Low-Level Programming

M. Anton Ertl
TU Wien

High-level vs. low-level programming language

- Prevents some bugs
  ~~arbitrary overwrites~~
  ~~dangling references~~
- Prevents some exploits
  ~~Buffer overflow~~
- More convenient
  automatic memory reclamation
  arbitrary integers
- Limitations

- Access to bits and bytes
  across abstraction boundaries
- Systems programming
  run-time systems of HLLs
  Libraries
- Efficiency
  possible, not guaranteed
  knowledge useful for HLLs
- Less convenient
  fixed buffers or
  manual memory reclamation

Low-level programming languages

- Assembly language: machine-dependent, full power
- Forth: interactive, no type checking
- C
- C++
- Rust: elaborate type system
  tries to combine safety and low-level featurs

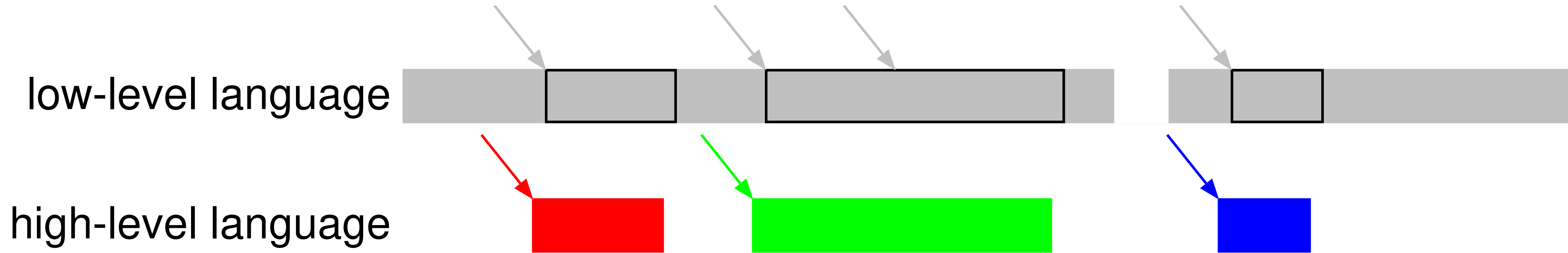
Debugging

- interactive testing
  use small words
- Stepping Debugger
  `dbg word`
  Only works with `gforth-itc`
- Tracer (`printf` debugging)
  `~~`
- Backtrace
- insert `assert( ... )`
- insert conditional tracers

IDE features

- `locate word`
  `edit word`
- `help word`
- `where word`
  `nw bw ( u ) ww`
- after a Backtrace
  `nt bt ( u ) tt`
- Decompiler
  `see word`
  `simple-see word`
  `see-code word`

Memory



- Address space (per-process)
- Accessible (mapped) memory (`pmap pid`)
  Readable **W**ritable e**X**ecutable

Example: linked list

```
public class ListQueueSimple {
  private ListNode head;

  public void add(String v) {
    if (head == null) {
      head = new ListNode(v, null);
    } else {
      ListNode last = head;
      while (last.next() != null) {
        last = last.next();
      }
      last.setNext(new ListNode(v, null));
    }
  }

  public String poll() {
    if (head != null) {
      String result = head.value();
      head = head.next();
      return result;
    }
    return null;
  }
}
```

Example: linked list (cont.)

```
public class ListNode {
  private String value;
  private ListNode next;

  public ListNode(String v, ListNode n) {
    value = v;
    next = n;
  }

  public String value() {
    return value;
  }

  public ListNode next() {
    return next;
  }

  public void setNext(ListNode n) {
    next = n;
  }
}
```

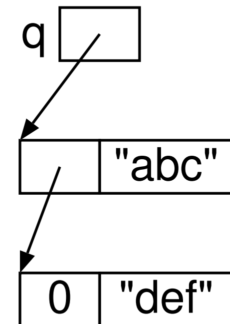
Example: linked list (cont.)

```
0
  field:  list-next
  2field: list-value
constant listnode

: list-add {: c-addr u list-addr -- :}
  list-addr begin {: list-addr1 :}
    list-addr1 @ dup while
      list-next repeat
  drop
  listnode allocate throw {: node :}
  0 node list-next !
  c-addr u node list-value 2!
  node list-addr1 ! ;
```

```
: list-poll {: list-addr -- c-addr u :}
  list-addr @ {: node :}
  node if
    node list-next @ list-addr !
    node list-value 2@
    node free throw
  else
    0 0
  then ;

\ Example
variable q 0 q !
"abc" q list-add
"def" q list-add
q list-poll type
```



Storage management

- Static allocation

  `create allot`

  Small systems ($<$ 64KB)

- Dynamic allocation with explicit deallocation

  `allocate free` regions/arenas

  Keep track of allocations: Memory leaks and double `free`

  Medium systems

- Dynamic allocation with automatic deallocation

  `allocate` and garbage collection

  Large systems ($>$ 16MB)

Storage management and APIs

`read-line ( c-addr u1 fid -- u2 flag wior )`

Reads a line from fid into the buffer at c-addr u1. Gforth supports all three common line terminators: LF, CR and CRLF. A non-zero wior indicates an error. A false flag indicates that 'read-line' has been invoked at the end of the file. u2 indicates the line length (without terminator): u2<u1 indicates that the line is u2 chars long; u2=u1 indicates that the line is at least u1 chars long, the u1 chars of the buffer have been filled with chars from the line, and the next slice of the line with be read with the next 'read-line'. If the line is u1 chars long, the first 'read-line' returns u2=u1 and the next read-line returns u2=0.

