Findates Crate

Guilherme Nunes Kobara

University of London

CSM500 Project Proposal

Table of Contents

Abstract

Any basic calculation for financial products references a notion of time. Multiple conventions exist so that computation of time in discrete periods can be achieved. While there are multiple resources for learning the theory of financial products pricing and, the sometimes very complex, models used, a lot of those fail to address the simple practicalities of getting the hard data to be used in those models. Moreover, there is no piece of data more elemental than getting the correct schedules, day counts, fractional periods for those calculations. The Rust Library Crate presented in this proposal aims to provide some basic, but essential, functionality for some of the most common necessities when dealing with dates in a financial products context.

Keywords: Rust, quantlib, finance, dates.

Problem Statement

Computing human dates and time can be considered one of the most essential building blocks of Information Technology (IT) and Information Systems. Although relatively simple at the surface, considering the multiple time zones, daylight savings time, calendar changes, leap years, leap seconds, etc. have proved to be an immense challenge for computer scientists [1].

The financial services industry has also been dealing with the challenges of date and time arithmetic since its early days. Or even before there were any institutions like the modern banks we see today, there has been loans, borrowing and interest rates since very early days of human society. Since interest can be defined as the value of money through time [2], an accurate way of measuring such time was essential. Although Banks were some of the earliest adopters of Computers for their day-to-day operations [3], tackling calculations using date and time arithmetic had to be done in a very precise manner even before that – as any calculation on how much money to pay or receive usually is subject to a great amount of scrutiny. To guarantee the accuracy and simplify these calculations, market practitioners have adopted conventions on which calendars to use, how to count the days in a year, duration between dates, etc. Moreover, as is the case for many other attempts of creating a unified standard, various different conventions were used for different products, regions, exchanges, etc.

This large variety of conventions then had to be modelled in the computer software that was later adopted by the financial industry, but similar to the issues with calendar changes and leap years, implementation was not as simple as it looks. It is a common site, even in these days, to see a business day calendar such as the one in Figure 1 in any trading floor of a bank or a fund, giving away how software historically might have had limitations in date calculations, causing lots of traders to rely on paper calendar and turning pages.

The proposed work aims to provide a library of functions and abstractions that can efficiently and reliably deal with date and time calculations and that can be incorporated into a larger piece of software catering to the financial industry. This library should deal with any of the current conventions and make it easy to implement any new ones and be implemented in a Programming Language that allows for fast computation while not compromising safety: Rust.

Background Research

The logical starting point for research given the current project would be to check what is already implemented in Rust in relation to date and time calculations. Rust is a open-source systems programming language that puts a large focus in safety, performance and concurrency. Its central repository *crates.io* [4], where most of the libraries made freely available by its developers can be found. There we can see two library crates of interest: *time* [4] and *chrono* [5]. Both crates seem to be actively maintained and widely used – as of the time of writing *chrono* had around 98.2 millions of downloads and *time* had 155.9 millions. Both crates provide abstractions for a date, a datetime, durations and simple parsing, but *time* aims to provide only the most basic functionality while *chrono* gives some more tools and is time zone aware – there are more features in *chrono* that will be explored through the course of this work.

The next research focus would be Rust libraries that cater to financial industry needs. Rust, being a fairly new language, adoption in production systems is still incipient, with the few examples that exist more related to crypto currencies than to the more traditional sectors of the industry. For such sectors C++ still dominates[1] and the, perhaps most well known and widely used, library for quantitative finance was written in this language: *quantlib* [6]. It provides a full suite of tools for risk analysis, pricing models, optionality, mark-to-market calculations and a "Date and Time" module that is incredibly comprehensive. A port from *quantlib* to Rust exist [7] but it is not actively maintained currently and seem to be under development still. Creating bindings from C++ *quantlib* in Rust could be an alternative, but bindings in a highly sensitive and dependable programming environments can create various issues such as memory and speed overhead; hard debugging with misleading error messages; maintenance of the bindings themselves, etc. So, the date and time module in *quantlib* can serve as an inspiration for the present project, but by implementing such functionalities in Rust would mean taking advantage of its strong points while bypassing C++ known weaknesses.

In Rust, there are also two quantitative finance crates published in *crates.io*: *rustquant* [9] and *quantmath* [10]. These are still incomplete in regards to date functionalities and have a small amount of downloads. More to the point, date operations in the context of financial markets and products are so essential that they should be kept as its own library to be maintained and

[1] Although there is no official source for which language is more used and in which parts of the financial sector, this can be gauged by observing the amount of jobs postings requiring C++. Anecdotal evidence from *quants* and IT also confirms that.

developed separately from the more comprehensive framework of quantitative finance. A simpler and lighter library for dealing *only* with dates can be re-used and re-exported in multiple other contexts without the need for the more sophisticated quantitative tooling that something like *quantlib* provides. The simple data needs to be computed first before those sophisticated models can be applied.

