UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

COS700 Research

Design Pattern Metaprogramming Foundations in Rust

Abstract Factory and Visitor

Student number: u19239395

Supervisor:
Dr. Linda Marshall

??? 2020

Abstract

Keywords:

1 Introduction

2 Rust

Rust is a relatively new language created by Mozilla to be memory safe yet have low level like performance [KN19]. Traditionally, memory safe languages will make use of a garbage collector which slows performance [HB05]. Garbage collector languages include C# [RNW+04], Java [GJS96], Python [Mar06], Golang [Tso18] and Javascript [Fla06]. Languages that perform well use manual memory management, which is not memory safe whenever the programmer is not careful. Dangling pointers [CGMN12], memory leaks [Wil92], and double freeing [Sha13] in languages like C and C++ are prime examples of manual memory management problems [Kok18]. Few languages have both memory safety and performance. However, Rust achieves both by using a less popular model known as ownership [MK14].

2.1 Ownership

In the ownership model, the compiler uses statical analysis [RL19] to track which variable owns a piece of heap data – this does not apply to stack data. Each data piece can only be owned by one variable at a time called the *owner* [KN19].

A variable also has scope. The scope starts at the variable declaration and ends at the closing curly bracket of the code block containing the variable. When the owner goes out of scope, Rust returns the memory by calling the *drop* method at the end of the scope. Ownership is manifested in two forms – moving and borrowing. These two forms are explained next [KN19].

2.1.1 Moving

Moving happens when one variable is assigned to another. The compiler's analysis moves ownership of the data to the new variable from the initial variable. The initial variable's access is then invalidated [KN19]. An analogy example would be to give a book to a friend. The friend can do anything from annotating to burning the book as they feel fit since the friend is the book's owner.

Listing 1: Example of ownership transfer

```
1 {
2     let s = String::from("string");
3     let t = s;
4 
5     println!("String len: {}", s.len()); - borrow of moved value: `s`
6 } // Compiler will 'drop' t here
```

In Listing 1, on line 2, a heap data object is created and assigned to variable *s*. Line 3 assigns *s* to *t*. However, because *s* is a heap object, the compiler transfers ownership of the data from *s* to *t* and marks *s* as invalid.

When trying to use the data on line 5, via *s*, the compiler throws an error saying *s* was moved. Any reference to *s* after line 3 will always give a compiler error.

Finally, the scope of t ends on line 6. Since the compiler can guarantee t is the only variable owning the data, the compiler can free the memory on line 6.

Listing 2: Function taking ownership

```
1  fn main() {
2      let s = String::from("string");
3      take_ownership(s);
4
5      println!("String len is {}", s.len()); - borrow of moved value: `s`
6  }
7
8  fn take_ownership(a: String) {
9      // some code working on a
10 } // Compiler will 'drop' a here
```

Having ownership moving makes excellent memory guarantees within a function; however, it is annoying when calling another function, as seen in Listing 2. The *take_ownership* function takes ownership of the heap data resulting in memory cleanup code correctly being inserted at the end of *take_ownership*'s scope on line 10. When *main* calls *take_ownership*, *a* becomes the new owner of *s*'s data, making the call on line 5 invalid. When taking ownership is not desired, the second form of ownership, borrowing, should be used instead.

2.1.2 Borrowing

Borrowing has a new variable take a reference to data rather than becoming its new owner [KN19]. An analogy is borrowing a book from a friend with a promise of returning the book to its owner once done with it.

Listing 3: Function taking borrow

```
1  fn main() {
2      let s = String::from("string");
3      take_borrow(&s);
4
5      println!("String len is {}", s.len());
6  } // Compiler will 'drop' s here
7
8  fn take_borrow(a: &String) {
9      a.push_str("suffix"); - cannot borrow *a as mutable
10 } // a is borrowed and will therefore not be dropped
```

As seen in Listing 3, borrowing makes the function *take_borrow* take a reference to the data. References are activated with an ampersand (*&*) before the type. Once *take_borrow* has ended, control goes back to *main* - where the cleanup code will be inserted. Having references as function parameters is called borrowing [KN19]. The ampersand is also used in the call argument on line 3 to signal the called function will borrow the data.

However, in Rust, all variables are immutable by default [KN19]. Hence changing the data in *take_borrow* causes an error stating the borrow is not mutable on line 9. Returning to the borrowed book analogy. One would not make highlights and notes in a book one borrowed unless the owner gave explicit permission.

