I want to install rustup in windows at D:\Programs\rustup.

Should I define environment variable RUSTUP\_HOME=D:\Programs\rustup and CARGO\_HOME=D:\Programs\rustup (Want cargo also at same place)

For linux I found this guidance:

curl https://sh.rustup.rs -sSf | sudo RUSTUP\_HOME=/opt/rustup sh -s -- -y

Yes, you're on the right track! Setting those environment variables before installing rustup will direct the installation to your desired location.

For Windows, here's what you should do:

**Set the environment variables first:**

cmd

set RUSTUP\_HOME=D:\Programs\rustup

set CARGO\_HOME=D:\Programs\rustup

Or in PowerShell:

powershell

$env:RUSTUP\_HOME = "D:\Programs\rustup"

$env:CARGO\_HOME = "D:\Programs\rustup"

**Then run the rustup installer:**

cmd

curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh -s -- -y

However, I'd recommend a slight modification to your setup. Instead of putting both rustup and cargo in the same directory, consider this structure:

cmd

set RUSTUP\_HOME=D:\Programs\rustup

set CARGO\_HOME=D:\Programs\cargo

This is cleaner because:

* RUSTUP\_HOME stores rustup's metadata, toolchains, and components
* CARGO\_HOME stores cargo's registry cache, git checkouts, and global cargo binaries

If you really want everything in one place, your original approach will work fine - rustup will create subdirectories to organize things properly.

**To make these environment variables permanent** (so they persist after closing the terminal), you can:

1. Use the System Properties dialog (search "environment variables" in Windows)
2. Or use PowerShell: [Environment]::SetEnvironmentVariable("RUSTUP\_HOME", "D:\Programs\rustup", "User")

After installation, you'll need to add the cargo bin directory to your PATH: D:\Programs\rustup\bin (or D:\Programs\cargo\bin if you separate them)

But these variables can be defined directly in environment variables via systems option in control panel.

Absolutely correct! That's actually the better approach for a permanent installation.

**Steps to set environment variables through System Properties:**

1. Right-click "This PC" or "Computer" → Properties
2. Click "Advanced system settings"
3. Click "Environment Variables"
4. Under "User variables" (or "System variables" if you want it system-wide), click "New"
5. Add:
   * Variable name: RUSTUP\_HOME
   * Variable value: D:\Programs\rustup
6. Add another:
   * Variable name: CARGO\_HOME
   * Variable value: D:\Programs\rustup (or D:\Programs\cargo if you prefer separation)

**Then simply run the rustup installer:** After setting those environment variables, just download and run the rustup installer normally:

* Go to <https://rustup.rs/>
* Download and run rustup-init.exe
* Or use: curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

The installer will automatically detect and use your custom RUSTUP\_HOME and CARGO\_HOME locations.

This method is much cleaner than setting temporary variables in the terminal, and the paths will be permanently configured for all future sessions. You'll just need to restart your terminal/command prompt after setting the environment variables for them to take effect.

I want to modify this:

curl --proto '=https' --tlsv1.2 -sSf [https://sh.rustup.rs](https://sh.rustup.rs/) | sh -s -- -y

to have a better control of which rustup version I want to install, etc. should I end like this:

curl --proto '=https' --tlsv1.2 -sSf [https://sh.rustup.rs](https://sh.rustup.rs/) | sh

Yes, removing the -s -- -y part is a good idea if you want more control over the installation process.

Here's what each part does:

**Your modified command:**

bash

curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

**What you removed:**

* -s -- passes the following arguments to the script
* -y automatically answers "yes" to all prompts (non-interactive mode)

**With your modification, the installer will:**

* Prompt you to choose installation options
* Ask about modifying your PATH
* Let you customize the installation directory (though your environment variables will override this)
* Allow you to select which components to install initially
* Give you a chance to review settings before proceeding

**The interactive installer will show options like:**

* Default installation vs. custom installation
* Which toolchain to install (stable, beta, nightly)
* Which targets and components to include
* Whether to modify shell profile

**If you want even more specific control, you can also:**

bash

*# Download the script first to examine it*

curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs -o rustup-init.sh

*# Then run it with specific options*

sh rustup-init.sh --default-toolchain stable --profile minimal

Your approach of removing -s -- -y is perfect for getting that interactive control while still using the official installer.

PS D:\Programs\Powershell> curl --proto '=https' --tlsv1.2 -sSf [https://sh.rustup.rs](https://sh.rustup.rs/) | sh sh: The term 'sh' is not recognized as a name of a cmdlet, function, script file, or executable program. Check the spelling of the name, or if a path was included, verify that the path is correct and try again. [General Feedback] The most similar commands are: > oh, si, sl, sp, sv

Ah, you're in PowerShell on Windows! The sh command isn't available by default on Windows PowerShell.

For Windows, you have a few better options:

**Option 1: Use the Windows executable directly (Recommended)**

powershell

Invoke-WebRequest -Uri "https://win.rustup.rs/x86\_64" -OutFile "rustup-init.exe"

.\rustup-init.exe

**Option 2: One-liner download and run**

powershell

Invoke-WebRequest -Uri "https://win.rustup.rs/x86\_64" -OutFile "rustup-init.exe"; .\rustup-init.exe

**Option 3: If you have Git Bash or WSL**

* Open Git Bash and use your original command
* Or use WSL (Windows Subsystem for Linux) if you have it installed

**Option 4: Direct download**

* Go to <https://rustup.rs/> in your browser
* Click "Download rustup-init.exe"
* Run the downloaded executable

The Windows executable (rustup-init.exe) will give you the same interactive installation options you wanted, and it will respect your RUSTUP\_HOME and CARGO\_HOME environment variables that you set through System Properties.

After running rustup-init.exe, it will walk you through the installation options interactively, just like the Unix shell script would.

Which crates are necessary for:

1. Interface with C, C++, Assembler and java
2. Interface with python
3. perl / pcre2
4. java script
5. Excel How to install them in rust?

Here are the essential crates for each category:

**1. Interface with C, C++, Assembly, and Java**

**C/C++:**

* No additional crates needed - Rust has built-in FFI support
* cc - for building C/C++ code from build scripts
* bindgen - automatically generate Rust bindings from C headers
* cbindgen - generate C headers from Rust code

**Assembly:**

* asm - inline assembly (built into Rust)
* nasm-rs - NASM assembler integration

**Java:**

* jni - Java Native Interface bindings
* java-locator - find Java installations

**2. Interface with Python**

* pyo3 - most popular, comprehensive Python bindings
* cpython - alternative Python FFI
* inline-python - embed Python code in Rust

**3. Perl/PCRE2**

* pcre2 - PCRE2 regular expression library
* regex - Rust's native regex (PCRE-compatible)
* onig - Oniguruma regex engine

**4. JavaScript**

* deno\_core - embed V8 JavaScript engine
* boa - pure Rust JavaScript engine
* quick-js - QuickJS engine bindings
* wasm-bindgen - for WebAssembly/JS interop

**5. Excel**

* calamine - read Excel/ODS files
* rust\_xlsxwriter - write Excel files
* simple\_excel\_writer - lightweight Excel writer
* office - comprehensive Office document handling

**How to Install Them**

**Add to your Cargo.toml:**

toml

[dependencies]

*# C/C++ integration*

cc = "1.0"

bindgen = "0.68"

*# Python*

pyo3 = { version = "0.20", features = ["auto-initialize"] }

*# Regex/PCRE2*

pcre2 = "0.2"

regex = "1.10"

*# JavaScript*

deno\_core = "0.236"

*# Excel*

calamine = "0.22"

rust\_xlsxwriter = "0.64"

[build-dependencies]

bindgen = "0.68"

cc = "1.0"

**Install via cargo:**

bash

*# Install specific crates*

cargo add pyo3

cargo add calamine

cargo add regex

cargo add jni

*# Or install multiple at once*

cargo add pyo3 calamine regex jni deno\_core

**Development dependencies (for build scripts):**

bash

cargo add --build bindgen cc

Most of these crates have additional system dependencies (like Python dev headers for PyO3), so check each crate's documentation for specific setup requirements.

in toml file I want all latest versions instead of hard coding

You can use flexible version specifiers in your Cargo.toml to automatically get the latest compatible versions:

**Option 1: Caret Requirements (Recommended)**

toml

[dependencies]

*# C/C++ integration*

cc = "\*"

bindgen = "\*"

*# Python*

pyo3 = { version = "\*", features = ["auto-initialize"] }

*# Regex/PCRE2*

pcre2 = "\*"

regex = "\*"

*# JavaScript*

deno\_core = "\*"

*# Excel*

calamine = "\*"

rust\_xlsxwriter = "\*"

[build-dependencies]

bindgen = "\*"

cc = "\*"

**Option 2: Just Omit Version (Cargo picks latest)**

toml

[dependencies]

cc = "\*"

bindgen = "\*"

pyo3 = { version = "\*", features = ["auto-initialize"] }

pcre2 = "\*"

regex = "\*"

deno\_core = "\*"

calamine = "\*"

rust\_xlsxwriter = "\*"

**Option 3: Using cargo add (automatically gets latest)**

Instead of manually editing TOML, use:

bash

cargo add cc bindgen pyo3 pcre2 regex deno\_core calamine rust\_xlsxwriter

cargo add --build bindgen cc

cargo add pyo3 --features auto-initialize

This automatically adds the latest versions with caret requirements like "^1.0".

**Understanding Version Specifiers:**

* "\*" - any version (gets latest)
* "^1.2" - compatible with 1.2.x (cargo add default)
* "~1.2" - 1.2.x only
* ">=1.2" - 1.2 or higher

**Note:** Using "\*" can potentially break builds if crates release breaking changes, but it ensures you always get the latest features and bug fixes. The caret notation (^1.2) is generally safer for production code.

When you run cargo update, it will fetch the latest versions that match your specifications.

Although Home variable is defined, why I get this error:

PS D:\Programs\Powershell> cargo add cc bindgen pyo3 pcre2 regex deno\_core calamine rust\_xlsxwriter error: could not find Cargo.toml in D:\Programs\Powershell or any parent directory

The error occurs because cargo add needs to be run from within a Rust project directory that contains a Cargo.toml file. You're currently in D:\Programs\Powershell, which isn't a Rust project.

**You need to create a Rust project first:**

**Option 1: Create a new project**

powershell

*# Create a new Rust project*

cargo new my\_rust\_project

cd my\_rust\_project

*# Now add dependencies*

cargo add cc bindgen pyo3 pcre2 regex deno\_core calamine rust\_xlsxwriter

**Option 2: Initialize in current directory**

powershell

*# Initialize a Rust project in current directory*

cargo init

*# Then add dependencies*

cargo add cc bindgen pyo3 pcre2 regex deno\_core calamine rust\_xlsxwriter

**Option 3: Navigate to existing project**

powershell

*# If you already have a Rust project somewhere*

cd path\to\your\rust\project

cargo add cc bindgen pyo3 pcre2 regex deno\_core calamine rust\_xlsxwriter

**What cargo init creates:**

* Cargo.toml - project manifest
* src/main.rs - main source file (for binary)
* Or src/lib.rs - library source file (if you use --lib flag)

**For a library project:**

powershell

cargo init --lib

The RUSTUP\_HOME and CARGO\_HOME environment variables you set control where Rust toolchains and cargo cache are stored, but they don't affect where your individual projects live - those can be anywhere on your system.

