CS Paint

The year is 1985 . . . Microsoft has just released Windows 1.0 and packaged with it is a beautiful program called Paint, later referred to as MS Paint. For many people, this program is the beginning of a wonderful journey into the world of digital art.

In this assignment, you will be implementing CS Paint, COMP1511's answer to the venerable drawing program. CS Paint is a program that allows us to draw images to our terminal using a series of commands. The commands are made up of integers and are typed directly into our program. Each command will make some change to a digital canvas, a space for drawing.

CS Paint is already capable of setting up and drawing its canvas, it will be up to you to write code so that it can read commands and make the correct changes in the canvas.

Note: At time of release of this assignment (end of Week 3), COMP1511 has not yet covered all of the techniques and topics necessary to complete this assignment. At the end of Week 3, the course has covered enough content to be able to read in a single command and process its integers, but not enough to work with two dimensional arrays like the canvas or be able to handle multiple commands ending in End-of-Input (Ctrl-D). We will be covering these topics in the lectures, tutorials and labs of Week 4.

The Canvas

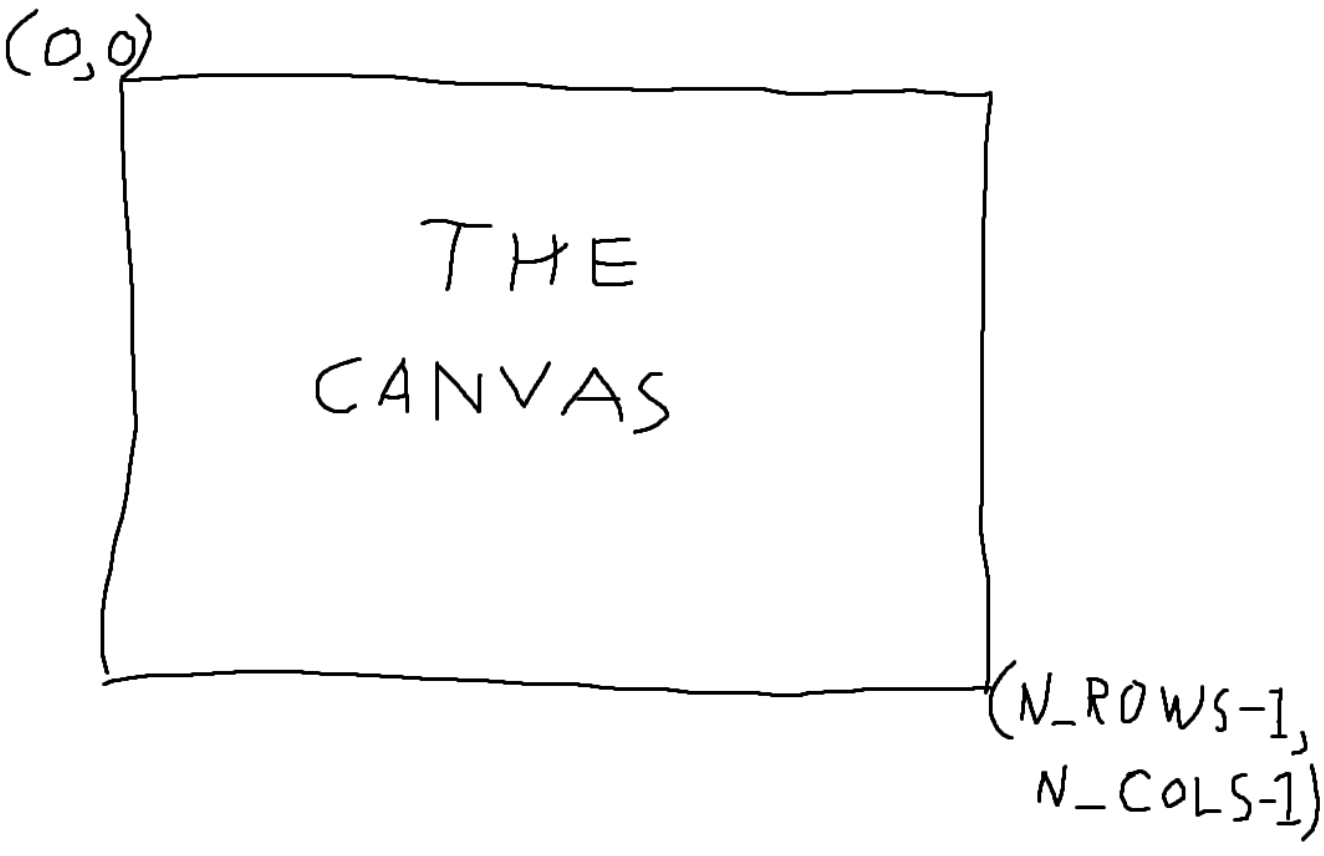
The canvas is a two dimensional array (an array of arrays) of integers that represents the space we will be drawing in. We will be referring to individual elements of these arrays as pixels on the canvas.

The canvas is a fixed size and has `N_ROWS` rows, and `N_COLS` columns. Both of these are defined constants.

Both the rows and columns start at 0, not at 1.

