ripgrep crates/searcher/src/searcher/mmap.rs: Code Companion

Reference code for the Memory Mapping lecture. Sections correspond to the lecture document.

Section 1: The Memory Mapping Trade-Off

```rust
use std::{fs::File, path::Path};

use memmap::Mmap;

/// Controls the strategy used for determining when to use memory maps.
///
/// If a searcher is called in circumstances where it is possible to use memory
/// maps, and memory maps are enabled, then it will attempt to do so if it
/// believes it will make the search faster.
///
/// By default, memory maps are disabled.
#[derive(Clone, Debug)]
pub struct MmapChoice(MmapChoiceImpl);
```

The `memmap` crate provides the cross-platform `Mmap` type that handles the actual memory mapping syscalls. The doc comment explicitly notes that memory maps are disabled by default—safety over performance.

Section 2: The Newtype Pattern for Configuration

```rust
/// Public struct wraps private enum - the newtype pattern
#[derive(Clone, Debug)]
pub struct MmapChoice(MmapChoiceImpl);

/// Private enum - external code cannot match on or construct this directly
#[derive(Clone, Debug)]
enum MmapChoiceImpl {
    Auto,   // Memory maps enabled when advantageous
    Never,  // Memory maps completely disabled
}
```

The inner `MmapChoiceImpl` enum has no `pub` visibility modifier, making it private to this module. External code must go through `MmapChoice`'s constructor functions to create values.

Section 3: Safety Contracts and Unsafe Constructors

```
impl MmapChoice {
    /// Use memory maps when they are believed to be advantageous.
    ///
    /// # Safety
    ///
    /// This constructor is not safe because there is no obvious way to
    /// encapsulate the safety of file backed memory maps on all platforms
    /// without simultaneously negating some or all of their benefits.
    ///
    /// The specific contract the caller is required to uphold isn't precise,
    /// but it basically amounts to something like, "the caller guarantees that
    /// the underlying file won't be mutated." This, of course, isn't feasible
    /// in many environments. However, command line tools may still decide to
    /// take the risk of, say, a `SIGBUS` occurring while attempting to read a
    /// memory map.
    pub unsafe fn auto() -> MmapChoice {
        MmapChoice(MmapChoiceImpl::Auto)
    }
}
```

The `unsafe` marker here is semantic rather than operational—the function body is trivial, but calling it requires acknowledging responsibility for the file's stability during mapped access.

Section 4: The Safe Default

```
impl Default for MmapChoice {
    fn default() -> MmapChoice {
        MmapChoice(MmapChoiceImpl::Never)
    }
}

impl MmapChoice {
    /// Never use memory maps, no matter what. This is the default.
    pub fn never() -> MmapChoice {
        MmapChoice(MmapChoiceImpl::Never)
    }
}
```

Both `Default::default()` and `never()` produce the same value. The named constructor makes intent explicit in calling code.

Section 5: The Open Method and Graceful Degradation

```
/// Return a memory map if memory maps are enabled and if creating a
/// memory from the given file succeeded and if memory maps are believed
/// to be advantageous for performance.
///
/// If this does attempt to open a memory map and it fails, then `None`
/// is returned and the corresponding error (along with the file path, if
/// present) is logged at the debug level.
pub(crate) fn open(
    &self,
    file: &File,
    path: Option<&Path>,
) -> Option<Mmap> {
    // First gate: is memory mapping enabled at all?
    if !self.is_enabled() {
        return None;
    }
    // Second gate: platform-specific heuristics
    if cfg!(target_os = "macos") {
        // I guess memory maps on macOS aren't great. Should re-evaluate.
        return None;
    }
    // ... actual mapping happens next
}
```

The `pub(crate)` visibility restricts this method to the searcher crate. Returns `Option<Mmap>` rather than `Result` because failure to map isn't an error—it's a signal to fall back to regular I/O.

Section 6: Propagating Safety Assertions

```
pub(crate) fn open(
    &self,
    file: &File,
    path: Option<&Path>,
) -> Option<Mmap> {
    // ... earlier checks omitted ...

    // SAFETY: This is acceptable because the only way `MmapChoiceImpl` can
    // be `Auto` is if the caller invoked the `auto` constructor, which
    // is itself not safe. Thus, this is a propagation of the caller's
    // assertion that using memory maps is safe.
    match unsafe { Mmap::map(file) } {
        Ok(mmap) => Some(mmap),
        Err(err) => {
            // Log at debug level - not an error, just an optimization that
didn't work
            if let Some(path) = path {
                log::debug!(
                    "{}: failed to open memory map: {}",
                    path.display(),
                    err
                );
            } else {
                log::debug!("failed to open memory map: {}", err);
            }
            None
        }
    }
}
```

The safety comment documents the reasoning chain: reaching this code with `MmapChoiceImpl::Auto` means `unsafe { MmapChoice::auto() }` was called somewhere, so the safety assertion was already made.

Section 7: The Enabled Check Helper

```
impl MmapChoice {
    /// Whether this strategy may employ memory maps or not.
    pub(crate) fn is_enabled(&self) -> bool {
        match self.0 {
            MmapChoiceImpl::Auto => true,
            MmapChoiceImpl::Never => false,
        }
    }
}
```

Simple pattern match on the inner enum. The `pub(crate)` visibility allows other modules in the searcher crate to check the memory map policy without accessing the private enum directly.

Quick Reference

Constructor	Safety	Returns	Use Case
`MmapChoice::auto()`	`unsafe`	`MmapChoice(Auto)`	Performance-critical, trusted environment
`MmapChoice::never()`	safe	`MmapChoice(Never)`	Safety-first, default behavior
`Default::default()`	safe	`MmapChoice(Never)`	Generic code, no explicit preference

Type Summary

```
// Public API type
pub struct MmapChoice(MmapChoiceImpl);

// Private implementation
enum MmapChoiceImpl { Auto, Never }

// Key method signatures
impl MmapChoice {
    pub unsafe fn auto() -> MmapChoice;
    pub fn never() -> MmapChoice;
    pub(crate) fn open(&self, file: &File, path: Option<&Path>) -> Option<Mmap>;
    pub(crate) fn is_enabled(&self) -> bool;
}
```

Safety Flow

```
Caller decides mmap is safe
          ↓
unsafe { MmapChoice::auto() }  ← Safety assertion made here
          ↓
MmapChoice value passed around (safe)
          ↓
open() called (safe method)
          ↓
unsafe { Mmap::map(file) }     ← Justified by earlier assertion
```