See errors. Guide:

PS D:\Programs\Powershell> cargo add cc bindgen pyo3 pcre2 regex deno\_core calamine rust\_xlsxwriter error: could not find Cargo.toml in D:\Programs\Powershell or any parent directory PS D:\Programs\Powershell> cargo new install\_essential\_Crates Creating binary (application) install\_essential\_Crates package warning: the name install\_essential\_Crates is not snake\_case or kebab-case which is recommended for package names, consider install\_essential\_crates note: see more Cargo.toml keys and their definitions at <https://doc.rust-lang.org/cargo/reference/manifest.html> PS D:\Programs\Powershell> cargo delete install\_essential\_Crates error: no such command: delete help: view all installed commands with cargo --list help: find a package to install delete with cargo search cargo-delete

I see the issues! Let me guide you through fixing them:

**1. Fix the package name and navigate to it:**

powershell

*# Remove the incorrectly named directory manually*

Remove-Item -Recurse -Force install\_essential\_Crates

*# Create with proper snake\_case name*

cargo new install\_essential\_crates

cd install\_essential\_crates

**2. Now add the dependencies:**

powershell

cargo add cc bindgen pyo3 pcre2 regex deno\_core calamine rust\_xlsxwriter

**Alternative: Create as library project**

If you want this as a library instead of a binary:

powershell

cargo new install\_essential\_crates --lib

cd install\_essential\_crates

cargo add cc bindgen pyo3 pcre2 regex deno\_core calamine rust\_xlsxwriter

**About the errors you encountered:**

**Package naming warning:** Rust recommends snake\_case or kebab-case for package names:

* ✅ Good: install\_essential\_crates or install-essential-crates
* ❌ Avoid: install\_essential\_Crates (mixed case)

**No cargo delete command:** Cargo doesn't have a built-in delete command. To remove projects:

* Use Remove-Item -Recurse -Force folder\_name (PowerShell)
* Or manually delete the folder in File Explorer

**Verify your setup:**

powershell

*# Check if you're in the right directory*

pwd

*# Should show: D:\Programs\Powershell\install\_essential\_crates*

*# Check that Cargo.toml exists*

ls Cargo.toml

*# Add dependencies*

cargo add cc bindgen pyo3 pcre2 regex deno\_core calamine rust\_xlsxwriter

This should work without errors now!

Which other crates Or packages you recommend to install?

Here are some essential crates I'd recommend for a well-rounded Rust development environment:

**Core Utilities**

bash

cargo add serde serde\_json *# Serialization/deserialization*

cargo add tokio --features full *# Async runtime*

cargo add clap --features derive *# Command-line parsing*

cargo add anyhow thiserror *# Error handling*

cargo add log env\_logger *# Logging*

**Data Structures & Algorithms**

bash

cargo add itertools *# Extended iterator methods*

cargo add rayon *# Data parallelism*

cargo add indexmap *# Ordered HashMap*

cargo add smallvec *# Stack-allocated vectors*

cargo add bytes *# Byte manipulation*

**Network & Web**

bash

cargo add reqwest --features json *# HTTP client*

cargo add axum *# Web framework (or actix-web, warp)*

cargo add hyper *# HTTP library*

cargo add url *# URL parsing*

cargo add uuid --features v4 *# UUID generation*

**Database & Storage**

bash

cargo add sqlx --features runtime-tokio-rustls,postgres,mysql,sqlite *# SQL toolkit*

cargo add diesel --features postgres,mysql,sqlite *# ORM*

cargo add redis *# Redis client*

cargo add rocksdb *# Embedded database*

**Development & Testing**

bash

cargo add --dev criterion *# Benchmarking*

cargo add --dev proptest *# Property-based testing*

cargo add --dev mockall *# Mocking*

cargo add --dev tempfile *# Temporary files for tests*

**Cryptography & Security**

bash

cargo add ring *# Cryptographic operations*

cargo add rand *# Random number generation*

cargo add sha2 *# SHA hashing*

cargo add base64 *# Base64 encoding/decoding*

**Text Processing**

bash

cargo add unicode-segmentation *# Unicode text segmentation*

cargo add lazy\_static *# Static variables*

cargo add once\_cell *# Lazy initialization*

**System & OS**

bash

cargo add crossbeam *# Concurrent data structures*

cargo add parking\_lot *# Fast synchronization primitives*

cargo add memmap2 *# Memory-mapped files*

cargo add walkdir *# Directory traversal*

**Date/Time**

bash

cargo add chrono --features serde *# Date and time*

cargo add time *# Alternative time library*

**Configuration & CLI**

bash

cargo add config *# Configuration management*

cargo add dirs *# Platform-specific directories*

cargo add indicatif *# Progress bars*

cargo add colored *# Terminal colors*

**Add them all at once:**

bash

*# Core essentials*

cargo add serde serde\_json tokio --features full

cargo add clap --features derive

cargo add anyhow thiserror log env\_logger

cargo add itertools rayon reqwest --features json

cargo add chrono --features serde

cargo add uuid --features v4

cargo add rand base64

*# Development tools*

cargo add --dev criterion proptest mockall tempfile

**Optional but useful global tools (install with cargo install):**

bash

cargo install cargo-watch *# Auto-rebuild on file changes*

cargo install cargo-expand *# Expand macros*

cargo install cargo-audit *# Security audit*

cargo install cargo-outdated *# Check for outdated dependencies*

cargo install ripgrep *# Fast text search*

cargo install fd-find *# Fast file finder*

Start with the core essentials, then add specific crates as your projects need them. This gives you a solid foundation for most Rust development scenarios.

error: Error calling dlltool 'dlltool.exe': program not found error: could not compile windows-strings (lib) due to 1 previous error warning: build failed, waiting for other jobs to finish... error: could not compile windows-result (lib) due to 1 previous error error: failed to compile cargo-watch v8.5.3, intermediate artifacts can be found at C:\Users\juhi\AppData\Local\Temp\cargo-installGR8hBI. To reuse those artifacts with a future compilation, set the environment variable CARGO\_TARGET\_DIR to that path.

This error occurs because you're missing the necessary build tools for compiling native Windows dependencies. The dlltool.exe is part of the GNU toolchain that's needed for some Rust crates on Windows.

**Solution: Install the required build tools**

**Option 1: Install Microsoft C++ Build Tools (Recommended)**

1. Download "Microsoft C++ Build Tools" from Microsoft's website
2. Or install Visual Studio Community with "Desktop development with C++" workload
3. This provides the Windows SDK and MSVC compiler

**Option 2: Install the complete Visual Studio**

* Visual Studio Community (free)
* Make sure to select "Desktop development with C++" workload during installation

**Option 3: Use the Rust toolchain installer approach**

powershell

*# Switch to the MSVC toolchain (recommended for Windows)*

rustup default stable-x86\_64-pc-windows-msvc

*# Or if you want to use GNU toolchain, install the full mingw-w64*

*# But this requires additional setup*

**Option 4: Install Git for Windows (includes some GNU tools)**

* Download Git for Windows which includes some build tools
* But this might not be sufficient for all crates

**Quick fix to try first:**

powershell

*# Make sure you're using the MSVC toolchain*

rustup show

rustup default stable-x86\_64-pc-windows-msvc

**After installing build tools, try again:**

powershell

cargo install cargo-watch

cargo install cargo-expand

cargo install cargo-audit

**If you continue having issues:**

powershell

*# Check what toolchains you have*

rustup toolchain list

*# Install MSVC toolchain explicitly if needed*

rustup toolchain install stable-x86\_64-pc-windows-msvc

rustup default stable-x86\_64-pc-windows-msvc

The MSVC toolchain is generally more reliable on Windows than the GNU toolchain, and doesn't require additional GNU tools like dlltool.exe.

**After fixing the toolchain, your cargo install commands should work properly.**

But I want to use Msys ucrt 64 - the reason I chose gnu toolchain while installing rust instead of msvc

I understand! You want to use MSYS2 UCRT64 with the GNU toolchain. The issue is that you need the complete MSYS2 environment with the proper development tools.

**Install MSYS2 UCRT64 development environment:**

**1. Install/Update MSYS2:**

* Download from <https://www.msys2.org/> if you don't have it
* Or update existing installation: pacman -Syu

**2. Install the complete development toolchain:** Open MSYS2 UCRT64 terminal and run:

bash

*# Update package database*

pacman -Syu

*# Install the complete development toolchain*

pacman -S mingw-w64-ucrt-x86\_64-toolchain

*# Install specific tools that might be missing*

pacman -S mingw-w64-ucrt-x86\_64-binutils

pacman -S mingw-w64-ucrt-x86\_64-gcc

pacman -S mingw-w64-ucrt-x86\_64-pkg-config

*# Optional but useful*

pacman -S base-devel

**3. Ensure MSYS2 is in your PATH:** Add these to your Windows environment variables (System Properties):

C:\msys64\ucrt64\bin

C:\msys64\usr\bin

**4. Verify the GNU toolchain is active:**

powershell

rustup show

*# Should show: stable-x86\_64-pc-windows-gnu (default)*

*# If not, switch to it:*

rustup default stable-x86\_64-pc-windows-gnu

**5. Verify dlltool is available:**

powershell

dlltool --version

*# Should show the GNU dlltool version*

**6. Now try installing cargo tools again:**

powershell

cargo install cargo-watch

cargo install cargo-expand

cargo install cargo-audit

**Alternative: Use MSYS2 terminal directly**

You can also run Rust commands directly from the MSYS2 UCRT64 terminal:

bash

*# From MSYS2 UCRT64 terminal*

cargo install cargo-watch

This approach gives you the full GNU development environment while keeping the performance benefits of UCRT64. The key is having the complete mingw-w64-ucrt-x86\_64-toolchain package installed.

But I already have these.

If you already have the MSYS2 UCRT64 toolchain installed, let's troubleshoot why dlltool.exe isn't being found:

**1. Verify dlltool is accessible:**

powershell

*# Check if dlltool is in PATH*

dlltool --version

*# Or try full path*

C:\msys64\ucrt64\bin\dlltool.exe --version

**2. Check your PATH environment variable:**

powershell

*# Check if MSYS2 paths are in your PATH*

echo $env:PATH

*# Look for: C:\msys64\ucrt64\bin and C:\msys64\usr\bin*

**3. Verify Rust is using the GNU toolchain:**

powershell

rustup show

rustc --version --verbose

*# Should show: host: x86\_64-pc-windows-gnu*

**4. Check what's actually missing:**

powershell

*# Try to see the full error with verbose output*

cargo install cargo-watch -v

**5. Common fixes:**

**If PATH is missing MSYS2:** Add these to your Windows environment variables (if not already there):

* C:\msys64\ucrt64\bin
* C:\msys64\usr\bin
* C:\msys64\mingw64\bin (sometimes needed)

**Restart PowerShell/Terminal** after PATH changes.

**If dlltool exists but still not found:**

powershell

*# Try setting explicit tool paths*

set CC=gcc

set AR=ar

set DLLTOOL=dlltool

cargo install cargo-watch

**6. Alternative: Install from MSYS2 terminal directly:**

bash

*# Open MSYS2 UCRT64 terminal*

cargo install cargo-watch

**7. Check for conflicting installations:**

powershell

*# Make sure no conflicting toolchains*

where dlltool

where gcc

*# Should point to MSYS2 locations*

What output do you get from dlltool --version and rustup show? This will help pinpoint the exact issue.

