|  |  |
| --- | --- |
| Rust de toekomst van de opleiding? | Abstract  Dit onderzoek gaat over de toepasbaarheid van de taal Rust. Er wordt onderzocht of Rust een goede aanvullende taal is voor de opleiding HBO-ICT met afstudeerrichting Technische Informatica. Eerst zal er gekeken worden naar wat de taal Rust te bieden heeft later wordt er gekeken waar de taal Rust mogelijk een plek zou kunnen hebben binnen de opleiding  Mike Hilhorst  1676029 |

# Over rust

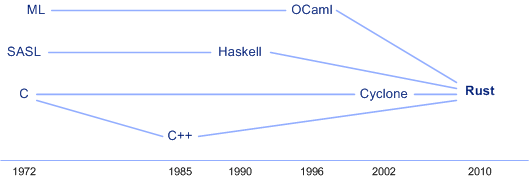
Rust

A recent [Stack Overflow survey](https://insights.stackoverflow.com/survey/2017?utm_source=so-owned&utm_medium=hero&utm_campaign=dev-survey-2017&utm_content=hero-questions#most-loved-dreaded-and-wanted) found that almost 80% of respondents loved using or wanted to develop with the Rust language. That’s an incredible number! So, what’s so good about Rust? This article explores the high points of this C-like language and illustrates why it should be next on your list of languages to learn.

Rust and its genealogy

First, let’s start with a quick history lesson. Rust is a new language relative to its predecessors (most importantly C, which preceded it by 38 years), but its genealogy creates its multiparadigm approach. Rust is considered a C-like language, but the other features it includes create advantages over its predecessors.

First, Rust is heavily influenced by Cyclone (a safe dialect of C and an imperative language), with some aspects of object-oriented features from C++. But it also includes functional features from languages like Haskell and OCaml. The result is a C-like language that supports multiparadigm programming (imperative, functional, and object oriented).



Key concepts in Rust

Rust has many features that make it useful, but developers and their needs differ. I cover five of the key concepts that make Rust worth learning and show these ideas in Rust source.

First, to get a feel for the code, let’s look at the canonical “Hello World” program that simply emits that message to the user (see [Listing 1](https://developer.ibm.com/articles/os-developers-know-rust/#listing01)).

**Listing 01. “Hello World” in Rust**

fn main()

{

println!( "Hello World.");

}

This simple program, similar to C, defines a main function that is the designated entry point for the program (and every program has one). The function is defined with the fnkeyword followed by an optional set of parameters within parentheses (()). The curly braces ({}) delineate the function; this function consists of a call to the println! macro, which emits formatted text to the console (stdout), as defined by the string parameter.

Rust includes a variety of features that make it interesting and worth the investment to learn. You’ll find concepts like modules for reusability, memory safety and guarantees (safe vs. unsafe operations), unrecoverable and recoverable error handling features, support for concurrency, and complex data types (called collections).

Reusable code via modules

Rust allows you to organize code in a way that fosters its reuse. You achieve this organization by using modules, which contain functions, structures, and even other modules that you can make public (that is, expose to users of the module) or private (that is, use only within the module and not by the module users — at least not directly). The module organizes code as a package that others can use.

[Listing 2](https://developer.ibm.com/articles/os-developers-know-rust/#listing02) provides a simple example. It starts by creating a new module called bits that contains three functions. The first function, called pos, is a private function that takes a u32 argument and returns a u32 (as indicated by the -> arrow), which is a 1 value shifted left bit times. Note that a return keyword isn’t needed here. This value is called by two public functions (note the pub keyword): decimal and hex. These functions call the private pos function and print the value of the bit position in decimal or hexadecimal format (note the use of  to indicate hexadecimal format). Finally, it declares a main function that calls the bits module’s two public functions, with the output shown at the end of Listing 2 as comments.

**Listing 02. Simple module example in Rust**

mod bits {

fn pos(bit: u32) ‑> u32 {

1 << bit

}

pub fn decimal(bit: u32) {

println!("Bits decimal {}", pos(bit));

}

pub fn hex(bit: u32) {

println!("Bits decimal 0x{:x}", pos(bit));

}

}

fn main( ) {

bits::decimal(8);

bits::hex(8);

}

// Bits decimal 256

// Bits decimal 0x100

Show more

Modules enable you to collect functionality in public or private ways, but you can also associate methods to objects by using the impl keyword.

Safety checks for cleaner code

The Rust compiler enforces memory safety guarantees and other checking that make the programming language safe (unlike C, which can be unsafe). So, in Rust, you’ll never have to worry about dangling pointers or using an object after it has been freed. These things are part of the core Rust language. But, in fields such as embedded development, it’s important to do things like place a structure at an address that represents a set of hardware registers.

Rust includes an unsafe keyword with which you can disable checks that would typically result in a compilation error. As shown in [Listing 3](https://developer.ibm.com/articles/os-developers-know-rust/#listing03), the unsafe keyword enables you to declare an unsafe block. In this example, I declare an unmutable variable x, and then a pointer to that variable called raw. Then, to de-reference raw (which in this case would print 1 to the console), I use the unsafe keyword to permit this operation, which would otherwise be flagged at compilation.

