Rust Programming: Creating a Phoronix Reader Application

Michael Aaron Murphy

Published with GitBook

Table of Contents

Introduction

Rust Programming: Creating a Phoronix Reader Application

This guide will present step-by-step instructions regarding how to write a **Phoronix Reader** command-line interface, or **CLI**, application in Rust with colored output, but will evolve to also include a **GTK3** graphical user interface, or **GUI**. The goal of this project is to teach prospective programmers the skills they need to start writing complete software in Rust beyond the run-of-the-mill CLI calculator tutorials.

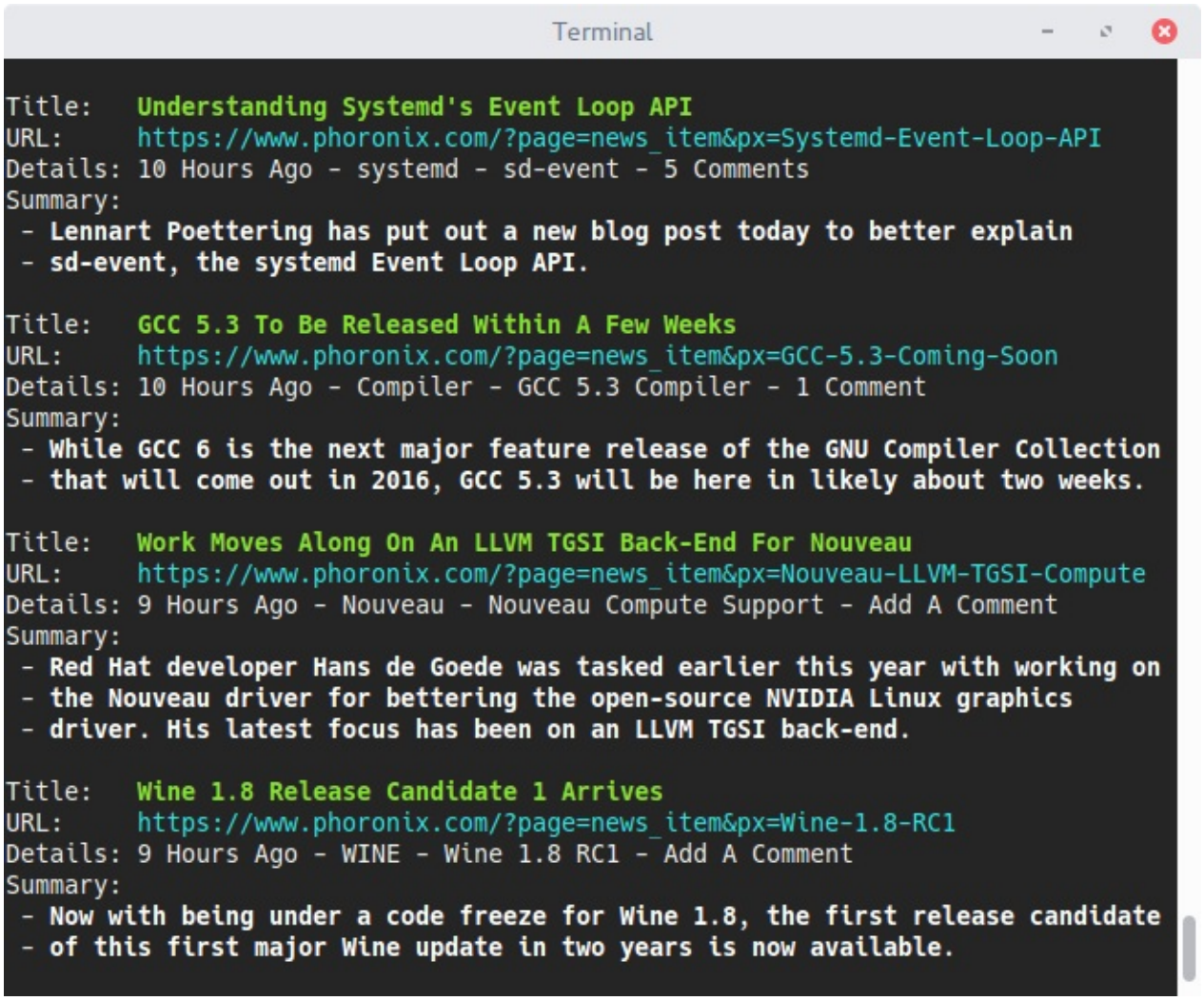
Before you can run by creating exquisite **GUI** applications, you must first learn to walk in creating **CLI** applications. **GTK3** is a **GUI toolkit** used primarily on Linux for writing GUI applications. It stands for the **GIMP ToolKit**, even though **GIMP** isn't the focus of **GTK** anymore.

You will perform an **HTTP request** to download the homepage of **Phoronix** with **`hyper`**, parse the information with **`select`** and then print it in full color to the terminal using the **`term`** crate. Then, you will use the **`gtk`** crate to create a GUI window and widgets, **`gdk`** for handling keyboard input as well as coloring GTK3 widgets and the **`pango`** crate for manipulating text styles.

