Porting gnome-shell styles to Rust
Federico Mena Quintero

>> Can people hear me?

>> Yes.

>> Awesome.

>> I think we can start easily. Welcome back to GUADEC 2020. Day two, Track 1. We continue with Federico Mena Quintero with porting gnome-shell styles to Rust. Please.

Federico: Thank you. Can people hear me?

>> Yes, loud and clear.

Federico: All right. Perfect. Let's get started. Thanks, everyone, for being here. I hope everyone had a chance to take a nice break. Anyway, my name is Federico Mena. And I have been doing a lot of work on porting things to Rust. As many of you already know. For the last two or three years the lead team ported the whole library to Rust. And now the trend it for the gnome-shell's styling machinery, which shares many of the things that they have for CSS processing.

The reason for doing this is two-fold. Gnome-shell uses a library called libcroco. Which is very old and unmaintained for processing CSS. It's written in C. It's had a couple of vulnerabilities in fact past. It would be nice to replace that with some maintained Rust code. By the way, that's my email address and my Mastodon account. I don't really post on Twitter, if you want to get a hold of me on social media, get on Mastodon. I'm there. Let's get on.

Gnome-shell uses its own toolkit for drawing windows and buttons, the top bar, all of those. And it's one of the things -- stands for shell toolkit. And the code -- the initial code for St is actually pretty old. It comes from HippoCanvas. It was a thing that some of the developers more or less during 2006. Back then there was this, I believe it was a Red Hat project called Mockshot. The idea was to kind of gather all the social media accounts that people had back then, like your RSS feed for your blog and maybe your microblog from Idenica, and Flickr photos together and gather all of them in the same place. But then Facebook and Twitter got popular. And the Mockshot project just --

But they left us with HippoCanvas which was a nice derivative of the old gnome-canvas. Gnome-canvas was a retained mode graphics infrastructure that many applications used back then. So, know now have HippoCanvas in the form of St. And it's basically a retained mode graphics system that does its styling with CSS. So, the only things left over from those times are StTheme, which holds the representation of the stylesheet, and the StThemeNode which I'll describe in the next slide.

The magic behind St is that it supports theming uses CSS. More or less just like GTK. The gnome-shell toolkit also is theming with CSS. Every widget that gnome-shell displaced, the top bar, the buttons, the activities button, the drop down menus, each one of those widgets has an object associated with it which is a type of StThemeNode. And an StThemeNode, it works like a Node if

you actually had a DOM like on a web page. Except that gnome-shell is not a web browser. It doesn't have a DOM. It just has a different tree of widget.

StTheme implements the CSS matching algorithm to see which gets which applied to them. And later CSS has cascading, which is about copying the parent's properties, you know, from parent to child. That happens in a very ad hoc fashion in StThemeNodes. I started porting this to Rust in November of last year and it's coming along. It's coming along pretty well.

You have been paying my daughter to do the illustrations for this talk. So, all the pretty drawings that you see here are her doing. The gnome-shell screen looks like this. You know? We have the root window. How does one forward here? Oh, yeah. We have the root window. Which corresponds to the root node, to the root StThemeNode. And for each of the subwidgets, the top bar has this StThemeNode. And then each of the -- like the activities button, the clock, all the little drop downs and dragged. Each of those has its own corresponding StThemeNode there.

And the magic behind St and StThemeNode is that it's able to do the CSS cascading on this tree to give the widgets here their final style. I'm sorry. We have a few duplicate slides.

An StThemeNode is the object that corresponds each styling -- to each styled element. And it has a bunch of properties. It has the font for that widget, the foreground -- the text foreground, text background. If the background of the widget has a gradient on an image. And all the CSS models for the outline, the border, the border_radius. These are kind of similar to those in the web world, but not entirely. And one thing I want to do here with this project is to actually make gnome-shell's styling be closer in spirit to the web. because that might make life easier for making themes.

Now, we're gonna be looking at a bit of code. I'm gonna try to highlight the things I'm talking about. One thing I found with BigBlueButton is between each slide transition, it paints the whole screen white. That breaks the flow for describing code. I'm going to highlight with the highlighter here. I don't have syntax lighting. But I tried installing the Libre for Python Extensions. But it was throwing code everywhere. Quality software. Everywhere.

