Red Hat Developer Tools 1

Using Rust 1.45.2 Toolset

Installing and using Rust 1.45.2 Toolset

Last Updated: 2020-11-25

Red Hat Developer Tools 1 Using Rust 1.45.2 Toolset

Installing and using Rust 1.45.2 Toolset

Eva-Lotte Gebhardt
egebhard@redhat.com

Zuzana Zoubkova
zzoubkov@redhat.com

Olga Tikhomirova
otikhomi@redhat.com

Peter Macko

Kevin Owen

Vladimir Slavik

Legal Notice

Abstract

Rust is a Red Hat offering for developers on the Red Hat Enterprise Linux platform. The Rust User Guide provides an overview of this product, explains how to invoke and use the Rust versions of the tools, and links to resources with more in-depth information.

Table of Contents

CHAPTER 1. RUST

1.1. ABOUT RUST TOOLSET

Rust Toolset is a Red Hat offering for developers on the Red Hat Enterprise Linux platform. It provides the Rust programming language compiler **rustc**, the **Cargo** build tool and dependency manager, the **rustfmt** tool, and required libraries. The **cargo-vendor** package is now built into the **cargo** command, but its usage remains the same.

Rust Toolset is distributed as a part of Red Hat Developer Tools for Red Hat Enterprise Linux 7. Rust Toolset is available as a module for Red Hat Enterprise Linux 8.

The following components are available as a part of Rust Toolset:

Table 1.1. Rust Components

Name	Version	Description
rust	1.45.2	The Rust compiler front-end for LLVM.
cargo	1.45.2	A build system and dependency manager for Rust.
rustfmt	1.45.2	A tool for automatic formatting of Rust code.

1.2. COMPATIBILITY

Rust Toolset is available for Red Hat Enterprise Linux 7 and Red Hat Enterprise Linux 8 on the following architectures:

- The 64-bit Intel and AMD architectures
- The IBM Power Systems architecture [1]
- The 64-bit ARM architecture [2]
- The little-endian variant of IBM Power Systems architecture
- The IBM Z Systems architecture

1.3. GETTING ACCESS TO RUST TOOLSET ON RED HAT ENTERPRISE LINUX 7

This chapter lists the steps to perform before installing Rust Toolset on a Red Hat Enterprise Linux 7 system. Complete the following steps to attach a subscription that provides access to the repository for Red Hat Developer Tools, and then enable the Red Hat Developer Tools and Red Hat Software Collections repositories.

Prerequisites

- Verify that **wget** is installed on your system. The tool is available from the default Red Hat Enterprise Linux repositories. To install it, run the following command as root:

```
# yum install wget
```

Procedure

1. Get the latest subscription data from the server:

```
# subscription-manager refresh
```

2. Use the following command to register the system:

```
# subscription-manager register
```

   You can also register the system by following the appropriate steps in Registering and Unregistering a System in the Red Hat Subscription Management document.

3. Display a list of all subscriptions that are available for your system and identify the pool ID for the subscription:

```
# subscription-manager list --available
```

   This command displays the subscription name, unique identifier, expiration date, and other details related to it. The pool ID is listed on a line beginning with **Pool ID**.

4. Attach the subscription that provides access to the **Red Hat Developer Tools** repository. Use the pool ID you identified in the previous step.

```
# subscription-manager attach --pool=<appropriate pool ID from the subscription>
```

5. Verify the list of subscriptions attached to your system:

```
# sudo subscription-manager list --consumed
```

6. Enable the **rhel-7-*variant*-devtools-rpms** repository:

```
# subscription-manager repos --enable rhel-7-variant-devtools-rpms
```

   Replace ***variant*** with the Red Hat Enterprise Linux system variant (**server** or **workstation**).

   Consider using Red Hat Enterprise Linux Server to access the widest range of the development tools.

7. Enable the **rhel-*variant*-rhscl-7-rpms** repository:

```
# subscription-manager repos --enable rhel-variant-rhscl-7-rpms
```

   Replace ***variant*** with the Red Hat Enterprise Linux system variant (**server** or **workstation**).

