Technische Hochschule Ingolstadt

Specialist area (faculty) Informatik

Bachelor's course Informatik

Bachelor's thesis

Subject:

Comparison of Rust to Ada and C in regards to safety-related software

Name and Surname: Philipp von Perponcher

Issued on: 24.12.2021
Submitted on: 11.02.2022

First examiner: Prof. Dr. rer. nat. Franz Regensburger
Second examiner: Prof. Dr. Stefan Hahndel

Declaration

I hereby declare that this thesis is my own work, that I have not presented it elsewhere for examination purposes and that I have not used any sources or aids other than those stated. I have marked verbatim and indirect quotations as such.

Ingolstadt, February 11, 2022

Philipp von Perponcher

This thesis was written as the conclusion of a 3.5 year dual program with Technische Hochschule Ingolstadt and MBDA Deutschland. During my time so far, I met and worked with a ton of people that helped me find a first and now firm foothold in software development.

I would like to thank all of them, first and foremost my supervisors Rico Lieback and Thomas Britzelmeier for accompanying me throughout my time at MBDA, providing me with exciting topics that complemented my parallel studies.

I would also like to thank Mr. Prof. Dr. rer. nat. Franz Regensburger for accompanying this thesis and always being available and helpful whenever questions arose.

A special mention also goes out to my family who supported and helped me throughout my entire studies.

Finally, I would like like to thank my good friends Dominik Bartl, Nico Borgsmüller, Dario Köllner, Tycho Mertens, Jan Mottl, Grady Orr, Sebastian Rabau, Javen Thompson, and Andrea Wiethüchter for proof-reading the thesis thoroughly, allowing me to improve it even further.

Abstract

The goal of this thesis is to give a basic understanding whether Rust can be suited for safety-critical systems programming projects where Ada is currently the mainly present language.

To get an overview over the features that make these languages suitable for safe programming, several common programming errors and the languages' reaction to them will be looked at. For this comparison part, C is added to get an understanding of how a programming language that is not targeted at safety handles these errors.

The outcome from these tests were that Rust covers all safety features that Ada has, even offering additional ways of dealing with certain errors. It also enforces stricter rules in some places, for example when dealing with concurrent programs and shared resources.

Apart from the technological background, the programming environment is also important for a language to be suitable for bigger projects. From an official side, Ada has a big advantage due to its long history and firm foothold in the aerospace industry while Rust is open-source and does currently not have any kind of certification for its safety features. Another part of the environment is the developers themselves, where Rust has a clear advantage due to its fairly young age and a strong online community and popularity.

The question whether Rust can be suited for safety-critical projects or even be a valid alternative to Ada in that regard can't really be answered with a clear Yes or No. In case any official guidelines that require some sort of certification have to be adhered to, Rust may be a little difficult to use as it does not have any, even though efforts in that direction are well on the way. If no guidelines have to be followed, Rust can definitely be a realistic alternative to Ada due to its popularity while still offering very safe programming.

Contents

1 Introduction

1.1 Introduction to Safe Software

Anyone who has ever written software, whether at home or in a corporate environment, has encountered errors in their code sooner or later. The most visible error is a program crash that can occur at some point during execution.

Here, the term "crash" is used when a process terminates unexpectedly. In some application areas, such as entertainment, a crash is merely an inconvenience; the software can simply be restarted. In other scenarios, such as control software for aircraft or military systems, a crash can have catastrophic consequences, such as the 1996 crash of the first version of the Ariane 5, estimated to have cost up to $500 million. [JM97]

Just as serious, though not as abrupt as a crash, can be unexpected or unwanted behavior in general. This can include small errors or inaccuracies that are accepted to some extent, but also malfunctions that occur during the execution of a program but are theoretically valid and therefore do not lead to a complete abort. If they go undetected for too long, small errors that may have been dismissed as irrelevant can add up and have even worse consequences than a crash, since it is not apparent to outsiders that anything is wrong. A good example is the failure of a U.S. American MIM-104 Patriot surface-to-air missile battery in Saudi Arabia during the 1992 Gulf War. Due to a mathematical rounding inaccuracy that added up over the hours the unit was online, an Iraqi Scud missile managed to penetrate the defenses, hit a barracks, and kill 28 soldiers. [DC92]
Had the system crashed earlier, it would have been obvious to an observer that something was wrong and action needed to be taken. By appearing to work as intended, the system gave its operators a false sense of security, which subsequently led to the 28 casualties mentioned above.

To ensure that as many errors as possible are removed from such systems, extensive development and testing work is required, especially for such critical systems. The earlier a defect is detected in these stages, the lower the cost to the company, whether in terms of extended development times or actual damages later on.

To help software developers avoid bugs as early as possible, certain programming languages are designed to be particularly “safe”, leaving very little room for the software to make mistakes. One of these languages is Ada, which was developed for the United States Department of Defense from 1977 to 1983 [Bar14, p. 3] . Another language that is also very strict about security, but at the same time fast and modern, and certainly a newcomer compared to Ada, is Rust, whose compiler was first released in 2012 [And12] . In order to also have a comparison with a language that is not focused on security, but leaves a lot of freedom to the developer, C was chosen. It is also “the classic” when it comes to programming, since it is still widely used and at the same time the oldest of the three languages mentioned [Rit93] .

1.2 Motivation and Scope

For companies who have to adhere to safety standards and currently use Ada for that, it could be beneficial to keep their eyes out for new, more attractive programming languages like Rust for various reasons. As will be discussed further in chapter 4.2, the introduction to Ada, it is very difficult to find qualified Ada developers because it is not a very common programming language.

If, at some point, another more common and modern programming language that is geared toward safety-critical systems like Rust is certified as ”safe” in some way, it will be advantageous for an organization that relies on these features to have already tested the language for usability and applicability for their product area. An educated guess as to whether these programming languages might even be a realistic alternative to Ada could give these companies a better estimate for their possible usage in future projects.

This thesis was written in partnership with MBDA Deutschland whose main product area is missile systems and their surrounding technical components. This requires that a lot of software is written at the system level, as well as for maintenance and C2 (command and control) applications for customers. The former, namely systems programming, is discussed in more detail in this paper, which also led to the selection of the three programming languages. Examples for software in that area include navigation algorithms on board a missile or communication between different components of an air defense system such as radar and launcher.

1.3 Structure

To get a comparison of whether Rust could be an alternative to Ada from a basic technological background, some of the most common reasons for unexpected behavior of software

are presented while the reaction of C, Ada and Rust is covered and explained. This will then help to draw a conclusion on whether Rust is suitable for software in safety-critical environments where Ada is currently the established programming language. If that's not fully the case, the missing factors that are necessary for it to be a considerable option will be talked over. During the main part, C is used as a comparison to show how a very open language reacts to these possible errors, it will not be taken into consideration in the conclusions.

Each exception is presented considering the three main points:

- Why does this error occur and what consequences could result from running into it?
- How easy is it to encounter this error in C, Ada, and Rust and does the respective compiler prevent the programmer from doing so?
- Summary how the error is handled by the languages.

It is to be noted that modern Integrated Development Environments (IDEs) help a lot with programming and are able to identify some mistakes while the code is being written. As this paper focuses on the compilers and their reactions and not on the development environments, this factor is not taken into account.

The thesis concludes by summarizing the responses to the errors of each of the three programming languages, as well as the advantages and disadvantages found. This final chapter also identifies the pros and cons of why and when a company should or should not consider that language for a safety-critical project. Additionally, an assessment is then given in the final conclusion as to whether Rust can be a real alternative to Ada in an industrial environment.

2 Background Information and Environment

2.1 Introduction of MBDA Deutschland

MBDA Deutschland GmbH is a German company specializing in the defense industry, particularly missile systems. It is a subsidiary of the European MBDA Group and has two direct sub-companies: TDW, which develops and maintains effectors, and Bayern-Chemie, which focuses on propulsion systems. MBDA Deutschland has its main site in Schrobenhausen, Bavaria, where development and most of the administration are located. Two smaller sites are worth mentioning, in Aschau am Inn, where Bayern-Chemie has its main research department, and in Freinhausen, which serves as a test and evaluation site for cooperation projects with the German Air Force. [Pre]

As NATO countries are MBDA's largest and most important customers, it is essential for the company to comply with their official software requirements, such as the DO-178C standard used by the European Aviation Safety Agency (EASA) for certification purposes [Eas, Ch. AMC 20-115D] . Since this means that the software must be very safe and thoroughly tested, using programming languages that already guarantee a certain level of safety is the first logical step to avoid unnecessary code. Besides, it only makes sense for any company in this sector to follow strict standards, since different countries may have varying requirements. If they are as strict as possible from the beginning, it helps sell the product, as customization for a new customer would only require an integration to the new platform instead of large changes to the code base to meet any new requirements.

Of course, it is also in the interest of MBDA itself that the products work and do not crash due to a software error that could have been ruled out in advance, since any negative press is rather undesirable.

2.2 Software Versions

All code examples in this paper were compiled using the following compiler and toolchain versions:

Language	Command	Version	Language Standard
C	gcc	9.3.0	C17
Ada	gprbuild	20190517	Ada 2012
Rust	rustc (cargo)	1.57.0	Rust 2021

3 Error Classification

Since every programming language provides more or less or even no explanations for why software terminates or behaves unexpectedly, it is difficult to find a general classification for unwanted behavior. In order not to go beyond the planned scope of this thesis, a reasonable division and selection is necessary. In this chapter, a rough overview of software errors will be given, along with a basic classification and selection of errors.

3.1 General Software Error Classes

One of the two main sources of software flaws which are classified and used for this thesis is The 2021 Common Weakness Enumeration (CWE) Top 25 Most Dangerous Software Weaknesses, maintained by a subsidiary of The MITRE Company and sponsored by the U.S. Department of Homeland Security, among others. In order to grade and order them, the CWE relies on data from the National Vulnerability Database (NVD) and scores them using the Common Vulnerability Scoring System (CVSS), which is based on characteristics and severity of software vulnerabilities. [Cwe]

The following list contains the 25 vulnerabilities with the greatest impact in 2021, as measured by both the NVD and CVSS. [Cwe] Since these weaknesses come from all sorts of areas of software development, many of them unfortunately do not apply to systems programming.

Hereby, some of these vulnerabilities can be collected and put under a collective topic, whereas all of them are included within the classes *Memory and Storage*, *Input Validation*, *Improper Access Control*, *Scope*,*Permission Management*, or *Types and Conversion*. Going through all of them, each weakness will be assigned a class while some of the classes will be singled out and given a more detailed description in the following section:

Rank	Name	Class
1	Out-of-bounds Write	Memory and Storage
2	Improper Neutralization of Input during Web Page Generation (‘Cross-site Scripting’)	Input Validation
3	Out-of-bounds Read	Memory and Storage
4	Improper Input Validation	Input Validation
5	Improper Neutralization of Special Elements used in an OS Command (’OS Command Injection’)	Input Validation
6	Improper Neutralization of Special Elements used in an SQL Command (’SQL Injection’)	Input Validation
7	Use After Free	Memory and Storage
8	Improper Limitation of a Pathname to a Restricted Directory (’Path Traversal’)	Input Validation
9	Cross-Site Request Forgery (CSRF)	Improper Access Control
10	Unrestricted Upload of File with Dangerous Type	Input Validation
11	Missing Authentication for Critical Function	Permission Management
12	Integer Overflow or Wraparound	Types and Conversion
13	Deserialization of Untrusted Data	Input Validation
14	Improper Authentication	Permission Management
15	NULL Pointer Dereference	Memory and Storage
16	Use of Hard-coded Credentials	Permission Management
18	Missing Authorization	Permission Management
19	Incorrect Default Permissions	Permission Management
20	Exposure of Sensitive Information to an Unauthorized Actor	Scope & Improper Access Control
21	Insufficiently Protected Credentials	Permission Management
22	Incorrect Permission Assignment for Critical Resource	Permission Management
23	Improper Restriction of XML External Entity Reference	Input Validation
24	Server-Side Request Forgery (SSRF)	Input Validation
25	Improper Neutralization of Special Elements used in a Command (’Command Injection’)	Input Validation

Another, albeit somewhat smaller, source is a paper by NASA engineers Stacy Nelson and Johann Schumann on trustworthy code review. The following table represents only an excerpt from their questionnaire on possible software bugs, but nevertheless offers the reader a very good selection of errors. They were evaluated on the basis of the rather subjective criteria "Importance" and "Difficulty", whereas the latter indicates the effort it takes to review the code and prove this error to be prevented. [NS04]

As these are quite different from the first example, the new error classes *Arithmetic Error*, *Clean Programming*, *Concurrency*, *Semantic Error*, and *Data Structures* are introduced.

Importance	Difficulty	Error Property	Class
5	3	Divide by zero	Arithmetic Error
5	3	Array index overrun	Memory and Storage
5	5	Mathematical functions sin, cos, tanh	Arithmetic Error
5	1	Use of un-initialized variables or constants	Memory and Storage
3	3	No unused variables or constants	Clean Programming
4	2	All variables explicitly declared	Memory and Storage
5	5	Proper synchronization in multi-threaded execution	Concurrency
4	4	Incorrect computation sequence	Semantic Error
5	3	Loops are executed the correct number of times	Semantic Error
5	3	Each loop terminates	Data Structures
3	2	All possible loop fall-throughs correct	Memory and Storage
4	3	Priority rules and brackets in arithmetic expression evaluation used as required to achieve desired results	Semantic Error
5	5	Resource contention	Concurrency
5	2	Exception handling	Clean Programming
5	5	The design implemented completely and correctly	Semantic Error
4	2	No missing or extraneous functions	Clean Programming
5	1	Error messages and return codes used	Clean Programming
5	1	Good code comments	Clean Programming

Of course, these two sources do not give an overview of all possible errors or error sources in the world of programming, but they provide a good selection to make classifications. For

reasons of relevance and not to go beyond the field of system programming, the categories *Input Validation*, *Permission Management*, *Improper Access Control*, *Clean Programming*, *Data Structures*, and *Semantic Errors* are omitted. The other classes (*Scope*, *Types and Conversion*, *Memory and Storage*, *Arithmetic Errors*, and *Concurrency*) are presented in more detail in the following section. If it's applicable, a short error message in another, unrelated programming language, will be displayed as an example of how this error could look like.

3.2 Detailed Description of the Covered Classes

3.2.1 Scope

Following the principle of "You don't need to know more than the minimum", sometimes also known as the "Need-to-know" or "separation of concerns", it's good practice to restrict methods and parameters to the area that they are needed for, they get "encapsulated" to their scope. This can happen on a bigger scale, such as application areas that only users with a certain level may access, like in the CWE weakness, or on a much lower level, which is when programs are being written. Whereas a programming language is unable to tell the developer which variables should be encapsulated in which scope, it can help with protecting privately declared entities to the point where a developer cannot accidentally use it from somewhere outside.

It can also support the developer since function and variable names may be reused as they are only visible within their small area, as opposed to being visible and callable from anywhere. An example could be a function for an internal calculation that sets a parameter, making it only available to other functions inside the same scope. The same goes for variables. Only those that are necessary to be used from the outside should be available publicly or, even better, have an access-method that provides the value. That way, any other object can only get the value instead of being able to do whatever it wants with it, like change it.

A major topic when it comes to scope is where and when the language and compiler check whether access to a variable is allowed or not. If it's at compile-time, it's fairly easy for the developer to work with it. On the other hand, it could have quite negative consequences when a program tries to access it during runtime. Possible outcomes here could be a termination of the program or the calculation continuing with an undefined value, which could end up corrupting the program. Java, for example, finds simple errors like accessing a variable marked as private during compilation, citing an error like the following:

```
error:  y has private access in Test
```

The Scope is closely related to *Improper Access Control* and *Permission Management*, which were more used in the bigger sense with multiple actors in a working system during the classification. This section here will set a smaller focus, namely variable access in a program.

3.2.2 Types and Conversion

In higher level programming, there are so-called "types" that tell the language how to interpret the value stored in a variable. Some example types are 32-bit integers, floating point values, or Boolean values. Some languages allow the programmer to use their own types by defining enumerations or structures. Since all values are stored in memory as bytes, the compiler must be prevented from misinterpreting values when using the wrong type to "translate" the byte value into its respective value. Three common errors are:

1. Trying to assign a value to a mismatched type or cast it to a type which is unable to interpret the currently saved value
   A typical exception message in Python could look like this (Case: trying to cast the character 'a' to an integer value):
   `ValueError: invalid literal for int() with base 10: 'a'`

2. Using a value that has a different type than required as a parameter for a certain function
   Exception message in Python (Case: Taking each character of the string "123", adding it up to get the sum):
   `TypeError: unsupported operand type(s) for +=: 'int' and 'str'`

3. Over- or underflow of data types
   Trying to to add to or subtract from a value whereas this operation results in a number that exceeds the range of all possible values for this type. Java handles this by wrapping around any value over the maximum to the absolute minimum of that type. (Case: Adding 1 to the maximum value of an Integer. The first output shows the maximum value of that number, whereas the second line is the result of said number with 1 added to it):

```
$ java Overflow
x: 2147483647
x: -2147483648
```

3.2.3 Memory and Storage

A large part of programming is memory management, although the focus has shifted in recent decades. Whereas in the past developers had to especially keep the total available memory in mind, this factor became irrelevant due to ever larger memory chips. The other big factor, runtime memory management, including memory allocation and fragmentation, has always remained an important component. This includes, above all, the guarantee that data is correctly stored where the program expects it to be. This is relevant with respect to the question "Can the required memory be fully allocated and has this been done before the programmer wants to use it?". If not, other relevant program data may be overwritten and cause unwanted behavior. Another special case that is very common when dealing with arrays is an out-of-bounds exception. This occurs when an array is initialized with n values, but the user wants to access the value at a position $> n$. Some programming languages prevent the user from reading these memory cells by terminating the program, while others do nothing about it. This can be very risky, since values that should not be used can be read and misinterpreted within functions, resulting in behavior that is not intended.

