Systems Programming

1 What is Systems Programming?

Systems programs are infrastructure components like operating systems, device drivers, and network protocols. They're constrained by memory management, I/O, and shared state.

Memory Management High performance requires control over data representation, ruling out languages like Java and Python. C, C++, and Rust offer this control.

I/O Efficient I/O is crucial. Format conversion hurts performance. I/O performance is fundamental to systems programs.

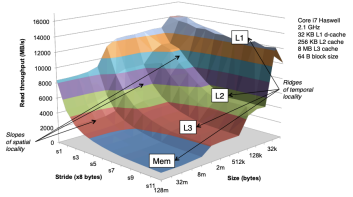
Shared State Management Systems programs manage shared state, accessed concurrently by multiple processes. Concurrency and parallelism are key concerns.

1.1 Memory Management and Data Representation

Key elements include:

- **Predictability**: Bounded timing and memory usage.
- **Data Locality**: Cache-friendly data alignment.
- **Data Representation**: Matching external formats exactly.

Systems programming languages offer memory and data control. Predictable memory usage is crucial, especially for real-time and embedded systems.

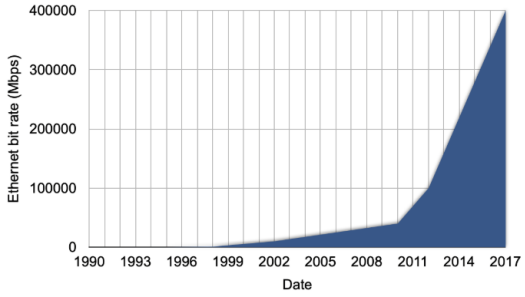


Smaller stride indicates better locality; size is total data accessed.

Data representation and locality impact performance. The diagram shows how object size and spacing (stride) affect read throughput. Effective systems programs arrange data for high performance, requiring precise control over data layout.

1.2 I/O Operations

I/O performance is a key constraint. Ethernet speeds have grown exponentially, while packet size (MTU) remains constant. This increases packets per second, straining processors.



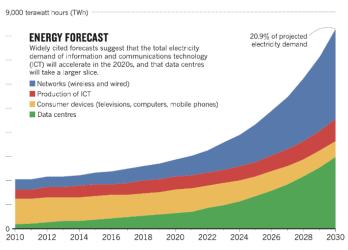
Network packet processing demands have increased, while CPU performance has plateaued. Encryption and streaming video exacerbate this. I/O is a bottleneck.

1.3 Management of Shared Space

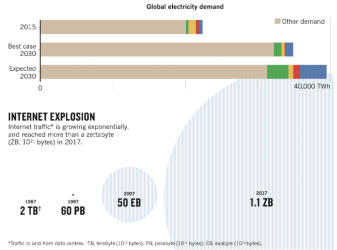
Systems programs coordinate shared resources. They manage kernel state, file systems, etc., accessed concurrently. Concurrency is unavoidable.

1.4 Performance

Systems program performance affects overall system performance. Inefficiencies reduce performance and increase power consumption.



The chart above is an 'expected case' projection from Anders Andrae, a specialist in sustainable ICT. In his 'best case' scenario, ICT grows to only 8% of total electricity demand by 2030, rather than to 21%.



Source: N. Jones, "The Information Factories", Nature, v.561, p.163–166, Sep. 2018.DOI:10.1038/d41586-018-06610-y

1.5 Systems Programming

These trends push systems programming languages toward low-level control. C and C++ remain popular due to this control.

2 The State of the Art

Most systems run Unix variants, written in C. This includes Android, iOS, and macOS. Windows is influenced by VMS, also written in C.

2.1 Unix and C: Strengths

Unix was portable and open-source. Its small, consistent API is robust and high-performance. C's pointers simplify portable device drivers.

```
struct {
    short errors     : 4;
    short busy       : 1;
    short unit_sel   : 3
    short done       : 1;
    short irq_enable : 1
    short reserved   : 3
    short dev_func   : 2;
    short dev_enable : 1;
} ctrl_reg;

int enable_irq(void)
{
    ctrl_reg *r = 0x80000024;
    ctrl_reg  tmp;

    tmp = *r;
    if (tmp.busy == 0) {
        tmp.irq_enable = 1;
        *r = tmp;
        return 1;
    }
    return 0;
}
```

C code cleanly writes portable device drivers, exemplified by the `ctrl_reg` struct and `enable_irq()` function.

2.2 Unix and C: Weaknesses

Unix APIs can be bottlenecks. Its security model is outdated. C's pointers are powerful but error-prone. The weak type system makes reasoning difficult. Concurrency support is limited.

2.3 Unix and C

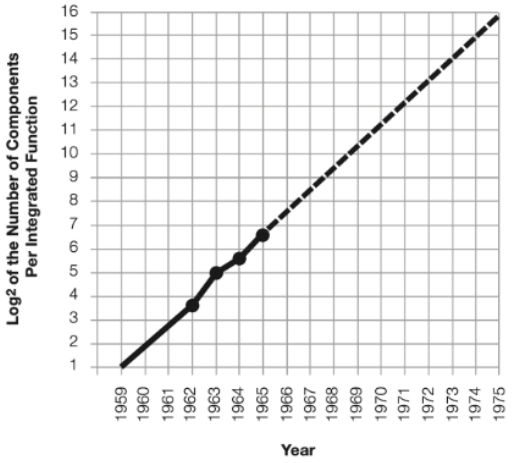
Unix is successful but has limitations. C is increasingly a liability due to its error-prone nature.

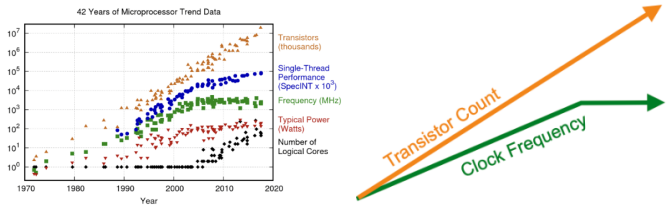
3 Challenges and Limitations

Four trends affect systems programs: end of Moore's Law, increased concurrency, security needs, and connectivity.

3.1 End of Moore's Law

Moore's Law is ending. Dennard scaling has broken down. Clock rates have stalled. Multi-core processors are now the norm.



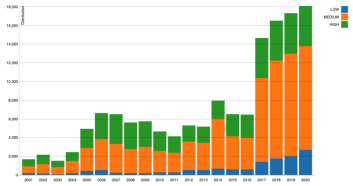


3.2 Increasing Concurrency

Multi-core processors increase the need for concurrent programming, making concurrency bugs more visible.

3.3 Increasing Need for Security

Connectivity increases security vulnerabilities. Many vulnerabilities are due to weak type systems and lack of memory safety.



3.4 Increasing Mobility and Connectivity

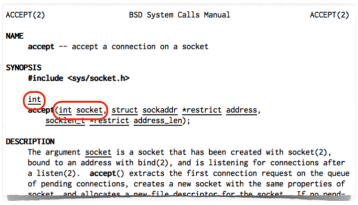
Mobile, connected devices are constrained by power and network limitations.

4 Next Steps in Systems Programming

Functional programming and modern type systems can improve systems programming.

4.1 What is a Modern Type System?

Modern type systems provide guarantees about program behaviour, like no out-of-bounds accesses or use-after-free bugs. They enable type-driven design.



Stronger types catch bugs at compile time.

4.2 What is Functional Programming?

Functional programming avoids side effects and shared state. It improves testability and makes concurrent code easier to write.

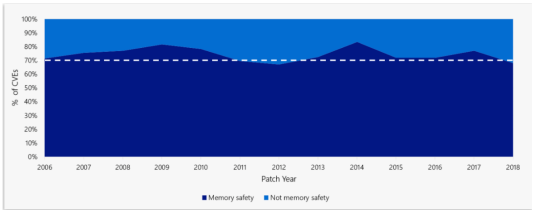
4.3 How to Improve Memory Management & Safety?

Manual memory management is error-prone. Managed languages offer safety but at a performance cost. Rust detects memory errors at compile time.

```c
#include <stdio.h>

int main()
{
  int x[5];

  x[3] = 42;

  int a = *(x + 3);
  printf("%d\n", a);

  int b = 3[x];
  printf("%d\n", b);
}
```

4.4 How to Improve Security?

Memory safety issues cause many security vulnerabilities.



Encoding assumptions in types improves security.

4.5 How to improve Support for Concurrency?

Locks are difficult to manage. Functional programming and ownership tracking are alternatives.

4.6 How to improve Correctness?

Modern languages and tools can eliminate classes of bugs. Type systems help check designs. Define specific types and use them consistently.

Types and Systems Programming

1 Types and Systems Programming

1.1 Strongly Typed Languages

1.1.1 What is a Type?

A type describes what an item of data represents, is it an integer? floating point value? file? username? sequence number? Conceptually it says what is the data represented by a variable and how is that data represented. Types are every familiar in programming, for example, the below C code shows a number of variable declarations, specifying the types of the variables. New types can also be defined, for example the struct sockaddr_in, a compound type.

```c
int    x;
double y;
char  *hello = "Hello, world";

struct sockaddr_in {
  uint8_t         sin_len;
  sa_family_t     sin_family;
  in_port_t       sin_port;
  struct in_addr  sin_addr;
  char            sin_pad[16];
};
```

1.2 What is a Type System

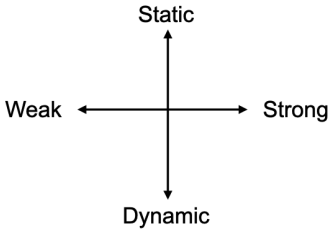
A type system is a set of rules constraining how types can be used, for example,

- What operations can be performed on a type?
- What operations can be performed with a type?
- How does a type compose with other types of data?

A type system proves the absence of certain program behaviours. It doesn't guarantee the program is correct, but it does guarantee that some incorrect behaviours do not occur. A good type system eliminates common classes of bug, without adding too much complexity. A bad type system adds complexity to the languages and doesn't prevent many bugs.

Type-relates checks can happen at compile time, run time, or both, e.g. array bounds checks are a property of an array type checked at run time.

1.3 Types Of Type System



1.3.1 Static and Dynamic Types

In a language with static types, the type of a variable is fixed - some require types to be explicitly declared; others can infer types from context. Just because the language can infer the type, does not mean the type is dynamic

```
> cat src/main.rs
fn main() {
    let x = 6;
    x += 4.2;
    println!("{}", x);
}
> cargo build
   Compiling hello v0.1.0 (/Users/csp/tmp/hello)
error[E0277]: cannot add-assign `{float}` to `{integer}`
 --> src/main.rs:3:7
  |
3 |     x += 4.2;
  |       ^^ no implementation for `{integer} += {float}`
  |
  = help: the trait `std::ops::AddAssign<{float}>` is not implemented for `{integer}`

error: aborting due to previous error
```

The Rust compiler infers that x is an integer and won't let us add a floating point value to it, since that would require changing its type.

In a language with dynamic types, the type of a variable can change, for example, in this following python snippet

```
> python3
Python 3.6.2 (v3.6.2:5fd33b5926, Jul 16 2017, 20:11:06)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> x = 6
>>> type(x)
<class 'int'>
>>> x += 4.2
>>> type(x)
<class 'float'>
>>>
```

Dynamically typed languages tend to be lower performance, but offer more flexibility. They have to store the type as well as its value, which takes additional memory. They can make fewer optimisations based on the type of a variable, since that type can change. Systems generally have static types, and be compiled ahead of time, since they tend to be performance sensitive.

1.3.2 Strong and Weak Types

In a language with strong types, every operation must conform to the type system. Operations that cannot be proved to conform to the typing rules are not permitted. Weakly typed languages provide ways of circumventing the type checker - this might be automatic safe conversion between types(left) or an open ended cast(right)

```
float  x = 6.0;
double y = 5.0;
double z = x + y;
```

```
char *buffer[BUFLEN];
int   fd = socket(…);
…
if (recv(fd, buffer, BUFLEN, 0) > 0) {
  struct rtp_packet *p = (struct rtp_packet *) buf;
  …
}
```

A common C programming idiom - casting between types using pointers to evade the type system.

Think of strong and weak types in the context of safe and unsafe languages: A safe language, whether static or dynamic, knows the types of all variables and only allows legal operation on those values; An unsafe language allows the types to be circumvented to perform operations the programmer believes to be correct, but the type system can't prove to be so.

1.4 Why is Strong Typing Desirable?

Results of a program using only strong types are well defined, i.e. a safe language. Type system ensures results are consistent with the rules of the languages, a strongly typed program will only every perform operations on a type that are legal, it cannot perform undefined behaviour.

Use of Strong types helps model the problem, check for consistency, and eliminate common classes of bug. Strong typing cannot prove that a program is correct, but can prove the absence of certain classes of bug.

1.5 Undefined Behaviour

Segmentation faults should never happen - Compiler and runtime should strongly enforce type rules, if a program violates them, it should be terminated cleanly. Security vulnerabilities come from undefined behaviour after type violations.

C has 193 kinds of undefined behaviour, defined as "behaviour, upon use of a nonportable or erroneous program construct or of erroneous data, for which this international standard imposes no requirements".

Undefined behaviour leads to entirely unpredictable results.

1.6 Types for Systems Programming

C is weakly typed and widely used for systems programming - This is mostly for historical reasons, the original designer of C were not type theorists, and the original machines on which C was developed didn't have the resources to perform complex type checks. Type theory was not particularly advanced in the early 1970s.

Strongly typed systems programming is however feasible - many examples of operating systems are written in strongly typed languages, for example, old versions of macOS in Pascal, the US DoD using the Ada programming languages, etc.

The popularity of Unix and C has lef to a belief that operating systems require unsafe code, this is true only at the very lowest levels, most systems, including device drivers, can be written in strongly typed, safe languages. Rust is a modern attempt to provide a type safe language suited to systems programming .

2 Introduction To Rust

Rust is a modern systems language with a strong static type system. Initially developed by Graydon Hoare as a side project in 2006, sponsored by Mozilla since 2009, V1.0 released in 2015, then in 2018, Rust v1.31, or "Rust 2018 Edition" released. New releases are made every 6 weeks with a strong backwards compatibility policy.

2.1 Basic Features and Types

- Function definition: `main()` takes no arguments, returns nothing
- Macro expansion: `println!()` with string literal as argument
- Import `env` module from standard library (`use std ::env;`)
- Use `for` loop to iterate over command line arguments (`for arg in env::args() {...}`)
- Function arguments and return type - mutable or immutable `fn gcd(mut n: u64, mut m: u64) -> u64 { ... }`
- Control flow: `while` and `if` statements
- Local variable definition (`let binding`), type is inferred
- function implicitly returns value of final expression, `return` statement allows early return

Rust's primitive types map closely to those in C, Rust has native `bool`, C uses `int` to represent boolean.

In C, a `char` is one byte, implementation defined if signed, and character set unspecified. In Rust, a `char` is a 32 bit Unicode scalar value.

Arrays work as expected - elements are all the same type, and the number of elements is fixed. The type of the array is inferred from the type of the elements. Array types are written `[T]`, where T is the type of the elements.

```
fn main() {
    let mut v : Vec<u32> = Vec::new();
    v.push(1);
    v.push(2);
    v.push(3);
    v.push(4);
    v.push(5);
}
```

Vectors, `Vec<T>` are the variable sized equivalent of arrays. The type parameter `<T>` specifies element. The : `Vec<u32>` modifier specifies the type for variable v and can usually be omitted, allowing the compiler to infer type.

The `vec![...]` macro is a shortcut to create vector literals

Vectors are implemented as the equivalent of a C program that uses `malloc()` to allocate space for an array, then `realloc()` to grow the space when it gets close to full.

Tuples are collections of unnamed values - each element can be a different type, `let` bindings can de-structure tuples, and tuple elements can be accessed by index. An empty tuple is the unit type, like void in C.

Structs are collections of named values, again, each element can have a different type. fields of a struct can be accessed using dot notation. A struct can be created with specified field values like: `let rect = Rectangle {width: 30, height: 50}`

Elements of a struct can be unnamed, known as tuple structs, which are useful as type aliases. Unit-like structs have no elements and take up no space, these are useful as markers or type parameters.

Methods are defined in an `impl` block, with explicit self references, like Python. Method calls use dot notation. Methods can be implemented on structs - somewhat similar to objects, but Rust does not support inheritance or sub-types.

3 Rust: Abstraction, Traits, Enumerated types, and pattern matching

3.1 Traits

A trait is defined with a single method that must be implemented, for example the following code has a struct type, rectangle, and the `Area` trait is implemented on the type.

```
trait Area {
    fn area(self) -> u32;
}

struct Rectangle {
    width: u32,
    height: u32,
}

impl Area for Rectangle {
    fn area(self) -> u32 {
        self.width * self.height
    }
}
```

Traits describe functionality that types can implement. Methods that must be provided, and associated types that must be specified, by types that implement the trait - but no instance variables or data. These are similar to type classes in Haskell or interfaces in Java.

A trait can also be implemented by multiple types, for example, the `Area` trait can also be implemented for a `Circle` class.

```
struct Circle {
    radius: u32
}


impl Area for Circle {
    fn area(self) -> u32 {
        PI * self.radius * self.radius
    }
}
```

Traits are an important tool for abstraction, playing a similar role to subtypes in many languages.

3.1.1 Generic Functions

Rust uses traits instead of classes and inheritance to define generic functions or methods that work with any type that implements a particular trait, first, you write a trait, then write functions that work on types that implement that trait.

