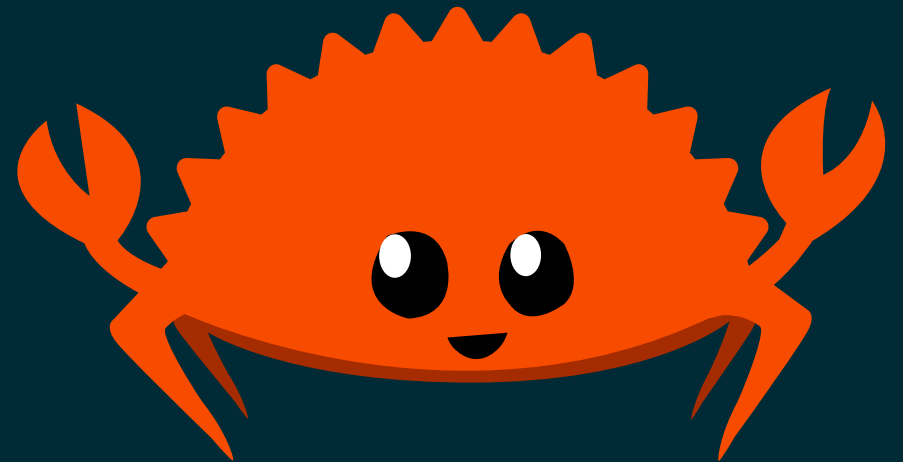


Replacing C library code with Rust:

What I learned with librsvg

Federico Mena Quintero
federico@gnome.org

GUADEC 2017
Manchester, UK



What uses librsvg?

GNOME platform and desktop:

gtk+ indirectly through gdk-pixbuf

thumbnailer via gdk-pixbuf

eog

gstreamer-plugins-bad

What uses librsvg?

Fine applications:

gnome-games (gnome-chess, five-or-more, etc.)

gimp

gcompris

claws-mail

darktable

What uses librsvg?

Desktop environments

mate-panel

Evas / Enlightenment

emacs-x11

What uses librsvg?

Things you may not expect

ImageMagick

Wikipedia ← *they have been awesome*

Long History

- First commit, Eazel, 2001
- Experiment instead of building a DOM, stream in SVG by using libxml2's SAX parser
- Renderer used libart
- Gill → Sodipodi → Inkscape
- Librsvg was being written while the SVG spec was being written
- Ported to Cairo eventually
 - I'll remove the last libart-ism any day now

Federico takes over

- Librsvg was mostly unmaintained in 2015
- Took over maintenance in February 2015
- Started the Rust port in October 2016

Pain points: librsvg's CVEs

- CVE-2011-3146 - invalid cast of RsvgNode type, crash
- CVE-2015-7557 - out-of-bounds heap read in list-of-points
- CVE-2015-7558 - Infinite loop / stack overflow from cyclic element references (thanks to Benjamin Otte for the epic fix)
- CVE-2016-4348 - *NOT OUR PROBLEM* - integer overflow when writing PNGs in Cairo (rowstride computation)
- CVE-2017-11464 - Division by zero in Gaussian blur code (my bad, sorry)

- Many non-CVEs found through fuzz testing and random SVGs on the net

- <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=librsvg>
- CVE = Common Vulnerabilities and Exposures

More CVEs

- **libcroco**

 - 4 CVEs – out of bounds read, malloc() failure, two unfixed ones (!)

- **Cairo**

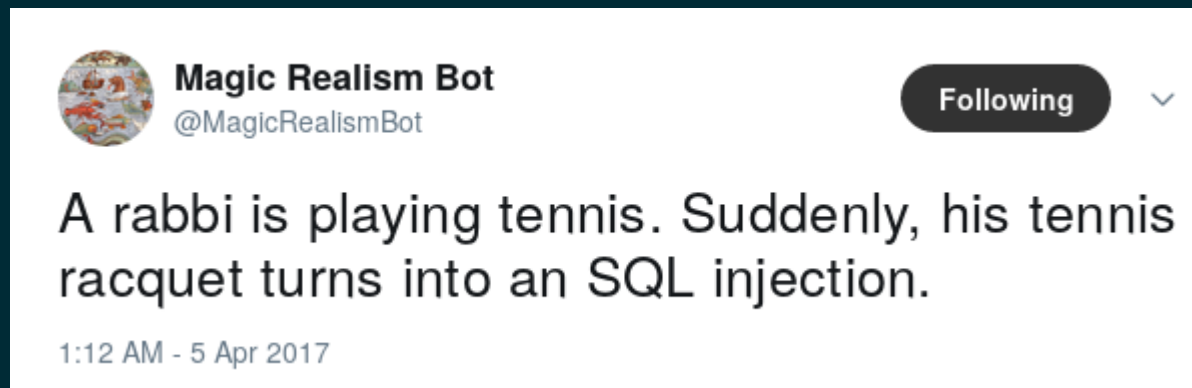
 - 20 CVEs

- **libxml2 / libxml**

 - 85 + 54 CVEs

Pain points – so what?

- Best case for people – app crashes
- Worst case for people – pwned by baddies
- Worst case for robots – Denial-of-Service, pwned by baddies



Hi. I'm Ferris the rustacean.



The decision to Rustify

- Allison's overflow-checked arithmetic macros in glib
- **"Rust out your C"**
Carol Nichols
<https://github.com/carols10cents/rust-out-your-c-talk>
- **"Writing GStreamer elements in Rust"** Sebastian Dröge
<https://coaxion.net/blog/2016/05/writing-gstreamer-plugins-and-elements-in-rust/>

Disclaimers

- Librsvg is lower level than most GNOME libs
- Doesn't use GTK+
- Public API is a **single** GObject from C
- I'm a beginner/intermediate Rust programmer
- Don't use this talk as a source of good Rust practices
- Do talk to me about your porting effort!
 - *Ask me about encouragement for getting in trouble*
- Do not expect to learn a lot of Rust from this talk
- We are naturalists; I'm showing you critters I found

Rust's killer features – for me

- Safe, modern language, blah blah blah
- Generates object code - `.o` / `.a` / `.so`
- Can be linked to C code
- No runtime, no garbage collection
- Can export and consume C ABI
- Makes it easy to write unit tests

Easy to write unit tests

```
#[test]
fn check_that_it_works () {
    assert! (internal_function() == expected_value);
}
```

```
$ cargo test
```

```
...
running 105 tests
test aspect_ratio::tests::aligns ... ok
test aspect_ratio::tests::conversion_to_u32_roundtrips ... ok
test aspect_ratio::tests::parses_valid_strings ... ok
test aspect_ratio::tests::parsing_invalid_strings_yields_error ... ok

test result: ok. 104 passed; 0 failed; 1 ignored; 0 measured
```

How I started porting librsvg

- Port internals (algorithms / parsers) to Rust
- Write unit tests; complement with black-box tests with SVG files
 - Document the black-box test suite
 - Thanks again to Benjamin for writing that test suite
- Leave public C API intact, for now
- C API is consumed by GObject-Introspection

How to port a bit of C code

- Refactor C to make it Rust-friendly
- Add tests
- Port to Rust
- Test
- Refactor Rust to make it pretty

Integrating into the build

- Librsvg uses autotools
- Rust uses cargo
- Build a librsvg_internals.a
- In Cargo.toml:

```
[lib]
name = "rsvg_internals"
crate-type = ["staticlib"]
```

```
librsvg/
  configure.ac
  Makefile.am
  *.[ch]
```

```
rust/
  Cargo.toml
  src/
    *.rs
  target/
```

Autotools thanks

- Hubert Figuière
 - <https://www.figuere.net/hub/blog/?2016/10/07/862-rust-and-automake>
- Luke Nukem
 - http://lukenukem.co.nz/gsoc/2017/05/17/gso_2.html
- Havoc Pennington
 - <https://blog.ometer.com/2017/01/10/dear-package-managers-dependency-resolution-results-should-be-in-version-control/>

Progression

- Once some of the leaf code was ported...
- **PORT ALL THE THINGS**
- Especially the shitty C parsers
- Big goals, not yet realized:
 - Replace libcroco (CSS parser)
 - Replace little CSS engine with Servo's
 - Replace libxml2

Debugging Rust is awesome

- gdb, valgrind, rr all work fine
- Print anything with the Debug trait

```
#[derive(Debug)]
struct BigAssStruct {
    tons_of_fields: ...;
    some_more_fields: ...;
}

println! ("{:?}", my_struct);

println! ("{:#?}", my_struct); // pretty-print
```

Error propagation

- Librsvg “handled” errors by reverting to default values (or garbage)

```
<rect x="bleh" y="5" width="eek".../>  
<path stroke="#000000" fill="#12wxyz"/>
```

- SVG spec says what to do
 - It doesn't agree with itself
 - Implementations don't agree with it
 - 冫(ツ)冫

Evolution of set_atts()

- No error handling at first:

```
void (*set_atts) (RsvgNode *self, RsvgHandle *handle, RsvgPropertyBag *pbag);

static void
rsvg_node_svg_set_atts (RsvgNodeSvg *svg, RsvgHandle *handle,
                        RsvgPropertyBag *pbag)
{
    if ((value = rsvg_property_bag_lookup (pbag, "preserveAspectRatio")))
        svg->preserve_aspect_ratio = rsvg_aspect_ratio_parse (value);

    if ((value = rsvg_property_bag_lookup (pbag, "viewBox")))
        svg->vbox = rsvg_css_parse_vbox (value);
    ...
}
```

```

enum MarkerUnits { UserSpaceOnUse, StrokeWidth };

impl Default for MarkerUnits {
    fn default () -> MarkerUnits { MarkerUnits::StrokeWidth }
}

impl FromStr for MarkerUnits {
    type Err = AttributeError;

    fn from_str (s: &str) -> Result <MarkerUnits, AttributeError> { ... }
}

impl NodeTrait for NodeMarker {
    fn set_atts (&self,
                _: &RsvgNode,
                _: *const RsvgHandle,
                pbag: *const RsvgPropertyBag) -> NodeResult {

        self.units.set (
            property_bag::parse_or_default (pbag, "markerUnits"?);
            ...
        )
    }
}

pub fn parse_or_default<T> (pbag: *const RsvgPropertyBag, key: &'static str) ->
    Result <T, NodeError>
    where T: Default + FromStr<Err = AttributeError>

```



```

enum MarkerUnits { UserSpaceOnUse, StrokeWidth };

impl Default for MarkerUnits {
    fn default () -> MarkerUnits { MarkerUnits::StrokeWidth }
}

impl FromStr for MarkerUnits {
    type Err = AttributeError;

    fn from_str (s: &str) -> Result <MarkerUnits, AttributeError> { ... }
}

impl NodeTrait for NodeMarker {
    fn set_atts (&self,
                _: &RsvgNode,
                _: *const RsvgHandle,
                pbag: *const RsvgPropertyBag) -> NodeResult {

        self.units.set (
            property_bag::parse_or_default (pbag, "markerUnits"?);
            ...
        )
    }
}

pub fn parse_or_default<T> (pbag: *const RsvgPropertyBag, key: &'static str) ->
    Result <T, NodeError>
    where T: Default + FromStr<Err = AttributeError>

```

```

enum MarkerUnits { UserSpaceOnUse, StrokeWidth };

impl Default for MarkerUnits {
    fn default () -> MarkerUnits { MarkerUnits::StrokeWidth }
}

impl FromStr for MarkerUnits {
    type Err = AttributeError;

    fn from_str (s: &str) -> Result <MarkerUnits, AttributeError> { ... }
}

impl NodeTrait for NodeMarker {
    fn set_atts (&self,
        _: &RsvgNode,
        _: *const RsvgHandle,
        pbag: *const RsvgPropertyBag) -> NodeResult {

        self.units.set (
            property_bag::parse_or_default (pbag, "markerUnits"?);
            ...
        )
    }
}

pub fn parse_or_default<T> (pbag: *const RsvgPropertyBag, key: &'static str) ->
    Result <T, NodeError>
    where T: Default + FromStr<Err = AttributeError>

```

```

enum MarkerUnits { UserSpaceOnUse, StrokeWidth };

impl Default for MarkerUnits {
    fn default () -> MarkerUnits { MarkerUnits::StrokeWidth }
}

impl FromStr for MarkerUnits {
    type Err = AttributeError;

    fn from_str (s: &str) -> Result <MarkerUnits, AttributeError> { ... }
}

impl NodeTrait for NodeMarker {
    fn set_atts (&self,
                _: &RsvgNode,
                _: *const RsvgHandle,
                pbag: *const RsvgPropertyBag) -> NodeResult {

        self.units.set (
            property_bag::parse_or_default (pbag, "markerUnits"?);
            ...
        )
    }
}

pub fn parse_or_default<T> (pbag: *const RsvgPropertyBag, key: &'static str) ->
    Result <T, NodeError>
    where T: Default + FromStr<Err = AttributeError>

