Project assignment

Programming (L.EIC009), April 2025

Introduction

In this project you will use C++ to create a pipeline for handling RGB (https://en.wikipedia.org/wiki/RGB) color images with 8-bits per RGB channel. Starting code is provided with the initial code skeletons, along with support for reading and writing images encoded in the PNG (https://en.wikipedia.org/wiki/PNG) format and test code for you to validate your work.

You will need to develop code for the following classes in the C++ namespace prog:

- `prog::Color` to represent RGB colors;
- `prog::Image` to represent images where individual pixels are represented by Color; and
- `prog::Command` to represent an image manipulation command used in a script language that we call **Scrim** *(Script for images)*

You will also have to extend the hierarchy of `prog::Command` with new commands, and to address additional challenges presented in this document.

Example script in Scrim

The content of an example **Scrim** file follows below:

```
blank 750 380
     0 0 0
fill 0 0
     250 380
     255 0 0
add input/lion.png
     255 255 255
     0 0
fill 250 0
     250 380
     0 255 0
add input/lion.png
     255 255 255
     250 0
fill 500 0
     250 380
     0 0 255
add input/lion.png
     255 255 255
     500 0
save output/extra4.png
```

As illustrated, the **Scrim** file contains image creation and manipulation commands, explained below in this document. The `output/extra4.png` image produced by the final `save` command is shown below (Figure 1).

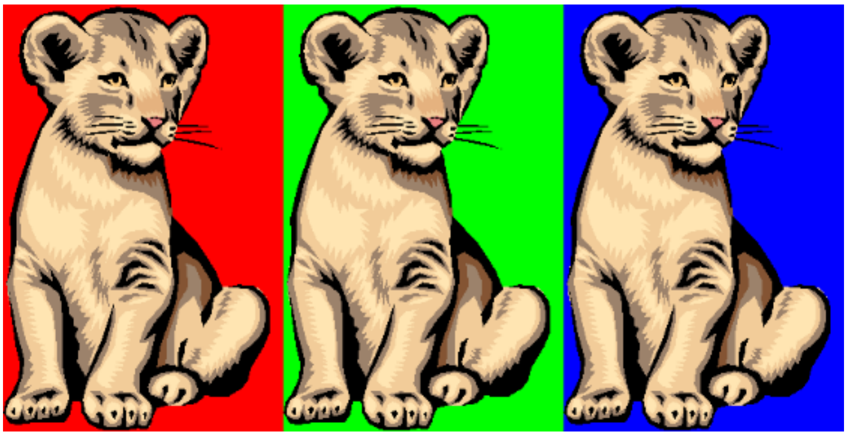


*Figure 1. Image produced by the example *Scrim** file.*

Constraints & logistics

Deadline

The deadline is **May 23, 2025, until 23:59**.

Group work

All group members must collaborate in the development of the project and everyone must be able to understand and explain all parts of the project.

During project presentation you will have to indicate the participation percentage of each group element. For example, if every element in a group of three participates equally, the participation percentage of each element will be 33%.

Plagiarism

Your code will be analyzed for plagiarism. Code that cannot be explained by the group members will be treated as plagiarism. Code that strongly appears to have been automatically written by a model (e.g., ChatGPT, Copilot) will also be considered plagiarism.

Plagiarism will result in the annulment of the project for involved groups (both for the provider of code and the receiver) and other possible disciplinary measures.

If you use code repositories like GitHub (and you should (https://www.freecodecamp.org/news/learn-the-basics-of-git-in-under-10-minutes-da548267cc91/)), make sure your repository for the project is **private** to avoid unintended dissemination of your work. If your code is public and another group uses it, your group will be considered as the provider of code.

Provided files

Getting started

Download the ZIP archive available at Moodle (https://moodle2425.up.pt/mod/resource/view.php?id=187508). Unzip the archive, then verify if compilation runs without errors. The commands below assume you are using the terminal and generating `make` build files to a folder `build` .

```
$ unzip project.zip
(...)
$ cd project
$ cmake -B build
-- The C compiler identification is (...)
(...)
-- Detecting CXX compile features - done
-- Configuring done (2.3s)
-- Generating done (0.0s)
-- Build files have been written to: (...)/build
$ cd build
$ make
[  3%] Building CXX object CMakeFiles/runscrim.dir/src/Color.cpp.o
[  7%] Building CXX object CMakeFiles/runscrim.dir/src/Command.cpp.o
(...)
[ 42%] Building CXX object CMakeFiles/runscrim.dir/src/Utils.cpp.o
[ 46%] Building CXX object CMakeFiles/runscrim.dir/main/RunScrim.cpp.o
[ 50%] Linking CXX executable runscrim
[ 50%] Built target runscrim
[ 53%] Building CXX object CMakeFiles/tester.dir/src/Color.cpp.o
[ 57%] Building CXX object CMakeFiles/tester.dir/src/Command.cpp.o
...
[ 92%] Building CXX object CMakeFiles/tester.dir/src/Utils.cpp.o
[ 96%] Building CXX object CMakeFiles/tester.dir/main/Tester.cpp.o
[100%] Linking CXX executable tester
[100%] Built target tester
```