Because, I am running your recommended packages / crates from powershell and may be dlltools will be in bash shell under Msys. So should I run rustup from bash because I have defined full path to include windows path in Msys.

Exactly! You've identified the issue correctly. When you run cargo install from PowerShell, it may not have access to the MSYS2 tools even if they're in your Windows PATH.

**Solution: Use MSYS2 UCRT64 terminal for cargo operations**

**1. Open MSYS2 UCRT64 terminal** (not the regular MSYS2 terminal)

**2. Verify your setup:**

bash

*# Check if Rust is accessible*

rustup show

cargo --version

rustc --version

*# Check if build tools are available*

dlltool --version

gcc --version

**3. Install the cargo tools from MSYS2 UCRT64:**

bash

*# Core cargo tools*

cargo install cargo-watch

cargo install cargo-expand

cargo install cargo-audit

cargo install cargo-outdated

cargo install ripgrep

cargo install fd-find

**4. For your project dependencies, also use MSYS2 UCRT64:**

bash

*# Navigate to your project*

cd /d/Programs/Powershell/install\_essential\_crates

*# Add dependencies*

cargo add cc bindgen pyo3 pcre2 regex deno\_core calamine rust\_xlsxwriter

cargo add serde serde\_json tokio --features full

cargo add clap --features derive

*# ... etc*

**Why this works better:**

* MSYS2 UCRT64 provides a complete Unix-like environment
* All GNU tools (gcc, dlltool, ar, etc.) are directly available
* No PATH conflicts between Windows and MSYS2 tools
* Better compatibility with crates that need native compilation

**Optional: Set up aliases in MSYS2**

Add to your ~/.bashrc in MSYS2:

bash

alias ll='ls -la'

alias la='ls -A'

alias l='ls -CF'

You can still use PowerShell for regular Windows tasks, but use MSYS2 UCRT64 for Rust development when you need the GNU toolchain. This gives you the best of both worlds!

Although RUSTUP\_HOME and CARGO\_HOME have been defined, these are not accessible from Msys: D:\Programs\rustup (rustup home) and D:\Programs\cargo (cargo home). Should I define this in .bash\_profile becuse: rustup show

and others give error of not found in bash

Yes, exactly! MSYS2 doesn't automatically inherit Windows environment variables, so you need to define them in your MSYS2 shell profile.

**Solution: Add environment variables to MSYS2**

**1. Open MSYS2 UCRT64 terminal**

**2. Edit your shell profile:**

bash

*# Create/edit .bashrc (or .bash\_profile)*

nano ~/.bashrc

*# or use vim if you prefer: vim ~/.bashrc*

**3. Add these lines to ~/.bashrc:**

bash

*# Rust environment variables*

export RUSTUP\_HOME="/d/Programs/rustup"

export CARGO\_HOME="/d/Programs/cargo" *# or "/d/Programs/rustup" if you used same location*

*# Add cargo bin to PATH*

export PATH="$CARGO\_HOME/bin:$PATH"

*# Optional: Add rustup bin to PATH (usually not needed as cargo/bin contains everything)*

*# export PATH="$RUSTUP\_HOME/bin:$PATH"*

**4. Reload your shell profile:**

bash

source ~/.bashrc

**5. Verify the setup:**

bash

echo $RUSTUP\_HOME

echo $CARGO\_HOME

echo $PATH

*# Now these should work*

rustup show

cargo --version

rustc --version

**Important Notes:**

**Path format in MSYS2:**

* Windows: D:\Programs\rustup
* MSYS2: /d/Programs/rustup (lowercase drive letter, forward slashes)

**Alternative locations for the profile:**

* ~/.bashrc - runs for interactive shells
* ~/.bash\_profile - runs for login shells
* ~/.profile - runs for any POSIX shell

**If rustup wasn't installed with these paths initially:**

You might need to reinstall rustup from within MSYS2:

bash

*# Download and install rustup in MSYS2*

curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

*# Choose option 2 (Customize installation) to verify paths*

After setting up the environment variables, all your cargo operations should work properly from the MSYS2 UCRT64 terminal.

After defining:

juhi@LAPTOP-BGI3SJP4 MSYS ~ $ rustup show Default host: x86\_64-pc-windows-gnu rustup home: D:/Programs/rustup installed toolchains -------------------- nightly-x86\_64-pc-windows-gnu (active, default) active toolchain ---------------- name: nightly-x86\_64-pc-windows-gnu active because: it's the default toolchain installed targets: x86\_64-pc-windows-gnu (venv) juhi@LAPTOP-BGI3SJP4 MSYS ~ $ rustc --version rustc 1.91.0-nightly (46c219bd2 2025-08-22) (venv) juhi@LAPTOP-BGI3SJP4 MSYS ~ $ cargo --version cargo 1.91.0-nightly (71eb84f21 2025-08-17)

Perfect! Your Rust setup is working correctly in MSYS2. You have the nightly GNU toolchain installed and it's properly recognizing your custom paths.

Now you can install the cargo tools and crates without issues:

**Install cargo tools:**

bash

cargo install cargo-watch

cargo install cargo-expand

cargo install cargo-audit

cargo install cargo-outdated

cargo install ripgrep

cargo install fd-find

**Navigate to your project and add dependencies:**

bash

*# Go to your project (adjust path as needed)*

cd /d/Programs/Powershell/install\_essential\_crates

*# Add all the dependencies we discussed*

cargo add cc bindgen pyo3 pcre2 regex deno\_core calamine rust\_xlsxwriter

*# Core essentials*

cargo add serde serde\_json tokio --features full

cargo add clap --features derive

cargo add anyhow thiserror log env\_logger

cargo add itertools rayon reqwest --features json

cargo add chrono --features serde

cargo add uuid --features v4

cargo add rand base64

*# Development tools*

cargo add --dev criterion proptest mockall tempfile

**Build/test your project:**

bash

*# Check that everything compiles*

cargo check

*# Build the project*

cargo build

*# Run tests*

cargo test

**Notes:**

* You're using **nightly** toolchain, which gives you access to the latest Rust features
* All the tools will be installed to D:/Programs/cargo/bin
* The compilation should work smoothly now with the complete MSYS2 UCRT64 environment
* If you want to switch to stable: rustup default stable-x86\_64-pc-windows-gnu (but nightly is fine for development)

Your setup is now properly configured for Rust development with the GNU toolchain on Windows!

Why this in bash in Msys: error: feature json must be qualified by the dependency it's being activated for, like itertools/json, rayon/json, reqwest/json

This error occurs because multiple crates in the command have the potential for a json feature, and cargo can't determine which one you want to enable it for.

**Fix: Specify which crate should have the json feature**

Instead of:

bash

cargo add itertools rayon reqwest --features json

Use this:

bash

*# Add them separately with specific features*

cargo add itertools rayon

cargo add reqwest --features json

*# Or specify explicitly which crate gets which feature*

cargo add itertools rayon reqwest --features reqwest/json

**Better approach - add them individually:**

bash

*# Core utilities*

cargo add serde serde\_json

cargo add tokio --features full

cargo add clap --features derive

cargo add anyhow thiserror log env\_logger

*# Data structures & algorithms*

cargo add itertools rayon

*# Network & web*

cargo add reqwest --features json

cargo add uuid --features v4

*# Other essentials*

cargo add chrono --features serde

cargo add rand base64

*# Your original crates*

cargo add cc bindgen pyo3 pcre2 regex deno\_core calamine rust\_xlsxwriter

*# Development tools*

cargo add --dev criterion proptest mockall tempfile

**Alternative - specify features explicitly:**

bash

cargo add itertools rayon reqwest --features reqwest/json

cargo add chrono uuid --features chrono/serde,uuid/v4

The key is that when multiple crates are added in one command, you need to specify which crate each feature belongs to using the crate\_name/feature\_name syntax.

This should resolve the error and properly add all your dependencies!

juhi@LAPTOP-BGI3SJP4 MSYS /d/Programs/rustup/install\_essential\_crates $ cargo add itertools rayon reqwest --features json error: feature json must be qualified by the dependency it's being activated for, like itertools/json, rayon/json, reqwest/json

You need to qualify which crate the json feature belongs to. Use this syntax:

bash

cargo add itertools rayon reqwest --features reqwest/json

Or add them separately (which is often clearer):

bash

cargo add itertools rayon

cargo add reqwest --features json

**Complete corrected commands:**

bash

*# Add them separately for clarity*

cargo add itertools rayon

cargo add reqwest --features json

cargo add chrono --features serde

cargo add uuid --features v4

cargo add tokio --features full

cargo add clap --features derive

*# Or if you want to do multiple at once, qualify each feature:*

cargo add chrono uuid --features chrono/serde,uuid/v4

**The complete sequence:**

bash

*# Core essentials*

cargo add serde serde\_json anyhow thiserror log env\_logger

cargo add tokio --features full

cargo add clap --features derive

cargo add itertools rayon

cargo add reqwest --features json

cargo add chrono --features serde

cargo add uuid --features v4

cargo add rand base64

*# Your original crates*

cargo add cc bindgen pyo3 pcre2 regex deno\_core calamine rust\_xlsxwriter

*# Development tools*

cargo add --dev criterion proptest mockall tempfile

The key rule: when adding multiple crates with --features, you must specify which crate each feature belongs to using crate\_name/feature\_name syntax.

I could install these without changing to: cd /d/Programs/Powershell/install\_essential\_crates

cargo install cargo-watch cargo install cargo-expand cargo install cargo-audit cargo install cargo-outdated cargo install ripgrep cargo install fd-find

Should I do it again or will it be duplication?

No need to install them again! The cargo install command installs global tools that are available system-wide, regardless of which directory you run the command from.

**What you installed are global tools:**

* cargo-watch - monitors files and rebuilds automatically
* cargo-expand - expands Rust macros
* cargo-audit - security vulnerability scanner
* cargo-outdated - checks for outdated dependencies
* ripgrep - fast text search tool
* fd-find - fast file finder

These are installed once to $CARGO\_HOME/bin (which is D:/Programs/cargo/bin in your case) and are available from any directory.

**To verify they're installed:**

bash

*# Check what's installed globally*

cargo install --list

*# Test that they work*

cargo-watch --version

cargo-expand --version

rg --version

fd --version

**What you need to do in your project directory:**

bash

*# Navigate to your project*

cd /d/Programs/Powershell/install\_essential\_crates

*# Add project dependencies (not global tools)*

cargo add serde serde\_json anyhow thiserror

cargo add tokio --features full

cargo add reqwest --features json

*# ... etc*

**The difference:**

* cargo install = global tools (install once, use anywhere)
* cargo add = project dependencies (must be in project directory, adds to Cargo.toml)

So you're all set with the global tools! Now just focus on adding the dependencies to your specific project.

