# Error Codes

Updated for “What’s the language?” Assignment

This table provides detailed explanations of the error codes indicated in the comments on your assignment. These codes were originally set for the letter frequency assignment, but have been updated for later assignments 😊!

Codes which are very specific to one assignment are now qualified with an identifier for the assignment, e..g; LF = Letter Frequency

Codes with no qualifier should be considered **generic** .. and applied to ANY assignment.

Generic comments may be the basis of exam questions: make sure you understand them – even if they did not appear on your assignment!

|  |  |
| --- | --- |
| **Code** | **Explanation** |
| **A** | **Algorithm** |
| A1  LF | Since there are a fixed number of characters (defined by Unicodes and therefore permissible in a Rust string), the simplest solutions creates a structure with the character (redundant, but simplifies code), the count of instances of it and the relative frequency.  **struct letters {**  **c: char, count: u64, freq: f64,   }**  **const uc\_low: xxx; // Lowest unicode**  **const uc\_high: yyy; // Highest unicode**  Works out how many table entries are needed (one for each Unicode in the target language  **const NC: uc\_high-uc\_low+2; // numer of codes + 1 for unspecified**  Then creates a table of this structure  **l\_table:[letters;NC];**  Then it reads each character from the file, one by one, converts it to a hex number, and works out the index in the table  **let ix:usize = cc\_hex – uc\_low;**  and increments the count in that position in the table**.**  After the whole file has been read, it makes another pass to calculate the frequency.  Then prints the table.  An example of this is found on the web – for Latin characters, but trivially changes to any other language by finding the low and high Unicodes for that language.  Following the directions, it counts any unrecognizable characters separately. You could make on separate structure for them OR add an extra final entry in the table. |
| A 2  LF | Since there are fixed numbers of characters to be counted, you should build a fixed tabel of lenth (number of legal ones + 1 – for anything else). A vector is not appropriate here. |
| A 3 | Your array can be fixed AND you can **index it directly** using the difference in the unicode value and the unicode for the first letter (in Thai 0E01 for ก). See Step 1 of the instructions. |
| A 4 | Design error handling into your program: routinely add code   * At the end of a loop – to check that the condition in the loop was satisfied * For the unmatched condition in a match statement   that outputs appropriate messages.  This ensures that you can quickly find problems and know that you have fixed them! |
| **RI** | **Not reading the instructions carefully** |
| RI1 | You were told to find the unicodes for your language and use that range of ‘legal’ characters. From the min an max unicodes, you know how many table entries are needed. |
| RI2 | Not counting all ‘illegal’ characters separately – add an entry at the end of the ‘legal’ table and count them there. |
| RI3 | Ignoring hints: Reading files with Rust is somewhat painful, but reading from stdin with  **cargo run <text.txt**  is about as simple as anything in Rust ever becomes! |
| RI4 | Adding unneeded crates (and unneeded complexity) – see also RI3 |
| RI5 | Keep it simple .. you were required only to count characters. Parsing into words in Thai is difficult. Some colleagues in KMITL’s School of IT will tell you how to find systems that do this though! |
| RI6 | The instructions told you that this was a **one person** assignment. At the end, you were to compare results for your text with those from your collaborators. |
| RI7 | Do not write ‘we’ in a individual assignment – write (correctly) ‘I’.  Using ‘we’ in the *final* comments, comparing your results with those of your collaborators, was OK though. |
| RI8 | If you had one (or more) collaborators, you should have had TWO (or more) files to discuss in your comments. |
| RI9 | Follow the instructions on the template: if you worked together, you are co-authors;  If you just compared your work with someone else, you are collaborators. |
| RI10 LF | You need several 1000 (or many more) characters to get sensible letter frequencies. Testing with a small sample is excellent, but when it is working, feed a much larger text to it. |
| RI 11 | Failure to check the test file OR report your program’s output when running it |
| C | **Coding** |
| C1 | Failing to add parentheses in complex expressions. The poor advice from Rust’s compiler provides further evidence that Rust is run by a community of hackers. Actually bothering to tell you to leave something out, that is legal code, but which saves you making a mistake (if you forgot the precedence rules) is simply bad advice! |
| C2 | Not submitting your code – so no marks can be assigned for code quality |
