C* - Language Reference Manual

Name	UNI	Role
Shannon Jin	sj2802	Manager
Khyber Sen	ks3343	Language Guru
Ryan Lee	dbl2127	System Architect
Joanne Wang	jyw2118	Tester

Github link: https://github.com/kkysen/cstar/blob/main/LRM.md

Table of Contents

Overview

C* is a general-purpose systems programming language. It is between the level of C and Zig on a semantic level, and syntactically it also borrows a lot from Rust (pun intended). It is meant primarily for programs that would otherwise be implemented in C for the speed, simplicity, and explicitness of the language, but want a few simple higher-level language constructs, more expressiveness, and some safety, but not so many overwhelming language features and implicit costs like in Rust, C++, or Zig.

It has manual memory management (no GC) and uses LLVM as its primary codegen backend, so it can be optimized as well as C, or even better in cases. All of C*'s higher-level language constructs are zero-cost, meaning none of those features give it any overhead over C, which often lead to a highly-optimized style where in C you would take less efficient shortcuts (e.x. function pointers and type-erased generics) and use dangerous constructs like goto. In the future, it may also have a C backend so that it can target any architecture where there is a C compiler.

While a general-purpose language, C* will probably have the most advantages when used in systems and embedded programming. It's expressivity and high-level features combined with its relative simplicity, performance, and explicitness is a perfect match for many of these low-level systems and embedded programs.

Lexical Conventions

Comments

C* implements single-line, multi-line and structural comments. Tokens followed by // are considered single line comments and tokens followed by /* are considered multi-line comments. C* also has an additional structural comment /- which will comment out the next item, whether that be the next expression, the next line, or the next function. /// are doc comments

Example:

```
// This is a regular single line comment

/* This is a multiline comment
Everything inside here is commented out until "*/"
```

```
  */

  /- let x = 25 //this comments out the entire let expression
```

Identifiers

All identifiers in C* must be created from ASCII letters and decimal digits. Identifiers may contain underscore but must begin with a letter. They can not be a C* keyword.

Examples:

```
//valid identifier
let validWord = 2
fn get_num(): = {}

//invalid identifier
let 2words = 2
fn static(): = {}
```

Operators

Operator	Description	Example
+	binary arthmetic addition	2+2, 4.0+2.0
-	binary arithmetic subtraction	2-2, 4.2-2.2
*	binary arithmetic multiplication	2 * 2, 4.0 * 2.0
/	binary arithmetic division	2/2, 4.0/2.0
%	binary arithmetic modulus	2%2
>	binary relational greater than	a>2
<	binary relational less than	a<2
>=	binary relational greater than or equal to	a>=2
<=	binary relational less than or equal to	a<=2
==	binary relational equal	a==2
!=	binary relational not equal	a!=2
-	unary negation	-a
&	binary logical AND	a && b
	binary logical OR
!	unary logical NOT	!a
?	conditional ternary operator	x? a : b

Keywords

Keywords are reserved identifiers that cannot be used as regular identifiers for other purposes. C*
keywords:

- for, if, else, return, int, float, char, void, let, match, defer, break, label, true, false

Separators

Separator	Description
(	Left parenthesis for expression
)	Right parenthesis for expression
{	Left bracket for function
}	Right bracket for function
<	Left arrow for generics
>	Right arrow for generics
;	Semicolon to separate expressions
,	Comma to separate elements

Literals

C* Literals: string, int, float, char, boolean, unit, struct, tuple, closure, function, range

String Literals

The default string literal type is String, which is UTF-8 encoded and wraps a *[u8]. This is a borrowed slice
type and can't change size. To have a growable string, there is the StringBuf type, but there is no special
syntactic support for this owned string. Strings are made of chars, unicode scalar values, when iterating
(even though they are stored as *[u8]). chars have literals like c'\n'.

Then there are byte strings, which are just *[u8] and do not have to be UTF-8 encoded. String literals for
this are prefixed with b, like b"hello" (and for char byte literals, a b prefix, too: b'c'). The owning version of
this is just a Box<[u8]>.