Python prevents the user from accessing such a value outside of the defined error by terminating with the following error:

```
IndexError:  list index out of range
```

3.2.4 Arithmetic Errors

Arithmetic errors can occur whenever mathematical operations are performed when executing code. This always involves the danger of trying to calculate a function with an input that is not included in the respective domain. Well-known examples of this are negative values for square roots (with the exception of imaginary numbers) or the division of any number by 0.

Depending on the function and language, these errors can have quite different effects. If they are implemented in a very restricted way, trying to compute something like this will result in an immediate exception and abort the program. Another behavior could be that the program tries to just go through with the defined operation and computes an unexpected value without issuing a warning, which can have dire consequences since these errors can go undetected for a long time. Attempting to use Integers and divide by 0 in Java would produce the following error:

```
Exception in thread "main" java.lang.ArithmeticException:  / by zero
```

3.2.5 Concurrency

Working with multiple threads always poses a risk to program integrity. One major issue that anyone working with concurrency must be aware of is the management of shared resources. Since the execution of compiled code is different on each operating system, it is very difficult, if not impossible, to predict how a program will run. The ability to lock resources with mutual exclusion, also called mutex, obviously helps in determining this, but also introduces new problems, namely deadlocks and livelocks. These can occur when all processes are waiting for each other or constantly trying to respond to each other. This is also a type of unwanted behavior and must usually be prevented by the developer themselves, although some programming languages can support this to some extent. Specifying a particular error here as an example is difficult, since concurrency that goes wrong usually results in unexpected behavior rather than a crash itself. As it is more critical and can pose greater risks to program integrity, this section focuses on access to shared resources rather than mutual exclusion and the resulting locks.

4 Programming Languages

To give a bit more background on C, Ada, and Rust, these three languages will be covered in more detail, namely regarding the following five points:

- **History**: How did this language develop, who made this language, and what were the intentions behind it?
- **Structure**: What is the general structure of a program in this language?
- **Characteristics**: What can this language do better than other common programming languages, and what are some special traits it has?
- **Environment**: What is the general state of the language and its developers?
- **Relevance**: Why and how is this language relevant in the modern world?

4.1 Introduction of C

History

The development of C was started by Dennis Ritchie at AT&T Bell Laboratories in the years 1969-1973 [Rit93] . Taking B and BCPL as successors, the development paid close attention to the pros and the cons of these two languages and took over some of the proven principles. The first ANSI (American National Standards Institute) standardization for C was ratified in 1989, swiftly followed by the ISO (International Organization for Standardization) in 1990. Theoretically, they both refer to different versions, namely C89 and C90, even though the second standard only includes formatting changes. [Sta03, p. 5]

The first extension, which was officially called Amendment 1, was published in 1995, giving it the nickname C95. The main changes were the addition of digraphs, improvements to the library, and macros like `__STDC_VERSION__`, which expands into the standard number of the C version being used. [SW, p. 21]

Four years later, the updated standard ISO/IEC 9899:1999 was released, which is commonly known as C99. Its biggest changes were the support of complex and imaginary numbers, as well as new core features such as variable length arrays or flexible array members. [ISO18, p. 476]
It ended up being updated three times by Technical Corrigenda in 2001, 2004, and 2007 [ISO99] .

It was then fully withdrawn in October 2011 after being revised into the C11 standard [ISO11] . This time, the update was not as big as beforehand, the most notable addition were the big improvements towards multithreading and atomic objects [ISO18, p. 476] .

In 2018, the latest version C17 was introduced. It did not add any new features but rather focused on fixing technical issues and clarifications. [ISO18, p. 476]

Following the usual ISO review period of 5 years, a new version of C is expected around 2023. As there weren't a lot of new features implemented in the past two editions, C can be seen as a very stable and well rounded language.

Structure

A C program usually consists of three parts: Preprocessor commands, declarations, and definitions. It's good to look at an example to understand their purposes. The following example includes the function `print_collatz(int num)` which calculates and prints the Collatz conjecture for the parameter `num`, which is set in the main function to be 42:

```c
#include <stdio.h>
#include <stdlib.h>

void print_collatz(int num);

int main()
{
    print_collatz(42);
    exit(0);
}

void print_collatz(int num)
{
    // Integer values can be negative or 0 as well which would result
    // in a non-terminating sequence, thus we need to rule it out
    if (num <= 0)
    {
        printf("Conjecture can not start with a 0 or negative value\n");
        return;
    }

    int x = num;
    while (1)
    {
```

```
25
26          // If 1 is reached, the Collatz conjecture terminates
27          if (x == 1)
28          {
29              printf("%d", x);
30              break;
31          }
32
33          // Print the current value of x
34          printf("%d, ", x);
35
36          // Following the definition of the Collatz conjecture
37          if (x % 2 == 0)
38          {
39              // If x is even => x = x/2
40              x = x / 2;
41          }
42          else
43          {
44              // If x is uneven => x = 3x + 1
45              x = 3 * x + 1;
46          }
47      }
48      printf("\n");
49  }
```

Listing 4.1: Collatz-Conjecture in C

The first statement of this program is the inclusion of the header file of C's standard I/O library *stdio.h*. When the preprocessor goes over the program for the first time, it replaces this line with the contents of that file, making the functions that were declared there available to be used. An *include*-statement might state a filename in angle brackets, which looks for system header files, or in quotation marks, which searches at the location of the current file. [SW, pp. 7–8]

After that, the declaration of functions for this file follows. These are optional but leaving them out prompts the compiler to output a warning. Declarations define the interface of functions in this file and may, together with other statements that include other parts of the program, be outsourced to a header file, which would then be included in return [SW, Ch. 2]. Splitting declarations and preprocessor instructions to a separate file makes sense in bigger projects as to give a better oversight over the purpose and connections of this file.

The main body of a C file then holds the definitions of the previously declared structures. Function names are not being preceded by a certain keyword but rather by the return type, with it being *void* if it does not have one (see lines 4 and 12).

Characteristics

C is a language that leaves its developers a lot of freedoms while giving them a lot of possibilities, too. C is made very versatile, offering both high-level structures, for instance user-defined *Structure* data types, as well as low-level operations such as directly manipulating memory. This, together with its long history and development parallel to the Unix system [Rit93, p. 1], made it a long time favorite and currently well prevalent language for systems programming. The 50+ years of almost continuous development rounded off a lot of errors which most other programming languages still in use today simply did not have the time for.

Environment

C uses compilers for and runs on virtually any platform, embedded or not. The long history and extensive documentation makes it a thorough language that greatly supports its user. Even though it's a fairly advanced language for a skilled developer, it's also suitable for basic programming classes in tertiary education as many principles such as pointers or the functionality of basic data structures can be explained very well in C. Especially due to the age of C, the documentation is very good. Apart from the official ISO standard, there are plenty of books over the past versions to learn from and work with.

Relevance

C is still very relevant today, both in new development and maintenance of older systems. The Stack Overflow survey for 2021 ranked C at rank 12, with 21,02% of the users giving it as an answer to the question *Which programming, scripting, and markup languages have you done extensive development work in over the past year, and which do you want to work in over the next year?*. [Staa]

Given that C is older than almost 90% of the users taking the survey, that is a very impressive number. With plans being already made for the next review going up in 2023, it also does not seem very likely that the language is going to become irrelevant in the foreseeable future.

4.2 Introduction of Ada

History

In 1974, the United States Department of Defense (DoD) discovered the need of a new programming language due to the multitude of currently used languages as well as the resulting high costs [Bar14, p. 3] . After a lengthy process of development which was all sponsored by the DoD, the ANSI (American National Standards Institute) standard for Ada was first released in 1983, thus naming it Ada 83 [ANS83] . After all, ISO (International Organization for Standardization) certification was not that far away, and so it became ISO standard 8652 in 1987 [ISO67] .

Shortly thereafter, some work on the language in accordance with ISO provisions led to a revised standard in 1988. This draft was then contracted to Intermetrics Inc. which led to the publication of the revised ISO standard in 1995, which became Ada 95. [Bar14, p. 4]

The next iteration was then handled by the Ada Rapporteur Group (ARG), who are still responsible for the maintenance of Ada today. Instead of a revised standard, the ARG only identified and corrected a few flaws, eventually releasing those as *Technical Corrigendum 1* on June 1st, 2001. [ISO01]

Working with other programming languages, some additional improvements to Ada were quickly identified and worked on. These changes were not considered extensive enough to warrant a fully revised standard and were therefore decided to only be an amendment to the language, which was then further developed by the ARG into Ada 2005. [Bar14, p. 4]

With the big steps in computer hardware in the early 2000s, the need for additions, especially in the direction of multiprocessors, was becoming more and more evident. This eventually led to the development of a new edition, which became an ISO standard in December of 2012, thus naming it Ada 2012. [ISO12]

Structure

An Ada program is made up of one to n library units. Each unit can be one of the following:

- **Subprogram** (executable algorithm)
- **Package** (collection of entities such as subprograms, types, etc.)
- **Task Unit** (concurrent computations)

- **Protected Unit** (Coordination for tasks sharing data)
- **Generic Unit** (Parameterized template for subprograms)

Usually, a library unit consists of two parts. The first is a specification which, similar to a header file in C, provides information on the interface to other units. The second one then contains the implementation of this module. As an example of some Ada code, we will take a look at a subprogram, as it's the smallest part of any program.

The following subprogram is designed to print the Collatz conjecture for the input N. Code comments in Ada are realized by a double minus ($--$):

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Collatz
4    (N: Natural := 1) is
5      X: Natural := N;
6  begin
7
8      -- As Natural also allows 0 to be used, we have to exit
9      -- before the loop runs infinitely
10     if X = 0
11     then
12         Put_Line("Conjecture can not start with a 0");
13         return;
14     end if;
15
16     Put_Line("Starting Collatz Conjecture:");
17
18     loop
19         -- If 1 is reached, the Collatz conjecture terminates
20         if X = 1
21         then
22             Put(Natural'Image(X));
23             exit;
24         end if;
25
26         -- Print the current Value of X
27         Put(Natural'Image(X) & ",");
28
29         -- Following the definition of the Collatz conjecture:
30         if X mod 2 = 0
31         then
32             -- If X is even => X = X/2
33             X := X / 2;
34         else
35             -- If X is uneven => X = 3X + 1
36             X := 3 * X + 1;
37         end if;
38     end loop;
39
40 end Collatz;
```

Listing 4.2: Collatz-Conjecture in Ada

The code is pretty straight forward and easy to follow for anybody who has experience in imperative programming. Line 1 contains the import that is necessary for this unit, this is only the standard library for Input/Output. Line 3 is probably the first time that an unknown keyword is met, a procedure. This is a subprogram that does not return a result, contrary to a function [Duf+12, pp. 806, 808] . Between the keywords *procedure* and *is*, the name and parameters are being defined. The latter may have any one of the types *in* (default), *out*, or *inout*. The details regarding functions, procedures, and parameter types is better highlighted in an example:

```
1 function Square_Fun
2   (N: Integer:= 1)
3    return Integer is
4 begin
5     return N * N;
6 end Square_Fun;
```

Listing 4.3: Square Function

```
1 procedure Square_Proc
2   (N: in Integer:= 1;
3    Result: out Integer) is
4 begin
5     Result := N * N;
6 end Square_Proc;
```

Listing 4.4: Square Procedure

Between *is* and *begin*, any variables that are being used inside the function have to be declared (see the Ada code example, line 4). After that, the body of the actual function follows, completed by an *end*. It is optional to state the name of the subprogram at this point but recommended for clarity.

In the previous, longer code sequence (Listing 4.2), the specification is to be found on the lines 3 and 4. It states that this subprogram, called *Collatz*, is a procedure, thus has no return-parameters and takes one input parameter. Between lines 5 and 42, the implementation follows. The clear-cut structure was intended to make the code maintainable.

One last notable thing about the structure is that each file may only contain one library unit, leading to Ada projects tending to get pretty big, as they're very split up from the beginning.

Characteristics

Ada was designed to be a language used in both military and civilian applications that require a high level of safety. At the same time, maintainability and ease of use for the developer were essential points. [Duf+12, p. xviii]
With all of this in mind, the above presented structure was developed and has kept up over the past 35 years. Especially the points about ease of use and maintainability are represented in Ada's syntax. A few examples for this are the clear-cut structure or the use of intuitive keywords such as *procedure*, *function*, *begin*, *end*, etc. that make it easy for an external developer to get a quick overview of the program. The distribution of units

over one file each also helps as functionalities are clearly separated and can be maintained easier.

Apart from the bigger structure and some keywords, there are a few additional notable things that stand out when looking at Ada's syntax, the first being the absence of curly brackets. This, together with the keyword *end* to close a control structure, has a simple reason: It prevents the programmer from accidentally doing something unintended to a control structure while still having the compiler detect no syntax error. It also helps to keep an overview over which loop is being closed in which location, again increasing readability. Regarding the prevention of unintended things, it's good to imagine what could happen in other programming languages where a semicolon behind a conditional statement simply finishes it. The next code block, which would have only been run had the condition been true, would then run every time, rendering the conditional statement basically useless.

Of course, any good developer would probably be able to find this error by testing thoroughly. The compiler on the other hand has no chance of detecting this as due to the very widespread usage of braces to enclose normal code blocks as well as those after a condition, the curly brackets are syntactically correct.

That this cannot happen in Ada on the other hand, can be seen in the following example:

```
1 if Takeoff_Permit = Approved then
2   Roll_To_Position();
3   Initiate_Takeoff();
4 end if;
```

Listing 4.5: Correct code

```
1 if Takeoff_Permit = Approved then ;
2   Roll_To_Position();
3   Initiate_Takeoff();
4 end if;
```

Listing 4.6: Stray semicolon in line 1, does not compile

If a semicolon were to be accidentally left in on line 1 in Listing 4.5 as it is in 4.6, it would be immediately detected by the compiler, which would fail with the warning `error: extra ";" ignored`, as it's missing the corresponding *end if*. This then prevents the developer from accidentally executing anything inside of the condition.

The second notable topic to look at is the difference in equality and assignment operators from Ada to most other programming languages. By using `:=` for assignments and `=` for equality comparison, the very strange assignment sequence, from a mathematical point of view, of $x = x + 1$ does not exist anymore. The equals sign is therefore being used for the (mathematically correct) comparison if x and y are equal by evaluating $x = y$.

To again prevent the developer from accidentally making easy mistakes, Ada forbids assignment in expressions such as an *if*-statement, the condition may only be a boolean expression [Duf+12, p. 124] . In other programming languages, an assignment within such

an expression could then produce results depending on the value that was assigned instead of the comparison with the value currently stored in the variable.

By forbidding this, and using more precise signs for assignment and equality, an Ada developer does not have to be as worried about making these by mistake, even though one should never completely let their guard down. The corresponding code in Ada could look like this:

```
1 if Takeoff_Permit = Approved then
2   Roll_To_Position();
3   Initiate_Takeoff();
4 end if;
```

Listing 4.7: Code with equality

```
1 if Takeoff_Permit := Approved then
2   Roll_To_Position();
3   Initiate_Takeoff();
4 end if;
```

Listing 4.8: Code with assignment in *if*-condition, does not compile

These two examples are just a few of the characteristics and ways on how Ada protects the developer from making easy mistakes that they might be used to doing from programming in other languages.

Environment

Due to the age and niche application area, Ada is not the first language that a hobby programmer would get into. This is also reflected in the Stack Overflow Developer Survey for 2021 which is taken as a reference value for language usage, where Ada is not listed anywhere, not even under the general point *Programming, scripting, and markup languages* [Staa] . A factor for that could be that safety on the level that Ada provides is not really necessary or is simply ignored or disregarded in the majority of applications, especially private projects. The documentation around Ada is scarce, the majority of information can be found in official documentation, reference guides and tutorials from Adacore, the biggest provider of Ada resources.

Relevance

As previously stated, Ada has its main purpose in the aerospace industry, especially in the military area. It's also still relevant today, a few example projects would be the Eurofighter Typhoon which was at some point the biggest Ada project in Europe, or the navigation component of the METEOR missile, developed by MBDA. [Adaa; Adab]

For licensing of an aviation product in the EU, the DO-178C standard is a very important milestone for a programming language to adhere to [Eas] .

Ada stands out because it has proven reliable over time and, most importantly, has the toolchain necessary to successfully develop a program that adheres to these standards, such as GNATcheck or GNATcoverage. [Gnac; Gnad]

In general, it could theoretically be beneficial for companies to start projects in Ada, simply due to its reliability. The biggest problem there lies within the rarity of skilled developers and rather than having each new software engineer go through an extensive training, it's usually decided to simply use another, more widespread language.

4.3 Introduction of Rust

History

Compared to C and Ada, Rust is definitely a newcomer. It was originally started as just a side-project by Graydon Hoare, an employee at Mozilla, in 2006. Over the next three years, the language was improved until basic functionalities were guaranteed. At this point, Mozilla started to sponsor the language and initiated the Servo project, an experimental browser engine, which was one of the first two big Rust projects. Over time, the language evolved with the work of thousands of contributors, being open-source, and had its first stable release, Rust 1.0 in May 2015. [Rusb; Rusd]

In 2017, an edition system was introduced to enable developers to shift between specific versions of the language without breaking existing code. Usually, new features are pushed to all current versions that are compatible with it, which is why the changes are not as big as in C or Ada. The changes that would require extensive rework of existing code are collected and released with a new edition. In retrospect, version 1.0 was titled Rust 2015 and had the theme of "stability". The next big release was Rust 2018, under the topic of "productivity". The biggest change here was the rework of the module system, which is the way that multiple files and parts are included within a project. The current release is Rust 2021, which does not have a specific theme but is aimed to "bring new capabilities and more consistency". Probably the biggest change includes the method of iterating over an array, changing an implicit call that would otherwise break Rust 2015 and Rust 2018 code. To easily switch between versions, each edition comes with an automatic migration tool which highlights the changes that have to be rewritten in order to be compatible with the new version. [Rush]

In August of 2020, Mozilla laid off around 250 employees, some of which were working on Rust [Tea20]. This prompted the Rust Core Team to create a Rust Foundation to hold all trademarks and the financial responsibility. From April of 2021 onwards, anybody can support and donate along the five founding member companies AWS, Huawei, Google,

Microsoft, and Mozilla. [Wil21]
Two months later, Meta, formerly known as Facebook, joined the foundation to support the development of Rust [Kam21] .

Structure

The structure of a Rust program is very similar to C or C++. A big difference is the abstraction of a program over multiple files, which have to be declared as modules if they want to be used somewhere else. When keeping everything within one file, the code looks very similar to C:

```rust
fn main() {
    print_collatz(42);
}

fn print_collatz(num: i32) -> () {
    // Integer values can be negative or 0 as well which would result
    // in a non-terminating sequence, thus we need to rule it out
    if num <= 0 {
        println!("Conjecture can not start with a 0 or negative number")
            ;
        return;
    }

    let mut x = num;

    loop {
        // If 1 is reached, the Collatz conjecture terminates
        if x == 1 {
            print!("{}", x);
            break;
        }

        // Print the current value of x
        print!("{}, ", x);

        // Following the definition of the Collatz conjecture
        if x % 2 == 0 {
            // If x is even => x/2
            x = x / 2;
        } else {
            // If x is uneven => 3x + 1
            x = 3 * x + 1;
        }
    }
    println!();
}
```

Listing 4.9: Collatz-Conjecture in Rust

Compared to the other languages, Rust does not have any declarations of functions or variables that are located in a separate part of the program. Regarding the code, the syntax looks to be a mix of Ada and C, both using braces to declare code blocks like in C and not needing parentheses for conditional control structures like `if` or `for`-loops in Ada.

Characteristics

Rust's uniqueness lies in its way of handling variables and ownership. When a variable is declared, it's marked as immutable by default [KN18, Ch. 3.1] . This is similar to constants in other programming languages and can prevent the accidental changing of variables that were not meant to be changed. If they are supposed to be mutable, the keyword `mut` can be preceded between `let` and the identifier. In case a variable is marked as mutable even though it is never changed, the Rust compiler will detect that and alert the developer that the program can be restricted even more. The following example together with the compilation output shows the reaction by the compiler:

```
1 fn main() {
2     let mut var = 1;
3     println!("var: {}", var);
4 }
```

Listing 4.10: Code with an unused mutability

Compilation output:

```
$ cargo build
   Compiling introduction v0.1.0 (/home/philipp/workspace/Rust/Other/
      introduction)
warning: variable does not need to be mutable
 --> src/main.rs:2:9
  |
2 |     let mut var = 1;
  |         ----^^^
  |         |
  |         help: remove this `mut`
  |
  = note: `#[warn(unused_mut)]` on by default

warning: `introduction` (bin "introduction") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.29s
```

This is only a warning and will not cause the compilation to fail. For good code quality though, any project would emphasize all compiler warnings to be solved before a code gets merged into a production branch. In the rare case that a warning like this has to be ignored, the check can be turned off using the `#[allow(unused_mut)]` command.

Apart from mutability, another big emphasis in Rust is the ownership of variables, which is checked and validated at compile-time. Instead of using a garbage collector or leaving the task of freeing up memory up to the developer, every pointer that is allocated has one dedicated variable that is considered its owner. Once the owner goes out of scope, which usually happens at the closing bracket after a code block, a function called `drop` is automatically called, which then frees all the memory that is not used anymore. To ensure that no two owners try to free the same memory at the same time, the one-owner-per-variable rule was implemented. While it can simply be copied, the ownership can also be fully transferred to the new scope which happens whenever a variable is assigned or passed as a parameter.

The following example highlights it quite well:

```
1  fn main() {
2      let str_1 = String::from("foo");
3
4      // str_1 is moved into the scope of do_stuff_with_string and is
           invalid after this statement
5      do_stuff_with_string(str_1);
6
7      // str_1 is now invalid and can't be printed out
8      println!("str_1 outside: {}", str_1);
9  }
10
11 fn do_stuff_with_string(s: String) {
12     println!("s in function: {}", s);
13 }
```

Listing 4.11: Invalid ownership

Compilation output:

```
$ cargo build
   Compiling introduction v0.1.0 (/home/philipp/workspace/Rust/Other/
       introduction)
error[E0382]: borrow of moved value: `str_1`
 --> src/main.rs:8:35
  |
2 |     let str_1 = String::from("foo");
  |         ----- move occurs because `str_1` has type `String`, which
      does not implement the `Copy` trait
...
5 |     do_stuff_with_string(str_1);
  |                          ----- value moved here
...
8 |     println!("str_1 outside: {}", str_1);
  |                                   ^^^^^ value borrowed here after
      move

For more information about this error, try `rustc --explain E0382`.
error: could not compile `introduction` due to previous error
```

The compiler is very strict on these kinds of errors as it ensures memory safety in Rust.

Obviously, using a variable only once cannot be the solution to gain security, which results in the concept of references and borrowing. References work just like in C, where a pointer to the variable is passed to a function. This transfers a value to a function without it taking ownership for it.

To correct the example above, only the parameter type and way of passing it in the main function have to be changed for this code to successfully compile:

```
fn main() {
    let str_1 = String::from("foo");

    // A reference of str_1 is passed into the scope of
       do_stuff_with_string
    do_stuff_with_string(&str_1);

    // str_1 is still valid and can thus be printed out
    println!("str_1 outside: {}", str_1);
}

fn do_stuff_with_string(s: &String) {
    println!("s in function: {}", *s);
}
```

Listing 4.12: Valid ownership

Program output:

```
$ ./target/debug/introduction
s in function: foo
str_1 outside: foo
```

Just like variables, references are also immutable by default. Due to this, there is no limit to the number of these immutable references as they do not pose a risk to the integrity of the program. In case it's necessary, there's also the possibility to pass a mutable reference but it's only possible to have one for each variable within one scope.

The following examples highlight the difference pretty well:

```
fn main() {
    let mut str_1 = String::from("foo");

    // A mutable reference of str_1 is passed into the scope of the
       function
    do_stuff_with_string(&mut str_1);

    do_stuff_with_string(&mut str_1);

    // str_1 is still valid and can thus be printed out
    println!("str_1 after function call: {}", str_1);
}

```

```rust
fn do_stuff_with_string(s: &mut String) {
    s.push_str("bar");
}
```

Listing 4.13: Valid mutable reference

Program output:

```
$ ./target/debug/introduction
str_1 after function call: foobarbar
```

This example compiles and runs without problem, due to the ownership being applied correctly. Both in line 5 and 7 of Listing 4.13, the `str_1` variable is mutably borrowed but as the reference goes out of scope again after each function call is finished, the multiple borrowing is no problem.

In a case where it does not go out of scope, the compiler reacts accordingly as can be seen in the following example:

```rust
fn main() {
    #[allow(unused_mut)]
    let mut str_1 = String::from("foo");

    let str_2 = &mut str_1;
    let str_3 = &mut str_1;

    println!(
        "str_1, 2, 3 after function call: {}, {}, {}",
        str_1, str_2, str_3
    );
}
```

Listing 4.14: Invalid mutable reference

Compilation output:

```
$ cargo build
   Compiling introduction v0.1.0 (/home/philipp/workspace/Rust/Other/
       introduction)
error[E0499]: cannot borrow ‘str_1‘ as mutable more than once at a time
  --> src/main.rs:6:17
   |
5  |     let str_2 = &mut str_1;
   |                 ---------- first mutable borrow occurs here
6  |     let str_3 = &mut str_1;
   |                 ^^^^^^^^^^ second mutable borrow occurs here
...
10 |         str_1, str_2, str_3
   |                ----- first borrow later used here

error[E0502]: cannot borrow ‘str_1‘ as immutable because it is also
   borrowed as mutable
  --> src/main.rs:10:9
```

```
   |
5  |     let str_2 = &mut str_1;
   |                 ---------- mutable borrow occurs here
...
10 |         str_1, str_2, str_3
   |         ^^^^^  ----- mutable borrow later used here
   |         |
   |         immutable borrow occurs here

Some errors have detailed explanations: E0499, E0502.
For more information about an error, try `rustc --explain E0499`.
error: could not compile `introduction` due to 2 previous errors
```

Here in Listing 4.14, both `str_2` and `str_3` are still in scope at the same time and would both hold the reference to `str_1`. This prompts the compiler to fail, citing two errors, with the first one being the multiple mutable borrow, which happens in lines 5 and 6. The second one is directly related as it forbids the immutable reference in the print-statement in line 10, as the mutable reference from line 5 is still active.

Any operations that are against Rust's rules are not completely disabled, the feature of so-called "unsafe" code blocks is able to circumvent these checks. Within such areas, the compiler does not enforce memory safety and will trust the developer to do the right thing. This, of course, breaks any guarantees that Rust can give in regards to that safety and by that a core language concept. In case a bigger project needs to use mechanics that are disallowed by the compiler, the code areas that need to be manually audited are limited to the unsafe code blocks.

Whenever a Rust program encounters an error status that is unrecoverable, a Panic value is thrown, leading to the immediate termination of the program [KN18, p. 9.3] . In the case that the developer thinks that a situation might be recoverable, they can secure their functions by using `Result` or `Option` types which can then be handled at the function call.

Environment

Rust's environment is very modern compared to C and Ada, with a lot more online resources than books. This is of course due to the age of the programming language itself but probably also because of the continuous release of new Rust versions, rather than big ones with huge changes every couple of years. An advantage of the copious amount of online resources is that actively working with and learning the language is a lot more comfortable than C or Ada. Forums like Stack Overflow are also very active, which is not surprising, given that its survey ranked Rust as the most loved language for six years in a row [Staa] . There's also various free e-books provided by the Rust Language itself, two very popular ones being *The Rust Programming Language* [KN18] and *Rust by Example* [Rusc].

One notable environmental factor that makes it very comfortable to work with Rust is the package manager **cargo**. From creating packages over downloading dependencies to compiling and running code in various configurations, it's a very diverse tool that takes a lot of manual work off of the developer's shoulders.

Relevance

Looking at the Stack Overflow Developer Surveys over the past years, it is clear that Rust has both been very relevant over the past years and will probably remain so [Stab] . This, as well as quite a few people stating that they would like to start or continue working with Rust over the next year as opposed to nobody currently working with it showing interest in another language [Staa, Survey topic *Worked with vs. want to work with*], demonstrates that there's definitely a need for Rust, within private or business projects alike. For companies, there's of course always an incentive to work with programming languages focused on safety and security, as it shows off a good image to customers, being able to promote their programs as safe. Because of this, Rust can definitely be a good choice due to its memory safety and popularity within the community at the same time. This can help companies to find employees who have already worked with the language, during either tertiary education, previous employments, or private projects. Even though the spread of Rust is, just like Ada, not as high as C, it has a big advantage due to being young and promising for the future. However, any argument regarding Rust is only really applicable for companies which are not involved in an industry with strict requirements to certain safety standards as it does not have an ISO certification. Another argument against Rust can be the steady change of the language with updates every six weeks. For being able to work around this, the ability to work with fixed editions was introduced, enabling projects to be set to a certain version to guarantee functionalities and big code bases not having to be completely re-written.

5 Comparison of the Languages

This chapter contains the comparison of errors and especially the reaction by the compilers and runtime environments of C, Ada, and Rust. When dealing with any of the following errors, a certain level of intelligence by the programmer is assumed. Additionally, no means that prevent the standard compiler from finding the error like *unsafe*-blocks in Rust are being used, as using these structures invalidates the entire point of this thesis.

The main point herein lies on possible errors made by developers with a standard knowledge level of the respective programming languages, as there may be complex methods of provoking these errors that fall outside the general knowledge scope. It's mainly focused on the question "Could this error occur in standard, every-day programming?".

5.1 Scope

5.1.1 Access to Entities Outside of the Current Scope

Description

There are plenty of reasons to restrict single modules to only be able to access their minimum necessary scope, like preventing them from doing what they want to across the entire program. Another small advantage of limiting functions and variables as much as possible is that identifiers may be used multiple times if they are only visible in their respective scope. In general, keeping strict scopes also supports the developer to not let them accidentally do things that were not intended, like changing or using a variable that was supposed to be private.

The example that will be shown to highlight how C, Ada, and Rust approach this topic, especially the final bit, is going to include a simple file structure with two files (`main` and `baz`). Here, one module will include the other and try to access both a public (`foo`) and a local (`bar`) variable or function.

C

In C, limiting variables or functions to the local file is realized by preceding the `static` keyword. This limits the identifier to having internal linkage, not making it accessible to anything outside of the current translation unit. [ISO18, p. 29]
The following code shows an example, where `bar` is supposed to be invisible to an outside function:

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    printf("foo: %d\n", foo);
    printf("bar: %d\n", bar);
    exit(0);
}
```

Listing 5.1: Variable access outside of the scope - main.c

```c
int foo = 13;
static int bar = 37;
```

Listing 5.2: Variable access outside of the scope - baz.c

This can only work if `baz.c` is correctly incorporated in the project structure. The seemingly easiest way of doing so would be simply including the file by using `#include "baz.c"`, which would replace the command with the contents of `baz.c`, making both variables available to our main function [SW, p. 7].

As this would be the same as simply copy/pasting the contents into the main file, there is no real use behind doing so. If a program unit is supposed to be shared between multiple other files, C uses header files which contain declarations and describe the interface of their respective unit [SW, Ch. 2].

This would change the code above to this:

```c
#include <stdio.h>
#include <stdlib.h>

#include "baz.h"

int main()
{
    printf("foo: %d\n", foo);
    printf("bar: %d\n", bar);
    exit(0);
}
```

Listing 5.3: main.c using a header file for baz.c

```c
int foo;
static int bar;
```

Listing 5.4: baz.h

```c
#include "baz.h"

int foo = 13;
static int bar = 37;
```

Listing 5.5: baz.c with header

To make use of the limited scope within separate translation units and also have the baz file run and assign the values, both files need to be compiled, for example using the command `gcc main.c baz.c -o output`.

This results in the compiler not throwing any errors, giving the following output:

```
$ ./output
foo: 13
bar: 0
```

This is due to both `foo` and `bar` being declared and initialized with zero as soon as the header file is included in `main.c` [ISO18, p. 101]. When running the program, both variables are defined in `baz` whereas the value of `bar` never leaves the scope of its current translation unit.

This could be circumvented by a policy that states static variables be declared in the source file instead of the header file, thus leading to the compiler not finding `bar` in the main unit:

```
$ gcc main.c baz.c -o output
main.c: In function 'main':
main.c:9:25: error: 'bar' undeclared (first use in this function)
    9 |     printf("bar: %d\n", bar);
      |                         ^~~
main.c:9:25: note: each undeclared identifier is reported only once for
   each function it appears in
```

In general, declaring variables within a header file could end up leading to a lot of confusion as the compiler does not say anything about `bar` being private to the other scope if declared in `baz.h`. This could lead to a developer mistakenly trying to use that variable, which contains the initialized value of zero instead of the one set in the unit.

Ada

In Ada, packages are separated into two files, an `.ads` one, which describes and specifies the specifications of the package, as well as one with the ending `.adb`, which then holds the package implementations [Och, Ch. 7]. To now draw a line between visible and invisible parts of the program, everything inside the `.ads`-file is public by default, whereas a private part may be defined. Anything that should not be visible to the outside of the library unit can be declared here. In practice, this could look like the following code:

```
with Baz;

procedure Main is

begin
    Baz.Foo;
    Baz.Bar;
end Main;
```

Listing 5.6: Visibility in Ada, main.adb

```
package Baz is

    procedure Foo;

private

    procedure Bar;

end Baz;
```

Listing 5.7: Visibility in Ada, baz.ads

```
with Ada.Text_IO; use Ada.Text_IO;

package body Baz is

    procedure Foo is
    begin
        Put_Line ("foo");
    end Foo;

    procedure Bar is
    begin
        Put_Line ("bar");
    end Bar;

end Baz;
```

Listing 5.8: Visibility in Ada, baz.adb

The compilation of this code fails, citing the following reason:

```
$ gprbuild ./src/main.adb -o output
using project file default.gpr
Compile
   [Ada]          main.adb
main.adb:7:07: "Bar" is not a visible entity of "Baz"
gprbuild: *** compilation phase failed
```

This is exactly what a developer would want, as it's directly forbidden at compile time, instead of something happening during the execution.

Rust

Rust has a similar approach to Ada using the concept of private and public instances, with the only big difference being that the private scope is the default, instead of the other way

around. In case a variable, struct, or function needs to be public, the keyword `pub` can be preceded:

```
mod baz;

fn main() {
    baz::foo();
    baz::bar();
}
```

Listing 5.9: Access outside of scope, main.rs

```
pub fn foo() {
    println!("foo");
}

fn bar() {
    println!("bar");
}
```

Listing 5.10: Access outside of scope, baz.rs

The compilation fails and tells us that the function we want to use is private. This is exactly what is intended, not leaving a chance up to unexpected behavior at runtime.

Summary

Trying to access a private variable is a fairly simple error to detect and Ada and Rust make sure to fully prevent the developer from accidentally accessing an entity that shouldn't be. Working with C header files on the other hand, it is definitely possible to use them just slightly incorrectly, resulting in the program compiling but values being used wrongly. The access to the variable that is limited to the internal scope is ultimately prevented, but the program does not mind calling them as they are theoretically correctly instantiated. An inexperienced developer might run into this mistake which could also go a long way without being detected as its effect on the execution is not clearly visible.

5.1.2 Ambiguity

Description

Another mistake that falls under this topic would be the possible ambiguity of a function. Having the same name for two separate procedures may produce outputs that could be very unexpected, especially if a program were to just use one of the two without giving a reason as to why it chose the one it did.