8. Add the Red Hat Developer Tools GPG key to your system:

```
# cd /etc/pki/rpm-gpg
# wget -O RPM-GPG-KEY-redhat-devel https://www.redhat.com/security/data/a5787476.txt
# rpm --import RPM-GPG-KEY-redhat-devel
```

Once the subscription is attached to the system and the repositories are enabled, install Rust Toolset as described in Section 1.4, "Installing Rust Toolset".

Additional resources

- For more information on how to register your system using Red Hat Subscription Management and associate it with subscriptions, see the Red Hat Subscription Management collection of guides.

1.4. INSTALLING RUST TOOLSET

Complete the following steps to install Rust Toolset on a Red Hat Enterprise Linux system.

NOTE

A valid subscription that provides access to the Red Hat Developer Tools content set is required to install Rust Toolset on a Red Hat Enterprise Linux 7 system. For detailed instructions on how to associate your Red Hat Enterprise Linux 7 system with an appropriate subscription and get access to Rust Toolset, see Section 1.3, "Getting access to Rust Toolset on Red Hat Enterprise Linux 7".

IMPORTANT

Before installing Rust Toolset, install all available Red Hat Enterprise Linux updates.

1. Install all of the components included in Rust Toolset for your operating system:
   - On Red Hat Enterprise Linux 7, install the **rust-toolset-1.45** collection:

     ```
     # yum install rust-toolset-1.45
     ```

   - On Red Hat Enterprise Linux 8, install the **rust-toolset** module:

     ```
     # yum module install rust-toolset
     ```

     This installs all development and debugging tools, and other dependent packages to the system. Notably, Rust Toolset has a dependency on LLVM Toolset.

Installable documentation

- Install *The Rust Programming Language* book and API documentation in HTML format:
  - On Red Hat Enterprise Linux 7, install the **rust-toolset-1.45-rust-doc** package:

    ```
    # yum install rust-toolset-1.45-rust-doc
    ```

    The book is available in **/opt/rh/rust-toolset-1.45/root/usr/share/doc/rust/html/index.html**

    The API documentation for all crates is available in HTML format in **/opt/rh/rust-toolset-1.45/root/usr/share/doc/rust/html/std/index.html**.

- On Red Hat Enterprise Linux 8, install the **rust-doc** package:

```
# yum install rust-doc
```

The book is available in **/usr/share/doc/rust/html/index.html**

The API documentation for all crates is available in HTML format in **/usr/share/doc/rust/html/std/index.html**.

Online documentation

- Rust documentation – Online documentation provided by the Rust project.

---

[1] Only available on RHEL 7.

[2] Only available on RHEL 8.

CHAPTER 2. CARGO

Cargo is a tool for development using the Rust programming language. **Cargo** serves as:

- Build tool and frontend for the Rust compiler **rustc**.

NOTE

Consider using **Cargo** over **rustc**.

- Package and dependency manager.
  Cargo allows Rust projects to declare dependencies with specific version requirements. **Cargo** will resolve the full dependency graph, download packages as needed, and build and test the entire project.

Rust Toolset is distributed with **Cargo 1.45.2**.

2.1. CREATING A NEW RUST PROJECT

To create a Rust program on the command line, run the **cargo** tool as follows:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.45 'cargo new --bin project_name'
```

- For Red Hat Enterprise Linux 8:

```
$ cargo new --bin project_name
```

There is a specific convention that defines directory structure and file placement within a Cargo package. Running the **cargo new** command generates the package directory structure and templates for both a manifest and a project file. By default, it also initializes a new Git repository in the package root directory.

For a binary program, it creates a directory ***project_name*** containing a text file named **Cargo.toml** and a subdirectory **src** containing a text file named **main.rs**.

To configure the project and add dependencies, edit the manifest file **Cargo.toml**. This file contains all the metadata needed during project compilation. See Section 2.6, “Configuring Rust project dependencies”.

To edit the project code, edit the main executable file **main.rs** and add new source files in the **src** subdirectory as needed.

In the command above, the option **--bin** was used to create a binary program. To create a library for a Cargo package instead of a program, run the **cargo** tool on the command line and pass the **--lib** option as follows:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.45 'cargo new --lib project_name'
```

- For Red Hat Enterprise Linux 8:

```
$ cargo new --lib project_name
```

As in the previous case, this will create a package directory structure with a root directory ***project_name*** containing a library file named **lib.rs** located in the **src** subdirectory.

