CS 420 Advanced Programming Languages
Fall Semester, 2022

Doc 8 Macros & Threads
Sep 22, 2022

C Macros

```
#define BUFFER_SIZE 1024
```

```
foo = (char *) malloc (BUFFER_SIZE);
```

```
foo = (char *) malloc (1024);
```

C Macro

```
#define WARN_IF(EXP) \
do { if (EXP) \
        fprintf (stderr, "Warning: " #EXP "\n"); } \
while (0)
```

```
 WARN_IF (x == 0);
```

```
do { if (x == 0)
        fprintf (stderr, "Warning: " "x == 0" "\n"); } while (0);
```

C Macros

Macros are expanded by the preprocessor

The result is sent to the compiler

Rust Macros

```
assert_eq!(5,6);
```

```
thread 'sample_macro' panicked at 'assertion failed: `(left == right)`
  left: `5`,
 right: `6`', src/main.rs:67:5
```

Rust Macros

```
let current_line = line!();
println!("defined on line: {current_line}");

let current_col = column!();
println!("defined on column: {current_col}");

let current_file = file!();
println!("defined in file: {current_file}");
```

```
defined on line: 67
defined on column: 23
defined in file: src/main.rs
```

Rust Macros

Rust compiler expands macros

Macros are applied to the abstract syntax tree

- Permits us to do far more than C marcos

```
macro_rules! say_hello {
    () => {
        println!("Hello!");
    };
}

say_hello!();
```

```
macro_rules! create_function {
    ($func_name:ident) => {
        fn $func_name() {
            // The `stringify!` macro converts an `ident` into a string.
            println!("You called {:?}()",
                     stringify!($func_name));
        }
    };
}

create_function!(foo);
create_function!(bar);

fn main() {
    foo();
    bar();
}
```

```
macro_rules! calculate {
    (eval $e:expr) => {
        {
            let val: usize = $e; // Force types to be integers
            println!("{} = {}", stringify!{$e}, val);
        }
    };
}
```

```
calculate! {
    eval 1 + 2
}
```

```
1 + 2 = 3
```

```
calculate! {
    eval (1 + 2) * (3 / 4)
}
```

```
(1 + 2) * (3 / 4) = 0
```