Every function in St that parses CSS properties looks more or less like this. This is the function to parse the foreground color property for -- for an StThemeNode. It takes in the node. This is the output value, it's a clutter color. And everything looks like this. If the thing has not been computed, then turn on the flag saying that it's computed. Load the properties for that object to make sure the parent had done the CSS cascade on there. And then for each property look at the list of properties in reverse order. Because in CSS, the last property, if you have the indicated names, the last one wins. So, we look at them in reverse.

Fetch the property with index I. If the property name is color, then parse the color from the rest of the properties value. Put it in this field. The foreground_color field. If the parser says that value was found. We're okay. We return. And otherwise, if the parser says that the property is like inherit, you know, inherit -- hard to write with a mouse... if a property value is inherit, it will break out of the loop. And let's look at what happens if we do that. We break out of the loop. If the node has a parent, then add the foreground color from the parent and copy it into our own foreground_color. This is the CSS cascading.

Otherwise, if a property couldn't be parsed, or if it couldn't be found, just default to black. Otherwise return. So, this works fine. It's really ugly code. It's very repetitive. It's error-prone. But, you know, it really screams, please refactor me.

It's very repetitive, as I told you. So, to summarize, every property parser ensures that the property served, if it was not computed, parse it. If inheritance is allowed for the property we're parsing, because not all CSS's properties allow automatic inheritance, if inheritance is allowed, ask the parent. Otherwise, use default value. All the code is written by hand and it looks like that. We can certainly make that shorter.

One last thing -- one last different between gnome-shell and librsvg is that gnome-shell actually has tests for the theme parsing. So, I was able to eyes those very effectively for the new Rust-based parsers. If this were not a PDF presentation, then the cat would be animated. So, you can imagine it. I'm sorry for the lack of animation.

So, let's see how the -- how the parsing code -- what the parsing code looks like. When I told you about parse_color, that was a lie. The function actually looks like this. It's called get_color_from_term. And term is the terminology for one of the terminal nodes in its syntax tree. It's a token, you know? It's a token that must be parsed. So, we get a linked list of tokens here. We need to parse it. And we need to return a value of this type. Which it has value_found, not_found, or inherit. Give me one second here, please.

And what this means is that if the parsing was okay, return value found. If there was a parsing error, or not found, return the value_not_found. The CSS specification says if there's a parsing error, just ignore the property. Don't exit program or bring up an error or anything. Just ignore the property. And if the value is inherit, then it will copy the property from the parent node.

So, let's start parsing this to Rust. This is Rust code right here. The repr(C) line should have the same in memory as the equivalent C enum. If you're already a Rust programmer, please forgive the basic level of this description. This is for hackers which may not have had the chance to look at Rust code yet. So, anyway, this is for their benefit.

Then I give the enum the same name as the C version. We have the same variant names for the -- for the enum. And then I started writing a little library in Rust called stylish. Stylish is, you know, for gnome-shells, stylish, it's the stylish with the S-h because it's the gnome-shell. I'm really good of thinking of names, as you can see. Anyway.

So, this function, stylish_get_color_from_term replaces the get_color_from_term in C. And it gets it from CRTerm, and returns ClutterColor. And return GetFromTermResult. The point is to make this API-compatible with the existing code so I don't have to refactor the whole C code before inserting Rust functions in there. This will get prettier later.

So, let's look closer at the actual implementation of the parsing function. Before we do that, let's talk about CSS values. When you write a little chunk of CSS, you can have something like -- how is one typing here? Text... you can have auto, like, color: Blue, inherit. So, if you wrote color: Blue, this is called a specified value. Because the user actually specified something here. If you write color column inherit, it's an inherited value. So, Rust makes it very convenient to differentiate between those two cases. We'll define a tagged enumeration. If you know a bit of C++, this is kind

of sort of like C++ templates. So, the value for this could be either inherit or a specified value of type T, you know? In the case, the specified value will be of type ClutterColor.

So, what does this look like? The parser's internals are really going to want to return a Rust result. Result is, you know, built in typing Rust. And then we're going to have to translate that to a GetFromTermResult which is the enumeration that tells us if the value is found, not found or inherit. We implement the parser. The actual details are not important here right now. I want to show the conversions between the result types.

So, we're gonna return -- or the parsing function is going to return a specified value for a color which we're gonna convert to ClutterColor. And it's going to return a parser. We match on the result to see how to translate it. If it's okay with an inherit, then we return GetFromTermResult: : Inherit. And then they can convert to the glib representation. The to_glib functions convert a Rust value to a C value that glib can read. In this case, a ClutterColor. And we will return GetFromTermResult found. And we can ignore the error and return GetFromTermResult: : : Not found. And here, the Rust mascot is a bit unhappy about this. She's happy that we are ignoring errors and this code is actually pretty ugly. It's conversion code between types, you know?

