COMP 2012H Honors Object-Oriented Programming and Data Structures

Assignment 7 PortaWordle

Honor Code

We value academic integrity very highly. Please read the Honor Code section on our course webpage to make sure you understand what is considered as plagiarism and what the penalties are. The following are some of the highlights:

- Do NOT try your "luck" - we use sophisticated plagiarism detection software to find cheaters. We also review codes for potential cases manually.
- The penalty (for **BOTH** the copier and the copiee) is not just getting a zero in your assignment. Please read the Honor Code thoroughly.
- Serious offenders will fail the course immediately, and there will be additional disciplinary actions from the department and university, upto and including expulsion.

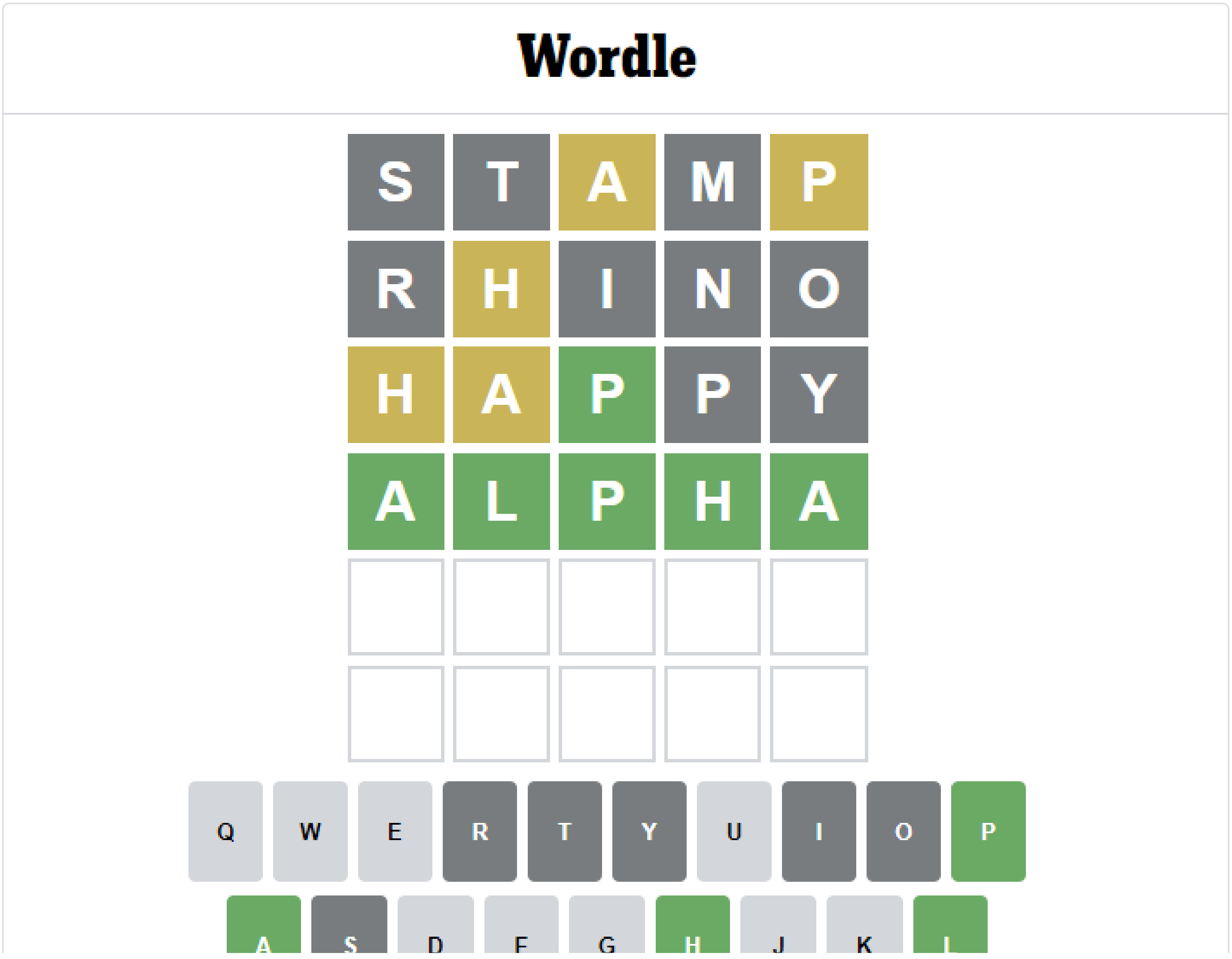
End of Honor Code

Objectives & Intended Learning Outcomes

The objective of this assignment is to familiarise you with Qt programming. After this assignment you should be able to:

1. Use Qt Designer to build the GUI of a program.
2. Use signals and slots to let Qt objects communicate with each other.
3. Implement a fully functional game in Qt.

End of Objectives & Intended Learning Outcomes

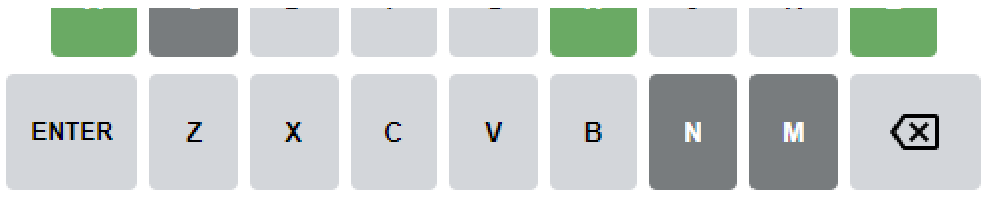


Page maintained by

DINH Anh Dung
Email: dzung@ust.hk
Last Modified: 10/22/2022 03:20:10

Homepage

Course Homepage



The popular game Wordle on The New York Times

Introduction

To get yourself familiar with creating a Qt program, let's build the popular game Wordle! Wordle is a word guessing game where you must try to guess a 5-letter word within 6 guesses. Hints are given after each guess about whether each letters are correct, appear in a different position, or do not appear at all. You can play the game on the New York Times website, if you are not already familiar with how the game works. The game only allows you to play once per day, which is why we will implement a Qt version of the game so that you can play any time you'd like, and with several other modes you can add in the bonus tasks!