The compilation generates two programs: `runscript` for running image processing scrims, and `tester` for validating your code using automated tests. You need to run them from the project root, hence after compiling you can run the tests as follows.

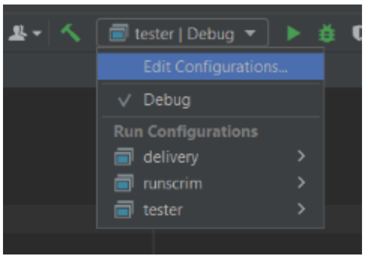
```
$ cd ..
$ build/tester
== 55 tests to execute  ==
[1] add1: fail
[2] add2: fail
...
[54] v_mirror3: fail
[55] v_mirror4: fail

== TEST EXECUTION SUMMARY ==
Total tests: 55
Passed tests: 0
Failed tests: 55
```

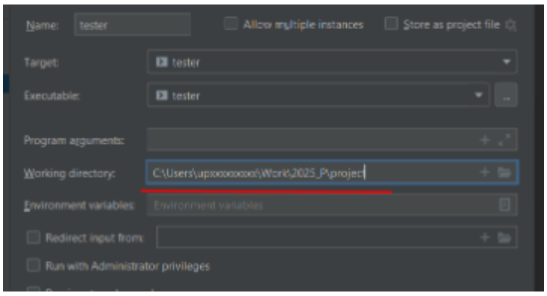
If you wish to recompile everything from scratch, execute `make clean all` within your `build` folder.

The structure of the `CMakeLists.txt` file already includes automatically all `.cpp` files in the `src` folder and all `.hpp` files in the `include` folder.

If you want to run the executables directly from inside CLion, you need to set, for each executable, the root folder of the project as the working directory.

Choose the executable, then “Edit Configurations…”, and finally put the path to the root of the project in the “Working directory”.




Name: tester
□ Allow multiple instances □ Store as project file
Target: tester
Executable: tester
Program arguments:
Working directory: C:\Users\upxxxxxxxx\Work\2025_P\project
Environment variables: Environment variables
□ Redirect input from:
□ Run with Administrator privileges

Project files

The project files and directories are divided in two sets: those that you can or should change (Table 1 below), and others that must not be changed (Table 2).

Table 1. Files/directories that can be changed.

File(s)/directory	Description
`CMakeLists.txt`	File to use when compiling the project.
`include/Color.hpp src/Color.cpp`	Header and code files for `prog::Color` .
`include/Image.hpp src/Image.cpp`	Header and code files for `prog::Image` .
`include/Command.hpp src/Command.cpp`	Header and code files for `prog::Command` .
`include/ScrimParser.hpp src/ScrimParser.cpp`	Header and code files for `prog::ScrimParser` .
`include/Command/...`	Header files for new commands.
`src/Command/...`	Implementation files for new commands.
`output`	Output images produced by scrims will be placed in this directory.

Table 2. Files/directories that *must not** be changed/removed.*

Files/directory	Description
`main/RunScrim.cpp`	Program that executes an image processing script.
`main/Tester.cpp`	Test program.
`include/Scrim.hpp src/Scrim.cpp`	Header and source files to represent an image script in **Scrim**.
`include/PNG.hpp src/PNG.cpp`	Header and source file for PNG image reading, writing and comparison.
`include/stb/stb_image.h include/stb/stb_image_write.h`	source code of a library called `stb` (https://github.com/nothings/stb) needed for the PNG support.
`input` directory	Directory containing image scripts and PNG images used as input.
`expected` directory	Directory containing PNG files that correspond to the expected outputs.