```

```

enum MarkerUnits { UserSpaceOnUse, StrokeWidth };

impl Default for MarkerUnits {
    fn default () -> MarkerUnits { MarkerUnits::StrokeWidth }
}

impl FromStr for MarkerUnits {
    type Err = AttributeError;

    fn from_str (s: &str) -> Result <MarkerUnits, AttributeError> { ... }
}

impl NodeTrait for NodeMarker {
    fn set_atts (&self,
                _: &RsvgNode,
                _: *const RsvgHandle,
                pbag: *const RsvgPropertyBag) -> NodeResult {

        self.units.set (
            property_bag::parse_or_default (pbag, "markerUnits"?);
            ...
        )
    }
}

pub fn parse_or_default<T> (pbag: *const RsvgPropertyBag, key: &'static str) ->
    Result <T, NodeError>
    where T: Default + FromStr<Err = AttributeError>

```


Expose stuff from C to Rust

The unsafe / “familiar” way

- Declare parallel structs/unions with `#[repr(C)]`
- Call C functions; dereference pointers they send back
- All dereferences marked as `unsafe{}`
- Quick code to write; easiest if you cut&paste from C
- You’ll probably end up with safe Rust wrappers over `unsafe{}` code
- You probably need this if you are plugging into “unchangeable” C code

Expose stuff from C to Rust

The safe way

- Create a C API, possibly internal, that is Rust-friendly
- Or refactor the relevant code to be Rust-friendly
- Expose only opaque pointers to structs
- Have C functions with setters/getters for those structs; call them from Rust
- Still need to mark calls as `unsafe{}`
- I like this way if you are completely free to change the C code
- Leave the public API intact; change only the internals

Expose stuff from Rust to C

- Parallel structs with `#[repr(C)]`
- Expose opaque pointers to structs, and functions to frob them
- Get the memory management right
- `new() / destroy()` - <https://people.gnome.org/~federico/news-2016-11.html#14>
- Reference counting - <https://people.gnome.org/~federico/news-2017-02.html#17>

Parsers

Bad C parser for #rrggbb hex colors

```
uint32
rsvg_css_parse_color (const char *str, ...)
{
    gint val = 0;

    if (str[0] == '#') {
        int i;
        for (i = 1; str[i]; i++) {
            int hexval;
            if (str[i] >= '0' && str[i] <= '9')
                hexval = str[i] - '0';
            else if (str[i] >= 'A' && str[i] <= 'F')
                hexval = str[i] - 'A' + 10;
            else if (str[i] >= 'a' && str[i] <= 'f')
                hexval = str[i] - 'a' + 10;
            else
                break;
            val = (val << 4) + hexval;
        }
        /* handle #rgb case */
        if (i == 4) {
            val = ((val & 0xf00) << 8) | ((val & 0x0f0) << 4) |
                (val & 0x00f);
            val |= val << 4;
        }

        val |= 0xff000000; /* opaque */
    }
}
```

Bad C parser for #rrggbb hex colors

```
uint32
rsvg_css_parse_color (const char *str, ...)
{
    uint val = 0;

    if (str[0] == '#') {
        int i;
        for (i = 1; str[i]; i++) {
            int hexval;
            if (str[i] >= '0' && str[i] <= '9')
                hexval = str[i] - '0';
            else if (str[i] >= 'A' && str[i] <= 'F')
                hexval = str[i] - 'A' + 10;
            else if (str[i] >= 'a' && str[i] <= 'f')
                hexval = str[i] - 'a' + 10;
            else
                break;
            val = (val << 4) + hexval;
        }
        /* handle #rgb case */
        if (i == 4) {
            val = ((val & 0xf00) << 8) | ((val & 0x0f0) << 4) |
                (val & 0x00f);
            val |= val << 4;
        }

        val |= 0xff000000; /* opaque */
    }
}
```

Bad C parser for #rrggbb hex colors

```
uint32
rsvg_css_parse_color (const char *str, ...)
{
    uint val = 0;

    if (str[0] == '#') {
        int i;
        for (i = 1; str[i]; i++) {
            int hexval;
            if (str[i] >= '0' && str[i] <= '9')
                hexval = str[i] - '0';
            else if (str[i] >= 'A' && str[i] <= 'F')
                hexval = str[i] - 'A' + 10;
            else if (str[i] >= 'a' && str[i] <= 'f')
                hexval = str[i] - 'a' + 10;
            else
                break;
            val = (val << 4) + hexval;
        }
        /* handle #rgb case */
        if (i == 4) {
            val = ((val & 0xf00) << 8) | ((val & 0x0f0) << 4) |
                (val & 0x00f);
            val |= val << 4;
        }

        val |= 0xff000000; /* opaque */
    }
}
```

Bad C parser for #rrggbb hex colors

```
uint32
rsvg_css_parse_color (const char *str, ...)
{
    uint val = 0;

    if (str[0] == '#') {
        int i;
        for (i = 1; str[i]; i++) {
            int hexval;
            if (str[i] >= '0' && str[i] <= '9')
                hexval = str[i] - '0';
            else if (str[i] >= 'A' && str[i] <= 'F')
                hexval = str[i] - 'A' + 10;
            else if (str[i] >= 'a' && str[i] <= 'f')
                hexval = str[i] - 'a' + 10;
            else
                break;
            val = (val << 4) + hexval;
        }
        /* handle #rgb case */
        if (i == 4) {
            val = ((val & 0xf00) << 8) | ((val & 0x0f0) << 4) |
                (val & 0x00f);
            val |= val << 4;
        }

        val |= 0xff000000; /* opaque */
    }
}
```

Bad C parser for #rrggbb hex colors

```
uint32
rsvg_css_parse_color (const char *str, ...)
{
    uint val = 0;

    if (str[0] == '#') {
        int i;
        for (i = 1; str[i]; i++) {
            int hexval;
            if (str[i] >= '0' && str[i] <= '9')
                hexval = str[i] - '0';
            else if (str[i] >= 'A' && str[i] <= 'F')
                hexval = str[i] - 'A' + 10;
            else if (str[i] >= 'a' && str[i] <= 'f')
                hexval = str[i] - 'a' + 10;
            else
                break;
            val = (val << 4) + hexval;
        }
        /* handle #rgb case */
        if (i == 4) {
            val = ((val & 0xf00) << 8) | ((val & 0x0f0) << 4) |
                (val & 0x00f);
            val |= val << 4;
        }

        val |= 0xff000000; /* opaque */
    }
}
```

Bad C parser for #rrggbb hex colors

```
uint32
rsvg_css_parse_color (const char *str, ...)
{
    uint val = 0;

    if (str[0] == '#') {
        int i;
        for (i = 1; str[i]; i++) {
            int hexval;
            if (str[i] >= '0' && str[i] <= '9')
                hexval = str[i] - '0';
            else if (str[i] >= 'A' && str[i] <= 'F')
                hexval = str[i] - 'A' + 10;
            else if (str[i] >= 'a' && str[i] <= 'f')
                hexval = str[i] - 'a' + 10;
            else
                break;
            val = (val << 4) + hexval;
        }
        /* handle #rgb case */
        if (i == 4) {
            val = ((val & 0xf00) << 8) | ((val & 0x0f0) << 4) |
                (val & 0x00f);
            val |= val << 4;
        }

        val |= 0xff000000; /* opaque */
    }
}
```

Bad C parser for #rrggbb hex colors

```
uint32
rsvg_css_parse_color (const char *str, ...)
{
    gint val = 0;

    if (str[0] == '#') {
        int i;
        for (i = 1; str[i]; i++) {
            int hexval;
            if (str[i] >= '0' && str[i] <= '9')
                hexval = str[i] - '0';
            else if (str[i] >= 'A' && str[i] <= 'F')
                hexval = str[i] - 'A' + 10;
            else if (str[i] >= 'a' && str[i] <= 'f')
                hexval = str[i] - 'a' + 10;
            else
                break;
            val = (val << 4) + hexval;
        }
        /* handle #rgb case */
        if (i == 4) {
            val = ((val & 0xf00) << 8) | ((val & 0x0f0) << 4) |
                (val & 0x00f);
            val |= val << 4;
        }

        val |= 0xff000000; /* opaque */
    }
}
```


Bad C parser for #rrggbb hex colors

```
uint32
rsvg_css_parse_color (const char *str, ...)
{
    uint val = 0;

    if (str[0] == '#') {
        int i;
        for (i = 1; str[i]; i++) {
            int hexval;
            if (str[i] >= '0' && str[i] <= '9')
                hexval = str[i] - '0';
            else if (str[i] >= 'A' && str[i] <= 'F')
                hexval = str[i] - 'A' + 10;
            else if (str[i] >= 'a' && str[i] <= 'f')
                hexval = str[i] - 'a' + 10;
            else
                break;
            val = (val << 4) + hexval;
        }
        /* handle #rgb case */
        if (i == 4) {
            val = ((val & 0xf00) << 8) | ((val & 0x0f0) << 4) |
                (val & 0x00f);
            val |= val << 4;
        }

        val |= 0xff000000; /* opaque */
    }
}
```

Bad C parser for #rrggbb hex colors

```
uint32
rsvg_css_parse_color (const char *str, ...)
{
    uint val = 0;

    if (str[0] == '#') {
        int i;
        for (i = 1; str[i]; i++) {
            int hexval;
            if (str[i] >= '0' && str[i] <= '9')
                hexval = str[i] - '0';
            else if (str[i] >= 'A' && str[i] <= 'F')
                hexval = str[i] - 'A' + 10;
            else if (str[i] >= 'a' && str[i] <= 'f')
                hexval = str[i] - 'a' + 10;
            else
                break;
            val = (val << 4) + hexval;
        }
        /* handle #rgb case */
        if (i == 4) {
            val = ((val & 0xf00) << 8) | ((val & 0x0f0) << 4) |
                (val & 0x00f);
            val |= val << 4;
        }

        val |= 0xff000000; /* opaque */
    }
}
```

Bad C parser for #rrggbb hex colors

```
uint32
rsvg_css_parse_color (const char *str, ...)
{
    uint val = 0;

    if (str[0] == '#') {
        int i;
        for (i = 1; str[i]; i++) {
            int hexval;
            if (str[i] >= '0' && str[i] <= '9')
                hexval = str[i] - '0';
            else if (str[i] >= 'A' && str[i] <= 'F')
                hexval = str[i] - 'A' + 10;
            else if (str[i] >= 'a' && str[i] <= 'f')
                hexval = str[i] - 'a' + 10;
            else
                break;
            val = (val << 4) + hexval;
        }
        /* handle #rgb case */
        if (i == 4) {
            val = ((val & 0xf00) << 8) | ((val & 0x0f0) << 4) |
                (val & 0x00f);
            val |= val << 4;
        }

        val |= 0xff000000; /* opaque */
    }
}
```

Bad C parser for #rrggbb hex colors

```
uint32
rsvg_css_parse_color (const char *str, ...)
{
    uint val = 0;

    if (str[0] == '#') {
        int i;
        for (i = 1; str[i]; i++) {
            int hexval;
            if (str[i] >= '0' && str[i] <= '9')
                hexval = str[i] - '0';
            else if (str[i] >= 'A' && str[i] <= 'F')
                hexval = str[i] - 'A' + 10;
            else if (str[i] >= 'a' && str[i] <= 'f')
                hexval = str[i] - 'a' + 10;
            else
                break;
            val = (val << 4) + hexval;
        }
        /* handle #rgb case */
        if (i == 4) {
            val = ((val & 0xf00) << 8) | ((val & 0x0f0) << 4) |
                (val & 0x00f);
            val |= val << 4;
        }

        val |= 0xff000000; /* opaque */
    }
}
```

Bad C parser for #rrggbb hex colors

```
uint32
rsvg_css_parse_color (const char *str, ...)
{
    gint val = 0;

    if (str[0] == '#') {
        int i;
        for (i = 1; str[i]; i++) {
            int hexval;
            if (str[i] >= '0' && str[i] <= '9')
                hexval = str[i] - '0';
            else if (str[i] >= 'A' && str[i] <= 'F')
                hexval = str[i] - 'A' + 10;
            else if (str[i] >= 'a' && str[i] <= 'f')
                hexval = str[i] - 'a' + 10;
            else
                break;
            val = (val << 4) + hexval;
        }
        /* handle #rgb case */
        if (i == 4) {
            val = ((val & 0xf00) << 8) | ((val & 0x0f0) << 4) |
                (val & 0x00f);
            val |= val << 4;
        }

        val |= 0xff000000; /* opaque */
    }
}
```

Parse a hex digit in Rust

```
fn from_hex(c: u8) -> Result<u8, ()> {  
    match c {  
        b'0' ... b'9' => Ok(c - b'0'),  
        b'a' ... b'f' => Ok(c - b'a' + 10),  
        b'A' ... b'F' => Ok(c - b'A' + 10),  
        _ => Err(())  
    }  
}
```

