Artifacts Available CBSoft 2025
Artifacts Functional CBSoft 2025

Approaching closures in unmanaged languages: a comparison between C++, Rust, and Swift

Luiz Romário Santana Rios
Instituto Tecgraf/PUC-Rio
Rio de Janeiro, RJ, Brazil
luizromario@gmail.com

Roberto Ierusalimschy
PUC-Rio
Rio de Janeiro, RJ, Brazil
roberto@inf.puc-rio.br

ABSTRACT

When using lambdas in managed languages, the programmer does not need to concern themselves with the disposal of the closure; in contrast, unmanaged languages that implement lambdas need mechanisms to make the closures be disposed safely at the right time. Through examples, we explore and compare the following languages: C++, Rust, and Swift. For each language, we implement examples of the usage of lambdas in increasing complexity, analyzing how they work while showcasing the implementations. After presenting the implementations, we contrast the languages regarding memory safety and typing. Finally, from those experiments, we assess some properties of lambdas in the languages and summarize them in a table, namely: safety features, safe by default, safety guarantees, verbosity, type flexibility, and memory overhead.

KEYWORDS

Programming paradigms and styles, Programming language design and implementation, Run-time environments

1 Introduction

In languages with garbage collection, the lifetime of closures is not a concern for the programmer, since memory is cleaned periodically without any programmer intervetion[1]. However, in more recent years, unmanaged languages started providing lambda expressions and closures [8, sec. 3.1.4]. As these languages do not have a background process dedicated to memory cleanup, they employ other strategies to determine when and how any given closure should be disposed of. We explored three programming languages through common applications of lambdas: **C++**, **Rust**, and **Swift**. In this article, we briefly describe how their *closures* work (Section 2), then we show some highlights of our example implementations of those applications (Section 3). After that, we compare the properties of closures in those languages based on what we learned from our implementations (Section 4), then we conclude with a summary of some assessed properties (Section 5).

2 Brief introduction of lambdas in the explored languages

Before going through the examples, we need to briefly touch on how lambdas work in each one of the languages.

C++. A very early feature of C++ is the possibility of defining "function objects"—i.e. objects that act like functions [8, Subsection 3.1.4]. The "operator call" (i.e. `operator()`) can be defined for any class as a member function of any type that takes the call arguments and returns the call result. C++'s lambdas were introduced in the C++11 standard and are syntax sugar for the creation of inline, anonymous function objects. To determine how external variables are captured and stored in the closure, C++ lambdas feature a *capture list*, where each of the variables can be explicitly listed by name and by captured by value (e.g. `[foo]`[2]), by reference (e.g. `[&bar]`) or with a custom initialization defined by the programmer (e.g. `[baz = std::move(baz)]`).

Rust. Memory is managed through a system of ownership [10, ch. 4.1] and borrowing [11, ch. 3.1] rules that are checked at compile-time. In summary, these rules determine that arguments are moved by default into functions, that there can be any number of immutable borrows, but only one mutable borrow, and that the borrower should not outlive the borrowed variable. These rules carry over to the design of lambdas. Rust allows the programmer to choose how a lambda will capture external variables, but it approaches the problem differently than C++. Instead of a capture list, Rust has two different kinds of lambdas: *borrowing* (borrows captured variables) and *moving* (moves them[3]).

Swift. It manages memory mainly through automatic reference counting [4]. By default, besides primitive values and `struct`s, all objects (including closures) are automatically and implictly reference-counted. Lambdas are reference types and automatically capture external variables by reference [5] as well[4].

3 Exploring closures through examples

For each of the studied languages, we explored the properties of closures through the implementation of three examples: counter function, regions as predicates, and the Z-combinator. All examples are fully available as artifacts, with comments all throughout the code explaining the rationale behind the implementations step by step (see the ARTIFACT AVAILABILITY section at the end of the paper). In this section, we comment the highlights we found in these implementations.

3.1 Counter function

Create a function that returns an incrementing value at each call (i.e. 0, 1, 2, 3, ...). The goal of this example is to illustrate how the language deals with lambdas that access variables past the end of their scope.

[1]Some mechanisms to control the process of garbage collection might be provided to the programmer by the language, but they are entirely optional.

[2]`foo` is copied into the closure

[3]Types that implement the **`Copy`** trait (e.g. primitive numbers) are copied into the closure.

[4]Though they are not explored in depth in this article, Swift lambdas do have *capture lists* [6, Capture Lists] for cases when e.g. capturing a weak pointer is necessary to avoid reference cycles.

