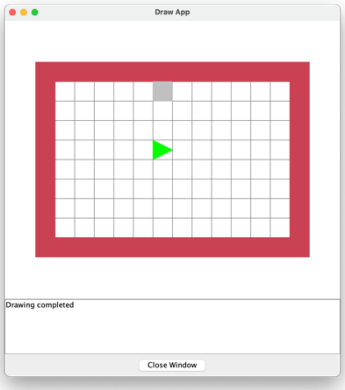
General Specification of the Program to Write

You should remember or review the content about the simple robot abstraction (Week 2 on Moodle). This coursework is to implement a drawing program that displays and animates a robot moving around an arena of any shape surrounded by a solid wall. The arena floor is laid out with square tiles, and the robot moves from the centre of one tile to the centre of the tile in front in the direction the robot is facing. The robot cannot move sideways or diagonally, just to the tile in front.

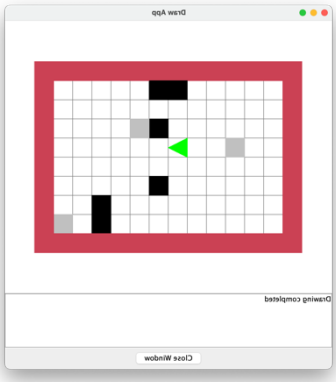
Obstacles can be positioned anywhere within the arena. Similarly, one or more markers can be positioned anywhere that is not occupied by an obstacle. Both obstacles and markers occupy one tile each, but multiple obstacles and markers can be on adjacent tiles.

The program should use version 4 of the drawing app on Moodle that supports foreground and background layers, and can draw polygon shapes (such as a triangle).

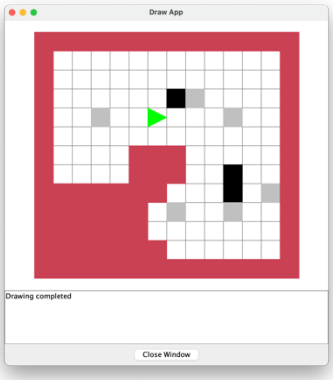
An arena in the drawing window might look like these examples (they are just examples, you don't have to copy them):



A basic rectangular arena with a gray coloured marker.



An arena with multiple obstacles and markers.



An irregularly shaped arena.

A drawing program has two layers, foreground and background, which can be drawn on independently. Anything drawn on the foreground layer appears in front of anything drawn on the background. Walls, obstacles and any other fixed items are drawn once on the background and stay in position regardless of what is drawn on the foreground. Only the robot is drawn on the foreground, and by repeatedly clearing and redisplaying the robot it appears to move around creating an animation effect. Updating the foreground does not affect the background, and the background does not need to be redrawn when the robot moves.

In the examples above the background layer displays the arena wall in red, black-filled tiles represent obstacles, and the grey-filled tiles are markers. The robot is on the foreground layer is represented by the green triangle pointing in the direction of movement. Right is east, left is west, up is north and down is south. You can use your own shapes, colours or images for your version of the program.

The robot implementation should support only the functions listed below to control it. Note that the parameters of these functions are not specified here – you should decide what parameters are needed. The (…) notation means that zero or more parameters are possible, this is not C syntax.

- void forward(…) – move forward to the next tile in the direction the robot is facing. If there is a wall or obstacle in front of the robot it does not move.
- void left(…) – turn the robot direction left (anti-clockwise) by 90 degrees, while remaining on the same tile.
- void right(…) – turn the robot direction right (clockwise) by 90 degrees, while remaining on the same tile.
- int atMarker(…) – return true if the robot is on a tile with a marker, otherwise false.
- int canMoveForward(…) – return true if the robot can move forward as there is no wall or obstacle in front of it. Return false otherwise.
- void pickUpMarker(…) – if the robot is on a tile with a marker the marker is picked up and carried by the robot. A robot can carry any number of markers.
- void dropMarker(…) – drop one marker on the tile the robot is currently on.
- int markerCount(…) – return the number of markers the robot is carrying, zero if it is not carrying any markers.

Function parameters might be added to pass in the information about the robot position and direction. The robot cannot move diagonally or turn through any other angle than 90 degrees.

The robot can have a memory, for example to build a map of the arena. You will need to decide how the memory is represented and used.

The Requirements for 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 basic rectangular arena in the drawing window, with a surrounding wall and the tile grid displayed inside the walls. Place a marker in a position next to a wall.

Basic Stage 2: Display a robot and animate it so that it moves around, finds the marker and stops. This can use the basic find and follow the wall algorithm. The robot should start from some random position and direction inside the arena that is not next to a wall.

Basic Stage 3: Add the code to create a rectangular arena of random size, with a marker placed at a random position next to a wall. The robot starts at a random position and direction

inside the arena, finds and picks up the marker, then goes to any corner, drops the marker and stops.

Stage 4: Allow the marker to be placed *anywhere* at a random position inside the arena, with the robot starting at a random position and direction to find it, and then going to any corner to drop the marker. A more complex algorithm will be needed to move the robot around.

Stage 5: Add one or more obstacles and markers at random positions, with the robot finding and picking up all the markers then going to a corner and dropping the markers. The robot algorithm should be as resilient as possible, such that the robot cannot get stuck in a never-ending cycle of movement due to the way that obstacles are positioned.

Stage 6 (challenge): Design a fixed size and shape arena with multiple obstacles and multiple markers. The robot should start at a random position and direction. Identify one tile as the home tile, such that the robot collects all the markers and returns them to the home tile. An additional function int isAtHome(…) can be added to the robot implementation. The arena can be any shape and does not need to be rectangular.

Stage 7 (challenge): Write the code to generate a random-sized and shaped arena with multiple obstacles and multiple markers. It should be possible for the robot to succeed in collecting all the markers and returning them to the home tile. A marker cannot be in an inaccessible location such as being surrounded by walks and obstacles with no open path to the marker.

Hints:

- The robot can be represented by its (x,y) position in the tile grid and direction. Think about using a struct to represent the robot, and pass the struct pointer as a parameter to the forward, left, right, etc. functions, so that each function can use the struct.
- The arena can be stored as a 2D array in the program, where each array element represents a tile and holds a value denoting whether the tile is empty or contains a wall, obstacle or 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 pointer to 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 arena and marker position?
  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 2 3 east | java -jar drawapp-4.0.jar
```

  Here the "2 3 east" are the command line arguments to the program, meaning the robot starts at row 2, column 3 in the arena, 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 2 3 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.