

# KDAB's Software Development Best Practices

## Building Hybrid Rust and C/C++ Applications

**Andrew Hayzen** | Senior Software Engineer, KDAB

**Leon Matthes** | Software Engineer, KDAB

**Florian Gilcher** | Managing Director, Ferrous Systems



As a modern programming language, Rust has benefitted from lessons learned from earlier established languages like C and C++. As a result, it has been designed to avoid many common programmer errors. Because the language structure and strict compiler checks make it more difficult to create bugs, Rust has been steadily attracting developers interested in creating more resilient code. Once developers fall in love with this safety net, they often try to “oxidize everything” – in other words, rewrite all code in Rust. However, while an all-Rust approach might work for very small bare-metal embedded systems or fully isolated microservices, it doesn’t instantly retire the vast amount of existing code in most company’s code bases. For the foreseeable future, most developers realize a more nuanced stance is necessary, one that harmoniously blends Rust with existing C and C++ libraries.

Thankfully, this pragmatic “Rust in the Real World” approach has been part of Rust’s heritage since its inception within Mozilla Firefox. So, in light of this historical combination of Rust and C++, what have we learned that can help address integration challenges today? To find out, we consulted Rust experts and practitioners within our engineering department as well as Ferrous Systems, an internationally recognized Rust consultancy. Here, we present guidelines and advice on combining Rust and C/C++ across a variety of scenarios.

## 1 Structure

### 1.1. Where to Start?

Because Rust began life within a C++ environment, combining the two languages is not unusual. This means the community experience for building a hybrid Rust and C++ solution is deep and the tooling is good. But when deciding where to begin integrating the two languages in your software, are there any special characteristics in your existing code that you should look for? We recommend picking a starting point that has these traits:

- **Problematic.** Pick something that is already a source of bugs and crashes, software that is messy and difficult to maintain, or software with many potential vulnerabilities. If you're going to the trouble of oxidizing it, you want to choose something where a Rust conversion is going to pay immediate benefits.
- **Self-contained.** Ensure the code is well-isolated without a lot of calls into the rest of the code base. Code with sharp boundaries will be the simplest to extract, rewrite, and replace. These will also be the easiest to create a C foreign function interface (FFI) that can be thoroughly unit tested to ensure a swapped-out component behaves identically.
- **Clean interface.** Choose C++ code with a reasonably clean interface and entry points that are clearly defined and as narrow as possible.
- **Idiomatically similar.** By this, we mean avoid C++ software that isn't well-suited to Rust and thus would require gymnastics to translate. For example, software that relies heavily on preprocessor macros or templates, or is inherently object-oriented rather than procedural are not ideal targets for a first translation attempt.

A good example of a suitable Rust target is often a leaf library at the edge of the software architecture. This might be an application's support for media or image files (which are also often plagued with memory safety issues that Rust can address). Other good choices are parsers and input handling routines. More generally, modules that manage and process untrusted, externally derived data often meet all the desired attributes.

In general, don't do a full rewrite in Rust, do it in pieces. Find one good place as the initial target. A Rust conversion also tends to improve the remaining C++ code because this enforces a clean separation between the layers of your application.

## 1.2. Rusting Outside-in or Inside-out?

Once you've had some smaller successes in oxidizing your code base, you might be tempted to tackle bigger conversions. Is it better to think about a large project as a C++ project that contains Rust components, or as a Rust application that calls into C++ libraries?

Either way can be made to work whether you're calling from C/C++ into Rust or the other way around. However, there are a couple items you might want to consider:

- **Safety approach.** In Rust terms, C/C++ code is trivially unsafe, while Rust is safe by default until you start marking code as "unsafe". So, if you consider your project primarily safe with a handful of carve-outs, then structure your application as Rust that calls into C++ functionality as needed. If it's better to think about your project as a big project with specific safe zones, then having a C++ main framework might make more sense.
- **Multi-threading.** It is very difficult to safely manage data accessed asynchronously from both sides of a Rust/C++ boundary. Thus, it's best to consider the Rust and C++ worlds as separate, keeping threads and thread data independent and isolated on each side. Depending on how your application uses threads, this may tilt the decision in the direction of either C++ or Rust as the main "host" application.

If you're building a new application from scratch, our recommendation is to start creating it in Rust. While the application is small, you'll probably be able to do it completely in Rust, which will avoid the hassle of needing C FFIs. As the application grows in size and functionality, the more likely it is that you'll find situations where a good Rust solution doesn't exist but a good C++ one does. Then you can progressively add C++ libraries and the requisite C FFIs as necessary.

