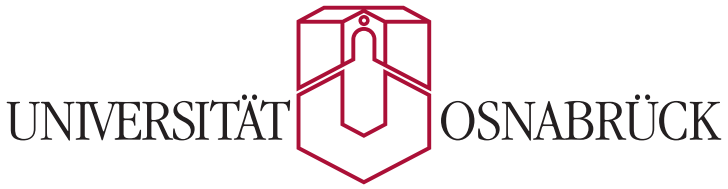

Embedded Software Systems

Seminar
The Rust Programming Language

Summer Semester 2023

Structs, Enums, and Patterns: Structuring and Matching Data in Rust

July 2023

Contents

1 Introduction

Rust is a modern, general-purpose programming language with several unique features that make it stand out from other languages. Rusts memory safety and its focus on performance make it suitable for scenarios where performance and reliability are essential. Over recent years, Rust has gained prominence and popularity [1] and is being used in areas where other languages like C or C++ have been used in the past [2] [3] [4]. Rust currently gets adopted into both the Linux [5] and the Windows Kernel [6]. Rust's structs, enums, and patterns provide powerful tools for organizing and manipulating data. Understanding these core components is essential for working with data in Rust and using the language effectively. This paper will explore these concepts and how to work with them. They will be compared to their C++ counterparts to examine how they might be similar or different.

2 Structs in Rust

The core principle of structs in Rust is similar to the concept of a struct in C++ or basic classes in languages like Java. A struct groups multiple values of potentially different types into a single type and gives them a name. This allows for the creation of custom, more complex types. Rust differentiates between three types of structs: *named structs*, *tuple structs*, and *unit structs*. [7, Chapter 6.6]

2.1 Named Structs

Named structs consist of named fields, each of which can have a different type. Code 1 shows the basic syntax of defining a named struct using the `struct` keyword. The naming convention for structs in Rust is to use *PascalCase* for the name of the struct and *snake_case* for the fields and methods of the struct [8]. A struct can be instantiated using a struct expression that follows the `StructName { field_name1: value, field_name2: value }` syntax. If a variable name is the same as a field name, Rust allows for a shorter expression by using the *field init shorthand* syntax `StructName { field_name1, field_name2 }` instead [9, RFC 1682]. After instantiation, each field can be accessed using the `.` operator and its name [10, Chapter 5.1]. Code 1 shows an example of the usage.

```rust
struct Person {
    name: String,
    age: u8,
    height: f32,
}

fn main() {
    let height = 1.80;
    let person = Person {
        name: String::from("John"),
        age: 42,
        height,
    };
    println!("{} is {} years old.",
        person.name, person.age);
}
```

Code 1: Basic Syntax of a named struct in Rust

Rust allows for another shorthand syntax when instantiating a struct. Using the `..` operator, a new struct can be created from an existing one using the *struct update* syntax. This will assign the fields of the new struct the same values as the fields of the existing struct. If the struct has fields that cannot be copied, the values of the old struct will be moved into the new struct, and the old struct will be invalidated. The `..` operator initializes the remaining fields and thus must be the last part of the struct expression [10, Chapter 5.1]. Code 2 shows an example of the struct update syntax.

```
let person2 = Person {
    name: String::from("Doe"),
    age: 42,
    ..person
};
```

Code 2: Struct update syntax in Rust

2.2 Tuple Structs

Tuple structs are similar to named structs, but their values, called *elements*, are unnamed. They can be used to group values of different types into a single type and give them a meaning by naming the struct itself without naming the values individually. This can be used when the name of the struct and the order of the elements is enough to describe their meaning. For accessing the elements of a tuple struct, the `.` operator is used in the same way as for named structs, with the difference that the elements are addressed by their index instead of their name. This means that the order of the fields is important, and a tuple struct is, therefore, very similar to the tuple type [10, Chapter 5.1] [11, pp. 212–213]. Code 3 shows the basic syntax of creating and accessing a tuple struct. If we would define another struct with the same fields as `Complex`, it would not be the same type, and attempting to compare them or use one of them where the other is expected would result in an error.

```
struct Complex(f64, f64);

fn main() {
    let z = Complex(1.0, 2.0);
    println!("Complex: ({}, {})",
        z.0, z.1);
}
```

Code 3: Syntax of a tuple struct in Rust

2.3 Unit Structs

A unit struct has no elements, holds no data, and therefore occupies no space in memory. To define a unit struct in Rust, the `struct` keyword is used, followed by the struct name and a semicolon, without enclosing curly braces. To instantiate a unit struct, the struct name is used. A unit struct can be helpful when working with *traits*. Traits allow for the definition of shared behavior in an abstract way that is *interfaces* in C++. However, they will not be further elaborated on, as they are not within the scope of this paper [11, p. 213] [10, Chapter 10.2].

```
struct UnitStruct;

fn main() {
    let unit_struct = UnitStruct;
}
```

Code 4: Syntax of a unit struct in Rust

2.4 Functionality on Structs

After covering the fundamentals of structs in Rust, the following section will examine how to add functionality to structs.