The following scenario is used to simulate how this mistake could occur: A program has two separate sub-units, called *foo* and *bar*, which are both imported by a `main` file. Each unit has a function `print_name` which prints a unique line. The key question in this case is if a developer is able to mistake these two functions for each other or if the programming

languages force them to distinguish between them. Another small point that will be looked at is the question if it's possible to implement a function twice within one module.

C

The implementation in C is pretty straight forward and easy. After importing both files, the main function calls `print_name` on line 7, even though it's unclear which one is meant:

```c
#include <stdlib.h>
#include "foo.h"
#include "bar.h"

int main()
{
    print_name();
    exit(0);
}
```

Listing 5.11: Ambiguity - main.c

```c
void print_name();
```

Listing 5.12: foo.h

```c
#include <stdio.h>
#include "foo.h"

void print_name()
{
    printf("Foo\n");
}
```

Listing 5.13: foo.c

```c
void print_name();
```

Listing 5.14: bar.h

```c
#include <stdio.h>
#include "bar.h"

void print_name()
{
    printf("Bar\n");
}
```

Listing 5.15: Ambiguity - bar.c

When compiling this program using `gcc main.c foo.c bar.c -o output`, the following output is returned:

```
$ gcc main.c foo.c bar.c -o output
/usr/bin/ld: /tmp/ccUnYt9p.o: in function 'print_name':
bar.c:(.text+0x0): multiple definition of 'print_name'; /tmp/ccv8gZbp.o:
   foo.c:(.text+0x0): first defined here
collect2: error: ld returned 1 exit status
```

This output is, compared to the others, not raised by the compiler but by the linker, which can be seen by the errors not appearing in the .c-file but rather in the object file with a .o-ending, which is located in the /tmp/ folder. The reason for it not being found by the initial compilation is the separate translation of each unit. Within their scope,

the function name is unique and thus no problem. As soon as the project is being linked together though, the same names are detected and the entire process is halted.

Pretty much the same error would come up if a function were to be defined more than once within one file, this time during the initial compilation process:

```c
#include <stdlib.h>
#include <stdio.h>

void print_name();

int main()
{
    print_name();
    exit(0);
}

void print_name()
{
    printf("Foo\n");
}

void print_name()
{
    printf("Bar\n");
}
```

Listing 5.16: main.c with an ambiguous function name

Compilation Output:

```
$ gcc main.c -o output
main.c:16:6: error: redefinition of 'print_name'
   16 | void print_name() {
      |      ^~~~~~~~~~
main.c:12:6: note: previous definition of 'print_name' was here
   12 | void print_name() {
      |      ^~~~~~~~~~
```

Ada

This error is only partially applicable in Ada. With each file containing only the unit that is the same as its filename, having two files and thus two library units with an identical filename in the same place is not possible. To try and circumvent this, two files with the same name will be placed in different subfolders *foo/* and *bar/*:

```
├── main.adb
├── foo/
│   └── baz.adb
└── bar/
    └── baz.adb
```

Both files contain the following code, only with a slight difference in the printed line ("Baz in Foo/Bar folder!"):

```
1 with Ada.Text_IO;
2 use Ada.Text_IO;
3
4 procedure Baz is
5
6 begin
7     Put_Line("Baz in Foo folder!");
8 end Baz;
```

Listing 5.17: Content of file baz.adb

When Ada compiles using `gprbuild`, it uses a project file and goes through the directories specified under *Source_Dirs*. In case it stumbles upon a file that has already been compiled with that same name, it is skipped. [Gpr, Ch. 2.2.2]
With the following project file, this results in `src/foo/baz.adb` being compiled as the first `baz.adb` and `src/bar/baz.adb` not being touched:

```
1 project Structure_Ambiguity is
2     for Source_Dirs use ("src/foo", "src/bar", "src");
3     for Object_Dir use "obj";
4     for Main use ("main.adb");
5 end Structure_Ambiguity;
```

Listing 5.18: Project file for Structure Ambiguity

This can also be seen in the compilation output, where only one file called `baz.adb` is being compiled:

```
$ gprbuild
using project file structure_ambiguity.gpr
Compile
   [Ada]          main.adb
   [Ada]          baz.adb
Bind
   [gprbind]      main.bexch
   [Ada]          main.ali
Link
   [link]         main.adb
```

As expected, the output is the following:

```
Baz in Foo folder!
```

Not saying anything about the second component not even compiling and not giving an error message is definitely a flaw that can be raised when talking about Ada. It of course makes it very safe by not even giving in to the possibility of an ambiguity but puts this burden onto the developer to make sure they do not accidentally have two modules with the same name within one project.

The equivalent for the second part of the question would be trying to define two procedures with the same name within one package. An implementation could look like this:

```
1 with FooBar; use FooBar;
2
3 procedure Main is
4 begin
5
6     PrintName;
7
8 end Main;
```

Listing 5.19: Ambiguity in Ada, main.adb

```
1 package FooBar is
2
3     procedure PrintName;
4
5 end FooBar;
```

Listing 5.20: Ambiguity in Ada, foobar.ads

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body FooBar is
4
5      procedure PrintName is
6      begin
7          Put_Line("Foo");
8      end PrintName;
9
10     procedure PrintName is
11     begin
12         Put_Line("Bar");
13     end PrintName;
14
15 end FooBar;
```

Listing 5.21: Ambiguity in Ada, foobar.adb

The Ada compiler detects this easy mistake as expected, giving the following response:

```
$ gprbuild main.adb -o output
using project file /usr/share/gpr/_default.gpr
Compile
   [Ada]          foobar.adb
foobar.adb:10:05: duplicate body for "PrintName" declared at foobar.ads
   :3
gprbuild: *** compilation phase failed
```

For this second part, Ada only tells the developer where the repetition takes place and not where all of the definitions are, like C did. Nevertheless, it fully detects the mistake and fails to compile.

Rust

Similar to Ada, the first problem is only applicable in Rust via a few detours. Trying to use the same function name found in different files would not carry much value for this test as calls to an external namespace must always be distinguished by preceding a `<namespace>::`. To circumvent this, we try to have the same filename and thus namespace name (in this case `baz::`) by implementing the following file structure:

```
├── main.rs
├── foo/
│   └── baz.rs
└── bar/
    └── baz.rs
```

A structure like that would not work if we want to use the `baz`-files. Looking at the reason behind it will also answer the question why an ambiguity like that cannot happen that easily. Importing another module in Rust is done by using the keyword `mod <modulename>`. This prompts the compiler to look for a file named `modulename.rs` or `modulename/mod.rs` based on the current directory. [KN18, Ch. 7.5]
In our case this would mean that a file named `foo.rs` or `foo/mod.rs` would be necessary. Thus, the final structure could look like this:

```
├── main.rs
├── foo.rs
├── foo/
│   └── baz.rs
├── bar.rs
└── bar/
    └── baz.rs
```

This new file (in our case `foo.rs`) then acts as the interface to the module and its contents. Depending on the use-case, these can be kept private by only using another `mod`, or be made available to the higher scope by using `pub mod`. Calling a function called `print_name()` in `foo/baz.rs`, which prints out the String "Baz in Foo folder!", can now be done in two different ways.

Using `mod`:

```
mod foo;

fn main() {
    foo::print_baz_name();
}
```

Listing 5.22: Call using `mod`, main.rs

```
mod baz;

pub fn print_baz_name() {
    baz::print_name();
}
```

Listing 5.23: Call using `mod`, foo.rs

Or using `pub mod`:

```
mod foo;

fn main() {
    foo::baz::print_name();
}
```

Listing 5.24: Call using `pub mod`, main.rs

```
pub mod baz;
```

Listing 5.25: Call using `pub mod`, foo.rs

Continuing with `pub mod` and using the keyword `use foo::baz` in the main file, we can now tell our namespace to include the content of baz to be able to use it directly. For this, the interface to `baz.rs` has to be made public. We can now skip the `foo::` and directly call the `print_name()`-function:

```
mod foo;
use foo::baz;

fn main() {
    baz::print_name();
}
```

Listing 5.26: Using namespace `foo::baz`

To now try to run into an ambiguity, we do the same thing with the bar-module. That way, the namespace would be confused as to which of the two possible `baz::` needs to be evaluated:

```
mod foo;
use foo::baz;
mod bar;
use bar::baz;

fn main() {
    baz::print_name();
}
```

Listing 5.27: Trying to import a namespace with the same name twice

The Rust compiler is good enough to detect this ambiguity, if maliciously provoked or not, and fails with the following message:

```
$ cargo build
   Compiling incorrect v0.1.0 (/home/philipp/workspace/Rust/Code/
      Comparison/Scope/Ambiguity/incorrect)
error[E0252]: the name `baz` is defined multiple times
 --> src/main.rs:4:5
  |
2 | use foo::baz;
  |     -------- previous import of the module `baz` here
3 | mod bar;
4 | use bar::baz;
  |     ^^^^^^^^ `baz` reimported here
  |
  = note: `baz` must be defined only once in the type namespace of this
     module
help: you can use `as` to change the binding name of the import
  |
4 | use bar::baz as other_baz;
  |     ~~~~~~~~~~~~~~~~~~~~~

warning: unused import: `bar::baz`
 --> src/main.rs:4:5
  |
4 | use bar::baz;
  |     ^^^^^^^^
  |
  = note: `#[warn(unused_imports)]` on by default

For more information about this error, try `rustc --explain E0252`.
warning: `incorrect` (bin "incorrect") generated 1 warning
error: could not compile `incorrect` due to previous error; 1 warning
   emitted
```

The error message gives clear feedback to the developer that a module (and thus namespace) name must only be defined once. Including multiple imports with the same name can still work by changing the identifier using the keyword `as`.

The second part of the ambiguity-question is the multiple definition of a certain function within the same module. The implementation is again very simple:

```
fn main() {
    print_name();
}

fn print_name() {
    println!("Foo");
}

fn print_name() {
    println!("Bar");
}
```

Listing 5.28: Ambiguity attempt in Rust

As expected, the compilation fails with the following output:

```
$ cargo build
   Compiling incorrect v0.1.0 (/home/philipp/workspace/Rust/Code/
      Comparison/Scope/Ambiguity_2/incorrect)
error[E0428]: the name `print_name` is defined multiple times
 --> src/main.rs:9:1
  |
5 | fn print_name() {
  | --------------- previous definition of the value `print_name` here
...
9 | fn print_name() {
  | ^^^^^^^^^^^^^^^ `print_name` redefined here
  |
  = note: `print_name` must be defined only once in the value namespace
     of this module

For more information about this error, try `rustc --explain E0428`.
error: could not compile `incorrect` due to previous error
```

Looking at both compilation outputs, Rust gives a lot of background details as to where and why it failed by giving the location of both occurrences.

Summary

The first part of this question was whether the same function name may appear twice in different files and if this could lead to some kind of mix up when using them. None of the three languages ended up being non-deterministic about their execution by somehow including and executing both. C fully detected this mistake during the linkage process, successfully compiling the individual units but failing as soon as these were connected to each other. Due to the rules that Ada has for naming units, and Rust has for using namespaces, these two languages had to be approached by trying to work with subfolders. Ada, on the one hand, was very interesting as the compilation only translated the first

occurrence of a unit but skipped the rest. This really puts the burden onto the developer to make sure that any two modules are not named the same, especially as the compilation was successful and did not raise any errors. This could lead to unexpected behavior, as those two modules might not necessarily behave the same, even though they have the same name. Rust, on the other hand, is very strict when using external modules. The attempt to get around the problem of the preceding namespace name was detected and prevented by the compiler. This forces the developer to use different namespace names and by that breaks apart any chance of accidentally running into an ambiguity.

The second part of this topic checked whether the same function is able to be defined multiple times within one unit. All three compilers were able to detect this easily made mistake. While Ada only gave the location of the repetitions, the compilation outputs of C and Rust gave a lot more details about what went wrong and where it failed by presenting all occurrences, not just the ones that are faulty, helping the developer to fix it.

5.2 Types and Conversion

5.2.1 Assigning a Wrong Type

Description

Assigning variables and storing values is one of the key concepts of imperative programming. To determine how these saved bytes are being interpreted, every programming language has types, the most known ones being Integers, Floating Point values, and characters.
It could now happen that a developer tries to assign to a variable but unknowingly passes a value using the wrong type. An example could be wanting to assign '9', which is written as a character with the single quotation marks, to a variable with the type Integer. If the programming language does not prevent a developer from doing so, it has two approaches at using this value. The first one is simply parsing it as a number which is what the developer would want in this case. This would only work with the characters '0' - '9' though, making it a not really practical way of dealing with characters. The other one would be taking the memory value of how the Char '9' is stored and interpret it as if it was an Integer. This would result in the number 57 being assigned as that is the ASCII-value of the character '9' [ISO03, Part 7: Latin/Greek alphabet, Ch. 6.1].

This example will also be used for this section and is going to be tested in C, Ada, and Rust.

C

Taking the same example from above, we try to assign the value '9' to a variable of type `int`:

```c
#include <stdlib.h>
#include <stdio.h>

int main()
{
    int number = 9;
    printf("Number: %d\n", number
        );
    exit(0);
}
```

Listing 5.29: Correctly assigning a number to an int variable

```c
#include <stdlib.h>
#include <stdio.h>

int main()
{
    int number = '9';
    printf("Number: %d\n", number
        );
    exit(0);
}
```

Listing 5.30: Attempting to assign a character

Output:

```
Number: 9
```

Output:

```
Number: 57
```

As already talked about in the beginning, C is a language that leaves a lot of power and responsibility to the programmer. This is the case here as well, by behaving just like the introduction predicted. The compilation finishes without any warnings as this is a perfectly valid assignment [ISO18, p. 73]. This means that the character '9' is automatically converted to Integer, assigning the representing ASCII value to the variable `number`.

Ada

The Ada implementation is very similar:

```ada
with Ada.Text_IO;
use Ada.Text_IO;

procedure Main is
    Number: Integer;
begin
    Number := 9;
    Put_Line("Number:" &
        Integer'Image(Number));
end Main;
```

Listing 5.31: Correctly assigning a number to an Integer variable

```ada
with Ada.Text_IO;
use Ada.Text_IO;

procedure Main is
    Number: Integer;
begin
    Number := '9';
    Put_Line("Number:" &
        Integer'Image(Number));
end Main;
```

Listing 5.32: Attempting to assign a character

While the correct implementation compiles and executes as expected, the incorrect code part returns the following error:

```
$ gprbuild ./src/main.adb -o output
using project file assignment.gpr
Compile
```

```
   [Ada]          main.adb
main.adb:7:14: expected type "Standard.Integer"
main.adb:7:14: found a character type
gprbuild: *** compilation phase failed
```

Ada's strong typing system prevents the developer from (accidentally) converting values to a different type when assigning it to a variable. This rule can be found in section 5.2 of the Ada Reference Manual - 2012 Edition which states that the type of the value to be assigned has to be the same as the variable's [Duf+12, p. 140].

Rust

The following Code tries to assign a value in Rust. The result would also compile if the `:i16` identifier were to be left out but would then assign the type char to the variable. As we want to force two different types, the Integer type is explicitly stated:

```
fn main() {
    let number: i16 = 9;
    println!("Number: {}", number
        );
}
```

Listing 5.33: Correctly assigning a number to an i16 variable

```
fn main() {
    let number: i16 = '9';
    println!("Number: {}", number
        );
}
```

Listing 5.34: Attempting to assign a character

As expected, the compilation fails, citing "mismatched types":

```
$ cargo build
   Compiling assignment v0.1.0 (/home/philipp/workspace/Rust/Code/
      Comparison/Types/assignment)
error[E0308]: mismatched types
 --> src/main.rs:2:23
  |
2 |     let number: i16 = '9';
  |                 ---   ^^^ expected `i16`, found `char`
  |                 |
  |                 expected due to this