Furthermore, for easier C FFI, there is also CString and CStringBuf, which are explicitly null-terminated. All
other string types are not null-terminated, since they store their own length, which is way more efficient
and safe. Literal CStrings have a c prefix, like c"/home".

And finally, there are format strings. Written `f"n + m = {n + m}"`, they can interpolate expressions
within {}. Types that can be used like this must have a format method (might change). Format, or f-strings,
don't actually evaluate to a string, but rather evaluate to an anonymous struct that has methods to convert
it all at once into a real string. Thus, f-strings do not allocate.

Number Literals

Number literals represent any sequence of integers between 0 and 9. They can be represented in decimal form, fraction form, or exponential form as well.

Ex. 123, 532, 6, 12.3, 43/23

Char Literals

Char literals represent any single character and make up string literals.

Ex. 'a', '1'

Bool Literals

Bool literals are booleans that represent true and false. In C* they are represented by the keywords true and false.

Unit Literal

Unit literals are a void type that return nothing

Struct Literal

Struct literals are the creation of new struct values by using the keyword "struct" and denoting the values of its fields.

Tuple Literal

Tuple literals contain a list of expressions that are separated by commmas and contained in parenthesis.

Ex. (3,2), (a,b)

Range Literal

Range literals are used to write ranges in values and are denoted using "..". A normal range literal is from an inclusive value a to an exclusive value b. They can be denoted using different variations such as using a "=" to denote inclusivity and "+" to denote a range from a inclusive to a+b exclusive.

Ex.

- `a..b`
- `a..=b`
- `a..+b`
- `a..`
- `..b`

Closure Literal

Closure literals are similar to normal functions but they can "enclose" over values in the current scope.

These are some example of how to create closures and how to call them. In particular:

- Closures have a generic, unnamed type. So when we take a closure as a parameter, we need to use a generic (this is because closure type depend on what they capture). You can also apply a type to a function type to get its return type, like `F(T)`.
- We can call a closure using the unified calling syntax: `.@call()`. Normal function calls are `()`, and we want to be explicit when we're actually calling a closure, so `.@call()` is needed. `.@call()` also works on normal functions, though, since all functions can be implicitly converted to non-capturing closures.
- The closure syntax is very similar to function syntax, with a few differences:
    - The return expression does not have to be a block like in normal functions; it can directly use an expression. Note that functions effectively just return a block. That's how `try` blocks work, for example.
    - Argument and return types are inferred, though they can still be specified if you want. This is because they are more local, and thus documented types are not as necessary.
    - If you want to capture variables, you specify an anonymous struct literal before the `fn`. This follows the same normal rules for struct literals, but you don't have to specify the type, since the type is anonymous. Then that struct's fields are available within the closure as variables.

The way closures are implemented is by creating an anonymous struct of the captured closure context. Then there is a method on that struct that takes the closure arguments and returns the closure body with the context struct destructured inside (so its variables are in scope). This is what is called by `.@call()`. Note that there are no indirect function calls, boxing, or allocations involved in this, but it requires the use of generics. If nothing is captured by a closure, though, then it can be cast to a function pointer: `fn(T, U): R`, which can be called indirectly and passed to C over FFI. The same is true of normal functions.

Func Literal

Func literals are function literals. A func literal is a closure so it can reference vairables that have been defined in a surrounding function. It can also share variables between the function literal and the surrounding function. Func literals can also be passed into other functions as parameters.

Algebraic Data Types

C* has `struct`s for product types and `enum`s for sum types. This is very powerful combined with pattern matching. `enum`s in particular, which are like tagged unions, are much safer and correct compared to C unions. These data types are also fully zero-cost; there is no automatic boxing, and the same performance as C can be easily be achieved. Sometimes even better, because the layout of compound types is unspecified in C*.

For example, you can do this to make a copy-on-write string.

```
struct String {
    ptr: *u8,
    len: usize,
}

struct StringBuf {
    ptr: *u8,
    len: usize,
```

```
        cap: usize,
    }

    enum CowString {
        Borrowed(String),
        Owned(StringBuf),
    }
```

Structs

C* has `struct`s for product types and `enum`s for sum types. This is very powerful combined with pattern matching. `enum`s in particular, which are like tagged unions, are much safer and correct compared to C unions. These data types are also fully zero-cost; there is no automatic boxing, and the same performance as C can be easily be achieved. Sometimes even better, because the layout of compound types is unspecified in C*.