The runscrim program

The `runscrim` program can be invoked to process one or more image processing scripts, e.g.,

```
build/runscrim scrims/basic_blank1.scrim
```

or

```
build/runscrim scrims/basic_blank1.scrim scrims/basic_blank2.scrim
```

The tester program

The `tester` program can be invoked to execute one or more automated tests:

- Color tests
  `build/tester Color`
- Basic image I/O tests
  `build/tester basic`
- Test all scrims related to command `x`, where `x` is the command name.
  `build/tester x`
- Run all scrim tests — supply no arguments.
  `build/tester`

Project development

Implementation of Color

A `Color` object represents an RGB color (https://www.google.com/url?q=https://en.wikipedia.org/wiki/RGB_color_model&sa=D&source=docs&ust=1745605088935477&usg=AOvVaw317Q9xA8_2P4zpqerfVJP9), that is, a color defined by 3 integer values which are the values for the red, green, and blue color channels for the color at stake. Each of these values takes one byte and can take values ranging from 0 to 255, as defined by type `rgb_value` in `include/Color.hpp`.

What must be done?

You should define appropriate fields to represent the RGB values, and implement the member functions already provided in the initial skeleton. These are described in Table 3 below.

If you find it necessary, you may define other member functions in the class.

Validation

Execute `build/tester Color` in the command line. The test will fail until the code in `Color` meets the expected functionality, e.g.,

```
$ build/tester Color
Assertion failed: (a.red() == 1), function color_tests, file Tester.cpp, line 97.
[1]    20632 abort      build/tester Color
```

Table 3. `Color` member functions.

Function(s)	Description
`Color()`	Default constructor. By default, the color should correspond to black, i.e., `(0,0,0)`.
`Color(const Color& c)`	Copy constructor.
`Color(rgb_value r, rgb_value g, rgb_value b)`	Constructor using supplied `(r,g,b)` values.
`rgb_value red() const rgb_value green() const rgb_value blue() const`	Get values for individual RGB color channels.

Function(s)	Description
`rgb_value& red() rgb_value& green() rgb_value& blue()`	Get (mutable) references for individual RGB color channels.

Implementation of Image

An `Image` object represents an image. It has an associated width (`width()`) and height (`height()`), and must hold a 2D matrix of colors with these dimensions. Each `(x,y)` position in this matrix, where `0 <= x < width()` and `0 <= y < height()` , is called a **pixel**. Pixel `(0,0)` corresponds to the upper-left corner of the image, and pixel `(width()-1, height()-1)` corresponds to the lower-right corner of the image, as illustrated in Figure 2.

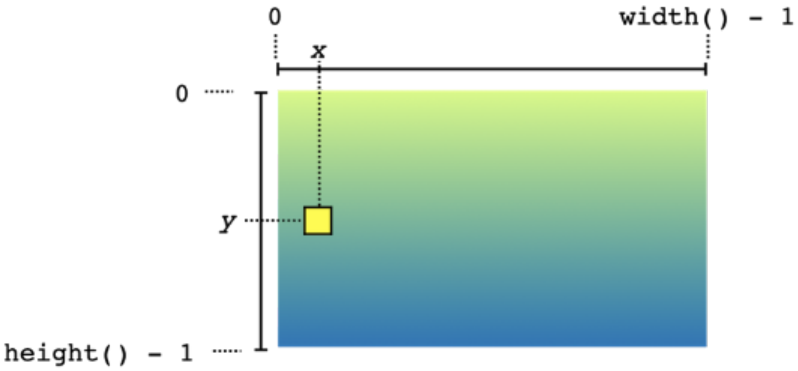


Figure 2. Coordinate system for images.

What must be done?

You should define appropriate fields to represent the image dimensions and the pixel matrix, and implement the member functions already provided in the initial skeleton. These are described in Table 4.

If you find it necessary, you may define other member functions in the class.

Validation

Execute `build/tester basic` in the command line. One or more tests will fail until the code in `Image` meets the expected functionality.

```
== 5 tests to execute  ==
[1] basic_blank1: fail
[2] basic_blank2: fail
[3] basic_blank3: fail
[4] basic_open1: fail
[5] basic_open2: fail
== TEST EXECUTION SUMMARY ==
Total tests: 5
Passed tests: 0
Failed tests: 5
```

Table 4. `Image` member functions.

Function(s)	Description
`Image(int w, int h, const Color& fill = { 255, 255, 255 })`	Constructor. Creates an image with width `w` , height `h` , and all pixels set to color `fill` . White is the default fill value, i.e., `(255,255,255)` .