For more information about this error, try `rustc --explain E0308`.
error: could not compile `assignment` due to previous error
```

Trying to assign the value in this place falls under the pretty strict Rust concept of coercion, which is an implicit way of converting a value at assignment [Coe]. As int and char are not compatible [The, Ch. 10.7], the coercion that were to take place here is invalid, prompting the compiler to exit with a mismatched types warning.

Summary

The example in C shows why it's important for programming languages focusing on safety and security to have a strong typing system. Detecting things like mismatched types during compilation is fairly easy and it's always better to have a compilation fail rather than have the program do unexpected things during runtime. Ada solves it by being very strict, not allowing assigning any different type without an explicit conversion expression. Rust leaves a bit more freedom by allowing an implicit conversion using the concept of coercion, even though the types that may be converted are pretty strict. This allows both those languages to be pretty safe regarding accidental implicit type conversions when assigning values that could cause unexpected behavior during runtime. Explicit conversion and type casting will be further discussed in section 5.2.2 Conversion and Type Limits. Rust is also the only one of the three languages that allows a variable to be assigned without explicitly stating a type, even though the type is still implicitly decided behind the scenes [Rusc, Ch. 5.2].

5.2.2 Conversion and Type Limits

Description

Every variable takes up space in a machine's memory. How much that eventually adds up to is determined by the type, which defines how many bytes are reserved to store the value.

When talking about Integers, types with more space can usually also store a bigger value range. An unsigned (only positive values) 8-bit number can therefore represent $2^8 = 256$ different values, whereas a 16-bit unsigned integer has one of $2^{16} = 65536$ possible values. Floating point types can also vary in size, determining both the value range as well as the accuracy of the stored number [Inf, Ch. 3.3].

When writing a program, the need to convert between these types can often arise, like parsing an array of characters into their number representation or converting an Integer into a floating point number for more accurate calculations. Whenever such a conversion takes place, it could always happen that the value to be parsed exceeds the maximum boundaries of the type of the target variable. The following section will check each language for the types it has, how conversion works, how it reacts to an overflow, and which problems could arise. They will mainly focus on the Integer types as converting Integer to floating point values is a little more complex due to various levels of accuracy defined by the language implementation.

C

C has five signed integer types, namely `char`, `short int`, `int`, `long int`, and `long long int`, all having an unsigned variant as well. Together with `char` and the three floating point types `float`, `double`, and `long double`, they make up the basic types of C. [ISO18, p. 31]
Table 5.1 below shows the minimum size of the integer types defined in `<limits.h>` and the limits of the number ranges that can be displayed with these types. [ISO18, pp. 20 sqq.]

These sizes are not the ones that will also be used in every operating system, they are just the minimum. The final size is determined by the operating system, whereas the only constraint is that the following equation about the types' sizes still applies: [ISO18, pp. 31, 37]

$$\texttt{char} \leq \texttt{short int} \leq \texttt{int} \leq \texttt{long int} \leq \texttt{long long int}$$

As the type size may be redefined by the individual implementation, the eventual type sizes on a machine can be bigger and thus display more values. These sizes can then found out by looking at the ranges, for example using the constants `INT_MIN` to `INT_MAX`. [ISO18, p. 369]

type	Bit size	sign	min limit	max limit
char	8	signed	$-(2^7-1)$	2^7-1
		unsigned	0	2^8-1
short int	16	signed	$-(2^{15}-1)$	$2^{15}-1$
		unsigned	0	$2^{16}-1$
int	16	signed	$-(2^{15}-1)$	$2^{15}-1$
		unsigned	0	$2^{16}-1$
long	32	signed	$-(2^{31}-1)$	$2^{31}-1$
		unsigned	0	$2^{32}-1$
long long	64	signed	$-(2^{63}-1)$	$2^{63}-1$
		unsigned	0	$2^{64}-1$

Table 5.1: Minimum type sizes in C

Hereby, the `char`-type is special as its signed variant is large enough and also used for storing any of the basic character set. [ISO18, p. 31]

A typecast in C is realized by preceding the value that is to be converted with the new type within parentheses [ISO18, pp. 65 sq.]. As the range of a type is automatically a subrange of a type of the same signedness but higher conversion rank, which follows the same order as the above equation, casting for example from an int to a long value does not hold any issues. The other way is also not an issue as long as the value does not exceed the maximum range of the target type. The following example shows what happens if that occurs:

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int main()
{
    signed int number = 4660;

    signed short new_number = (
       signed short)number;

    printf("Original Number: %d\n
       ", number);
    printf("Cast number: %hd\n",
       new_number);

    exit(0);
}
```

Listing 5.35: Typecast that stays within range

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int main()
{
    signed int number = 4660;
    number += pow(2, 18);
    signed short new_number = (
       signed short)number;

    printf("Original Number: %d\n
       ", number);
    printf("Cast number: %hd\n",
       new_number);

    exit(0);
}
```

Listing 5.36: Typecast that exceeds the type range

Output:

```
Original Number: 4660
Cast number: 4660
```

Output:

```
Original Number: 266804
Cast number: 4660
```

From getting the sizes beforehand, it was determined that `signed int` is 4, while `signed short` is 2 bytes long. The number 4660 in binary representation is 00010010 00110100 and within the range of a 16-bit signed number. This means that it fits well within the variable `new_number`. What happens if the same operation is now attempted but with a number too large for a 2-byte type can be seen in the second example. To make sure that the new number is out of the range for `short int`, 2^{18} is added to it. In the `signed int`-variable, this simply switches the 18th bit to a 1. If the number is now cast to `short int`, the value it holds is still 4660 as if nothing had been added. This is due to the other 2 bytes at the "front" of the number being cut off, leading to the 2^{18} being discarded.

Something similar can be observed when looking at an overflowing Integer value. The following code simply sets a variable to the maximum of its type and then adds 1 to it:

```c
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>

int main()
{
    unsigned short int number = USHRT_MAX;
    printf("Number: %d\n", number);

    number += 1;
    printf("Number: %d\n", number);

    exit(0);
}
```

Listing 5.37: Causing an Integer overflow in C

Program Output:

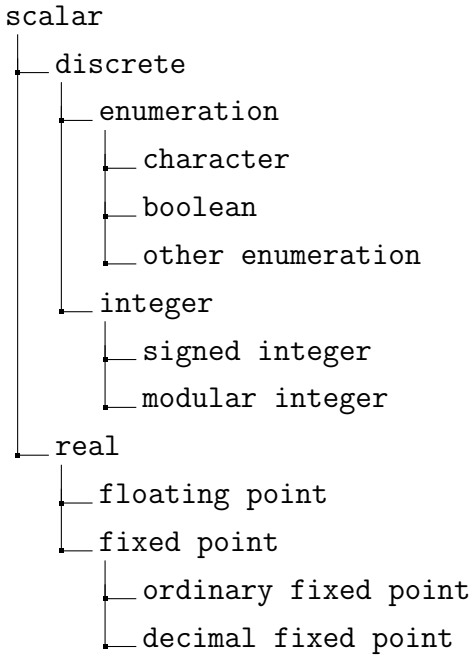
```
Number: 65535
Number: 0
```

The addition is regularly executed and the value is being stored back into the variable. As the value is now 17 bits long, exceeding the 16 bit length of `short int`, the first bit is being cut off, resulting in the final value being 0.

The biggest issue with this behavior is that C does this without hesitation or crashing the program. This is a big downside of a programming language giving a lot of power to its developer. If they do not know how to correctly use it, it's easy to make mistakes that might go a long way without being detected. A more experienced developer could on the other hand use this feature to their advantage, as it is very deterministic.

Ada

Ada pre-defines a type system that follows categories organized like a tree. The ones to focus on in our case are the scalar types. The full structure can be found in Chapter 3.2 of the *Ada 2012 Reference Manual*.



Focusing on the integer types, the distinction of modular and signed types catches the eye. The difference between those is the maximum value range as well as the behavior when an overflow takes place. Variables with a modular type are unsigned and will wrap around as soon as their limits are exceeded [Duf+12, p. 47]. The behavior of a signed type will be looked at in the following code examples.

Additionally to the predefined ones, an implementation may provide additional types. Similar to C, the only restraint is that the ranges of "smaller" types must not be wider than their next bigger neighbor [Duf+12, p. 47].
Working with the GNAT compiler, the following types are defined additionally:

Type	Byte size (signed)
Short_Short_Integer	1
Short_Integer	2
Integer	4
Long_Integer	4
Long_Long_Integer	8
Long_Long_Long_Integer	8

Long_Integer and Long_Long_Long_Integer are a bit special as they do not always have the same size, it depends on the system's architecture. Long_Integer has a guaranteed size of 32 bits and a size of 64 bits on most 64-bit targets, while Long_Long_Long_Integer is 16

bytes wide on 64-bit targets instead of 8. [Gnaa, Ch. 7]

Ada's type system also allows the developer to implement their own subtypes. Two predefined ones are *Natural* and *Positive*, staying close to the mathematical sets $\mathbb{N}$ and $\mathbb{N}^+$, whereas the former implements a range from 0 to the Integer Max, the latter the same but starting at 1.

A typecast or conversion of two values is only allowed following rules that are pretty strict, the one that is important to us being "*The operand type shall be covered by or descended from the target type [...]*" [Duf+12, pp. 126 sqq.]. This essentially means that the value range of the type to be cast to must be equal to or greater than the original type. The difference is illustrated in the following two examples:

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4      Number: Integer := Integer'Last;
5      Number_Long: Long_Integer;
6  begin
7      Put_Line("Number:" & Integer'Image(Number));
8      Number_Long := Long_Integer(Number);
9      Put_Line("Number Long:" & Long_Integer'Image(Number_Long));
10 end Main;
```

Listing 5.38: Typecasting in Ada from a smaller to a larger type

Compilation Output:

```
$ gprbuild src/main.adb
using project file typecast.
   gpr
Compile
   [Ada]          main.adb
Bind
   [gprbind]      main.bexch
   [Ada]          main.ali
Link
   [link]         main.adb
```

Program Output:

```
Number: 2147483647
Number Long: 2147483647
```

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
    Number: Integer := Integer'Last;
    Number_Short: Short_Integer;
begin
    Put_Line("Number:" & Integer'Image(Number));
    Number_Short := Short_Integer(Number);
    Put_Line("Number Short:" & Short_Integer'Image(Number_Short));
end Main;
```

Listing 5.39: Typecasting in Ada from a larger to a smaller type

Compilation Output:

```
$ gprbuild src/main.adb
using project file typecast.gpr
Compile
   [Ada]          main.adb
main.adb:8:20: warning: value not in range of type "Standard.
   Short_Integer"
main.adb:8:20: warning: "Constraint_Error" will be raised at run time
Bind
   [gprbind]      main.bexch
   [Ada]          main.ali
Link
   [link]         main.adb
```

Program Output:

```
Number: 2147483647

raised CONSTRAINT_ERROR : main.adb:12 range check failed
```

The first example (Listing 5.38) compiles and runs without any problem, as expected. This is due to the range of Integer being contained within the possible values of Long_Integer.

The second code snippet (Listing 5.39) shows the illegal conversion, trying to fit a value into a smaller type. Ada recognizes the large number that will be cast to the smaller type and outputs a warning, already foreshadowing the Constraint Error that will be thrown as soon as this conversion is attempted. That this warning can only work if there's a value written as a number literal will be shown in the following example that deals with a type overflow.

When trying to store a number too large for its type, Ada has two different reactions to the addition that would lead to the number being outside of the maximum range. The only difference in these two code snippets is that the number that has to be added is written as a number literal in the first example, while in the second one, it's coming from an external function:

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3
4 procedure Main is
5    Number: Integer := Integer'Last
        ;
6 begin
7    Number := Number + 1;
8    Put_Line("Max Integer:" &
        Integer'Image(Number));
9 end Main;
```

Listing 5.40: Directly adding a number

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3
4 procedure Main is
5     Number: Integer := Integer'
        Last;
6 begin
7     Number := Number + 1;
8     Put_Line("Max Integer:" &
        Integer'Image(Number));
9 end Main;
```

Listing 5.41: Adding the result of a function

Compilation Output:

```
$ gprbuild ./src/main.adb
using project file overflow.gpr
Compile
   [Ada]          main.adb
main.adb:7:21: warning: value not
   in range of type "Standard.
   Integer"
main.adb:7:21: warning: "
   Constraint_Error" will be
   raised at run time
Bind
   [gprbind]      main.bexch
   [Ada]          main.ali
Link
   [link]         main.adb
```

Compilation Output:

```
$ gprbuild ./src/main.adb
using project file overflow.gpr
Compile
   [Ada]          main.adb
Bind
   [gprbind]      main.bexch
   [Ada]          main.ali
Link
   [link]         main.adb
```

The compilation is successful both times, even though Ada again recognizes the explicit value of 1 in the first example and outputs a warning that it will overflow the variable Number, similar as it does for typecasting. It does not do so in the second part where the value to be added comes from an external module. Regardless of that, both programs terminate at the addition operation, citing a `CONSTRAINT_ERROR`.

Rust

Types in Rust are quite a bit more intuitive as their size can be directly derived from the name. The standard pre-defined types include the signed Integer values `i8`, `i16`, `i32`, `i64`, `i128` together with their unsigned counterparts (`u8`, `u16`, ...), the two floating point values `f32` and `f64`, `bool`, and `char`. The only types that are dependent on the computer's architecture are `isize` and `usize`, primarily used for indexing some kind of collection. [KN18, Ch. 3.2]

Typecasting in Rust can be explicit and implicit, whereas the latter was already briefly covered in section 5.2.1 Assigning a Wrong Type. Explicit conversions can be done using either `as` or `from`, whereas the former is an operator and the latter a trait implemented for the numeric types. The following two short examples show the difference in usage:

```
fn main() {
    let number: i32 = i32::MAX;
    let number_long: i64 = i64::from(number);
}
```

Listing 5.42: Rust Typecast using `from`

```
fn main() {
    let number: i32 = i32::MAX;
    let number_long: i64 = number as i64;
}
```

Listing 5.43: Rust Typecast using `asr`

As long as the target type is able to include the entire range of the origin type, both these casts have the same result. The other direction is more interesting as this is where errors could occur. Depending on how the program is supposed to react to a typecast that might be too big for the value, the operator can be chosen. While `as` simply truncates the value and cuts off any bits that are larger than the target type [The, Ch. 8.2.4], the trait `From` is not implemented for typecasts in the "wrong" direction. This can be seen when trying to cast an `i32`-value into an `i16`-type:

```
fn main() {
    let number: i32 = 1;
    let number_short: i16 = i16::from(number);
}
```

Listing 5.44: Typecasting a larger into a smaller type using `from`

Compilation Output:

```
$ cargo build
   Compiling typecast v0.1.0 (/home/philipp/workspace/Rust/Code/
      Comparison/Types/typecast)
error[E0277]: the trait bound `i16: From<i32>` is not satisfied
   --> src/main.rs:3:29
    |
3   |     let number_short: i16 = i16::from(number);
    |                             ^^^^^^^^^ the trait `From<i32>` is not
        implemented for `i16`
    |
    = help: the following implementations were found:
              <i16 as From<NonZeroI16>>
              <i16 as From<bool>>
              <i16 as From<i8>>
              <i16 as From<u8>>
note: required by `from`
   --> /home/philipp/.rustup/toolchains/stable-x86_64-unknown-linux-gnu/
      lib/rustlib/src/rust/library/core/src/convert/mod.rs:373:5
    |
373 |     fn from(_: T) -> Self;
    |     ^^^^^^^^^^^^^^^^^^^^^^

For more information about this error, try `rustc --explain E0277`.
error: could not compile `typecast` due to previous error
```

The help within the output tells the developer that the trait `From<i32>` is not implemented for `i16` but instead for those types whose ranges are smaller than the own. This can of course also be a disadvantage in case the developer has that in mind but wants to cast the value nevertheless. To not cut that opportunity completely and force them to use the more error-prone and undetermined `as`, the trait `TryFrom` is implemented for all numeric types. Instead of the target type directly, it returns a `Result` which is set to `Ok()` or `Err()`, depending on whether the value fits into the new type or not.

Regarding an overflow, Rust has an interesting approach that involves its two compile modes *Debug* and *Release*. The following code will be compiled in both modes:

```
fn main() {
    let mut number: i16 = i16::MAX;
    println!("Number: {}", number);

    number += 1;
    println!("Number after addition: {}", number);
}
```

Listing 5.45: Triggering an Integer overflow in Rust

This code compiles fine with either no flag (Debug mode), or the *–release*-flag appended to the *cargo build* command. A difference lies within the execution of both programs:

Program output from running the code in *Debug*:

```
$ ./target/debug/overflow
Number: 32767
thread 'main' panicked at 'attempt to add with overflow', src/main.rs
   :5:5
note: run with `RUST_BACKTRACE=1` environment variable to display a
   backtrace
```

Program output from running the code in *Release*

```
$ ./target/release/overflow
Number: 32767
Number after addition: -32768
```

While the Debug mode, which is usually used by programmers during the development cycle, crashes, the build using the Release flag keeps running and acts like C by wrapping around the added value using the two's complement [KN18, Ch. 3.2]. This is due to performance reasons as calculations and assignments are used plenty of times during programming and checking each one for an overflow can be time- and labor-intensive [Rusi, RFC #0560] and are thus left out. In case a wraparound is the intended behavior, other methods like `wrapping_add` are implemented as relying on the overflow wrapping is considered an error. Using that function on the above example returns the same, overflowing output:

```
fn main() {
    let mut number: i16 = i16::MAX;
    println!("Number: {}", number);

    number = number.wrapping_add(1);
    println!("Number after addition: {}", number);
}
```

Listing 5.46: Triggering an Integer overflow using `wrapping_add`

Output:

```
$ ./target/debug/overflow
Number: 32767
Number after addition: -32768
```

To make sure that even after thorough testing, no unexpected results are happening, a set of `checked_*` methods are also provided by the Numeric types which return an `Option` value with the value *None* in case an overflow occurs.

Summary

This is probably the topic with the most diverse reaction by all three programming languages. C starts off with the most liberal approach. Any type conversion or cast within Integer values is not explicitly taking place, the bytes are simply copied over, whereas any bits that are excess are cut off. The same happens in case of an integer overflow, the number within the memory cell is simply calculated and then evaluated as the data type, no matter if the result of the calculation fits within the type.

Ada already implements a set of rules for the conversion of types. These state that only smaller types shall be converted into bigger types, even though the compiler allows both directions. In the case of a bigger ranged type being converted into a smaller type, Ada can already predict that that value may not be within the value range and express a warning while continuing with the compilation. This is only possible if there's an explicit value written within the code that can be evaluated by the compiler. The same behavior was also visible in the overflow example, where a warning was raised when the value 1 was to be added to the variable but not if that value came from an external module. During both typecasting and evaluation of mathematical expressions, the program terminates as soon as a value outside the type of a variable has to be stored back into it, though this is only the case with Ada's signed integers. A *Modular* type would simply wrap around if the value gets larger than its limits.

Rust combines aspects of both C and Ada, leaving the developer some freedom but still being strict at the same time. The keyword `as` can be used for a completely unchecked conversion of types, like C, whereas the Trait `From` is only implemented in the correct "direction" of casting. Utilizing its `Option`-type, Rust also offers methods for various versions of arithmetic expressions, depending on what the developer wants to do in case the value exceeds the type limits. This gives the developer more freedom than for example Ada to safely do certain operations but at the same time provides them with options to handle any mistakes. For an unknowing developer, the only real downside of Rust in this case is the different behavior depending on the compilation configuration. This could lead to confusion and testing errors when running the same code in different modes like *Debug* or *Release*.

5.3 Memory and Storage

Description

Memory management is an essential point of every programming language as it has a direct impact on the abilities and determinability of said language. In this part, the typical management of a standard construct within programming is being shown, namely that of an array, a construct where multiple elements of the same type are saved within a structure and can be indexed to access it. Depending on how it is stored inside the memory, one approach for accessing something inside such an array could be counting the bytes within the memory, starting from the initial element at position zero. If no checks were to be performed, indexing anything further than the maximum length of the array could then access other variables and instances sitting within the memory.

The problem that is to be looked at is whether a program can access anything outside of the boundaries of a declared and defined array by simply trying to index one too far. This can easily happen in everyday programming, especially within loops for example.

C

C probably has the most straightforward approach to using arrays. Its memory management is just like explained in the introduction, all elements are next to each other in the memory. The pointer to the array itself is the address of the element at index 0, the position of the ones behind it are calculated by *address of element at index* n = *address of element at index* $0 + n *$ *sizeof(element type)*. [ISO18, p. 58]

The following code example creates an array and then iterates over the values, exceeding both the low and the high bounds:

```c
#include <stdlib.h>
#include <stdio.h>

int main()
{
    int other_number = 8;

    int array[5] = {1, 2, 3, 4, 5};

    for (int i = -1; i < 6; i++)
    {
        printf("array at index %d:
            %d\n", i, array[i]);
    }

    exit(0);
}
```

Listing 5.47: Accessing index outside of the defined array

Program output:

```
$ ./output
array at index -1: 8
array at index 0: 1
array at index 1: 2
array at index 2: 3
array at index 3: 4
array at index 4: 5
array at index 5: 32767
```

C does not check whether the accessed index is within the previously defined array. It simply follows the rules for how to return the value at a certain index and runs with it. This can be seen with the previously defined variable `other_number`, which can be seen in the memory right before the array at index -1. The value after the array changes every time the program is run, it's not directly used by this code snippet. This again follows C's philosophy of giving power to the developer who then has to deal with it. This is not only the case for arrays, as pointers can simply be edited and played around with, this example just highlights the freedoms and responsibilities of the developer pretty well.

Ada

Ada has, compared to a lot of other programming languages, a very unique approach to arrays. It is not automatically initialized with a range of 0 to n but rather by a specified index range which can be of any type or subtype, making this very similar to a dictionary in other programming languages [Duf+12, Ch. 3.6.1]. This concept makes a check if the index is valid inevitable in any case, as the type is not necessarily a numeric value that can be used to calculate a memory address like in C. What happens in case an invalid access is being attempted anyway can be seen in the following example:

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4     Number_Array: array (Integer range 1..5) of Integer := (1,2,3,4,5);
5 begin
6     for I in 1..6 loop
7         Put_Line("array at Index" & Integer'Image(I) & ":" & Integer'
              Image(Number_Array(I)));
8     end loop;
9 end Main;
```

Listing 5.48: Accessing index outside of the defined range

Program output:

```
$ ./obj/main
array at Index 1: 1
array at Index 2: 2
array at Index 3: 3
array at Index 4: 4
array at Index 5: 5

raised CONSTRAINT_ERROR : main.adb:7 index check failed
```

The raised error confirms the suspicion that every access needs to be checked by stating that the index check failed. This of course can have a negative effect on the runtime performance. On the other hand, a big advantage of the concept of types and especially subtypes being used for indexing arrays is that exactly those subtypes can then be used for an iteration later on.

With this in mind, the previous example could be re-written like this:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
    type Index_Range is range 1..5;
    Number_Array: array (Index_Range) of Integer := (1,2,3,4,5);
begin
    for I in Index_Range loop
        Put_Line("array at Index" & Index_Range'Image(I) & ":" & Integer
            'Image(Number_Array(I)));
    end loop;
end Main;
```

Listing 5.49: Using a subtype to define and then iterate over an array

Rust

The checks regarding arrays in Rust have to be done in two parts, checking the lower and upper boundaries. The reason for this is the different approach of the compiler, which can be observed in the following example:

```
fn main() {
    let array: [i32; 5] = [1, 2, 3, 4, 5];

    println!("array at index -1: {}", array[-1]);
}
```

Listing 5.50: Attempting to index an array at a negative position

Compilation output:

```
$ cargo build
   Compiling array_boundaries v0.1.0 (/home/philipp/workspace/Rust/
       Memory_Storage/array_boundaries)
error: negative integers cannot be used to index on a `[i32; 5]`
 --> src/main.rs:4:45
  |
4 |     println!("array at index -1: {}", array[-1]);
  |                                             ^^ cannot use a negative
      integer for indexing on `[i32; 5]`
  |
help: to access an element starting from the end of the `[i32; 5]`,
    compute the index
  |
4 |     println!("array at index -1: {}", array[array.len() -1]);
  |                                             ++++++++++++

error: could not compile `array_boundaries` due to previous error
```

The attempt to access an element in an array with the index being < 0 is being shut down fairly quickly by the compiler, as the index number must be of the type `usize`, which is

unsigned [The, Ch. 8.2.6]. If this is attempted to be calculated during runtime, which the compiler cannot detect, a type error like in 5.2.2 Conversion and Type Limits would be thrown. An upper boundary would be within the type limits of `usize`, which is why the following program compiles without any errors, even though the execution does not run without flaw:

```
fn main() {
    let array: [i32; 5] = [1, 2, 3, 4, 5];
    for i in 0..6 {
        println!("array at index {}: {}", i, array[i]);
    }
}
```

Listing 5.51: Attempting to exceed array boundaries

Program output:

```
$ ./target/debug/array_boundaries
array at index 0: 1
array at index 1: 2
array at index 2: 3
array at index 3: 4
array at index 4: 5
thread 'main' panicked at 'index out of bounds: the len is 5 but the
    index is 5', src/main.rs:4:46
note: run with `RUST_BACKTRACE=1` environment variable to display a
    backtrace
```

Trying to access a value outside of the array will lead to a panic of that thread, as expected [The, Ch. 8.2.6]. There are also no predefined access methods using a checked method like the various types offered for arithmetic operators. On the other hand, Rust offers, apart from using a range from zero to the array length, a `for ... in` loop to iterate over the elements of the array. This, together with immutable maximum ranges and the usage of constant values, offers various safe ways for an iteration. In case a specific index is wanted, this simply has to be checked manually or caught with a function like catch_unwind [Rusa].

Another option could be using a vector object `Vec<T>`, a growable array type, which offers access methods using Option types as a result, instead of panicking [Rusg].

Summary

Just like with the previous errors, C strictly follows what the programmer writes down. Doing this, there is no warning in case an element in an array that was not even part of it, is accessed. This could lead to a program accidentally corrupting other parts of a program unless the array size is manually checked each time. Ada's approach to arrays was more similar to how other programming languages implement a map or dictionary.

An advantage of this is that these ranges can easily be used for an iteration, possibly increasing the readability of bigger programs. Rust went with the same direction as C in regards to indexing with numbers but with a lot more restrictions. The one kind of possible out-of-bounds errors, which are any indexes lower than zero, is secured by the type for indexing being unsigned, thus leading to either the compiler detecting it, or to a type error at runtime. Any access that might exceed the index range by being larger will make the thread panic during the execution of the program. This could be prevented by a try-catch structure or by pre-empting this attempted access, either with a condition in each iteration, or using the size of the array for indexing.

5.4 Arithmetic Errors

Description

Using mathematical functions in a program always entails some risk, as only finite numbers are able to be stored within a program's memory. Infinite values have to be rounded at some point, possibly leading to inaccuracies. The error that will be highlighted in this section will be an example of programs trying to use values outside a function's domain. In mathematics, a domain D_f describes a set of values for which the current function $f : X \to Y$ is defined. Two well-known examples include the square root (considering that we do not work with imaginary numbers)

$$f : x \to \sqrt{x} \ with \ D_f = \mathbb{R}_0^+$$

and the common division.

$$f : x \to \frac{1}{x} \ with \ D_f = \mathbb{R}^{\neq 0}$$

To keep it as simple as possible and not have to use a math package for the square root, we will focus on the division.

Regarding arithmetic errors, one distinction can be made when looking at how a mathematical expression is being evaluated by the compiler.

Using binary numbers, an addition or subtraction can be realized by simple register operations on the assembler-level [Kor02, pp. sqq.13]. Using them, it is also possible to implement simple versions of multiplication and division of two numbers. The interesting question is now if the programming language in question adds checks before an arithmetic operation is attempted in case something illegal is being attempted. If not, the register will attempt to continue with the operation which could lead to unexpected results.

If checks like these are added, the program should crash during runtime to prevent the

assembler program from attempting to calculate it. To simulate this, each program will run from -5 to 5 in steps of 1 and print out the results. To make sure that the programming languages have a certain continuity, this calculation will be performed with different data types, namely floating-point and regular integer values.

C

The C program is pretty easy to understand. To ensure that the type *float* is being used in the first example, the division is performed with `1.0f` as the dividend:

```c
#include <stdio.h>

void divide_by_zero_float()
{

    for (int i = -5; i <= 5; i++)
    {
        float divisor = (float)i;
        float division_result = 1.0f / divisor;
        printf("i as float: %4.1f, result: %5.2f \n", divisor,
           division_result);
    }
}
```

Listing 5.52: Floating Point Division by zero in C

```c
#include <stdio.h>

void divide_by_zero_integer()
{

    for (int i = -5; i <= 5; i++)
    {
        int divisor = i;
        int division_result = 1 / divisor;
        printf("i as int: %2d, result: %2d \n", divisor, division_result
           );
    }
}
```

Listing 5.53: Integer Division by zero in C

The division using floating point values returns the following results:

```
$ ./output
i as float: -5.0, result: -0.20
i as float: -4.0, result: -0.25
i as float: -3.0, result: -0.33
i as float: -2.0, result: -0.50
i as float: -1.0, result: -1.00
i as float:  0.0, result:   inf
```

```
i as float:  1.0, result:  1.00
i as float:  2.0, result:  0.50
i as float:  3.0, result:  0.33
i as float:  4.0, result:  0.25
i as float:  5.0, result:  0.20
```

As one can see, the division by zero returns infinity, a constant which can be found in `math.h` [ISO18, p. 350]. This operation is following IEEE 754, the Standard for Floating-Point Arithmetic, which defines the result of a division by zero as ∞ [Inf, p. 49].

Using integers for the division returns quite a different result:

```
$ ./output
i as int: -5, result:  0
i as int: -4, result:  0
i as int: -3, result:  0
i as int: -2, result:  0
i as int: -1, result: -1
Floating point exception (core dumped)
```

This is expected and can be found in the C standard to be undefined behavior, resulting in the termination of the program [ISO18, §6.5.6].

From a technical standpoint, this is a fairly easy error to prevent by simply checking for the divisor being zero when using integers. But also when working with floating point numbers and then casting them, a developer should not be lulled into a false sense of security due to two major things. First off is the special case of $0/0$ with a result being Not a Number instead of infinity. Secondly, when trying to cast this or an infinite value to an integer type, no warning is being raised while the value is still converted. For 1- and 2-byte values, it is set to zero (e.g. 00000000_2), while 4- and 8-byte values display the minimum value (1, followed by $n-1$ zeroes).

In any case, this is very far away from a somehow close approximation for infinity for any of those data types and will inevitably result in miscalculations and inaccurate results if the normal calculations are just being continued. So even though the program will not crash when using floating-point types to calculate division results, it should still be checked for edge-cases like $x/0$ and also $0/0$ as continuing with those results may corrupt the program outcome.

Ada

The Ada implementation is very similar to C:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Div_Float is
    Dividend: Float := 1.0;
    Divisor: Float;
    Division_Result: Float;
begin

    for I in -5 .. 5 loop
        Divisor := Float(I);
        Division_Result := Dividend / Divisor;
        Put_Line ("1 / " & Float'Image (Divisor) & " = " & Float'Image (
           Division_Result));
    end loop;

end Div_Float;
```

Listing 5.54: Floating Point Division by zero in Ada

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Div_Int is
    Dividend: Integer := 1;
    Divisor: Integer;
    Division_Result: Integer;
begin

    for I in -5 .. 5 loop
        Divisor := I;
        Division_Result := Dividend / Divisor;
        Put_Line ("1 / " & Integer'Image (Divisor) & " = " & Integer'
           Image (Division_Result));
    end loop;

end Div_Int;
```

Listing 5.55: Integer Division by zero in Ada

The result is the same as in C, where the floating point division returns infinity while the integer division terminates due to a failed Division Check, as per 3.5.4 (20) and 11.5 (13/2) in the Ada Reference Manual [Duf+12, pp. 47, 275].

Division using Float numbers:

```
$ ./obj/output
1 / -5.00000E+00 = -2.00000E-01
1 / -4.00000E+00 = -2.50000E-01
1 / -3.00000E+00 = -3.33333E-01
1 / -2.00000E+00 = -5.00000E-01
1 / -1.00000E+00 = -1.00000E+00
1 /  0.00000E+00 = +Inf*******
1 /  1.00000E+00 =  1.00000E+00
1 /  2.00000E+00 =  5.00000E-01
1 /  3.00000E+00 =  3.33333E-01
1 /  4.00000E+00 =  2.50000E-01
1 /  5.00000E+00 =  2.00000E-01
```

Division using Integer numbers:

```
$ ./obj/output
1 / -5 =  0
1 / -4 =  0
1 / -3 =  0
1 / -2 =  0
1 / -1 = -1

raised CONSTRAINT_ERROR : div_int.
   adb:11 divide by zero
```

When trying to convert the floating point infinity to its respective value for the integer type, the program will also fail, raising a Constraint Error, this time from a failed Overflow Check [Duf+12, p. 275]. This is due to the value of infinity being bigger than $2^{15} - 1$, the maximum value for Integer [Duf+12, p. 47]. Conversion errors like these were already covered in 5.2.2 Conversion and Type Limits.

Just like in C, Ada will also have to check each division for the divisor being zero. Another option could be adding an exception handler wherever such an operation takes place. A function performing a safe division could also be implemented to avoid code repetition in case the entire program has more than a handful of such operations.

Rust

The implementation in Rust looks very straightforward as well, the only small point that might be confusing for the reader is the operator `..=` in the loop declaration, which indicates an inclusive range:

```
pub fn divide() {
    for i in -5..=5 {
        let divisor: f64 = i as f64;
        let division_result: f64 = 1.0 / divisor;
        println!("i as f64: {:5.2}, result: {:5.2}", divisor,
            division_result);
    }
}
```

Listing 5.56: Floating Point Division by zero in Rust

```
pub fn divide() {
    for i in -5..=5 {
        let divisor: i16 = i;
        let division_result: i16 = 1 / divisor;
        println!("i as i16: {:2}, result: {:2}", divisor,
            division_result);
    }
}
```

Listing 5.57: Integer Division by zero in Rust

Compiling this does not raise any warnings or errors, executing both programs is where a difference can be spotted. The division using a float value returns the following output:

```
$ ./target/debug/int_float_diff
i as f64: -5.00, result: -0.20
i as f64: -4.00, result: -0.25
i as f64: -3.00, result: -0.33
i as f64: -2.00, result: -0.50
i as f64: -1.00, result: -1.00
i as f64:  0.00, result:   inf
i as f64:  1.00, result:  1.00
i as f64:  2.00, result:  0.50
i as f64:  3.00, result:  0.33
i as f64:  4.00, result:  0.25
i as f64:  5.00, result:  0.20
```

Just like C and Ada, Rust follows IEEE 754 regarding the division by zero and returns infinity, an `f64` constant [F64]. Converting this to an integer variable returns the maximum value of the respective type, like 32767 for `i16` [The, Ch. 8.2.4].

Let's look at the same division but with `i16` values instead of `f64`:

```
$ ./target/debug/int_float_diff
i as i16: -5, result:  0
```

```
i as i16: -4, result:  0
i as i16: -3, result:  0
i as i16: -2, result:  0
i as i16: -1, result: -1
thread 'main' panicked at 'attempt to divide by zero', src/div_int.rs
    :4:36
note: run with `RUST_BACKTRACE=1` environment variable to display a
    backtrace
```

This time, the division fails as soon as the division by zero is attempted, just like in C and Ada. To prevent a crash like that but also stay away from manually checking every division for a divisor of zero, the trait `num::CheckedDiv` was implemented which returns a `None`-value in that case [Che]. This is the one of the traits that were already mentioned in the end of 5.2.2 Conversion and Type Limits when type overflows were covered.

The infinity value in Rust is, fortunately, very deterministic. When converting it to any other type, it is set to the maximum value of said type, although one has to always be aware of when the conversion takes place. If additional calculations like adding a value x to our result are performed after the division, a developer might want his program to follow the rules of $\infty + x = \infty$. Using the `f64` constant, this is implemented exactly as so and will return correct results. As soon as the type cast to an integer value is completed though, the now cast value will represent the maximum of that type, which is a regular number. Trying to add a value $x > 0$ to it will result in an overflow like talked about in 5.2.2 Conversion and Type Limits.

Summary

What any developer calculating with numbers that might ever become zero has to watch out for are conversion errors. One might think that dividing using integers or floats might be the same if the result is cast accordingly afterwards. This could get problematic after looking at the non-deterministic behavior of the division-operator, depending on the datatype (integer or floating-point). An example for that might be a Rust developer calculating in float values, converting to integer afterwards. Having the division by zero in mind, they secure the function with a Result value. In the edge-case of dividing by zero, instead of an error that would have been caught and could have for example been skipped, the method now returns the maximum value for the datatype that infinity was converted to.

The following code snippet highlights the difference between the division using an `f64` type and then casting that to `i16` and using the integer division directly:

```
pub fn divide() {
    for i in -3..=3 {
        let divisor_float: f64 = i as f64;
        let division_result_float: f64 = 2.0 / divisor_float;
        let division_result_casted = division_result_float as i16;
        let i16_div_result = (2 as i16).checked_div(i).ok_or_else(|| "
            Div by zero");
        println!(
            "Div: 2/{:2}, f64 result: {.2}, cast to i16: {:2}, i16
                direct result: {:?}",
            divisor_float, division_result_float, division_result_casted
                , i16_div_result
        );
    }
}
```

Listing 5.58: Comparison of `f64`- to `i16`-division

Program output:

```
$ ./target/debug/int_float_diff
Div: 2/-3, f64 result: -0.67, cast to i16:  0, i16 direct result: Ok(0)
Div: 2/-2, f64 result: -1.00, cast to i16: -1, i16 direct result: Ok(-1)
Div: 2/-1, f64 result: -2.00, cast to i16: -2, i16 direct result: Ok(-2)
Div: 2/ 0, f64 result: inf, cast to i16: 32767, i16 direct result: Err("
    Div by zero")
Div: 2/ 1, f64 result:  2.00, cast to i16:  2, i16 direct result: Ok(2)
Div: 2/ 2, f64 result:  1.00, cast to i16:  1, i16 direct result: Ok(1)
Div: 2/ 3, f64 result:  0.67, cast to i16:  0, i16 direct result: Ok(0)
```

Continuing to calculate with 32767 could now result in an overflow error as covered in 5.2.2 Conversion and Type Limits.

To sum it up, Arithmetic Errors are not really possible to catch during the compilation but have to be watched out for when the calculation is performed. Hereby, the three languages offer different levels of support. While preventing the error in C would only really work by checking whether the calculation could end up being invalid by verifying that all values are within the function domain, Ada goes further and is able to catch the specific Constraint Error if an operation during or after the calculation were to be invalid. Rust is probably the most comfortable of all three languages to deal with. By being deterministic with its constants, even continuing the calculation by converting infinity can be a valid approach. In case an invalid operation, like the division by zero using Integer types, is being attempted, it can be secured even further by features like `ok_or_else`.

5.5 Concurrency

Description

When working with programs on a processor with multiple cores and threads, using their full capabilities is often up to the developer and cannot be done automatically. Apart from the advantages of speeding up the program's execution, concurrency can also bring a lot of dangers to program reliability and expectability as schedulers and memory access calls are not deterministic and may perform differently each time a program is executed. This also includes common resources which have to be shared following some kind of ruleset to keep up the program's integrity. An example of that can be seen in the following example:

Thread 1	Thread 2
Read value = 2	
Increase value by one	
Write back value = 3	
	Read value = 3
	Increase value by one
	Write back value = 4

(a) Each thread increases the value by one, in sequence

Thread 1	Thread 2
Read value = 2	
Increase value by one	Read value = 2
Write back value = 3	Increase value by one
	Write back value = 3

(b) Each thread increases the value by one, concurrent

Figure 5.1: Concurrency example

The problem occurs as the value is not locked while one thread is using it. This can be secured against by using mechanisms like semaphores and mutual exclusion which will also be briefly presented, together with an example covering the question if the language theoretically allows such a behavior.

The following problem will try to provoke a situation where a program does not perform as expected due to an insecure resource. A bank account has a balance of 100€. There are now three threads spawned that both want to buy something for 70€. They will do this by first checking if the available balance is over those 70€ after which the transaction will be taking place. Afterwards, the balance will be reduced by the cost of whichever object was bought and the thread finishes.

In case the variable which represents the account balance is not secured properly, all three threads might check it and perform the transaction, even though only one of them should be allowed to do so.

C

Implementing a simple prototype in C is pretty straightforward. The balance is defined as a normal local variable in the main function and passed to the function used in the threads when they are created using `pthread_create`:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void *threadFunction(void *args)
{
    int *balance = (int *)args;

    int currentBalance = *balance;

    if (currentBalance < 70)
    {
        printf("Insufficient balance: %d\n", currentBalance);
        return NULL;
    }

    printf("Balance before starting transaction: %d\n", currentBalance);
    sleep(5); // "transaction time"
    *balance = currentBalance - 70;
    printf("Transaction successful, balance after transaction: %d\n", *
       balance);
}

int main()
{
    int balance = 100;
    int number_of_threads = 3;
    pthread_t thread_id[number_of_threads];

    printf("Initial balance: %d\n\n", balance);

    for (int i = 0; i < number_of_threads; i++)
    {
        pthread_create(&thread_id[i], NULL, threadFunction, &balance);
        sleep(1);
    }

    for (int i = 0; i < number_of_threads; i++)
    {
        pthread_join(thread_id[i], NULL);
    }

    printf("\nFinal balance: %d\n", balance);

    exit(0);
}
```

Listing 5.59: Unsafe concurrency in C

For the compilation, `-pthread` has to be added to the usual `gcc -main.c -o output` to be able to use said module. The execution then has the following output:

```
$ ./output
Initial balance: 100

Balance before starting transaction: 100
Balance before starting transaction: 100
Balance before starting transaction: 100
Transaction successful, balance after transaction: 30
Transaction successful, balance after transaction: 30
Transaction successful, balance after transaction: 30

Final balance: 30
```

This is even more problematic than the bank balance simply going below zero. By saving the current balance in its own variable which is then used to write back to the actual balance, the three transactions seem to go through but the final balance is still 30, which does not represent the actual value that would be deterministic (-110), similar to Figure 5.1. To prevent this, the variable should have been locked in some way to prevent access to the balance while the current thread is working with it.

The accessing problem could be fixed by using a mutex, for example the one provided by the UNIX pthread. By adding a global mutex variable that is locked when entering and unlocked when leaving the function, the entire function body of editing the variable would be secured, as only one thread at a time is able to access it [Pth]. The fixed `threadFunction` could look like the following:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <pthread.h>
5
6  pthread_mutex_t account_mutex = PTHREAD_MUTEX_INITIALIZER;
7
8  void *threadFunction(void *args)
9  {
10     pthread_mutex_lock(&account_mutex);
11
12     int *balance = (int *)args;
13
14     int currentBalance = *balance;
15
16     if (currentBalance < 70)
17     {
18         printf("Insufficient balance: %d\n", currentBalance);
19         pthread_mutex_unlock(&account_mutex);
20         return NULL;
21     }
22
23     printf("Balance before starting transaction: %d\n", currentBalance);
24     sleep(5); // "transaction time"
25     *balance = currentBalance - 70;
```

```
26      printf("Transaction successful, balance after transaction: %d\n", *
            balance);
27      pthread_mutex_unlock(&account_mutex);
28  }
```

Listing 5.60: Securing the program with a mutex

Program output:

```
1  $ ./output
2  Initial balance: 100
3
4  Balance before starting transaction: 100
5  Transaction successful, balance after transaction: 30
6  Insufficient balance: 30
7  Insufficient balance: 30
8
9  Final balance: 30
```

This is the result that was originally intended, as the balance is only accessed and edited by one thread at a time. The other two wait until its their turn after which the value is already too low to pass the condition, thus not getting it below zero.

This does not prevent the developer from being able to make the mistake due to oversight though. The mutex variable only helps with but does not fully solve the problem of an incorrect value stored in `currentBalance` being used. The access to the variable, which happens in line 14, must take place within the safe space that only one thread may enter at a time. If that's not the case, it will have the same result as seen in the prior example (Listing 5.59).

Ada

Concurrency in Ada works by executing one or more `task` instances which then run independently. In our case, the declaration body of `Main` includes the procedure `Subtract_70` which is then executed within the task body (Listing 5.61, lines 23-26). Just like in C, a `delay`, which is Ada's equivalent to a sleep-function, represents the "transaction" that needs to take place. The task is then declared and defined using **task type** and **task body**. In the execution part of the **Main** procedure, the three threads are being declared in their own declaration-block. This could have been done in the Main declaration as well but then the final output message would not have waited for the threads to finish. By wrapping them inside their own block (lines 30-34), this will be executed and waited for to finish before continuing with the execution:

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4
5      Balance: Integer := 100;
6
7      procedure Subtract_70 is
8          CurrentBalance: Integer;
9      begin
10         CurrentBalance := Balance;
11         if CurrentBalance >= 70 then
12             Put_Line("Balance before transaction: " & Integer'Image(
                   CurrentBalance));
13             delay 5.0;
14             Balance := CurrentBalance - 70;
15             Put_Line("Transaction successful, balance after transaction:
                    " & Integer'Image(Balance));
16         else
17             Put_Line("Insufficient balance: " & Integer'Image(
                   CurrentBalance));
18         end if;
19     end Subtract_70;
20
21     task type Transaction;
22
23     task body Transaction is
24     begin
25         Subtract_70;
26     end Transaction;
27
28 begin
29
30     declare
31         T1, T2, T3: Transaction;
32     begin
33         null;
34     end;
35
36     Put_Line("Final balance: " & Integer'Image(Balance));
```

```
37
38 end Main;
```

Listing 5.61: Unsafe concurrency in Ada

The output ends up being essentially the same as in C:

```
$ ./obj/output
Balance before transaction:  100
Balance before transaction:  100
Balance before transaction:  100
Transaction successful, balance after transaction:  30
Transaction successful, balance after transaction:  30
Transaction successful, balance after transaction:  30
Final balance:  30
```

This means that Ada also is not actively forcing its developer to protect its variable from concurrent and simultaneous access. There are mechanics in place, like `protected` units which can coordinate access to shared data [Duf+12, p. 215] but they are not enforced.

An example for this could be a `protected type Mutex` which is added to the main unit. By using a condition, the `entry_barrier`, anything that does not fulfill the condition is forced to wait for it [Duf+12, Ch. 9.5.2]. As soon as the barrier is lifted by unlocking this variable, the next task in the queue is able to take it, effectively implementing a Mutex variable just like C:

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4
5      protected type Mutex is
6          entry Lock;
7          procedure Unlock;
8      private
9          Locked : Boolean := False;
10     end Mutex;
11
12     protected body Mutex is
13         entry Lock when not Locked is
14         begin
15             Locked := True;
16         end Lock;
17         procedure Unlock is
18         begin
19             Locked := False;
20         end Unlock;
21     end Mutex;
22
23     Balance: Integer := 100;
24     Mut: Mutex;
25
26     procedure Subtract_70 is
27         CurrentBalance: Integer;
```

```
28      begin
29          Mut.Lock;
30          CurrentBalance := Balance;
31          if CurrentBalance >= 70 then
32              Put_Line("Balance before transaction: " & Integer'Image(
                    CurrentBalance));
33              delay 5.0;
34              Balance := CurrentBalance - 70;
35              Put_Line("Transaction successful, balance after transaction:
                     " & Integer'Image(Balance));
36          else
37              Put_Line("Insufficient balance: " & Integer'Image(
                    CurrentBalance));
38          end if;
39          Mut.Unlock;
40      end Subtract_70;
41
42      task type Transaction;
43
44      task body Transaction is
45      begin
46          Subtract_70;
47      end Transaction;
48
49  begin
50
51      declare
52          T1, T2, T3: Transaction;
53      begin
54          null;
55      end;
56
57      Put_Line("Final balance: " & Integer'Image(Balance));
58
59  end Main;
```

Listing 5.62: Concurrency in Ada using a protected type

Program output:

```
$ ./obj/output
Balance before transaction:  100
Transaction successful, balance after transaction:  30
Insufficient balance:  30
Insufficient balance:  30
Final balance:  30
```

Just like C, the place where the locking process takes place, is important. If `CurrentBalance` was to be initialized with `Balance` before the variable is locked, it would again have the same output as in Listing 5.61.

Rust

Looking at this question in Rust could get problematic, simply because of the way that memory is managed, as it was introduced in chapter 4.3. It is going to be attempted anyway, using the `thread::spawn` function:

```rust
use std::thread;
use std::time::Duration;

fn main() {

    let mut balance: i32 = 100;

    let handle = thread::spawn(|| {
        let current_balance: i32 = balance;
        if current_balance > 70 {
            thread::sleep(Duration::from_secs_f32(5.0));
            balance = current_balance - 70;
        }
    });

    handle.join().unwrap();
}
```

Listing 5.63: Concurrency attempt in Rust

Compilation Output:

```
$ cargo build
   Compiling balance v0.1.0 (/home/philipp/workspace/Rust/Multithreading
      /balance)
error[E0373]: closure may outlive the current function, but it borrows `
   balance`, which is owned by the current function
  --> src/main.rs:8:32
   |
8  |     let handle = thread::spawn(|| {
   |                                ^^ may outlive borrowed value `
      balance`
...
12 |             balance -= 70;
   |             ------- `balance` is borrowed here
   |
note: function requires argument type to outlive `'static`
  --> src/main.rs:8:18
   |
8  |       let handle = thread::spawn(|| {
   |  __________________^
9  | |         let current_balance: i32 = balance;
10 | |         if current_balance > 70 {
11 | |             thread::sleep(Duration::from_secs_f32(5.0));
12 | |             balance -= 70;
13 | |         }
14 | |     });
   | |______^
```

```
help: to force the closure to take ownership of `balance` (and any other
    referenced variables), use the `move` keyword
   |
8  |     let handle = thread::spawn(move || {
   |                                ++++

For more information about this error, try `rustc --explain E0373`.
error: could not compile `balance` due to previous error
```

Already when trying to spawn a single thread, the compilation fails. The reason that is given is that the value is simply borrowed, there is nothing preventing the main thread from manipulating or even dropping the value, thus possibly corrupting the execution of the thread. A solution attempt is also provided, suggesting to move the value inside of the function. This works and compiles totally fine, also with multiple threads:

```
1  use std::thread;
2  use std::time::Duration;
3
4  fn main() {
5      let mut handles: Vec<std::thread::JoinHandle<()>> = Vec::new();
6
7      let num_of_threads = 3;
8
9      let mut balance: i32 = 100;
10
11     for _ in 0..num_of_threads {
12         let h = thread::spawn(move || {
13             let current_balance: i32 = balance;
14             if current_balance > 70 {
15                 println!("Balance before transaction: {}",
                       current_balance);
16                 thread::sleep(Duration::from_secs_f32(5.0));
17                 balance = current_balance - 70;
18                 println!("Transaction successful, balance after
                       transaction: {}", balance);
19             } else {
20                 println!("Insufficient balance: {}", current_balance)
21             }
22         });
23         handles.push(h);
24     }
25
26     for handle in handles {
27         handle.join().unwrap();
28     }
29
30     println!("Final balance: {}", balance);
31 }
```

Listing 5.64: Rust concurrency when moving a variable inside a thread

The only issue is that in line 17, it's not the actual variable `balance` that is being edited, but only the moved copy with its ownership now belonging to that thread. This is why

the “fixed” program produces the following output:

```
$ ./target/debug/balance
Balance before transaction: 100
Balance before transaction: 100
Balance before transaction: 100
Transaction successful, balance after transaction: 30
Transaction successful, balance after transaction: 30
Transaction successful, balance after transaction: 30
Final balance: 100
```

To fix this example using mutual exclusion, the standard way provided by Rust makes use of the structs `Mutex` and `Arc`, both included in `std::sync`. Hereby, the memory management is handled by an object of type Arc which is passed into the thread instead of a direct reference to the shared object, while the excluding access to the variable is handled by a Mutex object: [Ruse; Rusf]

```
1  use std::sync::{Arc, Mutex};
2  use std::thread;
3  use std::time::Duration;
4
5  fn main() {
6      let mut handles: Vec<std::thread::JoinHandle<()>> = Vec::new();
7
8      let num_of_threads = 3;
9
10     let outside_balance = 100;
11
12     let lock = Mutex::new(outside_balance);
13
14     let arc = Arc::new(lock);
15
16     for _ in 0..num_of_threads {
17         let cloned_arc = Arc::clone(&arc);
18
19         let h = thread::spawn(move || {
20             let mut balance = cloned_arc.lock().unwrap();
21             let current_balance:i32 = *balance;
22
23             if current_balance > 70 {
24                 println!("Balance before transaction: {}",
                       current_balance);
25                 thread::sleep(Duration::from_secs_f32(5.0));
26                 *balance = current_balance - 70;
27                 println!("Balance after transaction: {}", balance);
28             } else {
29                 println!("Insufficient balance: {}", current_balance)
30             }
31         });
32         handles.push(h);
33     }
34
35     for handle in handles {
36         handle.join().unwrap();
37     }
```

```
38
39      println!("Final balance: {}", *arc.lock().unwrap());
40  }
```

Listing 5.65: Rust concurrency with Mutex and Arc

As it is forbidden to use the same Arc object in each thread (see Listing 5.63), it is cloned in each iteration and used by the spawned thread. The original as well as the cloned Arc objects all point to one memory location, thus allowing to edit the same variable instead of a copied value like in the previous example (Listing 5.64). The excluding capabilities of the Mutex object can then be used by using `lock()`, while the containing value can be accessed by an additional `unwrap()` operation. This locks the access until the variable that it was assigned to either goes out of scope or is manually unlocked.

Using this, the program output is as followed:

```
./target/debug/balance_mut
Balance before transaction: 100
Transaction successful, balance after transaction: 30
Insufficient balance: 30
Insufficient balance: 30
Final balance: 30
```

Looking back at the final remarks of C and Ada, it is notable that the mistake of reading the value of `balance` before applying the lock is not possible in Rust, as the Mutex is directly applied and tied to the variable. It is not completely impossible though, as the lock could manually be opened after assigning the variable `current_balance`, which does go against the logic of a mutual exclusion though.

Summary

Concurrency is always big risk for program integrity, especially because of consequences and possible unwanted behavior regarding locks and race conditions. While the latter is only really enforced in Rust, using its very strict memory management system, all three languages offer good ways of working with multiple threads and shared resources, even though some are a bit more tedious than others. In C and Rust for example, a mutual exclusion can quite easily be realized using predefined structures and methods, namely `pthread` and `Mutex` / `Arc`, whereas in Ada, a whole protected type has to be created by the developer themselves.

6 Summary

The past chapter talked about how C, Ada, and Rust deal with different kinds of programming errors. This section will now serve as a small summary of arguments for and against each of the three languages, ended by a final estimation as to how and why this language could be helpful for companies regarding safety-critical projects.

The following table summarizes the errors that were dealt with:

Topic	C	Ada	Rust
Outside of Scope	∼	✓	✓
Ambiguity	✓	✓	✓
Assigning wrong type	✗	✓	✓
Type conversion	✗	∼	∼
Array boundaries	✗	∼	∼
Arithmetic	∼	∼	∼
Race conditions	∼	∼	✓

Table 6.1: Summary of potential error sources and the programming languages' reactions to them

Legend:

✓: This error does not cause any unintended behavior during runtime.

∼ : The programming language provides special procedures or rules to safely implement this feature, but does not enforce them (this does not include simple if-clauses). This symbol is also used in case an error is thrown as that can easily be caught.

✗: This error can be provoked quite easily and will lead to unexpected behavior by the program.

There are, of course, certain official standards and rulesets that have to be applied to any project, depending on the industry. This is also going to be briefly talked about in each section, how much restriction each language would need to be applicable for a safety-critical project.

6.1 C

One of the biggest advantages of C in general is the freedom that a software developer gets when writing a program. As long as there's a very experienced and knowledgeable person sitting in front of the code editor, it can be a very powerful tool with the ability to do more than other programming languages, simply due to the close connection with the machine. This at the same time is also C's biggest disadvantage, as this can very quickly lead to unintended behavior when dealing with topics like pointers. This is reflected in the summary (Figure 6.1), where C is only really able to catch an attempt at provoking ambiguity. The first ∼-mark in this case mean that for the scope, the compilation has to be studied and used correctly. The second one, regarding arithmetic errors, could also be a checkmark. Due to the inconsistency when converting certain constant values like Infinity, the entire topic is marked only semi-good though.

Looking at the rest of the results, the philosophy of freedom for the developer has been reflected very well in the past chapter. A good example for this is the conversion of values, which trusted the developer to know what they're doing, interpreting the value as it is written in the memory cell. It's simply doing what it's told to do, without checking whether that would corrupt the program.

In case C is supposed to be used in a corporate environment, a lot of precautions in the form of standards or processes have to be taken. An example here could be MISRA C, the "de facto standard for developing software in C where safety, security and code quality are important", according to their website. [Mis] In general, this can be a good idea, getting the entire power of the C language while still being suitable for safety-critical systems. Another advantage can be the reputation of such frameworks being very high due to a long history in the industry, giving the product a certain code quality level off the bat in case it's used. On the other hand, working with a tight framework inevitably means the restriction of the programmer. The learning hurdle of getting used to the strict framework which new employees would need to climb is of course never as big as learning a language completely new but also means that an experienced C developer may not be able to start to their fullest extent right away.

For this thesis, C was only chosen as a comparison for how a language that is not focused on safety-critical applications reacts to certain errors. The main focus lies on Ada and whether Rust could be a valid alternative in a corporate environment. This is why C is not going to be considered any further when talking about applicability for companies.

6.2 Ada

Looking back at the previous chapter, Ada performed very well and caught nearly every error either at compile time or terminated during runtime. Even though it's usually better to get notified of these before a project is ran, throwing errors also offers a way of dealing with them by catching them and acting depending on the situation. This was visible when the Type errors and Array boundaries were talked about, as the program immediately terminated as soon as something illegal was attempted. In both cases, just like C, Ada trusted the developer to know what they're doing. The only difference is that when the value would not fit into the type constraints, if array index or value range of a type, the program terminated with an error instead of running with corrupted values.

Developing in Ada has the big advantage of the giant reputation of usage in countless critical systems. It's of course not perfect, as the comparison in this thesis and examples like the Ariane 5 crash in 1996 showed [JM97] but the additional effort around the basic language when writing safe software is a lot less compared to C. Apart from the basic compiler that already checks a lot of potential error sources, certain tools like the GCC Ada Compiler can even be expanded using flags and Pragmas to improve the development even more, like the `-gnatwa` flag that activates more warnings [Gnab, Ch. 4.3.3]. Due to the already strict compiler and the general design of Ada, any standards or guidelines do not affect and restrict the every-day development as much as one would in C.

Probably the biggest disadvantage of Ada is the scarcity of developers on the job market. Due to the very niche application area, there is no real ambition for universities to teach it, simply because other languages are far more relevant for the students both as a basic education and on the job market later on. Apart from the aerospace and military industry and other critical application areas like traffic control systems or power plants, there are not many areas where Ada is relevant [Fel14]. Especially in private projects, factors like widespread usage, ease-of-use, and speed are more important than long-term security and safety. This leads to a snowball effect where companies that do not necessarily need to use it will use a more popular language, not really helping to develop a demand for Ada developers.

6.3 Rust

The language with the best responses to the possible errors that were looked at in the previous chapter was Rust. The only factor where both Ada and Rust have the same reaction was the indexing of arrays. The other error sources where the compiler did not interfere with a piece of code were type conversion and arithmetic errors, even though these were more deterministic, like the infinity constant. Furthermore, Rust offers methods and

functions for a safe implementation which can extend the general error handling that Ada has even more, using `Option` and `Result` types.

The latter is also, together with memory safety, probably the biggest advantage of Rust. The provided ways of easily dealing with possible panics of a program make it easy for a new developer to get into safe code. In the beginning, it might be a bit overwhelming, since there are a lot of basic things that have to be paid attention to, but once the first steep ascent of the learning curve is left behind, it's fairly simple to get better and better. This is only encouraged by the infrastructure that Rust provides, some examples include the two books *The Rust Programming Language* and *Rust by Example*, or the usually very extensive output in case the compilation fails. Using the command `rustc --explain`, every error can be explained by showing an example in the command shell. The outputs are always very verbose, making it easier for the developer to find where and why the compilation failed. Additionally, the online community surrounding Rust has only steadily been growing over the past years, which is reflected in both the state of the libraries written for Rust [Stac], as well as in the annual Stack Overflow developer surveys, which voted Rust as the most loved language for its 6th year in a row in 2021 [Staa].

The biggest argument against using Rust in an industrial environment is probably the fact that it does not have anything comparable to a standardization. Due to the language being open-source and only reviewed instead of completely updated by a central instance, any company that has to adhere to certain guidelines would rather stick to other programming languages that have an official standardization, taking their guarantee that the language performs in a certain way. Taking it on themselves to prove that Rust does what it's supposed to do or risk consequences in case any problems arise can be a big obstacle in the way of anybody who has to deliver code according to some guideline.

Due to the promising future of Rust, there are of course other entities in the form of independent agencies or companies like Ferrous Systems GmbH that realized the potential of Rust and aim to offer toolchains that adhere to important industrial standards, as early as the end of 2022. [Mun19; Fer] In February of 2022, this company even announced a partnership together with AdaCore to support the development as both realized the common business and technical background and knowledge [OG22].

7 Conclusion

The central question that started the study of the errors within all three programming languages was whether Rust can be a good alternative to Ada in safety-critical projects.

From a technical standpoint, Rust definitely matches Ada, if not exceeds it in certain areas. A big plus here was the extended capabilities when dealing with errors, as well as the memory safety. In terms of the programming environment, Rust also has an advantage, as the community interest is a lot higher compared to Ada.

The biggest arguments in favor of Ada are both its good reputation due to the long history of the language as well as the full certifications and background with the United States Department of Defense. This makes it easy to decide in favor of continuing to use it when it has worked well in recent years. Switching to another programming language involves laborious certification processes and supporting tasks such as selecting tools for static code analysis or continuous integration. Therefore, it may be easier to take the safe route with an established, proven language than to change the entire development process and environment.

If a company does not have to adhere to certain standards, Rust can be a good choice for new projects. Apart from the fact that there is a larger selection of developers to hire from, it can also be helpful for the company's good image, presenting itself as modern and up-to-date. But also for corporations who do rely on these certifications, it can be worth looking into Rust and doing some basic research for it in case a standardization is reached at some point in the future.

Listings

List of Figures

Acronyms

ANSI American National Standards Institute

ARG Ada Rapporteur Group

CVSS Common Vulnerability Scoring System

CWE Common Weakness Enumeration

DoD United States Department of Defense

EASA European Aviation Safety Agency

IDE Integrated Development Environment

ISO International Organization for Standardization

NVD National Vulnerability Database

Bibliography

ISO Standards

[ISO01] ISO/IEC. *ISO/IEC 8652:1995/COR 1:2001 Information technology — Programming languages — Ada — Technical Corrigendum 1*. Tech. rep. International Organization for Standardization, June 2001.

[ISO03] ISO/IEC. *ISO/IEC 8859-7 Information technology — 8-bit single-byte coded graphic character sets*. Tech. rep. International Organization for Standardization, Oct. 2003.

[ISO11] ISO/IEC. *ISO/IEC 9899:2011: Information technology - Programming languages - C*. 2011. URL: `https://www.iso.org/standard/57853.html` (visited on 12/09/2021).

[ISO12] ISO/IEC. *ISO/IEC 8652:2012 Information technology — Programming languages — Ada*. Tech. rep. International Organization for Standardization, Dec. 2012.

[ISO18] ISO/IEC. *ISO/IEC 9899:2018(E) Programming languages - C*. Tech. rep. International Organization for Standardization, July 2018.

[ISO67] ISO/IEC. *ISO 8652:1987 Programming languages - Ada*. Tech. rep. International Organization for Standardization, June 1967.

[ISO99] ISO/IEC. *ISO/IEC 9899:1999: Programming languages - C*. 1999. URL: `https://www.iso.org/standard/29237.html` (visited on 12/08/2021).

Other Standards

[Eas] *Easy Access Rules for Acceptable Means of Compliance for Airworthiness of Products, Parts and Appliances (AMC-20)*. Mar. 2021. URL: `https://www.easa.europa.eu/document-library/easy-access-rules/online-publications/easy-access-rules-acceptable-means` (visited on 01/24/2022).

[Inf] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019). DOI: `10.1109/IEEESTD.2019.8766229`.

Articles

[Adaa] *AdaCore Case Study - BAE Systems Eurofighter Typhoon*. Oct. 2011. URL: https://www.adacore.com/uploads/customers/CaseStudy_Eurofighter.pdf (visited on 11/12/2021).

[Adab] "ARTiSAN - Software Development on Meteor missile program". In: *Ada User Journal* 26.3 (Sept. 2005), 158–159. ISSN: 1381-6551.

[DC92] Information Management and Technology Division and Ralph V. Carlone. *Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia*. United States General Accounting Office, Feb. 1992. URL: https://www.gao.gov/assets/imtec-92-26.pdf (visited on 11/10/2021).

[Fel14] Michael B Feldman. *Who's using Ada? Real-World Projects Powered by the Ada Programming Language*. Nov. 2014. URL: https://www2.seas.gwu.edu/~mfeldman/ada-project-summary.html (visited on 12/21/2021).

[Fer] *Ferrocene*. URL: https://ferrous-systems.com/ferrocene/ (visited on 02/01/2022).

[JM97] J.-M. Jazequel and B. Meyer. "Design by contract: the lessons of Ariane". In: *Computer* 30.1 (1997), pp. 129–130. DOI: 10.1109/2.562936.

[Kam21] Kathy Kam. *Facebook Joins the Rust Foundation*. Apr. 2021. URL: https://developers.facebook.com/blog/post/2021/04/29/facebook-joins-rust-foundation/ (visited on 01/09/2022).

[Mis] *MISRA*. URL: https://www.misra.org.uk (visited on 01/24/2022).

[Mun19] James Munns. *Ferrocene: Part 1 - The Pitch*. June 2019. URL: https://ferrous-systems.com/blog/sealed-rust-the-pitch/ (visited on 02/01/2022).

[OG22] Quentin Ochem and Florian Gilcher. *AdaCore and Ferrous Systems Joining Forces to Support Rust*. Feb. 2022. URL: https://blog.adacore.com/adacore-and-ferrous-systems-joining-forces-to-support-rust (visited on 02/03/2022).

[Rit93] Dennis M. Ritchie. "The Development of the C Language". In: *SIGPLAN Not.* 28.3 (Mar. 1993), 201–208. ISSN: 0362-1340. DOI: 10.1145/155360.155580.

[Staa] *Stack Overflow Developer Survey 2021*. Aug. 2021. URL: https://insights.stackoverflow.com/survey/2021 (visited on 12/23/2021).

[Stab] *Stack Overflow Developer Surveys*. URL: https://insights.stackoverflow.com/survey (visited on 12/23/2021).

[Stac] *State of the rust/cargo crates ecosystem*. URL: https://lib.rs/ (visited on 01/15/2022).

[Tea20] The Rust Core Team. *Laying the foundation for Rust's future*. Aug. 2020. URL: https://blog.rust-lang.org/2020/08/18/laying-the-foundation-for-rusts-future.html (visited on 01/09/2022).

[Wil21] Ashley Williams. *Hello World!* Feb. 2021. URL: https://foundation.rust-lang.org/posts/2021-02-08-hello-world/ (visited on 01/09/2022).

Official References

[ANS83] ANSI. *ANSI/MIL-STD-1815A-1983, MILITARY STANDARD - Ada Programming Language*. Tech. rep. American National Standards Institute, Inc, Jan. 1983.

[Bar14] John Barnes. *Programming in ADA 2012*. 6th printing 2017. Cambridge University Press, 2014. ISBN: 978-1-107-42481-4.

[Duf+12] Robert A. Duff et al. *Ada 2012 Reference Manual. Language and Standard Libraries, International Standard ISO/IEC 8652/2012 (E)*. Springer, 2012.

[Gnaa] *GNAT Reference Manual*. URL: https://docs.adacore.com/gnat_rm-docs/html/gnat_rm/gnat_rm.html (visited on 01/26/2022).

[Gnab] *GNAT User's Guide for Native Platforms*. URL: https://gcc.gnu.org/onlinedocs/gnat_ugn/index.html (visited on 01/26/2022).

[Gnac] *GNATcheck*. URL: https://www.adacore.com/gnatpro/toolsuite/gnatcheck (visited on 01/26/2022).

[Gnad] *GNATcoverage*. URL: https://www.adacore.com/gnatcoverage (visited on 01/26/2022).

[Gpr] *GPRbuild and GPR Companion Tools User's Guide*. URL: https://docs.adacore.com/gprbuild-docs/html/gprbuild_ug.html (visited on 01/26/2022).

[KN18] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. USA: No Starch Press, 2018. ISBN: 978-1-59327-828-1.

[Rusc] *Rust by Example*. URL: https://doc.rust-lang.org/rust-by-example/index.html (visited on 02/04/2022).

[Rush] *The Edition Guide*. URL: https://doc.rust-lang.org/edition-guide/ (visited on 01/10/2022).

[Rusi] *The Rust RFC Book*. URL: https://rust-lang.github.io/rfcs/ (visited on 01/23/2022).

[Sta03] Richard M. Stallman. *Using the GNU Compiler Collection*. Oct. 2003. URL: https://gcc.gnu.org/onlinedocs/gcc-9.2.0/gcc.pdf (visited on 01/18/2022).

[SW] Richard M. Stallman and Zachary Weinberg. *The C Preprocessor*. URL: https://gcc.gnu.org/onlinedocs/cpp.pdf (visited on 01/15/2022).

[The] *The Rust Reference*. URL: https://doc.rust-lang.org/1.57.0/reference/introduction.html (visited on 02/04/2022).

Online References

[Che] *Trait num::CheckedDiv*. URL: https://docs.rs/num/0.4.0/num/trait.CheckedDiv.html (visited on 02/03/2022).

[Coe] *Rfcs/0401-coercions.md at master · Rust-Lang/rfcs*. URL: https://github.com/rust-lang/rfcs/blob/master/text/0401-coercions.md (visited on 02/02/2022).

[F64] *Primitive Type f64 1.0.0*. URL: https://doc.rust-lang.org/std/primitive.f64.html#associatedconstant.INFINITY (visited on 02/03/2022).

[Pth] *UNIX Specification - pthread mutex lock*. 1997. URL: https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_mutex_lock.html (visited on 01/27/2022).

[Rusa] *catch_unwind in std::panic*. URL: https://doc.rust-lang.org/std/panic/fn.catch_unwind.html (visited on 01/06/2022).

[Rusb] *Frequently Asked Questions*. URL: https://prev.rust-lang.org/en-US/faq.html (visited on 11/10/2021).

[Rusd] *Rust Releases*. URL: https://github.com/rust-lang/rust/blob/master/RELEASES.md (visited on 01/06/2022).

[Ruse] *Struct std::sync::Arc*. URL: https://doc.rust-lang.org/std/sync/struct.Arc.html (visited on 01/27/2022).

[Rusf] *Struct std::sync::Mutex*. URL: https://doc.rust-lang.org/std/sync/struct.Mutex.html (visited on 01/27/2022).

[Rusg] *Struct std::vec::Vec*. URL: https://doc.rust-lang.org/std/vec/struct.Vec.html (visited on 01/12/2022).

Other References

[And12] Brian Anderson. *The Rust compiler 0.1 is unleashed*. Jan. 2012. URL: https://mail.mozilla.org/pipermail/rust-dev/2012-January/001256.html (visited on 01/20/2022).

[Cwe] *2021 CWE Top 25 Most Dangerous Software Weaknesses*. July 2021. URL: https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html (visited on 11/02/2021).

[Kor02] Israel Koren. *Computer Arithmetic Algorithms*. 2ed. A.K.Peters, 2002. ISBN: 1-56881-160-8.

[NS04] S. Nelson and J. Schumann. "What makes a code review trustworthy?" In: *Proceedings of the 37th Annual Hawaii International Conference on System Sciences, 2004*. 2004, 10 pp.–. DOI: 10.1109/HICSS.2004.1265711.

[Och] Quentin Ochem. *Ada for the C++ and Java Developer*. 2021-12. AdaCore. URL: https://learn.adacore.com/pdf_books/courses/Ada_For_The_CPP_Java_Developer.pdf (visited on 01/22/2022).

[Pre] *MBDA Deutschland - Standorte*. July 2017. URL: https://www.mbda-deutschland.de/das-unternehmen/standorte/ (visited on 11/02/2021).