`slurp-fid ( fid -- c-addr u )`

c-addr u is the content of the file fid

Types

Who knows the type of data?

		run-time system	
		no	yes
compiler	no	Forth	Python, Lisp
	yes	C, Pascal	C++, Java, Rust

- Type knowledge of the programmer (e.g., sorted array)
- uniformity (Lisp) vs. specialization (Java)

Types: Checking

```
'a' 5 *
```

- How do you find type errors in Forth? Testing!
- With a little experience type errors are easy to find.
  experience in interpreting program output
  experience in writing test cases
  experience in writing programs for easy testability
- More intensive testing $\Rightarrow$ you also find other errors

Types: Checking

As programmers learned C with Classes or C++, they lost the ability to quickly find the "silly errors" that creep into C programs through the lack of checking. Further, they failed to take the precautions against such silly errors that good C programmers take as a matter of course. After all, "such errors don't happen in C with Classes." Thus, as the frequency of run-time errors caused by uncaught argument type errors goes down, their seriousness and the time needed to find them goes up.
Bjarne Stroustroup

Types: Overloading

```
int n;      n*3         n @ 3 *
float r;    r*3.0       r f@ 3.0e f*
int n;      n<3         n @ 3 <
unsigned u; u<3         u @ 3 u<
```

Types: Advantages and disadvantages of Forth

+ More uniformity: Better reusability
+ Extensions as libraries that would need type system extension in other languages (OOP, meta-programming)
+ Less complexity, e.g., for containers
– No type knowledge for garbage collection, marshalling etc.
– For changes affecting many lines static type checking would be useful
  Poor man's checking: Use new names for changed interfaces
  New names allow gradual changes

Object-oriented extension: features

- Dynamic Dispatch (virtual functions/methods)
- Instance Variables
- Single inheritance
- Static binding (C++: `A::m`)
- Basic class `object`
- `new`

Object-oriented extension: Usage (1)

```
object class
  cell var text
  cell var len
  cell var x
  cell var y
  method init
  method draw
end-class button

:noname ( o -- ) >r
 r@ x @ r@ y @ at-xy  r@ text @ r> len @ type ;
 button defines draw
:noname ( addr u o -- ) >r
 0 r@ x ! 0 r@ y ! r@ len ! r> text ! ;
 button defines init
```

Object-oriented extension: Usage (2)

```
button class
end-class bold-button

: bold   27 emit ." [1m" ;
: normal 27 emit ." [0m" ;

:noname bold [ button :: draw ] normal ; bold-button defines draw

button new Constant foo
s" thin foo" foo init
page
foo draw
bold-button new Constant bar
s" fat bar" bar init
1 bar y !
bar draw
```

Object-oriented extension: source code

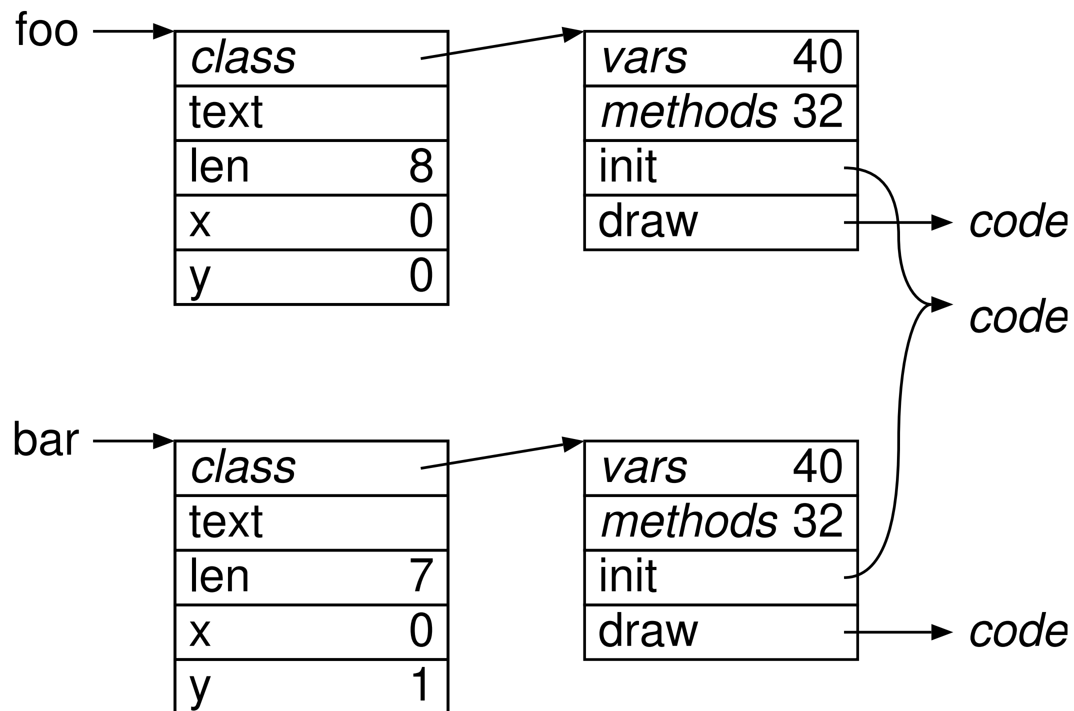
```
: method ( m v -- m' v ) Create  over , swap cell+ swap
  DOES> ( ... o -- ... ) @ over @ + @ execute ;
: var ( m v size -- m v' ) Create  over , +
  DOES> ( o -- addr ) @ + ;
: class ( class -- class methods vars ) dup 2@ ;
: end-class  ( class methods vars -- )
  Create  here >r , dup , 2 cells ?DO ['] noop , 1 cells +LOOP
  cell+ dup cell+ r> rot @ 2 cells /string move ;
: defines ( xt class -- ) ' >body @ + ! ;
: new ( class -- o )  here over @ allot swap over ! ;
: :: ( class "name" -- ) ' >body @ + @ compile, ;
Create object  1 cells , 2 cells ,
```

Explanation: `https://bernd-paysan.de/mini-oof.html`

Object-oriented extension: explanation

```
object class
  cell var text
  cell var len
  cell var x
  cell var y
  method init
  method draw
end-class button
button new Constant foo

button class
end-class bold-button
bold-button new Constant bar
```



Stateless control structures

```
... IF ... THEN
... IF ... ELSE ... THEN
BEGIN ... WHILE ... REPEAT
BEGIN ... UNTIL
CASE ... OF ... ENDOF ... OF ... ENDOF ... ENDCASE
?DO ... LOOP
```

Control structures: foundation

	forwards	backwards
Unconditional branch	`AHEAD ( -- orig )`	`AGAIN ( dest -- )`
Conditional branch	`IF ( -- orig )`	`UNTIL ( dest -- )`
branch target	`THEN ( orig -- )`	`BEGIN ( -- dest )`

```
: foo
  BEGIN ( C: dest )
     ... IF ( C: dest orig )
        ... THEN ( C: dest )
     ... UNTIL ( C: ) ;
```