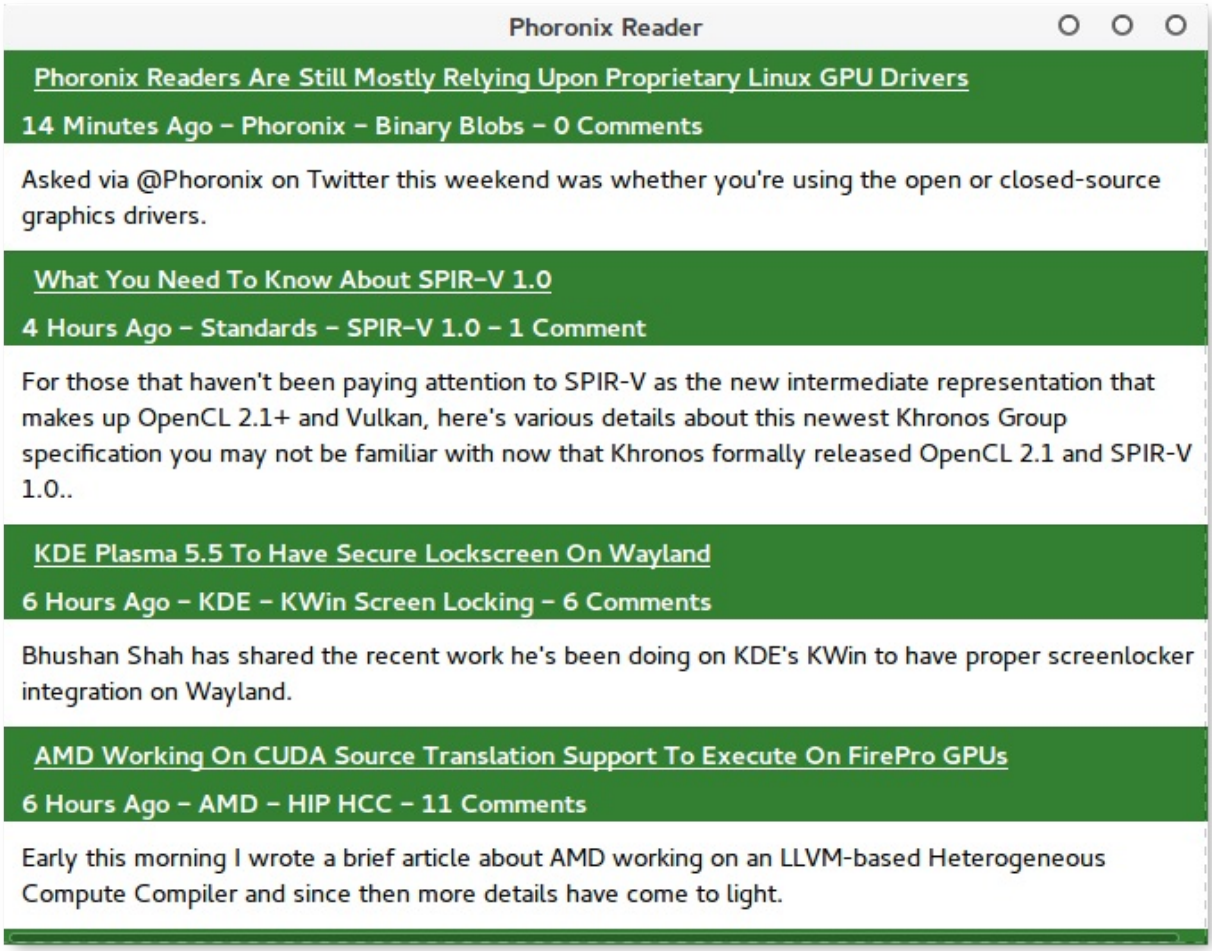
By the time you complete this tutorial, you should have a solid understanding of how to write applications in Rust using external libraries from **`Crates.io`** and **`GitHub`**, even without prior experience in programming. In the event that there is an area where you do not understand the material that well, feel free to send me a question and I will fix this tutorial to explain that portion in greater detail.

Screenshots

Command-Line Application



GTK3 Graphical User Interface



Source Code

The completed source code is available on `GitHub` at:

https://github.com/mmstick/phoronix-reader

You may clone the source code and compile it easily with `git`.

```
git clone https://github.com/mmstick/phoronix-reader
cd phoronix-reader
cargo run
```

Installing Rust and Cargo

Installing Rust and Cargo

Before you begin, you need to install both `Rust` and `Cargo` on your system. They should both be available in your Linux distribution's software repository. A `software repository` is a collection of packages available on a central server for download via a **package manager**. `Linux distributions` are operating systems whose sole mean of installing software is through these **software repositories** designed specifically for each distribution. Some **Linux distributions** feature more complete collections of software than others. I highly recommend a distribution from the **Arch Linux** family, such as **Antergos**.

If your Linux distribution does not offer either, you can find instructions available at **Rust** and **Cargo's** websites on how to install them for your platform. If you're using **Mac** or **Windows** though, you're using the **wrong** operating system.

The `Rust` Package

The `Rust` package contains the entirety of the **Rust Standard Library** alongside the `rustc` Rust compiler. A **library** is simply a collection of **public functions** that are made available to any application which imports them into their project. A **compiler** is a program that takes **source code** as input and translates it into a lower-level language, normally **Assembly**. They are used to convert **source code** into **machine code**, ultimately.

- **`Rust Homepage:`** https://www.rust-lang.org/
- **`Rust API Documentation:`** https://doc.rust-lang.org/stable/std/
- **`Cargo Getting Started:`** http://doc.crates.io/.
- **`GTK Documentation:`** http://gtk-rs.org/docs/gtk/

`Cargo`

If you are writing is a simple calculator without any external dependencies, `rustc` is all you need. However, you will need to use **Cargo** for **package management** of larger projects. The idea behind `package managers` is that users and developers shouldn't have to manually track down **dependencies** when installing new software. Instead, there should be a **package manager** that knows what **dependencies** are required that will automatically download, compile and/or install them.

As a **package manager**, **Cargo** reads from a single file, named `Cargo.toml`, in the root directory of your **Cargo project** to decide what **dependencies** are needed to be installed for your project. Before it begins compiling your project, it will download and compile all of the **libraries** listed under `[dependencies]` to get them ready for linking into your project's binary, if you are writing a binary. It will also **recurse** into the **dependencies** of your **dependencies** and grab them as well. If, instead, you are writing a library, no compiling is required at all.

Cargo also goes well beyond simple **package management** and also features the capability to **automatically generate documentation** for all libraries imported into a project using the `doc` argument. This documentation can then be viewed from a **web browser** or even hosted on a web server for others to view. This is the future of **API documentation**.

`Crates.io`

As you may have already noticed, **Cargo** hosts a universal **community repository** for **Rust crates** that Cargo can automatically download and build at compile time for your Rust projects at `crates.io`. A **crate** is simply a fancy term for a **Rust** software library. In your `Cargo.toml` file, you can specify what crates that you want in your project, as well as what version of each crate that you want. **Cargo** may also access **private** and **public git repositories**, allowing you to specify which **commit** or **branch** of the **git repository** you want to build with.

- **`Crates.io`**: http://crates.io/

Crates To Be Used

- **`hyper:`** https://crates.io/crates/hyper
- **`select:`** https://crates.io/crates/select
- **`term:`** https://crates.io/crates/term
- **`getopts:`** https://crates.io/crates/getopts
- **`gtk:`** https://github.com/gtk-rs/gtk
- **`gdk:`** https://github.com/gtk-rs/gdk
- **`pango:`** https://github.com/gtk-rs/pango

Functional Programming in Rust

Functional Programming in Rust

I will be using some **functional programming** concepts in this tutorial. Those who have never performed functional programming before may need some explanation. The strength in functional programming lies within how simple it is to perform calculations on lists of variables, regardless of their data type. A list of variables is known as a **vector** in Rust, abbreviated to `Vec`. Using functional techniques, you can manipulate vectors with a single line of code. This is all performed without having to rely on the traditional and complicated **for loop**.

`Iterator` Type

The `Iterator` type in Rust is used to perform much of the functional heavy-lifting. Any `Vec` of elements can be transformed into an `Iterator` using either the `iter()` or `into_iter()` functions. The former function, `iter()`, passes values of each element by reference to eliminate the need for copying, while `into_iter()` passes values by value -- copying each element.

`Maps`, `Filters` and `Folds`

The `Iterator` type contains basic methods such as `map()`, `filter()` and `fold()`, among many other highly useful methods. `Maps` apply a function to each element in a vector and return the result of each iteration to the next function. `Filters` works similarly only that it will only return elements that meet a certain criteria. `Folds` will apply a function whose purpose is to accumulate all the elements in the vector into a single value. At the end, the `collect()` function is used to return a new `Vec` of values.

How `Iterator` Works

You may think of methods like `map()`, `filter()` and `fold()` as specialized `for loops` that recurse across a vector using a series `next()` calls with a `function` or `closure` as the input to apply to each iteration. Each pass through these methods returns another `Iterator` type containing all of the results, so no matter how many methods you pass through, it will remain as an `Iterator` until it is collected with `collect()`.