Still don't get it which ones to install after changing to above folder and which ones are global?

Let me clarify the difference:

**GLOBAL TOOLS (already installed - don't repeat):**

These are command-line utilities you install once and use anywhere:

**✅ Already installed globally (don't install again):**

bash

cargo install cargo-watch *# File watching tool*

cargo install cargo-expand *# Macro expansion tool*

cargo install cargo-audit *# Security audit tool*

cargo install cargo-outdated *# Check outdated deps*

cargo install ripgrep *# Fast text search (rg command)*

cargo install fd-find *# Fast file finder (fd command)*

**PROJECT DEPENDENCIES (need to install in project folder):**

These are libraries your Rust code imports and uses:

**🔄 Need to install in your project directory:**

bash

*# First, navigate to your project*

cd /d/Programs/Powershell/install\_essential\_crates

*# Then add these LIBRARIES (not tools):*

cargo add cc bindgen pyo3 pcre2 regex deno\_core calamine rust\_xlsxwriter

cargo add serde serde\_json anyhow thiserror log env\_logger

cargo add tokio --features full

cargo add clap --features derive

cargo add itertools rayon

cargo add reqwest --features json

cargo add chrono --features serde

cargo add uuid --features v4

cargo add rand base64

*# Development dependencies*

cargo add --dev criterion proptest mockall tempfile

**How to tell the difference:**

* **Global tools**: Command-line programs you run (like cargo-watch, rg, fd)
* **Project dependencies**: Libraries you use in your Rust code (like serde, tokio, reqwest)

The global tools are installed to D:/Programs/cargo/bin and work from anywhere. The project dependencies are added to your Cargo.toml file and are specific to each project.

Suppose I run rg - then will it access cargo because msys also has fgrep (or something similar) which runs using rg.

Good question! There can indeed be conflicts between tools with similar names.

**Checking what rg points to:**

bash

*# Check which rg command will run*

which rg

type rg

*# Check if there are multiple rg commands available*

whereis rg

**Potential conflicts:**

**MSYS2 might have:**

* Native MSYS2 packages that also use rg
* Other grep-like tools

**Cargo installed:**

* ripgrep which provides the rg command

**To avoid conflicts:**

**Option 1: Check the version to confirm it's ripgrep:**

bash

rg --version

*# Should show: ripgrep x.x.x*

**Option 2: Use full path if needed:**

bash

*# Use the cargo-installed version explicitly*

$CARGO\_HOME/bin/rg --version

*# or*

/d/Programs/cargo/bin/rg --version

**Option 3: Check your PATH order:**

bash

echo $PATH

The first directory in PATH containing rg will be used.

**Recommended PATH order in ~/.bashrc:**

bash

*# Put cargo bin first to prioritize cargo-installed tools*

export PATH="$CARGO\_HOME/bin:$PATH"

*# Then other paths...*

**Quick test:**

bash

*# Test that it's actually ripgrep*

rg --version

*# If it says "ripgrep", you're using the right one*

*# If it gives an error or different output, there's a conflict*

Most likely, since you installed ripgrep via cargo and it's in your PATH, rg will use the cargo version. But it's good to verify with rg --version to be sure!

See because of different priorities rg is accessed from rustup instead of msys native ripgrep:

$ which rg /d/Programs/cargo/bin/rg (venv) juhi@LAPTOP-BGI3SJP4 MSYS /d/Programs/rustup/install\_essential\_crates $ echo $PATH /home/juhi/venv/Scripts:/d/Programs/rustup/bin:/d/Programs/cargo/bin:/usr/local/bin:/usr/bin:/bin:/opt/bin:/usr/bin/site\_perl:/usr/bin/vendor\_perl:/usr/bin/core\_perl:/ucrt64/bin:/ucrt64/lib:/ucrt64/include:/ucrt64/share:/ucrt64/local:/ucrt64/libexec:/ucrt64/sbin:/ucrt64/var:/ucrt64/x86\_64-w64-mingw32:/home/juhi/venv/Scripts:/c/ffmpeg/bin:/d/Programs/Python/Scripts:/d/Programs/Python:/d/Programs/Powershell:/d/Programs/7-zip:/d/programs/ffmpeg/bin:/d/Programs/gvim:/c/Users/juhi/AppData/Local/Microsoft/WindowsApps:/c/Users/juhi/AppData/Local/Programs/Python/Launcher:/c/windows/system32:/c/windows:/c/windows/System32/Wbem:/c/windows/System32/WindowsPowerShell/v1.0:/c/windows/System32/OpenSSH:/ucrt64/bin:/ucrt64/lib:/ucrt64/include:/ucrt64/share:/ucrt64/local:/ucrt64/libexec:/ucrt64/sbin:/ucrt64/var:/ucrt64/x86\_64-w64-mingw32

Perfect! Your PATH setup is working correctly. The rg command is being found from /d/Programs/cargo/bin/rg, which means you're using the **cargo-installed ripgrep** (the Rust version), not the MSYS2 native one.

**This is actually the preferred behavior because:**

1. **Cargo-installed ripgrep is usually newer** and more feature-rich
2. **It's the "official" ripgrep** written in Rust by the same author
3. **PATH priority is working correctly**: /d/Programs/cargo/bin comes before the MSYS2 paths

**Your PATH priority order:**

bash

1. /home/juhi/venv/Scripts *# Python virtual env (active)*

2. /d/Programs/rustup/bin *# Rustup tools*

3. /d/Programs/cargo/bin *# Cargo-installed tools ← rg found here*

4. /usr/local/bin:/usr/bin/... *# MSYS2 system tools*

5. /ucrt64/bin/... *# MSYS2 UCRT64 tools*

**To verify you have the right ripgrep:**

bash

rg --version

*# Should show something like: ripgrep 14.x.x*

**If you ever want to use the MSYS2 version instead:**

bash

*# Use full path to MSYS2 version*

/usr/bin/rg --version *# if it exists in MSYS2*

*# Or temporarily modify PATH for one command*

PATH="/usr/bin:$PATH" rg --version

But your current setup is ideal - you're getting the best tools (Rust-based ripgrep) while maintaining access to MSYS2 development tools. The PATH ordering ensures cargo tools take priority, which is exactly what you want for a Rust development environment.

It's not /usr/bin but /ucrt64/bin in my case:

$ /usr/bin/rg bash: /usr/bin/rg: No such file or directory (venv) juhi@LAPTOP-BGI3SJP4 MSYS /d/Programs/rustup/install\_essential\_crates $ /ucrt64/bin/rg rg: ripgrep requires at least one pattern to execute a search

You're absolutely right! In MSYS2 UCRT64, the tools are in /ucrt64/bin, not /usr/bin.

**Both versions are actually ripgrep:**

**Cargo-installed ripgrep:**

bash

/d/Programs/cargo/bin/rg --version

**MSYS2 UCRT64 ripgrep:**

bash

/ucrt64/bin/rg --version

**Compare the versions:**

bash

*# Check which version is newer/different*

echo "Cargo version:"

/d/Programs/cargo/bin/rg --version

echo "MSYS2 UCRT64 version:"

/ucrt64/bin/rg --version

**Your current setup is still optimal:**

Since both are ripgrep (the same tool), but the cargo-installed version is:

1. **More up-to-date** (you installed it recently)
2. **Directly managed** by your Rust toolchain
3. **Found first** in your PATH

**If you want to check what other tools might have duplicates:**

bash

*# See what's in MSYS2 UCRT64 that might conflict*

ls /ucrt64/bin/ | grep -E "(rg|fd|cargo)"

*# Compare with your cargo tools*

ls /d/Programs/cargo/bin/

The good news is that since both are the same tool (ripgrep), there's no functional difference - you're just using the newer cargo-installed version by default, which is perfect for your Rust development setup.

There is a catch - Cargo version no pcre2!:

$ echo "Cargo version:" /d/Programs/cargo/bin/rg --version Cargo version: ripgrep 14.1.1 features:-pcre2 simd(compile):+SSE2,-SSSE3,-AVX2 simd(runtime):+SSE2,+SSSE3,+AVX2 PCRE2 is not available in this build of ripgrep. (venv) juhi@LAPTOP-BGI3SJP4 MSYS /d/Programs/rustup/install\_essential\_crates $ echo "MSYS2 UCRT64 version:" /ucrt64/bin/rg --version MSYS2 UCRT64 version: ripgrep 14.1.1 features:+pcre2 simd(compile):+SSE2,-SSSE3,-AVX2 simd(runtime):+SSE2,+SSSE3,+AVX2 PCRE2 10.43 is available (JIT is available)

Excellent catch! This is a significant difference. The MSYS2 version has **PCRE2 support** while the cargo-installed version doesn't.

**PCRE2 is important for:**

* **Lookahead/lookbehind assertions** ((?=...), (?<=...))
* **Advanced regex features** not available in Rust's default regex engine
* **Perl-compatible regular expressions**

**Solutions:**

**Option 1: Reinstall ripgrep with PCRE2 support**

bash

*# Uninstall the cargo version*

cargo uninstall ripgrep

*# Reinstall with PCRE2 feature*

cargo install ripgrep --features 'pcre2'

**Option 2: Use the MSYS2 version by adjusting PATH** Edit your ~/.bashrc to prioritize MSYS2 tools for specific cases:

bash

*# Put UCRT64 bin before cargo bin for tools that need native features*

export PATH="/ucrt64/bin:$CARGO\_HOME/bin:$PATH"

**Option 3: Create an alias for the MSYS2 version**

bash

*# Add to ~/.bashrc*

alias rg-pcre='/ucrt64/bin/rg'

**Option 4: Use both strategically**

* Use MSYS2 rg for complex regex patterns
* Keep cargo rg for general use

**I recommend Option 1:**

bash

cargo uninstall ripgrep

cargo install ripgrep --features 'pcre2'

Then verify:

bash

rg --version

*# Should now show: features:+pcre2*

This gives you the best of both worlds: latest version managed by cargo WITH PCRE2 support!

