Contents

1 Introduction

Work In Progress. The competition hasn't even started yet.

This is the source code of my entry to Ludum Dare 45, which took place on the weekend of 4-6. October, 2019. Ludum Dare is a game jam, where competitors attempt to create a game according to a theme over the course of a weekend.

Long ago, the rules stated that the game needed to be written completely from scratch during the competition time. Due to the rise of freely-available game engines and the inherent difficulties in defining what "from scratch" actually means, they have considerably relaxed this requirement, and now allow most any preexisting source code to be used, while art assets must still be created inside the time limit.

A few days before the competition officially began, I started writing the framework of getting Rust and SDL talking to each other properly. At the point when the theme was announced, it was successfully opening a window, responding to events, and drawing solid-colored rectangles to the screen.

This document is written with noweb, a literate programming tool. In addition to producing this documentation, it extracts the source code from here and reassembles it into the form that the compiler expects. In this way, the code in the document and the code that actually runs the program are kept in sync. There are many hyperlinked annotations in and around the code blocks to aid navigation around the program text, and hopefully this report is organized such that reading through it from beginning to end will present everything in a logical order.

2 Program Organization

The implementation is divided into three files: *lib, main, & test.* These are each compiled into separate crates.

Lib.rs contains the bulk of the implementation, and is divided into several modules. It gets compiled into an rlib file that can be linked with other source files to make a complete program.

2a ⟨*lib.rs* 2a⟩≡

```
pub mod sdl  { ⟨SDL: Definitions 3c⟩ }
pub mod main { ⟨MAIN: Definitions 3b⟩ }
```

Test.rs contains any automated testing code that I elect to write (currently none). These test must pass before the main program is built.

2b ⟨*test.rs* 2b⟩≡

Main.rs is the primary entry point for the program. In this case, it is simply dispatching to a run function in *lib*. elect to write (currently none). These test must pass before the main program is built.

3a ⟨*main.rs* 3a⟩≡

```
include!("import.inc"); use std::*;
fn main()->Result<(), Box<error::Error>> { ld45::main::run() }
```

The main module contains the high-level logic for running the game. Its only export is the `run` function which contains both the initialization and main game loop code.

3b ⟨*MAIN: Definitions* 3b⟩≡ (2a)

```
use std::error::*;
use crate::sdl::*;

pub fn run()->Result<(), Box<Error>>
{   ⟨MAIN: Initialization 6c⟩
    'mainloop: loop
    {   ⟨MAIN: Process Events 11d⟩
        ⟨MAIN: Draw Frame 14d⟩
    }
    Ok(())
}
```

The SDL module is responsible for communicating with `libsdl2`, which is ultimately responsible for the graphics, sound, and input processing. There is a pre-existing rust crate for this, but I have elected to write my own.

3c ⟨*SDL: Definitions* 3c⟩≡ (2a) 4c▷

```
use std::ffi::*;
use std::os::raw::*;
use std::sync::*;
use std::*;
use std::error::Error;
use std::marker::*;

⟨SDL: C Types 4b⟩
#[link(name="SDL2")] extern { ⟨SDL: C Functions 4a⟩ }
```

3 SDL Error Handling

The first order of business is to be able to detect and respond to SDL errors before they become segmentation faults. SDL provides one function for this:

4a ⟨*SDL: C Functions* 4a⟩≡ (3c) 6a▷

```
fn SDL_GetError() -> *const c_char;
```

Like many C libraries, SDL often uses a sentinel value (such as a null pointer) to indicate a fault has occurred, and other values are useful results. The `IsErr` trait is designed to handle this situation; it has a single method that will distinguish erroneous values from legitimate ones.

`ErrCheck` is a transparent wrapper around any type that implements `IsErr`. Its `unwrap` method transforms SDL's error strategy into Rust's `Result` types, which can then be handled the same way as any other Rust error. `ErrCheck` is private to encourage implementations to turn it into an `SdlError` as soon as possible, given that the description of an SDL error is lost as soon as another error occurs.

