Draft. Please do not include this page.

Single chapter compiled via LaTeX.

Shing Lyu, April 25, 2022. Git commit: 1ef1cd7

Notes for the production team

Whitespace marker

You will see pink underline markers like these: `a␣b␣c`. One marker indicates one whitespace. So there is exactly one whitespace between "a" and "b" and one whitespace between "b" and "c". You'll also see them in code blocks like this:

```
#[derive(StructOpt)]
struct␣Options␣{
␣␣␣␣//␣...
␣␣␣␣#[structopt(short␣=␣"f",␣long␣=␣"file",␣parse(from_os_str))]
␣␣␣␣///␣Load␣the␣cat␣picture␣from␣the␣specified␣file
␣␣␣␣catfile:␣Option<std::path::PathBuf>,
}
```

These markers are only for you to see how many whitespaces there are. Please keep the exact number of whitespaces for inline code and code blocks. **Please do NOT include the markers during layout**.

The layout output should look like: `a b c`

```
#[derive(StructOpt)]
struct Options {
    // ...
    #[structopt(short = "f", long = "file", parse(from_os_str))]
    /// Load the cat picture from the specified file
    catfile: Option<std::path::PathBuf>,
}
```

Auto line wrapping marker

You'll also see arrows near the end of the line and beginning of the second line:

```
This is a very very very very very very very very very very ↵
    ↳very very very long line that was wrapped automatically.
```

This indiciates the original line was too long and it was automatically wrapped. **Please include these arrows.**

CHAPTER 3

Creating graphical user interfaces (GUIs)

Command-line tools are handy in some situations like small tools that don't require visual interaction or batch processing. But command-line programs' user interface is usually limited to text input/output and files. This is sometimes not sufficient when 2D (or even 3D) visual interaction is required. So in this chapter, we'll break out of the constraint of the command-line and talk about graphical user interfaces (GUIs).

Our end goal in this chapter is to show you how to build cross-platform desktop applications. Although there are frameworks like Electron[1], which allows you to build a desktop app in HTML, CSS, and JavaScript, they are actually wrapping a browser engine inside. Therefore the developer experience will be more close to building a website or web app, not like coding a native desktop application. We'll choose a framework that showcases the experience of building a native application in Rust.

As a bridge between command-line programs and actual GUI apps, we'll first introduce the text-based user interface (TUI). A TUI looks like a GUI, but it's drawn with text characters. Thus it can be created in a terminal environment. But because we are drawing with text characters, the resolution will be very low, and the screen real estate is very limited. Nevertheless, TUI is a good way to understand the high-level concept of event-driven architecture that is common in GUI programs. Once we acquire the knowledge of how a TUI program is structured, we'll apply that knowledge in implementing a full-fledged GUI program.

3.1 What are we building?

To avoid distraction from complex business logic, we'll be building a simplified version of the `catsay` program in TUI and GUI. For TUI, we'll build:

- An interactive form to receive the message. (Figure 3.1)
- An optional checkbox for the `--dead` option.
- Once the user clicks "OK", a dialog box will pop up and show a cat saying the message. (Figure 3.2)

[1] https://electronjs.org/

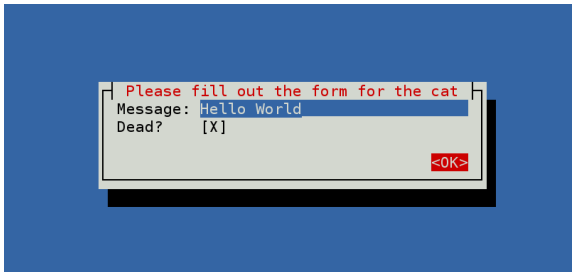
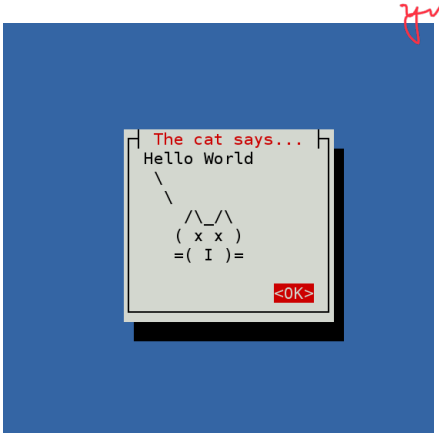


Figure 3.1: The input form of the TUI program

We'll then move on to build a GUI that has the same input as the TUI program. But this time, instead of the ASCII-art cat, we'll show a picture of a real cat (Figure 3.3).



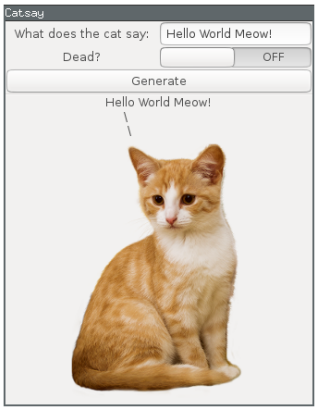


Figure 3.2: An example of the TUI program ***Figure 3.3:*** An example of the GUI program

We'll be building the GUI using `gtk-rs`, which is a Rust binding for the GTK+ 3 library and its underlying libraries. We'll first build the GUI in pure Rust code, and then we'll demonstrate how to use Glade, a user interface design tool that can help us design the layout in a more intuitive and easy-to-manage way. There are many other GUI libraries and frameworks out there; we'll discuss why we choose `gtk-rs` in the section *Other alternatives*.

3.2 Building a text-based user interface

We built a command-line tool in chapter 2, and we use `println!()` for most of our output. The problem with it is that we can only output one line at a time. Although we can create some kind of ASCII-art image by carefully align the lines we print, it won't really scale if we want to draw windows, dialog boxes, and buttons. Not to mention handle keyboard input and mouse clicks. Thankfully some libraries can help you build UI components easily. They are known as "text-based user interface" libraries. One example is the `ncurses` library for Unix-like systems. `Ncurses` stands for "new curses" because it was the "new" (in the 1990s) version of the old curses library in System V Release 4.0 (SVr4). We'll be using the Cursive crate, which uses `ncurses` by default. We'll discuss other alternatives at the end of the chapter, but we chose Cursive because it hides away many low-level details, so it's very easy to work with.

We'll be using the default `ncurses` backend for simplicity, but if you need cross-system support for Windows and/or macOS, you can choose other backends like `pancurses` or `crossterm`. To use ncurses we need to install `ncurses` on our system. For Ubuntu, simply run the following in the shell:

```
sudo apt-get install libncursesw5-dev
```

Then we can create a new project with `cargo new` and add the `Cursive` crate to `Cargo.toml`:

```
[dependencies]
cursive = "0.11.2"
```

In our `src/main.rs` we can write the minimal code shown in Listing 3.1.