See for other duplicates:

$ ls /ucrt64/bin/ | grep -E "(rg|fd|cargo)" avifdec.exe fdp.exe gif2rgb.exe ld.bfd.exe libboost\_prg\_exec\_monitor-mt.dll lli-child-target.exe llvm-dwarfdump.exe llvm-profdata.exe msgmerge.exe pal2rgb.exe pdfdetach.exe plmerge.exe rg.exe rga-fzf-open.exe rga-fzf.exe rga-preproc.exe rga.exe sfdp.exe tiff2rgba.exe tiffdither.exe tiffdump.exe ttfdump.exe (venv) juhi@LAPTOP-BGI3SJP4 MSYS /d/Programs/rustup/install\_essential\_crates $ ls /d/Programs/cargo/bin/ cargo-clippy.exe cargo-miri.exe rg.exe rust-gdb.exe rustc.exe rustup.exe cargo-expand.exe cargo.exe rls.exe rust-gdbgui.exe rustdoc.exe cargo-fmt.exe clippy-driver.exe rust-analyzer.exe rust-lldb.exe rustfmt.exe

Great analysis! Here's what I see:

**Duplicates found:**

* **rg.exe** - exists in both (we already discussed this)

**MSYS2 has additional related tools:**

* **rga.exe** - ripgrep-all (searches inside PDFs, archives, etc.)
* **rga-\*.exe** - ripgrep-all helper tools
* **fdp.exe** - not the same as fd (this is from Graphviz)

**Missing tools you installed:**

Your cargo installations of fd-find and others should create executables. Let's check:

bash

*# Check what cargo thinks is installed*

cargo install --list

*# Look for fd specifically (might be named differently)*

ls /d/Programs/cargo/bin/ | grep fd

which fd

**The fd tool might be named differently:**

bash

*# Try these commands to find fd*

fd --version

fd.exe --version

which fd

If fd isn't found, you might need to reinstall it:

bash

cargo install fd-find

**Recommendation for ripgrep:**

Since MSYS2 has the superior version with PCRE2, and some additional tools like rga (ripgrep-all), consider:

**Option A: Use MSYS2 versions by adjusting PATH priority:**

bash

*# Edit ~/.bashrc to put UCRT64 first for these tools*

export PATH="/ucrt64/bin:$CARGO\_HOME/bin:$PATH"

**Option B: Keep cargo versions but reinstall with features:**

bash

cargo uninstall ripgrep

cargo install ripgrep --features 'pcre2'

What does cargo install --list show for your installed tools?

Are there no equivalents in rustup for these:

MSYS2 has additional related tools: \* rga.exe - ripgrep-all (searches inside PDFs, archives, etc.) \* rga-\*.exe - ripgrep-all helper tools \* fdp.exe - not the same as fd (this is from Graphviz)

You're right to ask! Let me clarify:

**Rust/Cargo equivalents:**

**For rga.exe (ripgrep-all):**

bash

*# YES - there is a Rust equivalent!*

cargo install ripgrep-all

*# This installs rga and its helper tools written in Rust*

**For Graphviz tools like fdp.exe:**

bash

*# NO direct Rust equivalent for the full Graphviz suite*

*# But there are some Rust alternatives:*

cargo install graphviz-rust *# Rust bindings*

cargo install dot-graph *# Simple graph generation*

However, Graphviz is a mature, specialized tool ecosystem. Most people use the native Graphviz installation rather than Rust alternatives.

**Check what you actually have:**

bash

*# See if you already have rga from MSYS2*

/ucrt64/bin/rga --version

*# Check if cargo version exists*

cargo install --list | grep ripgrep

**My recommendation:**

**For ripgrep ecosystem:**

bash

*# Install the full ripgrep suite via cargo for consistency*

cargo uninstall ripgrep *# remove the one without pcre2*

cargo install ripgrep --features 'pcre2'

cargo install ripgrep-all

**For Graphviz:** Keep using the MSYS2 version (fdp.exe, dot.exe, etc.) - it's the standard implementation and more complete.

This gives you:

* Rust-managed ripgrep tools (with PCRE2)
* Native Graphviz tools from MSYS2
* Best of both ecosystems

Can you give commands with recommended features like pcre2 and other you think are important for rustup?

MSYS2 has additional related tools: \* rga.exe - ripgrep-all (searches inside PDFs, archives, etc.) \* rga-\*.exe - ripgrep-all helper tools \* fdp.exe - not the same as fd (this is from Graphviz)

Here are the recommended cargo install commands with important features:

**Core Search & File Tools (with features):**

bash

*# Ripgrep with PCRE2 support for advanced regex*

cargo install ripgrep --features 'pcre2'

*# Ripgrep-all for searching inside archives, PDFs, etc.*

cargo install ripgrep-all

*# fd-find (fast file finder)*

cargo install fd-find

*# bat (better cat with syntax highlighting)*

cargo install bat

*# exa/eza (better ls)*

cargo install eza

**Development Tools (with useful features):**

bash

*# Cargo development tools*

cargo install cargo-watch

cargo install cargo-expand

cargo install cargo-audit

cargo install cargo-outdated

cargo install cargo-tree

cargo install cargo-edit

*# tokei (count lines of code)*

cargo install tokei

*# hyperfine (benchmarking tool)*

cargo install hyperfine

*# dust (better du)*

cargo install du-dust

*# procs (better ps)*

cargo install procs

**System & Utility Tools:**

bash

*# bottom (system monitor, better htop)*

cargo install bottom

*# starship (customizable prompt)*

cargo install starship

*# zoxide (smart cd)*

cargo install zoxide

*# delta (better git diff)*

cargo install git-delta

*# Just (command runner)*

cargo install just

**Text Processing:**

bash

*# sd (better sed)*

cargo install sd

*# hexyl (hex viewer)*

cargo install hexyl

*# xsv (CSV toolkit)*

cargo install xsv

**Network Tools:**

bash

*# xh (better curl/httpie)*

cargo install xh

*# bandwhich (network monitor)*

cargo install bandwhich

**Install all recommended at once:**

bash

*# Core essentials with features*

cargo install ripgrep --features 'pcre2'

cargo install ripgrep-all fd-find bat eza

*# Development tools*

cargo install cargo-watch cargo-expand cargo-audit cargo-outdated cargo-tree cargo-edit tokei hyperfine

*# System utilities*

cargo install du-dust procs bottom starship zoxide git-delta just

*# Text processing*

cargo install sd hexyl xsv

*# Network*

cargo install xh

These tools will give you a comprehensive Rust-based command-line toolkit that's often faster and more user-friendly than traditional Unix tools, while maintaining the advanced features you need.

$ cargo install ripgrep-all Updating crates.io index error: could not find ripgrep-all in registry crates-io with version \*

You're right! The crate name is different. It's rga (not ripgrep-all):

bash

*# Correct command for ripgrep-all*

cargo install rga

**Here are the corrected commands with proper crate names:**

**Core search tools:**

bash

cargo install ripgrep --features 'pcre2'

cargo install rga *# ripgrep-all*

cargo install fd-find

cargo install bat

cargo install eza *# modern ls replacement*

**Development tools:**

bash

cargo install cargo-watch cargo-expand cargo-audit cargo-outdated

cargo install tokei *# count lines of code*

cargo install hyperfine *# benchmarking*

**System utilities:**

bash

cargo install dust *# better du (not du-dust)*

cargo install procs

cargo install bottom *# system monitor*

cargo install starship *# shell prompt*

cargo install zoxide *# smart cd*

cargo install git-delta *# better git diff*

**Text processing:**

bash

cargo install sd *# better sed*

cargo install hexyl *# hex viewer*

cargo install xsv *# CSV toolkit*

**Network:**

bash

cargo install xh *# better curl*

**Verify rga installation:**

bash

rga --version

*# Should show ripgrep-all version info*

Thanks for catching that error! The actual crate name is rga, not ripgrep-all.

$ cargo install ripgrep --features 'pcre2' cargo install rga # ripgrep-all cargo install fd-find cargo install bat cargo install eza # modern ls replacement Updating crates.io index Ignored package ripgrep v14.1.1 is already installed, use --force to override Updating crates.io index Downloaded rga v0.5.9 Downloaded 1 crate (14.9KiB) in 1.45s Installing rga v0.5.9 Updating crates.io index error: failed to compile rga v0.5.9, intermediate artifacts can be found at D:\Programs\msys64\tmp\cargo-installtdMU4p. To reuse those artifacts with a future compilation, set the environment variable CARGO\_TARGET\_DIR to that path.

Two issues here:

**1. Ripgrep without PCRE2 features**

The ripgrep installation was ignored because it's already installed without PCRE2. You need to force reinstall:

bash

cargo install ripgrep --features 'pcre2' --force

**2. RGA compilation failed**

This is likely due to missing system dependencies. RGA needs additional libraries to handle various file formats.

**Install RGA dependencies in MSYS2:**

bash

*# Install required libraries for rga*

pacman -S mingw-w64-ucrt-x86\_64-poppler

pacman -S mingw-w64-ucrt-x86\_64-tesseract-ocr

pacman -S mingw-w64-ucrt-x86\_64-ffmpeg

pacman -S mingw-w64-ucrt-x86\_64-pandoc

**Then try installing RGA again:**

bash

cargo install rga --force

**Alternative: Use the MSYS2 version of RGA**

Since MSYS2 already has rga.exe working, you might want to just use that:

bash

*# Test the MSYS2 version*

/ucrt64/bin/rga --version

**Complete corrected sequence:**

bash

*# Force reinstall ripgrep with PCRE2*

cargo install ripgrep --features 'pcre2' --force

*# Install the working tools first*

cargo install fd-find bat eza

*# Try rga after installing dependencies, or just use MSYS2 version*

*# /ucrt64/bin/rga (already available)*

The MSYS2 version of RGA might actually be more reliable since it's compiled against the system libraries properly.

error: target not found: mingw-w64-ucrt-x86\_64-pandoc

The pandoc package name is different. Let's fix the dependencies:

bash

*# Install available RGA dependencies*

pacman -S mingw-w64-ucrt-x86\_64-poppler

pacman -S mingw-w64-ucrt-x86\_64-tesseract-ocr

pacman -S mingw-w64-ucrt-x86\_64-ffmpeg

*# For pandoc, use the base package (not mingw-w64 version)*

pacman -S pandoc

**Or search for the correct pandoc package:**

bash

*# Search for pandoc packages*

pacman -Ss pandoc

**Then try installing RGA:**

bash

cargo install rga --force

**Alternative approach - check what's actually needed:**

**1. Try installing RGA with minimal dependencies first:**

bash

*# Some dependencies might be optional*

cargo install rga --no-default-features

**2. Or just use the MSYS2 version that's already working:**

bash

*# Test the existing MSYS2 version*

/ucrt64/bin/rga --version

/ucrt64/bin/rga --help

**3. Check what RGA actually needs:**

bash

*# Look at the build error details - it will tell you what's missing*

cargo install rga -v

The MSYS2 version might be the better choice since it's already compiled with all the right dependencies. You can create an alias if needed:

bash

*# Add to ~/.bashrc if you want 'rga' to use the MSYS2 version*

alias rga='/ucrt64/bin/rga'

$ cargo install rga --force Updating crates.io index Installing rga v0.5.9 Updating crates.io index error: failed to compile rga v0.5.9, intermediate artifacts can be found at D:\Programs\msys64\tmp\cargo-installuxOPpu. To reuse those artifacts with a future compilation, set the environment variable CARGO\_TARGET\_DIR to that path. Caused by: failed to select a version for the requirement cachedir = "^0.1.1" version 0.1.1 is yanked location searched: crates.io index required by package rga v0.5.9

This is a dependency issue - the rga crate depends on cachedir v0.1.1 which has been "yanked" (removed) from crates.io. This is a known issue with older versions of RGA.