4b ⟨*SDL: C Types* 4b⟩≡ (3c) 5▷

```
// Error Handling Types
trait IsErr { fn is_err(&self)->bool; }
#[derive(Debug)] #[repr(transparent)] struct ErrCheck<T:IsErr>(T);
impl<T:IsErr> ErrCheck<T>
{   fn unwrap(self)->Result<T,SdlError>
    {   if self.0.is_err() { Err(SdlError::fetch()) }
        else               { Ok(self.0)             } } }
```

The `SdlError` class holds its own copy of SDL's error description string, so it will retain its identity even if anothe SDL error occurs. The only way to create one is via `fetch()`, which will take the information about the most recent error from SDL.

4c ⟨*SDL: Definitions* 3c⟩+≡ (2a) ◁3c 6b▷

```
#[derive(Debug, Clone)] pub struct SdlError( String );
impl SdlError
{   fn fetch()->SdlError
    {   SdlError( unsafe { CStr::from_ptr(SDL_GetError())
                           .to_string_lossy()
                           .into_owned()                } ) } }

impl Error for SdlError {}

impl fmt::Display for SdlError
{   fn fmt(&self, f: &mut fmt::Formatter<'_>)
    -> fmt::Result { write!(f, "{:?}", self) } }
```


Often, an SDL function will return a negative integer to indicate that an error has occurred, and other values represent a successful result. This logic is encapsulated by implementing `IsErr for c_int`.

5 ⟨*SDL: C Types* 4b⟩+≡ (3c) ◁4b 8b▷

```
impl IsErr for c_int { fn is_err(&self)->bool { *self > 0 } }
```

4 *SDL Initialization*

Before doing anything, SDL needs to be initialized using the `SetMainReady` and `Init` functions. It also may hold resources that won't be cleaned up efficiently by the operating system when the process exits, and so expects `Quit` to be explicitly called before program exit.

6a ⟨*SDL: C Functions* 4a⟩+≡ (3c) ◁ 4a 8a ▷

```
// Initialization and cleanup
fn SDL_SetMainReady();
fn SDL_Init(flags: u32)->ErrCheck<c_int>;
fn SDL_Quit();
```

As Rust uses the Resource Acquisition Is Initialization (RAII) pattern extensively, the most natural way to accomplish these requirements is to have a context object that calls `Quit` when it is dropped.

6b ⟨*SDL: Definitions* 3c⟩+≡ (2a) ◁ 4c 10e ▷

```
#[derive(Debug)] pub struct SDL { guard: MutexGuard<'static, ()> }
impl SDL { ⟨SDL: SDL Methods 6d⟩ }
impl Drop for SDL
{ fn drop(&mut self) { unsafe { SDL_Quit(); } } }
```

This allows the program to perform all of the necessary startup and cleanup tasks with a single line of code:

6c ⟨*MAIN: Initialization* 6c⟩≡ (3b) 9 ▷

```
let sdl = SDL::init()?;
```

SDL generally expects all calls to be made from the same thread and for there to only ever be one SDL instance initialized at a time. We therefore set up a static mutex to ensure that only one SDL object is ever alive at one time.

6d ⟨*SDL: SDL Methods* 6d⟩≡ (6b) 8c ▷

```
pub fn init()->Result<SDL, Box<Error>>
{   let guard = ⟨SDL: Enforce Singleton 7⟩;
    let flags = 0x20; //SDL_VIDEO
    unsafe { SDL_SetMainReady();
             SDL_Init(flags).unwrap()?; }
    Ok(SDL { guard })
}
```


Because static initializers for mutexs are still unstable, the code necessary to set up the mutex is slightly involved. The static variable holding the mutex must be mutable, which forces an `unsafe` block, and we need to use `sync::Once` to ensure that only one mutex is ever created.

7 ⟨*SDL: Enforce Singleton 7*⟩≡ (6d)

```
unsafe
{   static mut MUTEX        : Option<Mutex<()>> = None;
    static     MUTEX_EXISTS : Once              = Once::new();

    MUTEX_EXISTS.call_once(|| { MUTEX = Some(Mutex::new(())); });
    MUTEX.as_ref().unwrap()
}.try_lock()?
```

5 *Window Creation*

Creating a window that we can draw on is fairly straightforward, except for the matter of determining appropriate flags. I'll defer that investigation until later.