Listing 3.1: Basic skeleton code for Cursive TUI

```
extern crate cursive;

use cursive::Cursive;

fn main() {
        let mut siv = Cursive::default();

        siv.run(); // starting the event loop
}
```

This will create a Cursive root object with `Cursive::default()`, and start the event loop with `siv.run()`. The event loop is a fundamental concept in building user interfaces. For a command-line program, interactions are usually limited to one input and one output. If we need to take user input, we have to pause the execution of the program and wait for the user input to finish. No other operations or output can be processed at that time. But in a GUI, it might be expecting multiple inputs, for example, keypress or mouse

click on any of the many buttons on a program's user interface. Since we can't predict which input will be triggered first, the program (conceptually) runs in an infinite loop that handles whatever next input that is triggered. If we have registered some event handler, then the event loop will invoke the handler when the event happens. For example, if we have an OK button in a dialog box. Clicking it will trigger a button click event, which might be handled by a handler that closes this dialog box.

3.3 Showing a dialog box

If we now run `cargo run`, we can see a blue screen (Figure 3.4). But that's not very exciting. To display our cat ASCII-art on the screen we just created, we add the code in Listing 3.2 to our `main.rs`:

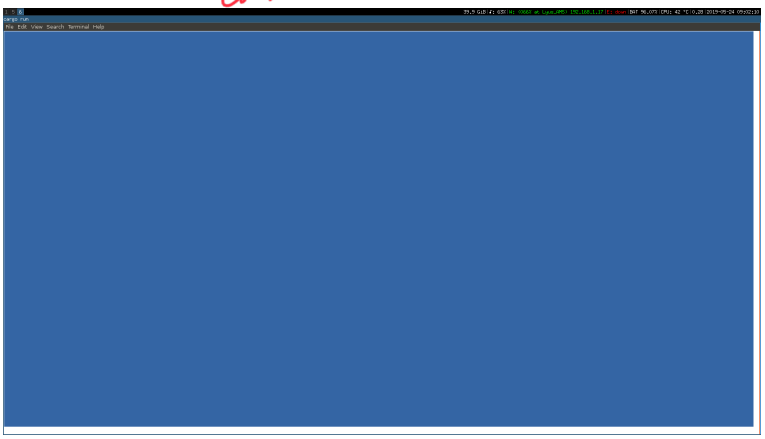


Figure 3.4: An empty Cursive screen

Listing 3.2: Showing a TextView

```
extern crate cursive;

use cursive::Cursive;
use cursive::views::TextView; // Import the TextView

fn main() {
    let mut siv = Cursive::default();
    let cat_text = "Meow!
 \\
  \\
    /\\_/\\
   (_o_o_)
    =(_I_)=";
```

```
    // Declaring the app layout
    siv.add_layer(TextView::new(cat_text));

    siv.run();
}
```

Notice that before we start the event loop, we set up the content of the app with:

```
siv.add_layer(TextView::new(cat_text));
```

We created a `TextView` to hold our cat ASCII-art. Views are the main building blocks of a Cursive TUI program. There are many pre-built views in the `cursive::views` module, for example, buttons, checkboxes, dialogs, progress bars, etc.. We can also implement custom views by implementing the `View` trait. The `TextView` is used to hold a fixed text, which was passed in as a constructor parameter. But the newly created `TextView` is not visible yet because it is not part of the main program. We can add it as a layer to the main Cursive program we just created by `siv.add_layer()`. Cursive uses layers to create a stacked view of the components (i.e., `Views`). Layers are stacked together such that the top-most one will be active and can receive input. They are also rendered with a shadow, so they look like 3D layers stacked together. You can see how that looks like in Figure 3.5.

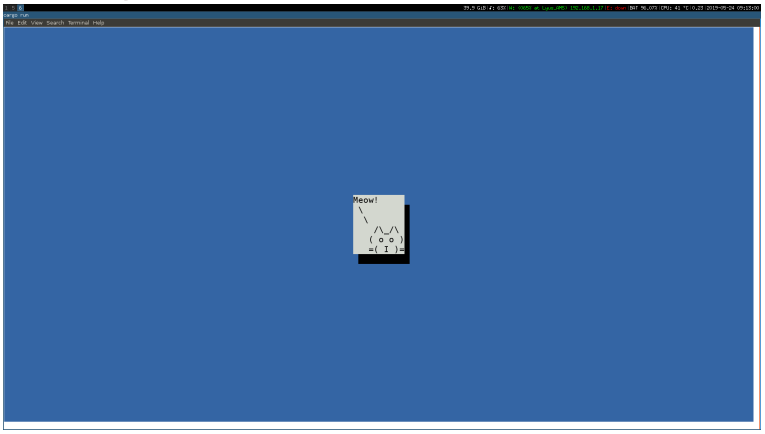


Figure 3.5: A TextView showing the cat

3.4 Handling simple keyboard inputs

Up until now our TUI program only produces output but can't handle any input. To close the program we have to press `Ctrl+C` to send a interrupt signal to force terminate it. We can try to make the program respond to the ESC key press and close the program gracefully by calling `Cursive.quit()`. We can modifiy our code into Listing 3.3

Listing 3.3: Listen to ESC key and close the program

```
// ...
use cursive::event::Key;

fn main() {
    let mut siv = Cursive::default();
    let cat_text = // ...

    siv.add_layer(TextView::new(cat_text));

    // Listen to Key::Esc and quit
    siv.add_global_callback(Key::Esc, |s| s.quit());

    siv.run();
}
```

In the code, we set up a global callback with `siv.add_global_callback()`. This function takes two arguments: an event and a callback function (`cb`). Whenever an event occurs, the callback function (closure) will be executed. We assigned `cursive::event::Key::Esc` as the event, which is triggered when the ESC key is pressed. In the callback argument we passed a closure `|s| s.quit()`. The `s` is a mutable reference to the `Cursive` struct itself, so `s.quit()` will gracefully quit the program.

The closure we created does not get executed right away, nor does the `siv.add_global_callback()` function blocks the execution until a key is pressed. The line simply registers the callback and continues the execution of the program. When the next line `siv.run()` is executed, the event loop starts and waits for keypresses and mouse clicks. By using a non-blocking event-based system, our user interface becomes more responsive to user input, and we are not limited to one kind of interaction at a time. We can set up multiple event handlers so they can handle different kinds of events regardless of the order. We'll see more event handlers in the coming sections.

3.5 Adding a dialog

The user interface we just created still feels a little rough around the edge. To give the program a more sophisticated look and feel, we can wrap the `TextView` with a `Dialog` (Listing 3.4).

Listing 3.4: Displaying a dialog

```
extern crate cursive;

use cursive::views::{Dialog, TextView};
use cursive::Cursive;

fn main() {
    let mut siv = Cursive::default();
    let cat_text = // ...

    siv.add_layer(
        Dialog::around(TextView::new(cat_text))
            .button("OK", |s| s.quit())
    );

    siv.run();
}
```

We use `Dialog::around()` to wrap our `TextView`, this will wrap the `TextView` inside the Dialog. We can also add a button to the dialog with a label ("OK") and a callback (`|s| s.quit()`). This callback will be triggered when the button is clicked. One nice feature about Cursive is that it supports keyboard and mouse interaction out-of-box, so we can close the program by either hitting the ENTER (Return) key or double-click the "OK" with the mouse. (Figure 3.6)

3.6 Multi-step dialogs

We are not limited to just one static layer at a time. We can actually build a multi-step flow. In the first step, the user is prompted to fill in a form and press "OK", then we hide the form and display the cat pictures using the information provided in the form. We can implement this as in Listing 3.5.