Control Structures: ground floor

```
: ELSE ( compilation: orig1 -- orig2 ; run-time: -- ) \ core
    POSTPONE ahead
    1 cs-roll
    POSTPONE then ; immediate restrict

: WHILE ( compilation: dest -- orig dest ; run-time: f -- ) \ core
    POSTPONE if
    1 cs-roll ; immediate restrict

: REPEAT ( compilation: orig dest -- ; run-time: -- ) \ core
    POSTPONE again
    POSTPONE then ; immediate restrict
```

Control structures: Usage

```
: foo
  ... if ( C: orig1 )
    ...
  else ( C: orig2 )
    ... begin ( C: orig2 dest )
      ... while ( C: orig2 orig3 dest )
      ... repeat ( C: orig2 )
  then ;
```

Macros

```
: endif POSTPONE then ; immediate
: foo ... if ... endif ... ;

: (map) ( a n -- a a')   cells over + swap ;
: map<   postpone (map)  postpone ?do postpone i postpone @ ; immediate
: >map   1 cells postpone literal  postpone +loop ; immediate
: step  0  array 1000 map< + >map drop ;
```

Code generation: LITERAL COMPILE,

```
: foo [ 5 cells ] literal ;
: ]cells ] cells POSTPONE literal ;
: foo [ 5 ]cells ;

: twice ( xt -- )
  dup compile, compile, ;
: 2+ [ ' 1+ twice ] ;
: 4* [ ' 2* twice ] ;
```

Code generation: Gray

```
: compile-test \ set -- )
 postpone literal
 test-vector @ compile, ;

: generate-alternative1 \ -- )
 operand1 get-first compile-test
 postpone if
 operand1 generate
 postpone else
 operand2 generate
 postpone endif ;
```

Implementation

5 constant c
: foo c * ;

header of c
5

header of foo
lit
5
*
;s

code in gforth executable

lit code

;s code

```
add  $8,%rbx
imul 8(%r13),%r8
add  $8,%r13
mov  (%rbx),%rax
jmp  *%rax
```

5 constant c
: foo c * ;

header of c
5

header of foo
lit
5
*
;s

code generated at gforth run-time

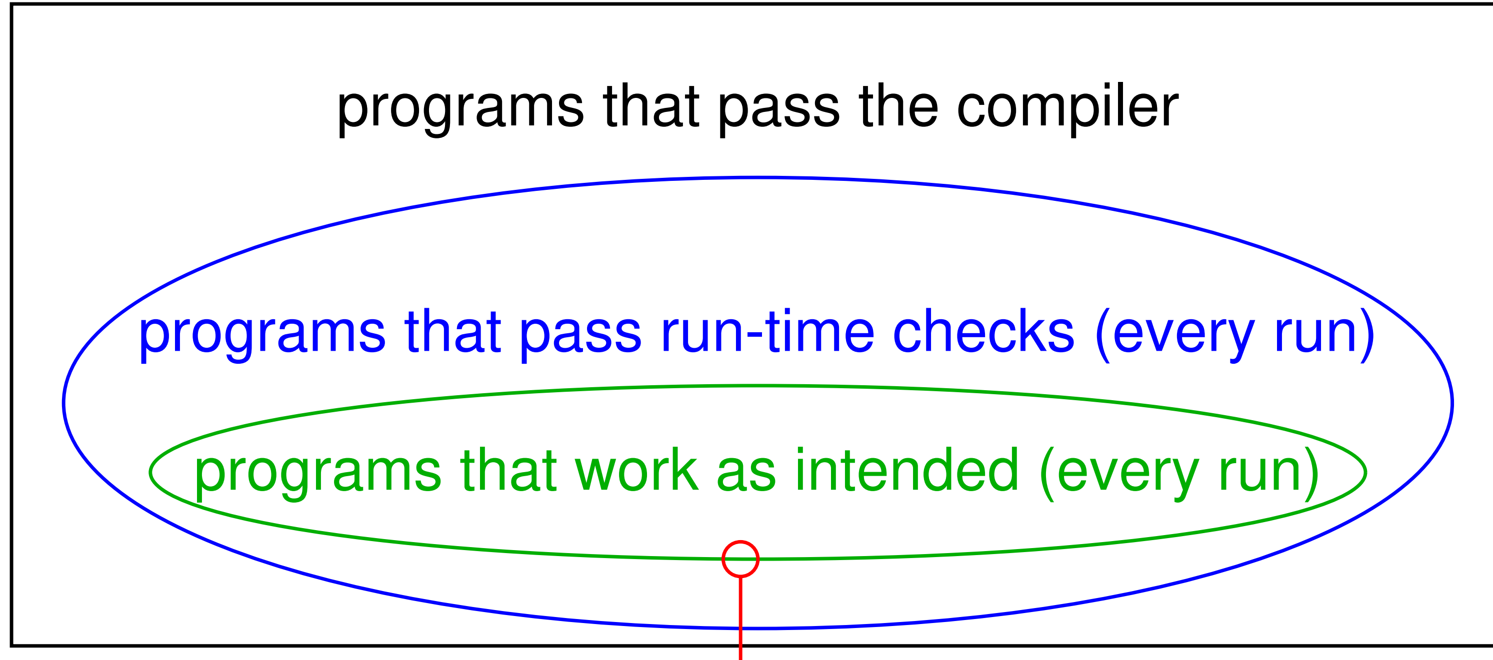
```
mov  8(%rbx),%r9
imul %r9,%r8
mov  (%r14),%rbx
add  $8,%r14
mov  (%rbx),%rax
jmp  *%rax
```