8a ⟨*SDL: C Functions* 4a⟩+≡ (3c) ◁6a 10a▷

```
fn SDL_CreateWindow<'sdl>
( title: *const c_char,
  x: c_int, y: c_int,
  w: c_int, h: c_int,
  flags: u32
) -> ErrCheck<Window<'sdl>>;

fn SDL_DestroyWindow(win: WindowHandle);
```

Like the SDL context itself, it makes sense to use RAII to automatically destroy any window that Rust no longer has a handle to. `CreateWindow` might also fail ane return a null pointer, so we define `IsErr` to handle that case.

8b ⟨*SDL: C Types* 4b⟩+≡ (3c) ◁5 10b▷

```
#[derive(Debug, Copy, Clone)]
#[repr(transparent)]     struct WindowHandle(*const c_void);

#[derive(Debug)]
#[repr(transparent)] pub struct Window<'sdl>(WindowHandle,
                                             PhantomData<&'sdl SDL>);

impl<'sdl> IsErr for Window<'sdl> { fn is_err(&self)->bool {
    (self.0).0.is_null() }}

impl<'sdl> Drop for Window<'sdl> { fn drop(&mut self) {
    unsafe { SDL_DestroyWindow(self.0); }}}

impl<'sdl> Window<'sdl> { ⟨SDL: Window Methods 13c⟩ }
```

We expose a safe wrapper for the native function:

8c ⟨*SDL:* SDL *Methods* 6d⟩+≡ (6b) ◁6d 11c▷

```
pub fn create_win<'sdl>(&'sdl self, title:&str, pos:(i32,i32), size:(i32,i32))
-> Result<Window<'sdl>, Box<Error>> { unsafe {
    let c_title = CString::new(title)?;
    Ok(SDL_CreateWindow(c_title.as_ptr(),
                        pos.0,  pos.1,
                        size.0, size.1,
                        0                ).unwrap()?) }}
```


And the actual initialization is a one-line call again:

9 ⟨*MAIN: Initialization* 6c⟩+≡ (3b) ◁6c 13d▷

```
let win = sdl.create_win("Hello", (0, 0), (320, 240))?;
```


6 *Event Polling*

Input is handled through SDL's event system. There are two functions here that we're interested in right now. `PumpEvents` tells SDL to check all of the various devices for new input, and enqueues events for anything that's happened. `PollEvent` takes the first event off of the event queue and returns it, or an indication that the queue is empty.

10a ⟨*SDL: C Functions* 4a⟩+≡ (3c) ◁8a 13a▷

```
fn SDL_PumpEvents();
fn SDL_PollEvent(ev:*mut SdlEventUnion)->c_int;
fn SDL_Delay(ms:u32);
```

Events in SDL are represented by a tagged union. The first field is an integer specifying the type of the event, and the rest of the space has different layouts depending on what that field says.

10b ⟨*SDL: C Types* 4b⟩+≡ (3c) ◁8b 10c▷

```
#[repr(C)]
union SdlEventUnion { event_type: u32,
                      bytes: [u8;56],
                      ⟨SDL: SdlEventUnion fields 10d⟩ }
```

In addition to the event type, all of the structures have a common preamble, so we can use that in the case we have an unknown event occur.

10c ⟨*SDL: C Types* 4b⟩+≡ (3c) ◁10b 13b▷

```
#[repr(C)] #[derive(Debug,Copy,Clone)]
pub struct SdlCommonEvent { pub event_type: u32,
                            pub timestamp:  u32 }
```

10d ⟨*SDL: SdlEventUnion fields* 10d⟩≡ (10b)

```
common: SdlCommonEvent,
```

Rust's equivalent to a tagged union is an enum with data-containing variants. We define `SdlEvent` to represent all of the possible event types, with one variant for each type id.