Listing 3.5: Mutli-step form

```
extern crate cursive;

use cursive::traits::Identifiable; // for .with_id(), .
    ↪ call_on_id()
use cursive::views::{Checkbox, Dialog, EditView, ListView,
    ↪ TextView};
```

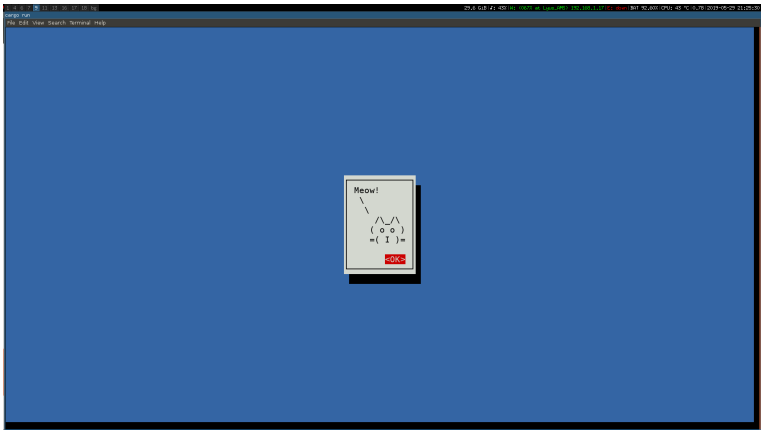


Figure 3.6: Displaying a dialog with an OK button

filename

```
use cursive::Cursive;

// wrap all form fields value in one struct so we can pass
    ↪ around easily
struct CatsayOptions<'a> {
    message: &'a str,
    dead: bool,
}

fn input_step(siv: &mut Cursive) {
    siv.add_layer(
        Dialog::new()
            .title("Please fill out the form for the cat") //
    ↪ setting the title
            .content(
                ListView::new()
                    .child("Message:", EditView::new().with_id("
    ↪ message"))
                    .child("Dead?", Checkbox::new().with_id("
    ↪ dead")),
            )
            .button("OK", |s| {
                let message = s
                    .call_on_id("message", |t: &mut EditView| t.
    ↪ get_content())
                    .unwrap();
                let is_dead = s
```

```
                    .call_on_id("dead", |t: &mut Checkbox| t.
    ↪ is_checked())
                    .unwrap();
                let options = CatsayOptions {
                    message: &message,
                    dead: is_dead,
                };
                result_step(s, &options) // [2]
            }),
    );
}

fn result_step(siv: &mut Cursive, options: &CatsayOptions) {
    let eye = if options.dead { "x" } else { "o" };

    let cat_text = format!(
        "{msg}
 \\
  \\
    /\\_/\\
   ( {eye} {eye} )
   =( I )=",
        msg = options.message,
        eye = eye
    );

    siv.pop_layer(); // [3]
    siv.add_layer( // [4]
        Dialog::around(TextView::new(cat_text))
            .title("The cat says...")
            .button("OK", |s| s.quit()),
    );
}

fn main() {
    let mut siv = Cursive::default();

    input_step(&mut siv); // [1]

    siv.run();
}
```

This example is slightly more complex, so we'll break it down a little bit. First, let's walk through the high-level flow. In the `main()` function, instead of directly setting up the layout with `add_layer()`, we move all the layout code into the `input_step(&mut siv)` function ([1]). Inside `input_step` we set up a form, which we'll discuss

in detail later. Notice that it has a button called "OK". In the callback function of the button, we call `result_step(s, &options)` ([2]), which handles the next step. In `result_step()`, we first hide the form by calling `siv.pop_layer()` ([3]). This "pops" the existing layer (i.e., the form layer) from the layers stack, and then we add our layer that displays the cat in a `TextView` ([4]). So the flow is:

- `main()`: create Cursive object and calls `input_step()`.
- `input_step()`: Setup the form layout and callbacks.
- `result_step()`: When "OK" is clicked, hide the form and show the cat dialog.

3.7 Reading user input

Now we understand how the program goes from one layer to another, but how does the user's input (the message and the "Dead?" flag) got carried from the form to the cat picture dialog? If we take a closer look at the `input_step()` in Listing 3.5, we can find that the step consists of two parts. First, we set up the input fields:

```
siv.add_layer(
    Dialog::new()
        .title("Please fill out the form for the cat") // setting the title
        .content(
            ListView::new()
                .child("Message:", EditView::new().with_id("message"))
                .child("Dead?", Checkbox::new().with_id("dead")),
        )
```

As before, we created a layer and added a `Dialog` element to it. We set the content of the `Dialog` using `.content()`. Inside the `Dialog` we created two input elements, an `EditView` and a `Checkbox`. In order to place them properly, we wrap them in a `ListView`, which is a layout container that will display its children in a scrollable list. Notice that we called `.with_id()` on the `EditView` and `Checkbox`, this gives each of them an unique ID, which we can use to identify and retrieve them later.

Then we added a button to the `Dialog` like so:

```
Dialog::new()
  .title(...)
  .content(...)
  .button("OK", |s| {
    let message = s
        .call_on_id("message", |t: &mut EditView| t.get_content())
        .unwrap();
    let is_dead = s
```

```
            .call_on_id("dead", |t: &mut Checkbox| t.is_checked())
            .unwrap();
        let options = CatsayOptions {
            message: &message,
            dead: is_dead,
        };
        result_step(s, &options)
    }),
```

In this button's callback, we read the message and the status of the "Dead?" flag, collect them into a `CatsayOptions` struct, then pass the `CatsayOptions` struct to the `result_step()` to display the final output. This is when the IDs come in handy. The first argument (`"message"`) of the `s.call_on_id()` call is the ID we just set. `call_on_id()` will try to find the element and pass its mutable reference into the callback closure (the second argument). Inside the closure we use `t.get_content()` (where `t` is `&mut EditView`) to get the text inside the `EditView`. We might fail to find any element with the given ID, so `call_on_id()` returns an `Option` wrapping the return value of the closure, that's why we have to `unwrap` it to get the actual string. We did the similar for the `Checkbox`. By calling `is_checked()` on `Checkbox` it will return a boolean indicating whether the checkbox is checked or not.

Then we simply wrap the two values into a `CatsayOptions` struct, so we can pass it to the `result_step()` . Inside the `result_step()` (Listing 3.5), we display the cat ASCII-art in a `Dialog` using the options from the previous step.