2.2 Immutable by default

Mutable borrows are an explicit indication that a function/variable is allowed to change the data.

Listing 4: Function taking mutable borrow

```rust
1  fn main() {
2      let mut s = String::from("string");
3      take_borrow(&mut s);
4  
5      println!("String len is {}", s.len());
6  } // Compiler will add memory clean up code for s here
7  
8  fn take_borrow(a: &mut String) {
9      a.push_str("suffix");
10 }
```

As seen in Listing 4, line 8, mutable borrows are activated using *&mut* on the type. Again, *mut* is also used in the call on line 3 to make it explicit that the function will modify the data. Variables – on the stack or heap – also need to be declared *mut* to use them as mutable [KN19] as seen on line 2.

The two ownership forms - moving and borrowing - together with mutable variables put some constraints on the code for variables and their calls: [KN19]

- Moving will always invalidate the variable.
- Borrowed variables cannot be mutated. However, more than one variable can borrow the data simultaneously in parallel and concurrent code.
- Mutable borrowing does allow mutations. But only one variable can hold a mutable borrow, while no other immutable borrows can exist at the same time.

The constraints will always be enforced by the compiler, thus requiring all code - written by hand or a macro - to meet them. Meeting these constraints also requires some shift in thinking. Another shift is required because Rust may not classify as an Object-Oriented Programming (OOP) language, which will now be discussed.

2.3 Not quite OOP

No single definition exists to qualify a language as Object-Oriented [Mey97, SB85, GHJV94, KN19]. Three Object-Oriented concepts will be explored to understand Rust better, these three are *Objects as Data and their Behaviour*, *Encapsulation*, and *Inheritance*.

2.3.1 Objects as Data and their Behaviour

An object holds both data and procedures operating on the data [Mey97, SB85, GHJV94, Mal09]. Rust holds data in *struct*s and the operations are defined in *impl* blocks [KN19].

Struct: A *struct* is the same as a *struct* in C [Str13] and other C-like languages [RNW$^+$04, WS15, Mal09]. Structs are used to define objects with named data pieces, as shown in Listing 5 lines 1 to 5. Each of the struct properties is named followed by a type.

Methods The operations to perform on a struct are defined in *impl* blocks, as seen in lines 7 - 19 in Listing 5.

The ownership rules apply to the struct as follow:

- The method *have_burial* moves *self* and will result in cleanup code being inserted on line 18. Thus, any objects of *Foo* after *have_burial* is called will be invalidated. This happens on line 33, resulting in a compile error - the same error happened in Listing 2.
- The method *get_age* will take a borrow of the struct object.
- To age, while having a birthday, *have_birthday* needs to take a mutable borrow of *self* - also why *bar* needs to be *mut* in *main* on line 22.

Listing 5: Example of a Struct and Methods

```rust
1  pub struct Foo {
2      pub name: String,
3      age: u8,
4      gender: Gender,
5  }
6
7  impl Foo {
8      pub fn get_age(&self) -> u8 {
9          self.age // age is returned [1]
10     }
11
12     pub fn have_birthday(&mut self) {
13         self.age += 1;
14     }
15
16     fn have_burial(self) {
```

[1] Rust does not always use the *return* keyword. Lines without a semi-colon that are the last lines in a scope act as return statements. Thus, line 9 acts as a return statement.

```
17             unimplemented!();
18         } // `self` will be dropped here
19     }
20 
21     fn main() {
22         let mut bar = Foo {
23             name: String::from("bar"),
24             age: 1,
25             gender: Gender::Male,
26         };
27 
28         bar.have_birthday();
29 
30         // Party was crazy
31         bar.have_burial();
32 
33         println!("Bar was {} years old", bar.get_age()); - borrow of moved
       ↪  value: 'bar'
34     }
35 
36     // Second implementation block [2]
37     impl Foo {
38         // Construct a new `Foo` with a given name [3]
39         pub fn be_born(name: String) -> Self {
40             Foo {
41                 name,
42                 age: 0,
43                 gender: Gender:Female,
44             }
45         }
46     }
```

2.3.2 Encapsulation

Encapsulation hides the implementation details from the client [KN19, Mey97, WS15]. In Rust, encapsulation is the default unless specified otherwise using the *pub* keyword. Encapsulation allows the struct creator to change the internal procedures of the struct without affecting the public interface used by clients. Therefore, Rust meets the encapsulation concept.

[2]Rust code can have multiple *impl* blocks for a single type.