**Listing 3. Unsafe operations in Rust**

fn main() {

let a = 1;

let rawp = &a as const i32;

unsafe {

println!("rawp is {}", rawp);

}

}

You can apply the unsafe keyword to functions as well as blocks of code within a Rust function. The keyword is most common in writing bindings to non-Rust functions. This feature makes Rust useful for things like operating system development or embedded (bare-metal) programming.

Better error handling

Errors happen, regardless of the programming language you use. In Rust, errors fall into two camps: unrecoverable errors (the bad kind) and recoverable errors (the not-so-bad kind).

Unrecoverable errors

The Rust panic! function is similar to C‘s assert macro. It generates output to help the user debug a problem (as well as stopping execution before more catastrophic events occur). The panic! function is shown in [Listing 4](https://developer.ibm.com/articles/os-developers-know-rust/#listing04), with its executable output in comments.

**Listing 04. Handling unrecoverable errors in Rust with panic!**

fn main() {

panic!("Bad things happening.");

}

// thread 'main' panicked at 'Bad things happening.', panic.rs:2:4

// note: Run with RUST\_BACKTRACE=1 for a backtrace.

From the output, you can see that the Rust runtime indicates exactly where the issue occurred (line 2) and emitted the provided message (which could emit more descriptive information). As indicated in the output message, you could generate a stack backtrace by running with a special environment variable called RUST\_BACKTRACE. You can also invoke panic! internally based on detectable errors (such as accessing an invalid index of a vector).

Recoverable errors

Handling recoverable errors is a standard part of programming, and Rust includes a nice feature for error checking (see [Listing 5](https://developer.ibm.com/articles/os-developers-know-rust/#listing05)). Take a look at this feature in the context of a file operation. The File::open function returns a type of Result<T, E>, where *T* and *E*represent generic type parameters (in this context, they represent std::fs::File and std::io::Error). So, when you call File::open and no error has occurred (E is Ok), T would represent the return type (std::fs::File). If an error occurred, E would represent the type of error that occurred (using the type std::io::Error). (Note that my file variable \_fuses an underscore [\_] to omit the unused variable warning that the compiler generated.)

I then use a special feature in Rust called match, which is similar to the switch statement in C but more powerful. In this context, I match \_f against the possible error values (Okand Err). For Ok, I return the file for assignment; for Err, I use panic!.

**Listing 05. Handling recoverable errors in Rust with Result<t, e=””>**

use std::fs::File;

fn main() {

let \_f = File::open("file.txt");

let \_f = match \_f {

Ok(file) => file,

Err(why) => panic!("Error opening the file {:?}", why),

};

}

// thread 'main' panicked at 'Error opening the file Error { repr: Os

// { code: 2, message: "No such file or directory" } }', recover.rs:8:23

// note: Run with RUST\_BACKTRACE=1 for a backtrace.

Show more

Recoverable errors are simplified within Rust when you use the Result enum; they’re further simplified through the use of match. Note also in this example the lack of a File::close operation: The file is automatically closed when the scope of \_f ends.

Support for concurrency and threads

Concurrency commonly comes with issues (data races and deadlocks, to name two). Rust provides the means to spawn threads by using the native operating system but also attempts to mitigate the negative effects of threading. Rust includes message passing to allow threads to communicate with one another (via send and recv as well as locking through mutexes). Rust also provides the ability to permit a thread to borrow a value, which gives it ownership and effectively transitions the scope of the value (and its ownership) to a new thread. Thus, Rust provides memory safety along with concurrency without data races.

Consider a simple example of threading within Rust that introduces some new elements (vector operations) and brings back some previously discussed concepts (pattern matching). In [Listing 6](https://developer.ibm.com/articles/os-developers-know-rust/#listing06), I begin by importing the thread and Duration namespaces into my program. I then declare a new function called my\_thread, which represents the thread that I’ll create later. In this thread, I simply emit the thread’s identifier, and then sleep for a short time to permit the scheduler to allow another thread to run.

My main function is the heart of this example. I begin by creating an empty mutable vector that I can use to store values of the same type. I then create 10 threads by using the spawn function and push the resulting join handle into the vector (more on this later). This spawn example is detached from the current thread, which allows the thread to live after the parent thread has exited. After emitting a short message from the parent thread, I finally iterate the vector of JoinHandle types and wait for each child thread to exit. For each JoinHandle in the vector, I call the join function, which waits for that thread to exit before continuing. If the join function returns an error, I’ll expose that error through the match call.

**Listing 06. Threads in Rust**

use std::thread;

use std::time::Duration;

fn mythread() {

println!("Thread {:?} is running", std::thread::current().id());

thread::sleep(Duration::from\_millis(1));

}

fn main() {

let mut v = vec![];

for \_i in 1..10 {

v.push( thread::spawn(|| { my\_thread(); } ) );

}

println!("main() waiting.");

for child in v {

match child.join() {

Ok() => (),

Err(why) => println!("Join failure {:?}", why),

};

}

}

Show more

On execution, I see the output provided in [Listing 7](https://developer.ibm.com/articles/os-developers-know-rust/#listing07). Note here that the main thread continued to execute until the join process had begun. The threads then executed and exited at different times, identifying the asynchronous nature of threads.

**Listing 07. Thread output from the example code in Listing 6**

main() waiting.