There are many more callbacks and UI patterns we can talk about. But TUI is relatively limited due to its low resolution. There is very limited space on the screen if you have to render each pixel as a character. Also, it's not very aesthetically pleasing for modern users, sometimes even a little bit intimidating. However, TUI is still very useful if you want to build some small tools that require simple interactions. Also, it might be useful on remote servers where users need to SSH into remotely. Due to these limitations, we are going to conclude our journey in TUI and move on to GUI, the Graphical User Interface.

3.8 Moving to graphical user interfaces (GUIs)

In the next half of this chapter, we'll be building a GUI version of the TUI program we just built. This time we'll be able to actually render a cat photo! For this purpose, we are going to use the `gtk-rs` crate, which is a Rust binding for the GTK library. GTK, originally known as GTK+ and GIMP Toolkit, is a free and open-source widget toolkit for building GUIs. It is written in C and supports multiple platforms like Linux, Microsoft Windows, and macOS. It provides many UI widgets out of the box so we can easily assemble a GUI program. Many popular programs like the GNOME desktop environment uses it.

There are many other GUI toolkits for Rust, but we choose `gtk-rs` for its popularity and maturity. It's one of the most downloaded GUI crates on crates.io, and it is one of

the most mature libraries in the domain because of the maturity of GTK itself. It can potentially support cross-platform development (by installing GTK libraries for the target platform). An additional benefit of using `gtk-rs` is that we have nice documentation and community support from the original GTK library. Because it's a binding around the C-based GTK library, whenever the Rust documentation is not clear, you can always find the C documentation and many discussions online. We'll discuss the other alternatives in section *Other alternatives*.

3.9 Creating a window

First, let's try to create a window with GTK. `gtk-rs` relies on the system GTK library. To install it on Ubuntu[2], simply run

```
sudo apt-get install libgtk-3-dev
```

Then, create a new project with `cargo new` and add the following dependeny to `Cargo.toml`:

```
[dependencies]
gio = "0.6.0" # the underlying GLib bindings

[dependencies.gtk]
version = "0.6.0"  # the gtk crate version
features = ["v3_22"]  # system gtk version
```

This is slightly more complex than the Cargo dependencies we've seen. Because `gtk-rs` relies on the system `gtk` library, it uses the Cargo "feature" to control which version of the system `gtk` library it is targeting. So the `version = "0.6.0"` is specifying the version of the `gtk-rs` crate itself, while the `v3_18` is the version of the system `gtk` library we are using. In case you don't know which version of the system `gtk` library you've installed, you can run `dpkg -l libgtk-3-0` to find out.

We now have done the groundwork, and we are ready to code. Create a `src/main.rs` file and copy Listing 3.6 into it.

Listing 3.6: Opening a GTK window

```
extern crate gio;
extern crate gtk;

use gio::prelude::*;
use gtk::prelude::*;
use gtk::{Application, ApplicationWindow};
use std::env;
```

[2] You can find installation instructions for other platforms here: https://gtk-rs.org/docs-src/requirements.html.

```
fn main() {
    let app = Application::new("com.shinglyu.catsay-gui", gio::
    ↪ ApplicationFlags::empty())
        .expect("Failed to initialize GTK.");

    app.connect_startup(|app| {
        let window = ApplicationWindow::new(app);
        window.set_title("Catsay");
        window.set_default_size(350, 70);

        window.connect_delete_event(|win, _| {
            win.destroy();
            Inhibit(false) // Don't prevent the default behavior
            ↪ (i.e. close)
        });

        window.show_all();
    });

    app.connect_activate(|_| {});
    app.run(&env::args().collect::<Vec<_>>());
}
```

You might notice that the code has a very similar structure to our TUI program. First, we created a GTK "Application" using `Application::new`. We have to set an application ID in the first argument. GTK uses a "reverse DNS" style for ID. So let's say our application will have a public website at https://catsay-gui.shinglyu.com, we should use "com.shinglyu.catsay-gui" for the ID. When the application starts up, the `startup` event is triggered. We should set up the application inside the `startup` event handler. In this case, we create an `ApplicationWindow` and set its title and size. We also register an event handler for the `delete` event, in which we destroy the window (`win.destory()`). This will be triggered when we try to close the window. Finally, we run `window.show_all()` so the window does not remain hidden when the application starts.

After `startup`, our application will receive an `activate` event. This is usually when we show a default new window. For example, a word processor might open a new empty document. But since our application does not require such a step, we simply pass an empty closure that does nothing. Finally, we run `app.run()` to start the event loop. We also pass all the command-line arguments `env::args()` to the application in case we need them later in the application.

■ **NOTE** Notice that we return a weird-looking `Inhibit(false)` in the `delete` event handler. Sometimes we don't want the default event handler to work. For example, when the user tries to close the window, we want to ask for a confirmation

before we close it. We'll have to stop the event from propagating to the default event handler because it will close the window right away. If we return `Inhibit(true)`, GTK will stop propagating the event to the default handler. But in our example, we do want the window to be closed, so we simply return `Inhibit(false)`.

Now if we run `cargo run`, the window will show up like Figure 3.7

Figure 3.7: An empty GTK window

3.10 Displaying an image

Now we can add a few lines to show a cat image in our window by adding the code in Listing 3.7 to our `main()`.

Listing 3.7: Showing text and a cat picture

```
use gtk::{Application, ApplicationWindow, Box, Image, Label,
    ↪ Orientation};
// ...

fn main() {
    // ...

    app.connect_startup(|app| {
        let window = ApplicationWindow::new(app);

        // ... setup window title, size and delete_event

        let layout_box = Box::new(Orientation::Vertical, 0); //
        ↪ [1]

        let label = Label::new("Meow!\n     \\\n      \\");
        layout_box.add(&label);
```

```
        let cat_image = Image::new_from_file("./images/cat.png")
    ↪ ; //[2]
        layout_box.add(&cat_image);

        window.add(&layout_box); //[3]

        window.show_all();
    });

    //...
}
```

We are trying to display a text (`Label`) and an image (`Image`). But to properly display the two together, we wrap them in a `GtkBox`. A `GtkBox` is a container that can display widgets in a single row or column. In [1], we create a box with `Orientation::Vertical`, which will display the widgets from top to down in a vertical column. The second parameter `0` is the spacing between each widget, which we set to none.

Then we create the text in a `Label`, which is similar to `TextView` in our TUI example. We also created an `Image` using `Image::new_from_file("./images/cat.png")`. This creates a GTK image widget that shows a PNG file. We'll create a folder in the same project called `images`, and put the `cat.png` image in there. These two elements are not showing yet,and we have to add them to the container with `layout_box.add(&label)` and `layout_box.add(&cat_image)`, then add the layout box to the window (`window.add(&layout_box)`). The result will look like Figure 3.8.