`Lazy` Programming

Because Rust uses a **lazy** model of functional programming, like **Haskell**, it only computes what it needs. It will only perform the minimal amount of calculations needed to obtain the results required. In the following example, because `take(5)` was added after `filter()`, it will stop filtering after the fifth successful filter. It will then pass those five values to `map()` and each will be collected into a new `Vec` by `collect()`. Because of this, this is actually much more efficient than it seems at first glance to programmers that are used to programming in traditional non-lazy languages. This can actually be as fast as or even faster than a traditional **for loop**.

Functional Programming With Numbers

```
fn main() {
    let vector = (1..)          // Vector of infinite integers
        .iter()                 // Convert to Iterator
        .filter(|x| x % 2 != 0) // Collect odd numbers
        .take(5)                // Only take five numbers
        .map(|x| x * x)         // Square each number
        .collect();             // Return as a new Vec
    println!("{}", vector);     // Print result
}
```

Functional Programming With Strings

```
fn main() {
    let sentence = "This is a sentence in Rust.";
    let words: Vec<&str> = sentence
        .split_whitespace()
        .collect();
    let words_containing_i: Vec<&str> = words
        .into_iter()
        .filter(|word| word.contains("i"))
        .collect();
    println!("{:?}", words_containing_i);
}
```

See the Rust Book

Feel free to reference the official Rust documentation for the `String`, `Vec` and `Iterator` types. Many features aren't covered here since this program won't use them. I recommend trying out some functional programming as practice to get used to the idea. The official Rust Book is a good source of information as well.

Creating the Rust Project

Creating the Rust Project

We will begin by using the `cargo` tool to create our project directory for us. Of the arguments that you can use, the most important ones are `new`, `build`, `install` and `doc`. What each argument does should be self-explanatory at this point. By default, the `new` argument will create a library project, but since we are going to create a binary executable, we need to further pass the `--bin` flag along with the name of the project.

```
cargo new --bin "phoronix-reader"
```

You will notice that a new directory has been created. Inside this directory is a `Cargo.toml` file as well as a `src` directory. We will start by adding the `crates` that we want `Cargo` to build to the `Cargo.toml` file. The `crates` that we are going to use are `hyper` for making a `HTTP request` to download the HTML of the front page of `Phoronix` and `select` for conveniently obtaining just the information we want from that HTML page. Add the following lines at the bottom of `Cargo.toml`:

`Cargo.toml`

```
[dependencies]
hyper = "*"
select = "*"
```

Adding Crates To Your Source Code

Now we can enter the `src` directory where we will find the `main.rs` source code file. The `main.rs` file is required when working with binary projects. It will signify to `Cargo` to begin compiling a binary starting from `main.rs` as the source. When working on a library, you will instead start writing code from the `lib.rs` file. Cargo will automatically detect whether to compile a binary or a library based on the existence of either file.

Inside `main.rs`, enter the following to import these crates into the project:

```
extern crate hyper;
extern crate select;
```

Reading a Cached Copy of Phoronix

Reading a Cached Copy of Phoronix

At this moment, we are going to focus primarily on the `select` crate and ignore `hyper` for now. Access https://www.phoronix.com right now and copy the HTML source into a new file inside the `src` directory with `phoronix.html` as the name. We will simply open this file and store it inside of a `String` to test our program without flooding Phoronix with traffic.

The first function we will write will be a function to open that file and return it as a `String` for testing. We can do this by using the `include_str!()` macro, which takes a filepath as input and returns the contents as a `&str`. Because a `&str` has a lifetime that won't live beyond the function call, we will convert it into a `String` with `String::from()`. Here's the code:

```
// open_testing() returns `phoronix.html` as a String
fn open_testing() -> String {
    String::from(include_str!("phoronix.html"))
}
```

Now that we have this, we can test to see if our code is working by changing the main function as follows:

```
fn main() {
    let phoronix = open_testing();
    println!("{}", phoronix);
}
```

This will store the HTML inside the immutable `phoronix` variable and print that to `stdout` using the `println!()` macro. Now run `cargo run` to try out the changes. If your code is working, it will print the HTML to the terminal via `stdout`.

Implementing the Article Struct: Part 1

Implementing the `Article Struct`: Part 1

As always, before beginning any new project, take some time to ask yourself these questions:

Questions

1. What is the purpose of my program?
2. What information do I want to collect?
3. Where will I get that information?
4. How will I get that information?
5. What output do I want from that information?

Answers

1. The purpose of this program is scrapping articles from Phoronix's homepage.
2. The information we want to collect is the `title`, `summary`, `details` and `links` of each article.
3. The information will come directly from the HTML of the webpage.**
4. The `Hyper` crate provides tools for downloading the HTML
5. `Select` provides the tools for obtaining the information from the HTML.**
6. The information will be outputted directly to the terminal.

Writing the `Article Struct`

Now that know what we are trying to accomplish, we need to create a struct to hold all of the data we need from each article: the `title`, `summary`, `details` and `links`.

```
struct Article {
    title:   String,
    link:    String,
    details: String,
    summary: String,
}
```

Now that we have our struct, we can start implementing functions for our struct. Let's start by implementing a function for obtaining a `Vec` of `Articles` containing just the titles. Comment out all the variables in the struct except for `title` to prevent the program from generating warnings.

Importing `Select's` Features

Before we begin using the `select` crate, it's time to add some `use` statements for the particular features that we want to use from the crate. Add this beneath the `extern crate` lines:

```
use select::document::Document;
use select::predicate::{Class,Name};
use select::node::Node;
```

The purpose of this is primarily to reduce how many key presses we need to access the above features inside the `select` crate. It's easier to type `Document::from_str()` than it is to type `select::document::Document::from_str()`.

Implementing `Article's Methods`

There are two methods that we are going to implement for our `Article` struct: `get_articles()` and `new()`. The `get_articles()` method will create a `Vec` of `Articles` by mapping key data from each `node` to an `Article` using the `new` method and finally collecting the final results as a `Vec<Article>`.

The definition of `node` in this usage is the contents of a specifc HTML tag. As this program is collecting a list of `nodes` which are individual `<article>` tags, each iteration of an `<article>` will be passed on to the `new()` method to collect the data from it and return it as an `Article`.

`Article::get_articles()`

```
impl Article {
    fn get_articles() -> Vec<Article> {
        Document::from_str(&open_testing())
            .find(Name("article")).iter()
            .map(|node| Article::new(&node))
            .collect()
    }
}
```

As you may read from the above function, a new `Document` is created from the `&str` of the HTML. This document allows us to perform a `find()` on all tags in the `Document` whose name is `article`. If you look at the HTML source directly, you will notice that each article is contained inside of a unique `<article>` tag. Hence, we are only collecting information, or `nodes`, from those specific tags and mapping them to a method we have yet to create: `new()`.