JSON

http://www.json.org/

JavaScript Object Notation

data-interchange format

rfc 4627

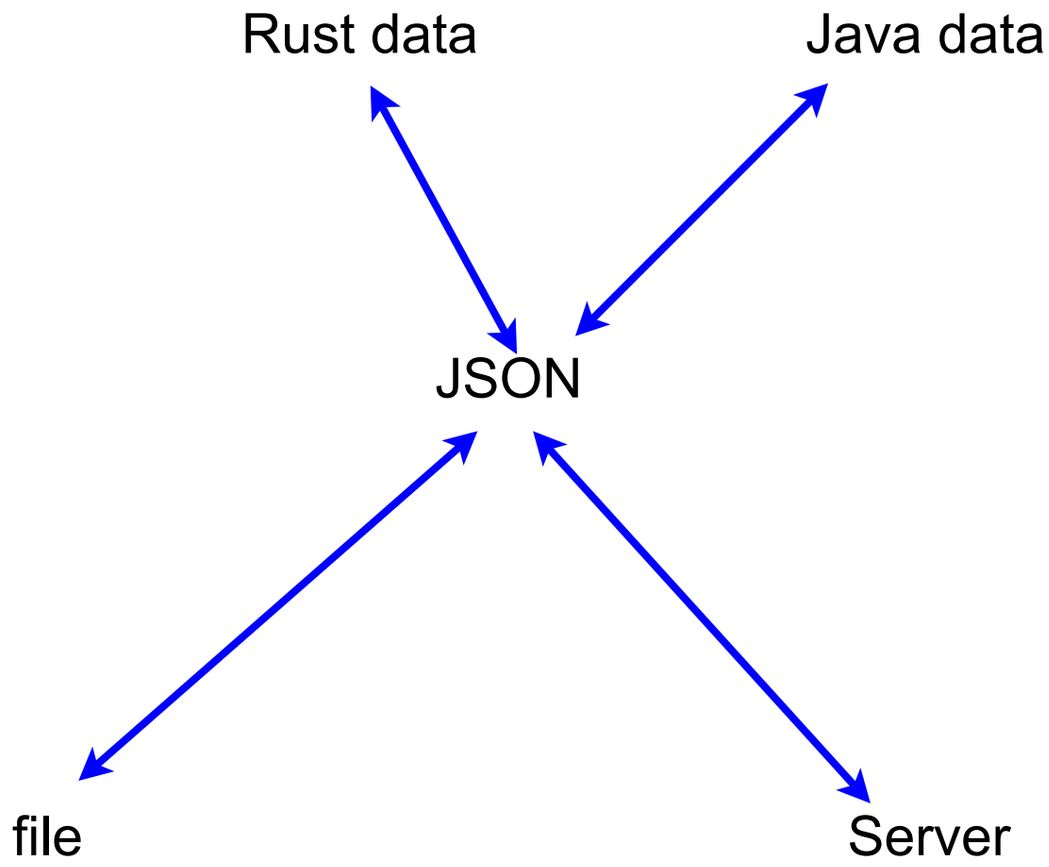
Maps to/from strings

- null
- true, false
- number
- string
- array
- objects

Implementations in

C, C++, C#, D, E, Java, Objective C
Cold Fusion, Delphi, Erlang, Haskell
JavaScript, Lisp, LotusScript, Perl,
PHP, Pike, Prolog, Python, Ruby, Smalltalk

Data Interchange



Arrays

[1, 2, 3]

["cat", 2, true]

[23.4, ["dog", null], 10]

JSONArrays can hold any valid JSON data type

Number & String & Constants

123
32.4
0.12
2.3e3
+5
-6
5.93e-2

"a string"

"a string with \" a quote char"

\ escape - normal special char
\n
\t
\\
\/
\b
\f
\u

null
true
false

Object (Dictionary)

{"key": "value"}

keys have to be strings

value can be any legal JSON data type

{"name": "Roger", "age": 21}

{"id":2,"office":"GMCS 407B","phone":"619-594-6191","email":"beck@cs.sdsu.edu","rating":{"average":5.0,"totalRatings":1},"firstName":"Dr. Leland","lastName":"Beck"}

Valid JSON Document

One top level item

- Array
- Object

Changed

- Any element can be top level

Sample JSON

```
[
    {
        "name": "Jim Blandy",
        "class_of": 1926,
        "major": "Tibetan throat singing"
    },
    {
        "name": "Jason Orendorff",
        "class_of": 1702,
        "major": "Knots"
    }
]
```

How to convert JSON data to Rust data

Rust Enum

```
#[derive(Clone, PartialEq, Debug)]
enum Json {
    Null,
    Boolean(bool),
    Number(f64),
    String(String),
    Array(Vec<Json>),
    Object(Box<HashMap<String, Json>>)
}
```

Basic Idea for Macro

```
macro_rules! json {
    (null)    => { Json::Null };
    ([ ... ]) => { Json::Array(...) };
    ({ ... }) => { Json::Object(...) };
    (???)     => { Json::Boolean(...) };
    (???)     => { Json::Number(...) };
    (???)     => { Json::String(...) };
}
```

```
macro_rules! json {
    (null) => {
        Json::Null
    };
  ([ $( $element:tt ),* ]) => {
      Json::Array(vec![ $( json!($element) ),* ])
  };
 ({ $( $key:tt : $value:tt ),* }) => {
        Json::Object(Box::new(vec![
            $( ($key.to_string(), json!($value)) ),*
        ].into_iter().collect()))
    };
    ( $other:tt ) => {
        Json::from($other)  // Handle Boolean/number/string
    };
}
```

```
impl From<String> for Json {
    fn from(s: String) -> Json {
        Json::String(s)
    }
}

impl<'a> From<&'a str> for Json {
    fn from(s: &'a str) -> Json {
        Json::String(s.to_string())
    }
}
```

```
macro_rules! impl_from_num_for_json {
    ( $( $t:ident )* ) => {
        $(
            impl From<$t> for Json {
                fn from(n: $t) -> Json {
                    Json::Number(n as f64)
                }
            }
        )*
    };
}
```

```
impl_from_num_for_json!(u8 i8 u16 i16 u32 i32 u64 i64 u128 i128
usize isize f32 f64 );
```

```
let mut json = json!( 5);
assert_eq!(json, Json::Number(5.0))

let cat = "cat";
json = json!( cat);

assert_eq!(json, Json::String("cat".to_string()));

json = json!(null);
assert_eq!(json, Json::Null);

json = json!([1, 2, 3]);
assert_eq!(json, Json::Array(vec![Json::Number(1.0), Json::Number(2.0), Json::Number(3.0)]));

json = json!({ "a": 1, "b": 2 });
assert_eq!(json,
        Json::Object(Box::new(vec![("a".to_string(),
        Json::Number(1.0)),
        ("b".to_string(),
        Json::Number(2.0))].into_iter().collect())));
```

