COMP0002 Principles of Programming

C Coursework

Submission: The coursework must be submitted online using Moodle by 4pm Tuesday 14th November 2023. Details are below. The upload link will be made available in the “Assessment Tasks and Submissions – Sitting: 2023, Main” section of the Moodle site.

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 and returned by 14th December 2023.

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 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 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. Also don't submit the drawing app files (drawapp.jar, graphics.h, graphics.c).

The readMe should include the following:

- A concise description of what the program does and how much you have completed. 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 submission and marking of coursework is anonymous, 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 based on information obtained from other sources.

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, or the clang compiler on MacOS, and should only use the standard C libraries that come with the compiler, 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 supplied.

Program Specification

You should remember or review the content about the simple robot abstraction (Weeks 1, 2 on Moodle). The coursework is to implement a drawing program that displays and animates a robot moving around a grid of squares in a rectangular area that is surrounded by a wall. The program should use version 2 or 3 of the drawing app that has the foreground and background layers, and can draw polygon shapes (such as a triangle).

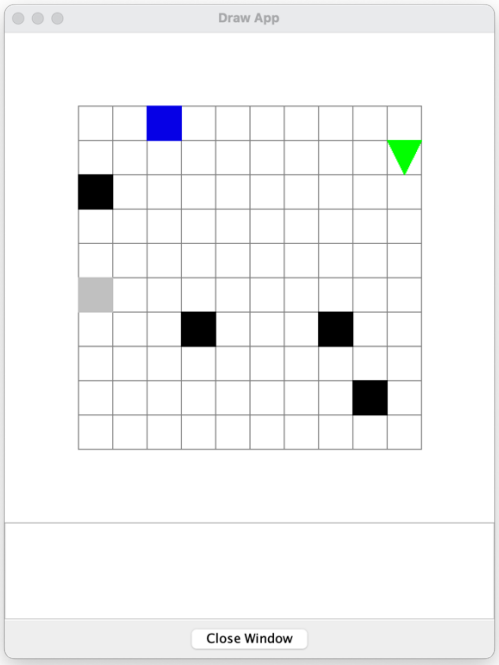
The robot is based on the examples given in the lectures but has some extra functionality. The robot can be facing in one of four directions only; north, south, east, west (or up, down, right, left). These functions should be implemented to control the robot:

- void forward() – move the robot forward to the next grid square in the direction the robot is facing. The robot is always in the middle of a square and moves forward to the middle of the next square. If there is a block or wall in front of the robot it does not move. A block occupies one square and prevents the robot from moving onto the square. The robot cannot move diagonally and can only move in the direction it is facing.
- void left() – turn the robot left (anti-clockwise) by 90 degrees, while remaining on the same square.
- void right() – turn the robot right (clockwise) by 90 degrees, while remaining on the same square.
- int atMarker() – return true if the robot is on a marker square, otherwise false. Markers can be placed on one or more squares.
- int canMoveForward() – return true if the robot can move forward, there is no wall or block in front of it. Return false otherwise.
- int atHome() – the ‘home’ square is the initial position of the robot. The function returns true if the robot is on the home square, otherwise false. By definition, there can only be one home square.

- void pickUpMarker() – the robot can pick up and carry a marker if it is on a marker square. If there is no marker on the square nothing happens. The robot can only carry a single marker at any time.
- Void dropMarker() – the robot can drop a marker on any square that is not occupied by a block. Multiple markers can be dropped on the same square.
- int isCarryingAMarker() – returns true if the robot is carrying a marker, otherwise false.

In addition, all the moves made by the robot from the home position can be stored in a data structure such as an array or list. This records the sequence of forward, left and right moves to where the robot currently is. This sequence can be used to help control the robot, for example by following the sequence in reverse to return to the home square. If required moves can be removed from the data structure, or it can be reset to remove all moves.

As an example, the drawing window might look like this (you don't have to copy this!):



Here the background layer displays the grid, with the black filled squares representing immovable blocks and the grey filled square representing a marker. The robot is on the foreground layer and is represented by the green triangle pointing in the direction of movement (down or south in this case). The blue filled square represents the home square where the robot started from.

When the program runs the robot moves depending on how it is programmed. It cannot move onto a square containing a block, but can move onto a square containing a grey marker and detect that it is on the marker square. The robot cannot move off the grid, so the edges around the entire grid are impassable walls.

The grid, blocks and home square are drawn once at the start of the program and remain unchanged on the background layer. A marker can also be drawn on the background layer, so will appear under the robot when it is on the same square, but move if the robot picks up the

marker. The robot is moved around the foreground layer, representing the animation of the robot carrying out a task.

Behaviour to Implement

Implement as many of these stages as you are able to. You should submit the final version of your program for all stages you have completed. You don't need to submit each stage separately.

Stage 1: Write the code to display a grid with a marker positioned against a wall and the robot positioned at the selected home square, which can be anywhere except where the marker is. Then move the robot around the grid to find the marker, where the robot stops.

Stage 2: Add the code to record the movements the robot makes to find the marker, so that it can pick up and return the marker to the home square by following the moves in reverse.

Stage 3: Position two or more markers next to the walls, so that the robot finds and returns all the markers to the home square. You might keep a counter to know if all the markers have been collected but is there a way another way of finding out?

Harder stages:
Stage 4: Add one or more blocks not next to any walls so that the robot can still collect all the markers. The markers are still next to the walls.

Stage 5: Allow markers to be positioned anywhere in the grid, not just next to a wall, so that the robot still collects them all.

Stage 6: Allow blocks to be positioned anywhere in the grid, including next to walls, so that robot can still collect all the markers.

Note that you can place blocks in positions that make it rather more difficult to navigate the robot around, or the robot can get stuck in a cycle so it keeps looping around the same set of squares.

Overall, this coursework is about getting the animation displayed and working, along with a working find and collect the marker(s) algorithm, using straightforward C code. You don't need to explore more complex robot representations and algorithms

Hints:

- The robot should 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 robot should not need to build a map of the grid (this is different from keeping a list of moves).

- The grid can be stored as a 2D array in the program, where each array element represents a grid square and holds a number denoting whether the square is empty or contains a marker, block or is a home square.
- 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);// Slow robot down so it can be seen!
 }
```

  In this code it is assumed that 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?
  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-3.0.jar
```

  Here the "2 3 east" are the command line arguments to a.out, meaning start the robot at position (2,3) in the grid, 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 = 6;
int initialY = 5;
char *initialDirection = "north";

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.