Thread ThreadId(7) is running

Thread ThreadId(9) is running

Thread ThreadId(8) is running

Thread ThreadId(6) is running

Thread ThreadId(5) is running

Thread ThreadId(4) is running

Thread ThreadId(3) is running

Thread ThreadId(2) is running

Thread ThreadId(1) is running

Support for complex data types (collections)

The Rust standard library includes several popular and useful data structures that you can use in your development, including four types of data structures: sequences, maps, sets, and a miscellaneous type.

For sequences, you can use the vector type (Vec), which I used in the threading example. This type provides a dynamically resizeable array and is useful for collecting data for later processing. The VecDeque structure is similar to Vec, but you can insert it at both ends of the sequence. The LinkedList structure is similar to Vec, as well, but with it, you can split and append lists.

For maps, you have the HashMap and BTreeMap structures. You use the HashMap structure to create key-value pairs, and you can reference elements by their key (to retrieve the value). The BTreeMap is similar to the HashMap, but it can sort the keys, and you can easily iterate all the entries.

For sets, you have the HashSet and BTreeSet structures (which you’ll note follow the maps structures). These structures are useful when you don’t have values (just keys) and you easily recall the keys that have been inserted.

Finally, the miscellaneous structure is currently the BinaryHeap. This structure implements a priority queue with a binary heap.

Installing Rust and its tools

One of the simplest ways to install Rust is by using curl through the installation script. Simply execute the following string from the Linux® command line:

curl ‑sSf https://static.rust‑lang.org/rustup.sh | sh

This string transfers the rustup shell script from rust-lang.org, and then passes the script to the shell for execution. When complete, you can execute rustc -v to show the version of Rust you installed. With Rust installed, you can maintain it by using the rustuputility, which you can also use to update your Rust installation.

The Rust compiler is called rustc. In the examples shown here, the build process is simply defined as:

rustc threads.rs

…where the rust compiler produces a native executable file called threads. You can symbolically debug Rust programs by using either rust-lldb or rust-gdb.

You’ve probably noticed that the Rust programs I’ve demonstrated here have a unique style. You can learn this style through the automatic Rust source formatting by using the rustfmt utility. This utility, executed with a source file name, will automatically format your source in a consistent, standardized style.

Finally, although Rust is quite strict in what it accepts for source, you can use the rust-clippy program to dive further in to your source to identify elements of bad practice. Think of rust-clippy as the C lint utility.

Windows considerations

On Windows, Rust additionally requires the C++ build tools for Visual Studio 2013 or later. The easiest way to acquire the build tools is by installing [Microsoft Visual C++ Build Tools 2017](http://landinghub.visualstudio.com/visual-cpp-build-tools) which provides just the Visual C++ build tools. Alternately, you can [install](https://www.rust-lang.org/en-US/install.html) Visual Studio 2017, Visual Studio 2015, or Visual Studio 2013 and during the install, select **C++ tools**.

For further information about configuring Rust on Windows, see the [Windows-specific rustup documentation](https://github.com/rust-lang-nursery/rustup.rs/blob/master/README.md#working-with-rust-on-windows).

Going further

In mid-February 2018, the Rust team released version 1.24. This version includes incremental compilation, automatic source formatting with rustfmt, new optimizations, and library stabilizations. You can learn more about Rust and its evolution at the [Rust blog](https://blog.rust-lang.org/) and download Rust from the [Rust Language website](https://www.rust-lang.org/). There, you can read about the many other features Rust offers, including pattern matching, iterators, closures, and smart pointers.

# Beantwoording deelvragen

Hier worden de deelvragen beantwoordt

## Voor welke platformen is/wordt Rust ontwikkeld?

Rust ondersteunt vele platformen. Denk aan bijvoorbeeld windows, MacOS, Linux en embedded microcontrollers. Rust ondersteunt deze platvormen op verschillende niveaus. Er zijn 4 niveaus, deze verschillen per ondersteuning denk hierbij aan ondersteuning voor de standaard bibliotheek, Rustc en cargo.   
Niveau 3 is per ongeluk ontstaan en nieuwe platformen zullen dit niveau nooit krijgen.

**Niveau 1**

Dit niveau is goed ondersteunt en kan gezien worden als “Gegarandeerd werkend”.   
Deze platformen hebben de volgende eigenschappen:

* Officiële binary’s worden geleverd voor deze platformen.
* Automatische testen bestaan voor deze platformen.
* Veranderingen in de taal zijn master worden pas doorgelaten na testen geslaagd zijn
* Documentatie over gebruik is beschikbaar voor deze platformen

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Target | std | rustc | cargo | notes |
| i686-apple-darwin | ✓ | ✓ | ✓ | 32-bit OSX (10.7+, Lion+) |
| i686-pc-windows-gnu | ✓ | ✓ | ✓ | 32-bit MinGW (Windows 7+) |
| i686-pc-windows-msvc | ✓ | ✓ | ✓ | 32-bit MSVC (Windows 7+) |
| i686-unknown-linux-gnu | ✓ | ✓ | ✓ | 32-bit Linux (2.6.18+) |
| x86\_64-apple-darwin | ✓ | ✓ | ✓ | 64-bit OSX (10.7+, Lion+) |
| x86\_64-pc-windows-gnu | ✓ | ✓ | ✓ | 64-bit MinGW (Windows 7+) |
| x86\_64-pc-windows-msvc | ✓ | ✓ | ✓ | 64-bit MSVC (Windows 7+) |
| x86\_64-unknown-linux-gnu | ✓ | ✓ | ✓ | 64-bit Linux (2.6.18+) |