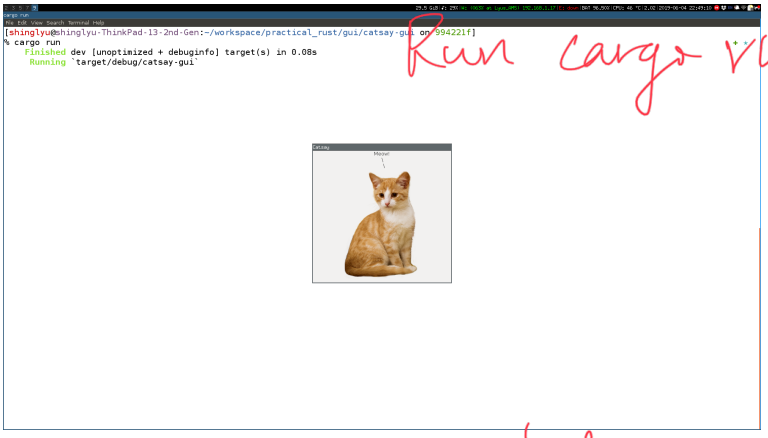


Figure 3.8: Displaying the text and text image in GTK

■ **TIP** We defined the layout of the widgets in Rust code. But sometimes, when the widgets are positioned incorrectly, it's pretty hard to figure out why they went astray just by reading the code. The good news is that GTK provides a visual debugger so we can see the widget tree and have them highlighted in the application window. Simply run your GTK application with the environment variable `GTK_DEBUG` set to `interactive`. For example `GTK_DEBUG=interactive cargo run`. You'll see the program starts along with the debugger window (Figure 3.9). In the "Objects" tab in the debugger, we can see the hierarchy of the widgets. If we click on one of the widgets, that widget will flash in the main application window to show its position and size. In Figure 3.10 we select the `GtkLabel` widget, and you can see it is being highlighted. This can help us debug and tweak the layout of the widgets much easier.

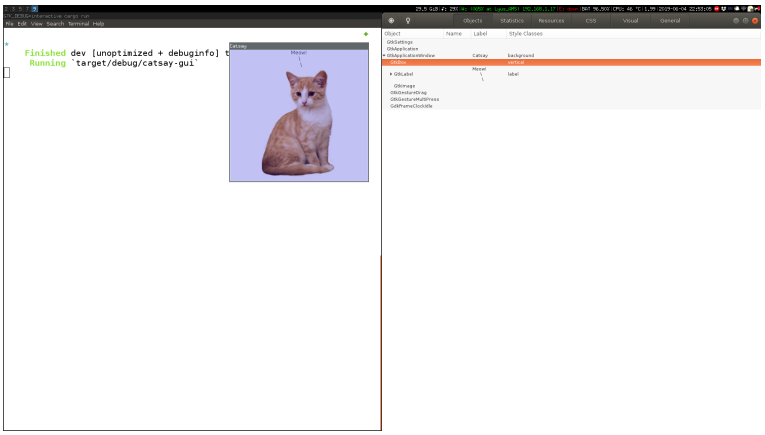


Figure 3.9: GTK debugger that highlights the `GtkBox`

3.11 Using Glade to design the UI

In Listing 3.7, we built the UI procedurally. That means we have to create widgets, put them in containers, put containers into bigger containers, then attach them to the window, and finally display them, all using Rust code. When the program grows larger and larger, this way of working is very error-prone and is hard to visualize. An alternative way is to define the UI layout declaratively. Instead, build the UI like:

- Create a GtkBox
- Create a GtkLabel

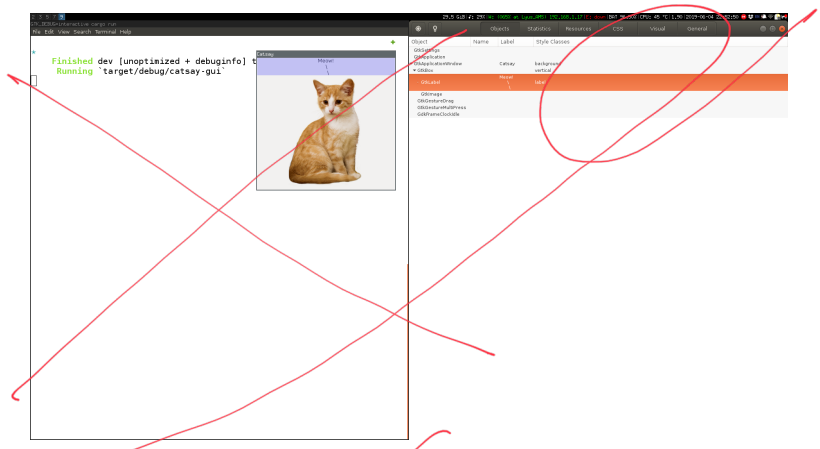


Figure 3.10: GTK debugger that highlights the `GtkLabel`

- Put the GtkLabel in the GtkBox
- Create a GtkImage
- Put the GtkImage in the GtkBox
- Put the GtkBox in the window

we can declare that we want a layout that is:

- window
  - GtkBox
    - GtkLabel
    - GtkImage

GTK provides a way to make the declaration by using an XML (eXtensible Markup Language) markup. The XML file contains the static declaration of the layout of the widgets and can be loaded using the `GTKBuilder` object and build the UI in runtime. If we write our example application in the previous section into the XML format, it will look like Listing 3.8. You can clearly see a hierarchy of a `GtkBox` containing a `GtkLabel` and a `GtkImage`.

Listing 3.8: An example of Glade XML

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <requires lib="gtk+" version="3.12"/>
```