3.1.2 Deriving Common Traits

The `derive` attribute makes the compiler automatically generate implementations of some common traits.

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}
```

The `#[derive(T)]` annotation makes compiler generate an `impl` block with standard implementation of methods for derived trait `T`.

Compiler implements this for traits in the standard library that are always implemented in the same way. This can also be implemented for user-defined traits, but is only useful if every implementation of the trait will follow the same structure.

3.1.3 Associated Types

Traits can also specify associated types - types that must be specified when a trait is implemented, e.g. `for` loops operate on iterators

```
fn main() {
    let a = [42, 43, 44, 45, 46];

    for x in a.iter() {
        println!("x={}", x);
    }
}
```

The `a.iter()` function call returns an iterator over the array, an iterator is something that implements the `Iterator` trait. An `impl` of `Iterator` must define the type of `item`, as well as implementing the methods.

3.2 Enumerated Types

Basic enums work just like in C. Enums also generalise to store tuple-like variants and struct like variants

```
enum RoughTime {
    InThePast(TimeUnit, u32),
    JustNow,
    InTheFuture(TimeUnit, u32)
}

let when = RoughTime::InThePast(TimeUnit::Years, 4*20 + 7);
```

```
enum Shape {
    Sphere {centre: Point3d, radius: f32},
    Cuboid {corner1: Point3d, corner2: Point3d}
}

let unit_sphere = Shape::Sphere{centre: ORIGIN, radius: 1.0};
```

An `enum` is used when a variable, parameter, or result can have on of several possible types.

- An `enum` defines alternative types for a type
- An `enum` can have parameters that must be defined when the enum is instantiated
- An `enum` can have methods and can implement traits.

Use `enum` to model data that can take one of a set of related values.

Standard library has two extremely useful standard `enum` types. The `Option` type represents optional values, in C, one might write a function to lookup a key in a database, this returns a pointer to the value or `null` if the key doesn't exist. In Rust, the equivalent function would return `Option<Value>`.

The `result` type similarly encodes success or failure

3.3 Pattern Matching

Rust `match` expression generalises the C `switch` statement, e.g. we can match against constant expressions and wildcards:

```
match meadow.count_rabbits() {
    0 => {} // nothing to say
    1 => println!("A rabbit is nosing around in the clover."),
    n => println!("There are {} rabbits hopping about in the meadow", n)
}
```

The value of `meadow.count_rabbits()` is matched again the alternatives. If matches the constants 0 or 1, the corresponding branch executes. If none match, the value is stored in the variable `n` and that branch executes. Matching against `_` gives a wildcard without assigning to a variable.

Patterns can be any type, not just integers.

```
let calendar = match settings.get_string("calendar") {
    "gregorian" => Calendar::Gregorian,
    "chinese"   => Calendar::Chinese,
    "ethiopian" => Calendar::Ethiopian,
    _           => return parse_error("calendar", other)
};
```

The match expression evaluates to the value of the chosen branch, allows, e.g. use in `let` bindings, as shown.

Patterns can match against enum values.

```
enum RoughTime {
    InThePast(TimeUnit, u32),
    JustNow,
    InTheFuture(TimeUnit, u32)
}

let when = RoughTime::InThePast(TimeUnit::Years, 4*20 + 7);
```

```
match rt {
    RoughTime::InThePast(units, count)   => format!("{} {} ago", count, units.plural()),
    RoughTime::JustNow                   => format!("just now"),
    RoughTime::InTheFuture(units, count) => format!("{} {} from now", count, units.plural())
}
```

Selects from different types of data, expressed as enum variants. Must match against all possible variants, or include a wildcard, else compile error.

C functions often return pointer to value or `null` if the value doesn't exist. It's easy to forget the `null` check when using the value. The program crashes with null pointer dereferences at run-time if user is not found.

Rust function returns an `Option` and pattern match on result

```
fn get_user(self, username : String) -> Option<Customer> {
  // ...
}

match db.get_user(customer_name) {
  Some(customer) => book_ticket(customer, event),
  None           => handle_error()
}
```

This is preferred because all enum variants must be handled, so won't compile if you forget to check the error case

4 Rust: Memory Allocation and Boxes

4.1 References

References are explicit, like pointers in C.

Create a variable binding `let x = 10;`. Take a reference (pointer) to that binding `let r = &x;`. Explicitly dereference to access value `let s = *r`. Functions can take parameters by reference `fn calculate(b: &Buffer) ->usize { ... }`

References can be immutable(&), `let r = &x`. An immutable reference can't be changed, trying to assign to `*r` gives a compile error.

References can be mutable with &mut, : `let r = &mut x;`, with this mutable reference, the reference value can change.

4.1.1 Constraints on References

References can never be null - they always point to a valid object, `Option<T>` indicates an optional value of type T where C would use a potentially null pointer. There can be many immutable references to an object in scope at once, but there cannot be a mutable reference to the same object in scope - an object becomes immutable while immutable references to it are in scope.

There can only be at most one mutable reference to an object in scope and there can be no immutable references to the object while the mutable reference exists - an object is inaccessible to its owner while the mutable reference exists.

These rules are enforced at compile time and prevent null pointer exceptions.

4.2 Memory Allocation and Boxes

A `Box<T>` is a smart pointer that refers to memory allocated to the heap

```
fn box_test() {
    let b = Box::new(5);
    println!("b = {}", b);
}
```

Rust guarantees about memory allocation:

- The Value returned by `Box::new()` is guaranteed to be initialised
- The allocated memory is guaranteed to match the size of the type it is to t=store
- Rust guarantees that the memory will be automatically deallocated when the box goes out of scope.

Boxes own and, if bound as `mut` may change the data they store on the heap

```
fn main() {
    let mut b = Box::new(5);
    *b = 6;
    println!("b = {}", b);
}
```

Boxes do not implement the standard `copy` trait, can pass boxes around, but only one copy of each box exist, again, to avoid data races between threads. A `Box<T>` is a pointer to the heap allocated memory, if it were possible to copy the box, we could get multiple mutable references to that memory.

4.3 Strings

Strings are Unicode text encoded in UTF-8 format. A `str` is an immutable string slice, always accessed via an `&str` reference. A `String` is a mutable string buffer type, implemented in the standard library

```
let s2 = String::new();
s2.push_str("Hello, World");
s2.push('!');
```

```
let s3 = String::from("Hello, World");
s3.push('!');
```

The `String` type implements the `Deref<Target=str>` trait, so taking a refernce to `String` actually returns an `&str`

```
let s = String::from("test");     s has type String
let r = &s;                       r has type &String
let t : &str = &s;                t has type &str
```

This conversion has zero cost, so functions that don't need to mutate the string tend to be only implemented for `&str` and not on `String` values.

4.4 Key Points

Rust is a largely traditional systems programming language with very familiar basic types, control flow, and data structures. Key innovations in a systems language:

- Enumerated types and pattern matching
- Structure types and traits as an alternative to object oriented programming
- Multiple reference types and ownership

Little in rust is novel. Rust adopts ideas from research languages, its syntax is a mixture of C and Standard ML, its basic data types are heavily influenced by C and C++, its enumerated types and pattern matching are adapted from Standard ML, and Traits adapted form Haskell type classes.

Many influences from C++, but generally in the opposite way from how C++ does it. References and ownership rules extend ideas originally developed in Cyclone - this is where Rust has new ideas.

4.5 Why is Rust Interesting?

A modern type system and runtime, with no concept of undefined behaviour, it is memory safe, without buffer overflows, dangling pointers, or null pointer dereferences. Zero cost abstractions to model problem space and check consistency of design.

A type system that can model data and resource ownership - Deterministic automatic memory management. This prevents iterator invalidation, use-after-free bugs, and most memory leaks. Rules around references and ownership prevent data races in concurrent code, enforces the design patterns common in well-written C programs.

Finally, Rust is a systems programming language that eliminates many **classes** of bug that are common in C and C++ programs.

Type-Based Modelling and Design

1 Type Driven Development

With expressive strongly typed languages - such as Rust, Swift, OCaml - the type system can be used to help ensure correctness. This approach, which we call *type-driven development*, involves first **defining** the types, then using these types as a guide to **write the functions** - writing the input and output types of the functions and writing the function using the structure of the types as a guide. Then, we **refine** and edit types and functions as necessary.

Fundamentally, this approach is thinking of the types as a plan or model for the solution, rather than as a way checking the code. You build up the design around the types then and fill in the details of how operations are performed.

1.1 Define the Types

The first stage is defining the types, we consider which types are needed to build a model of the problem domain. Consider who is interacting, what they're interacting with, and what sort of things they exchange. - This usually leads to types like `sender`, `receiver`, `employee`, `TcpSegment`, etc.

Next we consider what properties describe the people/things, and what data is associated with each, giving us types representing `name`, `temperature`, `SequenceNumber`, etc.

We might need to think about what states the system or a particular interaction can be in - we might need types for these, `connecting`, `loggedIn`, `AuthRequired`.

Types might be ill-defined and abstract to begin with, but we write them down anyway and refine later.

Once we have defined our necessary types, we need to associate properties with their types. What data types are associated with the things in our system, and what properties do those things have? We similarly need to think about which state an object might be in, and create definitions for these.

```
struct Sender {
  name    : Name,
  email   : EmailAddress,
  address : PostalAddress
}
```

```
enum State {
  NotConnected,
  Connecting,
  AuthenticationRequired,
  LoggedIn,
  ...
}
```

```
struct AuthenticatedConnection {
  socket : TcpSocket,
  ...
}
```

```
struct UnauthenticatedConnection {
  socket : TcpSocket,
  ...
}
```

The important thing at this stage is to write down the types which can then be refined and extended as required.

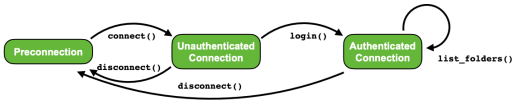
1.2 Write the Functions

Once we have our types written out, move onto functions using types as a guide, and leaving the concrete implementation of the system until later.

```
impl AuthenticatedConnection {
  fn list_folders(self) -> List<EmailFolder> {
    ...
  }

  fn list_messages(self, f : EmailFolder) -> List<EmailMessage> {
    ...
  }

  fn disconnect(self) {
    ...
  }
}
```

The important thing here is the behaviour is reflected in the types and operation of the function. The behaviour of system should be obvious from looking at the function prototypes, and the different types of objects should constrain the possible behaviours.

This means using specific types that model the problem domain rather than using generic types, i.e. passing a username parameter around rather than a string, or a temperature rather than an integer. By using more specific types, the compiler can check what we're doing, and that the behaviours we're doing makes sense, and can check our design.



If we structure the code wrong, it just won't compile. We can encode the states as the types, and the state transitions as functions manipulating those types. The types can represent the state machine, and the functions which transition between different states return different types.

Functions only get implemented on the types where they make sense to enforce the behaviour, logic, and state machine of the system.

1.3 Refine the Types and Functions

Types and functions provide a model of the system, defining what you're working with, and how the system moves between its various states as the different operations are performed.

Starting from an initial design, you iterate as you go filling just enough details to keep it compiling, gradually refining until the entire system is modelled. Then gradually adding the concrete implementations and refining these as needed.

1.4 Correct by Construction

This is an approach known as *correct by construction*, it uses the types and type system to model the problem space and check your design, and debug your design before you even begin to run the code. The idea is that nonsensical operations in the program won't cause the system to crash, instead it just doesn't compile. This change in perspective in the way we write code uses the type system and compiler as model checking tools to validate design.

2 Design Patterns

2.1 Numeric Types

When building a system - using a type driven design approach - we need to ask ourselves some key questions, for example, whether a numeric value is really best represented as a floating point value or an integer, or does it have some meaning that could be included in the type, for example, is it a temperature or speed? If so, in which units?

We should encode the meaning of a value in its type so the compiler checks for consistent usage of that type. Operations that mix different types should fail if the types don't match or don't perform safe unit conversion. Operations that are inappropriate for a type shouldn't be possible.

2.1.1 Strong Typing

Take this code for example;

```
fn main() {
    let c = 15.0;  // Celsius
    let f = 50.0;  // Fahrenheit

    let t = c + f;

    println!("{:?}", t); // 65.0
}
```

Numerically, this makes sense, we have two values, c=15, f=50, and we add these values for a total, t of 65. However, as the programmer, we know `c` to be temperature in Celsius, and `f` to be temperature in Fahrenheit, the compiler doesn't know this, so it gives the wrong answer, 15C + 50F is actually 109F or 42C, but since these constraints are not represented in the design, the compiler can't catch the mistake.

If we instead define more specific types representing temperatures in Celsius and Fahrenheit and use those instead of integers.

```
use std::ops::Add;

#[derive(Debug,PartialEq,PartialOrd)]
struct Celsius(f32);

#[derive(Debug,PartialEq,PartialOrd)]
struct Fahrenheit(f32);

impl Add for Celsius {
    type Output = Celsius;

    fn add(self, other : Celsius) -> Self::Output {
        Celsius(self.0 + other.0)
    }
}

impl Add for Fahrenheit {
    type Output = Fahrenheit;

    fn add(self, other : Fahrenheit) -> Self::Output {
        Fahrenheit(self.0 + other.0)
    }
}

fn main() {
    let c = Celsius(15.0);
    let f = Fahrenheit(50.0);
    let t = c + f;

    println!("{:?}", t);
}
```

If we now try to add Celsius and Fahrenheit values, the code won't compile, it's expecting a Celsius value, but found Fahrenheit, and there's no conversion between them. This is only a simple example, but shows the principle, we catch errors by using specific number types in place of the generic types, and in exchange of the increased complexity, we gain the ability to check the design for correctness.

2.1.2 Conversion

We can add implementations that perform unit conversion. In this example, we implement the `Add` trait, with Fahrenheit as a type parameter for the Celsius type, this describes how you add a Fahrenheit value to a Celsius value, allowing the code to successfully add the values and give the correct result

```
impl Add<Fahrenheit> for Celsius {
    type Output = Celsius;

    fn add(self, other: Fahrenheit) -> Self::Output {
        Celsius(self.0 + ((other.0 - 32.0) * 5.0 / 9.0))
    }
}

fn main() {
    let c = Celsius(15.0);
    let f = Fahrenheit(50.0);

    let t = c + f;

    println!("{:?}", t);   // Celsius(42.7) = Fahrenheit(109)
}
```

2.1.3 Operations

It's also important to consider whether all the standard operations make sense for the numeric types you define. Its reasonable, for example, to compare temperatures for equality, or compare them to see which is larger, so you'd implement the standard equality and ordinal traits that provide these operations.

However, this might make as much sense for a UserID type, it would be meaningless to add or compare UserID values, so you don't need all the standard operations for all your types.

2.1.4 No Runtime Cost

Wrapping values inside structs in this way adds no runtime overhead in Rust, however there is some programmer overhead. The lack of runtime cost is for a few reasons:

- No information added to the struct, so same size
- Passed in the same way - not automatically boxed on the heap
- Optimiser will recognise that the code collapses down to operations on primitive types, and generate the code to do so
- All the additions are a compile-time model of the ways the data can be used, they don't affect the compiled code.

2.2 Alternative Types: `enum`

Another useful thing in Rust is to use enum types and pattern matching to model alternatives, options, results, features and response codes, and flags. This lets the compiler check these types for correctness, and further helps us debug our design before we start running and debugging our code.

2.3 Optional Values: `Option<T>`

If we need to work with values that might not exist, we use the Option type to represent these in Rust, for example, if a function might return a value or not be able to find that value, we return an optional result. This is safer as the compiler forces us to pattern match to handle all cases, and is more semantically meaningful.

```
struct RtpHeader {
  v       : Version,
  pt      : PayloadType,
  seq     : SequenceNumber,
  ts      : Timestamp,
  ssrc    : SourceId,
  csrc    : Vec<SourceId>,
  extn    : Option<HeaderExtension>,
  payload : RtpPayload
}
```

We can also use optional values as part of struct definitions. For example, the Rust code representing an RTPHeader type containing an optional field. We can represent this in the struct by including an option type in the struct definition representing that format. This again makes the intent clear to both other programmers and the compiler.

In both cases, the compiler enforces that both variants are handled, the compiler ensures that we check both the Some case (where the value exists) and the None case (where the field or value doesn't). We can't accidentally write code that assumes the value is always present and crashes at runtime if not.

2.4 Results: `Result<T,E>`

The Result type represents a computation that can fail. In the same way that the option type encodes a better version of the idiom of returning a null pointer on failure, the result type is a better version of exception handling.

In the same way that option forces us to pattern match to extract the value, the Result type makes sure that we consider both the success and failure conditions, and the only way to get the result value is by pattern matching.

The result type is the equivalent of exception handling in rust, the difference being that it's more explicit.

2.5 Anti-Patterns

2.5.1 Features and Response Codes

Enum types are also useful to encode properties of the design that relate to the problem domain, they often are used to help avoid common anti patterns in system design, the first of which is known as **string typing**.

String typing is where method parameters, return types, and data values are coded as unstructured strings, rather than as some more appropriate type. By using an enum, we represent data such as this, that can take one of several possible values, in a structured way. This has some overhead, as we must define the enum that represents the different states, but it enables exhaustive checking, and we can be sure the compiler can check any case we miss.

2.5.2 Flags

The use of boolean flags as arguments to functions obscures meaning.