But that doesn't matter for two reasons. First, right now gnome-shell does not have enough machinery to give good errors to the user. So, I'm not entirely sure yet how designers write the CSS for gnome-shell. But I imagine that it can be pretty painful if they accidently mistype a property name or a property value and then gnome-shell presents no errors. Maybe puts something in sys log with the warning. That's painful. You want to have development toggle in gnome-shell that says, yeah, please give me a listing of all the errors I made in my CSS. But anyway, we're going to ignore them for now. And we're going to remove all this ugly code with the power of refactoring. And the refactor version looks nice and short and pretty like this. We're gonna get value from CR terms. The Rust representation is color. The C representation is ClutterColor. For parsing reasons that are not important here, sometimes we want a linked list of the terms. And sometimes we want to parse a single term so we specify is here. And then we convert the return value of that to its gnome-shell representation and we put it in the out parameter.

And this whole chain of calls gives out a GetFromTermResult. And it refactors very nicely. And since within this we already have the machinery to do the type conversions and everything, we can very easily start plugging in new types. We have the color to ClutterColor. We have a custom type from gnome-shell, StGradientType, and this is to no op. And this is all I need to add the type to the API that is visible from the C code. And I'll show you the definition for this value later. It's very nice.

So, back when we started doing the port of flip to Rust. I was writing C header files by hand. Basically because I was still learning the Rust infrastructure. And since I was parsing one function at a time, it was not a big deal to write the C prototype for it. So, I just did that by hand. But now there's some discourse in the Rust community that one should actually use a tool -- a tool to automatically generate header files back and forth. And they have very good reasons for that. So, I got convinced. There's a tool called cbindgen. And it's awesome. And lets you configure it and say things like, okay. I'm going to write C. A C header file. Not a C++ header file. My guard is going to be called stylish_H. If hash, not defined, if not defined, include the whole thing. Add a warning so they will not try to autogenerate the file by hand.

And finally, you have a list of type names to exclude because these types are built into gnome-shell or libcroco or St. So, I don't want the CRD file generated to have duplicates. And then it will let you rename enum variants like this. What does QualifiedScreamingSnakeCase mean? Let's look at that.

You tell it to call snake case. If this is the Rust code for enum, and this is the C code that corresponds, it's called StFoo. And the ST under foo. And the foobar, and we want to go with the naming conventions for gnome-shell. Identifiers in PascalCase look like this. CamelCase because of the hump here. Kebab-case, only works in list, not in C. Snake_case, screaming_snake_case is the same, but in all capitals. I mean, it's qualified because of this prefix on every single variant. So, QualifiedScreamingSnakeCase means to put the enum name before each variant name and make them capitals. Now you know the reason for that name.

Oh, yeah. Let's talk a bit about CSS -- [breaking up] gnome-shell. We'll use CSS lengths to specify many things. The length or size of various items in the user interface. Sometimes they are given in pixels. And converting this to actual device pixels is not a problem. We just multiply this by the device's scaling factor. If we have a high DPI screen. If we have length value in points, this means 12 points, it's not a problem. But we need to know the resolution of the display so that we can multiply the DPI by 12. And if it's given in terms of a font size, if a length is 1.5ems, we need to multiply 1.5 by the current font size. So, we need the DPI and the font size and the scaling factor to come from somewhere.

To convert them to device pixels. So, gnome-shell would -- the SQL for gnome-shell would just pull those out of, you know, various things in the -- in the gnome-shell environment like, you know, what's the parent node or what's the theme or what's the current display? But to communicate that to the Rust code, I made this little struct called normalization parameters. NormalizeParams. And it contains the font size, the resolution, the scale factor. Everything that's going to be needed to normalize lengths to their -- to the device pixels.

So, how do we do that? Rust traits are like kind of sort of like Java interfaces. So, we have a trait called normalize. With a method called normalize. It takings the object itself and some normalization parameters and outputs something that depends on the thing that knows how to normalize itself. For example, oh, and there's also some length values or some CSS properties in gnome-shell that instead of just converting a real number, they need to be converted to integers. Because they want to be not just to the pixel read. You don't want to have 2.3 pixels. You want 2 pixels or 3, but not something in the middle.