In this assignment, you are required to:

- Implement the Wordle coloring scheme and hard mode
- Implement the user input keyboard (both physical and on-screen)
- Implement GUI widgets for game mode options using Qt Designer

In addition, you may also attempt these tasks for bonus points:

- Implement additional modes: Simultaneous games, Timed mode, Absurd mode
- Write a solver to give the player an "optimal" next guess
- Use word frequency to generate more common answers

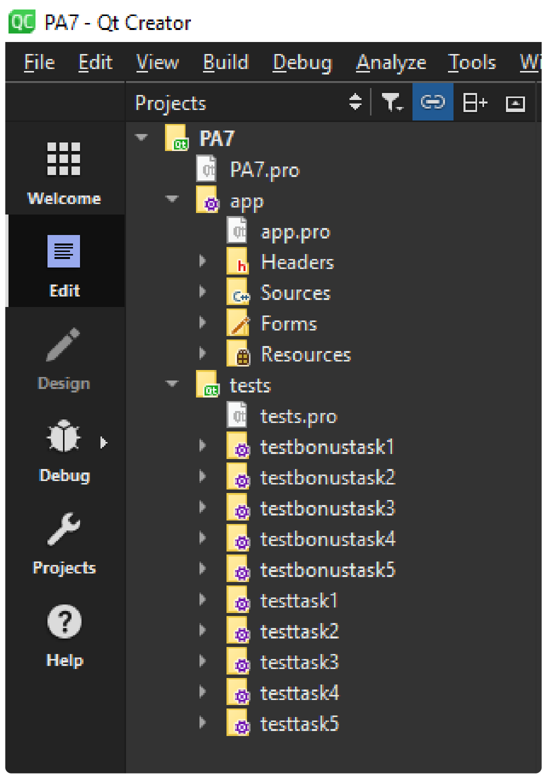
Please read the Qt Programming tutorial notes and lab example before starting this assignment to understand the basics of programming with Qt. We will use **Qt 6.3.2** for this assignment, so please install the same version to ensure your program can be tested correctly (Note that versions as low as 5.12 may still be able to run the skeleton code, but we have not tested with older versions yet).

End of Introduction

Description

Please read the FAQ section regularly, and do check it one day before the deadline to make sure you don't miss any clarification, even if you have already submitted your work by then. You can also raise questions on Piazza and remember to use the "pa7" tag.

Code structure



```
PA7
    PA7.pro
    app
        app.pro
        Headers
        Sources
        Forms
        Resources
    tests
        tests.pro
        testbonustask1
        testbonustask2
        testbonustask3
        testbonustask4
        testbonustask5
        testtask1
        testtask2
        testtask3
        testtask4
        testtask5
```

Hierarchy of the given skeleton files.

At first glance, the skeleton code can look overwhelming with the number of files. Don't worry though! This is because the project is built with testing in mind, so it is split into 2 sub-projects: **app** and **tests**. You can open the `app.pro` file in Qt Creator to edit and build only the application - it is recommended that you do this when you first start, so that you don't waste time building the test programs. The files in **tests** sub-project is only relevant in the Testing and Grading section.

There are multiple `.cpp` and `.h` files for the app. They are organized so that whenever you work on a task, you only need to handle a few source files at a time. Here is a quick breakdown of all the files.

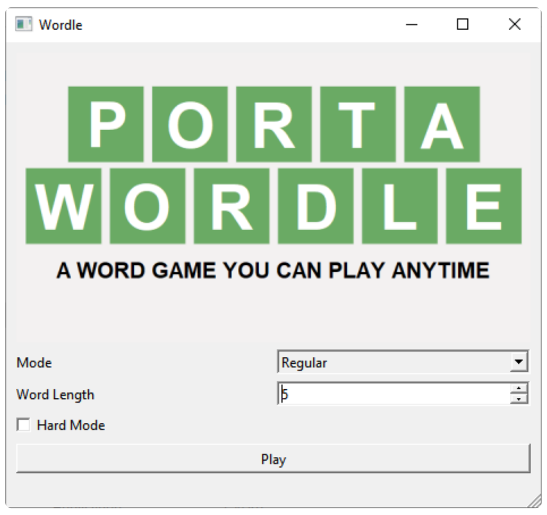
Note: You may find a few topics in this assignment not yet covered in lectures. Don't worry, you don't need to know them to complete the assignment. Any tasks that may require you to use these topics will be explicitly stated and will have thorough instructions.

- **Inheritance and polymorphism:** These are extremely useful when it comes to UI related classes, especially Qt. For this assignment, you don't have to know a lot about them, they are simply a way to implement similar objects (such as different game modes) where similar data members/functions can be reused or reimplemented accordingly. You will learn more about this topic in PA8, so feel free to come back to this assignment to understand the classes' relationships better.
- **Friend classes:** These are only relevant for test classes, you don't need to worry about them. Defining the test classes as "friends" of the main classes allow them to access said classes' private data members, which helps with testing.
- **Exception handling:** Used in `dictionary.cpp` and `mainwindow.cpp` as an easy way to exit sub-window's initialization should unexpected errors occur. You don't have to use this in the basic tasks; for some of the bonus tasks, you can refer to existing code for reference.
- **Macros:** A handy way to shorten repeating codes - any instances of macros will be replaced with the macro definition during compilation (you can think of this as similar to `Makefile` variables). If you are familiar with this, feel free to use them if necessary; otherwise, just write your code normally.
- **Containers/Templates:** Qt provides helpful containers such as `QSet` or `QList`. These are very similar to standard containers in the STL library, which will be taught in the lectures soon. These may be relevant when you try to do the bonus tasks. You can think of containers like `QSet<QString>` as "an object capable of storing strings that allows for easy insertion (`insert()`), deletion (`remove()`), and look-up (`find()`)".

main.cpp

This is simply the default main.cpp of Qt, where an application and a MainWindow is created. This file does not need to be changed.

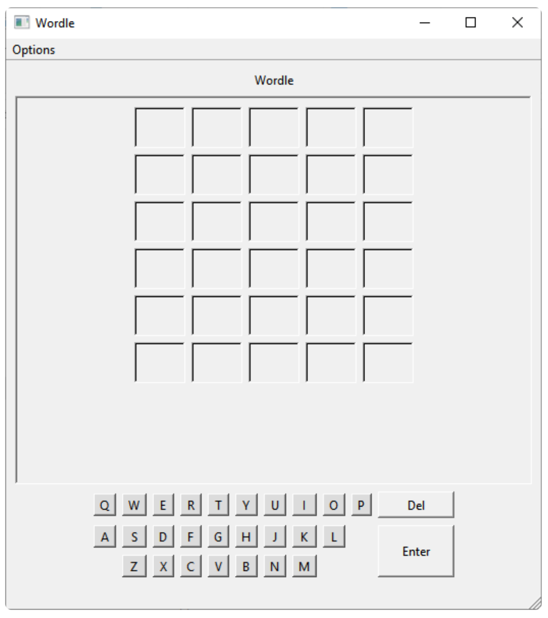
mainwindow.cpp/.h



Mode: Regular
Word Length: 5
Hard Mode

These files handle the MainWindow view, which is what the user sees when the program first opens. The UI for this window is designed in **mainwindow.ui**. Here, the user can select the game mode and options. Upon clicking "Play", this window will be hidden, and the corresponding RoundWindow will be shown.

roundwindow.cpp/.h



These files handle the RoundWindow view, which is what the user sees when starting a game. The UI for this window is designed in **roundwindow.ui**.

