```rust
/*
 * create-pt - a possible way to complete the AP Computer Science Principles
 * Create Performance Task.
 *
 * Created March 25, 2024.
 * Modified April 2, 2024.
 *
 * README!!
 * FOR AP READER:
 *
 * A couple of notes, mainly due to the complexity of the Rust
 * programming language (which this is written in):
 *
 * - Feel free to Google any Rust concepts you may not understand
 *   (for example, macros like `println!` and `include_str!` or structs
 *    and their `impl`s, especially since AP CSP is intended for JavaScript
 *    or Python, which  are high-level languages, whilst Rust is a
 *    low-level language, like C or C++).
 *   (Minor sub-note: I would ESPECIALLY Google the `include_str!` macro and
 *    what it does.)
 *
 * - Any comment (green in VS Code, which I recommend you copy/paste into
 *   for ease of reading) with THREE slashes (`///`) instead of TWO (`//`)
 *   can be read as documentation for the code they accompany. If you are
 *   using Visual Studio or Visual Studio Code to read, you can hover over
 *   the name of the function/struct/impl/variable they accompany and see
 *   the documentation cleanly formatted.
 *
 * - If you ever plan on executing this code, you will need to install
 *   the Rust tools for your platform (cargo, rustup, etc.) from
 *   https://rust-lang.org. (If not, disregard this bullet.) Once you
 *   install them, create a new project with `cargo new create-pt` on
 *   your command line, traverse into the `./create-pt` directory, then
 *   run `cargo add colored rand`, copy and paste the code in this file
 *   to `./src/main.rs`, copy and paste `answers.txt` and `words.txt`
 *   into `./src`, and run `cargo run --release` on your command line.
 *   If the build fails, it is likely a system or library issue.
 */

/// The colored crate/library: Has functions to format text/strings with
/// color and text formatting like bolding and italicizing; made by a Rust
/// community member (see https://crates.io/crates/colorize).
/// The Colorize trait, which
/// allows for said formatting, is imported from this crate. (The way Rust
/// trait implementing is like abstraction but the trait must be imported in
/// the current context for them to work)
use colored::Colorize;
/// The rand crate/library: Has functions to generate random numbers; made
/// by a Rust community member (see https://crates.io/crates/rand).
/// The thread_rng function and Rng trait are imported from this crate.
```

```rust
use rand::{thread_rng, Rng};
/// The Rust Standard Crate, providing methods to read from IO
/// (stdin and stdout) and other things
use std::{
    fmt::Display,
    io::{stdin, stdout, Result as IoResult, Write},
};

/// words.txt: holds all possible Wordle guesses, made by a GitHub user
/// (https://gist.github.com/dracos/dd0668f281e685bad51479e5acaadb93)
const WORDS_FILE: &'static str = include_str!("words.txt");
/// answers.txt: holds all possible Wordle answers, made by a GitHub user
/// (https://gist.github.com/cfreshman/a03ef2cba789d8cf00c08f767e0fad7b)
const ANSWER_FILE: &'static str = include_str!("answers.txt");

/// Util for printing since the macro doesn't flush to stdout
///
/// s: anything that the [`print!`] macro accepts using the below code:
///
/// ```no_run
/// print!("{}", s)
/// ```
///
///
/// For example, the equivalent of using [`print!`] like this:
/// ```no_run
/// print!("{}{}", "Hello ", "world")
/// ```
///
/// Can be translated to use the [`print`] function like:
/// ```no_run
/// print(format!("{}{}", "Hello ", "world"))
/// ```
fn print(s: impl Display) {
    print!("{}", s);
    stdout().flush().unwrap();
}

/// Util to read exactly 1 line from stdin (user input)
fn read_line_stdin() -> IoResult<String> {
    let mut buf = String::new();
    stdin().read_line(&mut buf)?;

    Ok(buf.trim().to_string())
}

/// Game object that just stores everything cleanly
#[allow(dead_code)]
struct Game {
    correct_word: String,
    guesses: Vec<String>,
    possible_words: Vec<String>,
    has_won: bool,
```

```rust
}

/// Functions for the game object
impl Game {
    /// Instantiate a new Game object
    pub fn new() -> Self {
        // Grab all the words/answers
        let mut words = WORDS_FILE
            .lines()
            .map(|v| v.to_lowercase())
            .collect::<Vec<String>>();
        let answers = ANSWER_FILE
            .lines()
            .map(|v| v.to_lowercase())
            .collect::<Vec<String>>();
        // Append the answers to the words since they aren't together for
        // some reason
        words.append(&mut answers.clone());
        Self {
            correct_word: answers
                .get(thread_rng().gen_range(0..answers.len()))
                .expect("well thats awkward")
                .clone(),
            guesses: vec![],
            possible_words: words,
            has_won: false,
        }
    }

    // NOTE FOR AP READER:
    // When `self`, `&self`, `mut self`, or `&mut self`, it means that
    // the function is meant to be called on an instance of an object,
    // and the block of the function will have access to an object of
    // `Self`, in this case, the `Game` struct. A capitalized `Self`
    // means that it is referring to the type of `self`. `mut` being
    // in front means it can be modified, and `&` means  it is a
    // reference (it doesn't consume or "destroy" the object and make
    // it unusable in the following code after the function call).

