# ripgrep crates/cli/src/lib.rs: Code Companion

Reference code for the CLI Utilities Overview lecture. Sections correspond to the lecture document.

## Section 1: The Crate's Documentation-First Design

```
/*!
This crate provides common routines used in command line applications, with a
focus on routines useful for search oriented applications. As a utility
library, there is no central type or function. However, a key focus of this
crate is to improve failure modes and provide user friendly error messages
when things go wrong.

To the best extent possible, everything in this crate works on Windows, macOS
and Linux.


# Standard I/O

[`is_readable_stdin`] determines whether stdin can be usefully read from...

# Coloring and buffering

The [`stdout`], [`stdout_buffered_block`] and [`stdout_buffered_line`] routines
are alternative constructors for [`StandardStream`]...

# Escaping

The [`escape`](crate::escape()), [`escape_os`], [`unescape`] and
[`unescape_os`] routines provide a user friendly way of dealing with UTF-8
encoded strings that can express arbitrary bytes...
*/

#![deny(missing_docs)]  // Compile-time enforcement: all public items must be
documented
```

The `#![deny(missing_docs)]` attribute turns missing documentation into a compile error rather than a warning. The `!` indicates this is an inner attribute applying to the entire crate.

## Section 2: Module Organization and Re-exports

```rust
// Private module declarations - implementation details hidden from users
mod decompress;
mod escape;
mod hostname;
mod human;
mod pattern;
mod process;
mod wtr;

// Carefully curated public API via re-exports
pub use crate::{
    // Decompression utilities - grouped together
    decompress::{
        DecompressionMatcher, DecompressionMatcherBuilder,
        DecompressionReader, DecompressionReaderBuilder, resolve_binary,
    },
    // Escape sequence handling
    escape::{escape, escape_os, unescape, unescape_os},
    // System hostname retrieval
    hostname::hostname,
    // Human-readable size parsing (e.g., "2M" -> bytes)
    human::{ParseSizeError, parse_human_readable_size},
    // Pattern parsing with good error messages
    pattern::{
        InvalidPatternError, pattern_from_bytes, pattern_from_os,
        patterns_from_path, patterns_from_reader, patterns_from_stdin,
    },
    // Subprocess execution with stderr capture
    process::{CommandError, CommandReader, CommandReaderBuilder},
    // Terminal-aware output streams
    wtr::{
        StandardStream, stdout, stdout_buffered_block, stdout_buffered_line,
    },
};
```

The `pub use crate::module::Item` pattern creates a flat public namespace. Users import `grep_cli::unescape` rather than navigating the internal module structure.

## Section 3: Heuristic Stdin Detection

```rust
/// Returns true if and only if stdin is believed to be readable.
///
/// Note that this isn't perfect and essentially corresponds to a heuristic.
/// When things are unclear (such as if an error occurs during introspection to
/// determine whether stdin is readable), this prefers to return `false`. That
/// means it's possible for an end user to pipe something into your program and
/// have this return `false` and thus potentially lead to ignoring the user's
/// stdin data. While not ideal, this is perhaps better than falsely assuming
/// stdin is readable, which would result in blocking forever on reading stdin.
pub fn is_readable_stdin() -> bool {
    use std::io::IsTerminal;

    // ... platform-specific imp() functions defined here ...

    // Two-step check:
    // 1. If stdin IS a terminal, user is typing - no piped data
    // 2. If stdin is NOT a terminal, check what kind of input it is
    !std::io::stdin().is_terminal() && imp()
}
```

The `IsTerminal` trait (stabilized in Rust 1.70) provides the first filter. The `imp()` function provides platform-specific refinement when stdin is not a terminal.

---

## Section 4: Platform-Specific Implementation with Conditional Compilation

```rust
#[cfg(unix)]  // Only compiled on Unix-like systems
fn imp() -> bool {
    use std::{
        fs::File,
        os::{fd::AsFd, unix::fs::FileTypeExt},  // Unix-specific extensions
    };

    let stdin = std::io::stdin();
    // Clone the file descriptor to get owned access
    let fd = match stdin.as_fd().try_clone_to_owned() {
        Ok(fd) => fd,
        Err(err) => {
            log::debug!(
                "for heuristic stdin detection on Unix, \
                 could not clone stdin file descriptor \
                 (thus assuming stdin is not readable): {err}",
            );
            return false;  // Defensive: don't crash, just return false
        }
    };

    // Convert file descriptor to File to access metadata
    let file = File::from(fd);
    let md = match file.metadata() {
        Ok(md) => md,
        Err(err) => {
            log::debug!(
                "for heuristic stdin detection on Unix, \
                 could not get file metadata for stdin \
                 (thus assuming stdin is not readable): {err}",
            );
            return false;
        }
    };

    // Check file type using Unix-specific extension trait
    let ft = md.file_type();
    let is_file = ft.is_file();      // Regular file (e.g., < file.txt)
    let is_fifo = ft.is_fifo();      // Named pipe (e.g., mkfifo)
    let is_socket = ft.is_socket();  // Unix socket
    let is_readable = is_file || is_fifo || is_socket;

    // Detailed logging for debugging heuristic decisions
    log::debug!(
        "for heuristic stdin detection on Unix, \
         found that \
         is_file={is_file}, is_fifo={is_fifo} and is_socket={is_socket}, \
         and thus concluded that is_stdin_readable={is_readable}",
    );
    is_readable
}
```

The `FileTypeExt` trait provides Unix-specific methods like `is_fifo()` and `is_socket()` that aren't available in the cross-platform `std::fs::FileType`.