**Solutions:**

**Option 1: Use the MSYS2 version (Recommended)** Since you already have a working RGA from MSYS2, just use that:

bash

*# Test it works*

/ucrt64/bin/rga --version

*# Make it accessible as 'rga' by adjusting PATH or alias*

echo 'alias rga="/ucrt64/bin/rga"' >> ~/.bashrc

source ~/.bashrc

**Option 2: Try installing a newer version of RGA**

bash

*# Try installing from git (latest version)*

cargo install --git https://github.com/phiresky/ripgrep-all rga

**Option 3: Install a working alternative**

bash

*# Try other similar tools*

cargo install ast-grep *# AST-based search*

cargo install grep-rs *# Pure Rust grep*

**My recommendation: Use MSYS2 RGA**

Since the MSYS2 version is:

* ✅ Already installed and working
* ✅ Has proper system integration
* ✅ Compiled with all necessary libraries
* ✅ No dependency conflicts

Just adjust your PATH or use an alias:

bash

*# Add this to ~/.bashrc*

export PATH="/ucrt64/bin:$CARGO\_HOME/bin:$PATH"

This puts the MSYS2 tools first for cases like this where the cargo version has issues, while still allowing your cargo-installed tools to work.

**Test it:**

bash

rga --version

*# Should now show the MSYS2 version*

The rest of your cargo tools should install fine:

bash

cargo install fd-find bat eza

$ # Try installing from git (latest version) cargo install --git <https://github.com/phiresky/ripgrep-all> rga Updating git repository https://github.com/phiresky/ripgrep-all error: could not find rga in <https://github.com/phiresky/ripgrep-all> with version \*

The git installation failed because the binary name is different. Let's try without specifying the binary name:

bash

*# Try without specifying 'rga' at the end*

cargo install --git https://github.com/phiresky/ripgrep-all

If that still fails, let's just go with the **MSYS2 version** which is the most practical solution:

**Use MSYS2 RGA (Recommended approach):**

**1. Test the MSYS2 version:**

bash

/ucrt64/bin/rga --version

/ucrt64/bin/rga --help

**2. Make it easily accessible:**

bash

*# Add alias to ~/.bashrc*

echo 'alias rga="/ucrt64/bin/rga"' >> ~/.bashrc

source ~/.bashrc

*# Or adjust PATH to prioritize MSYS2 for this tool*

export PATH="/ucrt64/bin:$CARGO\_HOME/bin:$PATH"

**3. Continue with other tools that work:**

bash

*# Install the tools that compile successfully*

cargo install ripgrep --features 'pcre2' --force

cargo install fd-find bat eza

cargo install cargo-watch cargo-expand cargo-audit cargo-outdated

cargo install tokei hyperfine dust procs bottom

**The reality is:**

* Some Rust tools work better when installed via system package managers
* MSYS2's RGA is compiled with proper system integration
* Mixing cargo and system tools is perfectly fine and often optimal

You'll have a working RGA from MSYS2 and the other Rust tools from cargo - best of both worlds!

This worked:

# Try without specifying 'rga' at the end cargo install --git <https://github.com/phiresky/ripgrep-all>

Excellent! That worked because when installing from git without specifying a binary name, cargo automatically detects and installs all the binaries from the repository.

Now you have the latest version of RGA installed via cargo. Let's verify it:

bash

*# Test the new installation*

rga --version

which rga

**Continue with the rest of the tools:**

bash

*# Make sure ripgrep has PCRE2 support*

cargo install ripgrep --features 'pcre2' --force

*# Install other useful tools*

cargo install fd-find bat eza

cargo install cargo-watch cargo-expand cargo-audit cargo-outdated

cargo install tokei hyperfine dust procs bottom starship

cargo install sd hexyl xsv git-delta

**The git installation method is useful when:**

* The crates.io version has dependency issues (like the yanked cachedir)
* You want the absolute latest features
* The published version is outdated

Your toolchain is now getting quite comprehensive! You have:

* ✅ ripgrep with PCRE2 support
* ✅ rga (ripgrep-all) from latest git
* ✅ Full Rust development environment
* ✅ MSYS2 UCRT64 build tools

This gives you a powerful, modern command-line environment for development.

This is confusing. What I should do?

warning: ripgrep\_all (lib) generated 5 warnings (run cargo fix --lib -p ripgrep\_all to apply 2 suggestions) Finished release profile [optimized + debuginfo] target(s) in 8m 56s Installing D:/Programs/cargo\bin\rga-fzf-open.exe Installing D:/Programs/cargo\bin\rga-fzf.exe Installing D:/Programs/cargo\bin\rga-preproc.exe Installing D:/Programs/cargo\bin\rga.exe Installed package ripgrep\_all v0.10.9 (https://github.com/phiresky/ripgrep-all#2ccf3eeb) (executables rga-fzf-open.exe, rga-fzf.exe, rga-preproc.exe, rga.exe) (venv) juhi@LAPTOP-BGI3SJP4 MSYS /d/Programs/rustup/install\_essential\_crates $ cargo fix --lib -p ripgrep\_all error: the working directory of this package has uncommitted changes, and cargo fix can potentially perform destructive changes; if you'd like to suppress this error pass --allow-dirty, or commit the changes to these files: \* .gitignore (dirty) \* Cargo.lock (dirty) \* Cargo.toml (dirty) \* src/ (dirty)

Don't worry about that warning! Here's what's happening and what you should do:

**The installation was SUCCESSFUL! ✅**

The important part is this line:

Installed package `ripgrep\_all v0.10.9` (executables `rga-fzf-open.exe`, `rga-fzf.exe`, `rga-preproc.exe`, `rga.exe`)

You now have RGA installed successfully with all its tools.

**Ignore the warnings:**

**1. The "5 warnings" during compilation:**

* These are just compiler warnings in the RGA source code
* They don't affect functionality
* You can ignore them

**2. The cargo fix suggestion:**

* This is for the RGA developers to fix their code
* **DON'T run it** - you're not developing RGA, just using it
* The error about "uncommitted changes" confirms this isn't for you

**What you should do now:**

**1. Test that RGA works:**

bash

rga --version

rga --help

**2. Continue installing other tools:**

bash

*# Continue with your other installations*

cargo install ripgrep --features 'pcre2' --force

cargo install fd-find bat eza

cargo install tokei hyperfine dust procs

**3. Clean up (optional):**

bash

*# Remove the temporary build directory if you want*

*# (It will be cleaned up automatically anyway)*

**Summary:**

* ✅ RGA installed successfully
* ⚠️ Ignore compiler warnings (not your problem)
* ⚠️ Ignore cargo fix suggestions (not for end users)
* 🚀 Continue with your setup

You're all good to go!

In between I got this error:

Downloaded dust v0.1.0 Downloaded 1 crate (3.0KiB) in 1.35s error: there is nothing to install in dust v0.1.0, because it has no binaries cargo install is only for installing programs, and can't be used with libraries. To use a library crate, add it as a dependency to a Cargo project with cargo add. Updating crates.io index

The issue is that dust v0.1.0 is a library crate, not a binary. The tool you want is called du-dust. Here's the fix:

**Correct command:**

bash

*# Use du-dust instead of dust*

cargo install du-dust

**Here's the corrected list of working cargo install commands:**

bash

*# Core tools*

cargo install ripgrep --features 'pcre2' --force

cargo install fd-find bat eza

*# Development tools*

cargo install cargo-watch cargo-expand cargo-audit cargo-outdated

cargo install tokei hyperfine

*# System utilities*

cargo install du-dust *# ← Correct name*

cargo install procs bottom starship zoxide git-delta

*# Text processing*

cargo install sd hexyl xsv

*# Network*

cargo install xh

**The pattern to remember:**

Some crates have different names for installation:

* ✅ cargo install du-dust (not dust)
* ✅ cargo install fd-find (not fd)
* ✅ cargo install ripgrep (not rg)

**If you're ever unsure about a crate name:**

bash

*# Search on crates.io*

cargo search dust

cargo search "better du"

Continue with the corrected du-dust and you should be fine!

Why no dlltools even though my rust is installed with gnu suffix and I run installers of cargo from msys bash (ucrt64):

error: Error calling dlltool 'dlltool.exe': program not found error: could not compile chrono (lib) due to 1 previous error warning: build failed, waiting for other jobs to finish... error: failed to compile fd-find v10.2.0, intermediate artifacts can be found at C:\Users\juhi\AppData\Local\Temp\cargo-installn1qPty. To reuse those artifacts with a future compilation, set the environment variable CARGO\_TARGET\_DIR to that path.

This is a frustrating issue! Even though you're using the GNU toolchain and running from MSYS2, some crates still can't find dlltool.exe. Let's troubleshoot:

**1. First, verify dlltool is available:**

bash

which dlltool

dlltool --version

echo $PATH | grep ucrt64

**2. Check if dlltool is in the expected location:**

bash

ls -la /ucrt64/bin/dlltool\*

ls -la /ucrt64/x86\_64-w64-mingw32/bin/dlltool\*

**3. Possible solutions:**

**Option A: Set explicit tool paths:**

bash

export CC=gcc

export AR=ar

export DLLTOOL=dlltool

export WINDRES=windres

*# Then try installing*

cargo install fd-find

**Option B: Add the mingw32 bin directory to PATH:**

bash

export PATH="/ucrt64/x86\_64-w64-mingw32/bin:$PATH"

cargo install fd-find

**Option C: Install missing tools explicitly:**

bash

*# Make sure all binutils are installed*

pacman -S mingw-w64-ucrt-x86\_64-binutils

pacman -S mingw-w64-ucrt-x86\_64-toolchain --needed

**Option D: Use Windows-style paths for tools:**

bash

*# Sometimes cargo needs Windows paths*

export DLLTOOL="C:/msys64/ucrt64/bin/dlltool.exe"

cargo install fd-find

**4. Alternative approach - use system temp directory:**

bash

export CARGO\_TARGET\_DIR="/tmp/cargo-target"

cargo install fd-find

**5. Debug what's happening:**

bash

*# Run with verbose output to see exactly what's failing*

cargo install fd-find -v

Try these solutions in order and let me know which one works or what specific error messages you get!

dlltools is already in path:

juhi@LAPTOP-BGI3SJP4 MSYS /d/Programs/rustup/install\_essential\_crates $ which dlltool /usr/bin/dlltool (venv) juhi@LAPTOP-BGI3SJP4 MSYS /d/Programs/rustup/install\_essential\_crates $ echo $PATH /home/juhi/venv/Scripts:/d/Programs/rustup/bin:/d/Programs/cargo/bin:/usr/local/bin:/usr/bin:/bin:/opt/bin:/usr/bin/site\_perl:/usr/bin/vendor\_perl:/usr/bin/core\_perl:/ucrt64/bin:/ucrt64/lib:/ucrt64/include:/ucrt64/share:/ucrt64/local:/ucrt64/libexec:/ucrt64/sbin:/ucrt64/var:/ucrt64/x86\_64-w64-mingw32:/home/juhi/venv/Scripts:/c/ffmpeg/bin:/d/Programs/Python/Scripts:/d/Programs/Python:/d/Programs/Powershell:/d/Programs/7-zip:/d/programs/ffmpeg/bin:/d/Programs/gvim:/c/Users/juhi/AppData/Local/Microsoft/WindowsApps:/c/Users/juhi/AppData/Local/Programs/Python/Launcher:/c/windows/system32:/c/windows:/c/windows/System32/Wbem:/c/windows/System32/WindowsPowerShell/v1.0:/c/windows/System32/OpenSSH:/ucrt64/bin:/ucrt64/lib:/ucrt64/include:/ucrt64/share:/ucrt64/local:/ucrt64/libexec:/ucrt64/sbin:/ucrt64/var:/ucrt64/x86\_64-w64-mingw32

The issue is that dlltool is found at /usr/bin/dlltool but cargo might be looking for the UCRT64 version. There's a mismatch between toolchains.