    /// Get the correct word (equal to [`self.correct_word`])
    pub fn get_correct_word(&self) -> String {
        // The string must be cloned (make a copy of)
        // so that memory/race conditions do not occur;
        // one of the many features of Rust (preventing
        // race conditions). The `String` type is not a
        // primitive and it's size is not known at compile
        // time, unlike `&str`s, which are string literals
        // typed as `"your string"` and are primitives,
        // whose sizes are always known; however they are
        // interchangable and can easily be converted to
        // each other
        self.correct_word.clone()
    }
```

```rust
    /// Get the amount of guesses taken (equal to [`self.guesses.len()`])
    pub fn get_guess_count(&self) -> usize {
        // This doesn't need to be cloned due to being a primitive (usize is a
        // positive integer that can index arrays or vectors, which are
        // unsized array)
        self.guesses.len()
    }

    /// Submit guess
    /// Takes the user's guess and sanity-checks it before validation
    /// Returns a boolean defining if the user has won based on their guess
    pub fn submit_guess(&mut self, guess: String) -> bool {
        if guess.len() != 5 {
            println!("{}", "your guess must be 5 letters long".red().italic())
        } else if self
            .possible_words
            .iter()
            .find(|v| **v == guess.to_lowercase())
            .is_none()
        {
            println!("{}", "invalid word, try again".red().italic())
        } else {
            self.guesses.push(guess.to_lowercase());

            if guess.to_lowercase() == self.correct_word.to_lowercase() {
                // yay winner!
                println!("{}", guess.to_lowercase().green().bold());
                self.has_won = true;
                return true;
            }
            let guess_chars = guess.to_lowercase().chars().collect::<Vec<char>
            let correct_chars = self.correct_word.chars().collect::<Vec<char>>

            let mut final_str: String = String::new();

            let mut chars_found: Vec<char> = vec![];

            // Iteration!!! (Iterate through each character, 0-4 in indices,
            // 1-5 in normal terms and validate them)
            for i in 0..=4 {
                let guess_char = guess_chars[i];
                let correct_char = correct_chars[i];

                if guess_char == correct_char {
                    chars_found.push(guess_char.clone());
                    final_str = format!(
                        "{}{}",
                        final_str,
                        guess_char.to_string().green().bold()
                    );
                } else if correct_chars.iter().find(|v| {
                        v == &&guess_char
```

```
                    }).is_some() && correct_chars
                        .iter()
                        .fold(0, |acc, v|
                            if v == &guess_char {
                                acc + 1
                            } else {
                                acc
                            }
                        ) > chars_found
                            .iter()
                            .fold(0, |acc, v|
                                if v == &guess_char {
                                    acc + 1
                                } else {
                                    acc
                                })
                {
                    chars_found.push(guess_char.clone());
                    final_str = format!(
                        "{}{}",
                        final_str,
                        guess_char.to_string().yellow().bold()
                    );
                } else {
                    final_str = format!(
                        "{}{}",
                        final_str,
                        guess_char.to_string().black()
                    );
                }
            }

            println!("{}", final_str);
        }
        false
    }
}

/// Main function that executes at runtime
fn main() {
    let mut wordle = Game::new();

    println!("{}", "welcome to wordle but its in rust".bold());
    while wordle.get_guess_count() < 6 {
        print(format!(
            "{} ({}): ",
            "type your guess".italic(),
            wordle.get_guess_count() + 1
        ));
        if let Ok(input) = read_line_stdin() {
            if wordle.submit_guess(input) {
                // Submits guess, if that function returns true the user
                // has won
```

```
                break;
            }
        } else {
            println!(
                "{}",
                "type something to guess before pressing enter"
                    .red()
                    .italic()
            )
        }
    }

    if wordle.has_won {
        println!(
            "{}",
            match wordle.get_guess_count() {
                1 => "Genius".green(),
                2 => "Magnificent".green(),
                3 => "Impressive".green(),
                4 => "Splendid".green(),
                5 => "Great".green(),
                6 => "Phew".yellow(),
                // anything that isn't in the range [1, 6] (interval)
                // isn't possible
                _ => "hacker 😣".red(),
            }
        );
    } else {
        println!("You {} :(", "lost".red().bold());
        println!(
            "{}: {}",
            "The word was".italic(),
            wordle.get_correct_word().bold()
        )
    }
}
```