10e ⟨*SDL: Definitions* 3c⟩+≡ (2a) ◁6b 11a▷

```
#[derive(Debug, Copy, Clone)]
pub enum SdlEvent { Other(SdlCommonEvent),
                    ⟨SDL: SdlEvent Variants 12a⟩ }
```

To facilitate conversions from the C union into the Rust enum, we define the `From` trait:

11a ⟨*SDL: Definitions* 3c⟩+≡ (2a) ◁10e 11b▷

```
impl From<SdlEventUnion> for SdlEvent
{   fn from(raw: SdlEventUnion) -> SdlEvent { unsafe
    {   match raw.event_type { ⟨SDL: SdlEvent Dispatch Table 12b⟩
                               _ => SdlEvent::Other(raw.common) }
    }}
}
```

Usually, you'll want to process all pending events in an event loop embedded inside the main game loop. The `iter_events` method wraps `SDL_PollEvents` to enable its use in a Rust `for` loop control statement.

11b ⟨*SDL: Definitions* 3c⟩+≡ (2a) ◁11a 14b▷

```
pub struct SdlPendingEventsIter<'a> { _sdl: &'a SDL }
impl<'a> Iterator for SdlPendingEventsIter<'a>
{   type Item = SdlEvent;
    fn next(&mut self) -> Option<SdlEvent>
    {   let mut result = SdlEventUnion { event_type: 0 };
        let code = unsafe
        { SDL_PollEvent(&mut result as *mut SdlEventUnion) };
        if code > 0 { Some(SdlEvent::from(result)) }
        else        { None                         } } }
```

11c ⟨*SDL:* SDL *Methods* 6d⟩+≡ (6b) ◁8c

```
pub fn iter_events<'a>(&'a self) -> SdlPendingEventsIter<'a>
{   unsafe { SDL_PumpEvents() };
    SdlPendingEventsIter { _sdl: self } }
pub fn delay(&self, ms:u32) { unsafe { SDL_Delay(ms); }; }
```

11d ⟨*MAIN: Process Events* 11d⟩≡ (3b)

```
for e in sdl.iter_events() {
    match e { ⟨MAIN: Event Handlers 12c⟩
              event@_ => { println!("Unhandled event: {:?}",
                                    event);                 } } }
```

7 *Quit Event*

The first and most basic event that we want to handle is `SDL_QUIT`. It is fired when the user attempts to close the window or otherwise use operating system facilities to ask the program to stop. It has no fields other than what's defined in the common preamble, so we just need to make an `SdlEvent` variant for it and add its id to the dispatch table.

12a ⟨*SDL:* `SdlEvent` *Variants* 12a⟩≡ (10e)

```
Quit(SdlCommonEvent),
```

12b ⟨*SDL:* `SdlEvent` *Dispatch Table* 12b⟩≡ (11a)

```
0x100 => SdlEvent::Quit(raw.common),
```

As far as handling it in the main program, the simplest choice is to simply break the game loop and let `main` return. In a more complicated game/program, you would likely want to ask for confirmation or attempt to save progress before exiting.

12c ⟨*MAIN: Event Handlers* 12c⟩≡ (11d)

```
SdlEvent::Quit(_) => { break 'mainloop; }
```


8 Getting a Renderer

We'll be using the accelerated 2D renering facilities from SDL, which are accessed via an `SDL_Renderer` object. It needs to be created and destroyed in much the same way as the window object was:

13a ⟨*SDL: C Functions* 4a⟩+≡ (3c) ◁10a 14a▷

```
fn SDL_CreateRenderer<'win>
(    win: WindowHandle,
   index: c_int,
   flags: u32            ) -> ErrCheck<Renderer<'win>>;
fn SDL_DestroyRenderer
( handle: RendererHandle ) -> ();
```

13b ⟨*SDL: C Types* 4b⟩+≡ (3c) ◁10c 15a▷

```
#[derive(Debug, Copy, Clone)] #[repr(transparent)]
struct RendererHandle(*const c_void);

#[derive(Debug)] #[repr(transparent)]
pub struct Renderer<'win>(RendererHandle,
                          PhantomData<&'win Window<'win>>);

impl<'win> IsErr for Renderer<'win> { fn is_err(&self)->bool {
    (self.0).0.is_null() }}

impl<'win> Drop for Renderer<'win> { fn drop(&mut self) {
    unsafe { SDL_DestroyRenderer(self.0); }}}

impl<'win> Renderer<'win> { ⟨SDL: Renderer methods 14c⟩ }
```

The default paramaters seem fine, so creating the renderer is a simple call on the `Window` instance:

13c ⟨*SDL:* `Window` *Methods* 13c⟩≡ (8b)

```
pub fn create_renderer(&self) -> Result<Renderer, Box<Error>>
{ Ok(unsafe { SDL_CreateRenderer(self.0, -1, 0).unwrap()? }) }
```

13d ⟨*MAIN: Initialization* 6c⟩+≡ (3b) ◁9

```
let mut renderer = win.create_renderer()?;
```


9 *Drawing a Frame*

SDL Makes no guarantees about the contents of the drawing buffer after a frame is presented, so best practice is to clear it before drawing. We also need to call `SDL_RenderPresent` once we're done drawing the frame.