`Article::new()`

Now let's implement `Article::new()`:

```
impl Article {
    fn get_articles() -> Vec<Article> {
        Document::from_str(&open_testing())
            .find(Name("article")).iter()
            .map(|node| Article::new(&node))
            .collect()
    }
    fn new(node: &node) -> Article {
        let header = node.find(Name("a")).first().unwrap();
        Article{ title: header.text() }
    }
}
```

Again, by looking at the HTML, we can find that the title is contained within an `<a>` tag. It is the first `<a>` tag inside the `<article>` node so we will only obtain the first item using `first()`. This returns a value that has the potential to error, but we will ignore errors and return the value with

`unwrap()`. It is generally better not to ignore errors though.

Once the information is collected, it is returned as a new `Article` type, assigning `header.text()` as the title. The `text()` function is used to convert a node into a `String` and ditching the other information not needed, so you will get an error without it.

Testing the New Code

Now we can go back to the `main()` function and modify it to use our new code:

```
fn main() {
    let phoronix_articles = Article::get_articles();
    for article in phoronix_articles {
        println!("{}", article.title);
    }
}
```

Try to compile and run this application with `cargo run`. Your source code should now look like this:

`main.rs`

```
// extern crate hyper;
extern crate select;
use select::document::Document;
use select::predicate::{Class,Name};
use select::node::Node;

fn main() {
    let phoronix_articles = Article::get_articles();
    for article in phoronix_articles {
        println!("{}", article.title);
    }
}

fn open_testing() -> String {
    String::from(include_str!("phoronix.html"))
}

struct Article {
    title:   String,
//    link:    String,
//    details: String,
//    summary: String,
}

impl Article {
    fn get_articles() -> Vec<Article> {
        Document::from_str(&open_testing())
            .find(Name("article")).iter()
            .map(|node| Article::new(&node))
            .collect()
    }
```

```
    fn new(node: &Node) -> Article {
        let header = node.find(Name("a")).first().unwrap();
        Article{ title: header.text() }
    }
}
```

Implementing the Article Struct: Part 2

Implementing the Article Struct: Part 2

The only items that we have yet to gather for the Article struct are `link`, `details` and `summary`. Let's implement these items in our existing `Article::new()` function.

Obtaining Links

The link is also inside the first `<a>` tag so we can simply obtain the `href` information directly, ignoring the text. This can be achieved with:

```
let mut link = String::from(header.attr("href").unwrap();
```

Links obtained on Phoronix's page don't include the URL of the homepage itself. We will manually add this later when printing the link. However, because some of the links start with `/` and others don't, this will cause us to print a double `//`. We will want to remove the first `/` if it exists. To do this, we can simply issue this line of code:

```
if link.starts_with("/") { assert_eq!(link.remove(0), '/'); }
```

This will remove the first character of `link` if the first character is `/` and will check to make sure that the character that we removed is a `/`. The `assert_eq!()` macro will panic with an error if something other than `/` is removed, so this will help us by error checking our code for us.

Obtaining Details

Now that we have the links taken care of, we need to get the details. The details are obtained within a HTML class, named `details` so we can just perform a `find()` for it as such:

```
let details = node.find(Class("details")).first().unwrap().text();
```

The above will search for each instance of `details`, only collect the first result, unwrap it and return it as a `String`. That's all we need here.

Replace `Add A Comment` with `0 Comments`

There is only one minor detail that we may want to fix here, which is to replace any instance of `Add A Comment` on details with no comments with `0 Comments` so as to not confuse yourself later. Update the previous declaration of `let details` with `let mut details` and add the following lines beneath it:

```
if details.contains("Add A Comment") {
    details = details.replace("Add A Comment", "0 Comments");
}
```

This will tell our program to check the contents of `details` and if the `String` contains `Add A Comment`, to replace it with a new `String` with `Add A Comment` replaced with `0 Comments`.

Obtaining Summaries

The last piece of information we want to collect is the summaries of each article. To do this, we will notice that each summary is stored in a single `<p>` tag. If we needed to collect multiple paragraphs we would write this a bit differently, but we only need one.

```
let summary = node.find(Name("p")).first().unwrap().text();
```

Collecting it as an `Article`

We can finally collect all of this information inside of a single `Article` by returning the following expression:

```
Article {
    title: header.text(),
    link: link,
    details: details,
    summary: summary,
}
```

Example

```
impl Article {
    ...
    fn new(node: &Node) -> Article {
        let header = node.find(Name("a")).first().unwrap();
        let mut link = String::from(header.attr("href").unwrap());
        if link.starts_with("/") { assert_eq!(link.remove(0), '/'); }
        let details = node.find(Class("details")).first().unwrap().te
        if details.contains("Add A Comment") {
            details = details.replace("Add A Comment", "0 Comments")
        }
        let summary = node.find(Name("p")).first().unwrap().text();
        Article {
            title: header.text(),
            link: link,
            details: details,
            summary: summary,
        }
    }
}
```

Testing Code

Now we only need to make a couple changes to our `main()` function so that we can see our new code in action:

```
fn main() {
    let phoronix_articles = Article::get_articles();
    for article in phoronix_articles {
        println!("Title:   {}", article.title);
        println!("Link:    https://www.phoronix.com/{}", article.link
```

```
        println!("Details: {}", article.details);
        println!("Summary: {}\n", article.summary);
    }
}
```

Now execute `cargo run` and see that our new program is almost complete.

Printing in Reverse Chronological Order

You might notice that the terminal output isn't ideal for printing to a terminal because the newest articles will be burried behind older articles. In order to remedy this problem and ensure that the newest articles are printed last, we can loop in reverse using the `rev()` function combined with `iter()`.

```
fn main() {
    let phoronix_articles = Article::get_articles();
    for article in phoronix_articles.iter().rev() {
        println!("Title:   {}", article.title);
        println!("Link:    https://www.phoronix.com/{}", article.link
        println!("Details: {}", article.details);
        println!("Summary: {}\n", article.summary);
    }
}
```

Getting HTML Pages From Hyper

Getting HTML Pages From Hyper

Now that our program is working, let's write a new function that utilizes the `hyper` crate for downloading the HTML page directly to get a live source. Let's start by adding the `use` cases for `hyper` and enabling it in our source code.

```
use hyper::Client;
use hyper::header::Connection;
use std::io::Read;
```

Downloading Phoronix's Homepage

And now we get to implementing the actual function. This function, `open_phoronix()`, will create a `Client` that will `get()` the homepage of Phoronix. It will then store the response inside a variable named aptly: `response`. Finally, we will restore the contents of the response inside a `String` named `body` and return `body`.

```
fn open_phoronix() -> String {
    let client = Client::new();
    let mut response = client.get("https://www.phoronix.com/").
        header(Connection::close()).send().unwrap();
    let mut body = String::new();
    response.read_to_string(&mut body).unwrap();
    return body;
}
```

`open_phoronix()` Usage

Now we just need to make a small change in our `Article::get_articles()` function to get it's input from `open_phoronix()` instead of `open_testing()`.

```
impl Article {
    fn get_articles() -> Vec<Article> {
        Document::from_str(&open_phoronix())
            .find(Name("article")).iter()
            .map(|node| Article::new(&node)).collect()
    }
    ...
}
```