## 2 Keeping C++

### 2.1. When Not to Throw Away Perfectly Good Code

Despite how much the Rust community is into embracing correctness, there is also a surprising amount of tolerance for getting things “mostly right”. That is, if you can get things 90 percent of the way there – make things as good as you can – then you can work on refining it later. That naturally leads to the following questions: Do we really need to redo everything in Rust, and if not, what should we leave alone?

A long-term plan to oxidize all your software is not always necessary. It's not even always a good idea since there is a lot of time-tested and high-quality C++ code. Algorithms like signal processing or cryptography have limited inputs and outputs. They can be very complex yet may not need to allocate memory – the source of 70% of C++ bugs according to both [Google](#) and [Microsoft](#). Rewriting this code, even if that rewrite is in a safety-focused language, creates an opportunity to introduce subtle new bugs that have nothing to do with memory or pointers. As a result, they may be extremely safe to leave as a C++ implementation. Similarly, simple driver implementations in C probably won't benefit much from a Rust redo.

Throwing software out the window for the sake of language purity doesn't make sense. Before you start a rewrite, ask yourself these questions:

- Do you actually have a problem you're solving?
- How many other people have used that code and how well is it exercised?
- How long has it been since it's been changed?
- How does it treat memory allocation and deallocation?

If your answers provide some confidence that the software is working and correct, then it's probably better off left alone. There's no point throwing away multiple years of knowledge and engineering that have been invested into that module. However, if you're unable to convince yourself that the software is correct, then maybe it's time for a rewrite.

## 2.2. Times to Avoid Rust

Rust isn't a panacea for all problems, and indeed, there are some areas where C++ has an advantage. Most challenges aren't due to the language itself but rather differences in maturity of the solutions.

- **Maintainer risk.** The Rust software ecosystem is still young, which means there are often very small teams responsible for maintaining critical bits of code. Many of the common crates that people use are maintained by a single person. If a maintainer moves on to another project, the software you depend on may end up being stranded. There may be no one to rely on fixing bugs except yourself.
- **Version stability.** Rust is still a new and growing language with a couple of significant compiler releases each year. Brand new language or compiler features may be essential to your development, which means you'll need to enforce using a compiler with a certain version number or higher. That can impose many difficulties on a project, forcing lots of software and toolchain updates, and preventing the use of modules built using older versions of Rust. While new C++ standards are also released on a regular cadence, the C/C++ community manages backward compatibility much more strictly.
- **Software availability.** While Rust has many solutions in lots of popular domains, there are areas where it's not yet as well practiced, such as embedded development. Embedded Rust

solutions that already exist to solve specific problems may be difficult to find, while C++ solutions exist and are easy to obtain.

Additionally, Rust's fluidity can be most challenging in several areas:

- If you're dealing with a very large codebase
- If you need specific embedded support
- If your software has long development cycles
- If your product needs in-field support for many multiple years.

In these environments, it's ideal to use C++ for the code that requires long-term development stability and Rust for software that requires safety but can survive a certain amount of development churn.

### 2.3. Places to Avoid Rust

There are a few places where converting code from C++ to Rust can be done but is tricky and troublesome. For instance, complex object-oriented hierarchies using multiple inheritance might be difficult to replicate cleanly.

Qt applications are another good example of code that's tricky to oxidize, since Qt code relies not just on objects and preprocessor macros but also requires an external pre-processing step to manage signals and slots (the moc). This doesn't mean that you can't have [Rust and Qt co-exist](#), but the complexity of getting Qt idioms to work in Rust means it's usually better to leave those chunks of code in C++. Wrap your Qt event loop and window management with an appropriate set of C FFIs and let Rust call into it when it needs to put up UI components or get user feedback.

Similar issues exist with C++ code that mixes a wide variety of

string types. Because the C and C++ language/library features have grown alongside global needs for expanded string encodings, C++ has acquired many independent ways to create and refer to strings. Here are a few:

- C-style character pointers (with `char`, `char16_t`, `char32_t`, and `wchar_t` types)
- `std::string` objects
- UTF-8 or UTF-32 literals
- string streams
- `QString`s

Rust has had the benefit of being built after the evolving string landscape and, as a result, has developed string options that are semantically coherent and safe. However, managing C++ interfaces with messy string APIs can be challenging on the Rust side. They can also take an FFI performance hit as we discuss in section 3.2.

Essentially, the further you get from clean interfaces and isolated subsystems, the more difficult any oxidation project will be.