Parse a hex digit in Rust

```
fn from_hex(c: u8) -> Result<u8, ()> {  
    match c {  
        b'0' ... b'9' => Ok(c - b'0'),  
        b'a' ... b'f' => Ok(c - b'a' + 10),  
        b'A' ... b'F' => Ok(c - b'A' + 10),  
        _ => Err(())  
    }  
}
```

Parse a hex color in Rust

```
impl Color {
    pub fn parse_hash(value: &[u8]) -> Result<Self, ()> {
        match value.len() {
            8 => Ok(rgba(
                from_hex(value[0])? * 16 + from_hex(value[1])?,
                from_hex(value[2])? * 16 + from_hex(value[3])?,
                from_hex(value[4])? * 16 + from_hex(value[5])?,
                from_hex(value[6])? * 16 + from_hex(value[7])?,
            )),
            6 => Ok(rgb(...)),
            4 => Ok(rgb(...)),
            3 => Ok(rgb(...)),
            _ => Err(())
        }
    }
}
```


Parse a hex color in Rust

```
impl Color {
    pub fn parse_hash(value: &[u8]) -> Result<Self, ()> {
        match value.len() {
            8 => Ok(rgba(
                from_hex(value[0])? * 16 + from_hex(value[1])?,
                from_hex(value[2])? * 16 + from_hex(value[3])?,
                from_hex(value[4])? * 16 + from_hex(value[5])?,
                from_hex(value[6])? * 16 + from_hex(value[7])?,
            )),
            6 => Ok(rgb(...)),
            4 => Ok(rgb(...)),
            3 => Ok(rgb(...)),
            _ => Err(())
        }
    }
}
```

Parse a hex color in Rust

```
impl Color {
    pub fn parse_hash(value: &[u8]) -> Result<Self, ()> {
        match value.len() {
            8 => Ok(rgba(
                from_hex(value[0])? * 16 + from_hex(value[1])?,
                from_hex(value[2])? * 16 + from_hex(value[3])?,
                from_hex(value[4])? * 16 + from_hex(value[5])?,
                from_hex(value[6])? * 16 + from_hex(value[7])?,
            )),
            6 => Ok(rgb(...)),
            4 => Ok(rgb(...)),
            3 => Ok(rgb(...)),
            _ => Err(())
        }
    }
}
```

Parse a hex color in Rust

```
impl Color {
    pub fn parse_hash(value: &[u8]) -> Result<Self, ()> {
        match value.len() {
            8 => Ok(rgba(
                from_hex(value[0])? * 16 + from_hex(value[1])?,
                from_hex(value[2])? * 16 + from_hex(value[3])?,
                from_hex(value[4])? * 16 + from_hex(value[5])?,
                from_hex(value[6])? * 16 + from_hex(value[7])?,
            )),
            6 => Ok(rgb(...)),
            4 => Ok(rgb(...)),
            3 => Ok(rgb(...)),
            _ => Err(())
        }
    }
}
```

Parse a hex color in Rust

```
impl Color {
    pub fn parse_hash(value: &[u8]) -> Result<Self, ()> {
        match value.len() {
            8 => Ok(rgba(
                from_hex(value[0])? * 16 + from_hex(value[1])?,
                from_hex(value[2])? * 16 + from_hex(value[3])?,
                from_hex(value[4])? * 16 + from_hex(value[5])?,
                from_hex(value[6])? * 16 + from_hex(value[7])?,
            )),
            6 => Ok(rgb(...)),
            4 => Ok(rgb(...)),
            3 => Ok(rgb(...)),
            _ => Err(())
        }
    }
}
```

Parse a hex color in Rust

```
impl Color {
    pub fn parse_hash(value: &[u8]) -> Result<Self, ()> {
        match value.len() {
            8 => Ok(rgba(
                from_hex(value[0])? * 16 + from_hex(value[1])?,
                from_hex(value[2])? * 16 + from_hex(value[3])?,
                from_hex(value[4])? * 16 + from_hex(value[5])?,
                from_hex(value[6])? * 16 + from_hex(value[7])?,
            )),
            6 => Ok(rgb(...)),
            4 => Ok(rgb(...)),
            3 => Ok(rgb(...)),
            _ => Err(())
        }
    }
}
```

It's not bad programmers, it's that C is a hostile language

- I did a “git blame” to see who wrote and modified the shitty parsers.
- Three or four of the best programmers GNOME has ever had.
- I know they wouldn't write *that* from scratch.
- The C language just makes it hard to do the right thing in the beginning, and makes refactoring hard.

Hand-written parsers in Rust

- No more awkward than in any other language
- Safe
- Can be made fast – zero-copy with slices and iterators
- Rust's `String.split()` family is AWESOME

```
let mut iter = "hello world".split_whitespace ();

while let Some (word) = iter.next () {
    println! ("{}", word);
}
```

Combinator parsers with nom

```
// parse something like
//
// 5.2em

named! (parse_number_and_units<(f64, &[u8])>,
        tuple! (double,
                take_while! (is_alphabetic)));
```

- Done with macros
- People seem very happy with nom for parsing binary files?

Yacc-like parsers with lalrpop

```
Transform: cairo::Matrix = {
    Matrix,
    Translate,
    Scale,
    Rotate,
    SkewX,
    SkewY
};

Scale: cairo::Matrix = {
    // scale (42)
    "scale" "(" <Num> ")" => cairo::Matrix::new (<>, 0.0, 0.0, <>, 0.0, 0.0),

    // scale (3.5, 4.6)
    "scale" "(" <sx: Num> maybe_comma <sy: Num> ")" =>
        cairo::Matrix::new (sx, 0.0, 0.0, sy, 0.0, 0.0),
};

Matrix: ...
Rotate: ...
SkewX: ...

// regex for floating-point number
pub Num: f64 = <s:r"[+-]?([0-9]*\.[0-9]+|[0-9]+(\.[0-9]*)?)([Ee][+-]?[0-9]+)?">
=>
    f64::from_str (s).unwrap ();

maybe_comma: () = ", " ?;
```

Yacc-like parsers with lalrpop

```
Transform: cairo::Matrix = {
    Matrix,
    Translate,
    Scale,
    Rotate,
    SkewX,
    SkewY
};

Scale: cairo::Matrix = {
    // scale (42)
    "scale" "(" <Num> ")" => cairo::Matrix::new (<>, 0.0, 0.0, <>, 0.0, 0.0),

    // scale (3.5, 4.6)
    "scale" "(" <sx: Num> maybe_comma <sy: Num> ")" =>
        cairo::Matrix::new (sx, 0.0, 0.0, sy, 0.0, 0.0),
};

Matrix: ...
Rotate: ...
SkewX: ...

// regex for floating-point number
pub Num: f64 = <s:r"[+-]?([0-9]*\.[0-9]+|[0-9]+(\.[0-9]*)?)([Ee][+-]?[0-9]+)?">
=>
    f64::from_str (s).unwrap ();

maybe_comma: () = ", " ?;
```

Yacc-like parsers with lalrpop

```
Transform: cairo::Matrix = {
    Matrix,
    Translate,
    Scale,
    Rotate,
    SkewX,
    SkewY
};

Scale: cairo::Matrix = {
    // scale (42)
    "scale" "(" <Num> ")" => cairo::Matrix::new (<>, 0.0, 0.0, <>, 0.0, 0.0),

    // scale (3.5, 4.6)
    "scale" "(" <sx: Num> maybe_comma <sy: Num> ")" =>
        cairo::Matrix::new (sx, 0.0, 0.0, sy, 0.0, 0.0),
};

Matrix: ...
Rotate: ...
SkewX: ...

// regex for floating-point number
pub Num: f64 = <s:r"[+-]?([0-9]*\.[0-9]+|[0-9]+(\.[0-9]*)?)([Ee][+-]?[0-9]+)?">
=>
    f64::from_str (s).unwrap ();

maybe_comma: () = ", " ?;
```

Yacc-like parsers with lalrpop

```
Transform: cairo::Matrix = {
    Matrix,
    Translate,
    Scale,
    Rotate,
    SkewX,
    SkewY
};

Scale: cairo::Matrix = {
    // scale (42)
    "scale" "(" <Num> ")" => cairo::Matrix::new (<>, 0.0, 0.0, <>, 0.0, 0.0),

    // scale (3.5, 4.6)
    "scale" "(" <sx: Num> maybe_comma <sy: Num> ")" =>
        cairo::Matrix::new (sx, 0.0, 0.0, sy, 0.0, 0.0),
};

Matrix: ...
Rotate: ...
SkewX: ...

// regex for floating-point number
pub Num: f64 = <s:r"[+-]?([0-9]*\.[0-9]+|[0-9]+(\.[0-9]*)?)([Ee][+-]?[0-9]+)?">
=>
    f64::from_str (s).unwrap ();

maybe_comma: () = ", "?;
```

Yacc-like parsers with lalrpop

```
Transform: cairo::Matrix = {
    Matrix,
    Translate,
    Scale,
    Rotate,
    SkewX,
    SkewY
};

Scale: cairo::Matrix = {
    // scale (42)
    "scale" "(" <Num> ")" => cairo::Matrix::new (<>, 0.0, 0.0, <>, 0.0, 0.0),

    // scale (3.5, 4.6)
    "scale" "(" <sx: Num> maybe_comma <sy: Num> ")" =>
        cairo::Matrix::new (sx, 0.0, 0.0, sy, 0.0, 0.0),
};

Matrix: ...
Rotate: ...
SkewX: ...

// regex for floating-point number
pub Num: f64 = <s:r"[+-]?([0-9]*\.[0-9]+|[0-9]+(\.[0-9]*)?)([Ee][+-]?[0-9]+)?">
=>
    f64::from_str (s).unwrap ();

maybe_comma: () = ", "?;
```

Yacc-like parsers with lalrpop

```
Transform: cairo::Matrix = {
    Matrix,
    Translate,
    Scale,
    Rotate,
    SkewX,
    SkewY
};

Scale: cairo::Matrix = {
    // scale (42)
    "scale" "(" <Num> ")" => cairo::Matrix::new (<>, 0.0, 0.0, <>, 0.0, 0.0),

    // scale (3.5, 4.6)
    "scale" "(" <sx: Num> maybe_comma <sy: Num> ")" =>
        cairo::Matrix::new (sx, 0.0, 0.0, sy, 0.0, 0.0),
};

Matrix: ...
Rotate: ...
SkewX: ...

// regex for floating-point number
pub Num: f64 = <s:r"[+-]?([0-9]*\.[0-9]+|[0-9]+(\.[0-9]*)?)([Ee][+-]?[0-9]+)?">
=>
    f64::from_str (s).unwrap ();

maybe_comma: () = ", " ?;
```

Yacc-like parsers with lalrpop

```
Transform: cairo::Matrix = {
    Matrix,
    Translate,
    Scale,
    Rotate,
    SkewX,
    SkewY
};

Scale: cairo::Matrix = {
    // scale (42)
    "scale" "(" <Num> ")" => cairo::Matrix::new (<>, 0.0, 0.0, <>, 0.0, 0.0),

    // scale (3.5, 4.6)
    "scale" "(" <sx: Num> maybe_comma <sy: Num> ")" =>
        cairo::Matrix::new (sx, 0.0, 0.0, sy, 0.0, 0.0),
};

Matrix: ...
Rotate: ...
SkewX: ...

// regex for floating-point number
pub Num: f64 = <s:r"[+-]?([0-9]*\.[0-9]+|[0-9]+(\.[0-9]*)?)([Ee][+-]?[0-9]+)?">
=>
    f64::from_str (s).unwrap ();

maybe_comma: () = ", "?;
```

Yacc-like parsers with lalrpop

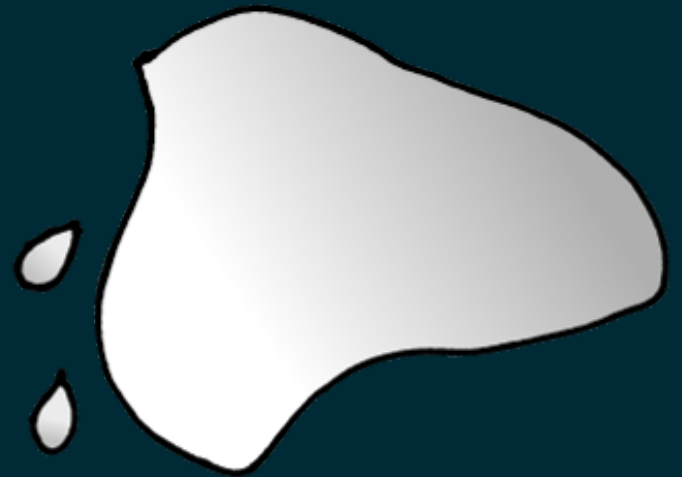
- Great documentation
- Possible to define one's own lexer
- Supports error recovery
- Love all around
- Generates big, fast parsers

Many other parsing libraries

- Rust-cssparser
- Serde (serialization / deserialization) is great
- Shop around!



3, 2, 1...