Note that this is a **base** class, and for the most part should not be initialized as an object directly. When user starts a game mode, MainWindow will initialize an object of one of the **derived** classes, defined in the `[mode]roundwindow.cpp/.h` files. You can consider this class as the "shared" properties of the different windows for each mode, and implement the tasks normally.

wordleround.cpp/.h

The WordleRound class handles the game logic for a game of Wordle. Receives key input and update the guess boxes accordingly. When a complete guess is entered, guess should be validated before being colored according to Wordle coloring convention. Handles game over conditions as well.

Note that this is a **base** class, and for the most part should not be initialized as an object directly. The derived RoundWindow class should initialize a corresponding WordleRound object of one of the **derived** classes, defined in the `[mode]wordleround.cpp/.h` files. You can consider this class as the "shared" properties of the different game modes, and implement the tasks normally.

regularroundwindow.cpp/.h, regularwordleround.cpp/.h

These are the derived classes of RoundWindow and WordleRound for the **Regular** mode. Since the base classes are essentially sufficient for building a regular game, not much is added within these files. There's no need to change these files.

absurdroundwindow.cpp/.h, absurdwordleround.cpp/.h

These are the derived classes of RoundWindow and WordleRound for the **Absurd** mode. You only need to modify these files for Bonus Task 1.

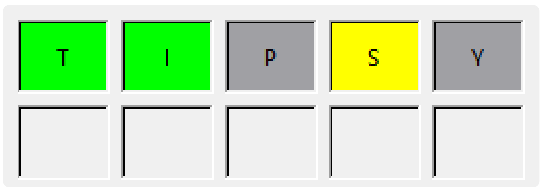
multiroundwindow.cpp/.h, multiwordleround.cpp/.h

These are the derived classes of RoundWindow and WordleRound for the **Multiple** mode. You only need to modify these files for Bonus Task 2.

timedroundwindow.cpp/.h, timedwordleround.cpp/.h

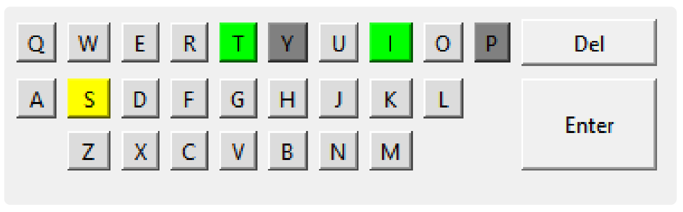
These are the derived classes of RoundWindow and WordleRound for the **Timed** mode. You only need to modify these files for Bonus Task 3.

letterbox.cpp/.h



The LetterBox class extends QLineEdit, the widget used for the guess boxes. Initialized by WordleRound, then added to the UI. Handles the backend of changing text and color, so that other classes can simply call `setLetter()` and `setColor()`.

keyboard.cpp/.h



The Keyboard class holds the 26 Keys and the 2 QPushButtons for Del and Enter. Initialized by RoundWindow. The keys are defined in the .ui file, so they need to be added to the Keyboard after initialization. Communicates with the current WordleRound when a key is pressed, and receives key color updates from WordleRound.

key.cpp/.h

The Key class extends QToolButton, the widget used for the 26 letter keys. Handles the backend of changing color, so that other classes can simply call `setColor()`.

dictionary.cpp/.h

The Dictionary class maintains a set of all English words of given length, by extracting from `words_{}.txt`. For Bonus Task 5, also maintain a list of common words with frequencies, used to generate answers. Initialized by WordleRound.

There is also a DictionaryException class defined for exception throwing. As mentioned in the note, you don't have to worry about this class.

solver.cpp/.h

The Solver class maintains a set of remaining possible answers as the WordleRound progressed, and calculates the "optimal" next guess to be hinted to the player. You only need to modify these files in Bonus Task 4.

mainwindow.ui, roundwindow.ui

These are the UI files that can be edited in Qt Creator's Designer mode.

dictionary/, images/

Additional files used by the program. They are specified in **resources.qrc** so that the program can locate them during runtime.

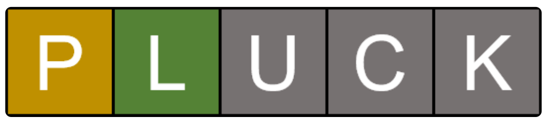
Wordle color convention

In Wordle, 3 colors are used to hint the "correctness" of a letter:

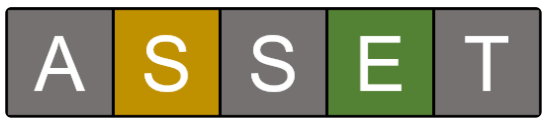
- **Green** letters are in the correct positions.
- **Yellow** letters are in the word, but are not in the correct positions.
- **Grey** letters do not appear in the word at all.

The coloring scheme can also get a bit tricky when a letter appears multiple times in the guess or the answer. To illustrate this clearly, let's look at an example.

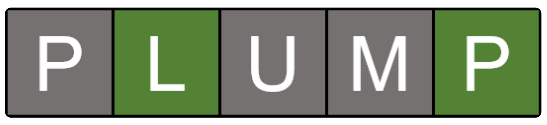
Suppose the answer to the Wordle game is "SLEEP". Here are the coloring schemes for some possible guesses:



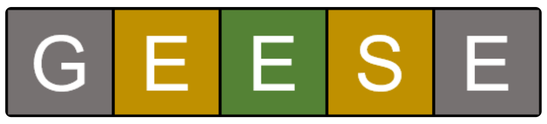
L is in the correct position, so it is **green**. P is in the word but not in the correct position, so it is **yellow**. U, C, K don't appear in the word, so they are **grey**.



There are 2 Ss in the guess, but only 1 S in the answer. The first S is **yellow**, telling you that the correct S is not in this position. The second S is **grey**, telling you that **there is no extra S**, and the correct S is also not in this position.

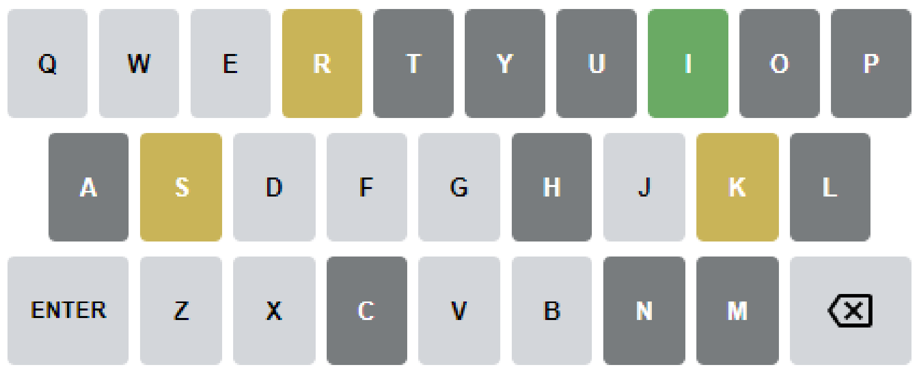


Same as above, except the second P is **green** because there is a P there. The first P is **grey** because there is no extra P.



The second E is **green** because there is an E there. The first E is **yellow** because there is a 2nd E, which is not at this position. The third E is **grey** because there is no 3rd E, and there is no E at this position.