2.4.1 Methods

Methods are very similar to functions. They differ from functions in that they are associated with an instance of a struct. In order to implement a method for a struct, an *implementation block* is used. The `impl` keyword, followed by the name of the struct, is used to define an implementation block. Methods can then be defined inside the curly braces of the implementation block. They are defined in the same way as functions, with a name, parameters, and a return type, but they are special functions in that they always take `self` as their first argument. Within an implementation block, `Self` is shorthand for the name of the struct the implementation block is for. Each method has to take either `self: Self`, `self: &Self`, or `self: &mut Self` as its first parameter. `self`, `&self`, and `&mut self` are then abbreviations for these three variables with their respective types. To access members of the struct within a method, `self` has to be used explicitly, followed by the `.` operator and the name of the field [11, pp. 214–215]. Code 5 shows an example of implementing a method for a struct. Calling a method is done by using the `.` operator on an instance of the struct followed by the name of the method and the arguments in parentheses.

```
impl Person {
    fn say_hello(&self) {
        println!("{} is {} years old.",
            self.name, self.age);
    }
}
```

Code 5: Syntax of a method in Rust

2.4.2 Associated Functions

Each function declared within an implementation block is called a *type-associated function* or *associated function* for short. These functions do not take `self` as their first argument and are, therefore, not associated with a specific instance and instead are associated with the type itself. They can be used like static functions in other languages. An associated function can be called using the `::` operator on the struct's name followed by the name of the function [11, pp. 219–220]. Code 6 shows an example of creating and calling an associated function.

```
struct Circle;

impl Circle {
    fn area(r: f64) -> f64 {
        std::f64::consts::PI * r * r
    }
}
fn main() {
    let area = Circle::area(2.0);
    println!("The area is {}.", area);
}
```

Code 6: Syntax of an associated function in Rust

2.5 Visibility of Structs

In Rust, by default, all fields, methods, and functions of a struct are private and can only be accessed within the same module. If a struct is marked as `pub`, it can be used from outside the module, but its fields, methods, and functions remain private. In order to access the fields of a struct from outside, each field that should be accessible has to be marked as `pub`. Private fields can still be accessed by functions and methods associated with the struct. Therefore, controlled access to private fields using functions and methods can be implemented. If at least one field of the struct is private, the default struct initialization will be private as well, as private fields cannot be accessed or initialized from outside, and Rust does not support partial initialization. Helper functions are then needed to create instances of the struct. Functions and methods can be marked as `pub` as well and can be accessed from outside the module, as long as the struct itself is marked as `pub` [11, pp. 210–211] [10, Chapter 7.3] [7, Chapter 12.6]. Appendix Code 14 presents an example of a struct with public and a struct with private fields.

2.6 Comparison with C++

The core concept of structs in Rust is to group data by creating a new type and the possibility to add functionality to this type. C++ has structs and classes that both serve the same purpose. The only difference between these two in C++ is that structs default to a public access level while classes default to a private access level. In Rust, all fields of a struct and the struct itself are private by default. Therefore, structs in Rust are in this aspect more similar to classes in C++ than to structs in C++, but not equivalent as the concept of visibility slightly differs between Rust and C++ [11, p. 187]. The syntax of structs in Rust is similar to that of structs in C++, but there are some minor differences. Rust struct's functions and methods are defined within an `impl` block and are separated from the struct definition. This allows for keeping the data and the functionality of a struct visually separated. Furthermore, `impl` blocks are not limited to structs, and the separation allows for a more general and similar syntax for different concepts in Rust [11, p. 220]. In C++, functions and methods can be defined within the struct definition. Traditionally, they are defined in a header file, while the implementation is provided in a separate source file, which allows for a similar separation [12, pp. 30–33]. Another difference is that methods in Rust take `self` as their first argument, which is not the case in C++. Inside a method, C++ allows for implicit access to the fields of the struct, while in Rust, `self` has to be used explicitly to access the fields. One key difference between the two languages is that classes/structs in C++ allow for inheritance, while structs in Rust do not but can implement *traits* instead. This is an intentional design decision of the Rust language, but will not be covered, as it is not a focus of this paper. While classes/structs in C++ have implicit or explicit constructors, this is often done in Rust by implementing an associated function that returns an instance of the struct, as the language does not support constructors, as C++ does. Instead of implementing a destructor, as done in C++, the `Drop` trait can be implemented along with a `drop` method