**Niveau 2**   
Dit niveau is ondersteunt en kan gezien worden als “Gegarandeerd bouwend”.   
Deze platformen hebben de volgende eigenschappen:

* Officiële binarys worden geleverd voor deze platformen.
* Automatische testen zijn opgezet voor deze platformen, maar deze kunnen misschien niet gerund worden.
* Veranderingen in de taal zijn master worden goedgekeurd door platform bouwen. Voor sommige talen houdt dit in dat alleen de standaard bibliotheek gecompileerd moet worden voor andere Rustc en Cargo ook.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Target | std | rustc | cargo | notes |
| aarch64-apple-ios | ✓ |  |  | ARM64 iOS |
| aarch64-fuchsia | ✓ |  |  | ARM64 Fuchsia |
| aarch64-linux-android | ✓ |  |  | ARM64 Android |
| aarch64-unknown-linux-gnu | ✓ | ✓ | ✓ | ARM64 Linux |
| aarch64-unknown-linux-musl | ✓ |  |  | ARM64 Linux with MUSL |
| arm-linux-androideabi | ✓ |  |  | ARMv7 Android |
| arm-unknown-linux-gnueabi | ✓ | ✓ | ✓ | ARMv6 Linux |
| arm-unknown-linux-gnueabihf | ✓ | ✓ | ✓ | ARMv6 Linux, hardfloat |
| arm-unknown-linux-musleabi | ✓ |  |  | ARMv6 Linux with MUSL |
| arm-unknown-linux-musleabihf | ✓ |  |  | ARMv6 Linux, MUSL, hardfloat |
| armv5te-unknown-linux-gnueabi | ✓ |  |  | ARMv5TE Linux |
| armv7-apple-ios | ✓ |  |  | ARMv7 iOS, Cortex-a8 |
| armv7-linux-androideabi | ✓ |  |  | ARMv7a Android |
| armv7-unknown-linux-gnueabihf | ✓ | ✓ | ✓ | ARMv7 Linux |
| armv7-unknown-linux-musleabihf | ✓ |  |  | ARMv7 Linux with MUSL |
| armv7s-apple-ios | ✓ |  |  | ARMv7 iOS, Cortex-a9 |
| asmjs-unknown-emscripten | ✓ |  |  | asm.js via Emscripten |
| i386-apple-ios | ✓ |  |  | 32-bit x86 iOS |
| i586-pc-windows-msvc | ✓ |  |  | 32-bit Windows w/o SSE |
| i586-unknown-linux-gnu | ✓ |  |  | 32-bit Linux w/o SSE |
| i586-unknown-linux-musl | ✓ |  |  | 32-bit Linux w/o SSE, MUSL |
| i686-linux-android | ✓ |  |  | 32-bit x86 Android |
| i686-unknown-freebsd | ✓ | ✓ | ✓ | 32-bit FreeBSD |
| i686-unknown-linux-musl | ✓ |  |  | 32-bit Linux with MUSL |
| mips-unknown-linux-gnu | ✓ | ✓ | ✓ | MIPS Linux |
| mips-unknown-linux-musl | ✓ |  |  | MIPS Linux with MUSL |
| mips64-unknown-linux-gnuabi64 | ✓ | ✓ | ✓ | MIPS64 Linux, n64 ABI |
| mips64el-unknown-linux-gnuabi64 | ✓ | ✓ | ✓ | MIPS64 (LE) Linux, n64 ABI |
| mipsel-unknown-linux-gnu | ✓ | ✓ | ✓ | MIPS (LE) Linux |
| mipsel-unknown-linux-musl | ✓ |  |  | MIPS (LE) Linux with MUSL |
| powerpc-unknown-linux-gnu | ✓ | ✓ | ✓ | PowerPC Linux |
| powerpc64-unknown-linux-gnu | ✓ | ✓ | ✓ | PPC64 Linux |
| powerpc64le-unknown-linux-gnu | ✓ | ✓ | ✓ | PPC64LE Linux |
| s390x-unknown-linux-gnu | ✓ | ✓ | ✓ | S390x Linux |
| sparc64-unknown-linux-gnu | ✓ |  |  | SPARC Linux |
| sparcv9-sun-solaris | ✓ |  |  | SPARC Solaris 10/11, illumos |
| wasm32-unknown-unknown | ✓ |  |  | WebAssembly |
| wasm32-unknown-emscripten | ✓ |  |  | WebAssembly via Emscripten |
| x86\_64-apple-ios | ✓ |  |  | 64-bit x86 iOS |
| x86\_64-fuchsia | ✓ |  |  | 64-bit Fuchsia |
| x86\_64-linux-android | ✓ |  |  | 64-bit x86 Android |
| x86\_64-rumprun-netbsd | ✓ |  |  | 64-bit NetBSD Rump Kernel |
| x86\_64-sun-solaris | ✓ |  |  | 64-bit Solaris 10/11, illumos |
| x86\_64-unknown-cloudabi | ✓ |  |  | 64-bit CloudABI |
| x86\_64-unknown-freebsd | ✓ | ✓ | ✓ | 64-bit FreeBSD |
| x86\_64-unknown-linux-gnux32 | ✓ |  |  | 64-bit Linux |
| x86\_64-unknown-linux-musl | ✓ |  |  | 64-bit Linux with MUSL |
| x86\_64-unknown-netbsd | ✓ | ✓ | ✓ | NetBSD/amd64 |
| x86\_64-unknown-redox | ✓ |  |  | Redox OS |

