Student ID: _____ Name: _____ TA: _____

Rust Lab 10 — GUI in Rust (Iced)

10/9/2025

Goal: Build a basic calculator desktop app using the Iced GUI framework in Rust. You will design the state and message flow, assemble the UI, and implement the update logic yourself. This handout intentionally contains no solution code.

Learning Outcomes

- Apply the Iced Application architecture: State → View → Message → Update.
- Design minimal, testable state for a GUI app (display text, stored operand, pending operation, flags).
- Lay out a grid of widgets, align text, and wire user interactions to messages.
- Handle edge cases (e.g., chaining operations, divide-by-zero) and define acceptance tests.

Prerequisites

- Completed reading the Iced Beginner Handout.
- Rust toolchain installed (rustup + cargo).
- Comfort with building and running a Cargo project.

What You Must Deliver

- A compiling Rust project that runs a working calculator GUI.
- A short README.md explaining your state design and message flow (max 1 page).
- Screenshots demonstrating all required behaviors (see Test Checklist).

Functional Requirements

1. Buttons: digits 0–9, operators + − × ÷, =, C (clear), Exit.
2. Display: shows the current number; right-aligned, large font.
3. Number entry: digits append to the current number; after operator or '=', the next digit starts a new number.
4. Operators: pressing +/−/×/÷ stores or computes and then remembers the new operator (chaining).
5. Equals: computes the result using the stored left operand, current display, and pending op.
6. Clear: resets the calculator to an initial state (display '0', no pending op).
7. Exit: closes the program (simple behavior acceptable for this lab).
8. Divide-by-zero behavior must be defined (you choose: e.g., 'Infinity', error message, or disabled '=').

UI Layout Specification

Use a grid layout with uniform square buttons. The display row is right-aligned.

Reference layout (you must implement visually similar behavior):

```
[Display.....]
[ 7 ] [ 8 ] [ 9 ] [ ÷ ]
[ 4 ] [ 5 ] [ 6 ] [ × ]
[ 1 ] [ 2 ] [ 3 ] [ − ]
[ 0 ] [ C ] [ = ] [ + ]
[    Exit (X)     ]
```

Technical Constraints

- Iced version: 0.12 (desktop).
- Use `iced` only; no other GUI crates.
- Write your own implementation.
- Follow the Application trait pattern (`new`, `update`, `view`, etc.).

Test Checklist (attach screenshots)

Input Sequence	Expected Display	Notes / Screenshot
2 + 3 =	5	
5 + 3 × 2 =	16	Chaining rule
9 ÷ 4 =	2.25 or policy	Define your rounding/policy
7 ÷ 0 =	Your chosen behavior	No crash

Student ID: _____ Name: _____ TA: _____

0 0 3	3	No leading zeros
1 2 C 4 =	4	Clear resets correctly
3 + = = =	9 (if repeat equals) or your policy	Document behavior
999999 + 1 =	1000000	Large numbers
Exit	App closes	

Hint:

```
use iced::alignment::Horizontal;
use iced::widget::{button, column, container, row, text};
use iced::{Application, Command, Element, Length, Settings, Theme};

pub fn main() -> iced::Result {
    Calc::run(Settings::default())
}

/* ------------------------------------------------------------------------
   STUDENT TODOs (a little more hint, no answers):
   1) Design/finish the update rules for SetOp and Equals.
   2) Decide chaining policy: when user presses an op twice, or 5 + 3 × 2 = .
   3) Decide divide-by-zero behavior (message? Infinity? block '=').
   This hint intentionally DOES NOT contain the compute logic.
   ------------------------------------------------------------------------ */

struct Calc {
    display: String,         // shows the current entry/result, starts at "0"
    first: Option<f64>,      // LHS stored when an operator is chosen
    op: Option<Op>,          // pending operation (+ - * /)
    entering_new: bool,      // true when the next digit should start a new number
}

#[derive(Debug, Clone, Copy)]
enum Op { Add, Sub, Mul, Div }

#[derive(Debug, Clone)]
enum Message {
    Digit(u8),     // 0..=9
    SetOp(Op),     // + - * /
    Equals,        // =
    Clear,         // C
    Exit,          // X
}

impl Default for Calc {
    fn default() -> Self {
        Self {
            display: "0".into(),
            first: None,
            op: None,
            entering_new: false,
        }
    }
}

impl Application for Calc {
```

