```
use std::fs::File;
use std::io::{BufRead, BufReader, Read};
use clap::{Arg, Command};
use lopdf::{dictionary, Document, Object, StringFormat};
use printpdf::*;
use rusttype::{Font, Scale};
use textwrap::{fill, Options};
use walkdir::WalkDir;
struct PdfWriter<'a> {
    doc: Option<PdfDocumentReference>,
    font: IndirectFontRef,
    font_path: &'a str,
    code_name: &'a str,
    code_version: &'a str,
    lines_per_page: usize,
    page_number: usize,
    line_index: usize,
    current_layer: PdfLayerReference,
    page_dimensions: (Mm, Mm),
    header_left_indent: Mm,
}
impl<'a> PdfWriter<'a> {
    fn new(
        font_path: &'a str,
        code_name: &'a str,
        code_version: &'a str,
        lines_per_page: usize,
        page_dimensions: (Mm, Mm),
    ) -> Self {
        let (doc, page1, layer1) = PdfDocument::new(
            "Code Document",
            page_dimensions.0,
            page_dimensions.1,
            "Layer 1",
        );
        let font_file = File::open(font_path).expect("Failed to open font file");
        let font = doc
            .add_external_font(font_file)
            .expect("Failed to add font");
        let current_layer = doc.get_page(page1).get_layer(layer1);
        let mut writer = Self {
            doc: Some(doc), // Wrap doc in Option
            font,
            font_path,
            code_name,
            code_version,
            lines_per_page,
            page_number: 1,
            line_index: 1,
            current_layer,
```

```
            page_dimensions,
            header_left_indent: Mm(-1.0),
        };
        writer.write_header();
        writer
    }
    fn write_header(&mut self) {
        // Center the header text
        let header = format!("{} {}", self.code_name, self.code_version);
        if self.header_left_indent < Mm(0.0) {
            let header_width = self.calculate_text_width(&*header, 10.0);
            self.header_left_indent = (self.page_dimensions.0 - Mm(header_width)) / 2.0;
        }
        self.current_layer.use_text(
            &header,
            10.0,
            self.header_left_indent,
            Mm(278.5),
            &self.font,
        );
        let line = Line {
            points: vec![
                (Point::new(Mm(20.0), Mm(277.0)), false),
                (Point::new(Mm(184.0), Mm(277.0)), false),
            ],
            is_closed: false,
        };
        self.current_layer.set_outline_thickness(1.2);
        self.current_layer
            .set_outline_color(Color::Rgb(Rgb::new(0.0, 0.0, 0.0, None)));
        self.current_layer.add_line(line);
        self.current_layer.use_text(
            format!("{}", self.page_number),
            11.0,
            Mm(189.0),
            Mm(276.5),
            &self.font,
        );
    }
    fn add_line(&mut self, line: &str) {
        if self.line_index > self.lines_per_page {
            self.new_page();
        }
        let wrapped_line = fill(line, Options::new(90).subsequent_indent("    "));
        let mut y = 272.0 - 5.4 * (self.line_index - 1) as f64;
        for wrapped_line in wrapped_line.lines() {
            if self.line_index > self.lines_per_page {
                self.new_page();
                y = 272.0;
            }
```

```
            self.current_layer.use_text(
                format!("{:>4}    {}", self.line_index, wrapped_line),
                11.0,
                Mm(6.0),
                Mm(y as f32),
                &self.font,
            );
            y -= 5.4;
            self.line_index += 1;
        }
    }
    fn new_page(&mut self) {
        self.page_number += 1;
        self.line_index = 1;
        let (page, layer) = self.doc.as_ref().unwrap().add_page(
            self.page_dimensions.0,
            self.page_dimensions.1,
            "Layer 1",
        );
        self.current_layer = self.doc.as_ref().unwrap().get_page(page).get_layer(layer);
        self.write_header();
    }
    fn save(mut self, output_pdf_path: &str) {
        let doc = self.doc.take().unwrap(); // Take ownership of doc
        let mut pdf_document = doc.save_to_bytes().expect("Failed to save to bytes");
        let mut lopdf_doc =
            Document::load_mem(&mut pdf_document).expect("Failed to load PDF document");
        // Convert the title string to UTF-16BE and add a BOM (Byte Order Mark)
        let title_str = format!("{} {}", self.code_name, self.code_version);
        let mut title_utf16be = vec![0xFE, 0xFF];
        title_utf16be.extend(
            title_str
                .encode_utf16()
                .flat_map(|u| vec![(u >> 8) as u8, u as u8]),
        );
        // Set the document info dictionary
        let info_dict = dictionary! {
            "Title" => Object::String(title_utf16be, StringFormat::Literal),
            "Creator" => Object::String(b"PrintCode".to_vec(), StringFormat::Literal),