```
  <object class="GtkApplicationWindow" id="applicationwindow1">
    <child>
      <object class="GtkBox" id="box1">
        <property name="orientation">vertical</property>
        <child>
          <object class="GtkLabel" id="label1">
            <property name="label" translatable="yes">Meow!</
    ↪ property>
          </object>
        </child>
        <child>
          <object class="GtkImage" id="image1">
            <property name="pixbuf">./images/cat.png</property>
          </object>
        </child>
      </object>
    </child>
  </object>
</interface>
```

But writing this XML by hand is very tedious. We can use Glade (Figure 3.11), the UI design tool that comes with GTK to generate the XML. You can install Glade with the command: `sudo apt-get install glade`. In Glade, you can drag and drop widgets in a WYSIWYG (What-You-See-Is-What-You-Get) editor. You can also tweak the parameters of individual widgets and get instant feedback. With Glade (and the XML layout definition), we can separate the concern of the visual presentation from the action and behavior. We can keep most of the visual design and layout in the XML and leave only event handlers logic in Rust code.

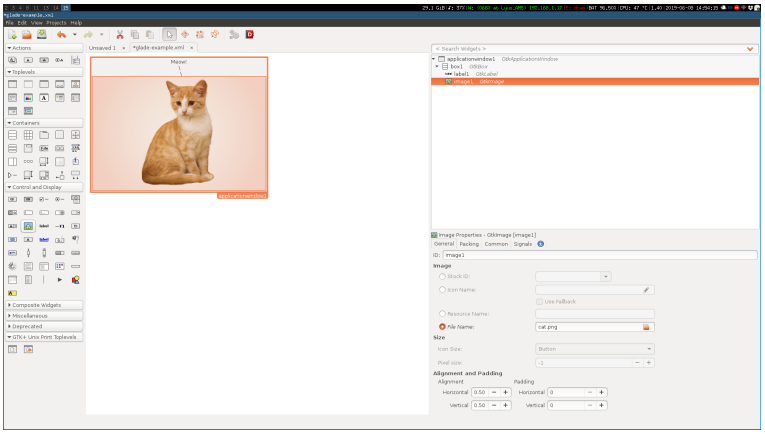


Figure 3.11: Glade UI design tool

So let's build a simple form with Glade as a foundation to demonstrate input events and handlers in the following sections. We can drag and drop a form that looks like Figure 3.12; Its widgets are organized as in Figure 3.13. Then we can click the menu "File", then "Save as..." to save this to an XML file called `layout.glade` (shown in Listing 3.9).

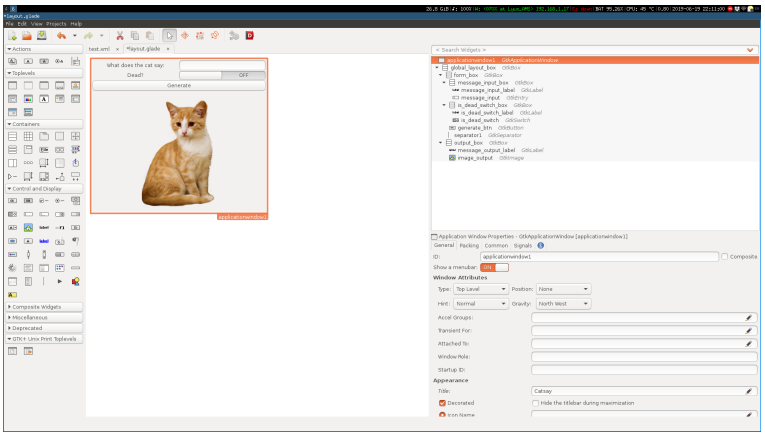


Figure 3.12: Building the form with Glade

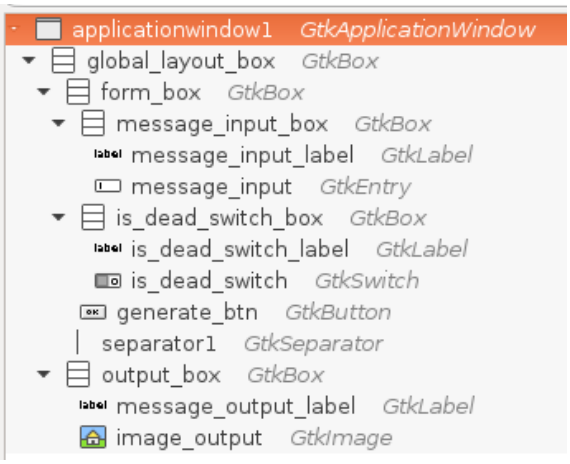


Figure 3.13: Widget hierarchy of the form

Listing 3.9: Glade Layout XML

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Generated with glade 3.18.3 -->
<interface>
  <requires lib="gtk+" version="3.12"/>
  <object class="GtkApplicationWindow" id="applicationwindow1">
    <property name="can_focus">False</property>
    <property name="title" translatable="yes">Catsay</property>
    <child>
      <object class="GtkBox" id="global_layout_box">
        <property name="visible">True</property>
        <property name="can_focus">False</property>
        <property name="orientation">vertical</property>
        <child>
          <object class="GtkBox" id="form_box">
            <property name="visible">True</property>
            <property name="can_focus">False</property>
            <property name="orientation">vertical</property>
            <child>
              <object class="GtkBox" id="message_input_box">
                <property name="visible">True</property>
                <property name="can_focus">False</property>
                <property name="resize_mode">immediate</property
    ↪ >
                <property name="homogeneous">True</property>
                <child>
```

```
                    <object class="GtkLabel" id="message_input_bo"
    ↪ >
                        <property name="visible">True</property>
                        <property name="can_focus">False</property>
                        <property name="label" translatable="yes">
    ↪ What does the cat say:</property>
                    </object>
                    <packing>
                        <property name="expand">False</property>
                        <property name="fill">True</property>
                        <property name="position">0</property>
                    </packing>
                </child>
                <child>
                    <object class="GtkEntry" id="message_input">
                        <property name="visible">True</property>
                        <property name="can_focus">True</property>
                    </object>
                    <packing>
                        <property name="expand">False</property>
                        <property name="fill">True</property>
                        <property name="position">1</property>
                    </packing>
                </child>
            </object>
            <packing>
                <property name="expand">False</property>
                <property name="fill">True</property>
                <property name="position">0</property>
            </packing>
        </child>
        <child>
            <object class="GtkBox" id="is_dead_switch_box">
                <property name="visible">True</property>
                <property name="can_focus">False</property>
                <property name="homogeneous">True</property>
                <child>
                    <object class="GtkLabel" id="
    ↪ is_dead_switch_label">
                        <property name="visible">True</property>
                        <property name="can_focus">False</property>
                        <property name="label" translatable="yes">
    ↪ Dead?</property>
                    </object>
                    <packing>
                        <property name="expand">False</property>
```

```
                        <property name="fill">True</property>
                        <property name="position">0</property>
                    </packing>
                </child>
                <child>
                    <object class="GtkSwitch" id="is_dead_switch">
                        <property name="visible">True</property>
                        <property name="can_focus">True</property>
                    </object>
                    <packing>
                        <property name="expand">False</property>
                        <property name="fill">True</property>
                        <property name="position">1</property>
                    </packing>
                </child>
            </object>
            <packing>
                <property name="expand">False</property>
                <property name="fill">True</property>
                <property name="position">1</property>
            </packing>
        </child>
        <child>
            <object class="GtkButton" id="generate_btn">
                <property name="label" translatable="yes">
    ↪ Generate</property>
                <property name="visible">True</property>
                <property name="can_focus">True</property>
                <property name="receives_default">True</property
    ↪ >
            </object>
            <packing>
                <property name="expand">False</property>
                <property name="fill">True</property>
                <property name="position">2</property>
            </packing>
        </child>
    </object>
    <packing>
        <property name="expand">False</property>
        <property name="fill">True</property>
        <property name="position">0</property>
    </packing>
</child>
<child>
    <object class="GtkSeparator" id="separator1">
```

```
            <property name="visible">True</property>
            <property name="can_focus">False</property>
          </object>
          <packing>
            <property name="expand">False</property>
            <property name="fill">True</property>
            <property name="position">1</property>
          </packing>
        </child>
        <child>
          <object class="GtkBox" id="output_box">
            <property name="visible">True</property>
            <property name="can_focus">False</property>
            <property name="orientation">vertical</property>
            <child>
              <object class="GtkLabel" id="message_output">
                <property name="visible">True</property>
                <property name="can_focus">False</property>
                <property name="ellipsize">end</property>
              </object>
              <packing>
                <property name="expand">False</property>
                <property name="fill">True</property>
                <property name="position">0</property>
              </packing>
            </child>
            <child>
              <object class="GtkImage" id="image_output">
                <property name="can_focus">False</property>
                <property name="pixbuf">cat.png</property>
              </object>
              <packing>
                <property name="expand">False</property>
                <property name="fill">True</property>
                <property name="position">1</property>
              </packing>
            </child>
          </object>
          <packing>
            <property name="expand">False</property>
            <property name="fill">True</property>
            <property name="position">2</property>
          </packing>
        </child>
      </object>
    </child>
```

```
  </object>
</interface>
```

To load this XML into an actual GTK program, let's create a new Rust project with `cargo new`, and copy the `layout.glade` into the `src` folder. Then we can open main.rs and fill in Listing 3.10. If we run the program now, we'll see Figure 3.14.