```
let f = File.Open("foo.txt", true, false);
```

and

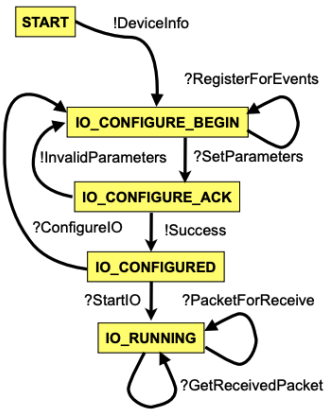
```
let f = File.Open("foo.txt", FileType::TextMode, FileMode::ReadOnly);
```

In the first of these examples, the meaning of the true and false flags is hidden, the second variant performs the same operation, but this time the arguments are encoded as enums rather than boolean. This is significantly easier to read and understand, and allows the compiler to check the arguments are passed correctly.

3 State Machines

State machines are common in systems code, frequently used to represent behaviour of network protocols, file systems, and device drivers. The system behaviour is modelled as a finite state machine comprising:

- **States** that reflect the status of the system
- **Events** that trigger transitions between states
- **State** variables that hold systems configuration



The state machine captures the essence of the system behaviour, capturing the high level structure of the design. It should be easy to reason about, to prove properties such as termination, absence of deadlocks, and whether all states are reachable, etc.

3.1 Implementing State Machines

It can be hard to cleanly implement state machines in code. The structure of code tends to not match the structure of state machines, it's often not easy to visualise the transitions.

There are new approaches to modelling state machines in strongly-typed functional languages, making the code clearer and the state machine more obvious. They encode states and events as enumerations and pattern match on state event tuples.

3.2 Enumerations for modelling state machines

There are two possible state machine implementations that leverage these insights and can be used in Rust. The first is to use enums to represent the enums and functions to represent state transitions and actions. In this approach, you define an enum to represent all possible states, and another to represent possible events. Functions are defined to take a tuple of state and event and returns the next state, and to represent the action performed on each transition.

3.2.1 Using `enum` to Model State Machines: Example

Start by defining the enums that represent the states and events.

```
enum ApcState {
    Initialize,
    WaitForConnect,
    Accept(TcpStream),
    StartTransfer(TcpStream),
    Waiting(TcpStream),
    ReceiveMsg(TcpStream, Vec<u8>),
    SendNop(TcpStream),
    Closed,
    Finish,
    Failure(String),
}
```

```
enum ApcEvent {
    TcpConnected(TcpStream),
    ResponseValid(bool),
    IncomingTcpClosed,
    AspMsgIn(Vec<u8>),
    NopTimeout,
    Finished,
    Uct,
}
```

Example adapted from comment on https://hoverbear.org/2016/10/12/rust-state-machine-pattern/

Both the states and events are encoded as enums, with parameters to those enums holding state variables that provide

additional context. Having defined the enums representing the states and events, define a function that maps between states.

```
impl ApcState {
    fn next(self, event: ApcEvent) -> Self {
        use self::ApcState::*;
        use self::ApcEvent::*;

        match (self, event) {
            (Initialize,         TcpConnected(tcp)   ) => Accept(tcp),
            (Initialize,         Finished            ) => Finish,
            (Accept(tcp),        ResponseValid(true) ) => StartTransfer(tcp),
            (Accept(_),          ResponseValid(false)) => Closed,
            (StartTransfer(tcp), Uct                 ) => Waiting(tcp),
            (Waiting(_),         IncomingTcpClosed   ) => Closed,
            (Waiting(_),         Finished            ) => Finish,
            (Waiting(tcp),       AspMsgIn(msg)       ) => ReceiveMsg(tcp, msg),
            (Waiting(tcp),       NopTimeout          ) => SendNop(tcp),
            (ReceiveMsg(tcp, _), Uct                 ) => Waiting(tcp),
            (SendNop(tcp),       Uct                 ) => Waiting(tcp),
            (s, e) => Failure(format!("Invalid State/Event combination: {:?}/{:?}", s, e)),
        }
    }
}
```

The enums representing the states and events and the state transition function that maps between the states are brought together in a new struct representing the struct itself

```
pub struct ApcStateMachine {
    state  : ApcState,
    addr   : SocketAddr,
    timeout: u64,
}

impl ApcStateMachine {
    fn new() -> ApcStateMachine {
        ...
    }

    fn run_once(&self) -> ApcEvent {
        match self.state {
            Initialize          => ...
            WaitForConnect      => ...
            Accept(tcp)         => ...
            StartTransfer(_)    => ...
            Waiting(tcp)        => ...
            ReceiveMsg(_, msg)  => ...
            SendNop(_)          => ...
            Closed              => ...
            Finish              => ...
        }
    }
}
```

```
fn run_state_machine() {
    let mut sm = ApcStateMachine::new();
    loop {
        let event = sm.run_once();
        sm.state = sm.state.next(event);
        if sm.state == ApcState::Finish {
            break;
        }
    }
}
```

This cleanly separates the actions to be performed in each state, from the code that manages the state transitions. It cleanly encodes the state transition logic into a single function.

3.3 Structures for Modelling State Machines

Rust also permits an alternative way to model state machines, based around structs. Each state is represented by a struct - one struct per state. Events are represented by method calls on those structs, and state transitions are modelled by returning a struct that represents the new state.

3.3.1 Using `struct` to Model State Machines: Example

Take an example system with three possible states (Unauthenticated Connection, Authenticated Connection, Not Connected). Each is represented by a struct type, a number of methods are implemented on these structs.

3.4 Approaches to Representing State Machines

Which approach is dependent on your priorities. The enum approach is compact, makes states and events clear in the types, and its good if the state machine is complex. It relies on a language that has expressive enum types, to allow its implementation.

The struct approach encodes states and state transitions in the types and events as methods on those types. The state transition table is less obviously explicit, since its encoded in return types.

4 Ownership

Systems programs care about the ownership of resources, it is important for memory and resource management. The programmer maintains a mental model of what parts of the code owns each resource,, i.e. what function is responsible for calling `free()`, `close()`, etc. Garbage collected languages still require understanding of ownership, but make free() calls automatically.

4.1 Ownership in Rust

Rust tracks ownership of data, it enforces that every value in the program has a has a single owner at all times. To do this, Rust's type system defines rules about the transfer of ownership of data in function and method calls. There are three cases;

- Taking explicit ownership of a value - passing by value

```
fn consume(r : Resource) {
  ...
}
```

- Borrowing a value - passing by reference

```
fn borrow(r : &Resource) {
  ...
}
```

- Return ownership of a value - gives ownership to caller

```
fn generate() -> Resource {
  ...
}
```

4.2 State Machines and Ownership

State transitions indicates changes to resource ownership. They indicate that some event has occurred, and that the system must move to a new state. Potentially consuming or releasing resources held by the old state, or keeping them for use by the new state.

Looking again at the struct approach to writing state machines, transitions are represented by methods implemented on a struct. Importantly, these methods take ownership of the struct, i.e. they consume the state they're transitioning from, ensuring there are no more references to that state and any resource they don't explicitly return are freed.

The transition methods return ownership of a value representing a new state, populated with any values that need to be retained from the previous state.

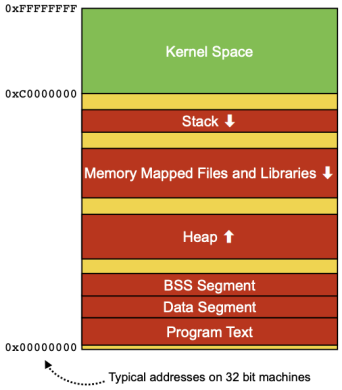
This is the advantage of struct based approach to state machines. It uses Rust's ownership rules to enforce state transitions, and to guarantee that resource and cleaned up when state transitions. The struct approach is good for ensuring that all resources are all cleaned up after use, which is better if your state machine manages a complex set of resources.

The enum approach makes the state transition diagram clearer, but relies of programmer discipline to manage and clean-up resources.

Resource Ownership and Memory Management

1 Memory

To understand memory management, we must first understand what memory is to be managed.



- Program text
- Heap allocated memory
- Stack
- Memory mapped files and shared library
- Operating system kernel

Program text, data, and global variables occupy the lowest address space. Page with address space 0 is reserved to trap null-pointer dereferences. Program text is compiled machine code of program.

Data segment is variables initialised in source code, such as string literals, initialised static global variables in c. Known as compile time, loaded along with program text.

BSS segment reserved fro uninitialised static global variables. Initialised to zero by runtime when the program loads.

1.1 The Stack

The stack holds function parameters, return addresses, local variables. Function calls push data onto stack, growing down. Includes parameters for the functions, return address, pointer to previous stack frames, local variables. When data is removed, stack shrinks. When function returns, stack is managed automatically.

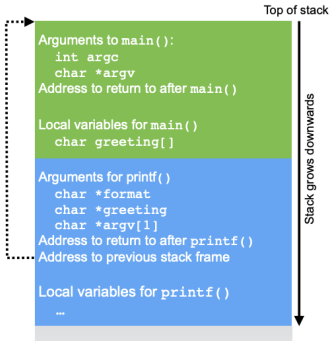
Compiler generates code to manage the stack as part of the compiled program - the calling convention for functions, how parameters are pushed onto the stack are standardised for given processor and programming language. The operating system generates the stack frame for main() when the program starts.

Ownership of stack memory follows function invocation.

1.1.1 Function Calling Conventions

Example: Code and contents of stack while calling `printf()` in code below

```
#include <stdio.h>

int
main(int argc, char *argv[])
{
  char greeting[] = "Hello";

  if (argc == 2) {
    printf("%s, %s\n", greeting, argv[1]);
    return 0;
  } else {
    printf("usage: %s <name>\n", argv[0]);
    return 1;
  }
}
```



Address of the previous stack frame is stored for ease of debugging, so stack trace can be printed, so it can easily be restored when function returns.

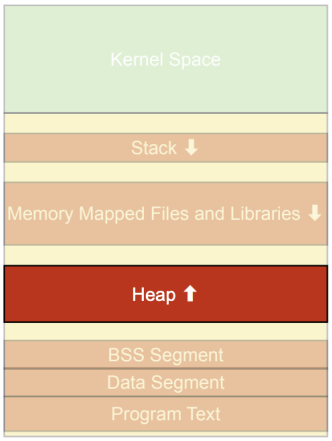
1.1.2 Buffer Overflow Attacks

In a classic buffer overflow attack - a not-type safe language is used which doesn't enforce abstractions, writes past array bounds, and overflows space allocated to local variables overwriting return address and following data. The overwritten function return address is made to point to that code, when a function returns, code written during overflow is executed.

Workarounds for this are making stacks non executable, randomising top of stack address each program run. However various more complex buffer overflow attacks still possible, so solve this, we use a language that is type safe and enforces array bounds checks.

1.2 The Heap

The heap holds explicitly allocated memory, allocated using `malloc()` or `calloc()` in C. Starts at a low address in memory, later allocations follow in consecutive addresses, sometimes padded to align to a 32 or 64 bit boundary, depending on the processor. Modern `malloc` implementations are thread aware, splitting heap into different parts for different threads to avoid cache sharing.



Memory management primarily concerned with reclaiming heap memory. Either manually using `free()`, or automatically via reference counting/garbage collection, or based on regions and lifetime analysis.

1.2.1 Managing the Heap

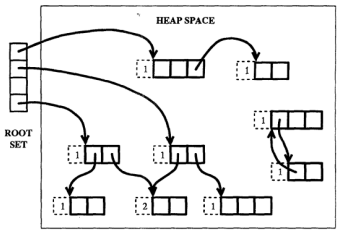
Our aim is to find object that are no longer used, and make their space available for reuse. An object is no longer used (ready for reclamation) if it not reachable by the running program via any path of pointer traversals. Any object that is

potentially reachable is preserved, it is better to waste memory than deallocate an object that's in use.

Primary approaches to automatic heap management are reference counting, region-based lifetime tracking, and garbage collection.

1.3 Reference Counting

Reference counting is the simplest automatic heap management approach. Allocations contain extra space for an additional reference count. An extra int is allocated along with every object, counting number of references to the object. This increments when a new reference to the object is created, and decrements when a reference is removed.



When a reference count reaches zero, there are no references to the object, and it may be reclaimed. Reclaiming objects removes references to other objects, which may reduce their reference count to zero, triggering further reclamation.

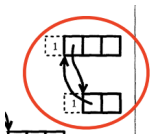
1.3.1 Benefits

This is an incremental operation, memory is reclaimed in small bursts. It is also predictable and understandable, it is:

- Easy to explain
- Easy to understand when memory is reclaimed
- Easy to understand overheads and costs
- Follows programmer intuition.

1.3.2 Costs

Cyclic data structures contain mutual references, objects that reference each other aren't reclaimed, as reference count doesn't go to zero. Memory leas unless cycle explicitly broken



Stores reference counts alongside each object, maybe also a mutex if concurrent access possible, per object. overhead significant for small objects, wasting memory.

Additional processor cost of updating references can be significant for short lived objects

1.3.3 Usage

Reference Counting is widely used in scripting languages (python, ruby, etc). Memory and processor overhead not significant in interpreted runtime.

Used on small scale for systems programming, e.g. objective C runtime on iOS. Ease of understanding is important, tends to be for large, long-lived data, reducing overhead. Not typically used in kernel code and high-performance systems.

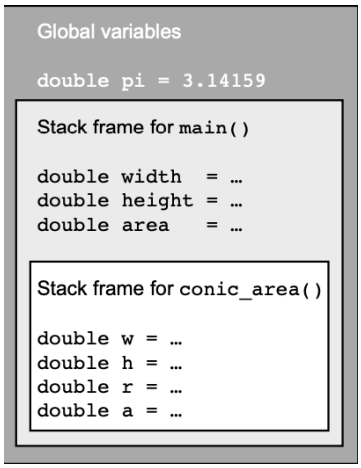
2 Region Based Memory Management

Reference counting has relatively high overhead, memory overhead to store the reference count, and processor time to update reference counts. Garbage collection has unpredictable timing ans high overhead. Manual memory management is too error prone.

Region-based memory management aims for a middle ground - safe, predictable timing, with no runtime cost, and accepts some impact on application design.

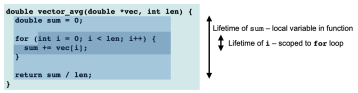
2.1 Stack-based Memory Management

Automatic management of stack variables is common and efficient



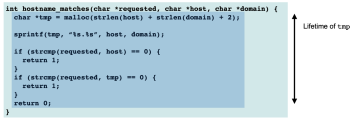
Hierarchy of regions correspond to call stack:

- Global variables
- Local variables in each function
- Lexically scoped variables within functions.



Variables live in regions, and are deallocated at th end of region scope.

This does come with a limitation, being that it requires data to be allocated on the stack, for example



Local variable `tmp` stored on the stack and is freed when function returns. Memory allocated by `malloc()` is not freed - memory leak.

2.2 From Stack-to Region-based Memory Management

Stack-based memory management is effective, but limited applicability - can we extend to manage the heap? Track lifetime of data - values on the stack and references to the heap.

A `Box<T>` is a value stored on the stack that holds a reference to data of type `T` allocated in the heap.

```
fn main() {
    let b = Box::new(5);
    println!("b = {}", b);
}
```

`b` is a pointer to heap allocated memory, holding integer value 5

The `Box<T>` is a normal local variable with lifetime matching the stack frame. The heap allocated `T` has lifetime matching the `Box<T>` - when the box goes out of scope, the referenced heap memory is freed, i.e. the destructor of the `Box<T>` frees the heap allocated `<T>`. This is RAII to c++ programmers

It is efficient but loses generality of heap allocation since heap lifetime is tied to stack frame lifetime.

2.3 Region Based Memory Management

For effective region based memory management

- Allocate objects with lifetimes corresponding to regions
- Track object ownership and changes to ownership
  - What region owns each object at any time
  - Ownership of objects can move between regions
- Deallocate objects at the end of the lifetime of their owning region, using scoping rules to ensure objects are not referenced after deallocation.

The Rust programming language is an example, builds on previous research with Cyclone language, Microsoft's Singularity operating system uses somewhat similar ideas.

2.4 Returning Ownership of Data

Ownership of return value is moved to the calling function. The value is moved into the calling function's stack frame. The origin value, in the called functions stack frame is deallocated. This allows us to return a `Box<T>` that references a heap allocated value of type `T`. The `Box<T>` is moved, but the reference `<T>` on the heap is not.

Variables not returned by a function go out of scope and are reclaimed. The heap allocated `T` is deallocated when the `Box<T>` goes out of scope and is reclaimed, i.e. the compiler generates the equivalent of a call to free when the box goes out of scope.

2.4.1 No Dangling References

Lifetime of local variable ends when function returns. Can't return a reference to an object that doesn't exist.