So, the Rust code has two traits. One to normalize to a real number. And one to normalize to integers, to the pixel grid. But what was I telling you? What was I talking about? Oh. CSS lengths, that's right. So, let's talk about CSS lengths. Lengths can have their numeric value and the unit. And the unit can be pixels, Ems, X values, integers, centimeters, points, right? How do we normalize a length to its device pixels?

Well, we look at the unit in the length. The end of the unit is here. If it's pixels, then we just multiply the numeric value by the displaced scaling factor. If it's in inches, we multiply it by the resolution. If it's Ems, just multiply it by the font size, et cetera. So, every length value now knows how to normalize itself to f64 which is the same as a double from C. Right?

And for the CSS value, can't be normalized adjusted to the pixel grid. For example, there's this new struct called St sides. Sides is used to introduce both the margins -- what is it? Margins, border and padding from each widget. And you'll notice that in the Rust -- in the Rust types, this has lengths. It has floating values. This is an F64 plus the units. But these are ints. An i32 is an int. We don't implement normalize for size. Rather, we implement normalized pixel grid for sides. And this has just a rounding attached. Apart from the normalization from the lengths. The important thing here is that from just looking at the code, it's very easy to figure out how each value type wants to be normalized to the pixel -- to pixels. Does it want to be normalized to non-integer pixels or to the pixel grid?

So, that makes the code eligible, I think. So, I have been porting the parsers for the different functions in -- in St to Rust. Fortunately, most of them can be recycled from the code that is already there in librsvg. librsvg knows how to parse CSS lengths, colors, font sizes, things like that. And gnome-shell, fortunately, keeps more or less the same CSS syntax.

Once all of this code is done, I'll be able to make the gnome-shell code actually do the parsing for the whole stylesheet in Rust. Because it's done partly in C and partly in Rust right now. And once that is done, gnome-shell is going to need to tell Rust about the -- about its representation for the widget tree. If you think that -- if you can imagine this as the widget tree, and we have a node here. Then we have an interface that can tell it to select element. If you have this node, this is the parent element. What's the sibling? If you have this node, this is the sibling. The previous sibling or the next sibling. If this is the node, you have class , you know, an class. Or does it have an element ID? Does it match pseudo element? Is it hover? One of the pseudo elements in CSS.

So, the Rust crate that I want to do to use CSS matching demands this kind of interface for your own representation of the DOM tree. It's not terribly hard to do. It took us a while to do in librsvg because there was so much code that needed to be refactored. But for gnome-shell, I expect it's going to be a lot easier.

Switch to the next slide. Oh, yeah. These are some macros that we already had in libraries that are completely reusable for gnome-shell code. For example, there are a couple of property types in St. There's IconStyle property and a TextAlign property. Which can just have values which are identifiers. For example, if the user says requested, we'll return an enumeration value with the same name. Regular, symbolic. So, this kind of macro makes it very, very easy to define the whole machinery for new properties. Like the property's gonna be called StIconStyle. The default property is requested from here. The possible values are just identifiers called this. And they map to these enumeration values. You can see whether the CSS property inherits automatically or not. This one doesn't, this one does.

This means that for this one, if it is not specified, then the value will be copied for the inherit mold. And this doesn't inherit automatically. Which means that if it's not specified, it will go back to the initial or default value. So, with just a few lines of code, the whole machinery now supports these two properties. Instead of having, you know, 50 lines of C code for each one to implement all the parsing and validation and cascading and everything. So, this is nice. This is actually quite nice.

One thing that I really am appreciating from porting the gnome-shell code to Rust is that it is also forcing me to fix some things in librsvg that have been long-standing bugs there. In CSS2, the font-weight from the could be font-weight bold or bolder or light or lighter. Or multiples of a

hundred. So, font-weight 100, font-weight 400. But always supported arbitrary numbers in there, not just multiples of a hundred. So, you know, for CSS4, I don't know if the CSS working group actually started supporting arbitrary values or if they gave up. But anyway, we now support any integer number between zero and a thousand. Which is the possible values for that property.

Gnome-shell has always used the font short hand. The font short hand is what lets you wrote font column, you know, font style, weight, size, font family. That's what the shorthand syntax looks like. And this is changed into properties. This font shorthand means that the font style will be italic. The font weight will be bold. The font size will be 12 pixels. And the font family will be Cantarell.