that will be called when the instance of the struct goes out of scope. Rusts associated functions are very similar to static functions in C++. While Rust does not have a direct equivalent to the static variables of a C++ class, Rust has *associated consts* that can be defined within the given `impl` block [11, pp. 220–221].

3 Enums

Enmus allow defining custom data types with a set of named values, called *variants* [11, p. 230]. At any given time, a variable of this type can be exactly one of its variants. This makes enums a versatile tool for modeling multiple scenarios, from representing different states of an object to encoding different options for a function. A basic enum in Rust resembles a C-like enum and is defined by the `enum` keyword followed by the name of the enum and a list of variants. By default, the first variant has the value 0, the second one the value 1, and subsequent variants increment by one.

```
enum Status {
    Valid,
    Error(String),
    Warning { message: String,
              code: i32 },
    Pending(i32, String),
}
fn main() {
    let valid = Status::Valid;
    let warning = Status::Warning {
        message: String::from("Warn!"),
        code: 42,
    };
}
```

Code 7: Enum with data

Each variant can be assigned a custom integer value that has to be unique. Casting from such an enum to an integer is possible, but the reverse is not, as Rust guarantees that the value of an enum is always a valid variant. In Rust, enums can optionally hold data. Each variant can be one of three types: a unit variant, a tuple variant, or a struct variant. The unit variant has no additional data associated with it, as seen in Code 8. The tuple variant holds a tuple of values and is defined just like a tuple struct. The struct variant holds a named struct of values and is defined by a name followed by curly braces containing the fields of the struct, just like a named struct. These variants can then be used like their struct counterparts [7, Chapter 6.7] [11, pp. 230–234]. Code 7 shows an example of an enum with data.

```
enum Direction {
    Up,         // 0
    Down,       // 1
    Left = 10,  // 10
    Right,      // 11
}
impl Direction {
    fn is_up(self) -> bool {
        self as i32 ==
            Direction::Up as i32
    }
}
fn main() {
    let up = Direction::Up;
    let down = Direction::Down;

    println!("Is up? {}", up.is_up());
    println!("Is down? {}", down.is_up());
}
```

Code 8: Syntax and functionality of enums

3.1 Functionality on Enums

Just like structs, enums can implement functionality. The syntax for implementing and using enum functionality is the same as for structs. The only difference is that an enum does not have fields but exactly one variant. This variant can be accessed using the `self` keyword [10, Chapter 6.1]. Code 8 shows an example of such a method.

3.2 Result and Option

Rust has two built-in enums often used, `Result` and `Option`, which will be briefly discussed as they show how enums can be used in practice. `Result` is a *generic* enum that has two variants, `Ok` and `Err`. The `Ok` variant is used to signal that the operation was successful and holds the result of the operation, while the `Err` variant is used to signal that the operation failed and holds an error message. `Option` is also a *generic* enum that has two variants, `Some` and `None`. Contrary to many other languages, Rust does not have a `null` value but instead uses `Option` to represent the absence of a value. The `Some` variant holds the value, while the `None` variant is used to signal that there is no value. This makes it similar to a value that can be `null` in other languages or pointers that can be `nullptr` in C++. Unlike C++, Rust forces to check whether a value is present before using it. This has the advantage that there will not be null pointer dereferences at runtime, and only variables of type `Option` have to be checked for `None`, as other accesses are guaranteed to be safe [11, pp. 236–237] [10, Chapter 6.1]. The Patterns and Control Flow section will show how the data within enums can be accessed. Similar to Rust's `Option`, C++ has a `std::optional` type that can be used to represent that a value might not be present. It can be checked if a value is present using the `has_value()` method and accessed using the `value()` method, which will throw an exception if the value is not present. As the `std::optional` type exists in addition to pointers that can be `nullptr`, it does not have the same advantages as Rust's `Option` type [15].

```rust
enum Result<T, E> {
    Ok(T),
    Err(E),
}

enum Option<T> {
    None,
    Some(T),
}
```