For example, you can do this to make a copy-on-write string.

```
    struct String {
        ptr: *u8,
        len: usize,
    }

    struct StringBuf {
        ptr: *u8,
        len: usize,
        cap: usize,
    }

    enum CowString {
        Borrowed(String),
        Owned(StringBuf),
    }
```

Generics

C* supports generic types and values, but they are at this point unconstrained. That is, they are like C++'s concept-less templates. They are always monomorphic, except when the exact same code can be shared (no boxing ever). They are not currently higher-kinded. Types and functions can be generic over both types and values, like this:

```
    enum Option<T> {
        None,
        Some(T),
    }

    enum ShortVec<T, N: u8> {
        Inline {
```

```
        array: [T; N],
        len: u8,
    },
    Allocated {
        ptr: Option<*T>,
        len: usize,
        cap: usize,
    },
}

fn short_vec_len<T, N: u8>(v: *ShortVec<T, N>): usize = {
    v.match {
        Inline {len, _} => len.@cast(),
        Allocated {len, _} => len,
    }
}
```

Statements and Expressions

Statements

Due to the expression oriented nature of C* all control flow statements are themselves expressions.

If-Else Statements

If-Else statements execute one of two cases. The first consists of typical C-style semantics wherein we have:

```
if (expr1)
    statement1
else
    statement2
```

Both `statement1` and `statement2` must evaluate to the unit type. Like C the `else` part of the If-Else control flow block is optional. In addition to the C-style control flow we also can have:

```
if (expr1)
    expr2
else
    expr3
```

In both cases the expressions in the `if` statement are evaluated and in the case they evaluate to a non-zero value the flow of execution continues down that path otherwise the body of the `else` statement is executed.

C* utilizes the same mechanism to eliminate ambiguity relating to a "dangling-else". An `else` is grouped to the nearest `if`. In the case of:

```
let i: i32 = 6;
let j: i32 = 7;

if(i > 4)
    if(j > i)
        println!("j is greater than i!");
    else
        println("j is less than or equal to i!");
```

While the indentation and print statements make clear which `if` the `else` clause is grouped with it should be clear that barring the use of additional brackets to direct control flow the `else` is grouped to the nearest `if` above it.

For Statements

For statements can execute over a range in the case of:

```
for season in seasons.iter()
    println!(season);
```

In addition to the use of an explicit iterator it is also possible to use a range literal to bound the execution of the body of a for loop in the case of:

```
let mut day_ = 1;
for x in 1..365{
    println!("Day {} of 365", x);
}
```

While Statements

Execution of the body of a while statement continues until the expression labeled `expr1` evaluates to zero. For example:

```
while(expr1){
    statement1;
}
```

Similiar to `if` statements due to the expression oriented nature of C* `statement1` must evaluate to the unit type and it is possible to replace `statement1` with `expr2`.

Defer

To aid in resource handling, C* has a defer keyword. defer defers the following statement or block until the function returns, but will run it no matter where the function returns from (but not panics/aborts) (actually, the defer will run when its block exits, but its easier to just think about function blocks first).

For example, you can use this to ensure you correctly clean up resources in a function:

```
extern "C" fn open(path: *u8, flags: i32): i32;
extern "C" fn close(fd: i32): i32;

fn open_file_in_dir(dir: *[u8], filename: *[u8]): Result<i32, String> try =
{
    let mut path = Vec.new(Mallocator());
    defer path.free();
    try {
        if (dir.len() > 0) {
            path.extend(dir).?;
            path.push(b'/').?;
        }
        path.extend(filename).?;
        path.push(0).?;
    }.map_err(fn(_) "alloc error").?;

    let path = path.as_ptr();
    let fd = open(path, O_RDWR).match {
        -1 => Err("open failed"),
        fd => fd,
    }.?;
    defer println(f"opened {fd}");
    return fd;
}
```

In this example, you have to allocate a path to store the directory and filename you combine, and then open that path and return the file descriptor if it was successful. You have to clean up the memory allocation, though, and do that while still handling all the allocation errors and the open error. The latter can be done elegantly with try and .?, but if you mix in the path.free(), you'd have to run it before every error return, which means you have to duplicate it and not use .? anymore.