When Rust clarifies your thinking

```
if (!cairo_matrix_invert (...))  
    return;
```

- Degenerate matrices were ignored
- They were actually real bugs:
 - Uninitialized data
 - Invalid floats with all-bits-zero
 - Malformed SVG files – no data validation
- Cairo-rs calls `panic!()` on degenerate matrices
 - Added `try_invert()`; love all around

Can distros ship this?

- “cargo vendor”
- Haven’t done this yet
- Firefox 53+ will help here

Self-sanity

- Rust is not easy to learn
- New concepts (ownership tracking, functional paradigms, big syntax, etc.)
- Learning stage: “fighting the borrow checker”
- C is a sea of aliased pointers;
Rust Does Not Do That™

Self-sanity

- You won't know how to port things, or what crates to reuse
- It's OK not to use the best idioms in the beginning. Learn some more; refactor later.
- Tests are super important!
- Learn the error handling scheme:
Result<Ok, Err>
- The Rust community is AWESOME.

Should we replace C library code with Rust?

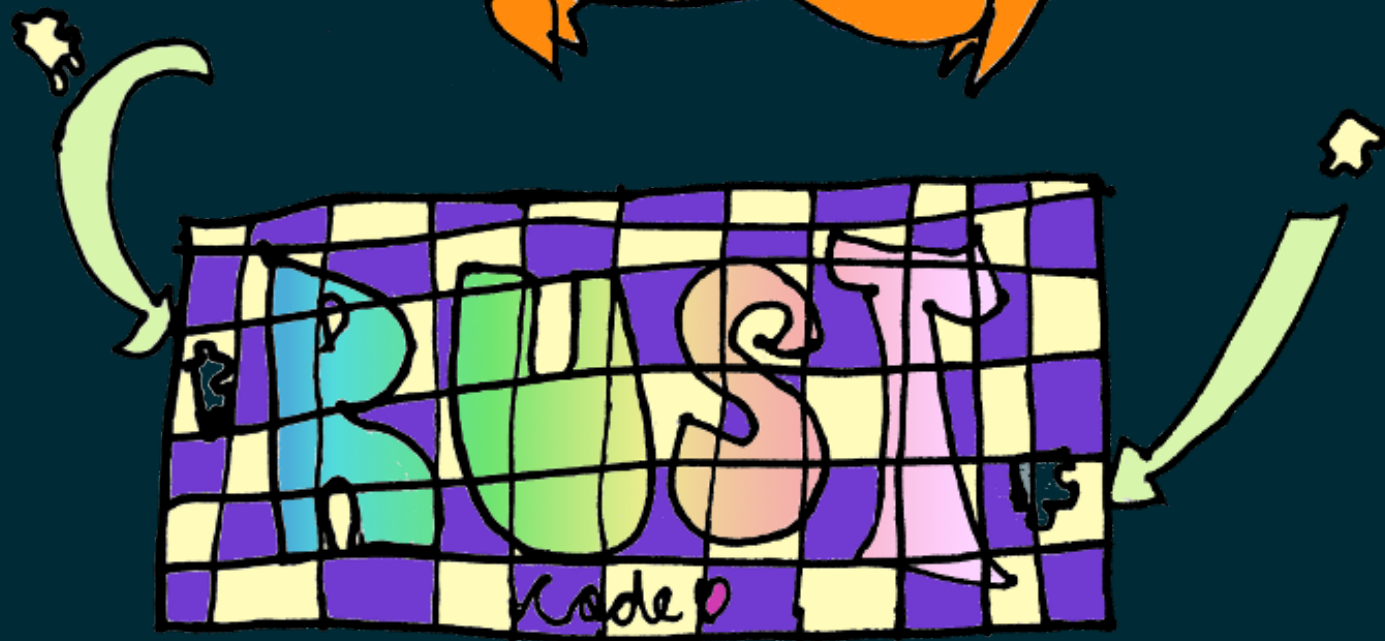
- Parsers
- Stuff that deals with untrusted data
- Leaf functions that should be easy to test, but aren't because of C
- Algorithmic code?
- Brittle concurrent code?
- Sea-of-pointers that already works - probably not a good idea at first
- GUI code? Try it; help improve gtk-rs?
- Exported GObject APIs - wait a bit; in prototype stage right now.

My personal take-away

- I'm a library writer
- Disenchanted with C libraries for years
 - Learned helplessness?
- Rust felt empowering
- Rust refreshed my spirit
- Rust showed me a community that GNOME can learn from



Surprise!!!❤



Thanks

- Luciana Mena-Silva (my daughter) for drawings of Ferris
- Katrina Owen for the technique of narrating/highlighting code - <http://www.kytrinyx.com/talks/>
- Alberto Ruiz, Joaquín Rosales for the GNOME/Rust hackfest
- [#rust-beginners](http://irc.mozilla.org)
- [#nom](http://irc.mozilla.org)
- Sebastian Dröge for nom macros

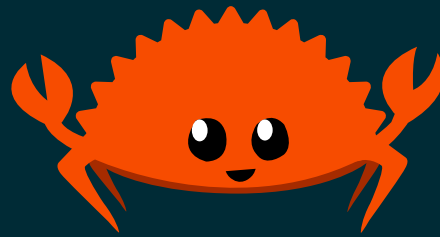
Blog posts

- Librsvg gets Rusty - <https://people.gnome.org/~federico/news-2016-10.html#25>
- Porting a few C functions to Rust - <https://people.gnome.org/~federico/news-2016-10.html#28>
- Bézier curves, markers, and SVG's concept of directionality - <https://people.gnome.org/~federico/news-2016-11.html#01>
- Refactoring C to make Rustification easier - <https://people.gnome.org/~federico/news-2016-11.html#03>
- Exposing Rust objects to C code - <https://people.gnome.org/~federico/news-2016-11.html#14>
- Debugging Rust code inside a C library - <https://people.gnome.org/~federico/news-2016-11.html#16>
- Reproducible font rendering for librsvg's tests - <https://people.gnome.org/~federico/news-2017-01.html#11>
- Algebraic data types in Rust, and basic parsing - <https://people.gnome.org/~federico/news-2017-02.html#03>
- How librsvg exports reference-counted objects from Rust to C - <https://people.gnome.org/~federico/news-2017-02.html#17>
- Griping about parsers and shitty specifications - <https://people.gnome.org/~federico/news-2017-02.html#24>
- Porting librsvg's tree of nodes to Rust - <https://people.gnome.org/~federico/news-2017-02.html#28>
- gboolean is not Rust bool - <https://people.gnome.org/~federico/news-2017-04.html#28>

Replacing C library code with Rust: What I learned with librsvg

Federico Mena Quintero
federico@gnome.org

GUADEC 2017
Manchester, UK



This is Ferris the Crustacean; she is the Rust mascot.

<http://www.rustacean.net/>

In 2017, can you imagine using a language that doesn't have a cute mascot?

What uses libsvg?

GNOME platform and desktop:

gtk+ indirectly through gdk-pixbuf

thumbnailer via gdk-pixbuf

eog

gstreamer-plugins-bad

Libsvg is an old library. We use it in many places in GNOME.

What uses libsvg?

Fine applications:

gnome-games (gnome-chess, five-or-more, etc.)

gimp

gcompris

claws-mail

darktable

And in applications.

What uses librsvg?

Desktop environments

mate-panel

Evas / Enlightenment

emacs-x11

And in desktops.

What uses libsvg?

Things you may not expect

ImageMagick

Wikipedia ← *they have been awesome*

And even in strange places. Libsvg is one of those “basic infrastructure” libraries.

Long History

- First commit, Eazel, 2001
- Experiment instead of building a DOM, stream in SVG by using libxml2's SAX parser
- Renderer used libart
- Gill → Sodipodi → Inkscape
- Librsvg was being written while the SVG spec was being written
- Ported to Cairo eventually
 - I'll remove the last libart-ism any day now

Librsvg was being written while the first SVG was being written. It was an experiment called Gill, to write an SVG parser using SAX streaming for XML, instead of building a DOM first.

Librsvg used libart, the first library in GNOME to provide anti-aliased vector rendering. Later we replaced it with Cairo.

Federico takes over

- Librsvg was mostly unmaintained in 2015
- Took over maintenance in February 2015
- Started the Rust port in October 2016

Librsvg was mostly unmaintained in 2015; kind people were committing patches every now and then.

I took over in February 2015, and started to port librsvg to Rust in October 2016.

Pain points: librsvg's CVEs

- CVE-2011-3146 - invalid cast of RsvgNode type, crash
- CVE-2015-7557 - out-of-bounds heap read in list-of-points
- CVE-2015-7558 - Infinite loop / stack overflow from cyclic element references (thanks to Benjamin Otte for the epic fix)
- CVE-2016-4348 - *NOT OUR PROBLEM* - integer overflow when writing PNGs in Cairo (rowstride computation)
- CVE-2017-11464 - Division by zero in Gaussian blur code (my bad, sorry)

- Many non-CVEs found through fuzz testing and random SVGs on the net

- <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=librsvg>
- CVE = Common Vulnerabilities and Exposures

Librsvg handles untrusted data from the Internet – SVGs that you download directly or indirectly. Thus, it needs to be secure and not let you get pwned if there's a bug.

We have had our fair share of CVEs – for an infrastructure library.

More CVEs

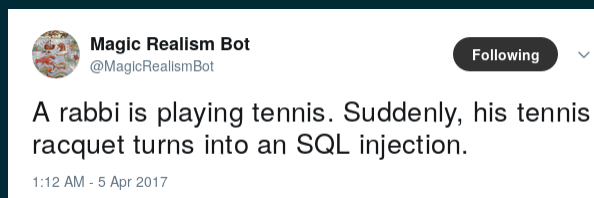
- **libcroco**
4 CVEs – out of bounds read, malloc() failure, two unfixed ones (!)
- **Cairo**
20 CVEs
- **libxml2 / libxml**
85 + 54 CVEs

Librsvg depends on libcroco for parsing CSS. It is unmaintained, and has outstanding CVEs as of July 2017.

We also depend on Cairo and libxml2; both are critical infrastructure and have had their fair share of security bugs.

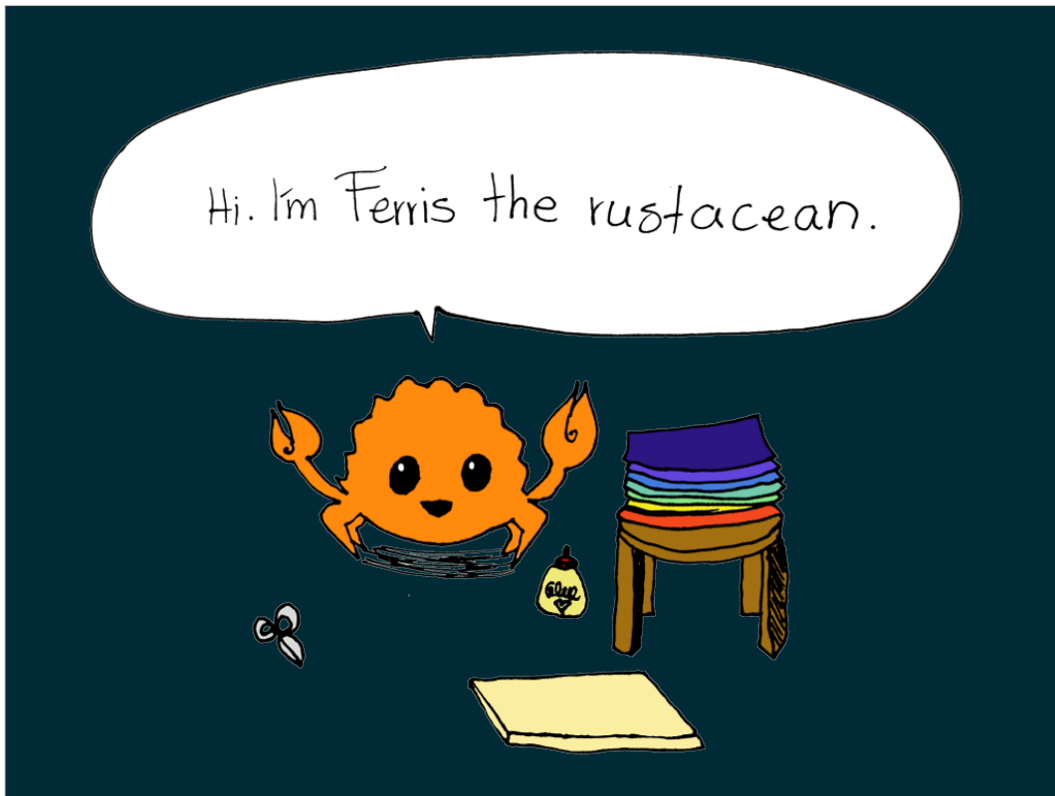
Pain points – so what?