Code 9: Result and Option enums [13] [14]

3.3 Comparison with C++

The fundamental concept of enums in Rust and C++ is mostly the same, but some differences exist. In C++, enums can hold other types than integers by changing the underlying type of how the enum is stored. They are limited to *integral types*, which means that enums can represent, for example, a `char` or a `short`, but not a `float` or a `double` [16, pp. 32, 834]. Rust enums can hold any type of data and even different types in different variants. Furthermore, enums in Rust can implement functionality, while enums in C++ cannot. Enums in Rust are more powerful than they are in C++, as they extend on the basic functionality of enums in C++, but this does not mean that C++

cannot achieve similar behavior. C++, for example, has the `std::variant` type, which can hold one of a fixed set of types. Appendix 13 shows how `std::variant` can be used to achieve a very similar behavior to Rust's enums that hold data [12, pp. 209–210].

4 Patterns and Control Flow

As the previous section shows, Rust allows enums to hold various data types. The control flow construct `match` can be used to determine the current variant of an enum, access its associated data, and perform various actions accordingly. Looking at Code 10, the syntax of `match` looks close to a `switch` statement in other languages would, as it takes an expression and then checks it against several *patterns*. It is evaluated from top to bottom, and only the arm of the first pattern that matches the expression is executed. The body of such a match arm is an expression, and the value of the matching arm's expression is then returned as the value of the whole match expression.

4.1 Matching Types of Patterns

The elements of a pattern can be deconstructed into one of the following groups: *literals*, *destructured arrays/enums/structs/tuples*, *variables*, *wildcards*, and *rests* [10, Chapter 18] [7, Chapter 9]. They will be covered in more detail in the following sections.

```
let number = 7;
match number {
    1 => println! ("One"),
    2 | 3 | 5 | 7 | 11 => {
        println! ("Prime")
    },
    13..=19 => println! ("Teen"),
    x => println! ("{x} not special"),
}
```

Code 10: Example match expression [17, Chapter 8.5]

4.1.1 Literals, Variables, and Wildcards

Literals are the simplest form of patterns. They only match against a fixed value evaluated at compile time and are, for example, integers, chars, or booleans [7, Chapter 8.2.1]. Using only literals makes the `match` expression behave similarly to a `switch` statement.

Using a variable as a pattern does not mean comparing the value of the variable against the value of the expression but instead binding the value of the expression to a newly created variable. This means that the variable will then hold the value of the expression, and therefore the original value might have been moved or copied, depending on the type of the expression. Variables bound by a pattern can be used inside the body of the matching arm and shadow variables of the same name in the outer scope. Code 10 shows an example where the value of `number` is bound to the variable `x` and then printed in the body of the matching arm.

The wildcard pattern is an underscore `_` and matches against any value. It differs from a variable in that it does not bind and therefore does not move or copy the given value [10, Chapter 6.2, Chapter 18.3].

4.1.2 Destructuring

Destructuring arrays, enums, structs, and tuples is a way of matching against the individual elements of these types. Using `let` statements, that take patterns as well, with the syntax `let pattern = expression`, makes it possible to assign multiple values at once by using destructuring. Function parameters take patterns as well, allowing the passed arguments to be destructured. When destructuring, the *rest pattern* `..` can be used to match against remaining elements. This pattern is sometimes called *wildcard* or *wildcards* while the underscore `_` is then called *placeholder* [18]. It has to be unambiguous what the rest pattern matches against, and it can therefore only be used once in a pattern [11, pp. 243–244] [10, Chapter 18.3] [7, Chapter 9]. Appendix Code 15 shows examples of destructuring and how the rest pattern can be used.