```
fn foo() -> &i32 {
  let n = 42;
  &n
}
```

```
% rustc test.rs
error[E0106]: missing lifetime specifier
 --> test.rs:1:13
  |
1 | fn foo() -> &i32 {
  |             ^ expected lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but there is no
  = help: value for it to be borrowed from
```

```
int *foo() {
  int n = 42;
  return &n;
}
```

Equivalent C code will compile but crash at runtime - good compilers give a warning for many but not all cases.

2.4.2 No Use-After-Free

Similarly, once memory is freed, it cannot be accessed, explicit `drop()` is equivalent of `free()` in C

```
use std::mem::drop; // free() equivalent

fn main() {
  let x = "Hello".to_string();
  drop(x);
  println!("{}", x);
}
```

```
error[E0382]: use of moved value: `x`
 --> test.rs:6:18
  |
5 |   drop(x);
  |        - value moved here
6 |   println!("{}", x);
  |                  ^ value used here after move
  |
  = note: move occurs because `x` has type `std::string::String`, which does not implement the `Copy` trait
```

```
#include <stdlib.h>
#include <stdio.h>

int main() {
  char *x = malloc(14);
  sprintf(x, "Hello, world!");
  free(x);
  printf("%s\n", x);
}
```

Equivalent C program compiles and runs, but has undefined behaviour

2.5 Taking Ownership of Data

Ownership of data passed to a function is transferred to that function. It is deallocated when function ends, unless data is returned by that function. Data cannot be later sued by the caller function - enforced at compile time.

```
fn consume(mut x : Vec<u32>) {
  x.push(1);
}

fn main() {
  let mut a = Vec::new();

  a.push(1);
  a.push(2);

  consume(a);

  println!("a.len() = {}", a.len());
}
```

Lifetime of a

Ownership of a transferred to consume()

this would give the error `15:29 error : use of moved value 'a'`

3 Resource Management

3.1 Borrowing Data

Functions can take references to data, borrowing the data rather than moving ownership. Ownership of the **reference** is moved, not the **referenced value**.

```
fn borrow(mut x : &mut Vec<u32>) {
  x.push(1);
}

fn main() {
  let mut a = Vec::new();

  a.push(1);
  a.push(2);

  borrow(&mut a);

  println!("a.len() = {}", a.len());
}
```

```
% rustc borrow.rs
% ./borrow
a.len() = 3
%
```

Functions can also return references to borrowed input parameters - the parameters are borrowed from the calling function, so safe to return them to it.

3.1.1 Safe Borrowing

Rust has two kinds of pointer:

- `&T` - shared reference to immutable object
- `&mut T` - unique reference to mutable object

The compiler and runtime control reference ownership and use. An object of type `T` can be referenced by one or more references of type `&T`, or by exactly one reference of type `&mut T`, but not both.

Cannot get an `&mut T` reference to data of type `T` that is marked as immutable, existence of an `&T` reference to mutable data makes the data immutable.

Allows functions to safely borrow objects, without needing to give away ownership. To change an object, you either own

the object, and it is not marked as immutable, or you own the only `&mut` reference to it.

This prevents iterator invalidation. The iterator requires an `&T` reference, so other code can't get a mutable reference to the contents to change them this is enforced by the compiler.

```
fn main() {
  let mut data = vec![1, 2, 3, 4, 5, 6];
  for x in &data {
    data.push(2 * x);
  }
}
```
fails, since push takes an &mut reference

3.1.2 Benefits

The type system tracks ownership, turning runtime bugs into compile time errors, preventing use-after-free bugs, iterator invalidation, and race conditions with multiple threads - borrowing rules prevent two threads from getting references to a mutable object.

Efficient run-time behaviour. Generates exactly the same code as a correctly written program using `malloc()` and `free()`. Timing and memory usage are as predictable as a correct C program. Deterministic when memory allocated and when memory freed.

3.2 Limitations of Region-Based Systems

Can't express cyclic data structures, for example, you can't build a doubly linked list in safe Rust.

Can't get a mutable reference to c to add the link to d since already referenced by b

Many languages offer an escape hatch from the ownership rules to allow these data structures, e.g. raw pointers and `unsafe` in Rust.

Can't express shared ownership of mutable data, which is usually a good thing since it avoids race conditions. Rust has `RefCell<T>` that dynamically enforces the borrowing rules - i.e. allows upgrading a shared reference to an immutable object into a unique reference to a mutable object, it it was the only such shared reference. Fails a run-time exception if there could be a race, rather than preventing it at compile time.

Forces considerations of object ownership early and explicitly - generally good practice, but increases conceptual load early in design process - may hinder exploratory programming.

4 Managing Resources

4.1 Deterministic Clean-up

Rust deterministically frees ("drops") memory when references go out of scope. Types can implement the `Drop` trait to get custom destructors

```
impl Drop for MyType {
    fn drop(&mut self) {
        ...
    }
}
```

```
pub trait Drop {
    fn drop(&mut self);
}
```
Definition of `std::ops::Drop`

Dropping is deterministic → clean-up resource ownership. Garbage collected languages typically give no guarantee when the destructor runs. For example, the File class uses a custom drop implementation to close the file when it goes out of scope. Python has special syntax for this, but this is unnecessary in Rust.

4.2 Ownership And States

Ownership transfer between different types can be used to model resources states. Manages the different states of a resource, and makes illegal operations compile time errors.

Garbage Collection

1 Basic Garbage Collection

Avoid problems of reference counting and complexity of compile-time ownership tracking via **garbage collection**. Explicitly trace through allocated objects, recording which are in use, and dispose of unused objects. Moves garbage collection to a separate phase of the program's execution, rather than an integrated part of an objects lifecycle - operation of the program (the **mutator**) and the garbage collector is interleaved.

Many tracing garbage collection algorithms exist - basic garbage collectors like mark-sweep, mark-compact, and copying collectors, as well as generational garbage collectors.

1.1 Mark-Sweep Collectors

The simplest garbage collection scheme, consisting of a two phase algorithm,

- Distinguish live objects from garbage (*mark*)
- Reclaim the garbage (*sweep*)

. This is non-incremental, the program is paused to perform collection when memory becomes tight.

The **marking phase** distinguishes live objects by:

- Determine the **root set** of objects, comprising any global variables, and any variables allocated on the stack, in any existing stack frame.
- Traverse the object graph starting at the root set to find other reachable object - starting from the root set, follow pointers to other objects, follow every pointer in every objects to systematically find all reachable objects. A cycle of objects that reference each other but aren't reachable from the root set will not be marked.
- Mark reachable objects as alive - set a bit in the object header, or in some separate table of live objects, to indicate that the object is reachable. Stop traversal at previously seen objects to avoid looping forever.

In the **sweep phase**, we reclaim objects that no longer live. Pass through the entire heap once, examining each object for liveness. If marked as alive, keep the object, otherwise, free the memory and reclaim the object's space.

When objects are reclaimed, their memory is marked as available. The system maintains a free list of blocks of unused memory. New objects are allocated in now unused memory if they fit, or in not-yet-used memory elsewhere on the heap. Fragmentation is a potential concern, but no worse than using malloc and free.

1.1.1 Mark-Sweep Summary

Mark-sweep collectors are simple but inefficient, their garbage collection is slow and has unpredictable duration. The program is stopped while the collector runs, and the time to collect is unpredictable and depends on the number of live

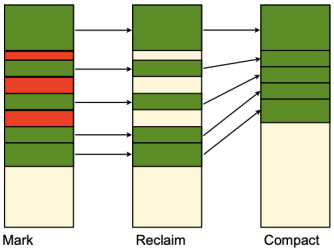
objects and size of the heap. Unlike reference counting, mark-sweep garbage collection is slower if the program has lots of memory allocated.

1.2 Mark-Compact Collectors

The goal is to solve the fragmentation problems and sped up allocation in comparison to mark-sweep collectors. Mark-compact has three logical phases.

- Traverse object graph, **mark** live objects
- **Reclaim** unreachable objects
- **Compact** live objects, moving them to leave contiguous free space

Reclaiming and compacting memory can be done in one pass, but still touches the entire address space.



1.2.1 Pros and Cons

Advantages

- Solves fragmentation problems, all free space is in one contiguous block
- Allocation is very fast - always allocating from the start of the free block, so allocation is just incrementing pointer to start of free space and returning previous value.

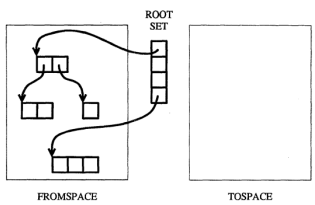
Disadvantages

- Collection is slow, due to moving objects in memory, and time taken is still unpredictable
- Collection has poor locality of reference
- Collection is complex, needs to update all pointers to moved objects.

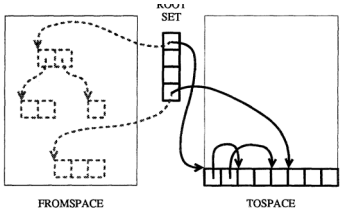
1.3 Copying Collectors

Copying collectors integrate traversal (marking) and copying phases into one pass. All live data is copied into one region of memory, all the remaining memory contains garbage or has not yet been used. This is similar to mark-compact but more efficient - time taken to collect is proportional to the number of live objects.

Stop-and-copy using semispaces - Divide the heap into two halves, each one a contiguous block of memory. Allocations made linearly from one half of the heap only. Memory is allocated contiguously, so allocation is fast (as in the mark-compact collector). No problems with fragmentation when allocating data of different sizes. When an allocation is requested that won't fit into the active half of the heap, a collection is triggered.



Collection stops execution of the program, a pass is made through the active space, and all live objects are copied to the other half of the heap. Cheney algorithm is commonly used to make the copy in a single pass. Anything not copied is unreachable, and is simply ignored, and will eventually be overwritten. The program is then restarted, using the other half of the heap as the active allocation region. The role of the two parts of the heap - the two semispaces - reverse each time a collection is triggered

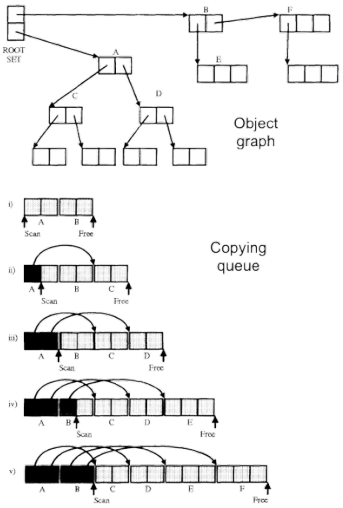


1.3.1 Cheney Algorithm

The Cheney algorithm - breadth first copying. A queue is created, to hold the set of live objects to be copied. The root set of objects, comprising global variables and all stack allocated variables, is found and added to the queue.

Objects in the queue are examined in turn - any unprocessed objects they reference are added to the end of the queue, the object in the queue is then copied into the other semispace, and the original is marked as having been processed, updating the pointers as the copy is made.

Once the end of the queue is reached, all live objects have been copied.



1.3.2 Copying Collectors Summary

Efficiency of copying collectors - time taken for garbage collection depends on the amount of data copied, which depends on the number of live objects. Collection only happens when a semispace is full.

If most objects die young, can trade-off collection time vs memory usage by increasing the size of the semispaces. A larger semispace takes longer to fill, increasing how long objects need to live before they're copied. If most objects die young, they aren't copied. Uses more memory, but spends less time copying data.

2 Generational and Incremental Garbage Collection

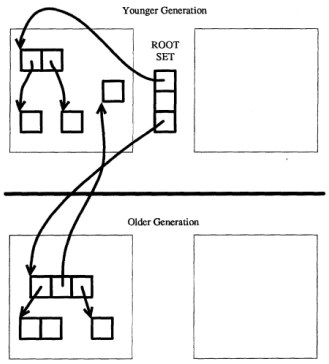
2.1 Object Lifetimes

Most objects have a short time, only a small percentage live much longer. This seems to be generally true, no matter what programming language is considered, across numerous studies. Although, obviously, different programs and different languages produce varying amount of garbage.

The implications are that when the garbage collector runs, live objects will be in a minority, statistically, the longer an object has lived, the longer it is likely to continue living. How can we design a garbage collector to take advantage of this

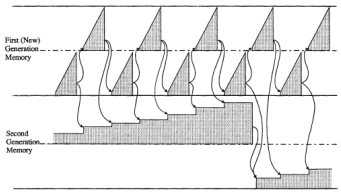
2.2 Copying Generational Collectors

In a generational garbage collector, the heap is split into regions for long lived and young objects. Regions holding young objects are garbage collected more frequently. Objects are moved to the region for long-lived objects if they're still alive after several collections. More sophisticated approaches may have multiple generations, although the gains diminish rapidly with increasing number of generations.



Example: stop-and-copy using semispaces with two generations :

- All allocations occur in the younger generation's region of the heap.
- When that region is full, collections occur as normal
- Objects are tagged with the number of collections of the younger generation they've survived, if they're alive after some threshold, they're copied to the older generation's space during collections.
- Eventually, the older generations space is full, and is collected as normal



Young generation must be collected independent of long-lived generation. But, there may be references between generations. **References from young to long-lived objects** - straight-forward - must young objects die before the long-lived objects are collected, treat the younger generation as part of the root set for the long-lived generation, when collection of the long-lived generation is needed.

References from long-lived to young objects - Problematic, since it requires a scan of long-lived objects to detect. Maybe use indirection table (pointers-to-pointers) for references from long-lived to young generation. The indirection table forms part of the root set of the younger generation, moving objects in younger generations requires updating indirection table, but not long-lived objects. Long-lived objects are collected infrequently, and may keep younger objects alive longer than expected.

Variations on copying generation collectors are widely used, e.g. the HotSpot JVM uses a generational garbage collector. Copying generational collectors are efficient, cost of collection is generally proportional to number of live objects, most objects don't live long enough to be collected; those that do not are moved to more rarely collected generation. Longer-lived generation must eventually be collected, but this can be very slow.

2.3 Incremental Garbage Collection

Preceding discussion has assumed the collectors stops the program when it runs, this is problematic for interactive or real time applications. We want a collector that can operate incrementally, interleaving small amounts of garbage collection with small runs of program execution.

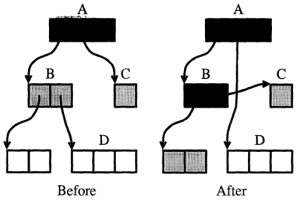
Implication: the garbage collector can't scan the entire heap when it runs, it must scan a fragment of the heap each time. Problem: the program (the mutator) can change the heap between runs of the garbage collector. We need to track changes made to the heap between garbage collector runs, be conservative and don't collect objects that might be referenced - can always collect on the next complete scan.

Tricolour marking: Each object is labelled with a colour

- White - not yet checked
- Grey - live, but some direct children not yet checked
- Black - live

Basic Incremental collector operation:

- Garbage collection proceeds with a wavefront of grey objects, where the collector is checking them, or objects they reference, for liveness.
- Black objects behind are behind the wavefront, and are known to be live.
- Objects ahead of the wavefront, not yet reached by the collection, are white - anything still white once all objects have been traced is garbage.
- No direct pointers from black objects to white - any program operation that will create such a pointer requires coordination with the collector.



Program and collector must coordinate, garbage collector runs - object A is scanned, known to be live, is black. Objects b and C are reachable via A, but some of their children have not been scanned, they are gray. Object D not checked, it is white.

Program runs, and swaps the pointers from A to C and B to D such that A $\rightarrow$ D and B $\rightarrow$ C. This creates a pointer from black to white. The program must now coordinate with the collector, else collection will continue, making B black and its

children grey, but D will not be reached since children of A have already been scanned.

Coordination Strategies

- Read Barrier: trap attempts by the program to read pointers to white objects, colour those objects gray, and then let program continue. Makes it impossible for the program to get a pointer to a white object, so it cannot make a black object point to a white.
- Write barrier: Trap attempts to change pointers from black objects to point to white objects. Either then recolour the black object as grey, or recolour the white object being referenced as grey. The object coloured grey is moved onto the list of objects whose children must be checked

There are many variants on read- and write-barrier tricolour algorithms. Performance trade-offs differ depending on hardware characteristics, and on the way pointers are represented. Write barrier generally cheaper to implement than read barrier, as writes are less common in most code. There is a balance between collector operation and program operation - if the program tried to create too many new references from black to white objects, requiring coordination with the collector, the collection may never complete. Resolve by forcing a complete stop-the-world collection if free memory is exhausted, or after a certain amount of time.

3 Practical Factors

3.1 Real-time Garbage Collection

Real time collectors are built from incremental collectors. Schedule an incremental collector as a periodic task - Runtime allocated determines amount of garbage that can be collected in each period. The amount of garbage that can be collected can be measured - how fast can the collector scan memory (and copy objects, if a copying collector). The programmer must bound the amount of garbage generated to within the capacity of the collector. Hard real time systems must always stay within the bounds of the collector. Soft real time systems meet statistical bounds.

3.2 Memory Overheads

Garbage collection trades ease-of-use for predictability and overhead. Garbage collected programs will use significantly more memory than correctly written programs with manual memory management. Many copying collectors maintain two semispaces, so double memory usage. But - many programs with manual memory management are not correct.

3.3 Interaction with Virtual Memory

Virtual memory subsystems page out unused memory in an LRU manner. Garbage collector scans objects, paging data back into memory. Leads to thrashing if the working set of the garbage collector is larger than memory - open research issue, combining virtual memory with garbage collector.

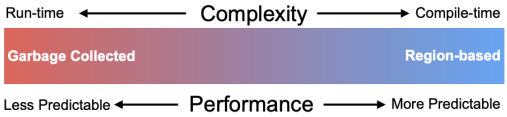
3.4 Garbage Collection for Weakly-Typed Languages

Collectors rely on being able to identify and follow pointers, to determine what is a live object - they rely on strongly typed languages. Weakly typed languages, such as C, can cast any integer to a pointer, and perform pointer arithmetic. Implementation defined behaviour, since pointers and integers are not guaranteed to be the same size.

Difficult, but not impossible, to write a garbage collector for C: need to be conservative, any memory that might be a pointer must be treated as one. Boehms-Demers-Weiser collector can be used for C and C++ - works for strictly conforming ANSI C code, but beware that much code is not conforming.

Other weakly typed languages may suffer similar problems, not an issue for strongly typed but dynamic languages such as python.

3.5 Memory Management Trade-Offs



Rust pushes memory management complexity onto the programmer, giving us predictable run-time performance, and low runtime overheads. Uniform resource management framework, including memory. Limits the programs that may be expressed - matches good patterns in good C code.

Garbage collection imposes run-time costs and complexity, simpler for programmer.

Concurrency

1 Implications of Multicore

1.1 Memory Models and Multicore Systems

Hardware is trending toward multicore systems with non-uniform memory access, increasing number of cores for performance. Cache coherency is increasingly expensive, cores communicate by passing messages, and memory is remote. Each core has its own private cache not shared with other cores, so each processor has its own distinct view of memory. Memory is also not always equally accessible to all cores.

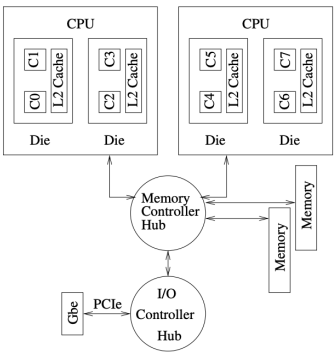


Figure 1. Structure of the Intel system

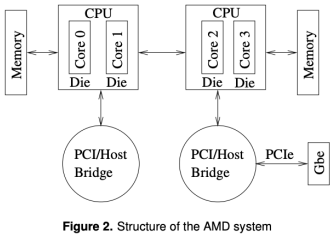


Figure 2. Structure of the AMD system

We must consider when writes made by one core become visible to the others? and what the memory model for the language is? It is prohibitively expensive for all threads on all cores to have the exact same view of memory. For performance, we allow cores inconsistent views of memory, except at synchronisation points. Introduce synchronisation primitives with well-defined semantics.

Hardware guarantees vary between processors, and differences are hidden by language runtime, provided language has a clear memory model.

1.1.1 Memory Models: Java

Java was a formally defined memory model. Changes to a field are seen in program order within a thread. Changes to a field are seen in program order within a thread, and changes to a field made by one thread are visible to other threads as follows.

- If a `volatile` field is changes, that change is done atomically and immediately becomes visible to other thread.
- If a non-`volatile` field is changes while holding a lock, and that lock is then released by the writing thread and acquired by the reading thread, then the change becomes visible to the reading thread.
- If a new thread is created, it sees the state of the system as if it had just acquired a lock that had just been released by the creating thread.
- If a thread terminates, changes it made become visible to other threads.
- Access to all 32-bit fields is atomic, i.e. you can never observe a half-way completed write, even if incorrectly synchronised. This is not the case for `long` and `double` fields, which are 64-bits in size, where writes are only atomic if field is `volatile` or if a lock is held

1.1.2 Memory Models: Others

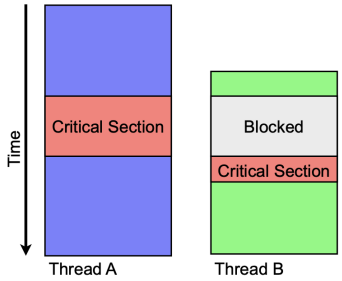
Java is unusual in having such a clearly-defined memory model. Other languages are less well specified, running the risk that new processor designs can subtly break previously working programs.

C and C++ have historically had *very* poorly specified memory models. Latest versions of standards address this, with memory models heavily influenced by the Java memory model, but this is not yet widely implemented.

Rust does not yet have a fully specified memory model, recognised as a limitation, but research efforts are underway to fix this. Complicated by multiplicity of reference types and `unsafe` code.

1.2 Concurrency, Threads, and Locks

Operating system exposes concurrency as processes and threads. Processes are isolated with separate memory areas. Threads share access to a common pool of memory. Most operating systems started with processes and message passing, and added threads later due to programmer demand.



The memory model specifies how concurrent access to shared memory works. Synchronisation by explicit locks around critical sections. `synchronized` methods and statements in Java, and `pthread_mutex_lock()/...unlock()` in C. Limited guarantees about unlocked concurrent access to shared memory.

1.2.1 Limitations of Lock-Based Concurrency

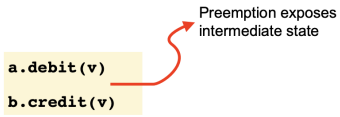
There lie some major problems with lock-based concurrency, namely:

- Difficult to define a memory model that enable good performance, while allowing programmers to reason about the code.
- Difficult to ensure correctness when composing code - difficult to enforce correct locking and guarantee freedom from deadlocks
- Failures are silent, errors tend to manifest only under heavy load
- Balancing performance and correctness difficult, easy to over- or under-lock systems

1.2.2 Composition of Lock-Based Code

Correctness of small-scale code using can, in theory, be ensured by careful coding. A more fundamental issue is that lock-based code does not compose to larger scale.

Assume a correctly locked bank account class, with methods to credit and debit money from an account. Want to take money from `a` and move it to `b`, without exposing an intermediate state where the money is in neither account.



Can't be done without locking all other access to `a` and `b` while the transfer is in progress. The individual operations are correct, but the combined operation is not.

This lack of abstraction is a limitation of the lock-based concurrency model, and cannot be fixed by careful coding. Locking requirements form part of the API of an object.

1.3 Alternative Concurrency Models

Concurrency is becoming increasingly important. Multicore systems are now ubiquitous, and we require asynchronous interactions between software and hardware devices. Threads and synchronisation primitives are problematic, but are there alternatives that avoid these issues?

- **Transactions**
- **Message Passing**

2 Managing Concurrency using Transactions

An alternative to locking, we can use atomic transactions to manage concurrency. A program is a sequence of concurrent atomic actions, atomic actions succeed or fail in their entirety, and intermediate states are not visible to other threads. The runtime must ensure actions have the ACID properties.

- **A**tomicity - All changes to the data are performed, or none are.
- **C**onsistency - Data is in a consistent state when a transaction starts and when it ends.
- **I**solation - Intermediate states of a transaction are invisible to other transactions.
- **D**urability - Once committed, results of a transaction persist.

This has the advantages that transactions can be composed arbitrarily without affecting correctness and avoids deadlock due to incorrect locking, since there are no locks.

2.1 Programming Model

Follows a simple programming model, blocks of code can be labelled `atomic {...}`, and these run concurrently and atomically with respect to every other `atomic {...}` block. This controls concurrency and ensures consistent data structures.

This is implemented using **optimistic transactions**. A thread-local transaction log is maintained, and records every memory read and write made by the atomic block. When an atomic block completes, the log is *validated* to check that it has seen a consistent view of memory. If validation succeeds, the transaction *commits* its changes to memory, otherwise, the transaction is rolled-back and retried from scratch. Progress may be slow if conflicting transactions cause repeated validation failures, but will eventually occur.

2.1.1 Consequences

Transactions may be rerun automatically, if their transaction log fails to validate. Places restrictions on transaction behaviour:

- Transactions must be referentially transparent
- Transactions must do nothing irrevocable (I/O operations), i.e. the example below might launch the missiles multiple times, if it gets rerun due to validation failure caused by `doMoreStuff()`. Might accidentally launch the missiles if a concurrent transaction modifies n or k while the transaction is running, which will cause transaction failure but too late to stop the launch
- These restrictions must be enforces, else we trade hard-to-find locking bugs for hard-to-find transaction bugs