In addition to coloring the guesses, we also need to color the keyboard. The letters on the keyboard are colored similar to the guesses to tell the player the current "correctness" status for each key.



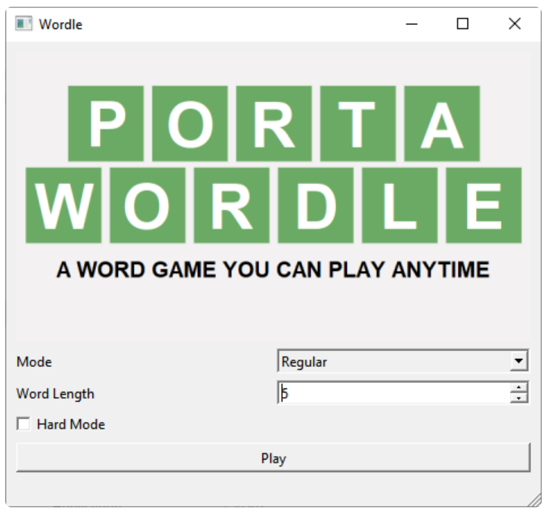
This keyboard is saying that the letter I has been found at the right position, and letters S, R, K are in the answer but at unknown positions. Grey letters are known to not be in the answer, while light grey letters have not been used in any guesses.

Whenever a guess is made, the keyboard is updated with new key colors. Only the "highest" color is kept: e.g. in the GEESE example, the key E would be colored **green**; in the ASSET example, the key S would be colored **yellow**.

We will describe in detail how to implement this coloring scheme in the Tasks section.

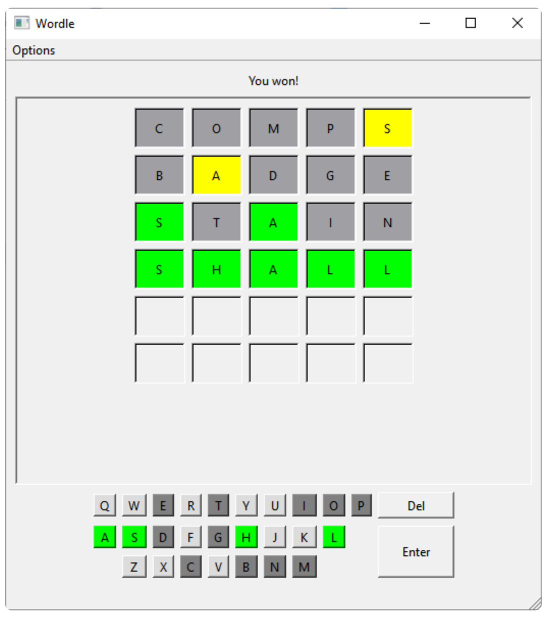
Demo program overview

This section is a brief overview of the demo program, which is essentially how your program is expected to behave if all tasks (including bonus tasks) are implemented correctly. Feel free to try out and play the game yourself.



Mode: Regular
Word Length: 5
Hard Mode

When starting the program, you will be taken to the main window. You can select the game mode and additional options for each mode, as well as word length. After clicking Play, the game window will be shown.



This is the window for Regular mode. This can be played exactly like the official Wordle version: you can either press the buttons to input letters or use the keyboard to type. Popups appear if the player types a non-English word or finishes the game.

In the Options menu, there are 4 options. "Give Up" ends the game and treat it as a loss. "Reset Round" ends the current game and starts a new game with a new word. "Cheat" reveals the answer. This is helpful if you would like to test game logic with the demo program or your program. Finally, "Hint" will try to give the player an "optimal" word to guess. This is implemented in Bonus Task 4.

For other game modes, there are a few UI changes. These are described in more detail in the bonus task section, and you don't need to worry about them if you just plan to do the basic tasks.

Additional remarks

- All of the needed header files have been included in the source files, so you don't need to add any new includes. If you do add new includes, **you are only allowed to add the given header files and Qt-provided header files**, such as `<QPushButton>`.
- **You are strictly forbidden from modifying the provided files not for submission**; in other words, all of the .h header files and a few .cpp files. During grading, we will use the version given in the skeleton code of these files, so any changes you made there will be lost during actual grading.

End of Description

Tasks

The tasks are divided into 2 sections: Basic tasks and bonus tasks. There are 5 tasks for each section, with weighting detailed below. The bonus tasks are quite challenging so don't spend too much time on them; you will get a total of 15% bonus points for completing all 5 bonus tasks.

<u>Tip:</u> All tasks have comment blocks near them detailing what needs to be done in the code. You can use the Search option in Qt Creator and look for the string `TODO` to find which part of the code you need to edit.

Basic Tasks

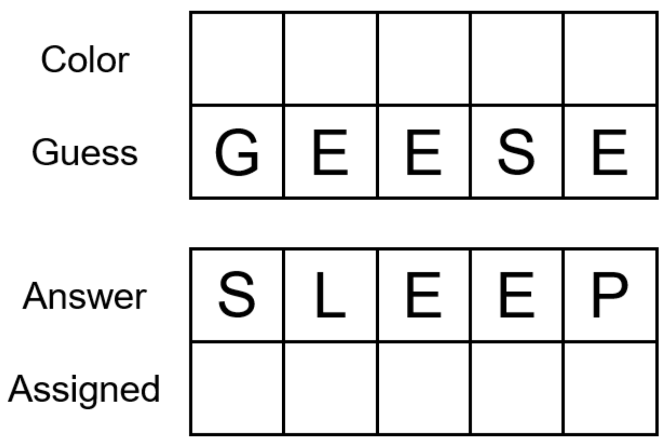
Task 1: Implement letter coloring scheme [20%]

You will need to implement the function

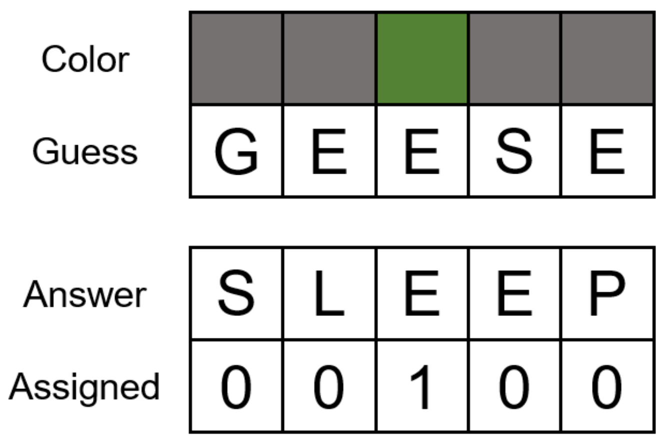
```
QColor* analyzeGuess(const QString& guess, const QString& answer)
```

in **wordleround.cpp**. This function compares the 2 strings `guess` and `answer`, and outputs an array of QColor corresponding to the color sequence of `guess`.