code in gforth executable

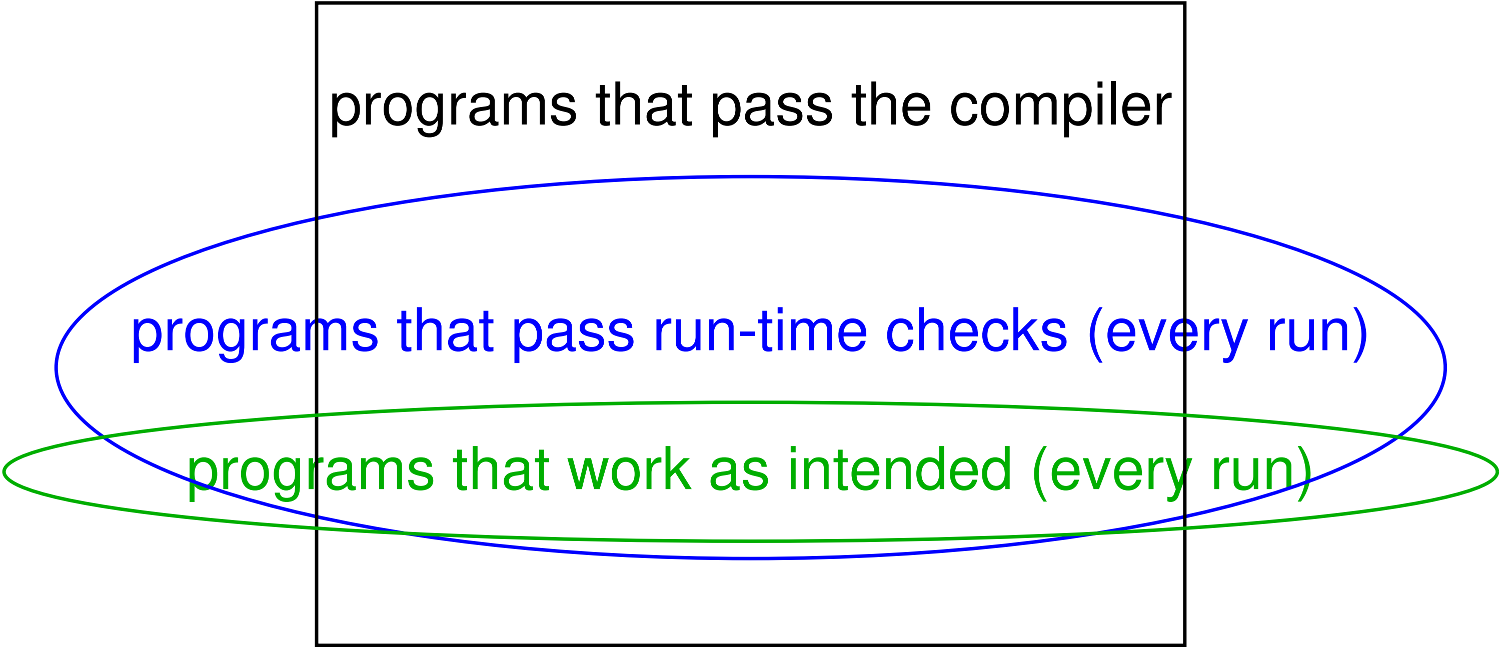
```
add  $8,%rbx
imul %r9,%r8
mov  (%rbx),%rax
jmp  *%rax
```

- `see word`
- `simple-see word`
- `see-code word`

Correctness and checking (one language)



Correctness and checking (across languages)



Reliability and security: common problems

- Buffer overflow

```
create buf 3 allot s" too long" buf swap move
```

  Countermeasure: bounds check

- Dangling reference, use-after-free, double-free

```
1 cells allocate throw dup value p value q
p free throw 5 q ! \ use after free
q free throw       \ double free
```

Rust: Building

```
cargo init myproject
cd myproject
emacsclient src/main.rs # edit
cargo run
# executable in target/debug/myproject
```

Rust: Generalities

- Syntax: `{}`, but many differences from C/C++/Java

  `let x = if a<0 { 5 } else { 6 };`

- variables are immutable by default

  `let x = ...;` vs. `let mut x = ...;`

  Redefinition shadows without warning

- type inference, except at function boundary

- integer types: `i8`...`i128`, `isize`, `u8` ... `u128`, `usize`

  no implicit conversion

  default: `i32`, if type inference does not produce a type

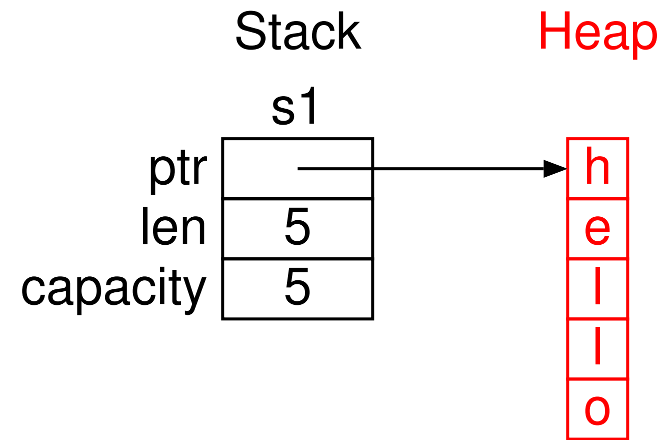
  ```
  let x=5_000_000_000; // error
  let x=5_000_000_000; let y:i64 = x; // ok
  ```

Values on the Stack

- Values of fixed-size types can be on the stack
- Values on the stack can be *copied*.

```
fn main() {
   let s = [3; 5];
   let t = s; // copy
   let x = s[2];
   println!("{x}");
}
```