Function(s)	Description
`~Image()`	Destructor. **If you use dynamically allocated memory explicitly, the destructor should take care of releasing that memory.** Otherwise, the destructor code can be empty.
`int width() const`	Get image width.
`int height() const`	Get image height.
`Color& at(int x, int y)`	Get mutable reference to the value of pixel `(x,y)` , where `0 <= x < width()` and `0 <= y < height()` .
`const Color& at( int x, int y) const`	Get read-only reference to the value of pixel `(x,y)` .

Implementation of Scrim

A `Scrim` object encapsulates the execution of an image processing script. The class has the public member functions described in Table 5. This code is fully implemented. However, it will be called by the partially implemented `ScrimParser` (Table 6) and it will use concrete `Command` instances which you will have to implement and place in folder `src/Command` (Tables 7-10).

Table 5. Public member functions in a `Scrim` object (already implemented).

Function(s)	Description
`Scrim(std::vector<Command*> &commands)`	Constructor, with a sequence of `commands` indicating the chain of manipulations to be applied. The commands should not be executed until `run()` is called (see below).
`~Scrim()`	Destructor. Destroys all internal commands when called.
`Image* run(Image *img)`	Applies all internal commands in sequence to an image `img` , and returns a transformed image.
`Image* run()`	Shorthand for the `run(Image* img)` function with no starting image.

Table 6. Public member functions in `ScrimParser` .

Function(s)	Description
`Scrim *parseScrim( std::istream &input);`	**[ALREADY IMPLEMENTED]** Parses an input scrim from a generic input stream.
`Scrim *parseScrim(const std::string &filename);`	**[ALREADY IMPLEMENTED]** Parses an input scrim from a file input stream, using the previous function.
`Command *parse_command( std::string command_name, std::istream &istream);`	**[PARTIALIY IMPLEMENTED]** Parses a single command called `command_name` from an input stream `istream` . You should extend this function to parse your new commands.

An image `Command` has a name and a virtual method to apply a transformation to an image. Its member functions are listed in Table 7. Concrete commands implement concrete transformations and extend the `Command` class. These are separated into three groups:

1. Commands for image initialization and PNG image I/O (**already implemented**), listed in Table 8;
2. Script commands for simple image manipulations, listed in Table 9; and

3. Script commands for manipulation that alter image dimensions, listed in Table 10.

Table 7. Public member functions in `Command` (virtual method can be overridden by subclasses).

Function(s)	**Description**
`Command()`	**[ALREADY IMPLEMENTED]** Default constructor, using default name “”.
`Command(std::string command_name)`	**[ALREADY IMPLEMENTED]** Constructor using supplied `command_name` for the name of the command.
`~Command`	**[ALREADY IMPLEMENTED]** Destructor.
`string name()`	**[ALREADY IMPLEMENTED]** Returns the name of the command.
`virtual Image* apply(Image* img);`	**[VIRTUAL]** Applies a transformation to a given image `img` and returns a transformed image. Needs to be defined in concrete subclasses.

Table 8. `Command` s for initialization and for I/O.

Command to parse	**Description**
`blank w h r g b`	**[ALREADY IMPLEMENTED]** Creates a new image with dimensions `w x h` and all pixels set to color `(r,g,b)` . The current image, if any, is discarded. Defined in file `include/Command/Blank.hpp` and implemented in file `src/Command/Blank.cpp` .
`open filename`	**[ALREADY IMPLEMENTED]** Reads a new image in PNG format from `filename` . The current image, if any, is discarded. Defined in file `include/Command/Open.hpp` and implemented in file `src/Command/Open.cpp` .
`save filename`	**[ALREADY IMPLEMENTED]** Saves current image in PNG format to `filename` . Defined in file `include/Command/Save.hpp` and implemented in file `src/Command/Save.cpp` .

Table 9. Missing commands for simple image manipulations (image dimensions are not altered).

Command to parse	**Description**
`invert`	Transforms each individual pixel `(r,g,b)` to `(255-r,255-g,255-b)` .
`to_gray_scale`	Transforms each individual pixel `(r,g,b)` to `(v,v,v)` where `v = (r+g+b)/3` . You should use integer division without rounding to compute `v` .
`replace r1 g1 b1 r2 g2 b2`	Replaces all `(r1,g1,b1)` pixels by `(r2,g2,b2)` .
`fill x y w h r g b`	Assigns `(r,g,b)` to all pixels of the image contained in the rectangle defined by the top-left corner `(x,y)` , width `w` , and height `h` , i.e., all pixels `(x',y')` such that `x <= x' < x + w` and `y <= y' < y + h` . Pixels outside the current image bounds should not be modified.
`h_mirror`	Mirror image horizontally. Swap pixels `(x,y)` and `(width()-1-x, y)` for all `0 <= x < width()/2` and `0 <= y < height()` .
`v_mirror`	Mirror image vertically. Swap pixels `(x,y)` and `(x, height()-1-y)` for all `0 <= x < width()` and `0 <= y < height()/2` .