**Niveau 3**

Dit niveau is ondersteunt en kan gezien worden als “Gegarandeerd bouwend”, maar dan zonder beschikbare versies via Rustup. Er worden geen automatische testen gerund dus er wordt niet gegarandeerd dat de geproduceerd build werkt, maar meestal werken deze platformen best goed. Deze platformen hebben de volgende eigenschappen:

* Automatische testen zijn opgezet voor deze platformen, maar deze kunnen misschien niet gerund worden.
* Veranderingen in de taal zijn master worden goedgekeurd door platform bouwen. Voor sommige talen houdt dit in dat alleen de standaard bibliotheek gecompileerd moet worden voor andere Rustc en Cargo ook.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Target | std | rustc | cargo | notes |
| aarch64-unknown-cloudabi | ✓ |  |  | ARM64 CloudABI |
| armv7-unknown-cloudabi-eabihf | ✓ |  |  | ARMv7 CloudABI, hardfloat |
| i686-unknown-cloudabi | ✓ |  |  | 32-bit CloudABI |
| powerpc-unknown-linux-gnuspe | ✓ |  |  | PowerPC SPE Linux |
| sparc-unknown-linux-gnu | ✓ |  |  | 32-bit SPARC Linux |

**Niveau 4**

De Rust codebase heft ondersteuning voor deze platformen, maar worden niet gebouwd of getest. Deze platformen kunnen misschien niet werken, verder zijn er geen officiële versies beschikbaar

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Target | std | rustc | cargo | notes |
| i686-pc-windows-msvc (XP) | ✓ |  |  | Windows XP support |
| i686-unknown-haiku | ✓ |  |  | 32-bit Haiku |
| i686-unknown-netbsd | ✓ |  |  | NetBSD/i386 with SSE2 |
| le32-unknown-nacl | ✓ |  |  | PNaCl sandbox |
| mips-unknown-linux-uclibc | ✓ |  |  | MIPS Linux with uClibc |
| mipsel-unknown-linux-uclibc | ✓ |  |  | MIPS (LE) Linux with uClibc |
| msp430-none-elf | \* |  |  | 16-bit MSP430 microcontrollers |
| sparc64-unknown-netbsd | ✓ | ✓ |  | NetBSD/sparc64 |
| thumbv6m-none-eabi | \* |  |  | Bare Cortex-M0, M0+, M1 |
| thumbv7em-none-eabi | \* |  |  | Bare Cortex-M4, M7 |
| thumbv7em-none-eabihf | \* |  |  | Bare Cortex-M4F, M7F, FPU, hardfloat |
| thumbv7m-none-eabi | \* |  |  | Bare Cortex-M3 |
| x86\_64-pc-windows-msvc (XP) | ✓ |  |  | Windows XP support |
| x86\_64-unknown-bitrig | ✓ | ✓ |  | 64-bit Bitrig |
| x86\_64-unknown-dragonfly | ✓ | ✓ |  | 64-bit DragonFlyBSD |
| x86\_64-unknown-haiku | ✓ |  |  | 64-bit Haiku |
| x86\_64-unknown-openbsd | ✓ | ✓ |  | 64-bit OpenBSD |
| [NVPTX](https://github.com/japaric/nvptx#targets) | \*\* |  |  | --emit=asm generates PTX code that runs on NVIDIA GPUs |

\* Dit zijn “bare-metal” microcontrollers doelen, deze hebben alleen toegang tot de kern bibliotheek en niet de standaard bibliotheek.

\*\* Er is back-end ondersteuning voor deze doelen, maar geen doelwit dat (nog) in rustc is ingebouwd. Je moet uw eigen doelspecificatiebestand schrijven. Deze doelen ondersteunen alleen de kernbibliotheek.

Dit zijn niet de enige platforms die Rust kan compileren! Dat zijn degenen met ingebouwde doeldefinities en/of standaard bibliotheekondersteuning. Tijdens het linken van de kernbibliotheek, kan Rust zich richten op "bare metal" in de x86-, ARM-, MIPS- en PowerPC-families, hoewel hiervoor mogelijk aangepaste doelspecificaties moeten worden gedefinieerd. Al dergelijke scenario's zijn niveau 4.

## Hoe werkt Rust intern op low- en highlevel?

## Nutteloos niet van belang voor dit onderzoek, mss memory menagment

## Vanuit welke talen leent rust concepten, wat dat zijn deze concepten?

Rust is een relatieve nieuwe taal en leent concepten uit al bestaande talen.   
Waarom leent Rust concepten uit andere talen? Omdat alles van de grond af zelf te maken is tijdsintensief en garandeert geen succes. De geleende concepten zijn bewezen goed te werken omdat ze al jaren in andere talen gebruikt worden.  
  