Stack and Heap

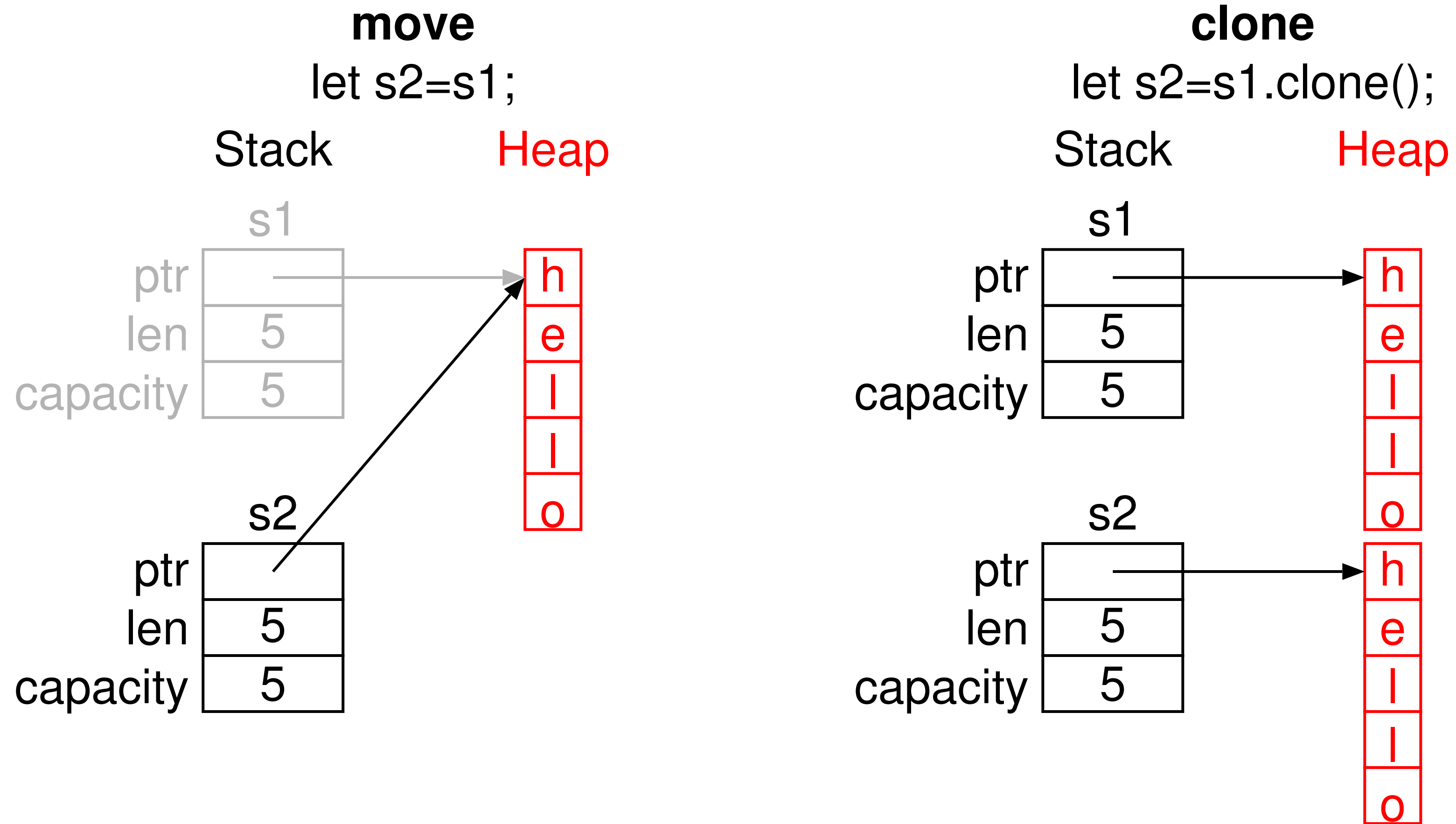


Heap and Ownership

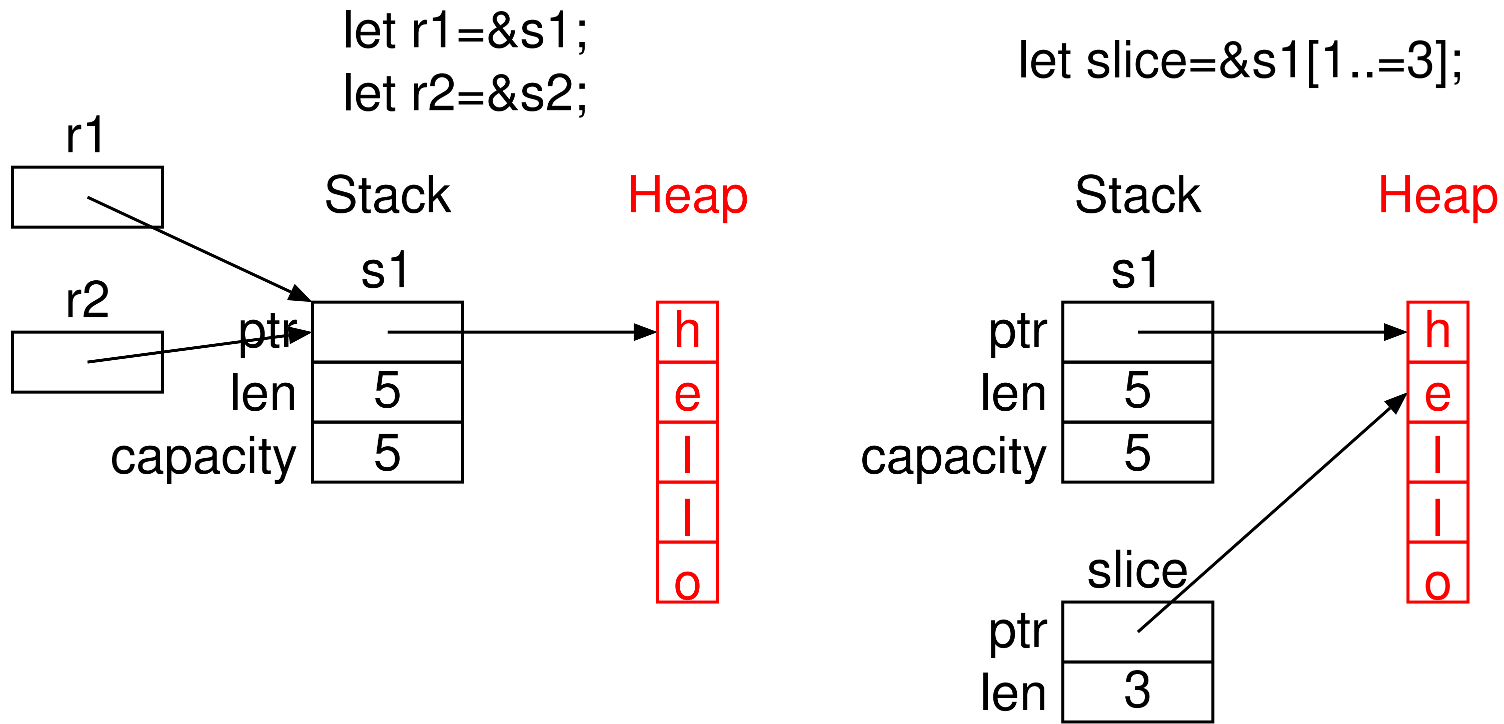
- Each value on the heap has an owner.
- There can only be one owner at a time.
  Values on the heap cannot be copied, assignment *moves*.
  Values on the heap can be explicitly cloned
- When the owner goes out of scope, the value will be dropped.
- Compiler complains only on usage.

```
fn main() {
   let s1 = "hello".to_string();
   let s2 = s1; // move
   println!("{s1}"); // Error
}
```

