Inlämningsuppgift, EDAF30, 2024

1 Anvisningar för redovisning

Inlämningsuppgifterna ska redovisas med en kort rapport och de program som du har skrivit. Gör så här för att lämna in inlämningsuppgiften:

1. Arkivera lösningen (med zip eller tar, men *inte något annat arkivformat* som t ex rar), med rapport (som .pdf) och kompilerbar källkod, Makefile, och eventuella indatafiler. Var noga att inte ta med genererade filer (t ex `.o` eller `.exe`) i arkivet. Koden ska gå att bygga med `g++` eller `clang++` och `make` (samt eventuelt `cmake`).
2. Undvik svenska tecken, mellanslag, och specialtecken (som t ex +, *, !, ?, etc.) i filnamn eftersom dessa kan ha speciella betydelser i vissa system och orsaka problem.
3. Instruktioner för hur uppgiften ska skickas in publiceras senare.

2 Krav på uppgiften

2.1 Rapport

Uppgiften ska redovisas med en kort (ett par sidor) rapport som översiktligt presenterar din lösning. Följande punkter ska diskuteras:

1. Övergripande design: beskriv klasser och funktioner, och deras relationer till varandra.
2. En kort användarinstruktion: hur bygger och testar man programmet? Försök att paketera så mycket som möjligt med regler i makefilen. Det underlättar både under ert arbete och gör att denna instruktion blir väldigt enkel att skriva.
3. Brister och kommentarer: Finns det något i lösning som du i efterhand anser borde gjorts annorlunda? Andra kommentarer?

Rapporten ska lämnas in som pdf-fil.

2.2 Program

Er lösning ska naturligtvis fungera och lösa den angivna uppgiften. Testning ingår som ett krav i uppgiften, och både hur väl testerna täcker uppgiften, och hur väl programmen fungerar bedöms. Provkör allting ordentligt innan ni lämnar in er lösning.

Exempelprogram och testprogram

Det är ett krav att det ska finnas dels enhetstester för alla ingående delar (klasser och funktioner), och dels ett exempelprogram som visar hur er lösning fungerar genom ett exempel som producerar någon sorts utdata i terminalen, och eventuellt är interaktivt. Testprogrammen ska vara skrivna så att det inte krävs någon manuell kontroll av utdata för att avgöra om testet lyckades eller misslyckades: varje testfall ska – på något sätt – svara "ja" eller "nej". Det ska även finnas ett huvud-testprogram som kör alla enhetstester och tydligt visar vilka som inte uppfylldes.

Generella krav

Programkoden i lösningen ska uppfylla följande krav:

1. Programmet ska ha en vettig design.
2. Klasser och funktioner ska ha tydligt avgränsade uppgifter och ansvarsområden.
3. Minneshantering ska vara korrekt: programmet får inte läcka minne.
4. Programkoden ska vara lätt att följa och förstå.
5. Koden ska vara formatterad på ett sätt som underlättar läsning. Detta innebär en vettig indentering och att raderna inte är för långa.
6. Funktions- och variabelnamn ska vara väl valda och återspegla funktionens eller variabelns innebörd.
7. Programmet ska kompilera med `-Wall -Wextra -Werror -pedantic-errors`

   En tumregel, både för design och läsbarhet, är att en funktion inte får vara längre än 24 rader eller ha fler än 5–10 lokala variabler eller mer än tre indenterings-nivåer. Det finns ibland goda skäl att göra ett undantag från detta, men ofta är det bättre att dela upp en lång, komplex, funktion i några enkla hjälpfunktioner. Varje funktion ska bara göra *en* sak, gör den flera – dela upp den.

3 Rättning

Vi rättar uppgifterna så snart vi hinner, normalt inom en arbetsvecka räknat från inlämningsdag. När uppgiften är rättad får du besked om uppgiften är godkänd eller ej. Du får en kort sammanfattande kommentar om din lösning samt i de fall uppgiften inte är godkänd kommentarer om vad som behöver förbättras. Om uppgiften inte är godkänd ska du inom rimlig tid lämna in en förbättrad version.

Functors and algorithms: filter a word list

1 Background

There are standard library algorithms (e.g., `std::copy_if`, `std::remove_if`, `std::count_if`, `std::all_of`, and `std::any_of`) that take a predicate, instead of a value, for selecting elements.

The predicates can be normal functions, but a very powerful way for creating parameterised predicates is to use *function objects* – objects that have the function call operator, `operator()`.

We will explore the use of standard algorithms and predicates, together with the string and stream classes presented in the labs, by writing a tool for solving Wordle [1] puzzles.

