****

# School of Computing: Assessment brief

|  |  |
| --- | --- |
| Module title | Programming Project |
| Module code | XJCO 1921 |
| Assignment title | PGM Utilities – Coursework 1 |
| Assignment type and description | In-course assessment  Require understanding specification, learning how to use software development tools, and applying good programming practices |
| Rationale | Provides an opportunity to practice tools used to design, implement and test a mini-project to apply good programming practices. |
| Word limit and guidance |  |
| Weighting | 40% |
| Submission deadline | 03/04/23 |
| Submission method | Minerva |
| Feedback provision | Oral feedback will be provided during the lab demonstration session. |
| Learning outcomes assessed | Apply professional programming practices to programming projects.  Design, implement, debug and test a modular programming solution to a real-world problem.  explain the importance of applying professional programming practices to programming projects. |
| Module lead | Zheng Wang |
| Other Staff contact | Xuying Yu (SWJTU module co-lead) |

XJCO 1921- Programming Project

Coursework 1 – PGM Utilities

**Deadline:** 12 PM BST on Monday, 3 April 2023

This work is the first coursework for this module. It corresponds to **40%** of the overall assessment for this module. Submissions should be made via Minerva.

# Course Specification

The goal of this assignment is to develop your skills in writing code systematically and well, with due attention to commenting, structure, modularity, defensive programming, version control and good programming habits.

**1. Coursework Specification:**

You are asked to extend the pgmEcho utility given on Minera and briefly discussed in the lecture. You will construct a set of Programs (or utilities) that manipulate PGM files in a number of ways.

A PGM is a standard bitmap-based format consisting of a 4 lines header and data stored in the unsigned char type, providing a maximum of 256 greyscale levels or 8-bit data per pixel. A short description of the format can be found at: <https://users.wpi.edu/~cfurlong/me-593n/pgmimage.html>

**THE TASKS:**

There are six sub-tasks, labelled I.-VI, each worth 5 or 10 marks.

1. Analyse pgmEcho.c to identify how it should be improved, then rewrite it so that it is more modular, more suitable for reuse, and properly defensive in terms of errors. Note that this should involve understanding the remaining tasks as well.

(**5 marks**)

1. Write a Program called pgmComp which takes two pgm **file names** as arguments, reads them in and tests whether they are **logically identical**.

(**5 marks**)

1. Modify pgmEcho and pgmComp to accept binary PGM as well as ASCII, and write utilities called pgma2b to convert the ASCII form to the binary, and pgmb2a to convert the binary form to the ASCII.

**(5 marks)**

1. Write a Program called pgmReduce which takes 3 arguments:

* an input file (either ASCII or binary)
* an integer factor n, and
* an output file

and reduces the input file by that factor in each dimension.

(**8 marks**)

For example, if the invocation is:

pgmReduce inputFile 5 outputFile

and inputFile is a 14x16 image, then outputFile should be a 3x3 image which only contains pixels whose row and column indices modular (%) 5 is 0. In other words, the reduced image only contains pixels at (0, 0), (0, 5), (0, 10), (0,15), (5, 0), (5, 5), (5, 10), (5, 15), (10, 0), (10, 5), (10, 10), (10, 15) of the inputFile.

As another concrete example, if we have an input image that is 14x16 with the following hex values for pixels:

 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F

 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F

 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F

 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F

 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F

 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F

 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F

 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F

 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F

 A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF

 B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF

 C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF

 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF

The reduced image with a factor of 5 should then contain the highlighted values of the input

00 05 0A 0F

50 55 5A 5F

A0 A5 AA AF

1. Write another program called pgmTile which takes 3 arguments, an input file (either ASCII or binary), an integer factor **n**, and an output file name template. It should divide the input file into **n x n** smaller images corresponding to parts of the image.