NOTE

You can execute any command using the **scl** utility on Red Hat Enterprise Linux 7, causing it to be run with the Rust binaries available. To use Rust Toolset on Red Hat Enterprise Linux 7 without a need to use **scl enable** with every command, run a shell session with:

```
$ scl enable rust-toolset-1.45 'bash'
```

Example 2.1. Creating a Rust project using Cargo

Create a new Rust project called **helloworld**:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.45 'cargo new --bin helloworld'
    Created binary (application) helloworld project
```

- For Red Hat Enterprise Linux 8:

```
$ cargo new --bin helloworld
    Created binary (application) helloworld project
```

Examine the result:

```
$ cd helloworld
$ tree
.
├── Cargo.toml
└── src
    └── main.rs

1 directory, 2 files
$ cat src/main.rs
fn main() {
    println!("Hello, world!");
}
```

A directory **helloworld** is created for the project, with a file **Cargo.toml** for tracking project metadata, and a subdirectory **src** containing the main source code file **main.rs**.

The source code file **main.rs** has been initialized by **Cargo** to a sample hello world program.

NOTE

The **tree** tool is available from the default Red Hat Enterprise Linux repositories. To install it:

```
# yum install tree
```

2.2. BUILDING A RUST PROJECT

To build a Rust project on the command line, change to the project directory and run the **cargo** tool as follows:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.45 'cargo build'
```

- For Red Hat Enterprise Linux 8:

```
$ cargo build
```

Running the **cargo build** command resolves all dependencies of the project, downloads the missing dependencies, and compiles the project using the **rustc** compiler.

By default, the project is built and compiled in debug mode. This mode is used for normal development and debugging, when you need to compile the source code quickly and do not need the code optimization.

To build the project in release mode, run the **cargo** tool with the **--release** option as follows:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.45 'cargo build --release'
```

- For Red Hat Enterprise Linux 8:

```
$ cargo build --release
```

Compilation time is increased in this mode due to optimizations of the source code. As a result, the compiled binary will run faster. Use this mode to produce optimized artifacts suitable for release and production.

Example 2.2. Building a Rust project using Cargo

This example assumes that you have successfully created the Rust project **helloworld** according to Example 2.1, “Creating a Rust project using Cargo” .

Change to the directory **helloworld** and build the project:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.45 'cargo build'
   Compiling helloworld v0.1.0 (file:///home/vslavik/helloworld)
    Finished dev [unoptimized + debuginfo] target(s) in 0.51 secs
```

- For Red Hat Enterprise Linux 8:

```
$ cargo build
   Compiling helloworld v0.1.0 (file:///home/vslavik/helloworld)
    Finished dev [unoptimized + debuginfo] target(s) in 0.51 secs
```

Examine the result:

```
$ tree
.
├── Cargo.lock
├── Cargo.toml
├── src
│   └── main.rs
└── target
    └── debug
        ├── build
        ├── deps
        │   └── helloworld-b7c6fab39c2d17a7
        ├── examples
        ├── helloworld
        ├── helloworld.d
        ├── incremental
        └── native

8 directories, 6 files
```

A subdirectory structure has been created, starting with the directory **target**. Since the project was built in debug mode, the actual build output is contained in a further subdirectory **debug**. The actual resulting executable file is **target/debug/helloworld**.

NOTE

The **tree** tool is available from the default Red Hat Enterprise Linux repositories. To install it:

```
# yum install tree
```

2.3. VERIFYING THAT A RUST PROJECT COMPILES

To verify that a Rust program managed by **Cargo** can be built, on the command line change to the project directory and run the **cargo** tool as follows:

For {RHEL}{nbsp}7:

+

```
$ scl enable rust-toolset-1.45 'cargo check'
```

- For Red Hat Enterprise Linux 8:

```
$ cargo check
```

NOTE

Consider using the **cargo check** command instead of the **cargo build** command to verify the validity of a Rust program when you do not need to build an executable. The **cargo check** command is faster than a full project build using the **cargo build** command, because it does not generate the executable code.