[3] Rust does not provide constructors like other OOP languages. Instead, Rust has what it calls *associate functions*. An associate function is a method definition not containing *self* in the parameter list [KN19]. They are used as constructors by returning an owned instance of the struct as seen on line 39.

pub As can be seen in Listing 5, the *Foo* struct and its *name* data member are explicitly made public, as are the methods *get_age* and *have_birthday*. All the other data members and methods are private - Rust's default - and unreachable by client code. *Main* is allowed to access the private members and methods - even though they are hidden - because they are all in the same file.

2.3.3 Inheritance

Inheritance allows an object to inherit some of its data members and procedures from a parent object [Mey97, SB85, GHJV94, WS15]. Inheritance is mostly used to reduce code duplication. Rust does not make provision for the inheritance concept.

Rust does make provision for "Program to an interface, not an implementation" and "Favor object composition over class inheritance" which the GoF include as concepts for OOP design [GHJV94]. Both concepts are realized using *traits* thereby allowing Rust to implement the GoF design patterns.

Traits are similar [KN19] to *interfaces* in other languages like C# [RNW$^+$04] and Java [GJS96]. In C++, *abstract classes* are the equivalent of *interfaces* [Mal09, Str13, Ale01]. Thus, *traits* allow the definition of abstract behavior to program to an interface as seen in Listing 6, lines 1 to 7.

Listing 6: Working with traits

```
 1  trait Show {
 2      fn show(&self) -> String;
 3
 4      // Method with a default implemetation
 5      fn show_size(&self) -> usize {
 6          self.show().chars().count()
 7      }
 8  }
 9
10  fn work<T: Show>(object: T) {
11      println!("{}", object.show());
12  }
13
14  fn complex<T: Show + Display + PartialEq<U>, U: Display>(first: T,
    ↪  second: U) {
15      if first == second {
16          println!("{}({}) is equals {}", first, first.show(),
    ↪  second);
17      }
18  }
19
20  fn complex_where<T, U>(first: T, second: U)
21  where
22      T: Show + Display + PartialEq<U>,
```

```rust
23      U: Display,
24  {
25      // Same as complex with easier to read trait bounds
26  }
27
28  struct Tester {}
29
30  impl Show for Tester {
31      fn show(&self) -> String {
32          String::from("Tester")
33      }
34  }
```

The compiler uses *traits* at compile time to guarantee an object implements a set of methods using *trait bounds*. Line 10 shows the *work* function having a *trait bound* on the generic *T* type. Thus, any object choosing to implement the *Show* trait can be passed to *work*. In turn, *work* knows it is safe to call *show()* on the object for type *T*.

More complex trait bounds can be constructed as shown on line 14 – lines 20 to 23 are equivalent to line 14 but easier to understand. Here *T* (*first*) has to implement the traits:

- *Show* to be able to call the *show()* method on it.
- *Display* to pass it to *println!*.
- *PartialEq* to compare it with *U* (*second*).

Traits are implemented on structs using the *impl* keyword followed by the trait name as seen on line 30. The *Show* trait has a default implementation for *show_size* on line 5. Thus, *Tester* does not need to implement *show_size*, it only needs to implement the *show* method.

Traits allow one to use composition to construct complex objects. Getting back to the abstract confirmation dialog presented in Section **??** (Design Patterns) for an open dialog and a safe dialog, using composition, the generic confirmation dialog will have to be composed of a button and text as seen in Listing 7.

Listing 7: Using composition

```rust
pub trait Button {
    fn draw(&self); // Button to draw itself
}
pub trait DisplayText {}

struct ConfirmationDialog {
    button: Box<dyn Button>,
    display_text: Box<dyn DisplayText>,
}
```

```
10
11  impl ConfirmationDialog {
12      // Constructor to build a new `ConfirmationDialog`
13      pub fn compose(button: Box<dyn Button>, display_text: Box<dyn
     ↪  DisplayText>) -> Self {
14          ConfirmationDialog {
15              button,
16              display_text,
17          }
18      }
19
20      pub fn show(&self) {
21          self.button.draw(); // Draw the button for the dialog
22      }
23  }
```

Using traits, an abstract button and display text is defined on lines 1 to 4. The confirmation dialog "has-a" [Mal09] button and display text on lines 7 to 8. Using associate functions[3], a composition constructor is created on line 13. To construct the open dialog, the open button will be passed to this constructor. The safe dialog will need the safe button to be passed in. Finally, the *show* method on line 18 will show the confirmation dialog - whether it is the open dialog, save dialog, or a new dialog definition. This is called polymorphism since a dialog will change form depending on the elements passed to the constructor at run-time [WS15, Mal09, GHJV94].