Instead, you can use defer for this. No matter where you return from the function, it will run its statement right before that. You can also use defer for any statement, not just resource cleanup, like logging for example.

However, sometimes you want to cancel a defer:

```
struct FilePair {
    fd1: i32,
    fd2: i32,
}

fn open_two_files(path1: *[u8], path2: *[u8]): Result<FilePair, String> try
```

```
= {
    let fd1 = open_file_in_dir(b"", path1).?;
    close: defer close(fd1);
    let fd2 = open_file_in_dir(b"", path2).?;
    close: defer close(fd2);
    println(f"opened {fd1} and {fd2}");
    undefer close;
    FilePair {fd1, fd2}
}
```

In this example, you want open two files and return them if successfull. If only one is successful, though, that's an error and you should close the first one before returning the error. In order to do that cleanly, you can use the undefer keyword, which cancels an earlier labeled defer, in this case labeled close.

defer and undefer are actually syntax sugar for something a bit more low-level and wordy:

```
fn open_two_files(path1: *[u8], path2: *[u8]): Result<FilePair, String> try
= {
    let fd1 = open_file_in_dir(b"", path1).?;
    let close1 = {fd1} fn() close(fd1);
    let close1 = close1.@defer());
    let fd2 = open_file_in_dir(b"", path2).?;
    let close2 = {fd1} fn() close(fd1);
    let close2 = close2.@defer());
    println(f"opened {fd1} and {fd2}");
    let close = [close2, close1];
    close.undo();
    FilePair {fd1, fd2}
}
```

That is, .@defer() places the closure on the stack and returns a Defer struct, which can be undone with Defer.undo() ([Defer].undo() just maps Defer.undo() over the array). Defer.undo() sets a bit in the Defer struct that it's been undone. Then when the stack unwinds, any none-undone Defers on the stack are run.

Expressions and Operators

Unary Operators

Unary operators are operators that can act on an expression. C* uses the unary operators "-" and "!" to represent negation and the logical not repectively. "-" negates a number literal such as

```
let x = -2
```

The logical not "!" represents negation for bool literals or boolean expressions such as

```
let a = true
let b = !a
```

where b returns the value of false.

Binary Operators

A binary operator acts on two expressions and can be show as follows:

Binary operator = expr * operator * expr

Assignment operator

The assignment operator stores values into vairables. It uses the keyword "let" and the = symbol so that the left side variable stores the expression on the right.

Ex.

```
let a = 23 // a stores the value 23
```

Arithmetic Operator

- The addition operator "+" adds two values of the same type. Automatic type conversion is applied when adding two number literals and can also be applied to string addition.

Ex.

```
1 + 2 // 3
12.3 + 10 // 22.3
"string" + "test" // "stringtest"
```

- The subtraction operator "-" subtracts two values of the same type. Automatic type conversion is applied when adding two number literals.

Ex.

```
1 - 2 // -1
12.3 - 10 // 2.3
```

- The multiplication operator "*" multiplies two values of the same type. Automatic type conversion is applied when adding two number literals.

Ex.

```
1 * 2 // 2
12.3 * 10 // 123
```

- The division operator "/" divides two values of the same type. Automatic type conversion is applied when adding two number literals.

Ex.

```
1 / 2 // .5
12.3 / 10 // 1.23
```

- The modulus operator "%" takes the modulus of two values of the same type. Automatic type conversion is applied when adding two number literals.

Ex.

```
1 % 2 // 1
12.3 % 10 // 2.3
```

Relational Operators

Relational operators represent how the operands relate to each other. Each expression using a relational operator has two values as inputs and outputs either true or false. The relational operators are: ==, !=, <, >, <=, >=, &, |.

```
1 < 2 // true
1 > 2 // false
1 != 2 // true
1 == 2 // false
true | false // true
true & false // false
```

Functions

Functions are a type of statement that can be declared one of two ways:

```
fn name(parameters): return type = body
```

or

```
fn name(parameters): return type = { body }
```

It takes in a list of parameters and returns a value based on the expression. Functions can be written with or without specifying the return type.