- Best case for people – app crashes
- Worst case for people – pwned by baddies
- Worst case for robots – Denial-of-Service, pwned by baddies



Tweet from @MagicRealismBot:

<https://twitter.com/MagicRealismBot/status/849505117510520837>



I contracted out with my daughter Luciana to get some drawings of Ferris made. She was quite happy to get paid money for her drawings. Ferris will cut up bits of vectorial paper and assemble a nice picture for us.

The decision to Rustify

- Allison's overflow-checked arithmetic macros in glib
- **"Rust out your C"**
Carol Nichols
<https://github.com/carols10cents/rust-out-your-c-talk>
- **"Writing GStreamer elements in Rust"** Sebastian Dröge
<https://coaxion.net/blog/2016/05/writing-gstreamer-plugins-and-elements-in-rust/>

Allison Lortie committed some macros to Glib to allow for overflow-checked arithmetic. I was thinking of going through the librsvg source code to use them.

However, I ran into a couple of presentations on how to port C code bit by bit into Rust. I wasn't aware that this was possible! I thought Rust made you write a whole program or library in one shot. However, it is quite possible to port C code bit by bit and link it to the compiled Rust code, like if it were any other C code.

Disclaimers

- Librsvg is lower level than most GNOME libs
- Doesn't use GTK+
- Public API is a **single** GObject from C
- I'm a beginner/intermediate Rust programmer
- Don't use this talk as a source of good Rust practices
- Do talk to me about your porting effort!
 - *Ask me about encouragement for getting in trouble*
- Do not expect to learn a lot of Rust from this talk

- We are naturalists; I'm showing you critters I found

If you are getting ideas of porting any random GNOME C library to Rust, listen carefully.

Librsvg doesn't use most of the GNOME library infrastructure. We just depend on Glib, Cairo, libxml2, and a couple other low-level libraries.

Our public API is a **single** GObject with just a few methods. There is nothing exotic here. Your library may be more complex.

We won't have time to learn Rust in this talk; rather, I hope that you'll get inspired by what is possible.

Rust's killer features – for me

- Safe, modern language, blah blah blah
- Generates object code - .o / .a / .so
- Can be linked to C code
- No runtime, no garbage collection
- Can export and consume C ABI
- Makes it easy to write unit tests

While Rust is touted as a safe, modern language, with fearless concurrency and all that, there is one thing that particularly interests me:

Rust generates object code (.o files)

You can link that code to what comes out of a C compiler, just with the normal linker.

Therefore, you can port bits of C to Rust, link the compiled code, and callers have no idea that it is Rust instead of C.

Plus other nice features, like making it really easy to write unit tests... unlike C.

Easy to write unit tests

```
#[test]
fn check_that_it_works () {
    assert! (internal_function() == expected_value);
}

$ cargo test
...
running 105 tests
test aspect_ratio::tests::aligns ... ok
test aspect_ratio::tests::conversion_to_u32_roundtrips ... ok
test aspect_ratio::tests::parses_valid_strings ... ok
test aspect_ratio::tests::parsing_invalid_strings_yields_error ... ok

test result: ok. 104 passed; 0 failed; 1 ignored; 0 measured
```

Have you tried to set up unit tests with `g_test_*`? It's a pain in the ass. Every other C framework for unit tests is similar. You need to jump through hoops to compile/link your program so that only the tests will run. You need to make static functions accessible to the tests, which is hard. It's a very messy process.

Rust removes the friction. You mark a function as being a test, run "cargo test", and you are done.

How I started porting librsvg

- Port internals (algorithms / parsers) to Rust
- Write unit tests; complement with black-box tests with SVG files
 - Document the black-box test suite
 - Thanks again to Benjamin for writing that test suite
- Leave public C API intact, for now
- C API is consumed by GObject-Introspection

I started by porting algorithms and parsers in librsvg from C to Rust. It is reasonably easy to write unit tests for those.

Librsvg didn't have any unit tests! Just toplevel integration tests that ran the whole library as a black box, on example .svg files. But we needed finer-grained tests of the machinery.

The public C API is unchanged. It gets used by language bindings through the magic of GObject Introspection.

How to port a bit of C code

- Refactor C to make it Rust-friendly
- Add tests
- Port to Rust
- Test
- Refactor Rust to make it pretty

I've converged on more or less this strategy for porting a bit of C code to Rust.

Your initially Rustified code may not be very pretty or use “proper” Rust idioms. This can be refactored later. Rust refactors very nicely, as it has better language facilities and a better standard library than C.

Integrating into the build

- Librsvg uses autotools
- Rust uses cargo
- Build a librsvg_internals.a
- In Cargo.toml:

```
[lib]
name = "rsvg_internals"
crate-type = ["staticlib"]
```

```
librsvg/
configure.ac
Makefile.am
*.[ch]
```

```
rust/
Cargo.toml
src/
*.rs
target/
```

It wasn't too hard to integrate the Rust code, which is built with cargo, into the Autotools machinery that librsvg uses.

Look at the git repository for a reference on how to do it. I'll write a blog post about this.

Autotools thanks

- Hubert Figuière
 - <https://www.figuire.net/hub/blog/?2016/10/07/862-rust-and-automake>
- Luke Nukem
 - http://lukenukem.co.nz/gsoc/2017/05/17/gso_2.html
- Havoc Pennington
 - <https://blog.ometer.com/2017/01/10/dear-package-managers-dependency-resolution-results-should-be-in-version-control/>

Some people provided very good blog posts on using Rust with Autotools; thanks to them.

Progression

- Once some of the leaf code was ported...
- **PORT ALL THE THINGS**
- Especially the shitty C parsers
- Big goals, not yet realized:
 - Replace libcroco (CSS parser)
 - Replace little CSS engine with Servo's
 - Replace libxml2

Originally I wanted to port only the parsers and some of the algorithms.

But then I wanted to port EVERYTHING. Especially, to get rid of the scary C libraries like libcroco and libxml2. This is not done yet, but we'll get there.

Ideally, only the public C API will remain as-is.

Debugging Rust is awesome

- gdb, valgrind, rr all work fine
- Print anything with the Debug trait

```
#[derive(Debug)]
struct BigAssStruct {
    tons_of_fields: ...;
    some_more_fields: ...;
}

println! ("{:?}", my_struct);

println! ("{:#?}", my_struct); // pretty-print
```

Rust has excellent infrastructure for debugging. Gdb just works. Valgrind and rr just work. (protip: rr is magic and if you already know gdb, you already know how to use it and it will make you super-powerful).

You know the pain of writing `print_my_c_struct()` functions? Rust takes care of that automatically.

Error propagation

- Librsvg “handled” errors by reverting to default values (or garbage)

```
<rect x="bleh" y="5" width="eek".../>  
<path stroke="#000000" fill="#12wxyz"/>
```

- SVG spec says what to do
 - It doesn't agree with itself
 - Implementations don't agree with it
 - ㄟ(̄)ㄎ

One big problem in librsvg is that it didn't really have error handling. When it encountered an unparsable value, it sometimes had fallbacks to default “safe” values, or it just returned garbage to the caller.

The Rust port made it easy to add error handling and error propagation everywhere.

Evolution of set_atts()

- No error handling at first:

```
void (*set_atts) (RsvgNode *self, RsvgHandle *handle, RsvgPropertyBag *pbag);

static void
rsvg_node_svg_set_atts (RsvgNodeSvg *svg, RsvgHandle *handle,
                       RsvgPropertyBag *pbag)
{
    if ((value = rsvg_property_bag_lookup (pbag, "preserveAspectRatio")))
        svg->preserve_aspect_ratio = rsvg_aspect_ratio_parse (value);

    if ((value = rsvg_property_bag_lookup (pbag, "viewBox")))
        svg->vbox = rsvg_css_parse_vbox (value);
    ...
}
```

Let me show you a bit of how this code evolved.

Set_atts() is the method that parses the attributes for each SVG element. For example, here we parse the preserveAspectRatio and viewBox attributes.

Notice how the parser functions just return a value, with no way to indicate if there was an error.

```

enum MarkerUnits { UserSpaceOnUse, StrokeWidth };

impl Default for MarkerUnits {
    fn default () -> MarkerUnits { MarkerUnits::StrokeWidth }
}

impl FromStr for MarkerUnits {
    type Err = AttributeError;

    fn from_str (s: &str) -> Result <MarkerUnits, AttributeError> { ... }
}

impl NodeTrait for NodeMarker {
    fn set_atts (&self,
                _: &RsvgNode,
                _: *const RsvgHandle,
                pbag: *const RsvgPropertyBag) -> NodeResult {

        self.units.set (
            property_bag::parse_or_default (pbag, "markerUnits"?);
            ...
        )
    }
}

pub fn parse_or_default<T> (pbag: *const RsvgPropertyBag, key: &'static str) ->
    Result <T, NodeError>
    where T: Default + FromStr<Err = AttributeError>

```

In SVG, you can add things like arrows to the endpoints of lines by defining *markers*.

The default in the SVG spec is to make markers scale relative to the stroke width of the line, so we use the Default trait to indicate that.

```

enum MarkerUnits { UserSpaceOnUse, StrokeWidth };

impl Default for MarkerUnits {
    fn default () -> MarkerUnits { MarkerUnits::StrokeWidth }
}

impl FromStr for MarkerUnits {
    type Err = AttributeError;

    fn from_str (s: &str) -> Result <MarkerUnits, AttributeError> { ... }
}

impl NodeTrait for NodeMarker {
    fn set_atts (&self,
                _: &RsvgNode,
                _: *const RsvgHandle,
                pbag: *const RsvgPropertyBag) -> NodeResult {

        self.units.set (
            property_bag::parse_or_default (pbag, "markerUnits"?);
            ...
        )
    }
}

pub fn parse_or_default<T> (pbag: *const RsvgPropertyBag, key: &'static str) ->
    Result <T, NodeError>
    where T: Default + FromStr<Err = AttributeError>

```

We implement our parser for MarkerUnits using the FromStr trait. You can see that the from_str() function takes a &str slice, and returns a Result.

If the result is successful, we'll get a MarkerUnits enum as expected; if it's an error, we'll get an AttributeError.

```

enum MarkerUnits { UserSpaceOnUse, StrokeWidth };

impl Default for MarkerUnits {
    fn default () -> MarkerUnits { MarkerUnits::StrokeWidth }
}

impl FromStr for MarkerUnits {
    type Err = AttributeError;

    fn from_str (s: &str) -> Result <MarkerUnits, AttributeError> { ... }
}

impl NodeTrait for NodeMarker {
    fn set_atts (&self,
                _: &RsvgNode,
                _: *const RsvgHandle,
                pbag: *const RsvgPropertyBag) -> NodeResult {

        self.units.set (
            property_bag::parse_or_default (pbag, "markerUnits"?);
            ...
        )
    }
}

pub fn parse_or_default<T> (pbag: *const RsvgPropertyBag, key: &'static str) ->
    Result <T, NodeError>
    where T: Default + FromStr<Err = AttributeError>

```

Then, in our `set_atts()` implementation, we set the “units” field by calling `property_bag::parse_or_default()`. We just pass it the name of the SVG attribute.

If `parse_or_default()` returns an error, the question mark means that `set_atts()` will return that error. In Rust, you can propagate an error to the caller with a single character, which is shorter than the code you would require to actually ignore the error.

```

enum MarkerUnits { UserSpaceOnUse, StrokeWidth };

impl Default for MarkerUnits {
    fn default () -> MarkerUnits { MarkerUnits::StrokeWidth }
}

impl FromStr for MarkerUnits {
    type Err = AttributeError;

    fn from_str (s: &str) -> Result <MarkerUnits, AttributeError> { ... }
}

impl NodeTrait for NodeMarker {
    fn set_atts (&self,
                _: &RsvgNode,
                _: *const RsvgHandle,
                pbag: *const RsvgPropertyBag) -> NodeResult {

        self.units.set (
            property_bag::parse_or_default (pbag, "markerUnits"?);
            ...
        )
    }
}

pub fn parse_or_default<T> (pbag: *const RsvgPropertyBag, key: &'static str) ->
    Result <T, NodeError>
    where T: Default + FromStr<Err = AttributeError>

```

parse_or_default() is a generic function. It will look up a key/value pair in the property bag, parse the value as a type T, and return a result with a T or a NodeError.

```

enum MarkerUnits { UserSpaceOnUse, StrokeWidth };

impl Default for MarkerUnits {
    fn default () -> MarkerUnits { MarkerUnits::StrokeWidth }
}

impl FromStr for MarkerUnits {
    type Err = AttributeError;

    fn from_str (s: &str) -> Result <MarkerUnits, AttributeError> { ... }
}

impl NodeTrait for NodeMarker {
    fn set_atts (&self,
                _: &RsvgNode,
                _: *const RsvgHandle,
                pbag: *const RsvgPropertyBag) -> NodeResult {

        self.units.set (
            property_bag::parse_or_default (pbag, "markerUnits"?);
            ...
        )
    }
}

pub fn parse_or_default<T> (pbag: *const RsvgPropertyBag, key: &'static str) ->
    Result <T, NodeError>
    where T: Default + FromStr<Err = AttributeError>

```

A T value must know how to generate its default, and it must implement FromStr. The FromStr parser must return an AttributeError.