Command to parse	Description
`add filename r g b x y`	Copy all pixels from an image stored in PNG file `filename` , except pixels in that image with “neutral” color `(r,g,b)` , to the rectangle of the current image with top-left corner `(x,y)` of the current image. The image should not be rescaled and pixels that do not fit in the current image bounds should not be copied.
`move x y`	Moves all the pixels horizontally by `x` amount to the right and vertically by `y` amount below. Values can only be positive, and will move the image to the right or down. Pixels that fall outside of the image bounds are discarded, and parts of the image that become without pixels get the `fill` color.
`slide x y`	Similar to `move` , but instead of discarding the pixels, they “warp” to the next available position. E.g., in an image with `w` \= 2 and `h` \= 2, `slide 1 0` will move the pixel at position `(1,1)` to position `(0,1)` and pixel at position `(0,1)` to position `(1,1)` .

Table 10. *Scrim** commands for dimension-changing operations*

Command to parse	Description
`crop x y w h`	Crop the image, reducing it to all pixels contained in the rectangle defined by top-left corner `(x,y)` , width `w` , and height `h` . Pixels that are not within the current image bounds should be ignored.
`resize x y w h`	Similar to `crop` . Updates the size of the image, without scaling it, keeping all pixels contained in the rectangle defined by the top-left corner `(x,y)` , width `w` , and height `h` . Pixels that are not within the current image are filled with the image’s fill color.
`rotate_left`	Rotate the image left by 90 degrees.
`rotate_right`	Rotate the image right by 90 degrees.
`scaleup x y`	Each pixel expands horizontally by integer factor `x` and vertically by integer factor `y` . E.g., an image with 3 pixels wide and 4 pixels tall, when scaled up with `x=2` and `y=3` , becomes 6 pixels wide and 12 pixels tall.

What must be done?

Implement a new class for each new command. The new class has two files, one `.hpp` file in folder `include/Command` and one `.cpp` file in the folder `src/Command` . After the class is implemented, write appropriate code to parse the commands in the file `src/ScrimParser.cpp` .

Validation

Execute `build/tester <command>` in the command line where `<command>` is the command that you wish to test. For instance, `build/tester invert` will test all scripts related to the `invert` command:

```
$ build/tester invert
== 4 tests to execute  ==
[1] invert1: fail
[2] invert2: fail
[3] invert3: fail
[4] invert4: fail
== TEST EXECUTION SUMMARY ==
Total tests: 4
Passed tests: 0
Failed tests: 4
```

Advanced functionality

Chaining commands

We have seen commands that transform a given image, possibly reading-from or writing-to a file. We would like to build a special command that reads other **Scrim** files and combines them.

Scrim should be extended with a command `chain <scrim-1> <scrim-2> … <scrim-n> end` that reads an arbitrary number `n` of scrim files until it finds the string `end` and executes them one after the other. Furthermore:

- All operations that save or discard the input image should be ignored. This means the commands `save` , `blank` and `open` ;
- Nested chains should be supported, i.e., a `chain` command can load a scrim that itself contains a `chain` command;
- Recursion in nested chains must be detected. If a chained scrim file is detected as being called by the second time in a given call chain, it should be ignored, in order to stop the infinite recursive loop. E.g., when executing the command `chain a.scrim` , and if loading it eventually leads to another chain call to `a.scrim` , then these new (recursive) calls to `a.scrim` must be ignored.

Validation

Execute `build/tester chain` in the command line.

What to submit

Near the deadline, a form will be made available in Moodle for project delivery. You will need to deliver **a ZIP file named delivery.zip**, containing the updated zip source and header files. To generate the ZIP file run `make delivery` in your build folder.

```
$ cmake -B build
$ cd build
$ make delivery
[100%] Creating zip archive: /.../build/delivery.zip
[100%] Built target delivery
```

Evaluation criteria

1. **[85 %]** Correctness of implementation — this means the code should work as expected and run without memory errors (buffer overflows, memory leaks, dangling references, etc), with the following components:

   - **[10 %]** Basic color and Image representation

   - **[35 %]** Simple image manipulations

   - **[30 %]** Dimension-changing operations

   - **[10 %]** Advanced functionality

2. **[15 %]** Well-structured, as simple as possible, commented code; appropriate use of variables and member fields.