4.2 if let, while let and for Loops

`match` expressions must be exhaustive, which means that they have to cover all possible cases of the given expression. This has the advantage that the compiler can check if all cases are handled and warn if a case was forgotten [10, Chapter 6.2]. Sometimes, only certain cases need to be handled and the rest can be ignored. A `match` expression could be used to check for all cases that need to be handled, and then the wildcard pattern `_` could act like a default case to ignore the rest, but this is not very concise. Rust provides the `if let` expression that allows checking for a single case and ignoring the rest or chaining multiple expressions together. The `if let` expression takes a pattern, followed by an equals sign and an expression. It checks if the pattern matches the expression and then executes the body if it does. Just like a match arm would, `if let` expressions introduce new local variables, shadowing variables of the same name in the outer scope, that can be used in the body. The `if let` expression can be followed by an `else` that acts like the `else` of an `if` expression would, but also allows for another `else if let` or `else if`. This gives more flexibility than a `match` expression would, as these expressions do not have to be related to each other and can use different patterns and expressions [10, Chapter 6.3]. `while let` follows a very similar meaning to `if let`, in that it takes a pattern and an expression and executes the body as long as the pattern matches the expression. `for` loops also take patterns, which allows destructuring arrays, vectors, slices, and other types as part of the loop [10, Chapter 18.1]. Code 11 shows examples of `if let`, `while let`, and `for` loops.

```rust
let option = Some(5);
let other = false;

if let Some(value) = option {
    println!("Value: {}", value);
} else if other {
    println!("Other is true");
} else {
    println!("Option is None");
}

let mut vec = vec![1, 2, 3, 4, 5];
while let Some(value) = vec.pop() {
    println!("Value: {}", value);
}

let a = [1, 2, 3, 4, 5];
for (i, num) in a.iter().enumerate() {
    println!("{}: {}", i, num);
}
```

Code 11: if let, while let and for loops

4.3 Refutable and Irrefutable Patterns

Patterns can be categorized into two groups, *refutable* and *irrefutable*. An irrefutable pattern is a pattern that will always match, no matter what the expression is, and a refutable pattern, on the other hand, might not match the given expression. Whether a refutable or irrefutable pattern can be used depends on the context. As seen before, the `let` statement takes a pattern. In this case, only irrefutable patterns are allowed, as there is no logical way to handle a pattern that might not match. The same goes for `for` loops and function parameters. The `if let` and `while let` expressions allow for irrefutable patterns, but the compiler will give a warning, as these expressions are meant to handle patterns that might not match. The `match` expression allows for one irrefutable pattern used in the last arm, similar to a default case, but all other arms must be refutable [11, pp. 142–144, 250] [10, Chapter 18.2].

4.4 Matching multiple Patterns at once

Rust allows for matching against multiple patterns simultaneously by using the `|` operator and concatenating the patterns. This is useful if multiple different values should be matched against but then proceed in the same way. If an integer should be either 1, 2, or 3, the pattern `1 | 2 | 3` can be used, but Rust allows for an even more concise way of doing this. The `..=` operator can be used to match against a range of values. The pattern `1..=3` matches against all values between 1 and 3, including 1 and 3 [11, p. 248].

4.5 Refine Matches with Guards

Sometimes, a match arm needs to check for something that cannot be expressed using only the patterns presented. In this case, a *match guard* can be used, which is an additional `if` condition that must also be true for the arm to match. Earlier, it was shown that using a variable inside a pattern does not mean that the variable's value is matched against, but rather that the value of the expression is bound to the variable. Using a match guard allows comparing against the variable's value. Variables bound in the pattern can then be used in the match guard [7, Chatper 8.2.16]. Code 12 shows an example of a match guard.

```rust
let opt = Some(5);
let number = 7;

match opt {
    Some(x) if x == number => {
        println!("x = {:?}", x);
    },
    _ => println!("Default case"),
}
```

Code 12: Match guard

4.6 @ Bindings

The `@` operator allows binding the value of a matched expression to a variable. By using `x @ pattern`, the pattern can be matched against, and on success, the value will be

bound to `x`. This could, for example, be useful if a struct of a certain type should be checked for inside a given expression and then proceed using the given instance. Without the `@` operator, all the fields of the struct would have to be matched against, and then the struct has to be rebuilt with the given values. This is not always possible, as there might not be a way to construct a new instance from the given scope. Using the `@` operator, the struct can be matched against using the pattern `StructName(..)`, and then the variable that holds the struct can be used to proceed. Appendix Code 17 shows an example of how this can be used. Furthermore, it can be helpful when using ranges. If a number should be checked for being between 1 and 3, the pattern `x @ 1..=3` can be used, and then the variable `x` can be used to proceed [11, pp. 248–249] [10, Chapter 18.3].

4.7 Comparison with C++

C++ has a `switch` statement that is similar to the `match` expression and allows for matching against several different values [16, pp. 178–179]. `switch` statements are limited to matching against a single value and do not allow for a system similar to Rust's patterns and additional features. C++ does not have a feature similar to Rust's patterns. Over recent years, features like *structured bindings* that allow for destructuring have been added to C++ that make it easier to achieve similar results in some places [19, pp. 41–42]. Appendix 16 shows how this can be used. There are currently proposals to add pattern matching to C++ [20] [21], and similar features can be archived using external libraries [22]. However, there is currently no counterpart to Rust's pattern matching in C++.