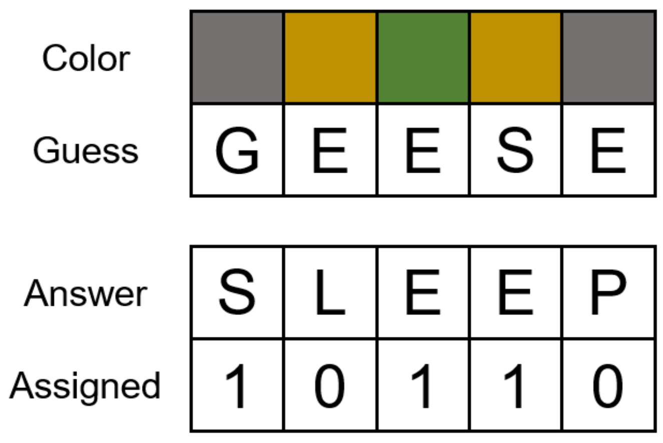
To do this, you can use a 2-pass strategy: Create 2 arrays, one to store the color sequence and the other to be the "assigned" status of `answer`'s letters.



For the first pass, compare each of `guess` and `answer`'s letters. If the letters are the same, color it **green** and set "assigned" to true. Otherwise, color it **grey** and set "assigned" to false.



For the second pass, compare each of `guess`'s remaining letters with all of `answer`'s "unassigned" letters. If a similar letter is found "unassigned", color it **yellow** and set "assigned" to true.



The rightmost E is grey because no "unassigned" E remains.

Now the color array can be returned. Since the array length is not constant, remember to manage any dynamically allocated variables and delete when unused.

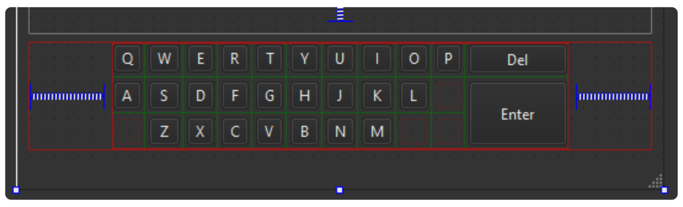
If you think you can come up with a faster algorithm to complete this task (the above algorithm is `O(n^2)`), feel free to give it a go, but make sure it is correct by double checking color sequence output for special cases.

*<u>Hint:</u> You can use the colors defined in **letterbox.h**.*

Task 2: Implement on-screen keyboard [25%]

For this task, you need to modify some functions in **roundwindow.cpp** and **keyboard.cpp**. You will also need to work with the **roundwindow.ui** to add the buttons to the UI.

First, open **roundwindow.ui**. Qt Creator should take you to the Design tab. Find the QGridLayout named `gridLayoutKey` on the UI (you may also look at the widget list on the side bar). Add 26 QToolButtons to the grid to resemble a keyboard. Set their text as *uppercase* letters.



Example grid layout of 26 keys and 2 buttons. Hint: widgets can span multiple grid cells.

Rename the widgets to `toolButton_{LETTER}` so that you can access them in **roundwindow.cpp** later. Select all keys, right click and choose `Promote to...`. Select Key as the class to promote to, or type in the class name "Key" and header file "key.h". This will convert the 26 QToolButtons to Key objects.

Also add 2 QPushButtons and set their text as "Del" and "Enter". Rename the widgets to `pushButton_Del` and `pushButton_Enter`.

Important: *While widget names can be anything as long as they can be accessed in the code, please name the widgets as above so that they can be tested. As an example, the key A should be named* `toolButton_A`*.*

Then, go to the RoundWindow constructor

```
RoundWindow::RoundWindow(QWidget *parent)
```

in **roundwindow.cpp**. As described in the Description section, the 26 keys and Del/Enter keys defined in **roundwindow.ui** need to be added to the Keyboard after it is initialized.

Tip: The most straightforward way is to write 26 lines to add the letter keys. However, this can be drastically shortened if you know how to use macro in C++.

Next, go to **keyboard.cpp** and implement the following functions:

```
void Keyboard::addKey(Key *key, const char keyChar)
void Keyboard::addDelKey(QPushButton *key)
void Keyboard::addEnterKey(QPushButton *key)
```

These functions should set the data members to the right buttons and connect the buttons' `clicked()` signals to the corresponding handling slots of Keyboard.

Finally, go to

```
void Keyboard::updateKeyColor(const QChar& key, const QColor& color)
```

and implement the keyboard coloring scheme described above. Simply put, the color of each key can only go up from **grey** to **yellow** and to **green**.

Task 3: Implement physical keyboard [15%]

The on-screen keyboard can be cumbersome to use. Therefore, we'd like to implement our existing keyboard as an alternative input method. To do this, we need to override RoundWindow's `keyPressEvent()` method, which gets called whenever the window receives physical keyboard input.

Simply put, you just need to complete the implementation of

```
void RoundWindow::keyPressEvent(QKeyEvent *event)
```

in **roundwindow.cpp**. You can get the key being pressed with `event->key()`, which will correspond to one of Qt's defined key enums. Depending on the key being pressed, call the corresponding Keyboard function.

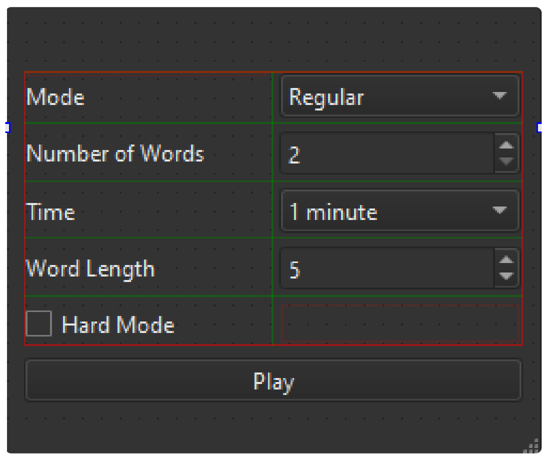
Hint: Slots can be called as a function normally.

Task 4: Implement round options [25%]

This task will involve editing the **mainwindow.ui** form, as well as **mainwindow.cpp**. You will need to add the widgets that allow players to configure settings before playing. While some of these are related to the extra modes implemented in bonus tasks, here we just need to implement the widgets so that they appear correctly (pressing Play when an extra mode is selected will not do anything yet).

First, open **mainwindow.ui**. Qt Creator should take you to the Design tab. Find the QGridLayout. You will need to add the following items to the Grid:

- QLabel `labelMode`, with text "Mode"
- QComboBox `comboBoxMode`. Double click on the widget to add the following options:
  - Regular
  - Absurd
  - Multiple
  - Timed
- QLabel `labelNumWords`, with text "Number of Words"
- QSpinBox `spinBoxNumWords`. Set its minimum and maximum values to 2 and 8. Set its default value to 2.
- QLabel `labelTime`, with text "Time"
- QComboBox `comboBoxTime`. Double click on the widget to add the following options:
  - 1 minute
  - 3 minutes
  - 5 minutes
  - 10 minutes
- QLabel `labelWordLength`, with text "Word Length"
- QSpinBox `spinBoxWordLength`. Set its minimum and maximum values to 1 and 31. Set its default value to 5.
- QCheckBox `checkBoxHardMode`, with text "Hard Mode"



Example grid layout of mode selection widgets.