Ex.

```
fn hello(): string = "hello world"

fn adding(a, b): = { return a + b }
```

Pattern Matching

Instead of having a `switch` statement like in C, C* has a generalized `match` statement, which can be used to match many more expressions, including integers (like in C), `enum` variants, dereferenced pointers, slices, arrays, and strings. Also, there is no fall-through, but `match` cases can be combined explicitly.

Furthermore, just like you can destructure to pattern match in a `match` statement, you can also do the same as a general statement, like in a `let`. It's like an unconditional `match`.

```
let cow = CowString::Borrowed("🐄");
let len = match cow {
    Borrowed(s) => s.len(),
    Owned(s) => s.len(),
};
let String {ptr, len} = "🐄";
```

Note that string literals are of the `String` type similarly defined as above, and you can redeclare/shadow variables like `len`.

Methods

C* has associated functions and simple methods, though these are largely syntactic sugar. To declare these for a type, simply write:

```
struct Person {
    first_name: String,
    last_name: String,
}

impl Hello {

    fn new(first_name: String, last_name: String): Self = {
        Self {first_name, last_name}
    }

    fn say_hi1(self: Self) = {
        print(f"Hi {self.first_name} {self.last_name}");
```

```
    }

    fn say_hi1(self: *Self) = {
        print(f"Hi {self.last_name}, {self.first_name}");
    }

    fn remove_last_name(self: *mut Self) = {
        self.last_name = "";
    }

}

fn main() {
    let mut person = Person.new("Khyber", "Sen");

    {
        person.say_hi1();
        person.&.say_hi2();
        person.&mut.remove_last_name();
        person.say_hi1();
    }
    {
        Person.say_hi1(person);
        Person.say_hi2(person.&);
        Person.remove_last_name(person.&mut);
        Person.say_hi1(person);
    }
}
```

In this example, we first declared a `struct Person`, and then an `impl` block for `Person` to define methods/associated functions for it. Note that this `impl` block can be anywhere, even in other modules.

In the `impl` block, we first declared an associated function `Person.new`, which is just a normal function but namespaced to `Person`. Similarly, the other three methods are just normal functions, too, as seen when we call them explicity in the second block in `main`. But we can also use `.` syntax to call them, which just allows us to explicitly name `Person`.

Inside an `impl` block, we can also use the `Self` type as an alias to the type being implemented. This is especially useful with generics.

Note that the `.&` and `*Self` are explicit, because we wan't these kinds of possible costs to be noted explicitly. For example, `Person.say_hi1` takes `Self` by value, which means it must copy the `Person` every time. If `Person` were a much larger struct, this could be very expensive and we don't want to hide that information. Also, the difference between `.&` and `.&mut` is explicit to make mutability explicit everywhere.

Postfix

Most unary operators and keywords can be used postfix as well.

- `.if {}`

- `.if {} else {}`
- `.match {}`
- `.for {}`
- `.*` for dereference
- `.&` for pointer to
- `.&mut` for mutable pointer to
- `.!` for negation
- `.@()` for builtins, like as (casting), size_of, etc.
  - `.@cast(T)`: convert to `T`, like an int to float cast, or an int widening cast
  - `.@ptr_cast<T>()`: cast a pointer like `*T` to `*U`
  - `.@bit_cast<T>()`: reinterpret the bits, like from `u32` to `f32`
  - `.@size_of()`: size of a type
  - `.@align_of()`: alignment of a type
  - `.@call(func)`: call a function or closure in a unified syntax

Combined with everything being an expression, `match`, and having methods, this makes it much easier to write programs in a very fluid style.

Furthermore, and perhaps most importantly in practice, this makes autocompletion vastly better, because an IDE can narrow down what you may type next based on the type of the previous expression. This can't be done with postfix operators and functions (rather than methods). You get to think in one forward direction, rather than having to jump from some prefix keywords to some postfix methods and fields.