Aims and Objectives

The main aim of the project is to develop a library in Rust for dealing with date operations that are typical for financial products valuation, risk measurement, cash flows generation, etc. This library *crate* will only deal with date and time calculations and could later be incorporated and used in a more general quantitative finance library and in any other software that deals with financial products. It should be relatively lightweight, reliable and performant.
To reach that aim, we lay out the following tasks and objectives:

1. Create an enumeration for the more commonly used day count conventions.
2. Functions for day count fractions using such conventions.
3. Create an abstraction for calendars and set operations between them.
4. Business day count and arithmetic.
5. Schedule Rule abstraction and subsequent schedule generation using this rule.
6. A simple parser for the creating Schedule rules from text.

Planned work and methodology

The major advantage of working with a modern programming language such as Rust is to make use of its rich abstractions and features, that might have been added as "bolt-on" to older languages such as C++. Enumerations, Algebraic Data types, Traits, Functional programming are all available in Rust "out of the box" and can provide an elegant and efficient way for modelling the data structures needed for this project. All these features allow for a multi-paradigm style of development with a rich set of possibilities on how to model data and functionality.

Although there is no agreed standard on what *idiomatic Rust* is, there are some key principles that the community adhere to, among others it can be cited: safety through *ownership* and *borrowing*; pattern matching; error handling using *Result* and *Option;* and higher order functions and closures. *Clippy* [11] is a very powerful linter that will be used to aid in idiomatic Rust code. *Rustfmt* [12] will also be used so that the actual code follows the conventions for spacing and naming currently adopted by the Rust community.

Rust provides automated testing facilities with its compiler (*rustc)* that can be run via *cargo*, although lightweight, the functionalities provided by *rustc* will be sufficient for unit testing of the *findates* features via the use of the *macros* and annotations it provide.

Git will be used as a version control system, with development taking place into separate branches and automated testing using Github Actions [13].

Since the main objective of the project is to provide a toolkit for dealing with dates, there will be no single API (*Application Programming Interface*) entry point for the crate nor a *build* to be delivered and automated. But rather, most of the functionalities and abstractions will be exposed, leaving internal only implementations in a separate module clearly marked as such. The functions and data abstractions exposed will be grouped into different modules according to the context in which they are used.

As mentioned previously, there are two popular and well-maintained crates that deal with time and dates *time* and *chrono* [14]. For the proposed project, *chrono* will be used so it is worth to expand on what this crate already provides and check on how some of the design choices made there can guide and influence the work on the *rustfin* crate. It already provides *enums* for days of the week and months [6]. Keeping with this, *findates* will provide *enums* for the different day count conventions, currencies settlements calendar and adjustment rules.

Actual dates are represented via *structs*: *Date* and *NaiveDate,* with the latter representing an ISO 8601 [15] calendar date and the former representing the same date but with a time zone component. Since you can easily construct a *Date* from a *NaiveDate* by specifying the time zone, *rustfin* will mostly use *NaiveDate* as the abstraction for a date and implement most of its functions and further abstractions using it.

The *NaiveDate* module in *chrono* also contain *trait* implementations for it such as *Add, Sub, Copy, Eq, Ord,* etc. aside from many convenient functions for construction, transformation and parsing. *Rustfin* will leverage most of this work and aim to not duplicate any functionality already present in *chrono* unless strictly necessary.

Documentation and examples to all available functionalities will be provided both in the form of code comments and using *rustdoc* [16], a tool that ships with the standard Rust distribution that automates the process of generating HTML, CSS and Javascript documentation. The ultimate purpose of the provided documentation will not only be to display the functionalities in *rustfin*, but also to show in which situations and contexts those can be used.

It is worth noting that Software in the financial services industry (like many others) does not exist in a vacuum: it is required to interact both with skilled and unskilled end-users, modern and legacy systems, multiple serialization formats, etc. So even though *rustfin* itself was initially conceptualized as a library to be incorporated into a larger piece of Software written in Rust, it will strive to keep to the concepts of modularity, reusability and clear API's while maintaining performance.

References

[1] Computerphile, "The Problem with Time & Timezones," Dec 30, 2013. [Online]. Available: https://youtu.be/-5wpm-gesOY.

[2] N. G. Mankiw, Principles of Economics, 8th edition, Boston, MA, USA: Cengage Learning, 2018.

[3] BCS, "When computers changed banking," 03 September 2021. [Online]. Available: https://www.bcs.org/articles-opinion-and-research/when-computers-changed-banking/.

[4] Rust Infrastructure Team, "crates.io," Rust Programming Language, [Online]. Available: https://crates.io/.

[5] J. Pratt, "time (Version 0.3.22)," [Online]. Available: https://github.com/time-rs/time.

[6] B. W. O. D. S. K. &. S. E. Maister, "chrono (Version 0.4.26) [Computer software]," [Online]. Available: https://github.com/chronotope/chrono.

[7] Q. Project, "QuantLib: a free/open-source library for quantitative finance," [Online]. Available: https://www.quantlib.org/.

[8] Piquette, "quantlib Rust," Piquette, [Online]. Available: https://github.com/piquette/quantlib.

[9] Multiple Contributors, "RustQuant," [Online]. Available: https://github.com/avhz/RustQuant.