Feel free to comment out `open_testing()` now that it is no longer required. Full source code should look as follows:

`main.rs` Code Example

```
extern crate hyper;
```

```
use hyper::Client;
use hyper::header::Connection;
use std::io::Read;
extern crate select;
use select::document::Document;
use select::predicate::{Class,Name};
use select::node::Node;

fn main() {
    let phoronix_articles = Article::get_articles();
    for article in phoronix_articles.iter().rev() {
        println!("Title:   {}", article.title);
        println!("Link:    https://www.phoronix.com/{}", article.link
        println!("Details: {}", article.details);
        println!("Summary: {}\n", article.summary);
    }
}

// fn open_testing() -> String {
//     String::from(include_str!("phoronix.html"))
// }

fn open_phoronix() -> String {
    let client = Client::new();
    let mut response = client.get("https://www.phoronix.com/").
        header(Connection::close()).send().unwrap();
    let mut body = String::new();
    response.read_to_string(&mut body).unwrap();
    return body;
}

struct Article {
    title:   String,
    link:    String,
    details: String,
    summary: String,
}

impl Article {
    fn get_articles() -> Vec<Article> {
        Document::from_str(&open_phoronix()).find(Name("article")).it
            .map(|node| Article::new(&node)).collect()
    }
    fn new(node: &Node) -> Article {
        let header = node.find(Name("a")).first().unwrap();
        let mut link = String::from(header.attr("href").unwrap());
        if link.starts_with("/") { assert_eq!(link.remove(0), '/'); }
        let details = node.find(Class("details")).first().unwrap().te
        if details.contains("Add A Comment") {
            details = details.replace("Add A Comment", "0 Comments");
        }
        let summary = node.find(Name("p")).first().unwrap().text();
        Article { title: header.text(), link: link, details: details,
```

```
    }
}
```

Splitting Article From Main

Splitting Article From Main

Now we are going to learn how to separate the code in our application into multiple files. We know that all source code for a binary program must be compiled starting from `main.rs`, but we can create as many source code files as we want. You may even create sub-directories to keep source code categorized when working with large software projects tens to hundreds or thousands of files.

We are going to split all of our `Article` code into it's own file to make our `main.rs` file more readable. Cut all of the code related to `Article` and paste it into a new file named `article.rs`. You will also have to include the `use` statements, but you don't need to cut the `extern crate` statements.

`article.rs`

Take careful note that when accessing types, variables and functions inside library files, you need to signify publicly-exposed code with the `pub` keyword. We don't need to mark `Article::new()` as `pub` though because this function is only called internally by `Article::get_articles()`.

We do need to make one change to `Article::get_articles()`, however, so that it can take an input containing a `&str` of the HTML from our `open_phoronix()` function that is still inside `main.rs`. Your code should look like this:

```
use select::document::Document;
use select::predicate::{Class,Name};
use select::node::Node;

pub struct Article {
    pub title:   String,
    pub link:    String,
    pub details: String,
    pub summary: String,
}

impl Article {
    pub fn get_articles(html: &str) -> Vec<Article> {
        Document::from_str(html).find(Name("article")).iter()
            .map(|node| Article::new(&node)).collect()
    }
    fn new(node: &Node) -> Article {
        let header = node.find(Name("a")).first().unwrap();
        let mut link = String::from(header.attr("href").unwrap());
        if link.starts_with("/") { assert_eq!(link.remove(0), '/'); }
        let details = node.find(Class("details")).first().unwrap().te
        if details.contains("Add A Comment") {
            details = details.replace("Add A Comment", "0 Comments")
        }
        let summary = node.find(Name("p")).first().unwrap().text();
```

```
        Article { title: header.text(), link: link, details: details,
    }
}
```

main.rs

Take note that to import our new `article.rs` file we have added these two lines:

```
mod article;
use article::Article;
```

This will allow us to continue using `Article` as we were before. Other than this change, one other change was needed to pass the HTML to our `Article::get_articles()` function:

```
let phoronix_articles = Article::get_articles(&open_phoronix());
```

With those changes, your `main.rs` file should now look like this.

```
extern crate hyper;
use hyper::Client;
use hyper::header::Connection;
use std::io::Read;
extern crate select;
mod article;
use article::Article;

fn main() {
    let phoronix_articles = Article::get_articles(&open_phoronix());
    for article in phoronix_articles.iter().rev() {
        println!("Title:   {}", article.title);
        println!("Link:    https://www.phoronix.com/{}", article.link
        println!("Details: {}", article.details);
        println!("Summary: {}\n", article.summary);
    }
}

// fn open_testing() -> String {
//     String::from(include_str!("phoronix.html"))
// }

fn open_phoronix() -> String {
    let client = Client::new();
    let mut response = client.get("https://www.phoronix.com/").
        header(Connection::close()).send().unwrap();
    let mut body = String::new();
    response.read_to_string(&mut body).unwrap();
    return body;
}
```

Try running `cargo run` again to verify that the program still works.

Creating a Phoronix CLI Front-End Library

Creating a Phoronix CLI Front-End Library

`homepage.rs`

Now we can go a step further and separate the `open_phoronix()` and `open_testing()` functions from `main.rs` into it's own library, `homepage.rs`. The `open_phoronix()` function shall be renamed as `online()` while `open_testing()` will be named `offline()`. That way, the code may be called conveniently with `homepage::online()` or `homepage::offline()`.

```
use hyper::Client;
use hyper::header::Connection;
use std::io::Read;

pub fn online() -> String {
    let client = Client::new();
    let mut response = client.get("https://www.phoronix.com/").
        header(Connection::close()).send().unwrap();
    let mut body = String::new();
    response.read_to_string(&mut body).unwrap();
    return body;
}

pub fn offline() -> String { String::from(include_str!("phoronix.html
```

`phoronix_cli.rs`

Now it might be a good idea to separate the printing code from `main.rs` into our new `phoronix_cli.rs` front-end. Let's call this new function `print()` so that we can later call this in `main()` with `phoronix_cli::print()`. Because we now need to get our homepage string from the `homepage.rs` file, we need to make that change too.

```
use article::Article;
use homepage;

pub fn print() {
    let phoronix_articles = Article::get_articles(&homepage::offline
    for article in phoronix_articles.iter().rev() {
        println!("Title:   {}", article.title);
        println!("Link:    https://www.phoronix.com/{}", article.link
        println!("Details: {}", article.details);
        println!("Summary: {}\n", article.summary);
    }
}
```

When you are finished with the program, make sure to make the correct changes in `main.rs` to reflect the newly-created libs. We will simply add `mod homepage` and `mod phoronix_cli` to the mod list and call our `print()` function in `phoronix_cli` with

`phoronix_cli::print()`.

`main.rs`

Now our `main.rs` file should look like such:

```
extern crate hyper;
extern crate select;
mod article;
mod homepage;
mod phoronix_cli;

fn main() {
    phoronix_cli::print();
}
```

Run `cargo run` again to make sure it still builds and runs.