This kind of code, where you say generically what types of things you can handle, but you constrain their expected behavior, is very very powerful.

Expose stuff from C to Rust

The unsafe / “familiar” way

- Declare parallel structs/unions with `#[repr(C)]`
- Call C functions; dereference pointers they send back
- All dereferences marked as `unsafe{}`
- Quick code to write; easiest if you cut&paste from C
- You’ll probably end up with safe Rust wrappers over `unsafe{}` code
- You probably need this if you are plugging into “unchangeable” C code

You can try to use Rust as if it were C: you can dereference raw pointers and mark those as “unsafe”. This is a quick way to port code to Rust, but it’s not very pretty.

Expose stuff from C to Rust The safe way

- Create a C API, possibly internal, that is Rust-friendly
- Or refactor the relevant code to be Rust-friendly
- Expose only opaque pointers to structs
- Have C functions with setters/getters for those structs; call them from Rust
- Still need to mark calls as `unsafe{}`
- I like this way if you are completely free to change the C code
- Leave the public API intact; change only the internals

The cleaner/safer way to do it is to refactor and write the necessary wrappers.

Expose stuff from Rust to C

- Parallel structs with `#[repr(C)]`
- Expose opaque pointers to structs, and functions to frob them
- Get the memory management right
- `new()` / `destroy()` - <https://people.gnome.org/~federico/news-2016-11.html#14>
- Reference counting - <https://people.gnome.org/~federico/news-2017-02.html#17>

To expose stuff from Rust to C, it's probably better to just define an internal API in Rust that is friendly to C.

Be sure to get the memory management right. If you use a `new()/destroy()` pattern, that's a blog post you can use as a reference. If you want to expose Rust reference counting to C, that's another blog post about it.

Parsers

Let's talk about parsers. C is infamous for making it easy to write buggy, insecure parsers.

Bad C parser for #rrggbb hex colors

```
guint32
rsvg_css_parse_color (const char *str, ...)
{
    gint val = 0;

    if (str[0] == '#') {
        int i;
        for (i = 1; str[i]; i++) {
            int hexval;
            if (str[i] >= '0' && str[i] <= '9')
                hexval = str[i] - '0';
            else if (str[i] >= 'A' && str[i] <= 'F')
                hexval = str[i] - 'A' + 10;
            else if (str[i] >= 'a' && str[i] <= 'f')
                hexval = str[i] - 'a' + 10;
            else
                break;
            val = (val << 4) + hexval;
        }
        /* handle #rgb case */
        if (i == 4) {
            val = ((val & 0xf00) << 8) | ((val & 0x0f0) << 4) |
                (val & 0x00f);
            val |= val << 4;
        }
        val |= 0xff000000; /* opaque */
    }
}
```

Let's look at a bad parser that we had for parsing hex colors.

Bad C parser for #rrggbb hex colors

```
uint32
rsvg_css_parse_color (const char *str, ...)
{
    gint val = 0;

    if (str[0] == '#') {
        int i;
        for (i = 1; str[i]; i++) {
            int hexval;
            if (str[i] >= '0' && str[i] <= '9')
                hexval = str[i] - '0';
            else if (str[i] >= 'A' && str[i] <= 'F')
                hexval = str[i] - 'A' + 10;
            else if (str[i] >= 'a' && str[i] <= 'f')
                hexval = str[i] - 'a' + 10;
            else
                break;
            val = (val << 4) + hexval;
        }
        /* handle #rgb case */
        if (i == 4) {
            val = ((val & 0xf00) << 8) | ((val & 0x0f0) << 4) |
                (val & 0x00f);
            val |= val << 4;
        }

        val |= 0xff000000; /* opaque */
    }
}
```

We start with a value of zero.

Bad C parser for #rrggbb hex colors

```
uint32
rsvg_css_parse_color (const char *str, ...)
{
    gint val = 0;

    if (str[0] == '#') {
        int i;
        for (i = 1; str[i]; i++) {
            int hexval;
            if (str[i] >= '0' && str[i] <= '9')
                hexval = str[i] - '0';
            else if (str[i] >= 'A' && str[i] <= 'F')
                hexval = str[i] - 'A' + 10;
            else if (str[i] >= 'a' && str[i] <= 'f')
                hexval = str[i] - 'a' + 10;
            else
                break;
            val = (val << 4) + hexval;
        }
        /* handle #rgb case */
        if (i == 4) {
            val = ((val & 0xf00) << 8) | ((val & 0x0f0) << 4) |
                (val & 0x00f);
            val |= val << 4;
        }

        val |= 0xff000000; /* opaque */
    }
}
```

If the first character in the string is a hash,

Bad C parser for #rrggbb hex colors

```
uint32
rsvg_css_parse_color (const char *str, ...)
{
    gint val = 0;

    if (str[0] == '#') {
        int i;
        for (i = 1; str[i]; i++) {
            int hexval;
            if (str[i] >= '0' && str[i] <= '9')
                hexval = str[i] - '0';
            else if (str[i] >= 'A' && str[i] <= 'F')
                hexval = str[i] - 'A' + 10;
            else if (str[i] >= 'a' && str[i] <= 'f')
                hexval = str[i] - 'a' + 10;
            else
                break;
            val = (val << 4) + hexval;
        }
        /* handle #rgb case */
        if (i == 4) {
            val = ((val & 0xf00) << 8) | ((val & 0x0f0) << 4) |
                (val & 0x00f);
            val |= val << 4;
        }

        val |= 0xff000000; /* opaque */
    }
}
```

Then for each of the remaining characters,

Bad C parser for #rrggbb hex colors

```
uint32
rsvg_css_parse_color (const char *str, ...)
{
    gint val = 0;

    if (str[0] == '#') {
        int i;
        for (i = 1; str[i]; i++) {
            int hexval;
            if (str[i] >= '0' && str[i] <= '9')
                hexval = str[i] - '0';
            else if (str[i] >= 'A' && str[i] <= 'F')
                hexval = str[i] - 'A' + 10;
            else if (str[i] >= 'a' && str[i] <= 'f')
                hexval = str[i] - 'a' + 10;
            else
                break;
            val = (val << 4) + hexval;
        }
        /* handle #rgb case */
        if (i == 4) {
            val = ((val & 0xf00) << 8) | ((val & 0x0f0) << 4) |
                (val & 0x00f);
            val |= val << 4;
        }

        val |= 0xff000000; /* opaque */
    }
}
```

If the character is a decimal digit, turn it into a number

Bad C parser for #rrggbb hex colors

```
uint32
rsvg_css_parse_color (const char *str, ...)
{
    gint val = 0;

    if (str[0] == '#') {
        int i;
        for (i = 1; str[i]; i++) {
            int hexval;
            if (str[i] >= '0' && str[i] <= '9')
                hexval = str[i] - '0';
            else if (str[i] >= 'A' && str[i] <= 'F')
                hexval = str[i] - 'A' + 10;
            else if (str[i] >= 'a' && str[i] <= 'f')
                hexval = str[i] - 'a' + 10;
            else
                break;
            val = (val << 4) + hexval;
        }
        /* handle #rgb case */
        if (i == 4) {
            val = ((val & 0xf00) << 8) | ((val & 0x0f0) << 4) |
                (val & 0x00f);
            val |= val << 4;
        }

        val |= 0xff000000; /* opaque */
    }
}
```

If it's a hexadecimal digit, turn it into a number,

Bad C parser for #rrggbb hex colors

```
uint32
rsvg_css_parse_color (const char *str, ...)
{
    gint val = 0;

    if (str[0] == '#') {
        int i;
        for (i = 1; str[i]; i++) {
            int hexval;
            if (str[i] >= '0' && str[i] <= '9')
                hexval = str[i] - '0';
            else if (str[i] >= 'A' && str[i] <= 'F')
                hexval = str[i] - 'A' + 10;
            else if (str[i] >= 'a' && str[i] <= 'f')
                hexval = str[i] - 'a' + 10;
            else
                break;
            val = (val << 4) + hexval;
        }
        /* handle #rgb case */
        if (i == 4) {
            val = ((val & 0xf00) << 8) | ((val & 0x0f0) << 4) |
                (val & 0x00f);
            val |= val << 4;
        }

        val |= 0xff000000; /* opaque */
    }
}
```

If it's an invalid character, "break" the loop; we'll see what happens later.

Bad C parser for #rrggbb hex colors

```
uint32
rsvg_css_parse_color (const char *str, ...)
{
    gint val = 0;

    if (str[0] == '#') {
        int i;
        for (i = 1; str[i]; i++) {
            int hexval;
            if (str[i] >= '0' && str[i] <= '9')
                hexval = str[i] - '0';
            else if (str[i] >= 'A' && str[i] <= 'F')
                hexval = str[i] - 'A' + 10;
            else if (str[i] >= 'a' && str[i] <= 'f')
                hexval = str[i] - 'a' + 10;
            else
                break;
            val = (val << 4) + hexval;
        }
        /* handle #rgb case */
        if (i == 4) {
            val = ((val & 0xf00) << 8) | ((val & 0x0f0) << 4) |
                (val & 0x00f);
            val |= val << 4;
        }

        val |= 0xff000000; /* opaque */
    }
}
```

Shift the whole color value to the left and add in the value of the new hex digit.

Bad C parser for #rrggbb hex colors

```
uint32
rsvg_css_parse_color (const char *str, ...)
{
    gint val = 0;

    if (str[0] == '#') {
        int i;
        for (i = 1; str[i]; i++) {
            int hexval;
            if (str[i] >= '0' && str[i] <= '9')
                hexval = str[i] - '0';
            else if (str[i] >= 'A' && str[i] <= 'F')
                hexval = str[i] - 'A' + 10;
            else if (str[i] >= 'a' && str[i] <= 'f')
                hexval = str[i] - 'a' + 10;
            else
                break;
            val = (val << 4) + hexval;
        }
        /* handle #rgb case */
        if (i == 4) {
            val = ((val & 0xf00) << 8) | ((val & 0x0f0) << 4) |
                (val & 0x00f);
            val |= val << 4;
        }

        val |= 0xff000000; /* opaque */
    }
}
```

If we exited the loop and it turns out we only had 4 characters, it's something like "#123", with four bits per channel, so turn that into 8 bits per channel.

Bad C parser for #rrggbb hex colors

```
uint32
rsvg_css_parse_color (const char *str, ...)
{
    gint val = 0;

    if (str[0] == '#') {
        int i;
        for (i = 1; str[i]; i++) {
            int hexval;
            if (str[i] >= '0' && str[i] <= '9')
                hexval = str[i] - '0';
            else if (str[i] >= 'A' && str[i] <= 'F')
                hexval = str[i] - 'A' + 10;
            else if (str[i] >= 'a' && str[i] <= 'f')
                hexval = str[i] - 'a' + 10;
            else
                break;
            val = (val << 4) + hexval;
        }
        /* handle #rgb case */
        if (i == 4) {
            val = ((val & 0xf00) << 8) | ((val & 0x0f0) << 4) |
                (val & 0x00f);
            val |= val << 4;
        }

        val |= 0xff000000; /* opaque */
    }
}
```

Finally, set the alpha to fully opaque.

Bad C parser for #rrggbb hex colors

```
uint32
rsvg_css_parse_color (const char *str, ...)
{
    gint val = 0;

    if (str[0] == '#') {
        int i;
        for (i = 1; str[i]; i++) {
            int hexval;
            if (str[i] >= '0' && str[i] <= '9')
                hexval = str[i] - '0';
            else if (str[i] >= 'A' && str[i] <= 'F')
                hexval = str[i] - 'A' + 10;
            else if (str[i] >= 'a' && str[i] <= 'f')
                hexval = str[i] - 'a' + 10;
            else
                break;
            val = (val << 4) + hexval;
        }
        /* handle #rgb case */
        if (i == 4) {
            val = ((val & 0xf00) << 8) | ((val & 0x0f0) << 4) |
                (val & 0x00f);
            val |= val << 4;
        }

        val |= 0xff000000; /* opaque */
    }
}
```

Let's go back to the error case. What if we got an invalid hex digit? We break out of the loop...

Bad C parser for #rrggbb hex colors

```
guint32
rsvg_css_parse_color (const char *str, ...)
{
    gint val = 0;

    if (str[0] == '#') {
        int i;
        for (i = 1; str[i]; i++) {
            int hexval;
            if (str[i] >= '0' && str[i] <= '9')
                hexval = str[i] - '0';
            else if (str[i] >= 'A' && str[i] <= 'F')
                hexval = str[i] - 'A' + 10;
            else if (str[i] >= 'a' && str[i] <= 'f')
                hexval = str[i] - 'a' + 10;
            else
                break;
            val = (val << 4) + hexval;
        }
        /* handle #rgb case */
        if (i == 4) {
            val = ((val & 0xf00) << 8) | ((val & 0x0f0) << 4) |
                (val & 0x00f);
            val |= val << 4;
        }
        val |= 0xff000000; /* opaque */
    }
}
```

And that takes us to unrelated code, so we will compute a garbage value.