Wordle

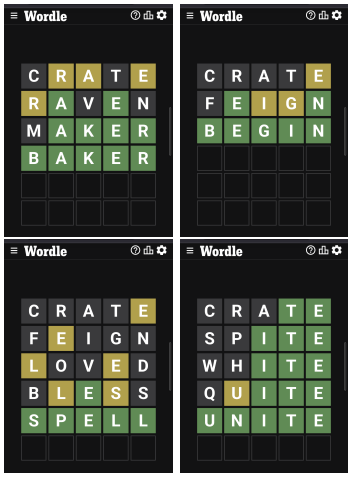
Wordle is a game where the goal is to guess a five-letter word. You start by making a guess, and then, for each letter in the word you guessed it is indicated if the letter is

a) the correct letter at the correct place (green),

b) in the solution, but at another place (yellow), or

c) not in the solution (grey).

Then you make another guess until you find the solution (or run out of attempts) as shown in Figure 1.



Figur 1: Examples of solved wordles

As an example we take an instance of the game where the solution is the word “BAKER”, as shown in figure 1 If the first guess the player makes is “CRATE”, the letters “C” and “T” will be grey, and “R”, “A”, and “E” will be yellow.

If the next guess is “RAVEN”, the letters “V” and “N” should be added to the grey list, “A” and “E” are green, and “R” yellow (in a new position). At this point, the green set contains the information that “A” and “E” are in the solution at indices 1 and 3 (zero-based), respectively. Likewise, the yellow set contains the information that “R” is in the solution but not at indices 0 or 1.

If the next guess is “MAKER”, the letter “M” is added to the grey set, and the letters “AKER” are added to the green set with the corresponding indices. That leaves, with the word list used when writing this, the candidates

```
baker daker faker laker raker saker waker
```

and the user can guess that “baker” is the one of the more likely solutions.

[1] popular on the web at `https://www.nytimes.com/games/wordle/index.html`

2 A wordle solving tool

Your task is to write a program to help solving wordle by filtering out words from a list of candidates. The approach is as follows:

- Generate the candidates by taking all five-letter words from a word list.

and then, iteratively

- Get input from the user about what letters are known to be at the correct place ("green"), part of the solution, but at the wrong place ("yellow"), or not part of the solution ("grey").
- Filter the candidates list, removing the words that cannot be the solution.

A series of steps towards implementing such a wordle tool is outlined below. In each step, write unit tests for each part to make sure that they work as expected before continuing.

1. Implement a function
   ```
   std::vector<std::string> read_candidates(std::istream &);
   ```
   that opens a word list from the an `istream` and gets all five-letter words. All words should be made lower-case and duplicates removed. [2]

2. For the predicates for grey, green, and yellow letters, first write helper functions like:
   ```
   bool contains_any_of(const std::string& s, const std::string& cs);
   ```
   which returns true if the string `s` contains any of the characters in `cs`, and
   ```
   bool contains_at(const std::string& s, char c, size_type pos);
   bool contains_but_not_at(const std::string& s, char c, size_type pos);
   ```
   which return true if the string `s` has the character `c` at position `pos`, or contains `c` but at any other position than `pos`, respectively. Then, to filter the candidates, the we will use classes to create predicates for including or excluding a word from the set of candidates.

3. First, to handle the grey letters, write a functor that is created with a list of letters[3] and then becomes a function that determines if a string contains any of those letters:
   ```
   struct wrong_fn {
       wrong_fn(const std::string& letters) : l{letters} {}
       bool operator()(const std::string& c);

     private:
       std::string l;
   };
   ```
   Then, with the helper functions from step 2 and the type aliases
   ```
   using size_type = std::string::size_type;
   using letters_and_indices = std::map<size_type, std::string>;
   ```
   implement two functor classes[4]:
   ```
   struct correct_fn {
       correct_fn(const letters_and_indices& idxs) : m{idxs} {}
       bool operator()(const std::string& c);
     private:
       letters_and_indices m;
   };
   ```
   which determines if a word contains all the green letters at the correct places, and

[2] *Hint:* Use `std::copy_if` with a pair of `std::istream_iterator<std::string>`, `std::sort`, `std::unique`, `std::for_each` (or a range-for loop) and `std::transform` with `tolower`.

[3] Here we use the class `std::string` to represent a list of letters. This may simplify the code a bit compared to using a set of characters.

[4] *Hint:* For `correct_fn`, you may use `std::all_of`. For `misplaced_fn` a range-for loop may be the simpler option.

```
struct misplaced_fn {
    misplaced_fn(const letters_and_indices& idxs) : m{idxs} {}
    bool operator()(const std::string& c);
  private:
    letters_and_indices m;
};
```

which does the same for the yellow ones. Implement the functor classes[5,6]. The implementations must use the standard library: use standard library algorithms, or member functions of standard classes, unless you have a good motivation for why you need to write your own implementation.