The *show* method will now work with more than one button. The open button will have a different implementation for the *draw* method, defined on line 2, than the save button. Since the buttons change at runtime, the compiler will be unable to determine the correct *draw* method to call in *show* on line 21. Dispatching, as discussed next, resolves this problem [PB02].

2.4 Dispatch

Dispatching is the matching of the correct method to an object at method invocations [DHV95]. Sometimes there is only one object to match against, making matching easy. But it can also be the case that multiple objects have the same method name, making matching harder. The matching can happen at compile-time - called *static dispatching* - or run-time - called *dynamic dispatching*.

2.4.1 Static Dispatch

Matching the method call at compile-time is called *static dispatch* [KN19, Ale01, AC12]. When Rust compiles generics on a function or type, as seen in Listing 6 line 10, Rust uses what it calls *monomorphization* [KN19]. *Monomorphization* creates a new function or type at compile time for each concrete object passed into the generic placeholders.

In Listing 6 line 10, we defined the generic *work* method. Suppose the *work* method is called with a *Tester*, defined on line 28, and later with a *String*, then an example of the *monomorphization* process is shown in Listing 8.

Listing 8: Example of *monomorphization*

```rust
fn work_tester(object: Tester) {
    println!("{}", object.show());
}
fn work_string(object: String) {
    println!("{}", object.show());
}
```

Each *show* call in the expanded functions can match only one object - *Tester* and *String* respectively - making this a simple case. Using *trait objects* - rather than generics - will result in multiple objects having the same method name and the need for dynamic dispatch.

2.4.2 Dynamic Dispatch

When the compiler cannot determine at compile-time which method to call because the object type is not fixed, dynamic dispatch [Ale01, KN19, AC12] takes place. At run-time, pointers held by the trait objects are used to determine which method to call [KN19] based on the object type the method is called on.

Listing 9 shows the *work* function implemented using dynamic dispatch rather than generics – refer to *work* in Listing 6 line 10 for the generic version. Since generics are not used, *monomorphization* will not happen. Calling this function with a *Tester* object means the *show* for *Tester* needs to match at the method invocation on line 2. But later, calling this function with a *String* object means *String*'s *show* needs to match. Since the objects passed in changes at run-time, knowing which object to match against can therefore only be known at run-time. Thus, pointers inside the *Show* trait object - line 1 of Listing 6 - are used at run-time to match each object against its method.

Listing 9: Dynamic Dispatch with *dyn*

```rust
1  fn work(object: &dyn Show) {
2      println!("{}", object.show());
3  }
```

Three changes need to be made to the function signature to use trait objects rather than generics.

- The generic *T* is removed and replaced with *Show*.
- The *dyn* keyword is added to the type to indicate that dynamic dispatch will take place explicitly [KN19].
- Borrowing (& before *dyn*) now takes place.

The static dispatch generic trait examples in Listing 6 can also be made to take a borrow, but taking ownership gives a compile-time error with dynamic dispatch which is caused by the *Sized* trait.

2.4.3 Sized trait

Rust keeps all local variables and function arguments on the stack. Having values on the stack requires their size to be known at compile-time. A special trait called *Sized* is used by the compiler to mark that the size of a type is known at compile-time [KN19]. This marking is the only use of the *Sized* trait and it has no meaning or representation after compilation. However, Rust automatically/implicitly adds the *Sized* trait bound to all function arguments and local variables.

The size of an object is influenced by the data it holds. Also, any object can choose to implement *Show*. Thus, two different objects implementing *Show* can have different sizes. Ownership will want to pass each object on the stack, but, with dynamic dispatch, each object will need a different stack size only known at run-time. Therefore, the *Show* dynamic trait's size cannot be determined at compile-time - i.e. the *Show* dynamic trait will not implement the *Sized* trait.

Since all function arguments expect the type to implement the *Sized* trait but having the *Show* dynamic trait not implement it, a compile error will be given stating *dyn Show* does not implement *Sized* when trying to use it as a function argument. However, pointers do implement *Sized*. Therefore, putting the dynamic trait behind any pointer will allow it to be used as a function argument or local variable.