5 Memory Layout

This section will briefly examine how Rust handles the memory layout of structs and enums. Both structs and enums have a *representation* that defines how the data is laid out in memory. The *representation* can be the *default representation*, also called *rust representation*, the *C representation*, the *primitive representation* or the, for us less important, *transparent representation* [7, Chapter 10.3].

5.1 Structs

Structs allow for the default representation and the C representation. The default representation is the default and guarantees that each field is properly aligned, that the fields do not overlap, and that the alignment of the struct itself is at least the same as the maximum alignment of its fields. The alignment defines that the address of the field is always a multiple of that alignment value. One thing to note is that this representation does not guarantee that the fields are laid out in the same order as defined in the struct or specify any order or padding. This might lead to added padding between fields, but depending on multiple factors, not necessarily. Furthermore, the compiler is free to reorder the fields in any way, which could be used for optimization purposes, as

long as the guarantees are upheld. This makes it unsuitable for cases where the order and predefined addresses of the fields are important. The C representation is more appropriate in this case, as it follows the same rules as structs in C and C++. This most notably means that the fields are laid out in the same order as defined in the struct [19, pp. 308–310].

Rust allows us to specify *alignment modifiers* that can specify that a struct should be *packed*, which means that no additional padding is added between fields. This can also be done in C++, using compiler directives. Changing the representation and packing of a struct is done by adding the `#[repr()]` attribute to the struct's definition and specifying the representation as an argument (e.g., `packed`, `C`) in the round brackets [11, pp. 213–214] [7, Chapter 10.3].

5.2 Enums

Enums allow for the default representation, the C representation, and the primitive representation. The default and C representations are equivalent to the ones for structs, and the primitive representation can be combined with one of the other. Enums have an additional *tag* field, which is used to identify which variant of the enum is currently stored. The tag field has to be at least the size of the smallest integer type that can hold all the variants of the enum and can be adjusted using the primitive representation by adding `#[repr(u16)]`, `#[repr(u32)]`, ... to the enum definition. The tag might then be followed by padding and by the data of the variant, just like in a struct. As Rust does not make any more guarantees in the default representation, as stated above, there is no guarantee of how the enum is laid out in memory by default. In general, its size is mostly the size of the tag field plus the size of the largest possible variant, including padding [11, pp. 233–234]. Suppose the used variant is smaller than the largest possible variant. In that case, the enum will still take up the same amount of space as if the largest variant was used, and the additional space will be unused [23, 3.3 Enums] [7, Chapter 10.3].

6 Conclusion

Rust and C++ exhibit significant similarities in handling structs and enums, sharing similar syntax and core concepts. The most notable difference is that Rust does not allow for inheritance, which is a big part of C++ and a concept that was intentionally left out of Rust. Rust extends these core concepts by adding more features, like enums that can hold data or the ability to implement functionality on enums. Notably, Rust's pattern-matching syntax and control flow constructs surpass the capabilities of the switch statement in C++, offering a broader range of possibilities beyond simple value comparisons. Regarding memory layout, Rust, in practice, defaults to a more efficient layout than C++ but still allows control over the layout if needed.

In conclusion, Rust offers an equally powerful alternative to C++, offering improved default behaviors and introducing new approaches to more advanced tasks.

Appendix

```cpp
#include <iostream>
#include <variant>

struct Error
{
    std::string message;
};

struct Warning
{
    std::string message;
    int code;
};

using Status = std::variant<Error, Warning>;

void printStatus(const Status &status)
{
    // check for variant and act accordingly
    if (auto value = std::get_if<Error>(&status))
    {
        std::cout << "Error: " << value->message << std::endl;
    }
    else if (auto value = std::get_if<Warning>(&status))
    {
        std::cout << "Warning: " << value->message
                  << " (" << value->code << ")" << std::endl;
    }
}

int main()
{
    Status status = Error{"Error while reading file"};
    printStatus(status);
    status = Warning{"File not found", 404};
    printStatus(status);
    return 0;
}
```