De eerste release van Rust gebruikte meer concepten dan nu. Deze concepten zijn weggevallen daarom staan dus niet in de onderstaande lijst. In de onderstaande lijst staan de huidige nog gebruikte concepten die geleend worden. Links staat de taal waarvan een of meerder concept(en) van geleend wordt. Rechts staan de concepten die geleend worden.

|  |  |
| --- | --- |
| Taal | Concepten |
| [SML](http://www.macs.hw.ac.uk/ultra/skalpel/html/sml.html), [OCaml](https://ocaml.org/) | algebraic data types,  pattern matching,  type inference,  semicolon statement separation |
| [C++](http://www.cplusplus.com/info/) | references,  Resource Acquisition Is Initialization (RAII),  smart pointers,  move semantics,  monomorphization,  memory model |
| [ML Kit](http://wiki.c2.com/?MlKit), [Cyclone](https://cyclone.thelanguage.org/) | region based memory management |
| [Haskell (GHC)](https://www.haskell.org/) | typeclasses, type families |
| [Newsqueak](http://dbpedia.org/page/Newsqueak), [Alef](https://en.wikipedia.org/wiki/Alef_(programming_language)), [Limbo](https://en.wikipedia.org/wiki/Limbo_(programming_language)) | channels, concurrency |
| [Erlang](https://www.erlang.org/) | message passing, thread failure |
| [Swift](https://developer.apple.com/swift/) | optional bindings |
| [Scheme](https://www.scheme.com/tspl4/) | hygienic macros |
| [C#](https://en.wikipedia.org/wiki/C_Sharp_(programming_language)) | attributes |
| [Unicode Annex #31](http://www.unicode.org/reports/tr31/) | identifier and pattern syntax |

[(2)](#_Bronnen)

## Wat zijn de voor- en nadelen van Rust tegenover C++ in de scope van de lesstof van de opleiding HBO-ICT met afstudeerrichting Technische Informatica?

Een van de voordelen kan performance zijn. Hierdoor zouden grotere programma’s uitgevoerd kunnen worden op bestaande hardware (Denk hierbij aan [ARM Cortex M3](https://store.arduino.cc/arduino-due) chips die de Hogeschool Utrecht gebruikt.) of bestaande programma’s kunnen worden uitgevoerd op kleinere (minder snelle) hardware (denk hieraan de ATmega328 van de [Arduino Nano](https://store.arduino.cc/arduino-nano)). Dit zou geld kunnen besparen want de Arduino Due kost €35 [(3.0)](#_Bronnen) en de Arduino Nano kost €20. [(3.1)](#_Bronnen).

Korte uitleg over de data hier onder

* Source: in welke taal de code is uit gevoerd.
* Seconde: uitvoer tijd van de code.
* Geheugen: geheugen gebruikt tijdens de uitvoer (in bytes).
* Grootte: grootte van de source files (in bytes).
* CPU lading: hoeveel elke kern beladen werd tijdens de uitvoer, deze code is uitgevoerd op een quad core en daarom ook 4 percentages.