Rc & Arc

Rc

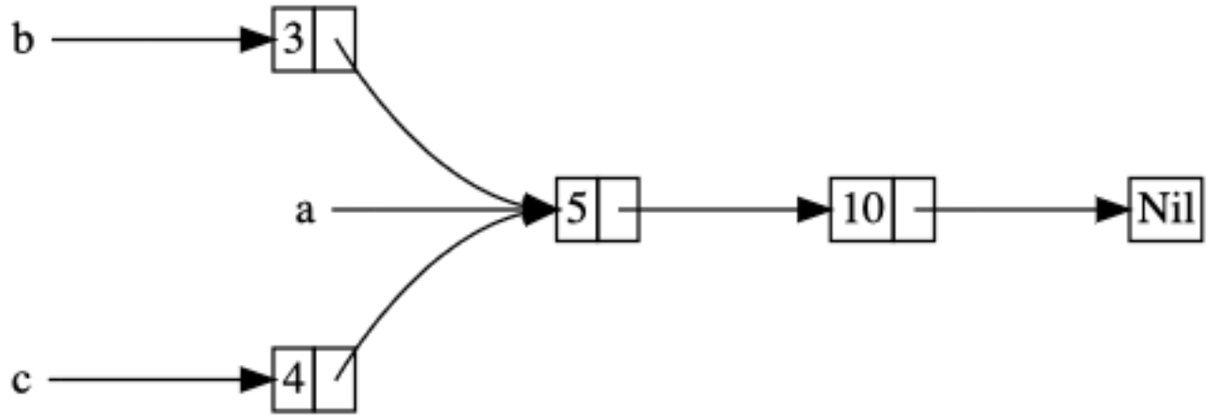
- Reference Counting

Use Rc when

- Data on the heap for multiple parts of our program to read
- Can’t determine at compile time which part will finish using the data last

Arc

- Atomic Reference Counting
- Same as Rc but for use with threads

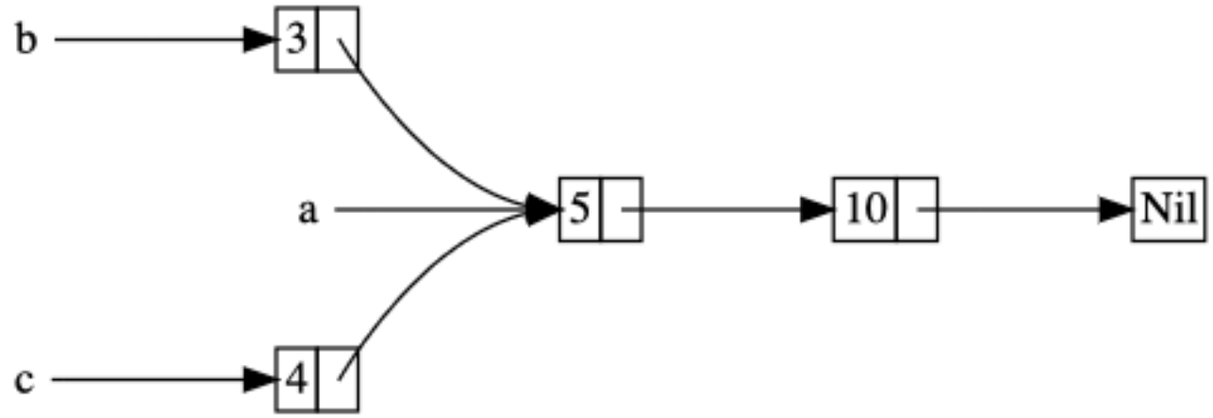



```
enum List {
    Cons(i32, Box<List>),
    Nil,
}
```

```
use crate::List::{Cons, Nil};

fn main() {
    let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
    let b = Cons(3, Box::new(a));
    let c = Cons(4, Box::new(a));
}
```

a moved here

Compile error




```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}
```

```
use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}
```

Rc::clone(&a)