Student ID: _____ Name: _____ TA: _____

```
    type Executor = iced::executor::Default;
    type Message = Message;
    type Theme = Theme;
    type Flags = ();

    fn new(_: Self::Flags) -> (Self, Command<Self::Message>) {
        (Self::default(), Command::none())
    }

    fn title(&self) -> String { "Iced — Calculator (Starter+)".into() }

    fn update(&mut self, msg: Self::Message) -> Command<Self::Message> {
        match msg {
            Message::Digit(n) => {
                // ✅ HINT IMPLEMENTED: number entry
                // If we're starting a fresh number or currently "0", replace.
                // Otherwise append.
                if self.entering_new || self.display == "0" {
                    self.display = n.to_string();
                    self.entering_new = false;
                } else {
                    self.display.push(char::from(b'0' + n));
                }
            }

            Message::SetOp(new_op) => {
                let current = parse_f64(&self.display);
                match (self.first, self.op) {
                    (None, _) => {
                        // ✅ First op after typing a number: store LHS
                        self.first = Some(current);
                    }
                    (Some(_lhs), Some(_op)) if !self.entering_new => {
                        // 🔷 TODO (student): CHAINING
                        // Compute: result = apply(_op, _lhs, current)
                        // self.first = Some(result)
                        // self.display = pretty(result)
                    }
                    _ => {
                        // Pressed operator twice? You choose a policy:
                        // - Replace pending op
                        // - Or ignore second press
                    }
                }
                // Set/replace the pending operator and start new entry
                self.op = Some(new_op);
                self.entering_new = true;
            }

            Message::Equals => {
                // 🔷 TODO (student): EQUALS
                // If we have Some(lhs) and Some(op):
                //   rhs = parse(display)
                //   result = apply(op, lhs, rhs)
                //   display = pretty(result)
                //   first = Some(result)  // (optional, for repeat '=' policy)
                //   op = None
                //   entering_new = true
```

Student ID: _____ Name: _____ TA: _____

```
            }

            Message::Clear => {
                // ✅ Reset to initial state
                *self = Self::default();
                self.display = "0".into();
            }

            Message::Exit => {
                // ✅ Simple close for the lab
                std::process::exit(0);
            }
        }
        Command::none()
    }

    fn view(&self) -> Element<Self::Message> {
        // Layout constants (avoid stretching to full window)
        const BTN: f32 = 64.0;
        const GAP: f32 = 8.0;
        const GRID_W: f32 = BTN * 4.0 + GAP * 3.0;

        // Uniform button helpers
        let btn = |label: &str, msg: Message| {
            button(text(label))
                .on_press(msg)
                .width(Length::Fixed(BTN))
                .height(Length::Fixed(BTN))
        };
        let digit = |n: u8| btn(&n.to_string(), Message::Digit(n));
        let op = |sym: &str, o: Op| btn(sym, Message::SetOp(o));

        // Display: fixed width, right aligned
        let display = container(text(&self.display).size(36))
            .width(Length::Fixed(GRID_W))
            .padding([8, 12])
            .align_x(Horizontal::Right);

        // Rows
        let r1 = row![ digit(7), digit(8), digit(9), op("/", Op::Div) ]
            .spacing(GAP).width(Length::Fixed(GRID_W));
        let r2 = row![ digit(4), digit(5), digit(6), op("*", Op::Mul) ]
            .spacing(GAP).width(Length::Fixed(GRID_W));
        let r3 = row![ digit(1), digit(2), digit(3), op("-", Op::Sub) ]
            .spacing(GAP).width(Length::Fixed(GRID_W));
        let r4 = row![ digit(0), btn("C", Message::Clear), btn("=", Message::Equals), op("+", Op::Add) ]
            .spacing(GAP).width(Length::Fixed(GRID_W));

        // Exit matches grid width (not Fill)
        let exit = button(text("Exit (X)"))
            .on_press(Message::Exit)
            .width(Length::Fixed(GRID_W))
            .height(Length::Fixed(48.0));

        column![display, r1, r2, r3, r4, exit]
            .spacing(GAP)
            .padding(12)
            .into()
```

Student ID: _____ Name: _____ TA: _____

```
    }
}

/* ------------------------- Small utilities ------------------------- */

fn parse_f64(s: &str) -> f64 {
    // HINT: safe parse; extend later for decimals
    s.parse::<f64>().unwrap_or(0.0)
}

// You will write this:
// fn apply(op: Op, a: f64, b: f64) -> f64 {
//     match op {
//         Op::Add => /* ... */,
//         Op::Sub => /* ... */,
//         Op::Mul => /* ... */,
//         Op::Div => /* choose divide-by-zero policy */,
//     }
// }
```