For example, suppose that the invocation is:

pgmTile sampleFile.pgm 3 sampleFile\_<row>\_<column>.pgm

Then, the program should read in sampleFile.pgm and output 9 smaller images named:

sampleFile\_0\_0.pgm

sampleFile\_0\_1.pgm sampleFile\_0\_2.pgm sampleFile\_1\_0.pgm sampleFile\_1\_1.pgm sampleFile\_1\_2.pgm sampleFile\_2\_0.pgm sampleFile\_2\_1.pgm sampleFile\_2\_2.pgm

i.e. substituting the relative position of the subimage in the larger image into the <row> and

<column> tags in the template name

In this example, each of the output P files corresponds to a tiled image in a 3 x 3 grid applied to the input image.

|  |  |  |
| --- | --- | --- |
| **0\_0** | **0\_1** | **0\_2** |
| **1\_0** | **2\_1** | **1\_2** |
| **2\_0** | **2\_1** | **2\_2** |

(**7 marks**)

**RETURN VALUES AND ERROR CODES:**

All Programs should return the value 0 in case of success, non-zero in case of failure, and a string to describe the result. This is a standard Unix/Linux convention: we return 0 on success because there is often only one way the Program can succeed but many ways it can fail, which allows us to use the return value to indicate what type of failure.

There is an exception to this: if no arguments are given, the Program should return a special message indicating the correct usage and return 0. This message (which is also a standard convention) is a minimal help message that starts with the string "Usage: " followed by the name of the executable (which can be found in argv[0]) and a brief description of the correct parameters. In this case, input files should be called "inputImage" and output files should be called "outputImage", so the forms are:

Usage: ./pgmEcho inputImage.pgm outputImage.pgm Usage: ./pgmComp inputImage.pgm inputImage.pgm Usage: ./pgmb2a inputImage.pgm outputImage.pgm

Usage: ./pgmb2b inputImage.pgm outputImage.pgm

Usage: ./pgmReduce inputImage.pgm reduction\_factor outputImage.pgm

Usage: ./pgmTile inputImage.pgm tiling\_factor outputImage\_<row>\_<column>.pgm

The string to be output on success depends on the Program:

**Value String Program Meaning**

0 ECHOED pgmEcho Program echoed the input

0 IDENTICAL pgmComp The two files were identical

0 DIFFERENT pgmComp The two files were not identical

0 CONVERTED pgma2b/pgmb2a The file was converted

0 REDUCED pgmReduce The file was successfully reduced in size

0 TILED pgmTile The tiles were successfully written

**Value String Condition**

0 ERROR: Bad Argument Count Program given wrong # of arguments

|  |  |  |
| --- | --- | --- |
| 2 | ERROR: Bad File Name (fname) | Program failed to open a file stream |
| 3 | ERROR: Bad Magic Number (fname) | Program failed on a magic number |
| 4 | ERROR: Bad Comment Line (fname) | Program failed on comment line |
| 5 | ERROR: Bad Dimensions (fname) | Program failed on image dimensions |
| 6 | ERROR: Bad Max Gray Value (fname) | Program failed on max gray value |
| 7 | ERROR: Image Malloc Failed | Malloc failed for image allocation |
| 8 | ERROR: Bad Data (fname) | Program failed when reading in data |
| 9 | ERROR: Output Failed (fname) | Program failed during output |
| 10 | ERROR: Bad Layout | Layout file for assembly went wrong |
| 100 | ERROR: Miscellaneous (text) | Any other error that is detected. |

In each case, fname should be replaced by the name of the file that failed. Similarly, in a miscellaneous error, "text" should be replaced by a description in no more than 100 characters of the error detected.

**Note that we do NOT guarantee that this is an exhaustive list of possible errors.**

**DEVELOPMENT MACHINE:**

Our example code is tested on Linux using gcc as the compiler, and we will not provide support if you use a Windows system. We recommend that you work in Linux, using vim, nano or emacs for command-line editing until you are comfortable using these tools correctly in future. After that, if you wish to use GUI-based text editors, you may.

**2. Good Programming Practice:**

You should follow good software development practices. For this exercise, you are asked to:

* Follow modular development by making sure the code is modular and well structured; code with proper comments
* Use Makefile for code compilation;
* Use a git repository for version control

**Notes**:

1. Your Program must have more than one module (C file).
2. The Program should provide adequate error handling, i.e., display meaningful messages if an error occurs and handle common errors. We will be running our own test cases to mark the assignments and doing our best to devise creative ways to break your code! You should therefore develop your test cases to beat us to it.
3. Your code should not hard code any relative paths. All file names should be specified by command-line arguments. This will allow us to test any file location for input and output.
4. **Makefile**: You should also submit a Makefile with at least two targets: *all* and *clean*. "*make all*" compiles your code to generate an executable binary, while "*make clean*" removes all object files (.o), all executables, and any *temporary* files created during your run.
5. **Version control**: We will check the commit logs of your git repository. We expect to see steady progress towards completion, as revealed in the pattern of git commits. One of the implications of this is that we will be penalising any student who develops their code without git and then dumps it all into git at the last minute.