Does not make a deep copy

Increases the reference count

```
fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    println!("count after creating a = {}", Rc::strong_count(&a));                1
    let b = Cons(3, Rc::clone(&a));
    println!("count after creating b = {}", Rc::strong_count(&a));                2
    {
        let c = Cons(4, Rc::clone(&a));
        println!("count after creating c = {}", Rc::strong_count(&a));            3
    }
    println!("count after c goes out of scope = {}", Rc::strong_count(&a));       2
}
```

Standard Prelude

Rust libraries can have a prelude

Crates, stunts, etc. that are imported automatically

std::marker::{Copy, Send, Sized, Sync, Unpin}
std::ops::{Drop, Fn, FnMut, FnOnce}
std::mem::drop
std::boxed::Box
std::borrow::ToOwned
std::clone::Clone,
std::cmp::{PartialEq, PartialOrd, Eq, Ord}.
std::convert::{AsRef, AsMut, Into, From}
std::default::Default
std::iter::{Iterator, Extend, IntoIterator, DoubleEndedIterator, ExactSizeIterator},
std::option::Option::{self, Some, None}
std::result::Result::{self, Ok, Err}
std::string::{String, ToString}
std::vec::Vec
std::convert::{TryFrom, TryInto},
std::iter::FromIterator

Concurrency & Parallelism

Concurrency

Dealing with lots of things at once

Parallelism

Doing lots of things at once

Timeslicing

Running multiple processes on one CPU core

Each process is run for a small time, then swapped out for another process

With multicore CPUs

- Multiple processes can run simultaneously

```
fn looper(label: &str,  n: i32) {
    for k in 0..n {
        println!("{}: {}", label, k);
    }
}


use std::iter::zip;

fn main() {
   let a = zip([5, 5, 5], ["a", "b", "c"]);
    for (amount, label) in a {
        looper(label, amount);
    }
}
```

```
a: 0
a: 1
a: 2
a: 3
a: 4
b: 0
b: 1
b: 2
b: 3
b: 4
c: 0
c: 1
c: 2
c: 3
c: 4
```

```
fn looper(label: &str,  n: i32) {
    for k in 0..n {
        println!("{}: {}", label, k);
    }
}
```

Output

```
use std::iter::zip;
use std::thread;

fn main() {
   let a = zip([5, 5, 5], ["a", "b", "c"]);
    for (amount, label) in a {
        thread::spawn(move || {
            looper(label, amount);
        });
    }
}
```

```
  fn looper(label: &str,  n: i32) {
      for k in 0..n {
          println!("{}: {}", label, k);
      }
  }

use std::iter::zip;
use std::thread;

fn main() {
    let mut thread_handles = vec![];
    let a = zip([5, 5, 5], ["a", "b", "c"]);
    for (amount, label) in a {
        thread_handles.push(thread::spawn(move || looper(label, amount)));
        };
    for handle in thread_handles {
        handle.join().unwrap();
    }
}
```

Output

```
a: 0
a: 1
a: 2
a: 3
a: 4
c: 0
c: 1
c: 2
c: 3
c: 4
b: 0
b: 1
b: 2
b: 3
b: 4
```

```
use std::iter::zip;
use std::time::Duration;
use std::thread;
fn looper(label: &str,  n: i32) {
    for k in 0..n {
        println!("{}: {}", label, k);
        thread::sleep(Duration::from_millis(1));
    }
}

fn main() {
    let mut thread_handles = vec![];
    let a = zip([5, 5, 5], ["a", "b", "c"]);
    for (amount, label) in a {
        thread_handles.push(thread::spawn(move || looper(label, amount)));
        };
    for handle in thread_handles {
        handle.join().unwrap();
    }
}
```

Output

```
c: 0
a: 0
b: 0
c: 1
b: 1
a: 1
a: 2
b: 2
c: 2
a: 3
b: 3
c: 3
a: 4
c: 4
b: 4
```

Pleasingly Parallel (Embarrassingly Parallel)

Little or no effort is needed to separate the problem into a number of parallel tasks