**The problem:**

* You have Rust GNU toolchain (x86\_64-pc-windows-gnu)
* dlltool is in /usr/bin/ (regular MSYS2, not UCRT64)
* But you're using UCRT64 environment

**Solutions:**

**Option 1: Use the UCRT64 dlltool explicitly:**

bash

*# Check if UCRT64 version exists*

ls /ucrt64/bin/dlltool\*

ls /ucrt64/x86\_64-w64-mingw32/bin/dlltool\*

*# If it exists, prioritize UCRT64 tools*

export PATH="/ucrt64/bin:/ucrt64/x86\_64-w64-mingw32/bin:$PATH"

which dlltool *# Should now show /ucrt64/bin/dlltool*

**Option 2: Set explicit tool paths for the UCRT64 toolchain:**

bash

export CC="/ucrt64/bin/gcc"

export AR="/ucrt64/bin/ar"

export DLLTOOL="/ucrt64/bin/dlltool"

export WINDRES="/ucrt64/bin/windres"

cargo install fd-find

**Option 3: Install the complete UCRT64 binutils:**

bash

*# Ensure you have the full UCRT64 toolchain*

pacman -S mingw-w64-ucrt-x86\_64-binutils

pacman -S mingw-w64-ucrt-x86\_64-gcc

**Option 4: Check if there's a toolchain mismatch:**

bash

*# Verify what dlltool version you're using*

/usr/bin/dlltool --version

*# vs*

/ucrt64/bin/dlltool --version *# if it exists*

Try Option 1 first - reordering PATH to prioritize UCRT64 tools. This should resolve the toolchain consistency issue.

There is no ucrt64 specific dlltools I checked - if you find that package let me know

You're right! Let me help you find the correct UCRT64 package for dlltool.

**Search for UCRT64 binutils packages:**

bash

*# Search for binutils packages in UCRT64*

pacman -Ss mingw-w64-ucrt-x86\_64 | grep binutils

*# Or search more broadly*

pacman -Ss ucrt.\*binutils

pacman -Ss ucrt.\*dlltool

*# Check what's available in the mingw-w64-ucrt-x86\_64 toolchain*

pacman -Ql mingw-w64-ucrt-x86\_64-toolchain | grep dlltool

**Alternative approaches while we find the right package:**

**Option A: Create a symlink to make the regular dlltool work:**

bash

*# Create a symlink in ucrt64/bin*

sudo ln -sf /usr/bin/dlltool /ucrt64/bin/dlltool

**Option B: Tell cargo to use the existing dlltool explicitly:**

bash

export DLLTOOL="/usr/bin/dlltool"

export CC="/ucrt64/bin/gcc"

export AR="/ucrt64/bin/ar"

cargo install fd-find -v

**Option C: Check if dlltool is part of another package:**

bash

*# Find which package provides dlltool in UCRT64*

pacman -F dlltool | grep ucrt64

Try the search commands above and let me know what packages show up. The dlltool might be bundled in a different package name in the UCRT64 repository.

If no UCRT64-specific dlltool exists, the symlink approach (Option A) should work to make cargo find it.

Already installed:

$ pacman -Ss mingw-w64-ucrt-x86\_64 | grep binutils ucrt64/mingw-w64-ucrt-x86\_64-arm-none-eabi-binutils 2.45-1 (mingw-w64-ucrt-x86\_64-arm-none-eabi-toolchain) ucrt64/mingw-w64-ucrt-x86\_64-avr-binutils 2.45-1 (mingw-w64-ucrt-x86\_64-avr-toolchain) ucrt64/mingw-w64-ucrt-x86\_64-binutils 2.45-2 (mingw-w64-ucrt-x86\_64-toolchain) [installed] ucrt64/mingw-w64-ucrt-x86\_64-m68k-apple-macos-binutils r808.8d32c583ae-2 cross-binutils for classic 68K Macintoshes (mingw-w64) ucrt64/mingw-w64-ucrt-x86\_64-powerpc-apple-macos-binutils r808.8d32c583ae-1 cross-binutils for PPC Macintoshes (mingw-w64) ucrt64/mingw-w64-ucrt-x86\_64-riscv64-unknown-elf-binutils 2.45-1 (mingw-w64-ucrt-x86\_64-riscv64-unknown-elf-toolchain)

See 3rd in the list above

Perfect! You already have mingw-w64-ucrt-x86\_64-binutils installed. Let's find where dlltool is located:

**Check where the UCRT64 dlltool is installed:**

bash

*# List all files from the binutils package*

pacman -Ql mingw-w64-ucrt-x86\_64-binutils | grep dlltool

*# Or check in the typical locations*

ls -la /ucrt64/bin/\*dlltool\*

ls -la /ucrt64/x86\_64-w64-mingw32/bin/\*dlltool\*

*# Check all binutils tools*

ls /ucrt64/bin/ | grep -E "(dlltool|ar|as|ld|objdump)"

**If it's in a subdirectory, add it to PATH:**

bash

*# If dlltool is in /ucrt64/x86\_64-w64-mingw32/bin/*

export PATH="/ucrt64/x86\_64-w64-mingw32/bin:/ucrt64/bin:$PATH"

*# Verify it's found*

which dlltool

dlltool --version

**If still not found, check the package contents:**

bash

*# See exactly what the binutils package installed*

pacman -Ql mingw-w64-ucrt-x86\_64-binutils | head -20

The UCRT64 dlltool should definitely be there since the package is installed. Once we find it, we can update your PATH or create the appropriate environment variables.

What do these commands show?

???

$ pacman -Ql mingw-w64-ucrt-x86\_64-binutils | grep dlltool mingw-w64-ucrt-x86\_64-binutils /ucrt64/bin/dlltool.exe mingw-w64-ucrt-x86\_64-binutils /ucrt64/share/man/man1/dlltool.1.gz mingw-w64-ucrt-x86\_64-binutils /ucrt64/x86\_64-w64-mingw32/bin/dlltool.exe (venv) juhi@LAPTOP-BGI3SJP4 MSYS /d/Programs/rustup/install\_essential\_crates $ dlltools bash: dlltools: command not found

I see the issue! The UCRT64 dlltool IS installed, but there are multiple versions and cargo might not be finding the right one.

**Check what's happening:**

bash

*# Check both dlltool versions*

which dlltool

/usr/bin/dlltool --version

/ucrt64/bin/dlltool --version

*# See your current PATH priority*

echo $PATH | tr ':' '\n' | grep -n -E "(usr/bin|ucrt64)"

**Fix the PATH priority:**

The issue is /usr/bin comes before /ucrt64/bin in your PATH. Let's fix this:

bash

*# Temporarily fix PATH for this session*

export PATH="/ucrt64/bin:$PATH"

*# Verify dlltool now points to UCRT64 version*

which dlltool

*# Should now show: /ucrt64/bin/dlltool*

*# Try installing fd-find again*

cargo install fd-find

**Make the PATH fix permanent:**

Add this to your ~/.bashrc:

bash

*# Add this line to prioritize UCRT64 tools*

export PATH="/ucrt64/bin:/ucrt64/x86\_64-w64-mingw32/bin:$PATH"

**The problem was:**

* You have dlltool in /usr/bin/dlltool (regular MSYS2)
* You also have dlltool in /ucrt64/bin/dlltool (UCRT64)
* But /usr/bin comes first in PATH, so cargo was getting the wrong one

Try the PATH fix and see if cargo install fd-find works now!

I can see you have recommended different cargo install packages above at various places. See above responses and give me one comprehensive list.

You're absolutely right! Let me give you one comprehensive, consolidated list based on all our previous discussions:

**Comprehensive Cargo Install List**

**Core Search & File Tools**

bash

cargo install ripgrep --features 'pcre2' --force

cargo install --git https://github.com/phiresky/ripgrep-all *# rga (ripgrep-all)*

cargo install fd-find

cargo install bat

cargo install eza

**Cargo Development Tools**

bash

cargo install cargo-watch

cargo install cargo-expand

cargo install cargo-audit

cargo install cargo-outdated

cargo install cargo-tree

cargo install cargo-edit

**System & Utility Tools**

bash

cargo install du-dust *# better du (NOT "dust")*

cargo install procs *# better ps*

cargo install bottom *# system monitor (better htop)*

cargo install starship *# customizable shell prompt*

cargo install zoxide *# smart cd*

cargo install tokei *# count lines of code*

cargo install hyperfine *# benchmarking tool*

**Text Processing & Development**

bash

cargo install sd *# better sed*

cargo install hexyl *# hex viewer*

cargo install xsv *# CSV toolkit*

cargo install git-delta *# better git diff*

cargo install just *# command runner*

**Network Tools**

bash

cargo install xh *# better curl/httpie*

**Development Dependencies (for testing)**

bash

cargo add --dev criterion proptest mockall tempfile

**Install All at Once (with PATH fix):**

bash

*# First fix PATH to prioritize UCRT64 tools*

export PATH="/ucrt64/bin:/ucrt64/x86\_64-w64-mingw32/bin:$PATH"

*# Then install everything*

cargo install ripgrep --features 'pcre2' --force

cargo install --git https://github.com/phiresky/ripgrep-all

cargo install fd-find bat eza du-dust procs bottom starship zoxide

cargo install cargo-watch cargo-expand cargo-audit cargo-outdated cargo-tree cargo-edit

cargo install tokei hyperfine sd hexyl xsv git-delta just xh

This is the definitive list combining all our recommendations!

include the exception for above that if the crates are installed already we can skip them

**Comprehensive Cargo Install List (Skip if Already Installed)**

**Check what's already installed:**

bash

cargo install --list

**Core Search & File Tools**

bash

*# Only install if not already present*

cargo install ripgrep --features 'pcre2' --force *# Force needed to add pcre2 feature*

cargo install --git https://github.com/phiresky/ripgrep-all *# rga (already installed)*

cargo install fd-find *# Skip if already installed*

cargo install bat *# Skip if already installed*

cargo install eza *# Skip if already installed*

**Cargo Development Tools**

bash

cargo install cargo-watch *# Already installed - SKIP*

cargo install cargo-expand *# Already installed - SKIP*

cargo install cargo-audit *# Already installed - SKIP*

cargo install cargo-outdated *# Already installed - SKIP*

cargo install cargo-tree *# Install if not present*

cargo install cargo-edit *# Install if not present*

**System & Utility Tools**

bash

cargo install du-dust *# Install if not present*

cargo install procs *# Install if not present*

cargo install bottom *# Install if not present*

cargo install starship *# Install if not present*

cargo install zoxide *# Install if not present*

cargo install tokei *# Install if not present*

cargo install hyperfine *# Install if not present*

**Text Processing & Development**

bash

cargo install sd *# Install if not present*

cargo install hexyl *# Install if not present*

cargo install xsv *# Install if not present*

cargo install git-delta *# Install if not present*

cargo install just *# Install if not present*

**Network Tools**

bash

cargo install xh *# Install if not present*

**Smart Installation Command (checks before installing):**

bash

*# First fix PATH*

export PATH="/ucrt64/bin:/ucrt64/x86\_64-w64-mingw32/bin:$PATH"

*# Install only what's needed (cargo will skip if already installed)*

cargo install ripgrep --features 'pcre2' --force *# Force for feature update*

cargo install fd-find bat eza du-dust procs bottom starship zoxide

cargo install cargo-tree cargo-edit tokei hyperfine sd hexyl xsv git-delta just xh

*# rga already installed from git - SKIP*

*# cargo-watch, cargo-expand, cargo-audit, cargo-outdated already installed - SKIP*

**Note:** cargo install automatically skips packages that are already installed (unless you use --force), so you can run the commands safely!