Slices

C* also has slices. These are a pointer and length, and are much preferred to passing the pointer and length separately, like you usually have to do in C.

They are implemented like this (not actually, but similarly):

```
struct Slice<T> {
    ptr: *T,
    len: usize,
}
```

But they can be written as `*[T]`. Actually, slices are unsized types, so their type is just `[T]`, but usually `*[T]` is used and that is what's equivalent to the above `Slice<T>`.

Unlike pointers like `*T`, slices can be indexed. By default, using the indexing operator, this is bounds checked for safety, but there are also unchecked methods for indexing. Usually, though, bounds checking can be elided during sequential iteration, so the performance hit is minimal, and can be side-stepped if really needed.

Slices can also be sliced to create subslices by indexing them with a range (e.x. `[1..10]` or `[1..]`). Again, this is bounds checked by default.

Monadic Error-Handling

There are no exceptions in C*, just like C. It uses return values for error handling, similarly to C. But C* has much better support for this using the `Option` and `Result` types.

The definitions of these types are:

```
struct Option<T> {
    None,
    Some(T),
}

struct Result<T, E> {
    Ok(T),
    Err(E),
}
```

That is, `Option` represents an optional value, and `Result` represents either a successful `Ok` value or an error `Err` value.

There is special syntactic support for using these two monadic types for error-handling using the `.?` postfix operator in `try` blocks:

```
struct IndexError {
    index: usize,
}

fn get_by_index<T>(a: *[T], i: usize): Result<T, IndexError> {
    if (i < a.len()) {
        Ok(a[i])
    } else {
        Err(IndexError {index: i})
    }
}

struct IndexPair {
    first: usize,
    second: usize,
}

fn get_two_by_index<T>(a: *[T], i: usize, j: usize): Result<T, IndexError>
try = {
    let first = try {
        get_by_index(a, i).?
    };
    let second = get_by_index(a, j).?;
    IndexPair {first, second}
}
```

This desugars to

```
fn get_two_by_index<T>(a: *[T], i: usize, j: usize): Result<T, IndexError>
={
    let first = try {
        get_by_index(a, i).match {
            Ok(i) => i,
            Err(e) => return Err(e),
        }
    };
    let second = get_by_index(a, j).match {
        Ok(i) => i,
        Err(e) => return Err(e),
    }
    Ok(IndexPair {first, second})
}
```

As you can see, without the try `.?` operator and `try` blocks, doing all the error handling with just `match` quickly becomes tedious. This is also kind of like a monadic `do` notation, except it is in C* limited to just the monads `Option<T>`, and `Result<T, E>` (over `T`).

Note also that `try` blocks can be specified at the function level as well as normal blocks.

Uncatchable Panics

While monadic error-handling with `Option` and `Result` is usually superior, there are still cases where you have unrecoverable errors (maybe you don't want to handle out of memory conditions), or where you'd rather just end the program than handle the error. In this case, you can `panic`, which will print an error message and immediately `abort`.

To do this with an `Option` or `Result`, you can just call `.unwrap()`, which will panic if it was `None` or `Err` and return the `Some` or `Ok` value.

There is no language-supported unwinding. `abort` is immediately called after a panic, and only the OS cleans things up. Nothing is stopping you from calling `setjmp` and `longjmp` from C, but no unwinding of `defer` statements is done, and it may result in undefined behavior. There is no undefined behavior, however, in a normal panic because you just simply `abort`.

Operator Precedence

The table below shows the operator precedence for binary and unary operators from lowest precedence to highest precedence.

Operator	Description	Associativity
;	sequencing	Left
=	assignment	Right
.	access	Left
		or

Operator	Description	Associativity
&	and	Left
== !=	equality/inequality	Left
< > <= >=	comparison	Left
+-	addition/subtraction	Left
*/	multiplication/division	Left
-	negation	Right
!	logical NOT	Right
?	conditional	Left

In C* generics have a higher precedence than comparison thus removing ambiguity from "< >".