By default, the project is checked in debug mode. To check the project in release mode, run the **cargo** tool with the **--release** option as follows:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.45 'cargo check --release'
```

- For Red Hat Enterprise Linux 8:

```
$ cargo check --release
```

Example 2.3. Verifying that a Rust project compiles with Cargo

This example assumes that you have successfully built the Rust project **helloworld** according to Example 2.2, “Building a Rust project using Cargo” .

Change to the directory **helloworld** and check the project:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.45 'cargo check'
   Compiling helloworld v0.1.0 (file:///home/vslavik/helloworld)
    Finished dev [unoptimized + debuginfo] target(s) in 0.5 secs
```

- For Red Hat Enterprise Linux 8:

```
$ cargo check
   Compiling helloworld v0.1.0 (file:///home/vslavik/helloworld)
    Finished dev [unoptimized + debuginfo] target(s) in 0.5 secs
```

The project is checked, with output similar to that of the **cargo build** command. However, the executable file is not generated. You can verify this by comparing the current time with the time stamp of the executable file:

```
$ date
Fri Oct 13 08:53:21 CEST 2017
$ ls -l target/debug/helloworld
-rwxrwxr-x. 2 vslavik vslavik 252624 Oct 13 08:48 target/debug/helloworld
```

2.4. RUNNING A RUST PROGRAM

To run a Rust program managed as a project by **Cargo** on the command line, change to the project directory and run the **cargo** tool as follows:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.45 'cargo run'
```

- For Red Hat Enterprise Linux 8:

```
$ cargo run
```

If the program has not been built yet, **Cargo** will run a build before running the program.

NOTE

Consider using **Cargo** to run a Rust program during development. It will correctly resolve the output path independently of the build mode.

By default, the project is built in debug mode. To build the project in release mode before running, run the **cargo** tool with the **--release** option as follows:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.45 'cargo run --release'
```

- For Red Hat Enterprise Linux 8:

```
$ cargo run --release
```

Example 2.4. Running a Rust program with Cargo

This example assumes that you have successfully built the Rust project **helloworld** according to Example 2.2, “Building a Rust project using Cargo” .

Change to the directory **helloworld** and run the project:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.45 'cargo run'
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/helloworld
Hello, world!
```

- For Red Hat Enterprise Linux 8:

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/helloworld
Hello, world!
```

Cargo first rebuilds the project, and then runs the resulting executable file.

In this example, there were no changes to the source code since last build. As a result, **Cargo** did not have to rebuild the executable file, but merely accepted it as current.

2.5. TESTING A RUST PROJECT

To run tests for a **Cargo** project on the command line, change to the project directory and run the **cargo** tool as follows:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.45 'cargo test'
```

- For Red Hat Enterprise Linux 8:

```
$ cargo test
```

By default, the project is tested in debug mode. To test the project in release mode, run the **cargo** tool with the **--release** option as follows:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.45 'cargo test --release'
```

- For Red Hat Enterprise Linux 8:

```
$ cargo test --release
```

Example 2.5. Testing a Rust project with Cargo

This example assumes that you have successfully built the Rust project **helloworld** according to Example 2.2, “Building a Rust project using Cargo” .

Change to the directory **helloworld**, and edit the file **src/main.rs** so that it contains the following source code:

```
fn main() {
    println!("Hello, world!");
}

#[test]
fn my_test() {
    assert_eq!(21+21, 42);
}
```

The function **my_test** marked as a test has been added.