If we get a super long string, so that instead of 6 hex digits it has 20, we'll get integer overflow and a garbage value.

If we get something like “#12”, with too few characters, we generate a garbage value.

Parse a hex digit in Rust

```
fn from_hex(c: u8) -> Result<u8, ()> {  
    match c {  
        b'0' ... b'9' => Ok(c - b'0'),  
        b'a' ... b'f' => Ok(c - b'a' + 10),  
        b'A' ... b'F' => Ok(c - b'A' + 10),  
        _ => Err(())  
    }  
}
```

How do we do this properly in Rust?

This is actual code from `rust-cssparser`, a library that is used in Mozilla's Servo, the web rendering engine that will replace the one in Firefox.

It is also the code that is used in `librsvg` now.

This `from_hex()` function parses a single hex digit.

Parse a hex digit in Rust

```
fn from_hex(c: u8) -> Result<u8, ()> {  
    match c {  
        b'0' ... b'9' => Ok(c - b'0'),  
        b'a' ... b'f' => Ok(c - b'a' + 10),  
        b'A' ... b'F' => Ok(c - b'A' + 10),  
        _ => Err(())  
    }  
}
```

It returns a Result with a u8 byte on success, and an “empty” error on failure.

The code is easy: any character except a hex digit gives back an error.

Parse a hex color in Rust

```
impl Color {
    pub fn parse_hash(value: &[u8]) -> Result<Self, ()> {
        match value.len() {
            8 => Ok(rgba(
                from_hex(value[0])? * 16 + from_hex(value[1])?,
                from_hex(value[2])? * 16 + from_hex(value[3])?,
                from_hex(value[4])? * 16 + from_hex(value[5])?,
                from_hex(value[6])? * 16 + from_hex(value[7])?,
            )),
            6 => Ok(rgb(...)),
            4 => Ok(rgb(...)),
            3 => Ok(rgb(...)),
            _ => Err(())
        }
    }
}
```

This is the `parse_hash()` function that parses a hex color. The initial “#” character has already been parsed.

Parse a hex color in Rust

```
impl Color {
    pub fn parse_hash(value: &[u8]) -> Result<Self, ()> {
        match value.len() {
            8 => Ok(rgba(
                from_hex(value[0])? * 16 + from_hex(value[1])?,
                from_hex(value[2])? * 16 + from_hex(value[3])?,
                from_hex(value[4])? * 16 + from_hex(value[5])?,
                from_hex(value[6])? * 16 + from_hex(value[7])?,
            )),
            6 => Ok(rgb(...)),
            4 => Ok(rgb(...)),
            3 => Ok(rgb(...)),
            _ => Err(())
        }
    }
}
```

Depending on the length of the string of digits,

Parse a hex color in Rust

```
impl Color {
    pub fn parse_hash(value: &[u8]) -> Result<Self, ()> {
        match value.len() {
            8 => Ok(rgba(
                from_hex(value[0])? * 16 + from_hex(value[1])?,
                from_hex(value[2])? * 16 + from_hex(value[3])?,
                from_hex(value[4])? * 16 + from_hex(value[5])?,
                from_hex(value[6])? * 16 + from_hex(value[7])?,
            )),
            6 => Ok(rgb(...)),
            4 => Ok(rgb(...)),
            3 => Ok(rgb(...)),
            _ => Err(())
        }
    }
}
```

If we have 8 digits, it is #rrggbbaa. Parse it by calling `from_hex()` on each character and doing arithmetic.

Parse a hex color in Rust

```
impl Color {
    pub fn parse_hash(value: &[u8]) -> Result<Self, ()> {
        match value.len() {
            8 => Ok(rgba(
                from_hex(value[0])? * 16 + from_hex(value[1])?,
                from_hex(value[2])? * 16 + from_hex(value[3])?,
                from_hex(value[4])? * 16 + from_hex(value[5])?,
                from_hex(value[6])? * 16 + from_hex(value[7])?,
            )),
            6 => Ok(rgb(...)),
            4 => Ok(rgb(...)),
            3 => Ok(rgb(...)),
            _ => Err(())
        }
    }
}
```

If it's 6 digits, or 4, or 3, do something similar.

Parse a hex color in Rust

```
impl Color {
    pub fn parse_hash(value: &[u8]) -> Result<Self, ()> {
        match value.len() {
            8 => Ok(rgba(
                from_hex(value[0])? * 16 + from_hex(value[1])?,
                from_hex(value[2])? * 16 + from_hex(value[3])?,
                from_hex(value[4])? * 16 + from_hex(value[5])?,
                from_hex(value[6])? * 16 + from_hex(value[7])?,
            )),
            6 => Ok(rgb(...)),
            4 => Ok(rgb(...)),
            3 => Ok(rgb(...)),
            _ => Err(())
        }
    }
}
```

Any other length is invalid, so we return an error.

Parse a hex color in Rust

```
impl Color {
    pub fn parse_hash(value: &[u8]) -> Result<Self, ()> {
        match value.len() {
            8 => Ok(rgba(
                from_hex(value[0])? * 16 + from_hex(value[1])?,
                from_hex(value[2])? * 16 + from_hex(value[3])?,
                from_hex(value[4])? * 16 + from_hex(value[5])?,
                from_hex(value[6])? * 16 + from_hex(value[7])?,
            )),
            6 => Ok(rgb(...)),
            4 => Ok(rgb(...)),
            3 => Ok(rgb(...)),
            _ => Err(())
        }
    }
}
```

Notice the little question marks.

Those mean, “if the previous expression resulted in an error, return from the current function and return that error”.

This is shorter to type than what you would require to ignore the error and just get the value in case of success. Rust has made it possible to write error-checking code that is **more legible** than code that ignores errors. I think that’s quite an accomplishment.

It's not bad programmers, it's that C is a hostile language

- I did a “git blame” to see who wrote and modified the shitty parsers.
- Three or four of the best programmers GNOME has ever had.
- I know they wouldn't write *that* from scratch.
- The C language just makes it hard to do the right thing in the beginning, and makes refactoring hard.

I'm thinking that C is actively hostile to writing and maintaining reliable code. It feels like you have to change EVERYTHING if you want to re-do error handling code paths, for example.

What if the language provided good facilities for this?
Rust does!

Hand-written parsers in Rust

- No more awkward than in any other language
- Safe
- Can be made fast – zero-copy with slices and iterators
- Rust's `String.split()` family is AWESOME

```
let mut iter = "hello world".split_whitespace ();  
while let Some (word) = iter.next () {  
    println! ("{}", word);  
}
```

I have rewritten most of the parsers for SVG's mini-languages in librsvg.

Parsers in Rust are as awkward to write as in any other language. However, Rust makes it safe, and provides nice tools.

Combinator parsers with nom

```
// parse something like
//
// 5.2em
named! (parse_number_and_units<(f64, &[u8])>,
        tuple! (double,
                take_while! (is_alphabetic)));
```

- Done with macros
- People seem very happy with nom for parsing binary files?

Nom is a library to do “combinator parsers” using Rust macros.

Combinator parsers are a beautiful idea from functional languages. I want to like them, but didn’t have a good time with nom for parsing strings.

However, people seem to use nom very productively to parse binary files. Maybe it’s better suited for that?

Yacc-like parsers with lalrpop

```
Transform: cairo::Matrix = {
    Matrix,
    Translate,
    Scale,
    Rotate,
    SkewX,
    SkewY
};

Scale: cairo::Matrix = {
    // scale (42)
    "scale" "(" <Num> ")" => cairo::Matrix::new (<>, 0.0, 0.0, <>, 0.0, 0.0),

    // scale (3.5, 4.6)
    "scale" "(" <sx: Num> maybe_comma <sy: Num> ")" =>
        cairo::Matrix::new (sx, 0.0, 0.0, sy, 0.0, 0.0),
};

Matrix: ...
Rotate: ...
SkewX: ...

// regex for floating-point number
pub Num: f64 = <s:r"[+-]?([0-9]*\.[0-9]+|[0-9]+(\.[0-9]*)?)" <[Ee][+-]?[0-9]+)?">
=>
    f64::from_str (s).unwrap ();

maybe_comma: () = ", " ?;
```

If you are familiar with yacc or bison, lalrpop is a similar parser generator for Rust.

You define a grammar made up of productions.

Here, parsing an SVG Transform will return a `cairo::Matrix` or an `Error` (the error is implied).

Yacc-like parsers with lalrpop

```
Transform: cairo::Matrix = {
    Matrix,
    Translate,
    Scale,
    Rotate,
    SkewX,
    SkewY
};

Scale: cairo::Matrix = {
    // scale (42)
    "scale" "(" <Num> ")" => cairo::Matrix::new (<>, 0.0, 0.0, <>, 0.0, 0.0),

    // scale (3.5, 4.6)
    "scale" "(" <sx: Num> maybe_comma <sy: Num> ")" =>
        cairo::Matrix::new (sx, 0.0, 0.0, sy, 0.0, 0.0),
};

Matrix: ...
Rotate: ...
SkewX: ...

// regex for floating-point number
pub Num: f64 = <s:r"[+-]?([0-9]*\.[0-9]+|[0-9]+\.[0-9]*)?([Ee][+-]?[0-9]+)?">
=>
    f64::from_str (s).unwrap ();

maybe_comma: () = ", "?;
```

A Transform can be an explicit Matrix, or more convenient transformations like Translate, Scale, Rotate, etc.

Yacc-like parsers with lalrpop

```
Transform: cairo::Matrix = {
    Matrix,
    Translate,
    Scale,
    Rotate,
    SkewX,
    SkewY
};

Scale: cairo::Matrix = {
    // scale (42)
    "scale" "(" <Num> ")" => cairo::Matrix::new (<>, 0.0, 0.0, <>, 0.0, 0.0),

    // scale (3.5, 4.6)
    "scale" "(" <sx: Num> maybe_comma <sy: Num> ")" =>
        cairo::Matrix::new (sx, 0.0, 0.0, sy, 0.0, 0.0),
};

Matrix: ...
Rotate: ...
SkewX: ...

// regex for floating-point number
pub Num: f64 = <s:r"[+-]?([0-9]*\.[0-9]+|[0-9]+\.[0-9]*)?([Ee][+-]?[0-9]+)?">
=>
    f64::from_str (s).unwrap ();

maybe_comma: () = ", "?;
```

A Scale transform also returns a `cairo::Matrix`...

Yacc-like parsers with lalrpop

```
Transform: cairo::Matrix = {
    Matrix,
    Translate,
    Scale,
    Rotate,
    SkewX,
    SkewY
};

Scale: cairo::Matrix = {
    // scale (42)
    "scale" "(" <Num> ")" => cairo::Matrix::new (<>, 0.0, 0.0, <>, 0.0, 0.0),

    // scale (3.5, 4.6)
    "scale" "(" <sx: Num> maybe_comma <sy: Num> ")" =>
        cairo::Matrix::new (sx, 0.0, 0.0, sy, 0.0, 0.0),
};

Matrix: ...
Rotate: ...
SkewX: ...

// regex for floating-point number
pub Num: f64 = <s:r"[+-]?([0-9]*\.[0-9]+|[0-9]+(\.[0-9]*)?)" <Ee>[+-]?[0-9]+"?>
=>
    f64::from_str (s).unwrap ();

maybe_comma: () = ", "?;
```

... and SVG lets us specify a Scale in two ways: with a single number, which scales things proportionally in the x/y directions.

The parser will recognize a “scale” string, whitespace, a left paren, a Num which we will define below, and a right paren.

On the right side of the production there’s the Rust code to create a `cairo::Matrix` with the value parsed from the Num: lalrpop substitutes the number for the angle brackets.

Yacc-like parsers with lalrpop

```
Transform: cairo::Matrix = {
    Matrix,
    Translate,
    Scale,
    Rotate,
    SkewX,
    SkewY
};

Scale: cairo::Matrix = {
    // scale (42)
    "scale" "(" <Num> ")" => cairo::Matrix::new (<>, 0.0, 0.0, <>, 0.0, 0.0),

    // scale (3.5, 4.6)
    "scale" "(" <sx: Num> maybe_comma <sy: Num> ")" =>
        cairo::Matrix::new (sx, 0.0, 0.0, sy, 0.0, 0.0),
};

Matrix: ...
Rotate: ...
SkewX: ...

// regex for floating-point number
pub Num: f64 = <s:r"[+-]?([0-9]*\.[0-9]+|[0-9]+(\.[0-9]*)?)" <[Ee][+-]?[0-9]+>?">
=>
    f64::from_str (s).unwrap ();

maybe_comma: () = ", " ?;
```

The other case is where you specify different scaling factor for x and y.

Here we will parse two Nums, one being called sx and the other one sy, and expand them into the code.