Line-Wrapping Summaries

Line-Wrapping Summaries

So our program is working well and it's printing everything to the terminal in a readable manner. However, are there ways that we can improve the output to be easier on the eyes? Indeed, there is.

The Problem

One of the issues with the output of our program is that article summaries are printed as a single line and aren't being properly wrapped around a specific character limit. You may notice that words are being broken as they are forcefully wrapped around your terminal without a care. Your terminal doesn't care about keeping words together.

Challenge

This is a good mental exercise if you'd like to solve this problem yourself. Try to get your summaries to print by wrapping them around an 80 char limit. Make it even better by making your output like so:

```
Summary:
 - This is a sentence that our
 - program is outputting and
 - is keeping our words together.
```

Solution

This is one such solution to the problem:

Create a file named `linesplit.rs` and import it into your `main.rs` file by adding `mod linesplit;` with the rest of the mods. Then, inside the `phoronix.rs` file, add `use linesplit;` so that you can use it in that file.

The following function will take the summary as the first input and the character length to wrap words around as the second.

```
pub fn split_by_chars(input: &str, length: usize) -> Vec<String> {
    let words: Vec<&str> = input.split_whitespace().collect();
    let mut lines: Vec<String> = Vec::new();
    let mut current_line = String::with_capacity(length);
    let mut chars: usize = 0;
    let mut initialized = false;
    for word in words {
        if chars + word.len() >= length {
            lines.push(current_line.clone());
            current_line.clear();
            current_line.reserve(length);
            current_line = String::from(word);
            chars = word.len();
        } else if !initialized {
```

```
            current_line = String::from(word);
            chars = word.len();
            initialized = true;
        } else {
            current_line = current_line + " " + &word;
            chars += word.len() + 1;
        }
    }
    if !current_line.is_empty() { lines.push(current_line); }
    return lines;
}
```

Inside your printing function, change it so that it makes use of this function:

```
pub fn print() {
    let phoronix_articles = Article::get_articles(&homepage::offline
    for article in phoronix_articles.iter().rev() {
        println!("Title:   {}", article.title);
        println!("Link:    https://www.phoronix.com/{}", article.link
        println!("Details: {}", article.details);
        println!("Summary:");
        for line in linesplit::split_by_chars(&article.summary, 77) {
            println!(" - {}", line);
        }
        print!("\n");
    }
}
```

Colored Terminal Output

Colored Terminal Output

Wrapping words around a character limit is one way to improve readability, but coloring the output improves readability even moreso. We can use the `term` crate to change the color of `stdout`. Add the term crate to your `Cargo.toml` file:

```
term = "*"
```

Then we can add it to our `main.rs` file:

```
extern crate term;
```

And finally in our `phoronix.rs` file:

```
use term;
```

How To Use the `Term` Crate

Let's create a new function named `phoronix::print_homepage_colored()`. We begin by creating a mutable stdout terminal, which gets a handle of stdout and will later allows us to change the colors and attributes.

```
let mut terminal = term::stdout().unwrap();
```

If you want to change the color of the terminal, you can use the `fg()` function with a `term::color` enum as the input.

```
terminal.fg(term::color::BRIGHT_GREEN).unwrap();
```

Now if you want to make your text bold, use the `attr()` function with a `term::Attr` enum as the input.

```
terminal.attr(term::Attr::Bold).unwrap();
```

The next time you print to the terminal, it will be bright green and bold. You can later reset the terminal back to defaults using the `reset()` function.

Writing a Colored Printer Function

`phoronix_cli.rs`

If we put all of this together, we can create a new function, which we shall name `print_colored()`.

```
pub fn print_colored() {
    let phoronix_articles = Article::get_articles(&homepage::offline
    let mut terminal = term::stdout().unwrap();
    for article in phoronix_articles.iter().rev() {
```

```
        print!("Title:   ");
        terminal.fg(term::color::BRIGHT_GREEN).unwrap();
        terminal.attr(term::Attr::Bold).unwrap();
        println!("{}", article.title);
        terminal.reset().unwrap();
        print!("Link:    ");
        terminal.fg(term::color::BRIGHT_CYAN).unwrap();
        println!("https://www.phoronix.com/{}", article.link);
        terminal.reset().unwrap();
        println!("Details: {}\nSummary:", article.details);
        for line in linesplit::split_by_chars(&article.summary, 77).i
            print!(" - ");
            terminal.attr(term::Attr::Bold).unwrap();
            println!("{}", line);
            terminal.reset().unwrap();
        }
        println!("");
    }
}
```

main.rs

Now we can rewrite `main.rs` so that your copy should look precisely as this:

```
extern crate hyper;
extern crate select;
extern crate term;
mod article;
mod homepage;
mod linesplit;
mod phoronix_cli;

fn main() {
    phoronix_cli::print_colored();
}
```

Managing Program Flag Arguments With Getopts

Managing Program Flag Arguments With Getopts

The `getopts` crate provides a simple library for parsing command-line arguments given to the program. We will first be using this crate to determine whether or not we should disable colors, but will ultimately whether we should launch a graphical or command-line interface.

Lets start by adding `getopts` to our `Cargo.toml` and `main.rs` files.

`Cargo.toml`

```
getopts = "*"
```

`main.rs`

```
extern crate getopts;
```

Configuring and Parsing `Flags`

The absolute first thing that needs to be performed is to get a `Vec` of `Strings` containing a list of all the arguments that were supplied to the program. Inside the `main()` function, add the following at the very top:

```
let args: Vec<String> = std::env::args().collect();
```

Next we need to configure all of the `flags` that our program will accept. A `flag` is a command-line switch, such as `-h` or `--help`, which flags to our program what it should be trying to do.

We will first initialize a new `getopts::Options` and make it mutable so that we can add on all of the arguments that we want to use. For now, we will just define options for disabling colors and displaying a help message.

```
let mut opts = getopts::Options::new();
opts.optflag("n", "no-color", "prints without colors");
opts.optflag("h", "help", "show this information");
```

Next we will actually parse the list of arguments from our `args` variable with `opts` using the `parse()` method available to `getopts::Options`. We want to ignore the first variable in `args` though because, as with almost every other programming language, the first argument is the name of the running program. We can do this by passing `&args[1..]` to signify to skip the 0th variable.

```
let matches = opts.parse(&args[1..]).unwrap();
```

Finally we will check what was matched and perform actions based on that.

```
if matches.opt_present("h") { print_help(); return; } // Print help a
match matches.opt_present("n") {
    true => phoronix_cli::print(),
    false => phoronix_cli::print_colored(),
};
```

Adding a Help Function

You may have noticed that we are calling a `print_help()` function but we don't actually have a `print_help()` function. That's because we are going to do that right now.

```
fn print_help() {
    println!("Prints the latest information from Phoronix.");
    println!("    -h, --help     : show this information");
    println!("    -n, --no-color : prints without colors");
}
```