**Assessment/marking criteria grid**

* (5 marks) Task I
* (5 marks) Task II
* (5 marks) Task III
* (8 marks) Task IV
* (7 marks) Task V
* (10 marks) Code quality and programming practices, including a self-reflection of the project. During marking, you will be asked to look back and reflect on your project development. Please avoid generic statements about time management. Focus on your C coding, design, and testing processes.

This coursework is worth 40% of the final grade, divided between the tasks outlined above. We specifically set aside a separate percentage mark (10%) for code quality. If you complete I.-V. flawlessly, your base mark will be 30/40. But if your code quality is poor (< 50%), this will be capped at 20/40.

Notice that Task V is more challenging than Tasks I.- IV. This is because they are intended to push the abilities of the class. However, it worthies fewer marks, which means that a student who completes I.-IV. with reasonable (60%) code quality can expect a passing mark for this assignment. A student who completes I.-IV. with high (80%) code quality can expect a good mark. And a student who completes I.-IV. meticulously (with perfect code) can expect 40/40 overall, and will already have passed the module, with the marks in Coursework 2 moving them up into higher classifications.

# Submission

Your work should be submitted to the submission portal on Minerva.

Your submission should be in a directory in the root of your git repository called "assignment\_1" and contain files that conform to the assignment specification. Your submission should contain all the relevant source code and header files, a brief README file explaining how to run your submission and the URL to your git repository, and a Makefile which builds all of the executables specified.

It is your responsibility to ensure that your submission compiles and executes on the lab machine. Your git commit and push must be made prior to the submission deadline.

You should also include a test script called "testscript.sh" to automatically execute your test cases and any test files you rely on.

In summary, your hand-in should include:

* 1. All necessary .c and .h files
  2. A makefile as described above
  3. A readme file called "readme.txt"
  4. A test plan file called "testplan.txt"
  5. A screenshot of your git commit history
  6. Any test data files which you use for your test script.

With the exception of the test files, all of these files must be in the assignment\_2 directory. You may place the test data files in a subdirectory if you wish.

Submit all your files in a single zip (.zip or .gz) file through Minerva. If there are ANY other files (e.g., object files or executable binary) or subdirectories in the directory when it has been handed in, you will lose marks (from the programming practices category). This includes any hidden file (prefixed with .), except for .git and .gitignore.

*You should follow the instructions below on how to prepare your submission. Late submissions are accepted up to 7 days late. Each day, or part of a day, will incur a 5% penalty.*

Important notes on the submission:

* Write the Program in standard C. If you write your code in any other language, it will not be assessed, and you will get a zero mark.
* This is an ***individual project***, and you are not supposed to work in groups or pairs with other students.
* Be aware that plagiarism in your code will earn you a zero mark and will have very serious consequences. If two (or more) students have large portions of their files nearly identical, they will be accused of plagiarism or collusion. If found guilty, all parties involved will incur the penalty, regardless of who was the author of the code. For this reason, never show or give access to your code to anyone. Do not help a colleague by sharing your code, or you will both be found guilty of collusion.
* It is your responsibility to make sure that nobody has access to your code. Lock the session if you leave your computer unattended.
* Make sure to download and check your submission. Corrupted files, binary files, wrong versions, copies of your project (over the years, we have seen it all), or anything other than what is requested in this document will be considered invalid submissions.
* We will not accept submissions other than through Minerva.

# Marking and Demonstration

* This exercise will be marked during a lab session after the submission deadline.
* You will be asked to demonstrate your work ***from your Minerva*** ***submission***.
* You need to demonstrate your work to a member of the course team; failing to do so will result in 0 marks.
* You need to come to the marking session with your completed exercise. We will not be able to provide support for this exercise during marking.
* You should be able to explain what you have done clearly, to show that you understand the concepts introduced.
* Checks for plagiarism and collusion will be carried out on all work.