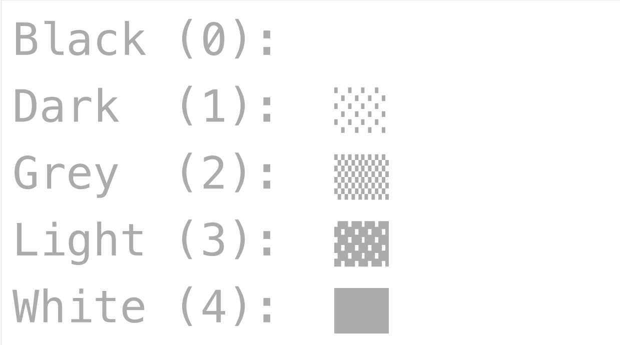
The top left corner of the canvas is `(0, 0)` and the bottom right corner of the canvas is `(N_ROWS - 1, N_COLS - 1)`. Note that we are using rows as the first coordinate in pairs of coordinates.

For example, if we are given an input coordinate `5 10`, we will use that to find a particular cell in our canvas by accessing the individual element in the array: `canvas[5][10]`



The integers in the pixels represent colours between black (which we call `0`) and white (which we call `4`). We will be starting with a white canvas and drawing black onto it, but as we progress, we will also be using shades of grey (not 50 of them, just a few). Note that these colours assume you have white text on a black background.

For reference, the shades are:



An empty canvas is shown below. In this documentation, we will always show you two versions of the output. In the "Output" you can see the version that your program is expected to produce (numbers between 0 and 4).

In the "Output (Stylized)" tab you can see a more readable version with the numbers converted to shades.

Note that you are *not* expected to produce this stylized output - we have tools that will convert it for you. Your program only needs to print the grid of numbers, as shown in the "Output" tab.

Empty Canvas

Program Output

```
$ dcc -o cs_paint paint.c
$ ./cs_paint < /web/cs1511/20T3/cs_paint/demos/empty_canvas.in
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
```

Stylized Output

```
$ 1511 canvas paint.c /web/cs1511/20T3/activities/cs_paint/demos/empty_canvas.in
```

Your program should not produce this output, `1511 canvas` will.

If you're curious, by the end of the assignment you'll be able to produce an image like this:

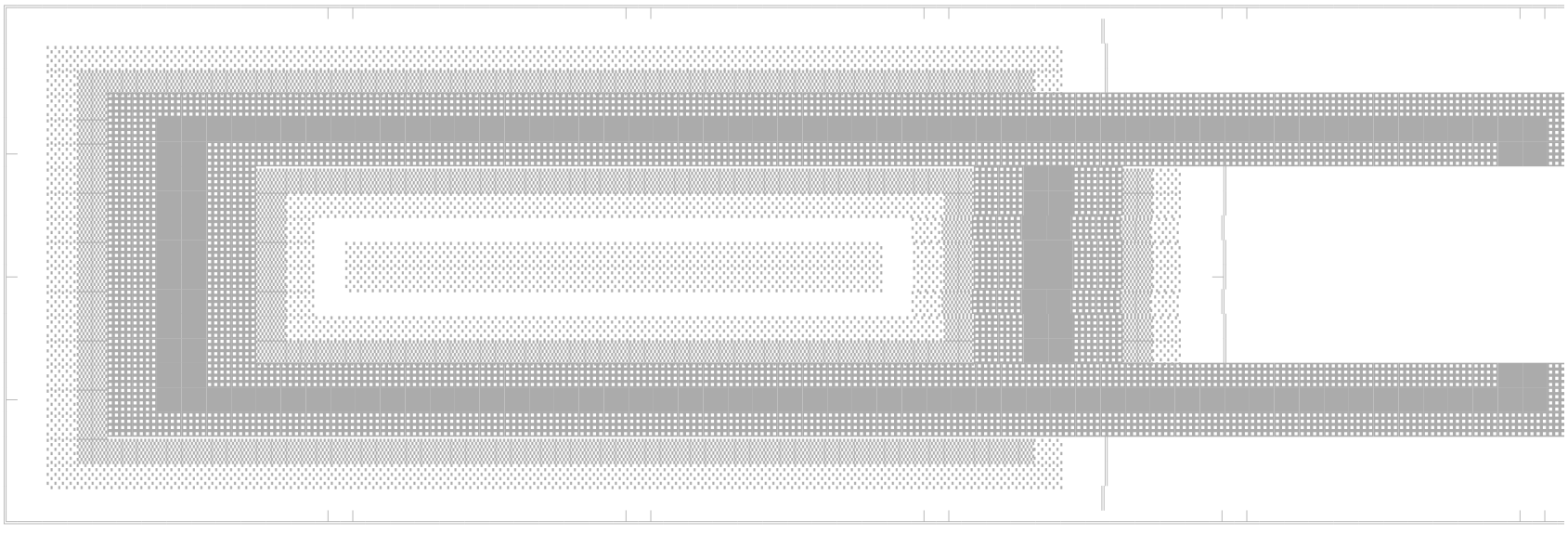
Technicolour

Program Output

```
$ dcc -o cs_paint paint.c
$ ./cs_paint < /web/cs1511/20T3/cs_paint/demos/technicolour.in
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
0 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 0
0 1 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 1 0
0 1 2 3 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 3 2 1 0
0 1 2 3 4 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 4 3 2 1 0
0 1 2 3 4 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 4 3 2 1 0
0 1 2 3 4 3 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 3 4 3 2 1 0
0 1 2 3 4 3 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 3 4 3 2 1 0
0 1 2 3 4 3 2 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 2 3 4 3 2 1 0
0 1 2 3 4 3 2 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 2 3 4 3 2 1 0
0 1 2 3 4 3 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 3 4 3 2 1 0
0 1 2 3 4 3 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 3 4 3 2 1 0
0 1 2 3 4 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 4 3 2 1 0
0 1 2 3 4 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 4 3 2 1 0
0 1 2 3 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 3 2 1 0
0 1 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 1 0
0 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Stylized Output