Moving and Cloning



References and Slices



- Reference lives only as long as the referenced value
- References important for parameter passing
- *Borrowing*

Mutable References

- There can only be one!
- No unmutable references allowed to live simultaneously

Last use ends liveness

```
fn main() {
    let mut s = String::from("hello");

    let r1 = &s; // no problem
    let r2 = &s; // no problem
    println!("{r1} and {r2}");
    // Variables r1 and r2 will not be used after this point.

    let r3 = &mut s; // no problem
    println!("{r3}");
}
```

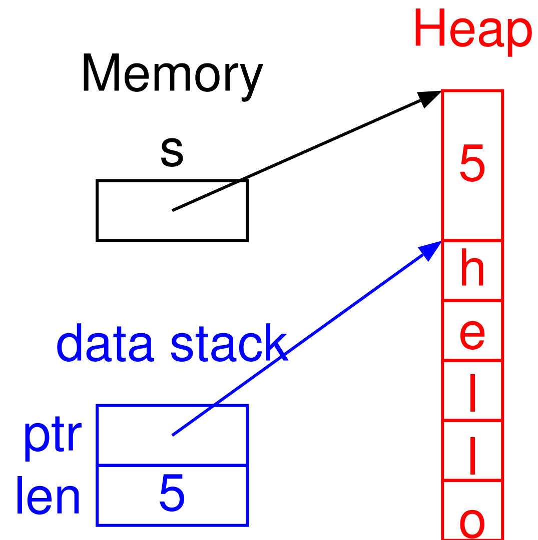
Last use ends liveness

```
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();
    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }
    &s[..]
}

fn main() {
    let my_string = String::from("hello world");
    let word = first_word(&my_string[..]);
    println!("the first word is: {word}");
}
```

Related: Gforth $trings

```
variable s
s $init
s" hello" s $!
s $@ type
```



Borrow checker

```
fn main() {
    let r;

    {
        let x = 5;
        r = &x;  // borrowed value does not live long enough
    }
    println!("r: {r}");
}
```

- reference lives at most as long as its value
- variables must be initialized before use
  but not necessarily at definition

Borrow checker

```
fn main() {
    let string1 = String::from("abcd");
    let result;    // not mut; initialized in if
    if longest(&string1,"foo")=="abcd" {
        let string2 = "xyzab".to_string();

        result = longest(string1.as_str(), &string2);
        // borrowed value does not live long enough
    } else {
        result = "baz";
    }
    println!("The longest string is {result}");
}
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}
```

- Lifetime annotation `'a` for functions
- lifetime relations
- sometimes automatic:
  per reference parameter
  one input ⇒ outputs
  self ⇒ outputs
- Initialization in `if`
- `'static` lifetime

Structs

```
struct Rectangle {
    width: u32,
    height: u32,
}
fn area(rectangle: &Rectangle) -> u32 {
    rectangle.width * rectangle.height
}
fn main() {
    let mut rect1 = Rectangle {
        height: 50,
        width: 30,
    };
    println!(
        "The area of the rectangle is {}.",
        area(&rect1));
}
```

- Fields not mutable
- Struct vars may be mutable
- automatic dereferencing

Struct methods

```
struct Rectangle {
    width: u32,
    height: u32,
}
impl Rectangle {                // <------
    fn area(&self) -> u32 {     // <------
        self.width * self.height // <------
    }
}
fn main() {
    let rect1 = Rectangle { ... };
    println!(
        "The area of the rectangle is {}.",
        rect1.area());  // <---------------
}
```

- `&self` is `self: &Self`
- different syntax
- namespace organisation
- no dynamic dispatch for now

Structs

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.').next().unwrap();
    let i = ImportantExcerpt {
        part: first_senence,
    };
    println!("{}",i.part);
}
```

- References in fields need lifetime

Enumerations and pattern matching

```
enum Message {
    Quit,                         // no data
    Move { x: i32, y: i32 },      // struct
    Write(String),                // tuple
    ChangeColor(i32, i32, i32), // tuple
}
fn main() {
    let m = Message::Write(String::from("hello"));
    let m = Message::Move{x:1, y:2};
    match m {
        Message::Quit => println!("quit"),
        Message::Move { y:a, x } => println!("{a},{x}"),
        Message::Write(s) => println!("{s}"),
        Message::ChangeColor(_r, _g, _b) => return,
    }
}
```

- not just variables in patterns
- `match` requires all cases
- syntax for matching one case

Option and Result

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

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

fn foo(...) -> Result<String, io::Error> {
  let file = File::open("foo.txt")?;
  let file_contents = ...;
  return Ok(file_contents);
}
```

- No `NULL`
- No exceptions
- Conveniences for these types

Panic! instead of Err Result

```
fn foo(...) -> Result<String, io::Error> {
  let file = File::open("foo.txt")?;
  let file_contents = ...;
  return Ok(file_contents);
}

fn foo(...) -> String {
  let file = File::open("foo.txt")
      .expect("foo.txt not found");
  let file_contents = ...;
  return file_contents;
}
```

`Vec<T>`: growable arrays

```
fn main() {
    let mut v = vec![1, 2, 3, 4, 5];
    let first = v[0];
    v.push(6);
    println!("The first element is: {first}");
}

fn main() {
    let mut v = vec![100, 32, 57];
    for i in &mut v {
        *i += 50;
    }
}
```

Genericity

```
fn largest<T: std::cmp::PartialOrd>(list: &[T]) -> &T {
    let mut largest = &list[0];
    for item in list {
        if item > largest {
            largest = item;
        }
    }
    largest
}
```

- Monomorphization

Traits

```
pub trait Summary {
    fn summarize(&self) -> String;
}

pub struct NewsArticle {
    ...
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        ...
    }
}
```