Er zijn 10 verschillende algoritmes geschreven voor 4 verschillende talen namelijk: C++, Rust, Python en Haskell. Deze zijn uitgevoerd dit zijn de resultaten. De source code van deze programma’s is te vinden in de appendix.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| [Reverse-complement](https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/revcomp.html#revcomp) | | | | [(1.0)](#_Bronnen) | | | |
| Source | Seconde | Geheugen | Grootte | CPU lading | | | |
| C++ | 2,94 | 980.524 | 2.280 | 15% | 50% | 52% | 40% |
| Rust | 1,60 | 995.212 | 1.376 | 24% | 25% | 96% | 30% |
| Python 3 | 16,76 | 1.005.252 | 814 | 65% | 21% | 44% | 17% |
| Haskell | 5,67 | 501.336 | 1.020 | 26% | 12% | 12% | 40% |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| [N-body](https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/nbody.html#nbody) | | | | [(1.1)](#_Bronnen) | | | |
| Source | Seconde | Geheugen | Grootte | CPU lading | | | |
| C++ | 9,42 | 1.712 | 1.763 | 100% | 1% | 2% | 0% |
| Rust | 13,25 | 1.808 | 1.805 | 0% | 0% | 1% | 100% |
| Python 3 | 14min | 8.212 | 1.196 | 91% | 0% | 1% | 9% |
| Haskell | 22,01 | 3.936 | 1.883 | 99% | 99% | 100% | 99% |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| [K-nucleotide](https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/knucleotide.html#knucleotide) | | | | [(1.2)](#_Bronnen) | | | |
| Source | Seconde | Geheugen | Grootte | CPU lading | | | |
| C++ | 3,83 | 156.104 | 1.624 | 72% | 73% | 98% | 72% |
| Rust | 5,98 | 137.956 | 1.648 | 78% | 49% | 90% | 85% |
| Python 3 | 79,79 | 250.948 | 1.967 | 98% | 96% | 96% | 99% |
| Haskell | 35,51 | 604.996 | 1.486 | 89% | 93% | 90% | 87% |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| [Regex-redux](https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/regexredux.html#regexredux) | | | | [(1.3)](#_Bronnen) | | | |
| Source | Seconde | Geheugen | Grootte | CPU lading | | | |
| C++ | 1,83 | 203.680 | 1.315 | 49% | 87% | 57% | 51% |
| Rust | 2,44 | 194.804 | 765 | 85% | 41% | 20% | 16% |
| Python 3 | 15,56 | 439.964 | 512 | 25% | 92% | 32% | 32% |
| Haskell | Bad output |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Binary-trees | | | | [(1.4)](#_Bronnen) | | | |
| Source | Seconde | Geheugen | Grootte | CPU lading | | | |
| C++ | 3,67 | 118.620 | 809 | 75% | 78% | 99% | 76% |
| Rust | 4,14 | 175.692 | 721 | 90% | 90% | 91% | 100% |
| Python 3 | 92,72 | 448.844 | 589 | 87% | 90% | 96% | 87% |
| Haskell | 12,59 | 478.012 | 592 | 95% | 90% | 88% | 94% |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| [Fasta](https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/fasta.html#fasta) | | | | [(1.5)](#_Bronnen) | | | |
| Source | Seconde | Geheugen | Grootte | CPU lading | | | |
| C++ | 1,33 | 1.744 | 2.711 | 82% | 82% | 81% | 81% |
| Rust | 1,46 | 3.112 | 1.906 | 84% | 83% | 84% | 89% |
| Python 3 | 62,88 | 680.736 | 1.947 | 60% | 56% | 48% | 62% |
| Haskell | 9,46 | 4.980 | 969 | 100% | 2% | 2% | 3% |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| [Spectral-norm](https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/spectralnorm.html#spectralnorm) | | | | [(1.6)](#_Bronnen) | | | |
| Source | Seconde | Geheugen | Grootte | CPU lading | | | |
| C++ | 1,98 | 1.168 | 1.044 | 100% | 99% | 99% | 99% |
| Rust | 1,97 | 2.600 | 1.126 | 100% | 100% | 100% | 99% |
| Python 3 | 193,86 | 50.556 | 443 | 98% | 98% | 99% | 99% |
| Haskell | 4,17 | 3.812 | 987 | 99% | 97% | 99% | 98% |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| [Fannkuch-redux](https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/fannkuchredux.html#fannkuchredux) | | | | [(1.7)](#_Bronnen) | | | |
| Source | Seconde | Geheugen | Grootte | CPU lading | | | |
| C++ | 10,07 | 1.856 | 980 | 99% | 100% | 100% | 95% |
| Rust | 9,87 | 1.848 | 1.020 | 100% | 95% | 100% | 100% |
| Python 3 | 9 min | 48.052 | 950 | 99% | 100% | 97% | 100% |
| Haskell | 18,20 | 4.024 | 842 | 99% | 100% | 95% | 100% |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| [Mandelbrot](https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/mandelbrot.html#mandelbrot) | | | | [(1.8)](#_Bronnen) | | | |
| Source | Seconde | Geheugen | Grootte | CPU lading | | | |
| C++ | 1,82 | 29.092 | 1.002 | 98% | 97% | 97% | 100% |
| Rust | 1,74 | 33.712 | 1.332 | 98% | 100% | 98% | 98% |
| Python 3 | 279,86 | 49.334 | 688 | 100% | 100% | 100% | 100% |
| Haskell | 11,60 | 38,744 | 782 | 100% | 100% | 100% | 100% |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| [Pidigits](https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/pidigits.html#pidigits) | | | | [(1.9)](#_Bronnen) | | | |
| Source | Seconde | Geheugen | Grootte | CPU lading | | | |
| C++ | 1,89 | 4,312 | 513 | 1% | 1% | 99% | 1% |
| Rust | 1,74 | 4.520 | 1.366 | 1% | 3% | 0% | 99% |
| Python 3 | 3,51 | 10.500 | 385 | 1% | 1% | 0% | 100% |
| Haskell | 4,20 | 9.724 | 585 | 27% | 8% | 8% | 81% |

### Hoe is dit gemeten?

Deze testen zijn uitgevoerd op een computer met:

Processor: Intel Q6600 quad-core op 2,4Ghz

RAM: 4GB ram.

OS: Ubuntu™ 18.10 Linux x64 4.18.0-10-generic.

Elk programma is uitgevoerd en gemeten bij de kleinste input waarde, de output werd opgeslagen in een bestand om vergeleken te worden met de verwachte output. Zolang de output overeen blijft komen met de verwachte output werd het programma getest met een grotere input. Totdat alle inputs waren doorlopen. Als het programma de verwachte output bleef geven tijdens het testen (binnen een redelijke tijd) dan werd er nog 5 keer gemeten. Tijdens deze testen werd de output geledigd (/dev/null).

Als het programma niet de verwachte output leverde binnen een redelijke tijd dan werd het programma geforceerd gestopt. Als de metingen met kleinere input wel de verwachte uitvoer leverde dan werd deze 5 keer gemeten. Ook tijdens deze testen werd de output geledigd /dev/null). [(1.10)](#_Bronnen)

### Hoe zijn de programma’s getimed?

Elk programma is uitgevoerd als een “Child-process” van een Python script.  
Er werd gebruik gemaakt van “time.time()” voordat het proces werd gestart en na dat deze gesloten was. [(1.10)](#_Bronnen)

### Hoe is het geheugen gebruik gemeten?

Door “GLIBTOP\_PROC\_MEM\_RESIDENT” te samplen voor het hoofd proces en zijn “Child-process” elke 0,2 seconde. [(1.10)](#_Bronnen)

### Hoe is de source file gemeten?

De code werd in een bestand gezet zonder commentaar, extra spaties en extra witregels. Op dit bestand werd een lichte GZip compressie uitgevoerd. De grootte van dit bestand werd gemeten. [(1.10)](#_Bronnen)