Example

```
extern crate hyper;
extern crate select;
extern crate term;
extern crate getopts;
mod article;
mod hompage;
mod linesplit;
mod phoronix_cli;

fn main() {
    let args: Vec<String> = std::env::args().collect();
    let mut opts = getopts::Options::new();
    opts.optflag("n", "no-color", "prints without colors");
    opts.optflag("h", "help", "show this information");
    let matches = opts.parse(&args[1..]).unwrap();
    if matches.opt_present("h") { print_help(); return; }
    match matches.opt_present("n") {
        true => phoronix_cli::print(),
        false => phoronix_cli::print_colored(),
    };
}

fn print_help() {
    println!("Prints the latest information from Phoronix.");
    println!("    -n, --no-color : prints without colors");
    println!("    -h, --help     : show this information");
}
```

Adding an Optional GUI Flag to Getopts

Adding an Optional GUI Flag to Getopts

Now that have `getopts` set, it's time to add an option to allow us to launch a GTK GUI instead of only providing command-line output. Before we begin though, we need to import the following crates for the upcoming GTK chapters: `gtk`, `gdk` and `pango`. However, we will need to use the latest `git` versions because the current stable packages conflict with some libs in `hyper`. This issue will probably be fixed the next time these packages are updated.

`Cargo.toml`

```
gtk = { git = "https://github.com/gtk-rs/gtk" }
gdk = { git = "https://github.com/gtk-rs/gdk" }
pango = { git = "https://github.com/gtk-rs/pango" }
```

And now we just need to add these crates to our `main.rs` file.

`main.rs`

```
extern crate gtk;
extern crate gdk;
extern crate pango;
```

Implementing the `GTK3 GUI Flag`

We will first add an extra flag argument for selecting the GUI.

```
opts.optflag("g", "gui", "display in a GTK3 GUI");
```

And then we will add an extra line to our help function to display the new flag option.

```
fn print_help() {
    println!("Prints the latest information from Phoronix.");
    println!("    -h, --help     : show this information");
    println!("    -g, --gui      : launches a GTK3 GUI instead of ou
    println!("    -n, --no-color : prints to stdout without using co
}
```

Choosing Between `CLI` and `GUI`

And then we need to change our code to let the program launch the GUI code instead of the CLI code. You should have your code look like this:

```
match matches.opt_present("g") {
    true => phoronix_gui::launch(),
    false => {
        match matches.opt_present("n") {
            true => phoronix_cli::print_homepage(),
```

```
            false => phoronix_cli::print_homepage_colored(),
        };
    },
};
```

This won't actually compile or do anything yet though because we have yet to create the `phoronix_gui.rs` file containing the `launch()` function.

Creating a GTK3 GUI Window

Creating a GTK3 Window

It's finally time to gain some experience with **GUI programming** using **GTK** from the `gtk-rs` wrapper project.

Setting the Use Statements

Like the command-line front-end, we are going to need to import `article::Article` and `homepage` for getting the homepage and collecting articles. However, now we are going to also import `gtk` for creating GUI windows and widgets, `gtk::traits::*` to allow widgets to use all their traits, `gdk::ffi::GdkRGBA` for creating colors to color our widgets and finally `pango` for manipulating font styles.

```
use article::Article;
use homepage;
use gtk;
use gtk::traits::*;
use gdk::ffi::GdkRGBA;
use pango;
```

Initializing GTK

The absolute first step to take when setting up a GTK GUI is to first initialize GTK. The following function will attempt to initialize GTK, and if it fails, will panic with an error that it failed to initialize.

```
gtk::init().unwrap_or_else(|_| panic!("Failed to initialize GTK."));
```

After this step, you will then set up your GUI, but at the very end you must end your `main()` function with `gtk::main()`, which will actually start your GUI application.

```
pub fn launch() {
    gtk::init().unwrap_or_else(|_| panic!("Failed to initialize GTK.'

    gtk::main();
}
```

Creating the Main Window

Between the `init()` and `main()` functions, we will begin setting up all of our GTK widgets and window. We will start by creating a **Toplevel Window**. The syntax for creating widgets is universally the same. Each widget provides a `new()` method, only varying in the input variables that it accepts. the `gtk::Window` type takes a `WindowType` enum as input to tell the wrapper what type of window to create. The main window of a program is of the `Toplevel` type.

```
let window = gtk::Window::new(gtk::WindowType::Toplevel).unwrap();
```

Now we need to make some configurations to our newly-created `gtk::Window` widget. The information that we want is the `default_size()`, `set_title()` and `connect_delete_event()`. The first, `default_size()`, will set the dimensions of the window upon launch. Then, `set_title()` will set the title of the window. Finally, `connect_delete_event()` will define the action to take when the window is closed.

```
window.set_title("Phoronix Reader");
let (width, height) = (600, 500);
window.default_size(width, height);
```

The hard part is understanding this next part: `connect_delete_event()`. All of the `connect` events take a function or closure as input, which is where the confusion lies in. This is how this event is generally configured:

```
window.connect_delete_event(|_,_| {
    gtk::main_quit();
    gtk::signal::Inhibit(true)
});
```

This signifies that it takes no variables as input into the closure, and simply executes `gtk::main_quit()` and `gtk::signal::Inhibit(true)`, which tells GTK not to interrupt this action.'

We can combine all of this together into a new function specific for configuring the window. We will pass the window to the function by reference as that's simply the best practice when working with GTK widgets.

```
fn configure_window(window: &gtk::Window) {
    window.set_title("Phoronix Reader");
    let (width, height) = (600, 500);
    window.set_default_size(width, height);
    window.connect_delete_event(|_,_| {
        gtk::main_quit();
        gtk::signal::Inhibit(true)
    });
}
```

Then we can call it in our launch function as such:

```
configure_window(&window);
```

To actually show the window after the program starts, you need to use the `show_all()` method.

```
window.show_all();
```

`phoronix_gui.rs` Review

```
use article::Article;
use homepage;
use gtk;
use gtk::traits::*;
use gdk::ffi::GdkRGBA;
use pango;
```

```
fn configure_window(window: &gtk::Window) {
    window.set_title("Phoronix Reader");
    let (width, height) = (600, 500);
    window.set_default_size(width, height);
    window.connect_delete_event(|_,_| {
        gtk::main_quit();
        gtk::signal::Inhibit(true)
    });
}

pub fn launch() {
    gtk::init().unwrap_or_else(|_| panic!("Failed to initialize GTK.'

    let window = gtk::Window::new(gtk::WindowType::Toplevel).unwrap(
    configure_window(&window);

    window.show_all();

    gtk::main();
}
```

Adding Widgets to the Window

Adding Widgets To GTK3

Now that we have our window, which currently does nothing but display a window itself, it's time to start generating some **widgets** to attach to the window. A **widget** is simply a GUI object that attaches to a window, like a text box or a label.