- similar to interfaces
- no dynamic dispatch for now

Cost of Result vs. panic

```
fn fibr(n:usize) -> Result<usize,()> {
    match n {
        0 => Ok(0),
        1 => Ok(1),
        _ => {
            let s=fibr(n-2)?;
            let t=fibr(n-1)?;
            match s.overflowing_add(t) {
                (r,false) => Ok(r),
                (_,true) => Err(())
            }
        }
    }
}
```

```
fn fibv(n:usize) -> usize {
    if n<2 {
        n
    } else {
        let s=fibv(n-2);
        let t=fibv(n-1);
        s.checked_add(t).unwrap()
        // s.wrapping_add(t)
        // s+t
    }
}
```

cyc	inst.	br.	per return
6.7	18.6	4.8	Result
4.6	14.5	3.5	`s.checked_add(t).unwrap()`
3.6	10.4	2.3	`s.wrapping_add(t)`, `s+t`

Result vs. exceptions

- Result: additional per-return cost
- Exceptions: cost per (`try`...)`catch`? Forth: yes, JVM: no
  With many cleanup blocks Forth's `catch` may cost more
- Exceptions: cost per not-taken `throw`: one branch
- Exceptions: cost per taken `throw`
  Does it matter? It's an exception
  Forth: a few cycles; JVM: more expensive (search for `catch`)
- Cost of exceptions in Forth
  Setting up catch frame
  checking on not-taken `throw`
- Cost of exceptions in Java
  no catch frame
  checking on not-taken `throw`
  search for try block on taken `throw`

Cost of reference vs. owned

```
fn longestref<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}
fn longeststring(x: &str, y: &str) -> String {
    let r = if x.len() > y.len() { x } else { y };
    r.to_string()
}
```

- average string length 128 bytes
  not cheaper with 32 bytes
  owned 25% more expensive with 512 bytes
-
cyc	inst	per call (includes benchmarking overhead)
6	18	reference
43	188	owned

- possibly more complex data structures in your program

(Smart) pointers (single-threaded)

- implement `Deref` (`deref()`) and `Drop` (`drop()`)
- you can implement your own smart pointers
- `Box<T>`: heap-allocated, owned, usual borrowing restrictions
- `Rc<T>` (reference-counted): multiple owners of data
- `RefCell<T>`: mutability checked at run-time
- `Rc<RefCell<T>>`: multiple owners of mutable data
- `Weak<T>`: can avoid leaking pointer cycles

Box<T>: Dereference

```
fn main() {
    let x = 5;
    let y = Box::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

Box<T>: Linked list

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

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

fn main() {
    let list = Cons(1, Box::new(Cons(2, Box::new(Cons(3, Box::new(Nil))))));
}
```

Define your own smart pointer: `Deref`

```
use std::ops::Deref;
impl<T> Deref for MyBox<T> {
    type Target = T;
    fn deref(&self) -> &T {
        &self.0
    }
}
struct MyBox<T>(T);
impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}
```

```
fn hello(name: &str) {
    println!("Hello, {name}!");
}
fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&m);
}
```

`Rc<T>`: multiple owners

```
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<RefCell<T>`: multiple owners of mutable data

```
#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}
use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;
fn main() {
    let value = Rc::new(RefCell::new(5));
    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));
    let b = Cons(Rc::new(RefCell::new(3)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(4)), Rc::clone(&a));
    *value.borrow_mut() += 10;
    println!("a: {a:?}, b: {b:?}, c: {c:?}");
}
```

Cycles in pointered structures

```
use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;
enum List {
    Cons(i32, RefCell<Rc<List>>),
    Nil,
}
impl List {
    fn tail(&self) ->
            Option<&RefCell<Rc<List>>> {
        match self {
            Cons(_, item) => Some(item),
            Nil => None,
        }
    }
}
```

```
fn main() {
    let a = Rc::new(Cons(5,
              RefCell::new(Rc::new(Nil))));
    let b = Rc::new(Cons(10,
              RefCell::new(Rc::clone(&a))));
    if let Some(link) = a.tail() {
      *link.borrow_mut() = Rc::clone(&b);
    }
}
```

Weak<T>: to avoid memory leak

```
use std::cell::RefCell;
use std::rc::{Rc, Weak};
#[derive(Debug)]
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),  }
    });
    println!("leaf parent = {:?}",
         leaf.parent.borrow().upgrade());
    let branch = Rc::new(Node {
      value: 5,
      parent: RefCell::new(Weak::new()),
      children: RefCell::new(
                vec![Rc::clone(&leaf)]),
    });
    *leaf.parent.borrow_mut() =
                Rc::downgrade(&branch);
    println!("leaf parent = {:?}",
         leaf.parent.borrow().upgrade());
```

trait objects

```
pub trait Draw { fn draw(&self); }
pub struct Button0 { }
impl Draw for Button0 {
    fn draw(&self) {
        println!("Button0"); } }
// Button1 and Button2 similar
fn main() {
    let mut v:Vec<Box<dyn Draw>>=vec![];
    v.push(Box::new(Button0{}));
    v.push(Box::new(Button2{}));
    v.push(Box::new(Button1{}));
    for i in v {
        i.draw();
    }
}
```

- `draw()` of run-time struct called dynamic dispatch (OOP)
- `:Vec<Box<dyn Draw>>` necessary
- `Box` necessary
- `dyn` necessary

Multiple traits and dynamic dispatch

```
pub trait Draw { fn draw(&self);}
pub trait Bla {  fn bla(&self); }
trait DrawBla: Draw + Bla {}
// Draw and Bla for Button0
impl DrawBla for Button0 {}
// likewise for Button1 and Button2
fn main() {
    let mut v:Vec<Box<dyn DrawBla>>=vec![];
    v.push(Box::new(Button0{}));
    // ...
    for i in v {
        i.draw();
        i.bla();
    }
}
```

- `Draw` and `Bla` are supertraits of `DrawBla`

Data representations and sizes