Listing 3.10: Loading the Glade XML file with `GtkBuilder`

```
extern crate gio;
extern crate gtk;

use gio::prelude::*;
use gtk::prelude::*;

use std::env::args;

fn build_ui(app: &gtk::Application) {
    let glade_src = include_str!("layout.glade");
    let builder = gtk::Builder::new_from_string(glade_src);

    let window: gtk::Window = builder.get_object("
    ↪ applicationwindow1").unwrap();
    window.set_application(app);

    window.show_all();
}

fn main() {
    let application = gtk::Application::new("com.shinglyu.catsay
    ↪ -gui-glade", Default::default())
        .expect("Failed to initialize GTK");

    application.connect_activate(|app| {
        build_ui(app);
    });

    application.run(&args().collect::<Vec<_>>());
}
```

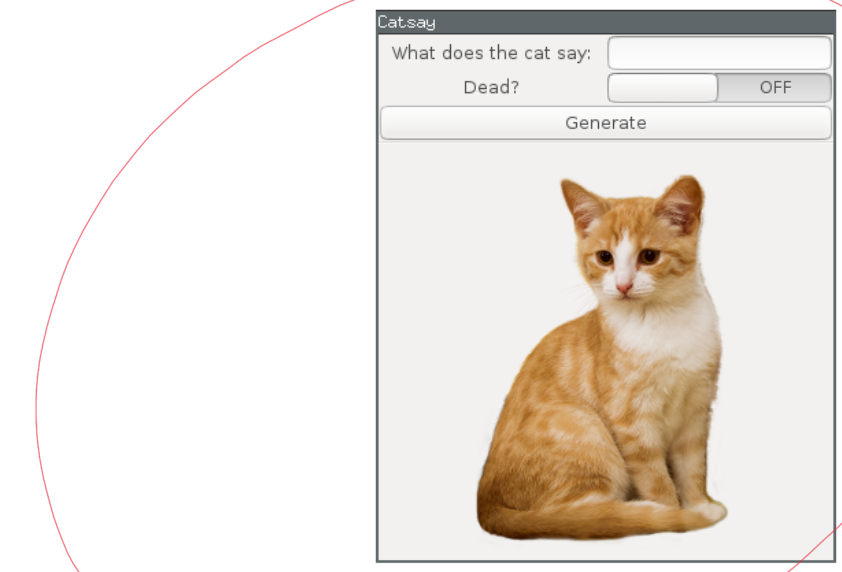


Figure 3.14: Result of Listing 3.10

Notice how the `connect_active` handler in Listing 3.10 is much more simpler compared to Listing 3.7. We no longer need to build up the hierarchy inside Rust code. Instead, we load the Glade XML file using `include_str!` in the `build_ui()` function. The built-in macro `include_str!()` will load a file into a string variable `glade_src`. We then use `gtk::Builder::new_from_string(glade_src)` to build the GTK program using the Glade XML definition string. However, because we are not building the widgets in Rust code, we don't have any Rust variable that points to the individual widgets, so we can't call functions like `window.show_all()` because `window` is not there. We can identify the widgets by their IDs inside the application built by the builder. Inside Listing 3.9 the `ApplicationWindow` has an `id="applicationwindow1"` which was auto-generated by Glade. We can use `builder.get_object("applicationwindow1")` to get the widget. Because we might provide an ID that doesn't exist, the function returns an `Option<T>`, so please remember to `unwrap` it or handle the error cases properly. The window created by the builder also doesn't know which `gtk::Application` it belongs to. We have to use `window.set_application(app)` to associate the application with the window. That's why we pass the `gtk::Application` created in the `main` function into the `build_ui()` function, so that the `window` knows which `Application` it should associate to.

3.12 Accepting inputs and button clicks

We can add interactivity to the GTK application in a similar way we did for the TUI application. Let's add some event handlers to the inputs and buttons in the `build_ui()` function (Listing 3.11).

Listing 3.11: Event handlers for the GTK application

```
fn build_ui(app: &gtk::Application) {
    let glade_src = include_str!("layout.glade");
    let builder = gtk::Builder::new_from_string(glade_src);

    let window: gtk::Window = builder.get_object("
    ↪ applicationwindow1").unwrap();
    window.set_application(app);

    // Inputs
    let message_input: gtk::Entry = builder.get_object("
    ↪ message_input").unwrap();

    // Submit button
    let button: gtk::Button = builder.get_object("generate_btn")
    ↪ .unwrap();

    // Outputs
```

```
    let message_output: gtk::Label = builder.get_object("
    ↪ message_output").unwrap();
    let image_output: gtk::Image = builder.get_object("
    ↪ image_output").unwrap();
    let image_output_clone = image_output.clone();

    button.connect_clicked(move |_| {
        message_output.set_text(&format!(
            "{}\n\\\n\\",
            message_input.get_text().unwrap().as_str()
        ));
        image_output_clone.show();
    });

    window.show_all();
    image_output.hide();
}
```

First, we get the handles for all the widgets we need using the `builder.get_object()` function, just like we did for getting the `ApplicationWindow`:

```
let message_input: gtk::Entry = builder.get_object("message_input").unwrap()
let button: gtk::Button = builder.get_object("generate_btn").unwrap();
let message_output: gtk::Label = builder.get_object("message_output").unwrap
let image_output: gtk::Image = builder.get_object("image_output").unwrap();
```

Then we want the cat image to remain hidden until we click the "Generate" button. So we call `image_output.hide()` right after `window.show_all()`. The order is important here because we first show everything then hide the image. If we hide the image first, then call `widow.show_all()`, the image will be shown again.

We then have to create a callback function on the "Generate" button to show the cat and the text message. We call `button.connect_clicked()` to set the callback for button's `clicked` event:

```
button.connect_clicked(|_| {
    message_output.set_text(&format!(
        "{}\n\\\n\\",
        message_input.get_text().unwrap().as_str()
    ));
    image_output.show();
});
```

The callback is a closure, in which we do three things:

- Read the input from `message_input.get_text()`. (`get_text()` returns a `glib::GString`, so we convert it to `&str` using `.as_str()`.)

- Set the `message_output` text using the text in `message_input`.
- Show the image with `image_output.show()`.