## 3 Moving to Rust

### 3.1 Advice for C++ Developers Attacking Rust

When you learn a new language – programming or otherwise – your brain naturally translates new concepts into terms you’re already familiar with. However, a C++ developer attacking a problem in Rust isn’t merely adjusting to a new syntax. Because Rust borrows some of its lineage from ML, a functional language, while C++ uses an object-oriented paradigm, developers must adopt a different mindset. Numerous aspects of these languages have significant underlying differences. Understanding the



subtleties of effectively managing ownership and object lifetimes, grappling with dramatically different approaches to concurrency, and comprehending the “what” and “why” of each language’s definition of undefined behaviors can take time and hands-on experience to acquire.

There is a steep learning curve to developing a “Rust-natural” style of coding, and getting into the proper headspace of a language is part of the learning process that can’t be rushed. This means that C++ developers coming to Rust shouldn’t be overconfident in their understanding of subtle differences in the code. Let new-to-Rust developers start in areas of the code that aren’t as critical so they can experiment with their new knowledge. Allow them extra time because they will inherently be less productive. Get external reviews from time to time that include an expert assessment of the code, allowing the expert to explain the why and how of proper “Rustacean” form.

### 3.2 The Cost of Rusting

Is there a performance downside to converting C++ code to Rust? Not really. Although Rust may insert runtime bound checking that C++ doesn’t, its performance is generally on par with C++. Both are compiled languages without garbage collection that share the same compiler back-end and optimizer as Rust has both [gcc](#) and [LLVM](#) variants. So, while you might be able to squeak some faster benchmarks out of C++, for the vast majority of real-life production code, Rust performs closely enough that you don’t need to worry about the difference.

While this is true for standalone Rust, there is a performance penalty for combining Rust and C++ when you force Rust code through a C FFI. This kills the effectiveness of the optimizer around any C FFI calls and may also require some Rust type conversion. The exact same problem arises on the other side of the bridge,

where calling Rust from C++ code ruins the C++ compiler's chance of optimizing across the call. It's important to be aware that there can be performance hits from interlanguage calls, particularly in hot loops or other code that needs maximum speed. ([Link time optimization](#) in LLVM can address some of these optimization issues, although it requires a bit of setup.)

### 3.3 The Benefits of Unsafe Rust

While C++ lacks a concept of safety, you might be wondering if “unsafe” Rust is essentially the same because you can dereference bad pointers in both. You can certainly generate a segfault in unsafe Rust, just like you can in C++. The likelihood may be lower, but you can no longer guarantee it won't happen.

However, there's still one huge benefit in using “unsafe Rust”. When your code does crash, you have a much clearer idea of where that crash is happening. Segfaults in a hybrid Rust/C++ application are caused by two things: Rust code marked as unsafe or some code on the C++ side. Since you can easily search for “unsafe” in the Rust source, it becomes easier to quickly eliminate or pinpoint the Rust code as a potential cause, helping you narrow down the source of the problem. The “unsafe” keyword also helps during code reviews since it highlights areas that merit extra attention.

## 4 Connective Tissue

### 4.1. Connecting Rust and C/C++

How do you actually call between Rust and C/C++ code? There are a lot of options to create bindings between the two languages. Let's briefly describe each and how they're most useful.

- **[Rust extern “C”](#)**: Rust has a built-in feature for accessing external code through a C FFI. It works for C or for C++

code with function prototypes marked with extern "C". This mechanism serves as the basic underlying tool that can be used to manually access functions if your code is written in simple C or if none of the following other tools can be made to work.

- [bindgen](#): This is a base tool used to build Rust APIs and structures from C/C++ headers, used by several of the following tools.
- [cbindgen](#): This tool functions in the opposite direction, generating C headers from Rust functions so that Rust functions can be called from C/C++.
- [CXX](#): This tool provides facilities for safe interoperation between Rust and C++, enabling you to define Rust functions and types that are callable from C++ and vice versa. The benefit of CXX over bindgen/cbindgen is that it is used to creating new APIs and thus limits what can be used over the FFI to things that are common between both languages, resulting in safe and straightforward interfaces on both sides.
- [CXX-Qt](#): This tool extends CXX to handle bidirectional use of Qt and Rust, which is necessary when you're trying to oxidize a Qt C++ application.
- [AutoCXX](#): Similar to CXX, AutoCXX is used to safely implement Rust and C++ interoperability, although it does this by using existing C++ headers.
- [c2rust](#): This tool converts C code into Rust code. It doesn't create bindings but aims to replace C code with Rust, which may be a better option for some smaller C libraries.
- [rustcxx](#): Mentioned for reference only, rustcxx is an obsolete project that previously allowed the inclusion of inline C++ code to Rust source files. Don't use it – it no longer compiles, and it's

not actively maintained.