Compute Sum

2 -3 5 9 1 7 8 2 1 6

2 -3 5 9 1

7 8 2 1 6

14

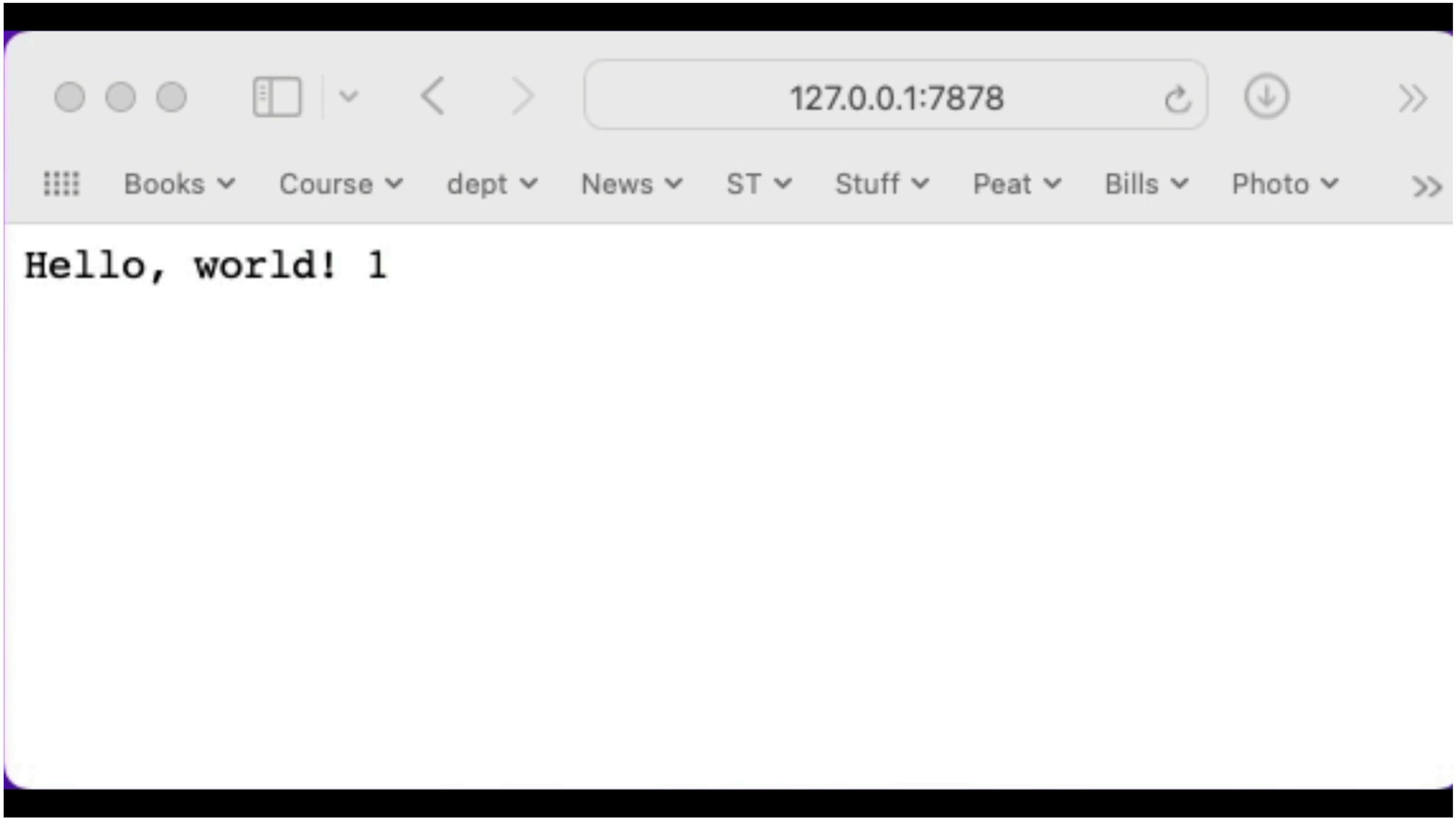
24

38

Simple Web Server

```
use std::{
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream},
};

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();
        handle_connection(stream);
    }
 }

fn handle_connection(mut stream: TcpStream) {
    let response = "HTTP/1.1 200 OK\r\n\r\nHello, world!";
    stream.write_all(response.as_bytes()).unwrap();
}
```

```
use std::{
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream},
};
use std::thread;

 fn main() {
     let mut counter = 0;
     let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

     for stream in listener.incoming() {
         let stream = stream.unwrap();
         counter += 1;
         thread::spawn(move || {
             handle_connection(stream,counter);
         });
     }
}            fn handle_connection(mut stream: TcpStream, count: i32) {
                 let response = "HTTP/1.1 200 OK\r\n\r\nHello, world! ".to_owned() +
                                 &count.to_string();
                 stream.write_all(response.as_bytes()).unwrap();
             }
```


127.0.0.1:7878
Books
Course
dept
News
ST
Stuff
Peat
Bills
Photo
Hello, world! 1