Important: *While widget names can be anything as long as they can be accessed in the code, please name the widgets as above so that they can be tested.*

Next, go to **mainwindow.cpp** and complete the missing implementations. In particular:

```
MainWindow::MainWindow(QWidget *parent)
```

Connect the Mode comboBox's signals to the corresponding MainWindow slots and hide the non-default options.

```
void MainWindow::resetExtraOptions()
```

This function hides the widgets for Number of Words, Time and Hard Mode, since they are not used by all modes.

```
void MainWindow::start()
```

Change `wordLength`, `mode` and `hardMode` to read from the widgets instead of using default values.

```
void MainWindow::modeChanged(const QString &mode)
```

Implement this function to show widgets that correspond to selected mode. It should call `resetExtraOptions()` at the beginning, then show widgets based on mode.

Task 5: Implement hard mode [15%]

Wordle has a Hard Mode option where subsequent guesses must use all letters known to be in the word (in other words, **yellow** and **green** letters). For example, if your guess was "SUPER" and the letter S is **green**, letters E and R are **yellow**, you must use S, E, R in your next guesses. The game rejects invalid guesses just like non-English words.

For this task, you will add some extra functions and cases to the implementations you made in Task 2, so that if Hard Mode is enabled, extra validation is made when guessing.

Implement the

```
bool validHardModeGuess(const QString& guess, const bool correctLetters[])
```

function in **wordleround.cpp**, which checks if `guess` contains all letters deemed "correct" by the array `correctLetters`.

Hint: `correctLetters` is a boolean array of length 26. If `correctLetters[3]` is true, this means the (3+1)-th letter ("D") is "correct".

Next, modify the

```
void WordleRound::registerEnterKey()
```

function to add Hard Mode validation, if hard mode is enabled. It should also maintain the `correctLetters` array after each guess.

Bonus Tasks

Bonus Task 1: Implement Absurd mode [2%]

This task is inspired by Absurdle, an adversarial variant of the game where the answer changes so that it takes the player as many guesses as possible to find the right word. Essentially, the answer is only determined when the set of player's guesses with colors could only have come from one possible answer. However, this means the answer fully depends on user inputs and will be the same for a given set of guesses (we can call this game "deterministic"), and has no replayability value. Therefore, we will implement a slightly different version of the game, where the answer **changes by at most one letter after each guess**.

You will only have to modify **absurdwordleround.cpp**, **absurdroundwindow.cpp** and **mainwindow.cpp**. For MainWindow, you just need to initialize the AbsurdRoundWindow when the Absurd mode is selected.

In **absurdroundwindow.cpp**:

- Complete the constructor (you may look at RegularRoundWindow's constructor as example).
- For this mode, we don't color the keyboard since the answer changes, so no need to connect the signals.

In **absurdwordleround.cpp**: Implement

```
QString alterAnswer(const QString& answer, const QSet<QString>& allAnswers)
```

which finds a word from the set that differs from `answer` by exactly 1 letter. If no such word exists, the function returns `answer`.

Bonus Task 2: Implement Multiple mode (Simultaneous games) [4%]

This task is inspired by Dordle, Quordle, and many other variants where players need to guess more than one word at a time. The guess is used across all answers, so there is a coloring sequence for each remaining unguessed word.

You will only have to modify **multiwordleround.cpp**, **multiroundwindow.cpp** and **mainwindow.cpp**. For MainWindow, you just need to initialize the MultiRoundWindow when the Multiple mode is selected. The player can select how many words to guess at the same time (between 2 and 8).

In **multiroundwindow.cpp**:

- Complete the constructor (you may look at RegularRoundWindow's constructor as example).
- We need to add another 2D grid of LetterBox* for each extra words that need to be guessed. The description in the skeleton code provides more detail on how to initialize them and add them to the UI.
  - ***Important:*** *You need to add the `QGridLayouts` to the horizontal layout so that the answers appear in order from left to right; that is, `ansMulti[0]` corresponds to the leftmost column of LetterBox grid. You can look at the demo program's behaviour for reference (use Cheat option to see the answers). This is important for some test cases in grading.*

In **multiwordleround.cpp**: Implement the following functions:

```
void MultiWordleRound::registerKey(const QChar& key)
void MultiWordleRound::registerDelKey()
void MultiWordleRound::revertGuess()
void MultiWordleRound::registerEnterKey()
```

You can refer to the implementations in **wordleround.cpp**; these should mostly be the same except that you need to handle a 3D array of LetterBox*. Since Hard Mode and Solver are not applicable in this mode, you don't need to implement them.

Note: Some answers may end up the same, especially if there are not many available words. This does not happen in Dordle, Quordle or other online variations, but for our game we allow it to happen. We also just update the keyboard color based on values from all "sub-games", so it's possible to see more than 5 green or yellow keys (in actual variations, the keys are colored in a grid).

Bonus Task 3: Implement Timed mode [4%]

In this mode, the player will try to guess as many words as possible within a given duration. For this task, you will need to make a few adjustments to the UI:

- Add widgets to display time, number of guessed words, button to skip word
- Add a widget to display invalid word, correct guess, etc. instead of popups, since we don't want to be interrupted while playing under time constraints.

While adding widgets can be done with Qt Designer, it would require creating another .ui form specifically for the Timed mode. Here we use inheritance so that we can reuse **roundwindow.ui**, so the extra widgets need to be added via code.

You will only have to modify **timedwordleround.cpp**, **timedroundwindow.cpp** and **mainwindow.cpp**. For MainWindow, you just need to initialize the TimedRoundWindow when the Timed mode is selected.

In **timedroundwindow.cpp**:

- Complete the constructor (you may look at RegularRoundWindow's constructor as example).
- We need to initialize several other widgets exclusive to this mode, look at the description in the skeleton code for more detailed instructions.
- Implement the `void TimedRoundWindow::updateTimer()` function. This is called every second in order to update the timer shown on the UI.

In **timedwordleround.cpp**: The `invalidWord`, `invalidHardWord`, `roundWin`, `roundLose` signals are connected to RoundWindow's popup slots. We don't want to have popups during the game, so use `disconnect` to remove them. Then connect the signals, along with `scoreUpdated`, to the corresponding slots in TimedRoundWindow.