---

## Section 5: The Windows Implementation Contrast

```rust
#[cfg(windows)]  // Only compiled on Windows
fn imp() -> bool {
    // Windows uses handles, not file descriptors
    let stdin = winapi_util::HandleRef::stdin();
    let typ = match winapi_util::file::typ(stdin) {
        Ok(typ) => typ,
        Err(err) => {
            log::debug!(
                "for heuristic stdin detection on Windows, \
                 could not get file type of stdin \
                 (thus assuming stdin is not readable): {err}",
            );
            return false;
        }
    };

    // Windows file type categories differ from Unix
    let is_disk = typ.is_disk();  // Analogous to Unix is_file()
    let is_pipe = typ.is_pipe();  // Analogous to Unix is_fifo()
    let is_readable = is_disk || is_pipe;

    log::debug!(
        "for heuristic stdin detection on Windows, \
         found that is_disk={is_disk} and is_pipe={is_pipe}, \
         and thus concluded that is_stdin_readable={is_readable}",
    );
    is_readable
}

#[cfg(not(any(unix, windows)))]  // Fallback for other platforms
fn imp() -> bool {
    log::debug!("on non-{{Unix,Windows}}, assuming stdin is not readable");
    false  // Conservative default
}
```

The `winapi_util` crate provides safe Rust wrappers around Windows API calls. The fallback ensures compilation succeeds on any platform Rust supports.

---

## Section 6: Deprecation Patterns and API Evolution

```rust
/// Returns true if and only if stdin is believed to be connected to a tty
/// or a console.
///
/// Note that this is now just a wrapper around
/// [`std::io::IsTerminal`](https://doc.rust-lang.org/std/io/trait.IsTerminal.html).
/// Callers should prefer using the `IsTerminal` trait directly. This routine
/// is deprecated and will be removed in the next semver incompatible release.
#[deprecated(since = "0.1.10", note = "use std::io::IsTerminal instead")]
pub fn is_tty_stdin() -> bool {
    use std::io::IsTerminal;
    std::io::stdin().is_terminal()  // Delegate to standard library
}

#[deprecated(since = "0.1.10", note = "use std::io::IsTerminal instead")]
pub fn is_tty_stdout() -> bool {
    use std::io::IsTerminal;
    std::io::stdout().is_terminal()
}

#[deprecated(since = "0.1.10", note = "use std::io::IsTerminal instead")]
pub fn is_tty_stderr() -> bool {
    use std::io::IsTerminal;
    std::io::stderr().is_terminal()
}
```

The `#[deprecated]` attribute generates compiler warnings with the specified message. Functions remain functional but delegate to the replacement API, ensuring identical behavior.

## Quick Reference

### Public API Overview

| Category | Items | Purpose |
|---|---|---|
| Decompression | `DecompressionReader`, `DecompressionMatcher` | Read compressed files via external tools |
| Escape Handling | `escape`, `unescape`, `escape_os`, `unescape_os` | Convert between literal and escaped byte strings |

| | | |
|---|---|---|
| Pattern Parsing | `pattern_from_bytes`, `patterns_from_path` | Parse regex patterns with good error messages |
| Process I/O | `CommandReader`, `CommandError` | Execute processes with stderr capture |
| Terminal Output | `StandardStream`, `stdout` | TTY-aware buffered colored output |
| Utilities | `hostname`, `parse_human_readable_size` | System info and size parsing |

## Conditional Compilation Attributes

```
#[cfg(unix)]                        // Unix-only code
#[cfg(windows)]                     // Windows-only code
#[cfg(not(any(unix, windows)))]     // Fallback for other platforms
```

## Key Traits Used

| Trait | Source | Purpose |
|---|---|---|
| `IsTerminal` | `std::io` | Check if stream is connected to TTY |
| `AsFd` | `std::os::fd` | Access file descriptor on Unix |
| `FileTypeExt` | `std::os::unix::fs` | Unix-specific file type checks |

## Deprecation Pattern

```
#[deprecated(
    since = "version",      // When deprecation occurred
    note = "migration hint" // How to update code
)]
pub fn old_function() -> T {
    new_function()  // Delegate to replacement
}
```