Save the file, and run the test:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.45 'cargo test'
   Compiling helloworld v0.1.0 (file:///home/vslavik/Documentation/rusttest/helloworld)
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.26 secs
     Running target/debug/deps/helloworld-9dd6b83647b49aec

running 1 test
test my_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

- For Red Hat Enterprise Linux 8:

```
$ cargo test
   Compiling helloworld v0.1.0 (file:///home/vslavik/Documentation/rusttest/helloworld)
    Finished dev [unoptimized + debuginfo] target(s) in 0.26 secs
     Running target/debug/deps/helloworld-9dd6b83647b49aec

running 1 test
test my_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

Cargo first rebuilds the project, and then runs the tests found in the project. The **helloworld** project has passed the test **my_test** successfully.

2.6. CONFIGURING RUST PROJECT DEPENDENCIES

To specify dependencies for a **Cargo** project, edit the file **Cargo.toml** in the project directory. The section **[dependencies]** contains a list of the project dependencies. Each dependency is listed on a new line in the following format:

```
crate_name = version
```

Rust code packages are called crates.

Example 2.6. Adding a dependency to a Rust project and building it with Cargo

This example assumes that you have successfully built the Rust project **helloworld** according to Example 2.2, “Building a Rust project using Cargo” .

Change to the directory **helloworld** and edit the file **src/main.rs** so that it contains the following source code:

```
extern crate time;

fn main() {
    println!("Hello, world!");
    println!("Time is: {}", time::now().rfc822());
}
```

The code now requires an external crate **time**. Add this dependency to project configuration by editing the file **Cargo.toml** so that it contains the following code:

```
[package]
```

```
name = "helloworld"
version = "0.1.0"
authors = ["Your Name <yourname@example.com>"]

[dependencies]
time = "0.1"
```

Run the **cargo run** command to build the project and run the resulting executable file:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.45 'cargo run'
   Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading time v0.1.38
 Downloading libc v0.2.32
   Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running `target/debug/helloworld`
Hello, world!
Time is: Fri, 13 Oct 2017 11:08:57
```

- For Red Hat Enterprise Linux 8:

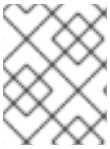
```
$ cargo run
   Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading time v0.1.38
 Downloading libc v0.2.32
   Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running `target/debug/helloworld`
Hello, world!
Time is: Fri, 13 Oct 2017 11:08:57
```

Cargo downloads the **time** crate and its dependencies (crate **libc**), stores them locally, builds all of the project source code including the dependency crates, and finally runs the resulting executable.

Additional resources

- Specifying Dependencies – official **Cargo** documentation.

2.7. BUILDING DOCUMENTATION FOR A RUST PROJECT



NOTE

Consider using the **cargo doc** command over **rustdoc**. The command **cargo doc** utilizes the **rustdoc** utility.

NOTE

cargo doc extracts documentation comments only for public functions, variables, and members.

Rust code can contain comments marked for extraction into documentation. The comments support the Markdown language.

To build project documentation using the **Cargo** tool, change to the project directory and run **cargo** tool:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.45 'cargo doc --no-deps'
```

- For Red Hat Enterprise Linux 8:

```
$ cargo doc --no-deps
```

This extracts documentation stored from the special comments in the source code of your project and writes the documentation in HTML format.

- Omit the **--no-deps** option to include dependencies in the generated documentation, including third-party libraries.
- Add the **--open** option to open the generated documentation in your browser.

Example 2.7. Building documentation for a Rust project with Cargo

This example assumes that you have successfully built the Rust project **helloworld** with dependencies, according to Example 2.6, "Adding a dependency to a Rust project and building it with Cargo".

Change to the directory **helloworld** and edit the file **src/main.rs** so that it contains the following source code:

```
//! This is a hello-world program.
extern crate time;

/// Prints a greeting to `stdout`.
pub fn print_output() {
    println!("Hello, world!");
    println!("Time is: {}", time::now().rfc822());
}

/// The program entry point.
fn main() {
    print_output();
}
```

The code now contains a public function **print_output()**. The whole **helloworld** program, the **print_output()** function, and the **main()** function have documentation comments.

Run the **cargo doc** command to build the project documentation:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.45 'cargo doc --no-deps'
 Documenting helloworld v0.1.0 (file:///home/vslavik/helloworld)
   Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
```

- For Red Hat Enterprise Linux 8:

```
$ cargo doc --no-deps
 Documenting helloworld v0.1.0 (file:///home/vslavik/helloworld)
   Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
```

Install the **tree** tool that is available in the default Red Hat Enterprise Linux repositories, if not already installed:

```
# yum install tree
```

Examine the result:

```
$ tree
.
├── Cargo.lock
├── Cargo.toml
├── src
│   └── main.rs
└── target
...
    └── doc
...
        ├── helloworld
        │   ├── fn.print_output.html
        │   ├── index.html
        │   ├── print_output.v.html
        │   └── sidebar-items.js
...
        └── src
            └── helloworld
                └── main.rs.html

12 directories, 32 files
```

Cargo builds the project documentation. To view the documentation, open the file **target/doc/helloworld/index.html** in your browser. The generated documentation does not contain any mention of the **main()** function, because it is not public.