Rust offers many pointer options: [KN19]

- A reference - also called a borrow.
- The *Box<T>* used to hold heap objects. *Box* differs from a reference since *Box* owns its *T* [4]. Thus, when the box is dropped, the object it holds is dropped too.
- The *Rc<T>* reference counting pointer that is essentially a run-time immutable borrow. When the reference count reaches zero, the *T* is dropped.
- The special *RefCell<T>* which exposes mutable access behind an immutable object using the *Interior Mutability Pattern*.
- The combined *Rc<RefCell<T>>* - the reference counting pointer allows multiple objects to exist, while the *RefCell* allows mutable access to each existing object.
- *Arc<T>* - the thread-safe version of *Rc*.
- *Mutex<T>* - a thread-safe version of *RefCell*. However, a lock needs to be acquired first.
- *RwLock<T>* - like *Mutex*, but distinguishes between a reader and a writer. It is the run-time equivalent of mutable and immutable borrows.
- Finally, *Arc<RwLock<T>>* which will be used by our Abstract Factory. It will hold *T*s for use in multiple threads using *Arc*, while *RwLock* will guarantee only one writer.

One more Rust uniqueness is left to be covered. Rust treats enums differently than other languages.

[4]https://joshleeb.com/blog/rust-traits-trait-objects/

2.5 Enums

In Rust, enums can also hold objects [KN19] as seen in Listing 10, lines 1 to 3. The *Option* enum, as defined here, is built into the standard Rust library [KN19] to replace *null* as used in other languages. An *Option* can either be *Some* object or *None*. This is yet another design Rust uses to be memory safe [5] by checking that the "*null*" (*None*) option is handled at compile-time.

Listing 10: Enums holding objects in Rust

```rust
 1  enum Option<T> {
 2      Some(T),
 3      None,
 4  }
 5
 6  fn main() {
 7      let age = Option::Some(5);
 8
 9      match age {
10          Option::Some(value) => println!("Age = {}", value),
11          Option::None => println!("Age is unknown"),
12      }
13
14      let valid = if let Option::Some(_) = age {
15          println!("Age is set");
16          true
17      } else {
18          false
19      };
20  }
```

Lines 9 to 12 show the use of *match* - called a *switch* in most languages - to match against each possible enum variant. Line 10 and 11 are each called a *match arm*. Line 10 shows how an object can be extracted on an arm and be assigned to a *value* variable. If any of the two arms are missing, the compiler will give an error stating not all the enum options are covered, because matches need to be exhaustive - i.e. all variants need to be covered - in Rust. There are two options for getting around the exhaustive check [KN19].

Either adding the _ (underscore) catch-all arm to handle the default case for all missing enum options. Alternatively, using the *if let* pattern as seen on line 14. The *if let* pattern also allows extraction of the enum object. However, since the object is not used inside the if block, no extraction needs to occur. The _ is, therefore, used to not extract the enum object.

Both *if* and *match* blocks are considered expressions in Rust. Thus, lines

[5]Tony Hoare, the inventor of the *null* reference, has called the *null* reference a billion-dollar mistake in his 2009 presentation "Null References: The Billion Dollar Mistake" (https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/)

16 and 18 are missing their ending semi-colon to return *true* and *false* from the *if* expression. The value returned from the *if* is assigned to *valid*. Returning from a *match* is quite common, especially with a particular enum used for error handling as discussed next.

2.5.1 Result enum for error handling

While other languages use exceptions to propagate errors up to the caller, Rust uses the *Result* enum instead. The definition for *Result* can be seen in Listing 11 on lines 1 to 4.

Listing 11: The *Result* enum

```
 1  enum Result<T, E> {
 2      Ok(T),
 3      Err(E),
 4  }
 5
 6  fn may_error() -> Result<i32, String> {
 7      Result::Err(String::from("Network down!"))
 8  }
 9
10  fn error_explicit_handle() {
11      let r = may_error();
12
13      let r = match r {
14          Result::Ok(result) => result,
15          Result::Err(error) => panic!("Operation failed: {}", error),
16      };
17
18      // Use r
19  }
20
21  fn error_short_handle() {
22      let r = may_error().expect("Operation failed");
23
24      // Use r
25  }
26
27  fn error_explicit_propogation() -> Result<i32, String> {
28      let r = match may_error() {
29          Result::Ok(result) => result,
30          Result::Err(error) => return Err(error),
31      };
32
33      Ok(2)
34  }
35
36  fn error_short_propogation() -> Result<i32, String> {
37      let r = may_error()?;
38
39      Ok(2)
40  }
```

A function will return *Result* to indicate if it was successful with the *Ok* variant holding the successful value of type *T*. In the event of an error, the *Err* variant is returned with the error of type *E* - like *may_error* on line 7. Any calls to *may_error* have to handle the possible error. A few error handling options exist: trying an alternative, panicking, or propagating the error.