```
$ 1511 canvas paint.c /web/cs1511/20T3/activities/cs_paint/demos/technicolour.in
```



Your program should not produce this output, `1511 canvas` will.

The Commands

Each command given to the program will be a series of integers. There could be a large number of inputs, so you **should not** store all the commands in an array -- you should scan them in individually.

The first integer will always be the type of command, e.g. `1` means **Draw Line**.

Depending on what command the first integer specifies, you will then scan in some number of "arguments" (additional integers) that have a specific meaning for that command.

For example:

Command	Meaning
`1 5 5 10 5`	DRAW LINE starting at (row 5, col 5) until (row 10, col 5) VERTICALLY.

(see below for more details on this and other commands).

Your Task: Implementation

Your task for this assignment is to write a program that reads in one or more commands and outputs a canvas that shows the result of the commands.

Your program will be given commands as a series of integers on standard input. Your program will need to scan in these integers and then make the necessary changes in the canvas.

Initial tests will be with a single command per run of the program, but more advanced tests will expect the program to be able to scan and run multiple commands in a row.

Allowed C Features

In this assignment, there are no restrictions on C Features, except for those in the Style Guide.

We **strongly** encourage you to complete the assessment using only features taught in lectures up to and including Week 4. You can get full marks using the following features:

- `int` variables.
- `if` statements, including all relational and logical operators.
- `while` loops.
- `int` arrays.
- `printf` and `scanf`.
- functions.

Using any other features will not increase your marks (and will make it more likely you make style mistakes that cost you marks). /

Particularly, you do not need to use any pointers (or `malloc`) to gain full marks. They will only complicate the assignment. You also do not need to use `for` loops, and they are discouraged.

If you choose to disregard this advice, you **must** still follow the Style Guide. You also may be unable to get help from course staff if you use features not taught in COMP1511.

Starter Code

Download the starter code (paint.c) here or copy it to your CSE account using the following command:

```
$ cp -n /web/cs1511/20T3/activities/cs_paint/paint.c .
```

Download the test files (tests.zip) here or copy it to your CSE account using the following command:

```
$ cp -n /web/cs1511/20T3/activities/cs_paint/tests.zip .
```

You should download the above files to start the assignment.

`paint.c` is the starting point for your CS Paint program. We've provided you with some starter code to construct a canvas and to display it on the screen; you'll be completing the rest of the program.

`tests.zip` is a collection of test files that you can use to test your program. Each test file contains a series of commands that your program can use to draw images on the canvas.

Input Commands

Input to your program will be via standard input (similar to typing into a terminal).

You can assume that the input will always be integers and that you will always receive the correct number of arguments for a command. You will never be given a integer that represents a command that doesn't exist.

You can assume that input will always finish with the "End of Input" signal (Ctrl-D in the terminal).

Details on each command that your program must implement are shown below.

If you ever have a question in the form of "What should my program do if it is given these inputs?" you can run the canvas reference and copy its behaviour.

```
$ 1511 canvas solution --show-numbers
```

If you'd like to see the stylised output:

```
$ 1511 canvas solution
```

Note: Your program should NOT produce the stylised output, 1511 canvas will.

Stage One

Stage One implements basic drawing functions, giving your program the ability to draw lines and rectangles.

Line Drawing

Line Drawing: Summary

Command	"Draw Line"
Instruction	`1`
Inputs	`start_row start_col end_row end_col`

Examples

Command	Meaning
`1 10 3 10 10`	DRAW LINE starting at (row 10, col 3) until (row 10, col 10) HORIZONTALLY.
`1 1 1 9 1`	DRAW LINE starting at (row 1, col 1) until (row 9, col 1) VERTICALLY.
`1 9 1 1 1`	DRAW LINE starting at (row 9, col 1) until (row 1, col 1) VERTICALLY.
`1 2 2 2 2`	DRAW LINE starting at (row 2, col 2) until (row 2, col 2) (A POINT).

In Stage 1, you will be implementing the **Draw Line** command to draw horizontal and vertical lines.

The **Draw Line** command is given four additional integers, which describe two pixels: the `start` and `end` pixels of the line.

Each pixel consists of two numbers: the index of the `row`, and the index of the `column`.

For example, the command `1 10 3 10 10` tells your program to draw a line (`1`), starting at the pixel at row `10` and column `3`, and ending at the pixel at row `10` and column `10`.

When given the Draw Line command, your program should set the colour of the relevant elements in the canvas array, starting at the provided `start` pixel location, and continuing along the horizontal or vertical line until it reaches the `end` pixel location (including both the start and end pixels themselves).

Hints

- Your program will only be drawing either horizontal or vertical lines in Stage 1, which means that either `row1` and `row2` will be the same, or `col1` and `col2` will be the same.
- If `row1 == row2 && col1 == col2`, your program should draw a single pixel, at the location `(row1, col1), (row2, col2)`.

Handling Invalid Input

- If the given command **both** starts and ends outside the canvas, you should do nothing, and ignore that **Draw Line** command. Note that ignoring the command should still read in it's arguments. Otherwise, if one end pixel of the line is within the canvas you should draw the section of the line that is within the canvas.
- If the given start and end pixels would not give an entirely horizontal or vertical line, your program should ignore that **Draw Line** command and do nothing. Note that ignoring the command should still read in it's arguments.

Examples

Horizontal Line

Vertical Line

Box

Invalid Lines

You can run the autotests for **Draw Line** by running the following command:

```
1511 autotest-task 01 cs_paint
```

You should also check that your style is OK:

```
1511 style paint.c
```

Rectangle Drawing

Rectangle Drawing: Summary

Command	"Fill Rectangle"
Instruction	2
Inputs	`start_row start_col end_row end_col`
Examples	

Command	Meaning
`2 0 0 10 10`	FILL RECTANGLE starting at (row 0, col 0) until (row 10, col 10).
`2 6 2 2 2`	FILL RECTANGLE starting at (row 6, col 2) until (row 2, col 2).
`2 2 2 6 2`	FILL RECTANGLE starting at (row 2, col 2) until (row 6, col 2).

For the second part of Stage 1, you will be implementing the **Fill Rectangle** function, to draw rectangles.

The **Fill Rectangle** command is given four additional integers, which describe two pixels: the `start` and `end` pixels that make up two corners of the rectangle.

Each pixel consists of two numbers: the index of the `row`, and the index of the `column`.

For example, the command `2 0 0 10 10` tells your program to draw a rectangle (`2`), with one corner at the pixel at row `0` and column `0`, and with the opposing corner at the pixel at row `10` and column `10`.

When given the Fill Rectangle command, your program should colour all the pixels in the rectangle formed by the two corners `start` and `end`. You can assume that the edges of the rectangle are either vertical or horizontal, there are no rotated rectangles.

You cannot assume that you will always get the top-left and bottom-right corners of the rectangle; or that the corners will be given in a particular order.

Hints

- Your program could be given the `start` and `end` points in any order, e.g. `2 6 2 2 2` is valid, and produces the same result as `2 2 2 6 2`.
- If `row1 == row2 && col1 == col2`, your program should draw a single pixel, at the location `(row1, col1), (row2, col2)`.

Handling Invalid Input

If the given command **both** starts and ends outside the canvas, you should do nothing, and ignore that **Fill Rectangle** command. Otherwise, if one given corner pixel is withing the canvas, you should draw the part of the rectangle that is within the canvas. Note that ignoring the command should still read in it's arguments.

Examples

Small Square

Big Rectangle

Single Pixel Boxes

You can run the autotests for **Rectangle Drawing** by running the following command:

```
1511 autotest-task 02 cs_paint
```

You should also check that your style is OK:

```
1511 style paint.c
```

Stage Two

In Stage 2, you will be extending the functionality of your **Draw Line** and **Fill Rectangle** commands from Stage 1.

We strongly recommend that you finish Stage 1 before attempting Stage 2, as it would be very hard to test whether Stage 2 is working without Stage 1.

Note that completing Stage 2 is not necessary to gain a passing mark in this assignment.

Diagonals

Line Drawing: Diagonals: Summary

Command	"Draw Line"
Instruction	`1`
Inputs	`start_row start_col end_row end_col`
Examples	

Command	Meaning
`1 0 0 9 9`	DRAW LINE starting at (row 0, col 0) until (row 9, col 9) DIAGONALLY.
`1 1 1 2 2`	DRAW LINE starting at (row 1, col 1) until (row 2, col 2) DIAGONALLY.

For the first part of Stage 2, you will be modifying your **Draw Line** command to be able to draw diagonal lines.

Your program must still be able to draw horizontal and vertical lines as specified in Stage 1.

Hints

- Your program will only be required to draw diagonal lines that are on a 45 degree angle.
- In addition to horizontal and vertical lines, your program will now need to draw 45 degree lines. This means that your input checking will now need to test if two points are precisely diagonally oriented.
- As before, if `row1 == row2 && col1 == col2`, your program should draw a single pixel, at the location `(row1, col1), (row2, col2)`.

Invalid Input

- Drawing a diagonal line should follow the same rules for invalid input as drawing a horizontal or vertical line.
- If the given start and end pixels would not give an entirely horizontal or vertical line, **or a diagonal line on a 45 degree angle**, your program should ignore that **Draw Line** command and do nothing. Note that ignoring the command should still read in it's arguments.

Examples

Diagonal Line

Multiple Diagonal Lines

Other Angled Lines

You can run the autotests for **Diagonal Lines** by running the following command:

```
1511 autotest-task 03 cs_paint
```

You should also check that your style is OK:

```
1511 style paint.c
```

Shade

Shades: Summary

Command	"Change Shade"
Instruction	3
Inputs	`new_shade`

Examples

Command	Meaning
`3 2`	USE BRUSH WITH SHADE Grey (code: 2).
`3 1`	USE BRUSH WITH SHADE Dark (code: 1).

For the second part of Stage 2, you will be implementing the **Change Shade** command, which gives you access to both an eraser and different shades of grey.

In CS Paint there are a total of five shades, which we call `{BLACK, DARK, GREY, LIGHT, WHITE}`. They are each represented by a number between `0` (for `BLACK`) and `4` (for `WHITE`).

The **Change Shade** command is given one additional integer: the new shade that you will draw in all future commands, until the shade is changed again.

By default, your program should start with the shade `BLACK`.

Hints

- Painting over any other colours in the canvas replaces them with whatever colour the current shade is.
- The new shade should be used for both the **Draw Line** and **Fill Rectangle** commands.

Handling Invalid Input

- If the given shade is invalid (i.e., if it is outside of the range 0 to 4), your program should ignore that **Change Shade** command and do nothing.

Examples

Coloured Lines

Coloured Boxes

You can run the autotests for **Colour** by running the following command:

```
1511 autotest-task 04 cs_paint
```

You should also check that your style is OK:

```
1511 style paint.c
```

Stage Three

In Stages 3 and 4 and 5, you will be implementing more advanced commands.

Again, we strongly recommend that you finish Stage 1 and Stage 2 before attempting Stage 3.

Note that completing Stage 3 is not necessary to gain a passing mark in this assignment. We strongly recommend that before you do stage 3, you go back through your code to fix up style, and to split code into functions.

Copy and Paste

Copy and Paste: Summary

Command	"Copy Paste"
Instruction	4

Inputs	`start_row start_col end_row end_col target_row target_col`

Examples

Command	Meaning
`4 0 0 9 9 0 10`	COPY RECTANGLE starting at (row 0, col 0) until (row 9, col 9); PASTE STARTING AT (row 0, col 10).

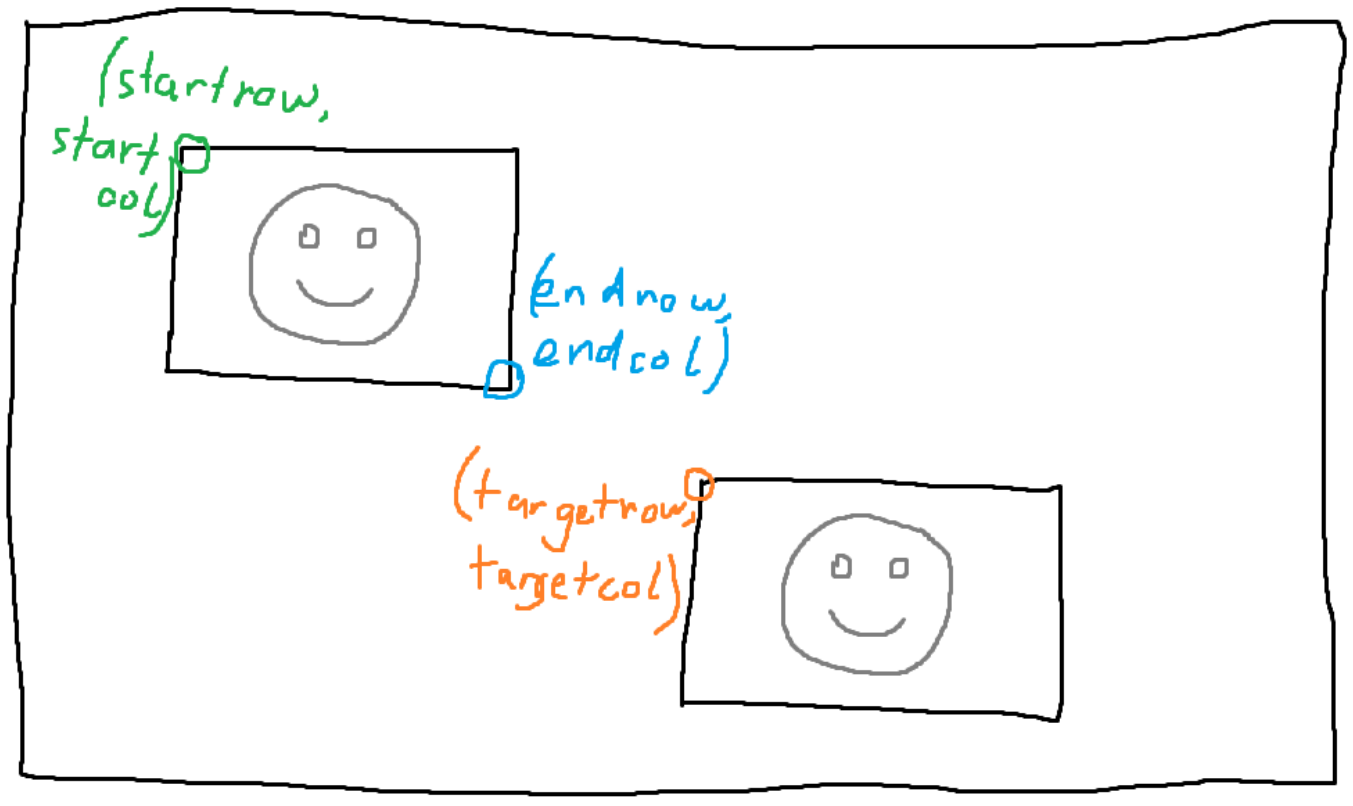
For Stage 3, you will be implementing the **Copy Paste** command, which allows you to *copy* a certain section of the canvas, and *paste* it elsewhere on the canvas.

The **Copy Paste** command is given six additional integers, which describe three pixels: `start`, `end`, and `target`.

The first two pixels, `start` and `end` describe the corners of a rectangle. This is the region that will be copied.

The third pixel, `target` describes the top-left pixel of the position on the canvas where that rectangle will be pasted.

The diagram below describes what these points are:



After calling the **Copy Paste** command, every pixel in the rectangle bounded by `start` and `end` should be copied to a rectangle of the same size that has `target` as its top left pixel.

Hints

- As with the **Fill Rectangle** command, your program could be given the `start` and `end` points in any order: they may not necessarily describe the top-left and bottom-right pixels of the rectangle.
- The pixels should be copied *exactly*, regardless of what the current shade is.
- It *is* valid to paste into a rectangle that overlaps with the copied rectangle. This means that, to earn full marks for this stage, the program must read all pixels in the copy rectangle before writing any pixels to the target rectangle. This will likely require a separate array to store the copied pixels before they're written to the target rectangle.

Invalid Input

- You may assume you will never get a command that would copy from outside the canvas.
- If a command would paste partially outside the canvas, you should only paste the parts that are inside the canvas.

Examples

Copy Paste Line Colours

Copy Paste Overlap

You can run the autotests for **Copy and Paste** by running the following command:

```
1511 autotest-task 05 cs_paint
```

You should also check that your style is OK:

```
1511 style paint.c
```

Stage Four

In Stage 4, you will again be implementing more advanced commands.

In Stages 4 and 5, we will not necessarily provide you with complete autotests -- it is your responsibility to check your code conforms to the specification and reference program.

Note that completing Stage 4 is not necessary to gain a passing mark in this assignment.

Ensure your code uses functions

We recommend that, if you can, you use functions from as early as stage 2 in this assignment. If you are not using them by this stage, you should take time to decompose your code into functions. You will find Stages 4 and 5 very difficult without having done this.

Macros

Macro Record: Summary

Command	"Macro Record"
Instruction	5
Inputs	`num_commands`

Examples

Command	Meaning
`5 2`	RECORD MACRO OF LENGTH 2.

Macro Playback: Summary

Command	"Macro Playback"
Instruction	6
Inputs	`row_offset col_offset`

Examples

Command	Meaning
`6 2 2`	PLAYBACK LAST MACRO, offset by 2 rows and 2 columns.

In computing, a 'macro' is a single command that does multiple things. For Stage 4, you will implement a simple 'macro' system, which allows you to record and play back the Draw Line and Draw Rectangle commands (you do not need to support the copy-paste command inside of macros).

You will need to implement two commands: **Macro Record** which saves the commands you enter; and **Macro Playback** which plays a macro back.

The number '5' means **Macro Record**. This command takes one argument, which is a number between 0 and 10. This tells you how many commands will follow this one. Those commands should be saved for later use (but should not immediately affect the canvas). If a macro was already recorded, this command should overwrite it completely, leaving no instructions from the previous macro.

The number '6' means **Macro Playback**. This command takes two arguments, which will be the number of rows or columns to shift the recorded commands down or to the right by. This number could be positive or negative (negative rows implying "shift up", negative columns implying "shift to the left"). This command should execute each of the saved commands in order. It should modify those commands so that they start and end at the shifted positions. The lines and rectangles should be created in the current colour (not necessarily the colour they were recorded in).

As an example:

```
5 2
1 1 1 3 3
1 5 5 7 7

1 2 2 4 2

6 1 0
```

Means:

```
5 2 // Start Recording, read in 2 commands.
1 1 1 3 3 // Record the command '1 1 1 3 3', but don't do it yet!
1 5 5 7 7 // Record the command '1 5 5 7 7', but don't do it yet!
// Two commands were now read in, so we go back to executing commands as we normally would.

1 2 2 4 2 // This command will be executed as normal

6 1 0 // Now, repeat the two recorded commands above , but offset by 1 row downwards
```

If we rewrite this into what the computer will interpret this as:

```
1 2 2 4 2

1 2 1 4 3
1 6 5 8 7
```

Handling Invalid Input

- If the command to draw on the canvas is invalid but becomes valid after it is shifted, you should execute the command.
- Using the '6' command before the '5' command has been used should result in no change to the canvas.
- You can re-use the '6' command. If you use it multiple times, it will continue to use the last recorded macro.
- It is possible that the '5' command will be reused. If this happens, only the most recent set of commands recorded should be saved. This means you should overwrite the previous macro.
- You will never be told to read in a negative number of commands, and you will never be told to read in more than 10 commands.
- The macro will never contain a command to change color, copy/paste, start another macro, or to save (as described in Stage Five).

Examples

Macro

You can run the autotests for **Macros** by running the following command:

```
1511 autotest-task 06 cs_paint
```

You should also check that your style is OK:

```
1511 style paint.c
```

Stage Five

In Stage 5, you will again be implementing more advanced commands.

Again, we strongly recommend that you finish Stages 1 - 4 before starting stage 5.

This stage has been designed to be challenging, and we have not provided you with complete autotests.

Save State

Save: Summary

Command	"Save"
Instruction	7
Inputs	[none]
Examples	

Command	Meaning
7	SAVE STATE.

In this final stage, you will be able to save multiple drawings. This could be used to save parts of an animation, or to show steps in a drawing.

The command '7' means "Save State". It takes no arguments. It should check that the current state of the canvas is different to the last save (if there was one). If it is different to the last save (or if this is the first time the canvas has been saved), the current state of the canvas should be saved. This should have no effect on the canvas. This command may be used as many times as you wish.

When your program finishes, it should print at most 6 canvases -- in order, it should print the fifth-last canvas saved (if it exists); the fourth, third and second last canvas saved (or however many exist); the last canvas saved (if it exists); and then the current state of the canvas.

If more than 5 saves are made within the program, only the last 5 should be kept -- you can discard any others. You could receive any number of save commands in your input.

Handling Invalid Input

You will never receive invalid input for this function.

Examples

Save State

You can run the autotests for **Save** by running the following command:

```
1511 autotest-task 07 cs_paint
```

You should also check that your style is OK:

```
1511 style paint.c
```

Hall of Fame Challenges

If you have completed these challenges, and at any point before the end of the term feel like returning to CS Paint, we have compiled below a list of extension challenges. You can feel free to invent your own challenges as well. Before starting these, you should have completed the assignment, and be passing all autotests. If you have questions about it, ask your tutor, or post a query on the course forum.

We'll choose a few entries to add to our Hall of Fame, based on code-style, what you've done, and general "awesomeness". Enteries are added at the course staff's discretion.

These challenges have no autotests, and completing them may result in your name being on a hall of fame for the assignment. They are not worth any marks. They are not bound by normal COMP1511 rules about style and features, though if the course staff can't understand them; we will not be able to determine whether to add them to the Hall of Fame.

To submit it for testing, you should send it to cs1511.challenge@cse.unsw.edu.au with the subject line "Hall of Fame Submission z5555555". Your email body should include a description of the challenge exercise, and a guide to how to run and test your solution. There are four challenges:

- Use the command `9` to mean "fill". It should take one pair of coordinates, and a new shade. Then, find all the pixels of the same color that indirectly connect to that pixel (in this case, that means that they share an edge with that pixel - pixels that are only diagonally connected do not count). Those pixels should all become the new shade mentioned in the command. It should operate similar to the paint-bucket in MS Paint.

- Use the command `7` to mean "draw solid circle". It should take one pair of coordinates to define the circle's centre, followed by one number to define the circle's radius. Any pixels where the distance from the center to that pixel is strictly less than the length specified by the radius should be set to the colour that a draw rectangle command would use.
- Write a program (not necessarily in C) that takes an output canvas and converts it to a set of instructions that prints a greyscale version of that image (without knowing what the input commands were). Programs that produce shorter sequences of commands will be rewarded with adulation in the hall of fame. Your program should not just print out pixel by pixel.
- Write a program (not necessarily in C) that takes some image (in a format of your choice) and converts it to a set of instructions that prints a greyscale version of that image. Again, the shorter output, the better (and it must not just print out pixel by pixel).
- Write a program that performs the same actions as CS Paint, that would be worth of entry into the IOCCC

These challenges are open to interpretation. If you have a question about how to complete it, ask on Discourse, or ask yourself "What would be cooler?". Submitting an entry isn't a guarantee that you will make the hall of fame -- we usually get around 20 emails in a term, so we've decided to accept a small number this term.

Testing

It is important to test your code to make sure that your program can perform all the tasks necessary to become CS Paint!

There are a few different ways to test (that are described in detail below):

- Typing in your own commands.You can use the commands shown above as examples, or work out your own.
- Testing from a series of commands written in a file. We have provided a set of test files that cover nearly all possible situations and commands that CS Paint should implement.
  Download the test files for the assignment here.
- Using autotests to run through all the test files at once.
- Running a Reference Implementation that we have created for you to compare against.

Testing your code

If you are testing with your own commands or commands written in a file, you can either use numerical output or our canvas output.

Getting raw numeric output

If you are debugging, or want to see the raw numbers as output, you can compile and run your program as follows:

```
$ ls
paint.c        tests/
$ dcc -o cs_paint paint.c
$ ./cs_paint
[type in your commands here, then type Control+D]
[your canvas will print out]
```

If you have an input file you want to run, you can specify them like this.

```
$ ls
paint.c        test_file1.in        test_file2.in
$ dcc -o cs_paint paint.c
$ ./cs_paint < test_file1.in
[the output of running the commands in test_file1.in]
```

This approach is limited to one input file at a time.

Coloured Blocks using `1511 canvas`

You can run the command `1511 canvas` on CSE computers (including via VLAB) along with the name of your C file, and we will compile it and show you the stylized output similar to in our examples in the Stages above.

```
$ 1511 canvas paint.c
You have run canvas without specifying any tests.
You may quit this program with Control + C
You can type lines below, and then press Control + D to see what output those
lines produce.

[type in your commands here, then press Control + D]
```

You can also see the direct numerical output by adding the `--show-numbers` flag.

```
$ 1511 canvas --show-numbers paint.c
```

If you have input files you want to run, you can specify them like this.

```
$ ls
paint.c        test_file1.in        test_file2.in
$ 1511 canvas paint.c test_file1.in test_file2.in
==> test_file1.in <==
[the output of running the commands in test_file1.in]
==> test_file2.in <==
[the output of running the commands in test_file2.in]
```

If you have many files you want to run, you can use the asterisk (`*`) instead of a name to mean "every".

```
$ ls tests/
paint.c        directory1        directory2
$ ls directory1/tests/
test_file1.in        test_file2.in
$ ls directory2/
other_test1.in        other_test2.in
$ 1511 canvas paint.c */*
==> directory1/test_file1.in <==
[the output of running the commands in directory1/test_file1.in]
==> directory1/test_file2.in <==
[the output of running the commands in directory1/test_file2.in]
==> directory2/test_file1.in <==
[the output of running the commands in directory2/test_file1.in]
==> directory2/test_file2.in <==
[the output of running the commands in directory2/test_file2.in]
$ 1511 canvas paint.c directory2/*
==> directory2/test_file1.in <==
[the output of running the commands in directory2/test_file1.in]
==> directory2/test_file2.in <==
[the output of running the commands in directory2/test_file2.in]
```

Automated Testing

On CSE computers (including via VLAB), the input files we have provided can all be checked at once using the command:

```
$ 1511 autotest cs_paint paint.c
```

Reference Implementation

If you have questions about what behaviour your program should exhibit, we have provided a sample solution for you to use.
You can use it by replacing the name of your C file with the word `solution` like so (on CSE Computers or via VLAB):

```
$ ls
test_file1.in        test_file2.in
$ 1511 canvas solution test_file1.in test_file2.in
==> test_file1.in <==
[the output of running the commands in test_file1.in]
==> test_file2.in <==
[the output of running the commands in test_file2.in]
```

You can also use `solution -n` instead of `solution` to just see the numbers produced by the sample solution.

Credits

Designed by Tom Kunc, Marc Chee and Andrew Bennett.

Assessment

Attribution of Work

This is an individual assignment. The work you submit must be your own work and only your work apart from exceptions below. Joint work is not permitted. At no point should a student read or have a copy of another student's assignment code.

You may use small amounts (< 10 lines) of general purpose code (not specific to the assignment) obtained from sites such as Stack Overflow or other publicly available resources. You should attribute clearly the source of this code in a comment with it.

You are not permitted to request help with the assignment apart from in the course forum, help sessions or from course lecturers or tutors.

Do not provide or show your assignment work to any other person (including by posting it on the forum) apart from the teaching staff of COMP1511. When posting on the course forum, teaching staff will be able to view the assignment code you have recently autotested or submitted with give.

The work you submit must otherwise be entirely your own work. Submission of work partially or completely derived from any other person or jointly written with any other person is not permitted. The penalties for such an offence may include negative marks, automatic failure of the course and possibly other academic discipline. Assignment submissions will be examined both automatically and manually for such issues.

Relevant scholarship authorities will be informed if students holding scholarships are involved in an incident of plagiarism or other misconduct. If you knowingly provide or show your assignment work to another person for any reason, and work derived from it is submitted you may be penalised, even if the work was submitted without your knowledge or consent. This may apply even if your work is submitted by a third party unknown to you.

Note, you will not be penalised if your work is taken without your consent or knowledge.

Submission of Work

You should submit intermediate versions of your assignment. Every time you autotest or submit, a copy will be saved as a backup. You can find those backups here, by logging in, and choosing the yellow button next to 'ass1_cs_paint'.

Every time you work on the assignment and make some progress you should copy your work to your CSE account and submit it using the `give` command below.

It is fine if intermediate versions do not compile or otherwise fail submission tests.

Only the final submitted version of your assignment will be marked.

You submit your work like this:

```
$ give cs1511 ass1_cs_paint paint.c
```

Assessment Scheme

This assignment will contribute 15% to your final mark.

80% of the marks for this assignment will be based on the performance of the code you write in `paint.c`

20% of the marks for this assignment will come from hand marking of the readability of the C you have written. These marks will be awarded on the basis of clarity, commenting, elegance and style. In other words, your tutor will assess how easy it is for a human to read and understand your program. The utility `1511 style` can help with this!

Marks for your performance will be allocated roughly according to the below scheme.

100% for Performance	Completely working implementation of Stages 1-5
90% for Performance	Completely working implementation of Stages 1-4
80% for Performance	Completely working implementation of Stages 1-3
70% for Performance	Completely working implementation of Stages 1 and 2
60% for Performance	Completely working implementation of Stage 1
50% for Performance	Partially working implementation of Stage 1 -- drawing horizontal and vertical lines
< 50% for Performance	Manually assessed based on closeness to a working program.

Marks for your style will be allocated roughly according to the scheme below

100% for Style	Perfect style
90% for Style	Great style, almost all style characteristics perfect.
80% for Style	Good style, one or two style characteristics not well done.
70% for Style	Good style, a few style characteristics not well done.
60% for Style	OK style, an attempt at most style characteristics.
<= 50% for Style	An attempt at style.

Tests will get marks as long as they serve a clear purpose, and don't overlap. Make sure you have a comment that tells us what the test is for!

Marks for tests will be allocated as below:

Stage One	2 marks per test function (1 mark per test-case)
Stage Two	1.5 marks per test function (0.75 marks per test-case)
Stage Three	2 marks for the test function (1 marks per-test case)
Stage Four	1 mark for the test function (0.5 marks per test-case)

Note that the following penalties apply to your total mark for plagiarism:

0 for the assignment	Knowingly providing your work to anyone and it is subsequently submitted (by anyone).
0 for the assignment	Submitting any other person's work. This includes joint work.
0 FL for COMP1511	Paying another person to complete work. Submitting another person's work without their consent.

The following is an indicative list of characteristics of your program that will be assessed, though your program will be assessed wholistically so other charactersitics may be assessed too:

- Header Commenting.
- Consistent, sensible indenting.
- Using Blank Lines & Whitespace.
- Using constants.
- Decomposing code into functions where relevent (especially after Stage 2).
- Using comments effectively (at least at top of functions, and not containing irrelevant info).

The course staff may vary the assessment scheme after inspecting the assignment submissions but it will remain broadly similar to the description above.

Due Date

This assignment is due Sunday 25 October 2020 20:00:00

If your assignment is submitted after this date, each hour it is late reduces the maximum mark it can achieve by 1%. For example if an assignment worth 74% was submitted 10 hours late, the late submission would have no effect. If the same assignment was submitted 30 hours late it would be awarded 70%, the maximum mark it can achieve at that time.

Change Log

Version 1.0
(2020-10-04 17:00)
- Released first assignment version.

Version 1.1
(2020-10-08 15:00)
- Added autotest-task information.
- Added more info about macro colour and about what 'ignoring' means.