Run the **cargo doc** command without the **--no-deps** option to build the project documentation, including the dependency libraries **time** and **libc**:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.45 'cargo doc'
 Documenting libc v0.2.32
 Documenting time v0.1.38
 Documenting helloworld v0.1.0 (file:///home/vslavik/helloworld)
   Finished dev [unoptimized + debuginfo] target(s) in 3.41 secs
```

- For Red Hat Enterprise Linux 8:

```
$ cargo doc
 Documenting libc v0.2.32
 Documenting time v0.1.38
```

```
   Documenting helloworld v0.1.0 (file:///home/vslavik/helloworld)
     Finished dev [unoptimized + debuginfo] target(s) in 3.41 secs
```

Examine the resulting directory structure with the **tree** command:

```
$ tree
...
92 directories, 11804 files
$ ls -d target/doc/*/
target/doc/helloworld/  target/doc/implementors/  target/doc/libc/  target/doc/src/  target/doc/time/
```

The resulting documentation now covers the dependency libraries **time** and **libc**, with each present as another subdirectory in the **target**/**doc**/ directory.

Additional resources

A detailed description of the **cargo doc** tool and its features is beyond the scope of this document. For more information, see the resources listed below.

- Making Useful Documentation Comments from the official Rust Programming Language documentation.

2.8. VENDORING RUST PROJECT DEPENDENCIES

Vendoring project dependencies means creating a local copy of the dependencies for offline redistribution and reuse. Vendored dependencies can be used by the **Cargo** build tool without any connection to the internet.

Note that the **cargo-vendor** package is included in **Cargo**, but there has been no change in the way it works.

Example 2.8. Vendoring Rust project dependencies

This example assumes that you have successfully built the Rust project **helloworld** with dependencies, according to Example 2.6, "Adding a dependency to a Rust project and building it with Cargo".

Change to the directory **helloworld** and run the **cargo vendor** command to vendor the project with dependencies:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.45 'cargo vendor'
 Downloading kernel32-sys v0.2.2
 Downloading redox_syscall v0.1.31
 Downloading winapi-build v0.1.1
 Downloading winapi v0.2.8
   Vendoring kernel32-sys v0.2.2 (/home/vslavik/.cargo/registry/src/github.com-
1ecc6299db9ec823/kernel32-sys-0.2.2) to vendor/kernel32-sys
   Vendoring libc v0.2.32 (/home/vslavik/.cargo/registry/src/github.com-
1ecc6299db9ec823/libc-0.2.32) to vendor/libc
   Vendoring redox_syscall v0.1.31 (/home/vslavik/.cargo/registry/src/github.com-
1ecc6299db9ec823/redox_syscall-0.1.31) to vendor/redox_syscall
   Vendoring time v0.1.38 (/home/vslavik/.cargo/registry/src/github.com-
1ecc6299db9ec823/time-0.1.38) to vendor/time
```

```
   Vendoring winapi v0.2.8 (/home/vslavik/.cargo/registry/src/github.com-
1ecc6299db9ec823/winapi-0.2.8) to vendor/winapi
   Vendoring winapi-build v0.1.1 (/home/vslavik/.cargo/registry/src/github.com-
1ecc6299db9ec823/winapi-build-0.1.1) to vendor/winapi-build
To use vendored sources, add this to your .cargo/config for this project:

    [source.crates-io]
    replace-with = "vendored-sources"

    [source.vendored-sources]
    directory = "/home/vslavik/helloworld/vendor"
```

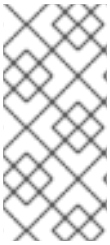
- For Red Hat Enterprise Linux 8:

```
$ cargo vendor
```