Although the structure of the code looks OK, the code won't compile. We'll receive the following error when we try to compile:

```
error[E0373]: closure may outlive the current function, but it borrows 'image_ou
  --> src/main.rs:36:28
   |
36 |     button.connect_clicked(|_| {
   |                            ^^^ may outlive borrowed value 'image_output'
...
41 |         image_output.show();
   |         ------------ 'image_output' is borrowed here
   |
note: function requires argument type to outlive ''static'
  --> src/main.rs:36:5
   |
36 | /     button.connect_clicked(|_| {
37 | |         message_output.set_text(&format!(
38 | |             "{}\n     \\\n      \\",
39 | |             message_input.get_text().unwrap().as_str()
40 | |         ));
41 | |         image_output.show();
42 | |     });
   | |_____^
help: to force the closure to take ownership of 'image_output' (and any other re
   |
36 |     button.connect_clicked(move |_| {
   |                            ^^^^^^^^
```

This is because once the callback function is set, it might get triggered anytime during the application's lifetime. But by the time the callback is triggered, the `build_ui()` function is probably already finished and the `image_output` variable gone out of scope. To mitigate this, we have to `move` the ownership of the variable to the closure, so the closure can keep it alive. But if we simply add a `move` keyword to the closure, the `image_output` variable won't be accessible after we move it into the closure, because the ownership has already moved to the closure. For example:

```
let image_output: gtk::Image = builder.get_object("image_output").unwrap();

button.connect_clicked(move |_| {
    // ...
```

```
    image_output.show();
});

image_output.hide(); // This will fail!
// error[E0382]: borrow of moved value: `image_output`
```

However, because `Gtk-rs` is a wrapper around the C GTK library, doing a Rust `clone` on an `Gtk-rs` object only copies the pointer. So since it's not a costly deep clone of the whole data structure, we can simply clone the handle and move it into the closure[3]:

```
let image_output: gtk::Image = builder.get_object("image_output").unwrap();
let image_output_clone = image_output.clone(); // low-
cost clone

button.connect_clicked(move |_| {
    message_output.set_text(&format!(
        "{}\n      \\\n      \\",
        message_input.get_text().unwrap().as_str()
    ));
    image_output_clone.show(); // move the clone into the closure
});

image_output.hide(); // we still keep the ownership of it
```

3.13 Reading a `gtk::Switch`

There is only one thing left in our Glade design, the "Dead?" switch. The code change (Listing 3.12) is pretty straightforward.

Listing 3.12: Handling input from `gtk::Switch`

```
fn build_ui(app: &gtk::Application) {
    // ...
    let is_dead_switch: gtk::Switch = builder.get_object("
    ↪ is_dead_switch").unwrap();

    let image_output: gtk::Image = builder.get_object("
    ↪ image_output").unwrap();
    let image_output_clone = image_output.clone();

    button.connect_clicked(move |_| {
```

[3] We are not cloning the `message_input` and `message_output` simply because we don't need to use them after we define the callback function. But if we need to use them after moving them into the callback, we should clone them just like we did for `image_output`.

```
        //...
        let is_dead = is_dead_switch.get_active();
        if (is_dead) {
            image_output_clone.set_from_file("./images/cat_dead.
    ↪ png")
        } else {
            image_output_clone.set_from_file("./images/cat.png")
        }
        image_output_clone.show();
    });

    window.show_all();
    image_output.hide();
}
```

As before, we get the handle of the switch by:

```
let is_dead_switch: gtk::Switch = builder.get_object("is_dead_switch").unwra
```

Then we can check if it's activated or not by reading `is_dead_switch.get_active()`. Based on the whether it's `true` or `false` we can load different cat images using `image_output_clone.set_from_file("path/to/file.png")`. This allows us to change the image in runtime.

Finally, our end product will look like Figure 3.15 to Figure 3.17.

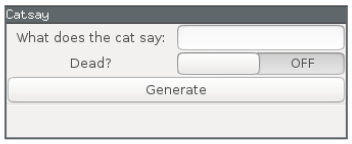


Figure 3.15: The form

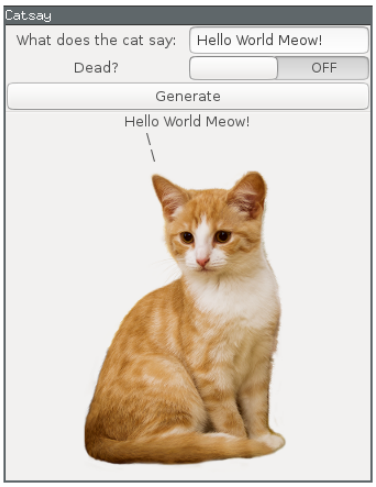


Figure 3.16: After clicking "Generate" with "Dead?" off

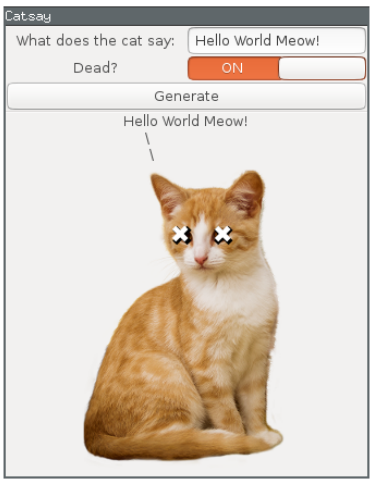


Figure 3.17: After clicking "Generate" with "Dead?" on

3.14 Other alternatives

This concludes our journey of building a text-based user interface (TUI) to a graphical user interface (GUI). We chose to use `ncurses`-based TUI library and GTK-based GUI library because they are more mature and stable. And their corresponding Rust library is also more production-ready. However, there are many exciting new Rust libraries out there that provide a more idiomatic Rust interface or provide better cross-platform support.

On the TUI side, `tui-rs` is also a good alternative to Cursive, but it doesn't support input handling out of the box, so it's better suited for building applications that don't require user interaction, like a monitoring dashboard. There are also some pure-rust alternatives, like `termion` (part of ReduxOS) and `crossterm`. If you are looking for cross-system support, including Linux and Window, `pancurses` is an abstraction layer above the platform-specific layers. It uses `ncurses-rs` for Linux and `pdcurses-sys` for Window underneath.

On the GUI side, there are alternatives like `Relm`, which is also GTK based, but with an Elm-inspired API. Another very popular GUI library is Qt, there are Rust bindings for it like `rust-qt` and `qmlrs` and `rute`. There is also `conrod`, which is written entirely in Rust and is part of the Piston game engine. If you are familiar with Flutter, Google's cross-device UI toolkit, there is a Rust binding for it, called `flutter-rs`. Other existing GUI libraries like IUP and Dear ImGui also have their Rust bindings (`iup-rust` and `imgui-rs` respectively).

There are many TUIs and GUIs, depending on which platform you are targeting and which existing library you are familiar with, there is always something to fulfill your need in building a user interface in Rust.