Code 13: Using `std::variant` to emulate a Rust enum that holds data in C++

```rust
mod my_module {
    pub struct Person {
        pub name: String,
        pub age: u32,
    }
    impl Person {
        pub fn get_name(&self) -> &String {
            &self.name
        }
    }

    pub struct Vector(f64, f64);    // these fields are private by default
    impl Vector {
        pub fn new(x: f64, y: f64) -> Self {
            Vector(x, y)
        }
        pub fn get_xy(&self) -> (f64, f64) {
            (self.0, self.1)
        }
    }
}
fn main() {
    let p = my_module::Person {
        name: String::from("John"),
        age: 32,
    };
    println!("Name: {}", p.get_name());

    let v = my_module::Vector::new(1.0, 2.0);
    println!("Vector: {:?}", v.get_xy());
}
```

Code 14: Visibility of structs in Rust

```rust
struct Point {
    x: i32,
    y: i32,
    z: i32,
}

fn main() {
    let triple = (0, -2, 3);
    let (x, y, z) = triple;
    // x: 0, y: -2, z: 3

    match triple {
        (0, y, z) => println!("x: 0, y: {y}, z: {z}"),
        (1, ..) => println!("x is 1, rest doesn't matter"),
        (3, .., 5) => println!("x: 3, z: 5"),
        _ => println!("It doesn't matter what they are"),
    }

    let p = Point { x: 0, y: 7, z: 0 };
    match p {
        Point { x: 0, y, z } => println!("x: 0, y: {y}, z: {z}"),
        Point { x: 1, y: a, z: b } => println!("x: {x}, y: {a}, z: {b}"),
        Point { x, .. } => println!("x: {x}, rest doesn't matter"),
    }
}
```

Code 15: Destructuring patterns [17, Chapter 8.5.1]

```cpp
#include <iostream>
#include <tuple>

int main()
{
    std::tuple<int, std::string> tup{42, "hello"};

    auto [i, str] = tup;

    std::cout << "i: " << i << std::endl;
    std::cout << "str: " << str << std::endl;

    return 0;
}
```

Code 16: Simple example for structured bindings in C++

```rust
mod structs {
    pub struct Person {
        name: String,
        age: u8,
    }

    impl Person {
        pub fn new(name: String, age: u8) -> Self {
            Person { name, age }
        }

        pub fn name(&self) -> &String {
            &self.name
        }

        pub fn age(&self) -> u8 {
            self.age
        }
    }
}

fn print_person(person: structs::Person) {
    println!("{} is {} years old.", person.name(), person.age(),);
}

fn main() {
    let person = structs::Person::new(String::from("Alice"), 20);

    match person {
        p @ structs::Person { .. } => {
            print_person(p);
        }
    }
}
```

Code 17: Binding with `@` [11, pp. 248–249]

List of Figures

References

[1] Stack Overflow. *Stack Overflow Developer Survey 2022*. Archive: https://web.archive.org/web/20230429083342if_/https://survey.stackoverflow.co/2022/. 2022. URL: https://survey.stackoverflow.co/2022/ (visited on 2023-04-29).

[2] Facebook. *A brief history of Rust at Facebook*. Archive: https://web.archive.org/web/20230514122621if_/https://engineering.fb.com/2021/04/29/developer-tools/rust/. 2021. URL: https://engineering.fb.com/2021/04/29/developer-tools/rust/ (visited on 2023-04-29).

[3] Thomas Claburn. *Google polishes Chromium code with a layer of Rust*. Archive: https://web.archive.org/web/20230423101745if_/https://www.theregister.com/2023/01/12/google_chromium_rust/. The Register. 2023. URL: https://www.theregister.com/2023/01/12/google_chromium_rust/ (visited on 2023-04-29).

[4] Catalin Cimpanu. *Rust Programming Language Takes More Central Role in Firefox Development*. Archive: https://web.archive.org/web/20230514122751if_/https://www.bleepingcomputer.com/news/software/rust-programming-language-takes-more-central-role-in-firefox-development/. Bleeping Computer. 2017. URL: https://www.bleepingcomputer.com/news/software/rust-programming-language-takes-more-central-role-in-firefox-development/ (visited on 2023-04-29).

[5] Ian Evenden. *Rust Programming Language To Land in Linux Kernel 6.1*. Archive: https://archive.ph/ZVmqe. Tom's Hardware. 2022. URL: https://www.tomshardware.com/news/rust-in-linux-kernel/ (visited on 2023-04-29).

[6] Thomas Claburn. *Microsoft is busy rewriting core Windows code in memory-safe Rust*. Archive: https://web.archive.org/web/20230514123103if_/https://www.theregister.com/2023/04/27/microsoft_windows_rust/. The Register. 2023. URL: https://www.theregister.com/2023/04/27/microsoft_windows_rust/ (visited on 2023-04-29).