| C3 | You should have constants for the min and max unicode for your language. They are available from web tables (or your earlier assignment) |
| C4 | Not separating operations into separate functions *or* the 10000 line main function syndrome |
| C5 | Clean code makes efficient use of boolean variables. If you have  **let mut exist : bool = false;**  and set it true or false later, you can just write  **if exist { … }**  whereas  **if exist == true { … }** looks a bit silly 😊 |
| C6 | Do not repeat any code!! Even a single line, with a moderately complex expression in it, is usually best put in a separate function. |
| C7 | Magic numbers in code: most literals should be implemented as intelligible named constants! This is important for later maintenance. |
| C8 | Making your code easy to read: Strongly recommended   1. Don’t use ‘l’ and ‘i’ as variable names: ‘j’, ‘k’, etc are less likely to be confused with 1 😊 2. Also, your program should be universal .. don’t use names like ‘non\_thai’, use generic, transportable names, eg ‘non\_printing\_char’ or similar.   See also L2 |
| C 9 | Boolean flag for ‘thai\_or\_not’ is not needed IF the unicode range was used as strong hint in instructions was followed! |
| **D** | **Discussion** |
| D1 | Where did your expectations come from?  In this case, maybe you can compare your frequencies with published ones.  Maybe your results are also aligned with the positions of characters on a Thai keyboard 😊  As a simple ‘accuracy’ test you were required to compare your table with another classmate and discuss (briefly) any differences. |
| D2 | Don’t tell us stories about fixing errors .. it is assumed that you fixed them all before you started on this report! |
| D3 | Accuracy of your results ..  If you only used a 1000 char test file, then your result is, at best, accurcate to 0.1%, so numbers should be reported to **at most 3 significant digits**. Even if your input has 105 characters, realistic accuracy is still only about 3 digits. |
| D4 | A clear referemce to your source (e.g. as a URL or book citation) expected. If a ‘private’ source (your own long essay 😊 ) it should be attached. |
| D5 | You were directed to compare your output with that of your collaborator(s) and report differences, e.g. different most common characters. |
| D 6 | Distinguish between **ERRORS** and **LIMITATIONS** of the problem!  For example, in the “What language?” problem, the specification was simple and would never distinguish between English and most European languages or Malay o Tagalog or between Chinese and Japanese.  Do not describe such limitations as errors. Report precisely – see also L5.  Here an error would include failing to read the whole file or crashing unexpectedly. |
| **L** | **Technical writing** |
| L1 | Unfortunately the expressions that you may have learnt from Hollywood movies are often **NOT** good examples for formal writing.  Avoid colloquial expressions and write formally.  Colloquial expression include ‘got’, ‘done’ and ALL contractions – isn’t, wasn’t, didn’t, ‘gonna’, .. etc – write them out in full – is not, was not, did not, … |
| L2 | Do not put light coloured text on a white background in a technical report. Use a strongly contrasting colour – black is boring, but much more easily read! Similarly, grey on black wastes your reader’s time .. 100% white on black works but uses excessive amounts of ink ☹. |
| L3 | Stating the obvious  Some things are expected for ***ANY*** experiment or report, e.g. reading the instructions, studying the language, etc, DO NOT put these in your methods section, save your words (and readers’ time) for things that are particular to this report. |
| L4 | Be kind to your readers:   1. Don’t spread your table over several pages with multiple blank lines. This makes following your report harder and wastes space and paper in a printed report. Removing the unneeded carriage returns is easy! Formatting lines for readibility requires some extra Rust understanding ☹ .. not covered yet .. just make it reasonable. 2. Make text large enough to be easily read - 12 pt is good, 10 pt is smallest acceptable 3. Don’t put your output on Google drive (or anywhere else) unless specifically requested. .. |
| L5 LF | Distinguish carefully between ‘**error**’ and ‘**variation**’ or ‘**discrepancy**’. An error implies your code was wrong and produced a wrong result. In this case, **variations** were expected from different sources (and your report was asked to comment on them), so they are not errors! However, it would be an error if two programs had different frequencies for the same input! |
| L 6 | Submit program code as separate files.  This enables them to be mechanically checked by the compiler.  Also, some compilerrs seem to be providing code listing in colours on a black background that are hard to read! But your xx.rs file is easily read in any text editor.  Only put short fragments of your code into your report files! |