Yacc-like parsers with lalrpop

```
Transform: cairo::Matrix = {
    Matrix,
    Translate,
    Scale,
    Rotate,
    SkewX,
    SkewY
};

Scale: cairo::Matrix = {
    // scale (42)
    "scale" "(" <Num> ")" => cairo::Matrix::new (<>, 0.0, 0.0, <>, 0.0, 0.0),

    // scale (3.5, 4.6)
    "scale" "(" <sx: Num> maybe_comma <sy: Num> ")" =>
        cairo::Matrix::new (sx, 0.0, 0.0, sy, 0.0, 0.0),
};

Matrix: ...
Rotate: ...
SkewX: ...

// regex for floating-point number
pub Num: f64 = <s:r"[+-]?([0-9]*\.[0-9]+|[0-9]+\.[0-9]*)?([Ee][+-]?[0-9]+)?">
=>
    f64::from_str (s).unwrap ();

maybe_comma: () = ", "?;
```

Here are the other productions for transformations with Matrix, Rotate, SkewX, etc.

Yacc-like parsers with lalrpop

```
Transform: cairo::Matrix = {
    Matrix,
    Translate,
    Scale,
    Rotate,
    SkewX,
    SkewY
};

Scale: cairo::Matrix = {
    // scale (42)
    "scale" "(" <Num> ")" => cairo::Matrix::new (<>, 0.0, 0.0, <>, 0.0, 0.0),

    // scale (3.5, 4.6)
    "scale" "(" <sx: Num> maybe_comma <sy: Num> ")" =>
        cairo::Matrix::new (sx, 0.0, 0.0, sy, 0.0, 0.0),
};

Matrix: ...
Rotate: ...
SkewX: ...

// regex for floating-point number
pub Num: f64 = <s:r"[+-]?([0-9]*\.[0-9]+|[0-9]+(\.[0-9]*)?)([Ee][+-]?[0-9]+)?">
=>
    f64::from_str (s).unwrap ();

maybe_comma: () = ", "?;
```

Finally we have a production for Num, which matches a regular expression for a floating-point number. We convert it to a double with `f64::from_str()`. We can `unwrap()` because the regex already checked for valid syntax for a floating-point number.

Yacc-like parsers with lalrpop

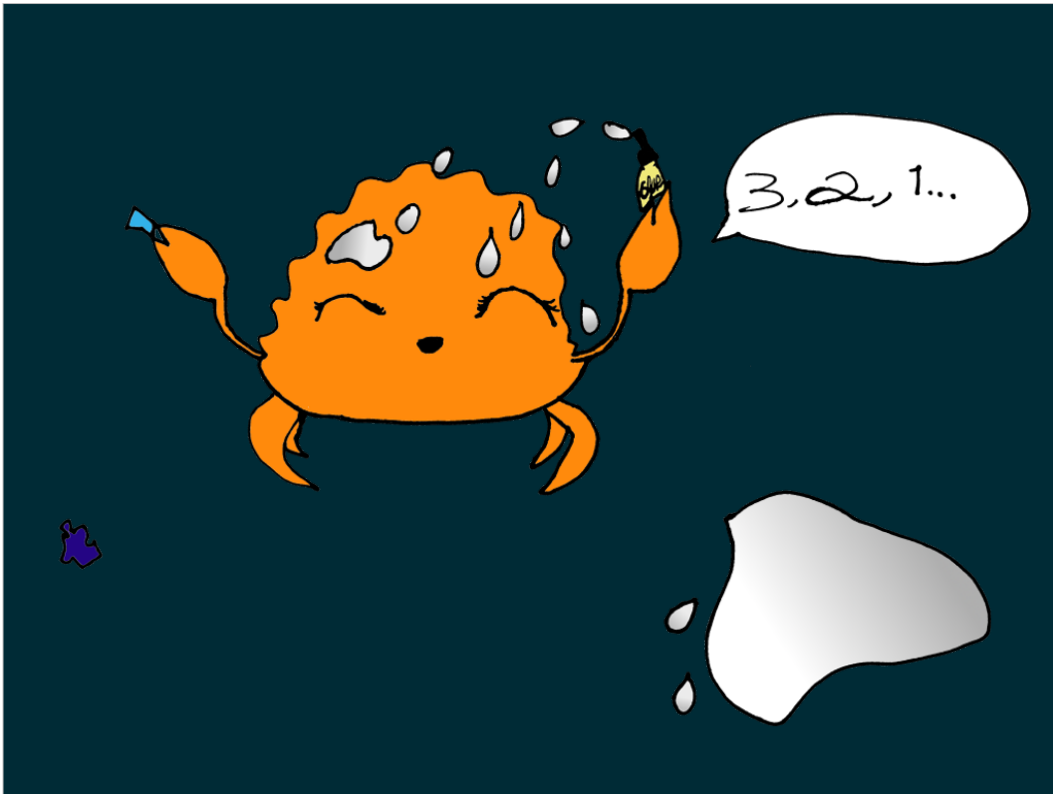
- Great documentation
- Possible to define one's own lexer
- Supports error recovery
- Love all around
- Generates big, fast parsers

I had a lovely time with lalrpop. If you are used to yacc-like parsers, I highly recommend it.

Many other parsing libraries

- Rust-cssparser
- Serde (serialization / deserialization) is great
- Shop around!

There are many libraries to write parsers with Rust. Do your shopping, and I hope you find something comfortable!



The most fun part of making something is using all the glue.

When Rust clarifies your thinking

```
if (!cairo_matrix_invert (...))  
    return;
```

- Degenerate matrices were ignored
- They were actually real bugs:
 - Uninitialized data
 - Invalid floats with all-bits-zero
 - Malformed SVG files – no data validation
- Cairo-rs calls panic!() on degenerate matrices
 - Added try_invert(); love all around

Rust has a mindset about pervasive correctness, and beautiful, powerful idioms.

Librsvg's had code to simply bail out if it found a degenerate matrix, one that cannot be inverted. At first I thought, "no biggie, they must be a fact of life in SVG".

They turned out to be real bugs, due to uninitialized data, or all-bits-zero floats (which is not 0.0), or plainly malformed SVG files that we were not handling properly.

Adding error handling for this in Rust clarified my thinking on how it should really work.

Can distros ship this?

- “cargo vendor”
- Haven’t done this yet
- Firefox 53+ will help here

How can Linux distros ship code which depends on Rust? They don’t want to let “cargo build” download dependencies at build time.

“cargo vendor” is a solution. It lets maintainers bundle their Rust dependencies along with their code, and related tools should let distros tell cargo that it should look for dependencies in `$distro_specific_path`.

I haven’t done this as of July 2017.

Distros will learn. After all, we haven’t had a new systems-level language since C++ got introduced. Firefox 53 now requires Rust, so distros are very much compelled to Deal With It.

Self-sanity

- Rust is not easy to learn
- New concepts (ownership tracking, functional paradigms, big syntax, etc.)
- Learning stage: “fighting the borrow checker”
- C is a sea of aliased pointers;
Rust Does Not Do That™

Rust is a big commitment, and it is not easy to learn if you are used to C’s way of working. It has some completely new concepts like ownership tracking, and the whole “borrow checker” in the compiler. Neither C nor garbage-collected languages made you think in that way!

Rust does not like big salads of pointers: it forces you to be very clear about ownership. It may be hard to get used to this way of thinking, but it will “click” one day and you’ll get it.

Self-sanity

- You won't know how to port things, or what crates to reuse
- It's OK not to use the best idioms in the beginning. Learn some more; refactor later.
- Tests are super important!
- Learn the error handling scheme: `Result<Ok, Err>`
- The Rust community is AWESOME.

Don't fear the initial uncertainty. Fight impostor syndrome. It's a big learning curve, but the Rust community is awesomely helpful.

Should we replace C library code with Rust?

- Parsers
- Stuff that deals with untrusted data
- Leaf functions that should be easy to test, but aren't because of C
- Algorithmic code?
- Brittle concurrent code?
- Sea-of-pointers that already works - probably not a good idea at first
- GUI code? Try it; help improve gtk-rs?
- Exported GObject APIs - wait a bit; in prototype stage right now.

What if you want to replace C library code in GNOME with Rust code? Some recommendations.

My personal take-away

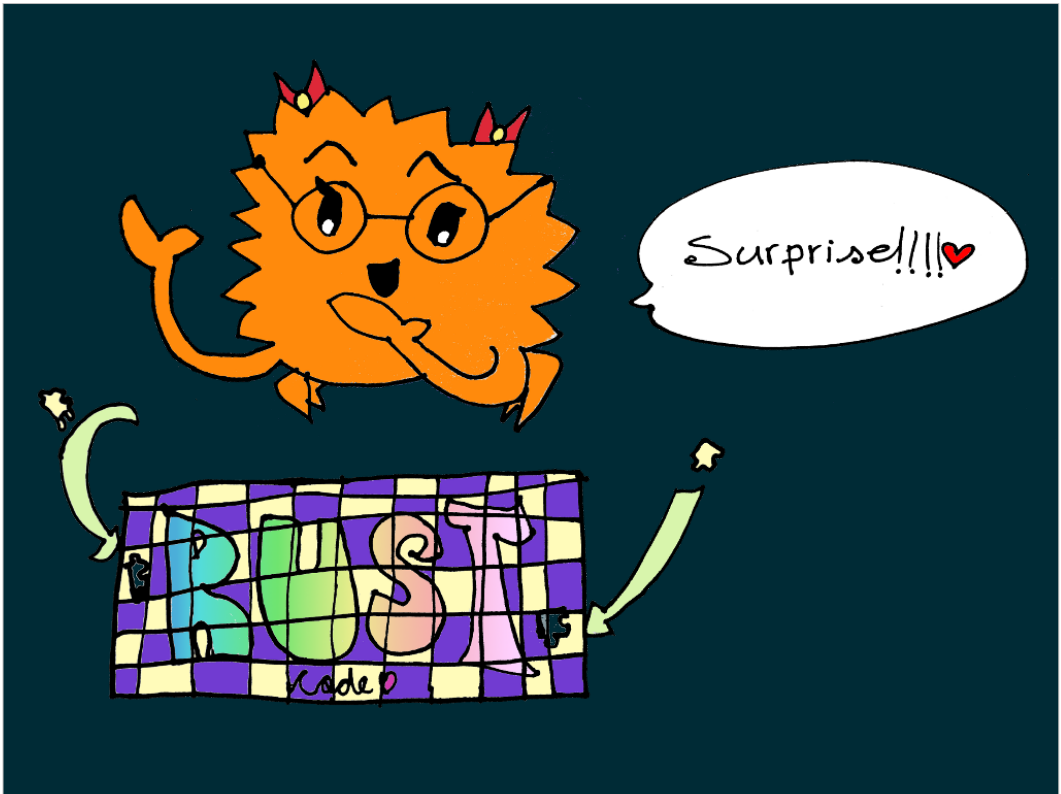
- I'm a library writer
- Disenchanted with C libraries for years
 - Learned helplessness?
- Rust felt empowering
- Rust refreshed my spirit
- Rust showed me a community that GNOME can learn from

I haven't written apps for years. I'm a library writer.

Rust revived my interest in maintainership.

I also think we can learn some things from the Rust community.

It is very empowering to finally have a good language that I can use at the level of the stack where I work.



Ferris is happy to present you with a nice drawing!

Thanks

- Luciana Mena-Silva (my daughter) for drawings of Ferris
- Katrina Owen for the technique of narrating/highlighting code - <http://www.kytrinyx.com/talks/>
- Alberto Ruiz, Joaquín Rosales for the GNOME/Rust hackfest
- [#rust-beginners](http://irc.mozilla.org)
- [#nom](http://irc.mozilla.org)
- Sebastian Dröge for nom macros

Blog posts

- Librsvg gets Rusty - <https://people.gnome.org/~federico/news-2016-10.html#25>
- Porting a few C functions to Rust - <https://people.gnome.org/~federico/news-2016-10.html#28>
- Bézier curves, markers, and SVG's concept of directionality - <https://people.gnome.org/~federico/news-2016-11.html#01>
- Refactoring C to make Rustification easier - <https://people.gnome.org/~federico/news-2016-11.html#03>
- Exposing Rust objects to C code - <https://people.gnome.org/~federico/news-2016-11.html#14>
- Debugging Rust code inside a C library - <https://people.gnome.org/~federico/news-2016-11.html#16>
- Reproducible font rendering for librsvg's tests - <https://people.gnome.org/~federico/news-2017-01.html#11>
- Algebraic data types in Rust, and basic parsing - <https://people.gnome.org/~federico/news-2017-02.html#03>
- How librsvg exports reference-counted objects from Rust to C - <https://people.gnome.org/~federico/news-2017-02.html#17>
- Griping about parsers and shitty specifications - <https://people.gnome.org/~federico/news-2017-02.html#24>
- Porting librsvg's tree of nodes to Rust - <https://people.gnome.org/~federico/news-2017-02.html#28>
- gboolean is not Rust bool - <https://people.gnome.org/~federico/news-2017-04.html#28>