Also implement the following functions:

```
void TimedWordleRound::endRound(const bool win)
void TimedWordleRound::resetTimedRound()
void TimedWordleRound::skipCurrentWord()
```

Bonus Task 4: Implement Solver [3%]

This task is inspired by this video from YouTuber 3Blue1Brown, who tried to find the optimal Wordle starting guess using information theory. For each word, we calculate the average amount of "information" we expect to learn if we used this word as a guess (also called entropy). The formula for entropy:

$$\mathrm{H}(X) := -\sum_{x\in\mathcal{X}} p(x)\log p(x) = \mathbb{E}[-\log p(X)],$$

where `X` is a random variable representing the color sequence that could appear from guessing this word. `x` is one possible outcome of `X` (for example, GREY-YELLOW-GREEN-GREY-GREY), and `p(x)` is the probability that `x` happens, given that the chosen word is used as a guess and any of the remaining possible answers could be the actual answer. Therefore, you need to calculate the color sequence for each possible answer with the given guess, and find the probability of each color sequence happening (assume uniform probability for possible answers). We use log base 2 for reasons described in the video, but any logarithm base would still work.

Finally, to get the "optimal" next guess, we select the word from the set of all possible guesses with the highest entropy - we expect to gain the most amount of information, and hence reduce the size of remaining answers, with this word.

Note: If there is only one possible answer left, all words will have entropy of 0 because the size of possible answers cannot be reduced further. In this case, solver should hint the only remaining possible answer.

Note: We ignore Hard Mode in this bonus task because the solver can hint words not using correct letters. If we were to implement Hard Mode, we just need to check if Hard Mode is enabled, and use the set of remaining possible answers as the possible guesses in entropy calculation. But you don't need to implement that for this assignment.

For this task, you only need to modify **solver.cpp** and **wordleround.cpp**. In WordleRound, you just need to update `void WordleRound::registerEnterKey()` to call the function to update solver's set of remaining possible answers after each guess.

In **solver.cpp**, implement the following functions:

```
double computeEntropy(const QString& word, const QSet<QString>& possibleAnswers)
```

This function calculates the entropy of a given word with the remaining possible answers using the formula calculated above. Follow the detailed description in the skeleton to implement this function.

```
QString Solver::hint()
```

This function returns the "optimal" next guess - the word with the highest entropy. If there is only one possible answer left, the function returns that answer.

```
void Solver::updateAnswers(const QString& guess, QColor* const colors)
```

This function updates the set of possible answers based on the result of newest guess. Answers that do not result in the given color sequence is discarded from the set.

*Hint: Header file `<cmath>` has been included in **solver.cpp**.*

*Note: Near the bottom of **wordleround.cpp**, you will see the `QString WordleRound::getHint()` function. This determines the condition for giving hint, since the computation time and space can become enormous if there are too many letters or too many answers to loop from (remember that we need a bin length of `3^word_length`). The current settings is to only give hint for words at most 10 letters, and the number of remaining answers times word length does not exceed 5000. You may modify these numbers to do your own testing (my computer was able to handle 18 letters), but don't forget to change back for submission.*

Bonus Task 5: Implement Word Frequency [2%]

If you tried playing the game for a while, you'll notice that a lot of games have very obscure words as answers. This is because the word list (**words_{}.txt**) contains all English words, many of which are very rarely used. In this task, we will incorporate a separate list of most common English words (**answers_{}.csv**) so that the answer word can be picked and will more likely be a common word that's guessable.

For this task, you only need to modify **dictionary.cpp**:

```
Dictionary::Dictionary(int word_length)
```

In the constructor, add a section to read from **answers_{}.csv** and add them into the answer list. You can make use of the WordFrequency struct.

```
QString Dictionary::getRandomAnswer() const
```

Modify this function to instead return a random word from the answer list. Word counts should determine how likely the word would be picked as the answer. If answer somehow cannot be picked, you can fall back to the default behaviour of picking from the word set instead.

```
QSet<QString> Dictionary::getAllAnswers()
```

Modify this function to instead return the words from answer list instead of word set.

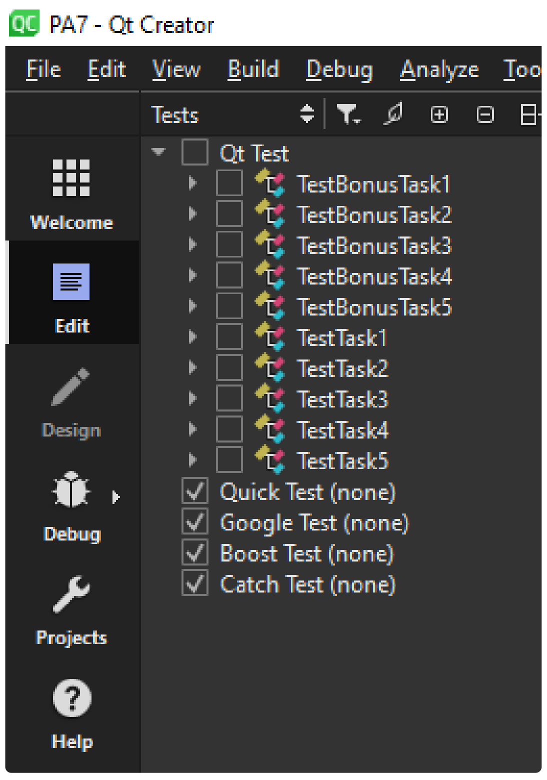
*Note: The words in **answers_{}.csv** are guaranteed to appear in **words_{}.txt**.*

End of Tasks

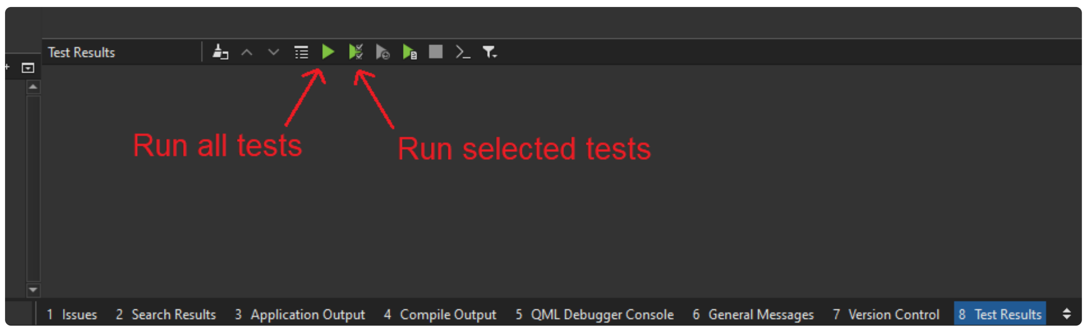
Testing and Grading

Due to ZINC system not being equipped to grade Qt projects, this assignment will be graded manually. This is why we have provided you with a set of test cases in **tests** sub-project, so that you can do testing on your own after you have finished the tasks.