```
atomic(n, k) {
  doSomeStuff()
  if (n > k) then launchMissiles();
  doMoreStuff();
}
```

2.2 Controlling I/O

Unrestricted I/O breaks transaction isolation, e.g. if a transaction reads and writes files, sends and receives data over the network, takes mouse or keyboard input, etc. then the progress of the transaction can be observed before it commits, breaking the ACID properties, isolation, and making it impossible to roll-back the transaction.

To address this, the language and runtime need to control when I/O is performed. To do this, they remove the global functions to perform I/O from the standard library, and add an I/O context object, local to `main()`, passed explicitly to functions that need to perform I/O. We can compare sockets, that behave in this manner, with file I/O that typically does not. I/O functions (e.g. `printf()` and friends) then become methods on the I/O context object. The I/O context is not passed to transactions, so they cannot perform I/O. This is how I/O works in Haskell, the I/O context object described is essentially the I/O monad.

2.3 Controlling Side Effects

Code that has side effects must be controlled. Pure and referentially transparent functions can be executed normally. Functions that only perform memory actions can also be executed normally, provided transaction log tracts the memory actions and validates them before the transaction commits, and can potentially roll them back.

A memory action is an operation that manipulates data on the heap, that could be seen by other threads. Tracking memory actions can be done by language runtime (STM; software transactional memory) or via hardware enforced transactional memory behaviour and rollback.

The principle is similar to that for controlling I/O. Disallow unrestricted heap access - only see data in transaction context. Pass transaction context explicitly to transactions, this has operations to perform transactional memory operations, and rollback if the transaction fails to commit. Very similar to the state monad in Haskell.

2.4 Monadic STM Implementation

Haskell limits the ability of a function to perform I/O or access memory by using monads, in other words, monads are a way to control side effects in functional languages.

A monad `M a` describes an action that produces a result of type `a`, that can be performed in the context `M`. A common use is for controlling I/O operations:

- The `putChar` function takes a character, operates on the `IO` context to add the character, and returns nothing.
- The `getChar` operates on the IO context to return a character.
- The main function has an IO context that wraps and performs other actions.

```
putChar :: Char -> IO ()
getChar :: IO Char
```

The definition of I/O monad type ensures that a function that is not passed the IO context cannot perform IO operations. One part of a software transactional implementation: ensure type of the `atomic {...}` function does not allow it to be passed an IO context, hence preventing IO.

How do we track side effecting memory actions? We define an STM monad to wrap transactions, based on the state monad, it manages side effects via a `TVar` type.

```
newTVar   :: a -> STM (TVar a)
readTVar  :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

- The `newTVar` functio takes a value of tpye a, returns new `TVar` to hold the value, wrapped in an STM monad.

- `readTVar` takes a `Tvar` and returns an STm context, when performed this returns the value of that `TVar`. `writeTVar` function takes a `TVar` and a vlue, and returns an STM context that can validate the transaction and commit the value to the `TVar`.

```
atomic :: STM a -> IO a
```

The `atomic{...}` function operates in an STM context and returns an IO context that performs the operations needed to validate and commit the transaction.

- the `newTVar`, `readTVar`, and `writeTVar` functions need an STM action, and so can only run in the context of an atomic block that provides such an action.
- IO prohibited within transactions, since operations in `atomic {...}`

2.5 Integration into Haskell

Transactional memory is a good fit with Haskell. Pure functions and monads ensure transaction semantics are preserved. Side effects are contained in `STM` context of an `atomic {...}` block.

The `Tvar` implementation is responsible for tracking side effects. The `atomic {...}` block validates, then commits the transaction by returning an action to perform in the IO context.

A `TVar` requires an `STM` context, but these are only available in an `atomic {...}` block. Can't update a `TVar` outside a transaction, so cant break atomicity guidelines - Haskell doesn't allow unrestricted heap access via pointers, so can't subvert. I/O cannot be performed within an `atomic{...}` block, the transaction is not in the IO context.

2.5.1 Integration into other languages

Atomic transactions in Haskell are very powerful, but rely on the type system to ensure safe composition and retry. Integrating this into mainstream languages is difficult, most languages cannot enforce use of pure functions, and cannot limit the use of IO and side effects. Transactional memory can be used without these, but requires programmer discipline to ensure correctness and has silent failure modes. It is unclear if the transactional approach generalises to other languages.

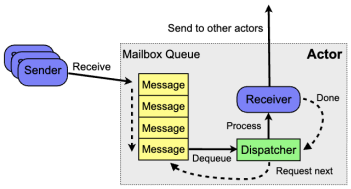
3 Message Passing Systems

System is structured as a set of communicating processes - actors with no shared mutable state. Actors communicate via sending messages to each other. Messages are generally required to be immutable, the data is conceptually copied between processes. In practice, its implemented by copying a reference, but since the data is immutable, this has no practical difference, you're exchanging messages that can't change. Alternatively, some systems use linear types to ensure messages are not referenced after they are sent, allowing mutable data to be safely transferred. This is essentially ownership tracking, to ensure that messages are not references after they are sent

The way this is implemented within a single system with shared memory is using threads and locks, passing a reference to the message, and rely on correct locking of message passing implementation. These systems are trivial to distribute, by sending the message down a network channel, the runtime needs to know about the network, but the application can be unaware that the system is distributed

3.1 Message Handling

Receivers pattern match against messages, against message types and not just values. Type systems can ensure an exhaustive match.



Messages arrive and are queued for processing. The dispatcher manages a thread pool servicing receiver components of the actors. Receivers operate in message processing loop - single threaded with no concern for concurrency. Sent messages enqueued for processing by other actors.

The entire system proceeds as a set of actors, each processing messages in a loop, sending messages out to other actors. It looks a lot like a networked system/server, it processes messages and it sends replies.

3.2 Types of Message Passing

There are several different message passing system designs:

- Synchronous vs asynchronous
- Statically or dynamically typed
- Direct or indirect message delivery

Each of these approaches has advantages and disadvantages.

3.2.1 Types of Message Passing: Interaction Models

Message passing can involve rendezvous between sender and receiver, in a synchronous message passing model, sender waits for receiver. Alternatively, the communication can be asynchronous, in which case the sender can send a message and continue without for the receiver to receive it. In this case, the message buffered and will eventually get delivered to the receiver. Synchronous rendezvous can be simulated by waiting for a reply.

3.2.2 Types of Message Passing: Typed Communication

Statically-typed communication:

- Explicitly defines the types of messages that can be transferred.
- Compiler checks that receiver can handle all messages it can receive - robustness is provided since a receiver is guaranteed to understand all received messages.

Dynamically-typed communication:

- Communication medium conveys any time of message, receiver uses pattern matching on the received message types to determine if it can respond to the messages.
- Potentially leads to run-time errors if a receiver gets a message that it doesn't understand

3.2.3 Types of Message Passing: Naming

Are messages send between named processes or indirectly via channels? Some systems directly send messages to actors (processes), each of which has its own mailbox. Others

use explicit channels, with messages being sent indirectly to a mailbox via a channel.

Explicit channels require more plumbing, but the extra levels of indirection between sender and receiver may be useful for evolving systems. Explicit channels are a natural place to define a communications protocol for statically typed messages.

3.3 Implementations

Actor based systems are actually starting to see fairly wide deployment, there are two different architectures being used.

Dynamically typed with direct delivery - adopted by the Erlang programming language, and by Scala and the Akka library. These systems are dynamically typed, you can send any type of message to any receiver. Messages are sent directly to named actors, not via channels, and both provide transparent distribution of processes in a networked system.

Statically typed with explicit channels - used in the Rust programming language, uses asynchronous statically typed messages via explicit channels

3.4 Examples

3.4.1 Scala + Akka

```
import akka.actor.Actor
import akka.actor.ActorSystem
import akka.actor.Props
class HelloActor extends Actor {
def receive = {
case "hello" => println("hello back at you")
case _ => println("huh?")
}
}
object Main extends App {
// Initialise actor runtime
val runtime = ActorSystem("HelloSystem")
// Create an actor, running concurrently
val helloActor = runtime.actorOf(Props[HelloActor],
                 name = "helloactor")
// Send it some messages
helloActor ! "hello"
helloActor ! "buenos dias"
}
```

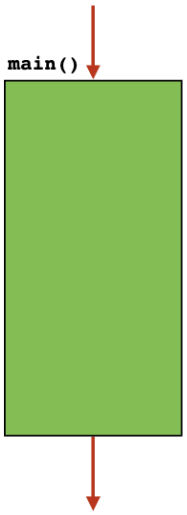
The actor comprises a receive loop that reacts to messages as they're received. A complete program is a collection of actors that exchange messages. We see the main object being created, "object Main extends App". This is the main object in the system, the equivalent of the main functions. This creates an actor runtime, and then creates am actor, for example, "val helloActor = runtime.actorOf" creates an actor of type HelloActor, and it gives it a name, and assigns this to an immutable variable HelloActor.

It then sends some messages, "hello", and "beunos dias" to the HelloActor. The HelloActor class has a receive method to pattern match on the types of message. The actor just runs in a loop and it just continually receieves these messages and processes them.

It can store data in the actor object, and that can include references to other actors which it's been sent, and that allows it to build up more complex communication patterns. Everything's very dynamic but it's quite low overhead. It's quite syntactically clean.

3.4.2 Rust

In the following Rust example. we have a main function, which creates a channel, spawns a thread, and sends a message between them.

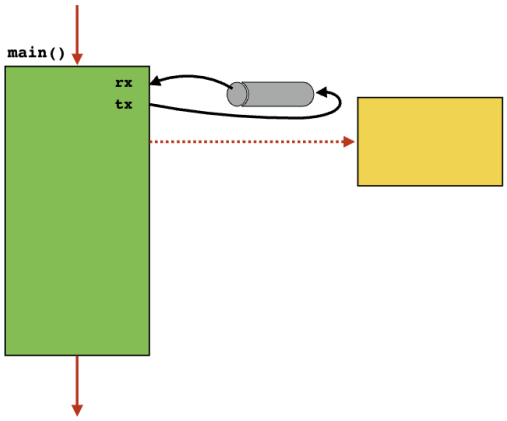



```
use std::sync::mpsc::channel;
use std::thread;
fn main() {
  let (tx, rx) = channel();
  thread::spawn(move|| {
    let _ = tx.send(42);
  });
  match rx.recv() {
    Ok(value) => {
      println!("Got {}", value);
    }
    Err(error) => {
      // An error occurred...
    }
  }
}
```

As is normal in Rust programs, we have a function, main(), which holds the entire state of the system. The first thing the main() function does it create a a channel, which is defined in the standard library, and is the inter-thread communication mechanism for Rust.