Create a Scrolled Window

The information that we are collecting will be displayed in a **scrolled window**, and because a `gtk::Window` can only contain one widget, that widget is going to be a `gtk::ScrolledWindow`. The `new()` method of a `gtk::ScrolledWindow` takes two input variables which are used to control alignment. We aren't going to set any alignment options so we will simply tell them `None`.

```
let scrolled_window = gtk::ScrolledWindow::new(None, None).unwrap();
```

It would also be a good idea to set a minimum width for our **scrolled_window** widget.

```
scrolled_window.set_min_content_width(600);
```

The new **scrolled_window** variable is useless by itself with no widgets to attach to it, however, so we need to create all the widgets we need for each of our articles. However, just like a `gtk::Window`, a `gtk::ScrolledWindow` will only allow one widget to be attached to it, so we will use a `gtk::Box` as a container for all of the articles.

Create a Container

A `gtk::Box` can have any number widgets attached to them, and are laid out either horizontally or vertically with `gtk::Orienation::Horizontal` and `gtk::Orientation::Vertical` respectively. You may also define the margins between widgets, but it's generally best to leave this at 0.

```
let container = gtk::Box::new(gtk::Orientation::Vertical, 0).unwrap(
```

Collecting Articles

Before we can start the creation of widgets for our GUI, we need to collect the list of **Articles**.

```
let articles = Article::get_articles(&homepage::offline());
```

Creating Widgets From the `Vec<Article>`

Each article will consist of a `gtk::Box` that contains the entire Article. First is a `gtk::LinkButton` containing both the **Title** and **URL**. The **details** and **summaries** will be contained within their own respective `gtk::TextView` widgets. To keep our `launch()` function cleaned up, we will create a new function specifically for the creation of the list of articles.

```
fn generate_article_widgets(container: &gtk::Box, articles: &Vec<Arti
    for article in articles {
        // Code Here
    }
}
```

This function will take the **container** variable we created earlier, along with the `Vec` of `Articles` as input, and simply iterate over each element -- attaching a new GUI widget with each loop.

Obtaining the Title and URL Widget

Let's start by creating a widget for the title and link, where we will create a `gtk::LinkButton`. The `gtk::LinkButton::new_with_label()` function takes two inputs: the URL and a label as `&str` types. By default, this will be centered so we can use the `set_halign()` method to set a left alignment with `gtk::Align::Start`. We can get the URL via `article.link` and the label with `article.title`.

```
for article in articles {
    let url = format!("https://phoronix.com/{}", article.link);
    let title_and_url = gtk::LinkButton::new_with_label(&url, &artic
    title_and_url.set_halign(gtk::Align::Start);
}
```

Obtaining Details

The next step is creating a `gtk::TextView` widget containing the details obtained from `article.details`. Like our title widget, we also want to left-align the text, and this time also set a left and right margin. The `set_left_margin()` and `set_right_margin()` may be used to specify how many pixels to use for the margins of the text inside the `gtk::TextView`. We also do not want users to be able to edit the text, so we will disable editing with `set_editable(false)`. You can't set the text during creation of the widget though, so that will have to be defined after creation using `get_buffer()` and `set_text()`.

```
    let details = gtk::TextView::new().unwrap();                // Crea
    details.set_halign(gtk::Align::Start);                      // Set
    details.set_left_margin(10);                                // 10 |
    details.set_right_margin(10);                               // 10 |
    details.set_editable(false);                                // Disa
    details.get_buffer().unwrap().set_text(&article.details); // Set
```

Obtaining Summaries

Now all we need to get is a widget for the summaries. The process is very much the same, but we will use a few extra features for setting how the widget operates, namely defining a wrap mode with `set_wrap_mode()` and setting the number of pixels above and below lines with `set_pixels_above_lines()` and `set_pixels_below_lines()`.

```
    let summary = gtk::TextView::new().unwrap();                // Crea
    summary.set_wrap_mode(gtk::WrapMode::Word);                 // Wra
    summary.set_left_margin(10);                                // 10 |
    summary.set_right_margin(10);                               // 10 |
    summary.set_pixels_above_lines(10);                         // 10 |
    summary.set_pixels_below_lines(10);                         // 10 |
```

```
    summary.set_editable(false);                                    // Disa
    summary.get_buffer().unwrap().set_text(&article.summary); // Set
```

Add Widgets to Container

Now all we just have to do is add these widgets to the **container** and we are pretty much finished with this function, albeit we have yet to perform any kind of coloring to these widgets.

```
container.add(&title_and_url);
container.add(&details);
container.add(&summary);
```

Adding Widgets to the Window and Displaying Them

Adding the **container** to our **scrolled_window**, and the **scrolled_window** to the **window** should be pretty straightforward.

```
scrolled_window.add(&container);
window.add(&scrolled_window);
window.show_all();
```

Review

phoronix_gui.rs

```
use article::Article;
use homepage;
use gtk;
use gtk::traits::*;
use gdk::ffi::GdkRGBA;
use pango;

pub fn launch() {
    gtk::init().unwrap_or_else(|_| panic!("Failed to initialize GTK.'

    // Create widgets for the articles
    let container = gtk::Box::new(gtk::Orientation::Vertical, 0).unwı
    let articles = Article::get_articles(&homepage::online());
    generate_article_widgets(&container, &articles);

    // Insert the articles into a scrolled window
    let scrolled_window = gtk::ScrolledWindow::new(None, None).unwra
    scrolled_window.set_min_content_width(600);
    scrolled_window.add(&article_box);

    // Add the scrolled window to a main window
    let window = gtk::Window::new(gtk::WindowType::Toplevel).unwrap(
    configure_window(&window);
    window.add(&scrolled_window);
    window.show_all();

    gtk::main();
}
```

```
// configre_window configures the given window.
fn configure_window(window: &gtk::Window) {
    window.set_title("Phoronix Reader");
    let (width, height) = (600, 500);
    window.set_default_size(width, height);
    window.connect_delete_event(|_,_| {
        gtk::main_quit();
        gtk::signal::Inhibit(true)
    });
}

// generate_article_widgets takes a vector of articles as well as a g
// with widgets generated from each article
fn generate_article_widgets(article_box: &gtk::Box, articles: &Vec<Ar
    for article in articles {
        // Creates the title as a gtk::LinkButton for each article
        let url = format!("https://phoronix.com/{}", article.link);
        let title_and_url = gtk::LinkButton::new_with_label(&url, &ar
        title_and_url.set_halign(gtk::Align::Start);
        title_and_url.set_margin_start(0);

        // Details of the article inside of a gtk::TextView
        let details = gtk::TextView::new().unwrap();
        details.set_halign(gtk::Align::Start);
        details.set_left_margin(10);
        details.set_right_margin(10);
        details.set_editable(false);
        details.get_buffer().unwrap().set_text(&article.details);

        // Summary of the article inside of a gtk::TextView
        let summary = gtk::TextView::new().unwrap();
        summary.set_wrap_mode(gtk::WrapMode::Word);
        summary.set_left_margin(10);
        summary.set_right_margin(10);
        summary.set_pixels_above_lines(10);
        summary.set_pixels_below_lines(10);
        summary.set_editable(false);
        summary.get_buffer().unwrap().set_text(&article.summary);

        // Attach the title+url, details and summary to the article_b
        container.add(&title_and_url);
        container.add(&details);
        container.add(&summary);
        container.add(&gtk::Separator::new(gtk::Orientation::Horizon
    }
}
```