### Hoe is de CPU lading gemeten?

De GTop CPU inactief en de GTop CPU totaal werden gemeten voor de opstarten van het “Child-process” en na het afsluiten hiervan. De percentages zijn het totaal min de inactieve waardes. [(1.10)](#_Bronnen)

### Conclusie performance

**Uitvoer snelheid in seconde**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Taal | RC | NB | KN | RR | BT | FA | SN | FR | MB | PD |
| Rust | 1,60 | 13,25 | 5,98 | 2,44 | 4,14 | 1,46 | 1,97 | 9,87 | 1,74 | 1,74 |
| C++ | 2,94 | 9,42 | 3,83 | 1,83 | 3,67 | 1,33 | 1,98 | 10,07 | 1,82 | 1,89 |

Uit dit tabel is te halen dat rust en c++ soms veel en soms weinig van elkaar afzitten met snelheid en dat beide talen 5 keer de snelste waren. Maar als we wat gaan rekenen dan blijkt dat als C++ sneller is dat het dan gemiddeld is met ongeveer 22% en als Rust sneller is dan is dat gemiddeld ongeveer 12%.

**Geheugen gebruik in bytes**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Taal | RC | NB | KN | RR | BT | FA | SN | FR | MB | PD |
| Rust | 995.212 | 1.808 | 137.956 | 194.804 | 175.692 | 3.112 | 2.600 | 1.848 | 33.712 | 4.520 |
| C++ | 980.524 | 1.712 | 156.104 | 203.680 | 118.620 | 1.744 | 1.168 | 1.856 | 29.092 | 3.212 |

Als we kijken naar het tabel van geheugengebruik dan blijkt dat het gebruik hoger light dan die van C++, met ongeveer 15%. Dat is een vors verschil zeker voor microcontrollers. Bijvoorbeeld de Arduino Due die de hogenschool gebruikt, deze microcontroller heeft maar 96 KB SRAM (twee banken van 64KB en 32KB) [(3.0)](#_Bronnen). Dit is al weinig geheugen, dus elke byte telt hier.

De source file grootte van Rust ligt gemiddeld ook boven die van C++

C++ heeft een gemiddelde grootte van 1044 bytes en Rust heeft een gemiddelde grootte 1319 bytes[(1.10)](#_Bronnen). Rust geeft ongeveer 26% grotere bestande.

In het kort: Rust is gemiddeld 10% langzamer dan C++, gebruikt gemiddeld 15% meer geheugen en geeft gemiddeld 26% grotere source files.

## Wat zou een student leren van Rust tegenover C++ scope van de opleiding HBO-ICT met afstudeerrichting Technische Informatica?

## Waar zou de taal Rust een plek hebben in de opleiding HBO-ICT met afstudeerrichting Technische Informatica?

Leesbaarheid van code bij mensen testen.

Leesbaarheid van code bij ti’ers testen.

En verband trekken met sourcefile size

## Bronnen

1.0: Reverse-complement | Computer Language Benchmarks Game. (z.d.). Geraadpleegd op 27 november 2018, van <https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/revcomp.html>

1.1: N-body | Computer Language Benchmarks Game. (z.d.). Geraadpleegd op 27 november 2018, van <https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/nbody.html>

1.2: K-nucleotide | Computer Language Benchmarks Game. (z.d.). Geraadpleegd op 27 november 2018, van <https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/knucleotide.html>

1.3 Regex-redux | Computer Language Benchmarks Game. (z.d.). Geraadpleegd op 27 november 2018, van <https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/regexredux.html>

1.4: Binary-trees | Computer Language Benchmarks Game. (z.d.). Geraadpleegd op 27 november 2018, van <https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/binarytrees.html>

1.5: Fasta | Computer Language Benchmarks Game. (z.d.). Geraadpleegd op 27 november 2018, van <https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/fasta.html>

1.6: Spectral-norm | Computer Language Benchmarks Game. (z.d.). Geraadpleegd op 27 november 2018, van <https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/spectralnorm.html>

1.7: Fannkuch-redux | Computer Language Benchmarks Game. (z.d.). Geraadpleegd op 27 november 2018, van <https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/fannkuchredux.html>

1.8: Mandelbrot | Computer Language Benchmarks Game. (z.d.). Geraadpleegd op 27 november 2018, van <https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/mandelbrot.html>

1.9: Pidigits | Computer Language Benchmarks Game. (z.d.). Geraadpleegd op 27 november 2018, van <https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/pidigits.html>

1.10: How programs are measured | Computer Language Benchmarks Game. (z.d.). Geraadpleegd op 27 november 2018, van <https://benchmarksgame-team.pages.debian.net/benchmarksgame/how-programs-are-measured.html>

2.0: Appendix: Influences - The Rust Reference. (z.d.). Geraadpleegd op 27 november 2018,  
 van <https://doc.rust-lang.org/reference/influences.html>

2.1: Rust Platform Support · The Rust Programming Language. (z.d.). Geraadpleegd op 27 november   
 2018, van <https://forge.rust-lang.org/platform-support.html>

3.0: Arduino Due. (z.d.). Geraadpleegd op 27 november 2018, van   
 <https://store.arduino.cc/arduino-due>

3.1: Arduino Nano. (z.d.). Geraadpleegd op 27 november 2018, van   
 <https://store.arduino.cc/arduino-nano>

5

6

7

8

9

10

11

12

13

14