Examine the result:

```
$ ls
Cargo.lock  Cargo.toml  src  target  vendor
$ tree vendor
vendor
├── kernel32-sys
│   ├── build.rs
│   ├── Cargo.toml
│   ├── README.md
│   └── src
│       └── lib.rs
├── libc
│   ├── appveyor.yml
│   ├── Cargo.toml
...
75 directories, 319 files
```

The **vendor** directory contains copies of all the dependency crates needed to build the **helloworld** program. Note that the crates for building the project on the Windows operating system have been vendored, too, despite running this command on Red Hat Enterprise Linux.



NOTE

The **tree** tool is available from the default Red Hat Enterprise Linux repositories. To install it:

```
# yum install tree
```

2.9. ADDITIONAL RESOURCES

A detailed description of the **Cargo** tool and its features is beyond the scope of this document. For more information, see the resources listed below.

Cargo documentation

- *cargo*(1) – The manual page for the **cargo** tool provides detailed information on its usage. To display the manual page for the version included in Rust Toolset:
  - For Red Hat Enterprise Linux 7:

    ```
    $ scl enable rust-toolset-1.45 'man cargo'
    ```

  - For Red Hat Enterprise Linux 8:

    ```
    $ man cargo
    ```

- *Cargo, Rust's Package Manager* HTML book is provided as a package:
  - On Red Hat Enterprise Linux 7:

    ```
    # yum install rust-toolset-1.45-cargo-doc
    ```

    The HTML is available at **/opt/rh/rust-toolset-1.45/root/usr/share/doc/cargo/html/index.html**.

  - On Red Hat Enterprise Linux 8:

    ```
    # yum install cargo-doc
    ```

    The HTML is available at **/usr/share/doc/cargo/html/index.html**.

Online Cargo documentation

- Official Cargo Guide

See also

- Chapter 1, *Rust* – An overview of Rust Toolset and more information on how to install it on your system.

CHAPTER 3. RUSTFMT

The **rustfmt** tool provides automatic formatting of Rust source code.

3.1. INSTALLING RUSTFMT

Run the following command to install **rustfmt**:

- For Red Hat Enterprise Linux 7:

```
# yum install rust-toolset-1.45-rustfmt
```

- For Red Hat Enterprise Linux 8:

```
# yum install rustfmt
```

3.2. USING RUSTFMT AS A STANDALONE TOOL

To format a Rust source file and all its dependencies with the **rustfmt** tool:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.45 'rustfmt source-file'
```

- For Red Hat Enterprise Linux 8:

```
$ rustfmt source-file
```

Replace *source-file* with the path to a source file.

By default, **rustfmt** modifies the affected files in place without displaying details or creating backups. To change the behavior, use the **--write-mode *value*** option. For further details see the help message of **rustfmt**:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.45 'rustfmt --help'
```

- For Red Hat Enterprise Linux 8:

```
$ rustfmt --help
```

Additionally, **rustfmt** accepts standard input instead of a file and provides its output in standard output.

NOTE

You can execute any command using the **scl** utility on Red Hat Enterprise Linux 7, causing it to be run with the Rust binaries available. To use Rust Toolset on Red Hat Enterprise Linux 7 without a need to use **scl enable** with every command, run a shell session with:

```
$ scl enable rust-toolset-1.45 'bash'
```

3.3. USING RUSTFMT WITH CARGO

To format all source files in a cargo crate:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.45 'cargo fmt'
```

- For Red Hat Enterprise Linux 8:

```
$ cargo fmt
```

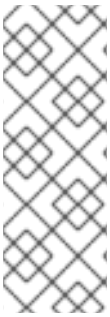
To change the **rustfmt** formatting options, create the configuration file **rustfmt.toml** in the project directory and supply the configuration there. For further details see the help message of **rustfmt**:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.45 'rustfmt --help'
```

- For Red Hat Enterprise Linux 8:

```
$ rustfmt --help
```



NOTE

You can execute any command using the **scl** utility on Red Hat Enterprise Linux 7, causing it to be run with the Rust binaries available. To use Rust Toolset on Red Hat Enterprise Linux 7 without a need to use **scl enable** with every command, run a shell session with:

```
$ scl enable rust-toolset-1.45 'bash'
```