By calling the channel() call, it returns a tuple of the transmitting and receiving ends of that channel. The channels are unidirectional, you transmit into one end, and it comes out the other. The main() function now has references to both ends of the channel.

We call `thread::spawn()` to create a new thread in Rust, and again, this comes from the standard library, and the thread spawn() call creates a new thread of execution.



What gets executed by that thread is is the contents of its argument. It's passed what's known as a closure, its passed a block of code, which captures the necessary variables from its environment. There's two ways of writing an anonymous closure like this;

1. Specify the arguments within the bars, then the code. In that case, the closure borrows its environment, the values from its arguments, and any variables it references from the enclosing functions.
2. Define it by saying "move", followed by the arguments in bars, followed by the code. That takes ownership of its arguments, and of the values it needs from the environment. So it transfers ownership of the data to the closure.

Define a closure that *borrows* its environment:

```
|args| {
    code
}
```

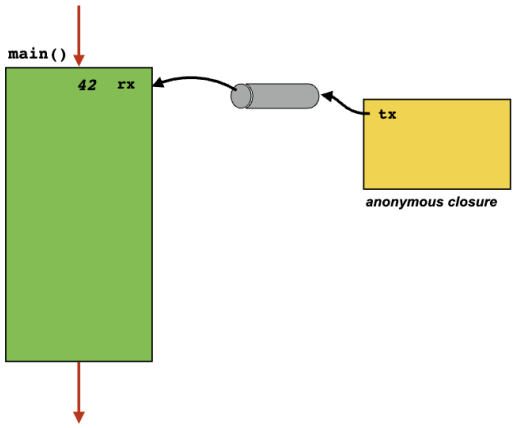
Define a closure that *takes ownership of* its environment:

```
move |args| {
    code
}
```

In the earlier case, we specify a move closure, so it taking ownership. There are two bars with no arguments between them, because this particular closure takes no explicit arguments. The body of this closure is simply the code within the braces, `let _ = tx.send(42);`. This references the value of tx, which was defined in main(), so its referenced a value from its environment.

Because its a move closure, it takes ownership of that, so the ownership of tx has been passed to the `thread::spawn()` call, this moves the ownership of tx to the other thread. At this point, we have two threads of execution running, the main thread, the new thread which has been spawned, which has taken ownership of tx, so its got access to the transmit descriptor of the channel.

The receive descriptor still remains with the main() function, it calls tx.send(), and transmits the number 42 down that channel, it assigns the return value to underscore that its not waiting for a response. This passes the value back to



the main thread, down the channel. Back in the main thread, you call recv() on the rx variable - the receive side of the channel - and you pattern match on the result. If you get a success value, you can look at the contexts of the value and process that. Otherwise, it can error, if something went wrong with the transmission. Its a much more explicit process than in Scala - we explicitly plumb together the sender and receiver by creating the channel.

3.5 Trade-Offs

The two approaches behave quite differently;

- **Scala+Akka** let weakly coupled processes communicate via asynchronous and dynamically typed messages. This is expressive, very flexible, and has a interesting error handling model using separate processes. It makes it relatively easy to upgrade the system by inserting actors into the system as needed. Checking happens at runtime, so you get probabilistic guarantees of correctness. The system can however fail at runtime.
- **Rust** uses statically typed message passing providing compile time checking that a process can respond to messages. It, however, is more explicit, requires more plumbing to connect the channels together, and it needs more explicit error handling so there may be more overhead there.

Essentially, this is the static vs dynamic typing debate again, but playing out in types of messages you're exchanging.

4 Race Conditions

A race condition can occur when the behaviour of a system depends on the relative timing of different actions, or when a shared value is modified without coordination. This introduces non deterministic behaviour and hard-to-debug problems. Its difficult to predict exact timing of program behaviour, and its difficult to predict effects of asynchronous, uncontrolled, modification to shared values.

4.1 Message races

In message passing systems, messages can be received from multiple senders. Runtime ensures receiver processes messages sequentially, in the order they are received, but order of receipt can vary due to system and network load, external events, etc.

- A **race condition** occurs when messages arrive in unpredictable order
- A **deadlock** occurs when a cycle forms, with actors waiting for messages from each other.

To avoid these, we need to structure the communication differently, so there are no loops of messages and there's no chance of things ending up mutually dependant. In message passing systems, you're building a network protocol, and the tools which we have to analyse network protocols, to detect deadlocks and race conditions, can be applied to the patterns of communication in actor based systems.

4.2 Data Races

In shared memory systems, data is conceptually moved from sender to receiver. In practice, a reference to the data is often copied, for performance reasons, and the underlying data remains in place.

A data race condition occurs if the data is modified after it is sent, and the modification is visible to the receiver via the reference. It s unpredictable if the receiver sees the old or new versions, depending on the timing of changes, scheduling, etc. There are two approaches to avoid this, one is immutable data, and the other is ownership tracking

4.2.1 Avoiding Data Races: Immutable Data

Race conditions cannot occur if data is not modified. Ensure any objects to be sent between threads is immutable. Languages like Erlang , which make extensive use of message passing, ensure this by making all variables immutable. You simple *cant* change the value of a variable in Erlang.

Alternatively, in Scala with Akka, which again is widely used for message passing systems, the language doesn't enforce immutability. It has tools to enable it, but it requires programmer discipline, the programmer needs to track what values can be passed between threads, and treat them as immutable.

This opens up the potential for race conditions if the message data is modified after the messages are sent, and its a source of bugs in these types of languages.

4.2.2 Avoiding Data Races: Ownership Transfer

Race conditions cannot occur is data isnt shared. If we make sure that the ownership of the object is actually being transferred, so the sender cant access it after it has been sent. If this is enforced by the runtime and complier, you are guaranteed that races cant occur.

This is a natural fit for Rust, because it tracks ownership, so can make this work so that once the value is sent into a channel, the function takes ownership, and the sender doesn't have access anymore.

On the receiving side, once the receiver calls recv() and returns ownership, the data is not accessible by the channel. The usual ownership rules in rust ensure the data is not accessible once its been sent. The original (sending) thread loses ownership.

It can't access and change the object once its been sent, so race conditions *cant* happen. That works well in rust because it has the ability to track ownership, but not so much in other languages. Rust is unusual in having ownership tracking.

4.3 Efficiency of Message Passing

Assuming immutable message or linear types, message passing has efficient implementation. In distributed systems, copy the message data, and in shared memory systems, its implemented by passing a pointer to the data to be transferred. Neither case needs to consider shared access to message data.

Garbage collected systems often allocate messages from a shared *exchange heap*. These are collected separately from per-process heaps, and are expensive to collect, since data in exchange heap is owned by multiple threads, so needs synchronisation. Per-process heaps can be collected independently and concurrently, which ensures good performance.

4.4 Summary

Message passing as an alternative concurrency mechanism is becoming increasingly popular, available in languages like Erlang and Scala, and works very well. Its also, of course, available in Rust, although not as popular or fashionable.

Its also possible in languages like Go, and with systems like ZeroMQ for C programs.

5 Robustness of Message Passing Systems

5.1 System Upgrade and Evolution

Message passing allows for easy system upgrade. Rather than pass messages directly to server, pass them via proxy. Proxy can load a new version of the server and redirect messages, without disrupting existing clients. Eventually, all clients are talking to the new server, and old server is garbage collected. Allows for gradual transparent system upgade without disrupting service.

Use of dynamic typing can make the upgrade easier. New components of the system can generate additional messages, which are ignored by old components. Supervisor hierarchy allows system to notice if components fail, and fallback to known good version. Backwards compatible extensions are simple to add in this manner.

5.2 Error Handling

The system is massively concurrent, errors in one part can be handled elsewhere, the error handling philosophy in Erlang is to;

- Let some other process do the error recovery
- If you can't do what you want to do, die
- Let it crash
- Do not program defensively

Be concerned with the overall system reliability, not the reliability of any one component.

5.2.1 Let it Crash

In a single-process system, that process must be responsible for handling errors. If the single process fails, then the entire application has failed.

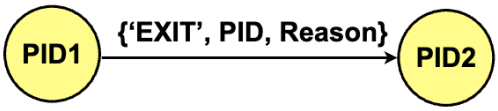
In a multiprocess system, each individual process is less precious, its just one of many. This changes the philosophy of error handling. A process which encounters a problem should not try to handle that problem, instead, fail loudly, cleanly, and quickly - "let it crash". Let another process cleanup and deal with the problem.

Processes become much simpler, since theyre not cluttered with error handling code.

5.3 Remote Error Handling

How do we handle errors in a concurrent distributed system? First isolate the problem, let an unaffected process be responsible for recovery. Dont trust the faulty component. This is analogous to hardware fault tolerance.

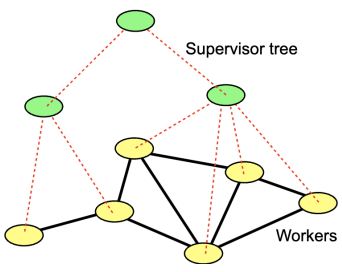
Processes are linked, runtime is set to trap errors and send a message to the linked process on failure. In the example below, PID2 has requested notification of failure of PID1, runtime sends an "EXIT" message on failure, to tell PID2 that PID1 failed, and why. Process PID2 then restarts PID1, and any other dependent process



5.4 Supervisor Hierarchies

We organise problems into tree-structured groups of processes, letting the higher nodes in the tree monitor and correct errors in the lower nodes.

Supervision trees are trees of supervisors, processes that monitor other processes in the system. Supervisors monitor workers - which perform tasks - or other supervisors. Work-



ers are instances of behaviours, processes whose operation is

characterised by callback functions, i.e. the Erlang equivalent of objects, for example server, event handler, finite stat machine, supervisor, application.

Abstract common behaviours into objects, workers are managed by supervisor processes that restart them in the case of failure, or otherwise handle error.

5.4.1 Robustness of Erlang Systems

Example: Ericsson AXD301 ATM switch. This was dimensioned to handle ~50,000 simultaneous flows with ~120 in setup or teardown phase at any one time, it processes traffic at 160Gbps (16 x 10Gbps links).

99.9999999% reliable in real-world deployments on 11 routers at major Ericsson customer (0.5 seconds downtime per year). Failures do occur, but are recovered by supervision hierarchy and distributed error recovery. Employs restart-and-recover semantics per connections so failures disrupt only one connection out of thousands.

Coroutines and Asynchronous Programming

1 Motivation

1.1 Blocking I/O

```
fn read_exact<T: Read>(input: &mut T, buf: &mut [u8]) -> Result<(), std::io::Error> {
    let mut cursor = 0;
    while cursor < buf.len() {
        cursor += input.read(&mut buf[cursor..])?;
    }
}
```

If we look at regular I/O code, you tend to have functions which look like the example above, the `read_exact()` function - what this function is doing, is reading an exact amount of data from some input stream, and storing it in a buffer.

It might be reading from a file, or it might be reading from a socket. In some cases, this function might have work in a loop, repeatedly reading from a socket for example, until its received the requested amount of data.

This works, its simple and straightforward. The problem with this type of code, though, is twofold. First, the IO operation can be very slow, the function can block for a long time while reading from the file descriptor, or the disk/network might just be slow. Calls to the read() function can take many millions of cycles, they can be very slow. The other issue is that calling the read() function blocks the thread, stopping other computations from running while the read() call is completing. If the thread is part of your user interface, it disrupts the user experience.

Ideally, we want to be able to overlap the IO and computation, we want to be able to run them concurrently. We want to be able to allow multiple concurrent IO operations.

1.2 Blocking I/O using Multiple Threads

This is traditionally solved by moving the blocking IO operations out of the main thread and put them into other threads to do the IO and report back. In Rust, you might do that in the way outlined below. Create a channel, spawn a thread to perform IO, that thread sends the result down the channel, then the rest of the program continues. This is relatively simple, it doesnt require any new language or runtime features, its the same blocking calls and multithreading functions we already have. It doesnt change the way we do IO, its not changing the fundamental model of either IO or threading, but we do have to move the IO code to a separate thread, which is some programming overhead.

```
fn main() {
    …
    let (tx, rx) = channel();
    thread::spawn(move|| {
      …perform I/O…
      tx.send(results);
    });
    …
    let data = rx.recv();
    …
}
```

It has the advantage that it can run in parallel if the system has multiple cores. Its safe, especially in rust, because the ownership rules prevent data races so its easy and straightforward to push the code into a separate thread.

The disadvantages on the other hand are that it adds complexity. Of course, creating a thread, while easy, is harder than simply not. Spawning a thread, partitioning the IO operations, isnt conceptually difficult in Rust, but complicated the code. Its a more complex programming model and it obfuscates the structure of the code.

It's relatively resource heavy, you have the overhead of context switching to the threads, and each thread has its own stack and its own memory overhead. This is of course not an enormous overhead, but still some, and much heavier than just running read() within the main thread.

Also, running threads in parallel if you have multicore hardware is a relatively limited benefit, because the thread spends most of its time blocked waiting for IO anyway. So, its a bit of a waste starting a new thread, just to call a read() function which then blocks 90% of its lifetime.

1.3 Non-Blocking I/O and Polling

Blocking IO using threads is problematic. Threads provide concurrent IO abstraction, but with high overhead. Multithreading can be inexpensive (e.g. Erlang), but has high overhead on general purpose operating systems - i.e. higher context switch overhead due to security reqs, higher memory overhead due to separate stack, and higher overhead due to greater isolation and preemptive scheduling. There are limited opportunities for parallelism with IO bound code, threads **can** be scheduled in parallel, but to little benefit unless CPU bound.

We'd like something more lightweight as an alternative, to somehow be able to multiplex IO operations in a single thread. We'd like to allow IO operations to complete asynchronously without having to have separate threads of control.

So, we want a mechanism which lets us start asynchronous IO, start an IO operation, and let it run in the background, and allow us to poll the kernel to see if it has finished yet, all running within a single application thread.

The idea would be to start the IO, then it continues in the background while the program continues performing other computations, and then, at some point, once the data is available, either theres a callback which is executed to provide the data, or the main thread can just poll it and say "Has it finished?" and pull the data in if so.

This is a reasonably common abstraction, for C programmers, this is the select() function in the Bekeley sockets API, for example. There are many versions of this, like epoll() for

linux and android, kqueue for FreeBSD, macos, and ios, or IO completion ports for windows.

This tends to get wrapped in libraries, like libevent, libev, or libuv - a common portable API for such system services. In Rust, the mio library provides an abstraction for this. These things provide the ability to trigger non-blocking operations, a way of saying "read (or write) asynchronously" from a file to a socket. They provide a poll() abstraction, so you can periodically check to see if it completed and retrieve the data. These are efficient, and only need a single thread, and they build on features of the operating system kernels. The problem with them, is that they require restructuring the code to avoid blocking.

```
FD_ZERO(&rfds);
FD_SET(fd1, &rfds);
FD_SET(fd2, &rfds);

tv.tv_sec  = 5;  // Timeout
tv.tv_usec = 0;

int rc = select(1, &rfds, &wfds, &efds, &tv);
if (rc < 0) {
  … handle error
} else if (rc == 0) {
  … handle timeout
} else {
  if (FD_ISSET(fd1, &rfds)) {
    … data available to read() on fd1
  }
  if (FD_ISSET(fd2, &rfds)) {
    … data available to read() on fd2
  }
  …
}
```

Above is an example of network code using sockets and the select() function in C. We see here the select() call, where you pass 3 parameters, the set of readable file descriptors, set of writeable file descriptors, the set of file descriptors that might deliver errors, and a timeout.

You bundle up file descriptors that may have outstanding asynchronous IO, fill in these parameters, call it, then poll each of these in turn to see if the `FD_ISSET` calls, to see which of those different file descriptors have data available to read or write.

This is a relatively low level API, which is reasonably well suited to C programming, but it does require quite a restructuring of the code. This is no longer as simple as calling read() in a loop. Conceptually, you have to restructure the program as an event loop, where you trigger the asynchronous IO, and every so often you poll it to see if its completes.

1.4 Alternatives to Non-blocking I/O

Now, these approaches have the advantage that they're very efficient. Because the asynchrony is handled by the OS kernel, a single thread can efficiently handle multiple sockets. All it does is trigger the asynchronous operations, and a kernel thread handles the rest.

Fundamentally, we have to choices.

- We can structure the code as a set of multiple threads, involving spawning threads and restructuring the code as a set of multiple threads which pass the data back once we've read the data
- We can restructure the code using the asynchronous IO primitives, involving turning itno an event loop with polling etc.