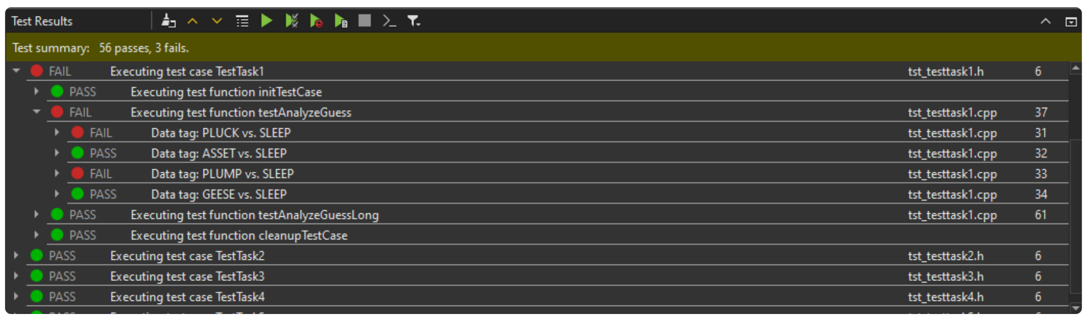
Open the `PA7.pro` project. Click on "Projects" on the top left corner and navigate to the "Tests" section. You should see something like this:



First, build the project. This will compile both the application and the test programs (one for each test class).



Then, open the Test Result panel. You can select either "Run All Tests" to run all 10 test classes, or "Run Selected Tests" to run only the tests checked on the "Tests" section.



```
Test Results
Test summary:  56 passes, 3 fails.
FAIL   Executing test case TestTask1                          tst_testtask1.h     6
  PASS   Executing test function initTestCase
  FAIL   Executing test function testAnalyzeGuess             tst_testtask1.cpp   37
    FAIL   Data tag: PLUCK vs. SLEEP                          tst_testtask1.cpp   31
    PASS   Data tag: ASSET vs. SLEEP                          tst_testtask1.cpp   32
    FAIL   Data tag: PLUMP vs. SLEEP                          tst_testtask1.cpp   33
    PASS   Data tag: GEESE vs. SLEEP                          tst_testtask1.cpp   34
  PASS   Executing test function testAnalyzeGuessLong         tst_testtask1.cpp   61
  PASS   Executing test function cleanupTestCase
PASS   Executing test case TestTask2                          tst_testtask2.h     6
PASS   Executing test case TestTask3                          tst_testtask3.h     6
PASS   Executing test case TestTask4                          tst_testtask4.h     6
```

Example of test report.

After the tests complete, you can see the results of which test cases passed or failed. You can click on them to see in detail which specific test failed.

Important: Many of the test cases require the unimplemented UI widgets in tasks 2 and 4, or the physical keyboard in task 3. You need to make sure **the widgets are named according to specification**, otherwise the tests cannot find those widgets. The tests have been designed to compile even if you have not implemented the widgets or named them incorrectly, but would result in failed tests.

Note: During testing, some windows may pop up and close rapidly. This is normal as the test cases are running GUI tests. If a pop up window appears and do not close, close it manually (this will most likely correlate to a failed test case where pop up window is not expected).

After submission deadline, your project will be graded **using a different test sub-project**, with many more test cases. Your assignment will be graded based on how many test cases passed, scaled according to task scoring scheme described above.

End of Testing and Grading

Resources & Sample Program

Last updated: 03:20 22/10/2022

- Skeleton code:
  - Windows/Linux: PA7.zip

    - Mac: PA7-mac.zip
- Demo program:
    - Windows executable: app.exe
    - Windows executable with dependencies: PA7-bundle.zip
    - Linux: app-linux
    - Mac (Intel): app-mac-intel
    - Mac (M1): app-mac-m1.zip

The Windows Qt executable cannot run by itself; it requires several Qt dynamic-link libraries. To run the provided executable, copy the folder `bin` inside the Qt distribution MinGW folder (example: `Qt/6.3.2/mingw_64/bin`) to a different place, then put the `app.exe` inside. Also copy the `platform` folder from plugins (example: `Qt/6.3.2/mingw_64/plugins/platform`) inside the new folder. Now you can execute the `app.exe` program. Alternatively, download the .zip with dependencies and run the `app.exe` inside.

End of Download

Submission

Deadline: 23:59:00 on 29 October, 2022.

ZINC Submission

Submit the following files in .zip format to ZINC:

```
absurdroundwindow.cpp
absurdwordleround.cpp
dictionary.cpp
keyboard.cpp
mainwindow.cpp
multiroundwindow.cpp
multiwordleround.cpp
roundwindow.cpp
solver.cpp
timedroundwindow.cpp
timedwordleround.cpp
wordleround.cpp

mainwindow.ui
roundwindow.ui
```

In other words, submit **all .ui files and .cpp files, except key.cpp, letterbox.cpp, main.cpp, regularroundwindow.cpp, regularwordleround.cpp**. ZINC will only be used to check if you have submitted the correct files, and whether you accidentally submitted the skeleton files. Even if you have not completed a task related to some files, please still submit them (you can use the skeleton files, which will only trigger ZINC warnings). Grading will be done manually using a different test set after the deadline.

End of Submission

Frequently Asked Questions

Please see the Qt FAQ page for common questions related to Qt.

Q: My code doesn't work, there is an error/bug, here is the code, can you help me fix it?
A: As the assignment is a major course assessment, to be fair, you are supposed to work on it on your own and we should not finish the tasks for you. We are happy to help with explanations and advice, but we shall not directly debug the code for you.

Q: Am I allowed to use local function declarations (function declaration inside an existing function) for my helper functions?
A: You are strongly discouraged from doing so, as that "feature" is a leftover merely for backwards compatibility with C. In C++, it is superseded with class functions and lambda functions, which will be taught later in this course. However, you are free to define your own helper functions in the `.cpp` files that you submit.

Q: For tasks 2 and 4, do I need to position the widgets exactly according to the demo program/given illustrations?
A: No need, the test cases are designed so that as long as the widgets have the correct name, they can be accessed by the test programs. You can position the widgets and give them some expansion policy however you want. But it is important that **the widgets are named according to the specifications**.

Q: How can I access the widgets added via the .ui forms in the Window's code?
A: Widgets added via the .ui form will become members of the data member `ui` of the Window class. You can access the widget by using `ui->[widget_name]`.

Q: How do I print messages to the terminal? I would like to use it for debugging.
A: You can use `qDebug()` the same way you would use `cout`. Note that `endl` is not needed. You will be able to see the debug messages in the Application Output terminal.

End of Frequently Asked Questions

Change Log

17:20 - 14/10/22: Added source code and demo program for MacOS, containing a few differences:

- **dictionary.h**: Changed the signature of DictionaryException `what()` so that the code can compile.
- **resource.qrc**: Changed encoding to UTF-8.

If the Mac executable does not work, please try using the Windows executable/dependencies bundle on the virtual barn machines.

14:45 - 19/10/22: Added a FAQ question about widget accessing in code.

03:15 - 22/10/22: Updated the following source files:

- multiwordleround.h
- multiroundwindow.h
- timedroundwindow.h
- roundwindow.h
- wordleround.h

The only changes are making the destructors of the classes `virtual`. It should not affect the tasks or test cases - you should still be able to complete the assignment with the old files, but it is recommended that you use the new files in case of rare bugs.

Additionally, added a FAQ question about console debugging.

End of Change Log