[7] The Rust Project Developers. *The Rust Reference*. Archive: https://web.archive.org/web/20230522145504if_/https://doc.rust-lang.org/stable/reference/. 2023. URL: https://doc.rust-lang.org/stable/reference/ (visited on 2023-04-29).

[8] The Rust Project Developers. *Rust API Guidelines - Naming*. Archive: https://web.archive.org/web/20230522143309if_/https://rust-lang.github.io/api-guidelines/. 2023. URL: https://rust-lang.github.io/api-guidelines/naming.html (visited on 2023-05-22).

[9] The Rust Project Developers. *Rust RFCs*. Archive: https://web.archive.org/web/20230522143230if_/https://rust-lang.github.io/rfcs/. 2023. URL: https://rust-lang.github.io/rfcs/ (visited on 2023-05-19).

[10] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. Archive: https://web.archive.org/web/20230430092242if_/https://doc.rust-lang.org/stable/book/. 2023. URL: https://doc.rust-lang.org/stable/book/ (visited on 2023-04-29).

[11] Jim Blandy, Jason Orendorff, and Leonora Tindall. *Programming Rust: Fast, Safe Systems Development*. English. Paperback. O'Reilly Media, July 20, 2021. ISBN: 978-1492052593.

[12] Bjarne Stroustrup. *A Tour of C++ (3rd Edition)*. English. Paperback. Addison-Wesley Professional, Sept. 24, 2022, p. 320. ISBN: 978-0136816485.

[13] The Rust Project Developers. *Result in std::result*. Archive: https://web.archive.org/web/20230522190753if_/https://doc.rust-lang.org/std/result/enum.Result.html. 2023. URL: https://doc.rust-lang.org/std/result/enum.Result.html (visited on 2023-05-22).

[14] The Rust Project Developers. *Option in std::option*. Archive: https://web.archive.org/web/20230522190716if_/https://doc.rust-lang.org/std/option/enum.Option.html. 2023. URL: https://doc.rust-lang.org/std/option/enum.Option.html (visited on 2023-05-22).

[15] *std::optional*. Archive: https://web.archive.org/web/20230701114736if_/https://en.cppreference.com/w/cpp/utility/optional. 2022. URL: https://en.cppreference.com/w/cpp/utility/optional (visited on 2023-07-01).

[16] Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo. *C++ Primer (5th Edition)*. English. Paperback. Addison-Wesley Professional, Aug. 6, 2012, p. 972. ISBN: 978-0321714114.

[17] The Rust Project Developers. *Rust by Example*. Archive: https://web.archive.org/web/20230522143831if_/https://doc.rust-lang.org/stable/rust-by-example/. 2023. URL: https://doc.rust-lang.org/stable/rust-by-example/ (visited on 2023-04-29).

[18] The Rust Project Developers. *The Rust Reference*. Archive: https://web.archive.org/web/20230522145556if_/https://github.com/rust-lang/reference/blob/bbcacbb633a9963434ad8aaa5407e3c6ad95e4b9/src/expressions/match-expr.md. 2018. URL: https://github.com/rust-lang/reference/blob/bbcacbb633a9963434ad8aaa5407e3c6ad95e4b9/src/expressions/match-expr.md (visited on 2023-05-19).

[19] Bjarne Stroustrup. *Programming: Principles and Practice Using C++ (2nd Edition)*. English. Paperback. Addison-Wesley Professional, May 15, 2014, p. 1312. ISBN: 978-0321992789.

[20] Bruno Cardoso Lopes et al. *Pattern Matching*. Archive: https://web.archive.org/web/20230701164249if_/https://open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1371r3.pdf. 2020. URL: https://open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1371r3.pdf (visited on 2023-07-01).

[21] Herb Sutter. *Pattern matching using is and as*. Archive: https://web.archive.org/web/20230701164408if_/https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2392r1.pdf. 2021. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2392r1.pdf (visited on 2023-07-01).

[22] Yuriy Solodkyy. *Mach7: Pattern Matching for C++*. Archive: https://web.archive.org/web/20230701164711if_/https://github.com/solodon4/Mach7. 2022. URL: https://github.com/solodon4/Mach7 (visited on 2023-07-01).

[23] The Rust Project Developers. *Rust's Unsafe Code Guidelines Reference*. Archive: https://web.archive.org/web/20230522203232if_/https://rust-lang.github.io/unsafe-code-guidelines/. 2023. URL: https://rust-lang.github.io/unsafe-code-guidelines/ (visited on 2023-05-22).