```
fn main() {
    use std::mem::{size_of,size_of_val,
                   offset_of};
    enum Enum0 { E0a(u8) }
    println!("Enum0: {}",size_of::<Enum0>());
    let s = "bla".to_string();
    println!("&*s: {}",size_of_val(&*s));
    println!("&s: {}",size_of_val(&s));
    struct Node {
        value: isize,
        parent: RefCell<Weak<Node>>,
        children: RefCell<Vec<Rc<Node>>>,
    }
    println!("{}",offset_of!(Node,value));
}
```

- pointer only when necessary
- default: representation not defined
  order of fields can vary
  NULL optimization
- specific representations
  C
  others

Data sizes

```
 1 enum Enum0 { E0a(u8) }
 2 enum Enum1 { E1a(u8), E1b }
16 enum Enum2 { E2a(u64), E2b }
 8 enum Enum3<'a> { E3a(&'a u8), E3b }
16 enum Enum4<'a> { E4a(&'a u8), E4b, E4c }
16 enum List3 {Cons(isize, Box<List3>), Nil3 }
16 enum List0 { Cons0(isize, Rc<List0>), Nil0 }
16 enum List1 { Cons1(Rc<RefCell<i32>>, Rc<List1>), Nil1 }
24 enum List2 { Cons2(i32, RefCell<Rc<List2>>), Nil2 }
56 struct Node { value: isize, parent: RefCell<Weak<Node>>,
        children: RefCell<Vec<Rc<Node>>> }
   offsets: value: 48, parent: 32, children: 0
 8 Rc<Node>
24 Vec<Rc<Node>>
32 RefCell<Vec<Rc<Node>>>
16 Box<dyn Draw>
 3 &*s // let s = "bla".to_string();
```

Unsafe Rust

- Promise from programmer that invariants are kept
- Safe abstraction over unsafe code
- “Unsafe superpowers”
  - Dereference a raw pointer
  - Call an unsafe function or method
  - Access or modify a mutable static variable
  - Implement an unsafe trait
  - Access fields of unions
- Run-time checking with Miri

Example: `unsafe`

```
use std::slice;
fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = values.len();
    let ptr = values.as_mut_ptr();
    assert!(mid <= len);
    unsafe {
        (   slice::from_raw_parts_mut(ptr, mid),
            slice::from_raw_parts_mut(ptr.add(mid), len - mid) )
    }
}
fn main() {
    let mut vector = vec![1, 2, 3, 4, 5, 6];
    let (left, right) = split_at_mut(&mut vector, 3);
}
```

Example: `unsafe`

```
unsafe extern "C" {
    safe fn abs(input: i32) -> i32;
}

fn main() {
    println!("Absolute value of -3 according to C: {}", abs(-3));
}
```

Concurrent programming

- parallel: two CPUs work on different data
- concurrent: two threads work on the same data
  dining philosophers
- race conditions
- deadlocks

... in Rust

- high level: channels `std::sync::mpsc`
- medium level: `Mutex<T>`,`Arc<T>`
- low level: `std::sync::atomic` memory accesses
- `async/await`: concurrency within a thread

Multiple threads with channels

```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;
fn main() {
  let (tx, rx) = mpsc::channel();
  thread::spawn(move || {
    let vals = vec!["hi","from","the","thread"];
    for val in vals {
        tx.send(val.to_string()).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
  });
  for received in rx {
      println!("Got: {received}");
  }
}
```

- `spawn` starts thread
  `move` data to thread
- `join` waits for finish
- `recv` (not here)
- receiver as iterator
- multiple producers
  with `tx.clone()`

threads vs. async

```
use std::sync::mpsc;
use std::thread;
fn main() {
    let (tx, rx) = mpsc::channel();
    let mut handles = vec![];
    for i in 0..100000 {
        let tx1 = tx.clone();
        let handle = thread::spawn(move || {
            tx1.send(i.to_string()).unwrap(); });
        handles.push(handle); }
    for handle in handles {
        handle.join().unwrap(); }
    drop(tx);
    for received in rx {
        println!("Got: {received}"); }
}
```

- 2.5s elapsed
- 4.66s CPU time
- 1.84 CPUs used
- thread spawn overhead
- not efficient for short tasks
- limit active threads

threads vs. async

```
fn main() {
    trpl::run(async {
        let (tx, mut rx) = trpl::channel();
        let mut futures = vec![];
        for i in 0..100000 {
            let tx1 = tx.clone();
            let tx1_fut = async move {
                tx1.send(i.to_string()).unwrap(); };
            futures.push(tx1_fut); }
        trpl::join_all(futures).await;
        drop(tx);
        while let Some(value) = rx.recv().await {
            println!("Got: {value}"); }
    });
}
```

- 0.04s elapsed
- 0.04s CPU time
- 1.00 CPUs used
- use for short tasks

Linked list queue

```
pub struct List<T> { head: Link<T> }
type Link<T> = Option<Box<Node<T>>>;
struct Node<T> {
    elem: T,
    next: Link<T>,
}
impl<T> List<T> {
    pub fn new() -> Self {
        List { head: None }
    }
    pub fn pop(&mut self) -> Option<T> {
        self.head.take().map(|node| {
            self.head = node.next;       }
            node.elem
        })
    }
```

```
fn getp(&mut self) -> &mut Link<T> {
    let mut l = &mut self.head;
    while let Some(s)=l {
        l = &mut s.next;
    };
    l
}

pub fn add(&mut self, elem: T) {
    *self.getp() = Some(Box::new(
          Node {elem, next: None}));
}
```

```
use std::ptr;
pub struct List<T> {
    head: Link<T>,
    tail: *mut Node<T> }
type Link<T> = *mut Node<T>;
struct Node<T> {
    elem: T,
    next: Link<T> }
impl<T> List<T> {
  pub fn new() -> Self {
    List { head: ptr::null_mut(),
         tail: ptr::null_mut() } }
  pub fn add(&mut self, elem: T) {
    unsafe {
      let new_tail =
       Box::into_raw(Box::new(Node {
        elem: elem,
        next: ptr::null_mut() }));
      if !self.tail.is_null() {
        (*self.tail).next = new_tail;
      } else {
        self.head = new_tail; }
      self.tail = new_tail; } }
  pub fn pop(&mut self) -> Option<T> {
    unsafe {
      if self.head.is_null() {
        None
      } else {
        let head = Box::from_raw(self.head);
        self.head = head.next;
        if self.head.is_null() {
          self.tail = ptr::null_mut(); }
        Some(head.elem) } } } }
```