Examples

GCD

Here is how you write simple algorithms like GCD in C*:

```
fn gcd(a: i64, b: i64): i64 = {
    (fn gcd(a: u64, b: u64): u64 = {
        match b {
            0 => b,
            _ => gcd(b, a % b),
        }
    })(a.abs(), b.abs()).@cast(i64)
}
```

Systems Programming

Here is an example program in C* for part of a simple HTTP/1.0 server, equivalent to part0 of hw3 in Jae's OS class (https://gist.github.com/RyanLee64/hash-redacted). It showcases many of C*'s notable features, like enums, methods, generics, defer, expression-orientedness, postfix operators, pattern matching, closures, monadic error handling, and byte, c, and format strings.

That code (the ported part) is ~230 LOC, while the C* below is only ~80 LOC, and it is more correct in error handling and edge cases, faster in places (though IO dominates here), and the business logic stands out more (while less important aspects like errors, resource cleanup, allocations, and string handling stay in the background). That is, C* allows you to be simulatenously more expressive while still staying correct and explicit, and the performance is just as good if not better.

```
enum Status {
    Ok,
    NotImplemented,
```

```
        BadRequest,
        // rest skipped for brevity
    }

    struct RequestLine {
        method: *[u8],
        uri: *[u8],
        version: *[u8],
    }

    impl RequestLine {
        fn check(self: *Self): Result<(), Status> try = {
            let Self {method, uri, version} = self.*;
            match (method, version) {
                (b"GET", b"HTTP/1.0" | b"HTTP/1.1") => {},
                _ => Err(Status.NotImplemented).?,
            }
            if uri.starts_with(b'/').! || uri.equals(b"..") ||
    uri.contains(b"/../") {
                Err(Status.BadRequest).?;
            }
        }
    }

    fn main(): Result<(), AnyError> try = {
        let (port, web_root) = std.env.argv().match {
            [_, port, web_root] => (port.parse<u16>().?, web_root),
            [program, ...] => Err(f"usage: {program} <server_port>
    <web_root>").?,
        };
        let server_socket = Socket.new(PF_INET, SOCK_STREAM, IPPROTO_TCP).?;
        defer server_socket.&.close();
        server_socket.&.bind(SocketAddr {
            family: AF_INET,
            addr: InetAddr {
                addr: INADDR_ANY.to_be(),
            },
            port: port.to_be(),
        }).?;
        server_socket.&.listen(5).?;
        let mut request_line_buf = Vec.new();
        defer request_line_buf.free();
        let mut line_buf = Vec.new();
        defer line_buf.free();
        loop try {
            let client_socket = server_socket.&.accept().?;
    client_socket_close:
            defer client_socket.&.close();
            let mut client_stream = fdopen(client_socket.fd, c"r").?;
            undefer client_socket_close; // stream (`FILE *` in C) takes
    ownership
            defer client_stream.&.close();
            let line_or_status = try {
                // read and parse request line
```

```
            let line = client_stream.&mut.read_line(buf.&mut)
                .map_err(fn(_) Status.BadRequest).?
                .split(fn(b) " \t\r\n".contains(b)).match {
                    [method, uri, version] => RequestLine { method, uri,
version },
                    _ => Err(Status.NotImplemented).?,
                };
            line.&.check().?;
            // read headers, skip them
            loop {
                client_stream.&mut.read_line(buf.&mut)
                    .map_err(fn(_) Status.BadRequest).?
                    .match {
                        "\n" | "\r\n" => break,
                        _ => {},
                    }
            }
            line
        }
        let (line, status) = match line_or_status {
            Ok(line) => (line, Status.Ok),
            Err(status) => (RequestLine { method: b"", uri: b"", version:
b"" }, status),
        };
        client_socket.write(f"HTTP/1.0 {status.code()}
{status.reason()}\r\n\r\n").?;
        match line_or_status {
            Ok(_) => handle_request(web_root, line.uri, client_socket).?,
            Err(_) => client_socket.write(f"<html><body>\n<h1
{status.code()} {status.reason()}</h1>\n</body></html>").?;
        }
        eprintln(f"{client_socket.addr} \"{line.method} {line.uri}
{line.version}\" {status.code()} {status.reason()}").?;
    }
}
```