3.4. ADDITIONAL RESOURCES

- Help message of **rustfmt**:
  - For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.45 'rustfmt --help'
```

  - For Red Hat Enterprise Linux 8:

```
$ rustfmt --help
```

- *Configuring Rustfmt* in **Configurations.md**:
  - Location in Red Hat Enterprise Linux 7:
    /opt/rh/rust-toolset-1.45/root/usr/share/doc/rust-toolset-1.45-rustfmt-1.45.1/Configurations.md
  - Location in Red Hat Enterprise Linux 8:
    /usr/share/doc/rustfmt/Configurations.md

CHAPTER 4. CONTAINER IMAGES WITH RUST TOOLSET

Rust Toolset is available as container images for Red Hat Enterprise Linux 7 and Red Hat Enterprise Linux 8. Container images can be downloaded from the Red Hat Container Registry.

4.1. IMAGE CONTENTS

The Red Hat Enterprise Linux 7 and Red Hat Enterprise Linux 8 container images provide content corresponding to the following packages:

Component	Version	Package
Rust	1.45.2	**rust-toolset-1.45-1.45.2**

4.2. ACCESSING THE IMAGES

To pull the image, run the following command as **root**:

- For the Red Hat Enterprise Linux 7 container images:

```
# podman pull registry.redhat.io/devtools/rust-toolset-rhel7
```

- For the Red Hat Enterprise Linux 8 container images:

```
# podman pull registry.redhat.io/rhel8/rust-toolset
```

4.3. USING AS BUILDER IMAGES WITH SOURCE-TO-IMAGE

The Rust Toolset container image is prepared for use as a Source-to-Image (S2I) builder image in Red Hat Enterprise Linux 7. Source-to-Image is not supported on Red Hat Enterprise Linux 8.

Example 4.1. Building a Rust application image using Source-to-Image

To build the **rust-test-app** from its GitHub repository, complete the following steps.

Prerequisites

- Run the following commands to prepare the builder image:

```
$ git clone https://github.com/openshift-s2i/s2i-rust.git
$ cd s2i-rust
$ make build
```

Procedure

Use the **s2i** tool to build the **rust-test-app** by running the following command:

```
s2i build ./examples/test-app/ devtools/rust-toolset-rhel7 rust-test-app
```

To run the image, type the following into the command line:

```
$ docker run rust-test-app
```

A locally available application image **rust-test-app** is built using the **Rust** Toolset container image.

To fully leverage the Rust as a S2I builder image, build custom images based on it, with modified S2I assemble scripts and further modifications to accomodate the particular application being built.

A detailed description of the Rust usage with Source-to-Image is beyond the scope of this document. For more information about Source-to-Image, see:

- *OpenShift Container Platform 4.5 Image Creation Guide* , OpenShift Container Platform-specific guidelines
- *Using Red Hat Software Collections Container Images* , Chapter 2. Using Source-to-Image (S2I).

4.4. ADDITIONAL RESOURCES

- Rust Toolset Container Images – entries in the Red Hat Container Registry
- Using Red Hat Software Collections Container Images

CHAPTER 5. CHANGES IN RUST 1.45.2 TOOLSET

Rust Toolset has been updated from version **1.43.0** to **1.45.2**. Notable changes include:

- The subcommand **cargo tree** for viewing dependencies is now included in **cargo**.
- Casting from floating point values to integers now produces a clamped cast. Previously, when a truncated floating point value was out of range for the target integer type the result was undefined behaviour of the compiler. Non-finite floating point values led to undefined behaviour as well. With this enhancement, finite values are clamped either to the minimum or the maximum range of the integer. Positive and negative infinity values are by default clamped to the maximum and minimum integer respectively, Not-a-Number(NaN) values to zero.
- Function-like procedural macros in expressions, patterns, and statements are now extended and stabilized.

For detailed instructions regarding usage, see the upstream Rust 1.44.0 Release Notes and Rust 1.45.0 Release Notes.