The following is C++'s simplest implementation of a function that returns a newly-created counter function:

```
auto mk_counter() {
  int n = 0;
  return [=]() mutable { return n++; };
}
```

Although simple, this already has many details the programmer needs to pay attention to in order to properly use the lambda. The need for **mutable** is one of those[5], but the most sensible detail here is the capture list itself: if we captured by reference (i.e.: `[&]` instead of `[=]`), this would compile *without any relevant warnings*, letting the program modify and access a reference to a dead temporary variable—leading to undefined behavior.

Rust's safety would prevent a problem like this—since a closure outliving a reference it captures causes a compile error–but they also can potentially make programs much more verbose, as is made clear by the split-counter variant of the counter in Rust[6]:

```
fn mk_counter() -> (impl Fn(),
                    impl Fn() -> u64) {
  let count = Rc::new(Cell::new(0));
  ({ let count = count.clone();
     move || count.set(count.get() + 1) },
   { let count = count.clone();
     move || count.get() })
}
```

3.2 Regions as predicates

Define regions as predicates that return true if a given point is inside the region. We define simple primitive regions (e.g. circles) as well as operations to combine multiple regions into one. This example demonstrates the total transparency of data storage in closures.

This example highlights the efficiency of Rust's and C++'s closures. In both cases, the closure is stack-allocated and has the size of all captured variables. Swift's closures are dynamic objects and are necessarily heap-allocated and reference-counted.

An interesting feature of Swift is the `@escaping` attribute. It indicates that an argument that is a function *escapes*—that is, the function that takes it brings it elsewhere instead of executing it locally. For example: the original region passed to `outside` is captured by the resulting closure, so it needs to be escaping:

```
func outside(
  _ region: @escaping (Point) -> Bool)
  -> (Point) -> Bool {
```

3.3 Z-combinator

The Z-combinator (from now on called just "Z") is an eager fixed-point combinator—preferred over the Y-combinator in eager languages. Roughly speaking, it takes a function and applies it to itself and it can be used to build recursive functions without using recursion. The complex interaction between the combinator's and the function's closures, which recursively reference each other, poses an important challenge to the languages being compared. As our example, we tried implementing Ackermann's function using Z.

C++ was the only language with a clean, direct Z implementation. This implementation takes advantage of generic lambdas and type deduction to avoid dealing with the typing of functions:

```
auto z(auto le) {
  auto a = [le](auto f) {
    return le([f](auto... args){
      return f(f)(args...);
    });
  };
  return a(a);
}
```

Both Rust and Swift suffered from the same problem: the typing of lambdas *must* be explicit, but, since Z takes a function that takes itself as an argument, it would require a recursive type—a feature all languages here lack. C++ escapes this problem because of a unique feature: the type of closures is unrelated to the parameters it takes.

4 Comparing the properties of closures

Through these examples, we explored many strengths and weaknesses of each language in their approach to closures.

4.1 Memory safety

C++. Though it provides some tools that enable the programmer to safely manage memory[7], at every turn, safety must be an active choice. Even simple scenarios that look inoffensive might be unsafe, but valid. Capture lists help, but they can be easy to misuse.

Rust. In total contrast to C++, its ownership and borrowing rules give very strong memory safety guarantees. However, those rules can make even simple uses of lambdas become overwhelming.

Swift. The main approach to safety is making every object that is neither a primitive nor a **struct** be reference-counted. This makes the examples in Swift usually more succint than their counterparts and, in particular, safer by default than C++. But this approach does not bring safety *guarantees*—e.g. the programmer still needs to watch out for reference cycles.

In summary. From least safe to safest:

- **C++** provides safety features, but they need to be an active choice of the programmer;
- **Swift** is safer by default, allowing less safety only if the programmer actively chooses it, but it does not protect the programmer from reference cycles that can lead to leaks;
- **Rust** gives very strong safety guarantees, but ensuring such safety leads to a lot of verbosity, particularly when dealing with lambdas.

4.2 Typing

C++. Its generic typing through templates is the most flexible of the explored languages, since it allows for the concrete types to be completely omitted in the generic functions. This flexibility allowed the C++ implementations to be the closest to the reference

[5]The compilation fails without **mutable**, since variables captured by value are immutable by default.
[6]Instead of the same function returning the value and incrementing it, incrementing and getting is done by two separate functions.
[7]`std::shared_ptr`, capture lists, RAII in general.

Scheme implementations, especially in the z-combinator example, where being able to omit types made it possible to declare a lambda that takes itself as an argument—something that would generate an infinitely recursive type.

Rust. Making generic functions that infer the argument types and the return type is possible through the **`impl`** keyword. The effect is similar to C++'s **`auto`**, but `impl` is more restrictive, as specifying a trait that the type should conform to is mandatory. Also, lambdas that take other functions as arguments have to explicitly specify the type of the function they take—even though the types of lambda paremeters can otherwise be omitted. As a consequence, it is impossible to declare a lambda that takes itself as an argument, forcing us to work around that limitation.