Alternative Trying an alternative is quite simple. If the *Err* enum is returned, then just run the alternative instead.

Panicking The caller will use a *match pattern* to extract the error and panic as seen on lines 13 to 16. However, writing matches all the time for possible errors breaks the flow of the code. So the *Result* enum has some helper methods defined on it [6] - Rust enums are like ordinary objects that can have methods.

The helper method *expect*, as seen on line 22, does the exact same as the match on lines 13 to 16.

Propagation The caller might decide more information is need to panic. So the caller's caller will need to handle the error instead.

Line 30 shows how to propagate the error up the stack - the *return* is to return from the function and not the match. Line 29 uses the result if it is fine - the lack of *return* returns from the match and assigns *result* to *r*. Again, the match is verbose and Rust offers a helper to make it shorter. The helper is the *?* (question mark) operator. The *?* operator can be used in any function returning a *Result* or *Option* - or type implementing the *std::ops::Try* trait [KN19]. Line 37 shows how to use *?* - doing precisely the same as lines 28 to 31.

3 Reporting

4 Conclusion

References

[AC12] Martin Abadi and Luca Cardelli. *A theory of objects*, chapter 2, page 18. Springer Science & Business Media, 2012.

[6]https://doc.rust-lang.org/std/result/enum.Result.html

[Ale01] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied.* Addison-Wesley, 2001.

[CGMN12] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, page 133âĂŞ143, New York, NY, USA, 2012. Association for Computing Machinery.

[DHV95] Karel Driesen, Urs Hölzle, and Jan Vitek. Message dispatch on pipelined processors. In *European Conference on Object-Oriented Programming*, pages 253–282. Springer, 1995.

[Fla06] David Flanagan. *JavaScript: The Definitive Guide*, chapter 11, pages 171–172. O'Reilly Media, Inc., 2006.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1994.

[GJS96] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*, chapter 9, 12, pages 199–208, 245. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1996.

[HB05] Matthew Hertz and Emery D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. *SIGPLAN Not.*, 40(10):313âĂŞ326, October 2005.

[KN19] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018).* No Starch Press, 2019.

[Kok18] Konrad Kokosa. *Pro .NET Memory Management: For Better Code, Performance, and Scalability*, chapter 1, pages 28–34. Apress, USA, 1st edition, 2018.

[Mal09] Davender S Malik. *Data structures using C++*, chapter 1, 2, 3, pages 4, 33, 79, 170. Cengage Learning, 2009.

[Mar06] Alex Martelli. *Python in a Nutshell (In a Nutshell (O'Reilly))*, chapter 13, pages 269–272. O'Reilly Media, Inc., 2006.

[Mey97] Bertrand Meyer. *Object-oriented software construction*, volume 2. Prentice hall Englewood Cliffs, 1997.

[MK14] Nicholas D Matsakis and Felix S Klock. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014.

[PB02] Benjamin C Pierce and C Benjamin. *Types and programming languages*, chapter 18, page 226. MIT press, 2002.

[RL19] Morten Meyer Rasmussen and Nikolaj Lepka. Investigating the benefits of ownership in static information flow security. 2019.

[RNW$^+$04] Simon Robinson, Christian Nagel, Karli Watson, Jay Glynn, and Morgan Skinner. *Professional C#*, chapter 3, 4, 7, pages 101–104, 123–130, 192–193. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2004.

[SB85] Mark Stefik and Daniel G. Bobrow. Object-oriented programming: Themes and variations. *AI Magazine*, 6(4):40, Dec. 1985.

[Sha13] John Sharp. *Microsoft Visual C# 2013 Step by Step*, chapter 14, pages 320–321. Microsoft Press, USA, 1st edition, 2013.

[Str13] B. Stroustrup. *The C++ Programming Language*, chapter 8, 20, pages 205, 598. Addison-Wesley, 2013.

[Tso18] Mihalis Tsoukalos. *Mastering Go: Create Golang production applications using network libraries, concurrency, and advanced Go data structures*, chapter 2, pages 47–49. Packt Publishing Ltd, 2018.

[Wil92] Paul R Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, pages 1–42. Springer, 1992.

[WS15] Kenrich Mock Walter Savitch. *Problem Solving in C++*, chapter 1, 10, 15, pages 47, 572–583, 892–893. Pearson Education Ltd, Edinburgh Gate, Harlow, England, 2015.