And the librsvg never supported that because property short hands cannot be used in attributes. So, if you have rectangle in SVG, and then write a font attribute, it doesn't work because it's a shorthand. You need to write all this individually. If you write a style attribute with inline CSS or something, that actually works with the font shorthand. Anyway, it's fixed in both librsvg. And using that code in my branch. I'm really liking in thing about standardizing the property syntax for both projects now.

What I've described so far is what is done. What is implemented in the code right now. If screen sharing worked, which it doesn't in my machine because I'm using Wayland and GNOME 336, open, screen sharing doesn't work. But anyway, you wouldn't see any changes from the normal gnome-shell. And that's because the parsers produce exactly the same values as before. That's guaranteed by the tests. So, my gnome-shell looks no different from your gnome-shell. But mine is running some Rust code now.

But let's talk about stuff that doesn't exist yet. The bright future, right? StThemeNode is an important place to reduce memory consumption in gnome-shell. Because every single widget on the screen has one of these. Every single widget on the screen has an StThemeNode. It needs to be as small as possible. They are cached right now with a scheme that I'm not sure will work if we add complex CSS selectors. But anyway, librsvg is starting some machinery to cache the styling. We can probably share that code in the end.

I am fairly confident that we will have faster CSS matching in the end. In theory, as fast as the one in Firefox. Because the Rust crates that librsvg uses for matching are the same ones that are used in Mozilla Servo, that's experimental for Firefox and merged into the main line these days. And it's really, really fast. They do all the things done these days for CSS matching. They use bloom filters and things like that. So, we'll have a pretty good CSS matching engine.

That thing also supports complex selectors out of the box. Sibling combinators, children combinators, all those that browsers support, they are supported there. You just need to implement the big interface I showed a couple of slides ago.

And one thing I really would like to have is a shared Rust crate between librsvg and gnome-shell. Because they want to process pretty much the same CSS properties. And they have the same matching code and the same cascading. So, we might as well share some code. It's kind of funny that my original plan was to replace libcroco. And that's exactly what I ended up doing. If I'm writing a shared Rust crate, it's probably not going to be called libcroco. But it's the equivalent. Maybe others can use it. I thought gnome-shell was the last that used libcroco .

but Inkscape embeds their own copy of libcroco. And Inkscape, doing SVG and CSS and the code is the same, we might as well do that and share some code.

For the designers, I want to have really nice things for the designers. Gnome-shell does not support @media rules yet. I don't know what will be useful there. Maybe the CSS media rule syntax for color schemes for the dark themes maybe or for the device resolution. If you have 200dpi or higher, then use more detailed icons instead of, I don't know, hiding at runtime or having other machinery. Do we want to support light levels in the environment? If it's dim, change to night mode?

So, these things are possible through media rules. We need to add support for them in the parsing code. That's easy to do. And GNOME needs to decide which of the properties they want to support and whether they would be useful for designers to have, you know, fewer stylesheets or simpler CSS or something like that.

As I told you already, I really want to have new machinery for error reporting. We have been having better error reporting internally. And I want to expose it to the -- to the outside API somehow. You know, give me a listing of all the parsing errors or, you know, which properties got cascaded where to make like a, you know, CSS explorer like in web browsers. I don't know. But to have better error reporting to make designer's lives easier. All of this is vaporware. It doesn't exist yet. But we'll get there. Rust, fortunately, makes this fairly -- fairly nice to do.

This is Ferris doing the CSS cascade. So, feel free to partake. And that's all I have. I want to thank a few people for making this possible. My daughter Lucian for all the illustrations. My dog for debugging. He barks if he detects a bug. And he doesn't if I get it wrong. And my wife because I sit on my butt all day writing code and she supports with schoolwork and stuff. The Mozilla Servo people have been awesome in supporting us with the CSS machinery and the CSS processing crates.

The gnome-shell people are lovely. They build basically the most important use for GNOME. Everybody goes to the shell to do their thing and go into the little applications like individual buildings and such. And finally, the technique of writing code comes from Katrina Owen's videos on YouTube. They are great. You should look at them. And that's all I have. I'll be happy to answer questions.

>> Thank you Federico. You can find the questions in the Etherpad. And in the shared notes you can click one-on-one Etherpad link and find your presentation. Talk five.

Federico: Thank you. Yeah. So, the first question is, you said replace libcroco with a nice, maintained Rust module? How does that work? Do Rust maintainers just appear by magic? Let me show you a trick. Rust maintainer. Poof! A Rust maintainer appears. So, yeah, I intend to maintain this for a while because if we have a shared crate between librsvg and gnome-shell, I tend to maintain that. It's fortunately not one. But already knows that code fairly well. So, we have two people who know how to work on it at least. So, yeah. That's my promise for a few years. To keep maintaining that.