## 4.2. The Microservice Model

When using Rust and C++ together, don't pass memory between the two environments; keep C++ data on the C++ side and Rust data on the Rust side. This will solve a lot of issues. Consider going a step further by making separate loops on each side that communicate via FFI calls. In other words, if your application involves Rust calling into C++, create a microservice loop on the C++ side that the Rust application can call into. Similarly, if it's C++ calling into Rust, set up a Rust microservice for the same purpose.

The point of this extra layer of abstraction is to avoid thread blocking and data ownership issues. It shifts the problem from "I want to call this C++ code" to "I want to perform this service". A further benefit is that once you isolate your C++ code into microservices – as abstracted functions with simplified APIs – they become perfect candidates for oxidizing later.

## 4.3. Rust, ABIs, and FFIs

The Rust compiler doesn't use a consistently stable application binary interface (ABI). While this might change in the future, it's how things are currently structured. Consequently, there are a few guidelines to consider:


- Create Rust applications as a single statically linked executable rather than breaking them into multiple chunks, which is common for C++ application layouts.
- Use extern C FFIs to access Rust code when you have a design pattern where software on either side of the divide may come from different Rust compilers (for example, shared libraries, plug-ins, etc).
- Compiling something as big as Rust itself is not for the faint of

heart, so use a consistent Rust version throughout your entire project, including all Rust crates. This may mean backporting libraries to an older version of Rust that's supported by your Linux build, with the exception of pieces isolated behind a C FFI, where the Rust version shouldn't matter.)

## 4.4. Build Systems

When it comes to building applications that combine Rust and C++, there are several options. We've listed these in increasing order of capability – and complexity. Select the earliest approach on this list as your project can tolerate.

- **Cargo.** If your main application is in Rust, and you're not using too much C++, then you can use Cargo (the Rust package and build manager) [to manage your whole build](#). While this is a great place to start because it's simple, it does require manual editing of C++ dependencies in a crate build script. If your project has a lot of C++ code and demands frequent script editing, you may want to consider another option.
- **CMake.** When managing more C++ code than just a couple of libraries, a small step in the right direction is to have Cargo launch [CMake](#) to handle the C++ stuff. This approach can simplify your crate build script and reduce the need for frequent modifications. CMake is also a good choice if your C++ application contains Rust pieces. In this case, you use CMake to call cargo for package management rather than build functionality.
- **Bazel/Buck2.** If your code base starts to get more complex, you could move to a build system that is aware of both languages such as [Bazel](#) or [Buck2](#). (In fact, Google created Bazel and Meta created Buck/Buck2 for just such a reason.) These systems handle both languages, treating Rust and C++ as equal entities. This results in several benefits, like consistent



---

local/remote execution and test invocation, letting you make complex and custom build systems.

- **Custom.** Of course, there's always the custom option, which is to create your own build with a mix of scripts, existing makefiles, and custom code. Given the availability of high-quality, freely available build tools, this approach is less common but worth mentioning.

---

### *What is KDAB's Software Development Best Practice series?*

*This series of whitepapers captures some of the hard-won experience that our senior engineering staff has developed over many years and projects. Offered up as a grab bag of techniques and approaches, we believe that these tips from KDAB engineers and other industry experts have helped us improve the overall development experience and quality of the resulting software. We hope they offer the same benefits to you.*

View all the parts of this whitepaper series online at: [www.kdab.com/publications/bestpractices/](http://www.kdab.com/publications/bestpractices/)

---

## About the KDAB Group

The KDAB Group is the world's leading software consultancy for architecture, development and design of Qt, C++ and OpenGL applications across desktop, embedded and mobile platforms. KDAB is the biggest independent contributor to Qt and is the world's first ISO 9001 certified Qt consulting and development company. Our experts build run-times, mix native and web technologies, solve hardware stack performance issues and porting problems for hundreds of customers, many among the Fortune 500. KDAB's tools and extensive experience in creating, debugging, profiling and porting complex applications help developers worldwide to deliver successful projects. KDAB's trainers, all full-time developers, provide market leading, hands-on, training for Qt, OpenGL and modern C++ in multiple languages.

[www.kdab.com](http://www.kdab.com)

© 2023 the KDAB Group. KDAB is a registered trademark of the KDAB Group. All other trademarks belong to their respective owners.

The KDAB logo consists of a blue speech bubble shape pointing to the right. Inside the bubble, the word "KDAB" is written in white, bold, sans-serif capital letters. To the left of the letters is a white icon of a lightning bolt or a stylized 'K'.