            "Producer" => Object::String(b"https://github.com/yliu7949/
    PrintCode".to_vec(), StringFormat::Literal),
        };
        // Set document information properties
        let info = lopdf_doc.add_object(info_dict);
        lopdf_doc.trailer.set("Info", info);
        // Save the final PDF file
        lopdf_doc
            .save(output_pdf_path)
            .expect("Failed to save PDF document");
    }
```

```
    fn calculate_text_width(&mut self, text: &str, font_size: f32) -> f32 {
        // https://github.com/fschutt/printpdf/issues/49#issuecomment-1110856946
        let font_file = File::open(&self.font_path).expect("Failed to open font file");
        let mut font_cache = BufReader::new(font_file);
        let mut buffer = Vec::new();
        font_cache
            .read_to_end(&mut buffer)
            .expect("Error reading font file");
        let font = Font::try_from_bytes(&buffer).expect("Error loading font");
        let scale = Scale::uniform(font_size);
        let str_width: f32 = font
            .glyphs_for(text.chars())
            .map(|g| g.scaled(scale).h_metrics().advance_width)
            .sum();
        str_width * 25.4 / 72.0
    }
}
fn main() {
    let matches = Command::new("printcode")
        .version("0.1.0")
        .author("yliu7949")
        .about("Generates a PDF from code files with pagination and custom headers.")
        .arg(
            Arg::new("font-dir")
                .short('f')
                .long("font-dir")
                .value_name("FONT_DIR")
                .help("Directory where the font files are located")
                .default_value("C:/Windows/Fonts")
                .num_args(1),
        )
        .arg(
            Arg::new("font-name")
                .short('t')
                .long("font-name")
                .value_name("FONT_NAME")
                .help("Name of the font file to use")
                .default_value("simsun.ttc")
                .num_args(1),
        )
        .arg(
            Arg::new("code-folder")
                .short('d')
                .long("code-folder")
                .value_name("CODE_FOLDER")
                .help("Directory containing code files")
                .required(true)
                .num_args(1),
        )
        .arg(
```

```
            Arg::new("verbose")
                .long("verbose")
                .help("Print detailed information")
                .action(clap::ArgAction::SetTrue),
        )
        .arg(
            Arg::new("code-name")
                .short('n')
                .long("code-name")
                .value_name("CODE_NAME")
                .help("Code name for the PDF document")
                .required(true)
                .num_args(1),
        )
        .arg(
            Arg::new("code-version")
                .short('v')
                .long("code-version")
                .value_name("CODE_VERSION")
                .help("Code version for the PDF document")
                .default_value("V1.0.0")
                .num_args(1),
        )
        .arg(
            Arg::new("output-path")
                .short('o')
                .long("output-path")
                .value_name("OUTPUT_FILE")
                .help("Path to the output PDF document")
                .default_value("output.pdf")
                .num_args(1),
        )
        .get_matches();
    let verbose = matches.get_flag("verbose");
    let font_dir = matches.get_one::<String>("font-dir").unwrap();
    let font_name = matches.get_one::<String>("font-name").unwrap();
    let code_folder = matches.get_one::<String>("code-folder").unwrap();
    let code_name = matches.get_one::<String>("code-name").unwrap();
    let code_version = matches.get_one::<String>("code-version").unwrap();
    let output_pdf_path = matches.get_one::<String>("output-path").unwrap();
    let font_path = format!("{}/{}", font_dir, font_name);
    let mut pdf_writer = PdfWriter::new(
        &font_path,
        code_name,
        code_version,
        50,                      // lines_per_page
        (Mm(210.0), Mm(297.0)), // A4 page dimensions
    );
    for entry in WalkDir::new(code_folder).into_iter().filter_map(|e| e.ok()) {
        if entry.path().is_file() {
```

```
            let file = File::open(entry.path()).expect("Failed to open code file");
            let reader = BufReader::new(file);
            for line in reader.lines() {
                let line = line.expect("Failed to read line");
                if !line.trim().is_empty() {
                    pdf_writer.add_line(&line);
                }
            }
            pdf_writer.add_line("\n");
        }
    }
    pdf_writer.save(output_pdf_path);
    if verbose {
        println!("PDF document generated successfully.");
    }
}

```