Swift. While it does have generics, that feature was not relevant for dealing with closures, as their type is a shared, concrete, reference type—in contrast with C++ and Rust, where each closure has a unique type. This simplifies the typing of functions in Swift at the cost of reference-counting overhead that is not present in C++ and Rust.

In summary. From least to most flexible:

- **Rust**'s rigid typing and safe defaults guarantee memory safety and efficience, but makes it particularly cumbersome to deal with simple closure scenarios;
- **Swift**'s choice to wrap most objects in reference counted pointers makes it easier to reason about closures at the cost of memory efficiency;
- **C++**'s template system allows a high degree of flexibility when dealing with lambdas, while maintaining memory efficience, as the generic functions can work directly with the concrete type of the lambdas without needing any wrappers.

5 Conclusion

With the exploration done in this article, we were able to summarize some properties of lambdas in C++, Rust, and Swift in Table 1. Some noteworthy properties are the following:

- All three languages offer features that enable memory safety, suggesting this is crucial (or at least very important) for closures to work in unmanaged languages;
- Rust is the safest language of the three, but this safety costs readability, as it is by far the most verbose of them;
- The memory overhead Swift imposes on closures suggests it is not in the same level of memory efficiency as C++ and Rust;
- C++ and Swift offer capture lists for greater control of the closures, but Rust has a different strategy, suggesting that more than one strategy might work to solve variable capturing.

5.1 Further considerations

We did not explore the interaction between closures and objects[8]. Such interactions could bring to light better fitting strategies for unmanaged closures. Though there have been comparisons of unmanaged languages to a wider range of managed languages [8, ch. 5], there is a number of other unmanaged languages that have first-class functions in some way that could be compared to the languages in this article: Objective-C [7] (Swift's ancestor), Zig [2], Odin [1], and even C[9].

ARTIFACT AVAILABILITY

All code examples used as the basis of this comparison are available at the Open Science Framework platform [3].

REFERENCES

[1] Bill, G. Odin programming language. https://odin-lang.org/. Accessed in June 15th, 2025.

[2] Foundation, Z. S. Zig programming language. https://ziglang.org/. Accessed in June 15th, 2025.

[3] Ierusalimschy, R., and Rios, L. R. S. Approaching closures in unmanaged languages: a comparison between c++, rust, and swift (sblp 2025 short article's artifacts). https://doi.org/10.17605/OSF.IO/CWKNU, 2025.

[4] Inc., A. Automatic reference counting — the swift programming language (swift 5.4). https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html. Accessed in May 22nd, 2021.

[5] Inc., A. Closures — the swift programming language (swift 5.4). https://docs.swift.org/swift-book/LanguageGuide/Closures.html. Accessed in May 21st, 2021.

[6] Inc., A. Expressions — the swift programming language (swift 5.4).

[7] Inc., A. The objective-c programming language. https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html, 2013. Accessed in June 15th, 2025.

[8] Rios, L. R. S., and Ierusalimschy, R. A survey of function values in imperative programming languages. Master's thesis, Pontifícia Universidade Católica do Rio de Janeiro, 2019.

[9] Team, T. G. The gtk project - a free and open-source cross-platform widget toolkit. https://www.gtk.org/. Accessed in June 15th, 2025.

[10] Team, T. R. The rust programming language. https://doc.rust-lang.org/stable/book/2018-edition/, 2018. Accessed in April 18th, 2019.

[11] Team, T. R. The rustonomicon. https://doc.rust-lang.org/nomicon/, 2021. Accessed in May 21st, 2021.

[8]Lambdas in C++ being a special case of function object might hint at a deeper relationship between the two.

[9]It would be interesting to compare the approaches presented here to the manual approaches to closures present in many C libraries such as GTK+ [9].

	Safety features	Safe by default	Safety guarantees	Verbosity	Type flexibility	Memory overhead	Capture list
C++	**Yes**	No	No	Medium[1]	High	**No**	Yes
Rust	**Yes**	**Yes**	**Yes**	High	Low	**No**	No
Swift	**Yes**	**Yes**	No	Medium[1]	Medium	Yes	Yes (optional)

Table 1: Summary of the properties of lambdas in C++, Rust, and Swift according to what was seen in the examples explored in this article. [1]Reference-counting by default makes Swift less verbose than C++ in some scenarios (counter function), but the more flexible typing makes C++ less verbose than Swift in others (z-combinator), so they are both being placed in "Medium" verbosity.