[10] M. Rainbow, "QuantMath," [Online]. Available: https://github.com/MarcusRainbow/QuantMath.

[11] The Rust Programming Language, "Github/rust-clippy," [Online]. Available: https://github.com/rust-lang/rust-clippy.

[12] The Rust Programming Language, "Github/rust-fmt," [Online]. Available: https://github.com/rust-lang/rustfmt.

[13] Github, "GitHub Actions," [Online]. Available: https://docs.github.com/en/actions.

[14] Multiple Contributors, "Blessed - An unofficial guide to the Rust ecosystem," [Online]. Available: https://blessed.rs/crates.

[15] International Organization for Standardization, "ISO 8601 Date and time format," [Online]. Available: https://www.iso.org/iso-8601-date-and-time-format.html.

[16] The Rust Programming Language, "The rustdoc book," [Online]. Available: https://doc.rust-lang.org/rustdoc/what-is-rustdoc.html.

[17] F. M. Last Name, "Article Title," *Journal Title,* pp. Pages From - To, Year.

[18] F. M. Last Name, Book Title, City Name: Publisher Name, Year.

Figures

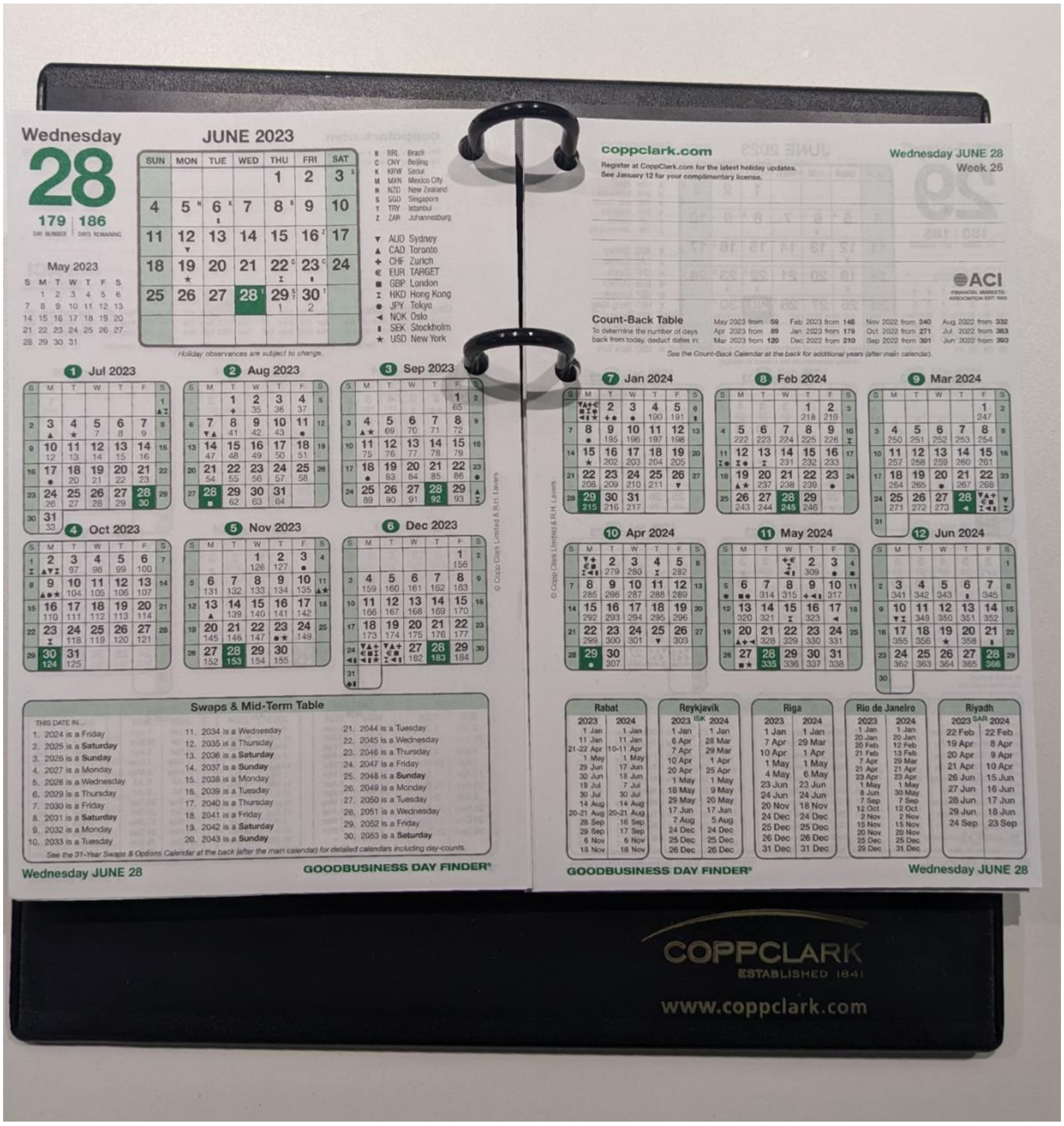

Figure 1. A Daycount calendar. Still a common sight in many trading floors.