I still see that some are compiled multiple times or am I missing?

Compiling regex v1.11.1

You're absolutely right! You're seeing regex being compiled multiple times because it's a **dependency** of many different crates you're installing.

**What's happening:**

When you install multiple cargo tools, many of them depend on common crates like:

* regex - used by ripgrep, fd-find, sd, xsv, etc.
* serde - used by almost everything for serialization
* tokio - used by network tools, async tools
* clap - used by CLI tools for argument parsing

**Why it compiles multiple times:**

Each tool might use:

* **Different versions** of the same dependency
* **Different features** enabled for the same crate
* **Different compilation profiles** (release vs debug)

**This is normal behavior, but you can optimize:**

**Option 1: Install in batches (reduces some recompilation):**

bash

*# Install related tools together*

cargo install ripgrep fd-find bat *# All use similar regex/text processing*

cargo install du-dust procs bottom *# All are system tools*

cargo install cargo-watch cargo-expand cargo-audit cargo-outdated *# All cargo tools*

**Option 2: Use shared compilation cache:**

bash

*# Set a shared target directory (optional)*

export CARGO\_TARGET\_DIR="/tmp/cargo-shared"

**Option 3: Just let it compile:**

* This is normal Rust behavior
* Each tool gets optimized builds
* Compilation is a one-time cost
* The final binaries are independent and fast

**The repeated compilation of regex and other common crates is expected and normal! Each tool needs its own optimized version.**

Can I copy this rustup and cargo folders to the new laptop (if I buy in future) and use this installations out-of-shelf by defining proper paths, etc. - Just to avoids this long process of installation?

Yes, you can! But there are some important considerations and best practices:

**What CAN be copied:**

**✅ Cargo global tools (CARGO\_HOME/bin):**

* All your cargo installed tools (rg, fd, bat, etc.)
* These are standalone executables

**✅ Rustup toolchains (RUSTUP\_HOME):**

* Installed Rust versions (stable, nightly, etc.)
* Target platforms and components

**✅ Cargo registry cache (CARGO\_HOME/registry):**

* Downloaded crate sources (saves download time)

**What needs attention:**

**⚠️ Architecture compatibility:**

* Same CPU architecture (x86\_64)
* Same OS (Windows)
* Same ABI (GNU vs MSVC) - yours is GNU

**⚠️ Path dependencies:**

* Tools compiled with GNU toolchain need MSYS2/mingw
* System library dependencies must match

**Best approach for laptop migration:**

**Option 1: Full Copy + Setup (Recommended)**

bash

*# On new laptop:*

1. Install MSYS2 UCRT64 first

2. Copy your folders:

- D:\Programs\rustup → new laptop

- D:\Programs\cargo → new laptop

3. Set environment variables:

- RUSTUP\_HOME=D:\Programs\rustup

- CARGO\_HOME=D:\Programs\cargo

- Add D:\Programs\cargo\bin to PATH

4. Install MSYS2 development packages

**Option 2: Hybrid Approach (Faster)**

bash

*# Copy toolchains but reinstall tools*

1. Copy RUSTUP\_HOME (Rust toolchains)

2. Set environment variables

3. Run: cargo install --list *# to see what you had*

4. Reinstall key tools: cargo install ripgrep fd-find bat eza

**Option 3: Script the Installation**

Create a setup script:

bash

#!/bin/bash

*# install-rust-env.sh*

export RUSTUP\_HOME="D:/Programs/rustup"

export CARGO\_HOME="D:/Programs/cargo"

*# Install toolchain*

curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh -s -- -y

*# Install all tools*

cargo install ripgrep --features 'pcre2'

cargo install fd-find bat eza du-dust procs bottom

*# ... rest of your tools*

**Recommended approach:**

**Copy the folders + proper setup** - it will work, but ensure the new laptop has the same MSYS2 environment setup. This saves significant compilation time!

I can see many rustc installation threads in tak manager. How I can set all prorities of rustc to high at once

You can set process priorities for all rustc instances at once using these methods:

**Method 1: PowerShell (Recommended)**

powershell

*# Set all rustc processes to High priority*

Get-Process -Name "rustc\*" | ForEach-Object { $\_.PriorityClass = "High" }

*# Or more specific (in case of multiple rust processes)*

Get-Process | Where-Object { $\_.ProcessName -like "\*rustc\*" } | ForEach-Object { $\_.PriorityClass = "High" }

**Method 2: Command Prompt (wmic)**

cmd

# Set all rustc processes to high priority

wmic process where "name like '%rustc%'" CALL setpriority "high priority"

**Method 3: Task Manager (Manual but Persistent)**

1. Open Task Manager (Ctrl+Shift+Esc)
2. Go to "Details" tab
3. Find all rustc.exe processes
4. Right-click each → "Set Priority" → "High"
5. Or select multiple with Ctrl+Click, then right-click

**Method 4: Process Hacker (Third-party tool)**

* Download Process Hacker
* Select multiple rustc processes
* Right-click → Priority → High

**Method 5: Set Default Priority for Future rustc**

cmd

# Create a registry entry to always start rustc with high priority

reg add "HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\rustc.exe\PerfOptions" /v CpuPriorityClass /t REG\_DWORD /d 3 /f

**Method 6: PowerShell Script for Monitoring**

powershell

*# Keep monitoring and setting priority*

while ($true) {

Get-Process -Name "\*rustc\*" -ErrorAction SilentlyContinue | ForEach-Object {

if ($\_.PriorityClass -ne "High") {

$\_.PriorityClass = "High"

}

}

Start-Sleep -Seconds 5

}

**The PowerShell method (Method 1) is the quickest for immediate results!**

Okay - Now clear. Done with these. Now since I am on nightly channel when I update using rustup update, it does update rustup, but what about these crates?

Do I need to manually run for each crate, above commands every time? That would be quite cumbersome. Also, if every time, all dependencies are installed, it will nearly take 4-5 hours

Great question! Let's clarify what gets updated and how to manage it efficiently:

**What rustup update does:**

* ✅ Updates Rust toolchains (nightly, stable, etc.)
* ✅ Updates rustup itself
* ❌ **Does NOT update cargo-installed crates**

**Your cargo-installed crates are separate and need manual updates**

**Efficient update strategies:**

**Method 1: Use cargo-update (Recommended)**

bash

*# Install cargo-update tool (one-time)*

cargo install cargo-update

*# Update all installed crates at once*

cargo install-update -a

*# Check what would be updated without updating*

cargo install-update -l

**Method 2: Use cargo-outdated (You already have this)**

bash

*# Check which global tools are outdated*

cargo outdated --root

*# Then selectively update only the outdated ones*

cargo install ripgrep --force

cargo install fd-find --force

**Method 3: Batch update script**

Create update-cargo-tools.sh:

bash

#!/bin/bash

*# List of your essential tools*

TOOLS="ripgrep fd-find bat eza du-dust procs bottom cargo-watch cargo-expand"

for tool in $TOOLS; do

echo "Updating $tool..."

cargo install $tool --force

done

**Method 4: Smart update workflow**

bash

*# 1. Update Rust toolchain first*

rustup update

*# 2. Check what's outdated*

cargo install-update -l

*# 3. Update only specific tools you use frequently*

cargo install-update ripgrep fd-find bat

*# 4. Update development tools less frequently*

cargo install-update cargo-watch cargo-expand --once-per-month

**Time-saving tips:**

**Don't update everything every time:**

* Core tools (ripgrep, fd-find, bat): Update monthly
* Development tools (cargo-watch, cargo-expand): Update when needed
* Specialized tools: Update only when you encounter issues

**Use incremental compilation:**

bash

*# Set this to speed up rebuilds*

export CARGO\_INCREMENTAL=1

**The cargo-update tool is your best friend** - it makes updating all crates as simple as cargo install-update -a!

These "creates" are same as libraries of python or packages in linux or msys?

Great question! Yes, there are similarities but also key differences. Let me clarify:

**Crates vs Other Package Systems:**

**Crates are like:**

* **Python packages/libraries** (PyPI packages)
* **Linux packages** (apt, yum, pacman packages)
* **MSYS2 packages** (pacman packages)
* **Node.js modules** (npm packages)
* **Ruby gems**

**Key Similarities:**

* ✅ Reusable code libraries
* ✅ Dependency management
* ✅ Central registry (crates.io ≈ PyPI ≈ npm registry)
* ✅ Version management
* ✅ Installation commands (cargo install ≈ pip install ≈ pacman -S)

**Key Differences:**

**Two Types of Crates:**

**1. Library Crates (like Python libraries):**

bash

cargo add serde tokio regex *# Added to your project's Cargo.toml*

*# These are like: pip install requests pandas*

**2. Binary Crates (like system packages):**

bash

cargo install ripgrep fd-find bat *# Global command-line tools*

*# These are like: pacman -S grep fd bat*

**Compilation vs Interpretation:**

* **Rust crates**: Compiled to native binaries (fast execution)
* **Python packages**: Mostly interpreted (some C extensions)
* **System packages**: Mix of compiled binaries and scripts

**Dependency Isolation:**

* **Rust**: Each project has its own dependency versions
* **Python**: Global installs can conflict (hence virtual environments)
* **System packages**: Shared system-wide

**Example Comparisons:**

| **Rust** | **Python** | **Linux/MSYS2** | **Purpose** |
| --- | --- | --- | --- |
| cargo add requests | pip install requests | - | HTTP client library |
| cargo install ripgrep | - | pacman -S ripgrep | Search tool |
| cargo install cargo-watch | pip install watchdog | pacman -S entr | File watching |

**So yes, crates are Rust's equivalent to:**

* **Python's pip packages**
* **Linux's apt/yum/pacman packages**
* **MSYS2's pacman packages**
* **JavaScript's npm packages**

The main difference is Rust compiles everything to fast native binaries!