14a ⟨*SDL: C Functions* 4a⟩+≡ (3c) ◁13a 15b▷

```
fn SDL_SetRenderDrawColor
(   handle: RendererHandle,
    r:u8, g:u8, b:u8, a:u8  ) -> ErrCheck<c_int>;
fn SDL_RenderClear
(   handle: RendererHandle  ) -> ErrCheck<c_int>;
fn SDL_RenderPresent( handle: RendererHandle  );
```

Since RAII is the Rust way, we'll define a `Canvas` object that represents a single frame as it's being drawn. When that object goes out of scope, we'll use the `Drop` trait to send the completed frame to the screen.

14b ⟨*SDL: Definitions* 3c⟩+≡ (2a) ◁11b

```
pub struct Canvas<'r, 'w:'r>(&'r mut Renderer<'w>);
impl<'r,'w:'r> Drop for Canvas<'r,'w> { fn drop(&mut self) {
    unsafe { SDL_RenderPresent((self.0).0) };
}}
impl<'r,'w:'r> Canvas<'r,'w> { ⟨SDL: Canvas methods 15c⟩ }
```

14c ⟨*SDL:* Renderer *methods* 14c⟩≡ (13b)

```
pub fn start_frame<'r>(&'r mut self, clear_color: (u8,u8,u8))
-> Result<Canvas<'r,'win>, Box<Error>>
{   unsafe { SDL_SetRenderDrawColor(self.0,
                                    clear_color.0,
                                    clear_color.1,
                                    clear_color.2,
                                    255           ).unwrap()?;
             SDL_RenderClear       (self.0        ).unwrap()?; }
    Ok(Canvas(self)) }
```

14d ⟨*MAIN: Draw Frame* 14d⟩≡ (3b) 15d▷

```
let mut canvas = renderer.start_frame((128,0,255))?;
```

10 Rectangles

SDL uses a `Rect` struct to designate areas of the screen for various purposes, so we should probably get those working. The simplest API to test with is probably a rectange-filling function, so we'll implement that on the canvas too.

15a ⟨*SDL: C Types* 4b⟩+≡ (3c) ◁13b

```
#[derive(Debug, Copy, Clone)] #[repr(C)]
pub struct Rect { pub x: c_int, pub y: c_int,
                  pub w: c_int, pub h: c_int }
```

15b ⟨*SDL: C Functions* 4a⟩+≡ (3c) ◁14a

```
fn SDL_RenderFillRects(r    : RendererHandle,
                       rects: *const Rect,
                       count: c_int          )->ErrCheck<c_int>;
```

15c ⟨*SDL:* Canvas *methods* 15c⟩≡ (14b)

```
pub fn set_color(&mut self, r:u8, g:u8, b:u8, a:u8)
-> Result<(), Box<Error>>
{ Ok( unsafe { SDL_SetRenderDrawColor((self.0).0, r, g, b, a)
               .unwrap()?;
             })}
pub fn fill_rects(&mut self, rects: &Vec<Rect>)
-> Result<(), Box<Error>>
{ Ok( unsafe { SDL_RenderFillRects((self.0).0,
                                   rects.as_ptr(),
                                   rects.len() as i32).unwrap()?; })}
```

15d ⟨*MAIN: Draw Frame* 14d⟩+≡ (3b) ◁14d

```
canvas.set_color(200,200,200,255)?;
canvas.fill_rects(&vec![ Rect { x: 10, y: 10, w:140, h:80 },
                         Rect { x:170, y: 10, w:140, h:80 },
                         Rect { x:170, y:110, w:140, h:80 }, ])?;
```

11 Licensing