Both ways involve a fairly fundamental rewrite of the code to get these efficiency gains. **We want to get this efficiency, of non-blocking IO, in a much more usable manner. The idea is that coroutines and asynchronous code are one way of doing that.**

2 asynch and await

2.1 Coroutines and Asynchronous Code

```
fn read_exact<T: Read>(input: &mut T, buf: &mut [u8]) -> Result<(), std::io::Error> {
    let mut cursor = 0;
    while cursor < buf.len() {
        cursor += input.read(&mut buf[cursor..])?;
    }
}
```

This code calls blocking IO function (read()), and so stalls the execution of the program waiting for these calls to complete. The work-arounds for this, using multiple threads or asynchronous IO functions, such as select, require extensive restructuring. The goal of using coroutines with asynchronous code is to allow IO and computation to be performed concurrently on a single thread without restructuring the code. The hope is that this will avoid the overheads of multithreading while retaining the original code structure.

```
async fn read_exact<T: AsyncRead>(input: &mut T, buf: &mut [u8]) -> Result<(), std::io::Error> {
    let mut cursor = 0;
    while cursor < buf.len() {
        cursor += input.read(&mut buf[cursor..]).await?;
    }
}
```

Essentially, we hope to transform the code earlier into this asynchronous, non-blocking code, and provide the language runtime with the ability to execute those asynchronous functions concurrently with low overhead asynchronous IO operations.

2.2 Programming Model

The programming model we're considering structures the code as a set of concurrent coroutines that accept data from IO sources and yield in place of blocking. The coroutines execute concurrently, overlapping IO and computation, all within a single thread of execution, and so avoid the overhead of multithreading.

To understand this programming mode, we first must understand what a coroutine *is*. Unlike a normal function, which is called, executes for a while, and returns a result, a coroutine has the ability to pause its execution. Rather than return a single value or even a list of values, it lazily generates a sequence of values.

```
def countdown(n):
  while n > 0:
    yield n
    n -= 1

>>> for i in countdown(5):
...   print i,
...
5 4 3 2 1
>>>
```

This is an example, in python. In this case, the countdown function is a coroutine that yields a sequence of integers, counting down to zero from the value given. for example, we can see that, calling countdown(5) yields the values 5,4,3,2,1, and that these can be processed by a for loop.

Importantly, the countdown function is not retuning a single value, comprising a list of numbers counting down, rather, calling countdown returns a generator object. A generator object alone doesnt do anything, but the generator object implements a next() method and the for loop in Python takes a generator and repeatedly calls next().

Each time next() is called, the function is executed until the yeild statement, or until the function ends, in which case

it returns None and the generator is completed. Essentially the function is turned into a heap allocated generator object that maintains state, executes lazily in response to calls to next, and repeatedly yields different values.

Coroutines in python do nothing until the next() function is called on the generator object representing the coroutine. Normally, this happens automatically, as part of the operation of a loop for example, but we can also call it manually, as we can see here.

```
def grep(pattern):
  print(F"Looking for {pattern}")
    while True:
      line = (yield)
      if pattern in line:
        print line

>>> g = grep("python")
>>> g.next()
Looking for python
>>> g.send("Yeah, but no, but yeah, but no")
>>> g.send("A series of tubes")
>>> g.send("python generators rock!")
python generators rock!
>>>
```

the grep() function is a coroutine, and the call to `g=grep("python")` instantiates the generator object. Instantiating the generator doesnt cause it to run.

The print() call at the start of the grep() function doesnt execute until we call the next() method, for example, forcing the coroutine to run until it yields. In this case, the function yields to consume a value, so we call send() rather than next(), and pass in a value. We do this repeatedly, passing in a different value each time, and each time causing a single iteration of the while loop in the grep() function to execute.

So, a coroutine is a function that executes concurrently to the rest of the ode. It's event driven, it only executes when the runtime calls its next() or send() method, which causes its execution to resume until it next yields, at which point control passes back to the runtime.

Its possible to have many coroutines and have the runtime loop calling their next() methods, this will cause the different coroutines to execute concurrently, each one executing for a while until it yields to the next. This is an approach sometimes known as cooperative multitasking. The system context switches each time a coroutine yields the processor, and if it doesnt yield, it keeps running.

This gives us a basis for efficient IO handling within a single thread. We structure the code to execute within a thread as a set of coroutines that trigger an asynchronous IO operation, and yield rather than blocking on IO. We label the function as being async, this tells the language runtime that those functions are coroutines that call asynchronous IO operations. IO operations that would normally block are labelled in the code with an await tag.

This causes the coroutine to trigger the async version of that IO operation, yield, pass control over to another coroutine while the IO is performed. This provides concurrent IO, without parallelism. The coroutines operate concurrently within a single thread.

Calls to await tell the kernel to start an async IO operation and yield the file descriptor representing that operation to the runtime, which operates in a loop. It repeatedly calls select(), and adds the coroutines that yielded any readable or writeable file descriptors to the end of a run queue, then resumes the coroutine at the head of the run queue, calling its next() or send() method as appropriate. When that coroutine yields and returns a file descriptor, its moved to the list of blocked tasks, and its file descriptor is added to the set to be polled in future. The loop continues, calling select() again.

2.3 `async` Functions

An async function is one that can act as a coroutine, that performs asynchronous IO operations. Below is an example in Python.

```
#!/usr/bin/env python3

import asyncio

async def fetch_html(url: str, session: ClientSession) -> str:
    resp = await session.request(method="GET", url=url)
    html = await resp.text()
    return html
…
```

The async functions are executed asynchronously by the runtime in response to IO events, and the program is written as a set of async functions. The main() function calls into the runtime giving it the top-level async function to run. This starts the asynchronous runtime, starts polling for IO and runs until that async function completes.

2.4 `await` Future Results

await statements cause the function to yield control to the runtime while an asynchronous IO operation is performed. It puts the coroutine into a queue to be woken at some later time, when the IO operation has completed.

If another coroutine is ready to execute, then the runtime schedules the yielding function to wake up once the IO completes, and control passes to that other coroutine. Otherwise, runtime blocks until either this, or some other IO operation becomes ready, then passes control back to the corresponding async function.

2.4.1 `async` and `await` Programming Model

The resulting asynchronous code follows the structure of the blocking code, like we saw earlier.

```
fn read_exact<T: Read>(input: &mut T, buf: &mut [u8]) -> Result<(), std::io::Error> {
    let mut cursor = 0;
    while cursor < buf.len() {
        cursor += input.read(&mut buf[cursor..])?;
    }
}
```

```
async fn read_exact<T: AsyncRead>(input: &mut T, buf: &mut [u8]) -> Result<(), std::io::Error> {
    let mut cursor = 0;
    while cursor < buf.len() {
        cursor += input.read(&mut buf[cursor..]).await?;
    }
}
```

If we look at the async version of the `read_exact` Rust function, for example, we see that the only differences are the async and await annotations, and that the input is declared to be something that implements the AsyncRead trait.

The code structure remains unchanged, aside from the call to main that wraps the async functions into a call to start the runtime. The compile and runtime work together to generate code that efficiently executes the asynchronous IO operations..

2.5 Runtime Support

In python, as seen earlier, the coroutine was instantiated as a generator object with a next() method that allowed it to run and yield the next value.

Rust does something similar, the async functions are compiled into instances of structs that maintain the function state, and that implement a trait known as Future. The Future trait has a member type that describes the return value, and defines a poll() function that runs the function until it yields an instance of an enum. Thats either Ready, with the yielded value, or Pending, to indicate that the async function is waiting for IO.

3 Design Patterns for Asynchronous Code

3.1 Compose `Future` Values

async functions should be small, and in a limited scope. An async function should perform a single, well defined task. For example, it should read and parse a file or read, parse, process, and respond to a network request.

If functions are structured in this way, they tend to be fairly straightforward, written in a fairly natural style, and compose pretty straightforwardly. The Rust async libraries provide some combinator functions that can help compose futures, to help combine future values and produce a new value. There are functions like `read_exact()` which allow it to read an exact number of bytes, or like select() which allow it to respond to different futures which are operating concurrently. There are functions such as `for_each()` and `and_then()` which can ease the composition of the asynchronous functions.

3.2 Avoiding Blocking Operations

When writing asynchronous code theres two fundamental constraints to be aware of, first, due to the nature of asynchronous code, is that it multiplexes multiple IO operations onto a single thread, and the runtime has to provide async-aware versions of all these IO operations. It has to provide asynchronous reads and writes to files, to the network, tcp, udp, unix sockets, etc. In all cases, it has to provide a non-blocking version of the IO operation that returns a Future that can interact with the runtime.

Rather than natively calling the blocking function provided by the operating system, it has to wrap the underlying OS provided async io operations into Futures. Importantly, it doesn't interact well with blocking IO, if you call the synchronous version and it blocks, it will block the entire runtime, because the runtime is operating within the context of a single thread and the underlying system doesnt know about Futures.

The programmer has to have the discipline to avoid calling the blocking functions of the code, otherwise the entire asynchronous runtime grinds to a halt while that blocking operation completes. To some extent, this runs the risk of fragmenting the ecosystem, it means that libraries which are supposed to be used in an asynchronous context have to be written to use async functions, have to call await(), and have to be written to use async versions of the IO libraries.

This is a potential source of bugs, because the Rust compiler (and language) cant catch this sort of behaviour, the programmer is required to have the discipline to avoid the blocking IO operations.

3.3 Avoid Long-running Computations

Similarly, the programmer has to avoid long running computations. Control passing between Futures, different async functions, is explicit. Control passing happens when you call await, to wait for an IO operation. At that point, the next runnable Future or async function is scheduled.

In the same way that calling blocking functions is problematic, if instead you dont call await and perform some long running computation, the runtime wont know, wont be able to switch away from that computation, and starve other tasks from running. If you have a long running computation, spawn a separate thread, and explicitly pass messages to and from that thread to avoid this.

Again, the language and the runtime doesnt help with this, the programmer needs to be aware and avoid long running computations.

3.4 When to use Asynchronous IO?

Well, the use of async and await lets us structure the code in a way that allows us to efficiently multiplex large numbers of IO operations on a single thread. This can give a very natural programming model when you're performing operations which are very heavily IO bound.

This lets us structure code which performs asynchronous non blocking IO in a way that looks very similar to code that uses blocking IO, that can efficiently multiplex IO operations onto a single thread, and efficiently allow them to run concurrently, without the overhead of starting up multiple threads.

It can be problematic, as we saw earlier, if there are blocking operations, they can lock up the entire runtime and not just one task. It means all libraries that use blocking calls need to be updated to use these asynchronous IO operations. This either means everything has to use asynchronous IO, or people need to build 2 versions of all libraries, synchronous and asynchronous.

Its also problematic when it comes to long running computations, as before, these can starve other tasks, because runtime only switches away when you call asynchronous operations (like await). So, if you try to mix long running, compute heavy, functions with asynchronous IO functions and IO bound tasks, it gets problematic and starve IO tasks.

3.5 Performance of Asynchronous IO

Do we really need asynchronous IO, do we really need this performance? Short answer, probably not.

Threads *are* more expensive than async functions and tasks, but threads are *not that* expensive. A properly configured modern machine can run many thousands of threads without difficulty. If you're doing something very unusual, its likely you can spawn a thread, or use a preconfigured thread pool, and perform blocking IO, and just communicate using channels. The performance will be just fine.

Even if this means spawning up thousands of threads, modern servers can run thousands, or 10s of thousands of simultaneous threads. Threading is not that expensive these days. Asynchronous IO can give a performance benefit, but its usually not that great. Choose asynchronous programming for the style, not for the performance, unless it **absolutely will significantly improve performance.**

Security Considerations

1 Memory Safety

1.1 Memory Safety in Programming Languages

A memory **safe** language is one that ensures that only the memory that can be accessed is that which is owned by the program, and that all access to that memory is done in a manner which is consistent with the types of data.

So, the program can only access memory through its global or local variables, or through explicit references from them. It **cant** access memory which isnt in any of its variables, or isnt referenced from any of those variables.

It can only access heap memory if its got an explicit reference to that memory, and it has to access memory in a consistent way with the memory type. If the value in memory is an integer, it cant be read back as a floating point value unless theres an explicit conversion. It cant treat the data inconsistently for different types.

A memory **unsafe** is a language which doesnt prevent accesses which break typing rules, it allows access to memory which the program doesnt explicitly own, or allows access to memory in a type unsafe way. There are many memory safe langauges; java, scala, rust, go, python, haskell, and many many more. There are far fewer unsafe languages, c, c++, and objective-c.

In memory unsafe language, accesses that break type rules are not simply declared to be undefined. They tell the programmer that they shouldnt do something, not that they **cant**. The program will continue if the undefined accesses are performed, its just unclear what the behaviour will be.

1.2 Undefined Behaviour

So, what does this mean in practice? When a language defines certain types of behaviour to be undefined, what happens? what types of behaviour are undefined?

One example is a type unsafe allocation, as below:

1.2.1 Type Unsafe Allocation

```
double *d = malloc(sizeof(float));
```

This C code allocates memory to hold the size of floating point value, and we reference it as if it was a double precision float. In this case, what we see is the call to malloc() does not check if the amount of memory allocated corresponds to the type by which its being accessed.

On a typical 64-bit machine for example, this could be allocating memory for a 32 bit floating point type, then accessing it as a 64 bit type, overflowing the space. A memory safe language would require that the size of the allocation matches the size of the value stored.

The operation ought to be, allocate memory for an object of type T, return a reference to type T, which can only be assigned a variable which holds a reference of that type. What we have in C, though, is a function that allocates a certain amount of memory and returns an untyped reference. Thats where the lack of safety comes in, theres no typing of the allocation.

C doesnt know whats going to be stoed in a `malloc()`ed value, and so it cant check that its big enough to hold the values that are stored there.

1.2.2 Use Before Allocation

Another undefined behaviour, we could be doing a use-before-allocation. In the case below, the code reads from a file descriptor, and it has a char* pointer for a buffer into which it is reading, and it calls the recv() call to read into that buffer, but it never allocated memory for the buffer

```
char *buffer;

int rc = recv(fd, buffer, BUFLEN, 0);
if (rc < 0) {
    perror("Cannot receive");
} else {
    // Handle data
}
```

It passes a pointer to the buffer to the recv() function, without allocating memory, or the programmer has assumed the recv() function will do the allocation for them. The result will be that it writes to an arbitrary location, depending on what value happens to be in this uninitialised pointer.

A memory safe language would require that references are initialised, and require that they refer to valid data. It shouldnt be possible to have an uninitialised reference, a reference which points to nothing. Languages like rust enforce this, a reference always points to a valid value.

Good C compilers will *warn* you about this behaviour, if you turn on the warnings, but the C language doesn't require that warning and allows you to access arbitrary pointers, whether or not they point to valid memory.

1.2.3 Use After Explicit `free`

Similarly, C and C++ allows you to use memory after youve freed it.

```
#include <stdlib.h>
#include <stdio.h>

int main() {
  char *x = malloc(14);
  sprintf(x, "Hello, world!");
  free(x);
  printf("%s\n", x);
}
```

In this example, it allocates 14 bytes, writes the string "Hello, world!", frees the memory, and prints the value. This will work in most cases, because nothing else is using that memory and this is a very simple example, but with a much larger program with concurrent accesses going on, that memory could have been reused by another thread, and could print anything. Its result is entirely **undefined**

1.2.4 Use After Implicit Free

We can also access memory implicitly after its ben freed. The example below is a function which returns a reference to stack allocated memory, and once the function returns, the stack frame is destroyed, the reference now points to unallocated memory. Again, you have an arbitrary reference, its not clear whats in the memory, which is being referred to by that reference.

```
int *foo() {
  int n = 42;
  return &n;
}
```

Again, automatically memory managed languages eliminate this type of bug, they either hold onto the value, or it just wont compile if you return a reference which goes out of scope.

1.2.5 Use of Memory as the Wrong Type

Memory unsafe languages can also allow memory to be accessed as the wrong type. This is a little more subtle, looking at the code here, we allocate space for a buffer and read data into it. Assuming it successfully reads into that buffer, it then cases the buffer to be a pointer of a different type.

This is quite a common idiom in C code, you see casts from char* pointers representing buffers, to a certain struct for ex-

```
char *buffer = malloc(BUFLEN);

int rc = recv(fd, buffer, BUFLEN, 0);
if (rc < 0) {
    …
} else {
    struct pkt *p = (struct pkt *) buffer;
    if (p->version != 1) {
        …
    }
    …
}
```

ample, representing a network packet format, or representing the header of a file format.

This is efficient, theres no memory copies, but its unsafe since it makes assumptions of struct layout in memory, and that the size of the block being cast matches the size of the struct. This makes it unsafe, these assumptions have a habit of becoming untrue over time

Because the type system is being overridden, the programmer is essentially saying *"Trust me, i know what im doing, pretend this set of bytes is a value of this other type"*. Such changes in layout or behaviour tend not to get detected, and the program fails silently when used on a different compiler or on a different machine.

1.2.6 Use of string functions on non-string values

This is a common security vulnerability in C programs, where the C language assumes strings are NUL terminated, theres a terminating 0 value at the end of a string, but the recv() function, reading from a file or the network, doesnt add that terminating zero.

We have to explicitly add the terminating zero, and if you forget, and call a string function on that, the string function keeps reading until it finds the first zero in memory, and who knows what it reads in memory.

Best case is that it just leaks some contents of memory, maybe some sensitive data, worse case, it runs into some unallocated memory and crashes the program.

Memory safe languages apply string bounds checks to make sure the types match and that you cant access over the end of the string. They check that they either succeeded, or if theyre going to fail, they fail with a runtime exception which closes the program cleanly without undefined behaviour.