4. With the function objects for determining if a word contains grey, green and yellow letters, we can create a function object for the predicate "cannot be the solution". This takes a list of wrong (grey) letters, and maps from indices to a list of letters (green and yellow, respectively) at that index:

```
struct exclude_word {
    exclude_word(const std::string& wrong,
                 const letters_and_indices& correct,
                 const letters_and_indices& misplaced);
    bool operator()(const std::string& w);
  private:
    wrong_fn wrong;
    correct_fn correct;
    misplaced_fn misplaced;
};
```

With such a predicate, write a function that filters the candidates, removing the words that cannot be the solution. Here, `std::remove_if` is useful.

5. For the user interaction, get the set of green and yellow letters from the user by prompting the user to input whitespace separated pairs of (letter index). For instance, if the green letters were `LI-E-` that would be input as the string `"l 0 i 1 e 3"`. With a helper function:

```
letters_and_indices build_list(const std::string& line);
```

which reads that format, the function that prompts the user for input can then look like:

```
std::tuple<std::string, letters_and_indices, letters_and_indices>
prompt()
{
    std::string wrong;
    std::cout << "enter wrong letters:\n";
    std::getline(std::cin, wrong);

    std::string correct;
    std::cout << "enter correct letters (letter index)*:\n";
    std::getline(std::cin, correct);
    auto corr = build_list(correct);

    std::string misplaced;
    std::cout << "enter misplaced letters (letter index)*:\n";
    std::getline(std::cin, misplaced);
    auto misp = build_list(misplaced);

    return {wrong, corr, misp};
}
```

Implement the prompt function.

[5] The class interfaces are identical, so they could be generalised to a single class by making the actual function to be performed by `operator()` a parameter. Here, two separate functor classes are used to make the code less complex.

[6] Note that the use of a list of green letters for an index is overly general, but done to enable reuse of functions for manipulating objects of the type `letters_and_indices`. You may assume (or assert!) that all the strings in the greenmap have length `==1`.

6. Now, we can write a function
   ```
   void do_filter(std::vector<std::string>& c, std::string wrong,
                  letters_and_indices green, letters_and_indices yellow);
   ```
   that takes the candidate vector, and the grey, green, and yellow sets of letters, creates the corresponding functors, and removes the candidates that cannot be in the solution.

   To test your function, you can use the same word list you used in the spell-checker lab, the examples in Figure 1 and the following example sets. For the given input, candidates should include the word after `-->`:

   grey: `"crtin"`, green: `"a 0 l 1 e 3"`, yellow: `"l 0 e 1 a 4" --> "alley"`

   grey: `"cratdnmkd"`, green: `"l 0 i 1"`, yellow `"e 1 i 3 e 4" --> "libel"`

   grey: `"cratbluswd"`, green: `"e 3 i 1"`, yellow: `"n 4" --> "piney"`

7. Finally, to enable an iterative solution, the program should be able to repeatedly prompt the user for more characters, and add that to the grey, green, and yellow sets. For the grey set, there is `std::string::append`, but for the map type, a helper function is useful:
   ```
   void append(letters_and_indices& dest, const letters_and_indices& src);
   ```
   Implement such a function, and write a main program that repeatedly prompts the user for grey, green, and yellow lettersand prints the remaining candidates, until the user doesn't give more input, or there is zero or one candidate left.

   Note that if a letter is in the green or yellow sets, and a guess is made with that same letter in a wrong position, wordle will mark that letter grey if there is only one occurence of that letter in the solution, as seen in the "SPELL" example in Figure 1. However, adding the letter to the grey set will – in the filtering approach described here – also exclude the solution. To fix that, any letter present in the green or yellow sets must not be included in the grey set.

3 Summary

The task is to write a terminal-based, interactive tool for solving Wordle puzzles, which fulfills the following requirements:

- The tool shall be interactive and iterative. The program shall prompt the user for more input until there is at most one candidate left, or the user gives no more input.
- The tool must be able to use an arbitrary word list. The file name for the word list shall be a command line argument. See exmple code from lab 1 for how to use command line arguments.
- The user dialog in the program shall be easy to use and include proper error handling. Wrong input must not make the program crash.
- The program shall be case insensitive: words shall be found in the word list independent of any capital letters there, and independent of the case used in user input.
- The implementation shall use the standard library when possible. If you write your own implementation of something that is available in the standard library, you must motivate why that is better.
- There shall be unit tests for all functions and classes. There shall be a separate program that runs all unit tests. The test program shall determine and display if all tests pass, or which tests fail.