Also, the -- like the CSS matching crates that we have from Servo, those are maintained by Mozilla people. And I think it's safe to say that Mozilla will be sticking around for a while. So, you know, hopefully we won't have problems there.

Second question in the Etherpad. Do you feel less productive as in speed of implementing features, I fix bugs with et cetera, writing code in Rust versus other languages? Not at all. The thing that I experienced when learning Rust is that, yes, it took me longer to write during the -- during my learning period. And that seems to match other people's experience in the Rust world. But once you get it, once you -- every Rust developer seems to pass this phase which is called fighting the borrow checker. But once you figure it out, once you figure out the data flow in your program, you stop fighting the borrow checker, you start understanding what the compiler is telling you and if flows very smoothly. Once you get to that point. It took me maybe a year. But it's not -- it's not bad. It's not bad, don't be scared. I actually am very productive in Rust because I'm able to use a very strong type system, a very nice macro system.

So, you know, the macros I showed you for -- for the -- where is it? For the properties here, these macros just makes life so much easier. And they let you write -- or not write so much code really fast. So, it's really nice. I find myself very productive in Rust these days. So, I hope that's reassuring.

Next question. Are there projects that are next on your list for corrosion? Or rewriting in Rust? This is total vaporware except for some working code on some tests that pass. I am trying to refactor libjpeg-turbo into safe Rust code. To use the SIMD codes to do a new JPEG Codec. Send me an email if you would like too that. If you are experienced with image Codex, send me an item.

Next question, is there step by step rough and simple instructions for porting gnome-shell to Rust? Not yet. But that's an excellent question. I should do that and write something up in a blog post. So, yeah. Maybe I should write one next week or so. Thank you. A note about -- next question -- about @media rules. Would be cool to explore for desktop/pad/phone resolutions? Maybe. I don't know. I'm the kind of person that knows how to implement a CSS processing infrastructure. Not how to actually use CSS in the web or something like that. It's designers who will have the last say on this what they will find useful and whatnot. You know, if there's useful properties that you would like to have in gnome-shell, tell me about them or file bugs in the stylish project and I will definitely look into implementing them.

Next question, you replaced CSS parsing in librsvg and gnome-shell, is GTK next? I don't know. GTK3 and 4 have their own CSS parsing code. It's not libcroco. I think Benjamin and others wrote for CSS GTK. I was looking at it. It's pretty good. So, I don't know. It would definitely be useful to standardize on the supported syntax for properties. You know? Do they all support the same kinds of -- of colors, for example? Gnome-shell uses the -- for the color properties, gnome-shell uses the -- or it used to use the libcroco for colors, supported the RGB syntax. The harbor the digits. But it didn't support HSL and HSV, I think. With the new confirmed and Servo out of the box, I don't know if we support that. But it would be nice to standardize on at least the property syntax.

I don't have plans to replace CSS parsing in GTK with Rust. But I don't know. I don't know. We'll see in the future. Next question, does Rust build on all interesting architectures where we need gnome-shell these days? Someone answered, Debian has been building Rust toolchain for although architectures Debian supports. That's enough for me. There are still architectures out there. Building for Spark and others. Now there's a proposal that came out last week to make ARM

64 a tier 1 supported architecture in the Rust compiler. That means that it's like one of the good ones. Like it'll work just as well in the cross-compiler as 64. For those thinking of buying ARM 64 hardware, that should work just fine.

Next question, a lot of this work would seem relevant to GTK. Will this work in stk be usable by GTK. It's the same. Libaligator is the new name. I like it. Maybe. Maybe libaligator instead of stylish. Yeah, I'm sure we'll be able to make all sorts of nice puns with libaligator. The big question, what is the dog so happy about. Let me show you my dog again.

Oh, so, that photo is from two weeks ago. It's the first time that my poor dog was able to go to a park with trees and everything during the quarantine. So, he was just out of his mind happy. So, that's why he's so happy. Anyway. I don't see any other questions from the Ether Pad. If that's everything?

>> That's everything. Thank you.

Federico: Thank you. Thanks for attending.

>> Thank you Federico for an excellent not to rusty talk. And we will be back in a minute with the next one. Thank you!

Federico: Right. Thank you.