1.2.7 Heap Allocation Overflow

```
#define BUFSIZE 256
int main(int argc, char *argv[]) {
    char *buf;
    buf = malloc(sizeof(char) * BUFSIZE);
    strcpy(buf, argv[1]);
}
```

In a memory unsafe language, you can allocate, in this case, memory for 256 bytes of memory, but copy in an arbitrary sized buffer. Theres no checks to make sure the value thats being copied in fits within the 256 bytes, and it can overwrite the end of the array, corrupt whatever happens to be in memory next.

Memory safe languages on the other hand apply bounds checks. Doing the same thing in rust or Java, the program will fail with an exception, or a panic in rust, which closes the program cleanly. It doesnt fail in an undefined way, it has defined failure semantics. It doesnt leak information, or corrupt the state.

1.2.8 Array Bounds Overflow

You can do the same thing with arrays. The memory doesnt have to be allocated on the heap for you to ba able to overflow it in C or C++. Again, theres no bounds checks, whereas in safe languages there are.

1.2.9 Arbitrary Pointer Arithmetic

```
int buf[INTBUFSIZE];
int *buf_ptr = buf;

while (havedata() && (buf_ptr < (buf + sizeof(buf)))) {
  *buf_ptr++ = parseint(getdata());
}
```

This C program is looking at the size of the bugger, so the check for `buf_ptr` is less than `buf+sizeof(buf)` looks like its checking the pointer is within range, it looks like its manually implemented a bounds check.

The bug here is that the `buf_ptr` is a pointer and int and `sizeof()` returns a size in bytes. What this program is doing when it increments `buf_ptr`, is incrementing by int sized chunks, rather than byte sized chunks, so the sizes dont match up.

This ends up with a buffer overflow because its doing arithmetic on pointers, but its not doing it correctly. The language shouldnt allow accesses where the types dont match up

1.2.10 Use of Uninitialised Memory

Memory unsafe languages allow uninitialised memory to be accessed.

```
static char *
read_headers(int fd)
{
  char     buf[BUFLEN];
  char    *headers   = malloc(1);
  size_t   headerLen = 0;
  ssize_t  rlen;

  while (strstr(headers, "\r\n\r\n") == NULL) {
    rlen = recv(fd, buf, BUFLEN, 0);
    if (rlen ==  0) {
      // Connection closed by client
      free(headers);
      return NULL;
    } else if (rlen < 0)  {
      free(headers);
      perror("Cannot read HTTP request");
      return NULL;
    } else {
      headerLen += (size_t) rlen;
      headers = realloc(headers, headerLen + 1);
      strncat(headers, buf, (size_t) rlen);
    }
  }
  return headers;
}
```

In this example, we allocate one byte to store the headers. This could be reading HTTP headers, and growing a buffer, reading the headers in byte-by-byte and checking after each read() to see whether its received the complete set of headers, if they end with a double carriage return newline.

The problem is the initial allocation allocates space for one byte but doesnt fill in that memory. It doesnt have an end-of-string marker there to say that this is an empty string, so the first strstr() call runs off the end of the uninitialised memory and keeps searching for the zero.

A memory safe language would require that all the memory is initialised, and it would require that any allocation has known contents. Unsafe languages dont do that, they let you perform allocations on any arbitrary unpredictable value.

1.2.11 Use of Memory via `null` Pointer

We have languages which allow you to access memory via dangling pointers, via null pointers. Which can access values where the pointer points to some unknown value. You might get a null pointer returned from lookup() and try to dereference that, and you will fail. You might have a pointer to something that doesnt exist anymore, and fail.

1.2.12 More and More...

This is a long, yet non-exhaustive list. There are many more types of memory unsafe behaviour that are possible in C or C++

1.3 Impact of Memory Unsafe Languages

The moral of the story is that, **the lack of memory safety breaks the abstractions**. The program starts doing unpredictable things.

If you're lucky, the program crashes early and visibly, so you can tell theres a memory safety bug and you can fix it. If you're unlucky, the program accesses memory which it doesnt own, or in a way the types dont match, or corrupts some other value. Maybe that memory isnt being used, and the corruption has no effect, but maybe it it, and maybe its storing some other value in the program, and its just been corrupted, with no way of knowing when or where that happened.

1.3.1 Impact of Memory Unsafe Languages - Security

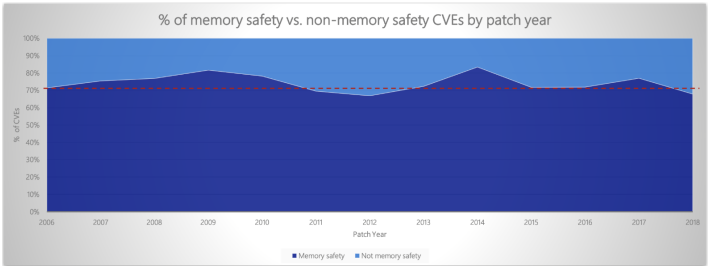
Vulnerabilities By Type

Year	# of Vulnerabilities	DoS	Code Execution	Overflow	Memory Corruption	Sql Injection	XSS	Directory Traversal	Http Response Splitting	Bypass something	Gain Information	Gain Privileges	CSRF	File Inclusion	# of exploits
1999	894	177	112	172			2	7		25	16	103			2
2000	1020	257	208	206		2	4	20		48	19	139			
2001	1677	403	403	297		7	34	123		83	36	220		2	2
2002	2156	498	553	435	2	41	200	103		127	74	199	2	14	1
2003	1527	381	477	371	2	49	129	60	1	62	69	144		16	5
2004	2451	580	614	410	3	148	291	111	12	145	96	134	5	38	5
2005	4935	838	1627	657	21	604	786	202	15	289	261	221	11	100	14
2006	6610	893	2719	663	91	967	1302	322	8	267	271	184	18	849	30
2007	6520	1101	2601	954	95	706	884	339	14	267	324	242	69	700	44
2008	5632	894	2310	699	128	1101	807	363	7	288	270	188	83	170	74
2009	5736	1035	2185	700	188	963	851	322	9	337	302	223	115	138	738
2010	4652	1102	1714	680	342	520	605	275	8	234	282	238	86	73	1493
2011	4155	1221	1334	770	351	294	467	108	7	197	409	206	58	17	557
2012	5297	1425	1459	843	423	243	758	122	13	343	389	250	166	14	624
2013	5191	1455	1186	859	366	156	650	110	7	352	511	274	123	1	205
2014	7946	1598	1574	850	420	305	1105	204	12	457	2104	239	264	2	401
2015	6484	1791	1826	1079	749	218	778	150	12	577	748	367	248	5	127
2016	6447	2028	1494	1325	717	94	497	99	15	444	843	600	87	7	1
2017	14714	3154	3004	2805	745	503	1516	274	11	629	1706	459	327	18	6
2018	16555	1852	3035	2451	400	516	2001	509	11	709	1374	247	461	31	4
2019	4	2				1									
Total	110603	22685	30435	17226	5043	7438	13667	3823	162	5880	10104	4877	2123	2195	4333
% Of All		20.5	27.5	15.6	4.6	6.7	12.4	3.5	0.1	5.3	9.1	4.4	1.9	2.0	

This is a list of all the Common Vulnerabilities and Exposures(CVEs), the common vulnerability database, its a list of all security vulnerabilities in software reported over a 20 year period. The columns highlighted show the vulnerabilities which are due to buffer overflows, memory corruption, or treating data as executable code.

Almost half of all vulnerabilities are memory safety. They are a problem due to lack of memory safety in the language, which should be caught by a modern type system. The compiler should be able to detect these problems.

By modelling the problem domain in the types, even if theyre not directly related to memory unsafety, the compiler can help check for these problems. Things like SQL injection, parsing bugs, the type system can check for consistency, and can help prevent these.



Here is another graph, this time from microsoft looking at the vulnerabilities in Microsoft software, over a 10 year period. In this case, about 70% of their security updates fix bugs relating to unsafe memory usage. This is a little higher than the previous graph, perhaps because of the type of software microsoft develops - a lot of systems code, a lot of applications code - whereas the previous graph was looking at all software.

Whats interesting is that we're not getting any better at this. We have known that memory safety bugs are a problem for 20 years, and we're not getting better at writing secure code in unsafe languages. The number of memory safety bugs isnt going down. Microsoft doesnt not know that this is a problem, its just a hard problem to fix.

1.4 Mitigations for Memory Unsafe Languages

If you *have* to write in an unsafe language, use the modern tooling. If you're compiling C code, at the very minimum turn on all the warnings you can find. The minimum set of flags for compiling C code, if you're using clang, for example, should be `-W -Wall -Werror`.

Youd think that this turns on all warnings and makes them errors, but what thsi actually does is turn on warnings for a particular version of GCC, and make them errors. Find the warnings that exist in addition to `-Wall` and turn them on.

Fix all the warnings. When the compiler says theres a warning, treat is as an error. When you're debugging your code, use static analysis tools like valgrind, or more modern address and memory sanitisers, undefined behaviour sanitisers, thread sanitisers, all built into clang.

Dont use C++

Generally speaking, it can be a nice, somewhat safe-r language, but its too complicated. Features keep getting added, nothing gets removed. If you can understand it, it can be good, but very few people *properly* understand it. Some bad choices were made in early versions, it had some inappropriate defaults, and you need a lot of knowledge to use C++ correctly, and know what is safe to use.

1.5 Rewrite Code into a Memory Safe Language

Ultimately, whether its C or C++, if you have an unsafe code base, you should be looking to rewrite it, or at least its critical sections into a memory safe language, for example Rust.

One of the nice things about Rust, you can compile it in a way that the functions are compatible with C code, you can rewrite a program function at a time, and link in the Rust code, gradually converting it over a long period of time, testing as you go.

The same can be done with Objective C, gradually rewriting to Swift to get the safety benefits, for anything Apple related.

This can be difficult, requiring languages to have compatible runtime models so you can link the code together, and to write a program in multiple languages, it requires that you know both. This gradual rewrite however is still much better than a fresh rewrite from scratch in most cases, you can keep testing as you go, rather than having to throw everything away and start over.

2 Parsing and Language Security

2.1 Remotely Exploitable Vulnerabilities

Theres a large number of perhaps somewhat subtle vulnerabilities and bugs that can occur, which lead to unsafe behaviour, these bugs can lead to exploitable flaws in the code.

Its often easy to look at these problems and think, "How could anyone write code like that?" and the problem is that a lot of these remotely exploitable vulnerabilities fall under this category. The example below, is a real security vulnerability from the Apple TLS code This clearly is a mis-

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
  goto fail;
  goto fail;
... other checks ...
fail:
  ... buffer frees (cleanups) ...
  return err;
```

merge or misedit, such that the `goto fail` line was duplicated. Because indentation is not significant in C, what this does is always go to the failure case, skips a bunch of checks. One particular check skipped led to code that was remotely exploitable, because the checks that validated input were skipped, which then led to memory unsafety bugs down the line.

Of course, bugs like this *can* happen, people make mistakes, programmers are expected to be perfect and never make mistakes, but of course its inevitable that they do.

2.2 Parsing Untrusted Input

One area it is important to proactively address, is parsing untrusted input, whether that is input from the network to the rest of the program, or the input from a file to a parser, or to/from the rest of the program.

If we focus on networked systems in very high level terms, you get some untrusted protocol data from the network. You need to parse it, take the protocol message, parse them into some internal data structure. The protocol code then operates on those data structures, and serialises the results for transmission back out the network in response.

The input parser is **critical** to the security of any networked system, it's taking untrusted input from the network, and trying to generate validated strongly-typed data structures that can be processed by the rest of the code.

2.3 The Robustness Principle (Postel's Law)

Traditionally, Postel's law describes the guidance around writing networked systems. It discusses interoperability and the robustness of network protocols, and that you should be liberal in what you accept, conservative in what you send.

When generating data, make sure it conforms to the protocol, but when parsing input data, the Postels law suggests you should be forgiving of errors and of precise details in the format, providing you can make sense of what was sent.

Postels law might not be so appropriate for todays internet. We should more clearly specify what is legal, what the code will and wont accept, and be strict about this. We should be strict about how and when our code fails.

2.4 Define Legal Inputs and Failure Responses

The approach we should be taking to clearly specify what is and what isnt legal. Input parsers are designed to take untrusted data, validated it, and generate strongly typed and safe data which we can then use. In other words, they take something untrusted, and turn it into a form which can be trusted and is safe and usable.

The problem with a lot of traditional parsers is that it hasnt been clear what the grammar is, and what the failure conditions are. This is the problem of Postels law, it encourages each application to extend the grammar in arbitrary ways to be a little more flexible or forgiving. These inconsistencies in behaviour lead to gaps where particular implementations can be exploited.

Ideally, we should define what is and isnt legal in the protocol specification, so every implementation can use the same grammar, and accept the same inputs. Theres no ambiguity in what is and isnt acceptable. This should be done in as restrictive a manner as its feasible to do.

The more expressive power we have, the more likely we are to have risks, the more scope for vulnerabilities we have, and the more scope for ambiguity. We should specify what happens if data doesnt match the grammar, and what causes a failure, and how these are handled.

2.5 Generate The Parser

Once we have the rules set out, we then should generate the parsing code using the simplest possible parsing technique. If we've got a regular input language, use a regular expression. If we've got a context-free grammar, use a context-free parser.

If need be we should use more sophisticated techniques, but should use those of minimal computational power to ominimise risks. We should also generate strongly types data structures with explicit types to identify the different types of data, so we know what has been parsed.

Manually written parsing code is very difficult to reason about, it tends to perform low level bit manipulation and detailed string parsing, etc., which is hard to get right and to structure.

We can write down a grammar, formally verify it, formally generate a parser automatically for that grammar, and we'll know that its consistent and correct.

One thing we can do to improve the security of systems code is to use existing, well tested parser libraries, and use them for the input of out networked applications.

For Rust, this can be something like nom or combine - well tested, well structured parser combinator libraries which let you specify the grammar and parsing rules. For C or C++, use the Hammer parser.

We specify the types into which the data is parsed, describe the parser using a formal language, and generate the parser. Ensure that parsing is performed first, and either succeeds or fails entirely.

Input parsers are difficult to get right if written in an adhoc way, especially in a low-level memory unsafe language, but we have tools - parser generator tools - lets make use of them, even if we're still using C or C++, we can make these safer with these tools.

2.6 Define Parsable Protocols

When we design network protocols, we should be thinking about ease of parsing. We should think about minimising the amount of state and amount of look-ahead required to parse the protocol data.

We should be designing network protocols such that they are predictable, can be parsed using regular expressions or context free grammars, rather than needing complex, state-dependant parsing, because this allows us to use simpler

parser generators, and simpler parsing code, reducing the chances of bugs.

The benefic of saving a few more bits by having a more complex format goes down over time because the network gets faster. but the security vulnerabilities remain. We should be aiming for simplicity and ease of parsing, and we should be using the best parsing tools to get rid of these vulnerabilities.

3 Modern Type Systems and Security

3.1 Causes of Security Vulnerabilities

Security vulnerabilities tend to be caused by the attacker persuading the program to do something the programmer doesnt expect, this could be persuading the program to write past the end of a buffer, treat input as executable, confuse a permission check, etc., all behaviours which the attacker can potentially force the program to do but the programmer isnt expecting.

Essentially, the goal is to violate assumptions in the code, to confuse the program into doing something its not expected to do. A consequence of that, is that anything we can do to make these assumptions explicit, and check these assumptions , helps avoid security vulnerabilities.

This brings us back to **strong typing**, strong types make the assumptions explicit. They let the compiler check that what we're doing in the program makes sense, for this reason, strong typing might help reduce security vulnerabilities in code.

By using explicit types rather than generic types, and by defining conversion functions, and using the type system to add semantic tags to the data to help us understand what is means, we can be clear how the data is to be processed. We can be clear what the data, what has been done to it, and what can be done to it, and the compiler can help us check illegal operations are not performed.

3.2 Prefer Explicit Types

When dealing with data, you should prefer explicit types, because vulnerabilities can come from inconsistent data processing. Its easy to accidentally take data which arrives off the network, and pass it to a function unsanitised, and then execute that data. A popular example of this is passing what is expected to be a name as SQL data, but since the name has a valid SQL command embedded, it gets executed and corrupts the database.

This happens because we're conflating different types of data, in this case a name and SQL commands. It should be possible to write code that uses one type of value to represent untrusted input data, and another to represent SQL commands, and to have a specific conversion function, to validates and converts the data formats.

This stops us accidentally passing untrusted input as a SQL string, but this only works if we use different types for different things, and not using strings everywhere.

3.3 Convert Data Carefully

When we convert data, we need explicit type conversion functions, and use these to enforce security boundaries. Untrusted data is one thing, it should be handled with care, we can process and escape it, and convert it into a safer, trusted form.

We can write explicit conversion functions, to convert between the type representing the input data, and the type representing the data to be processed. This can ensure that only legal conversions occur, and in order to be converted, data has to be validated.

3.4 Phantom Types to Add Semantic Tags

We should think about ways of labelling data, and whether data has been checked or not, and tagging it with the assumptions made about its processing. In rust, its entirely possible to add tags to types. For example, we can have a strut with no fields, the result has no size, it has no content, but it can be used as a type parameter to add a semantic tag to the data.

The point is to label the data, to specify what has been done to it, to make the assumptions explicit. The compiler can help us by checking that the data we're passing around has been correctly tagged as having had the particular sanitising operations performed. It doesn't stop the bugs, just limits their scope.

3.5 No Silver Bullet

To be clear, all of these methods, memory safe languages, strong typing, etc. are not going to eliminate security vulnerabilities entirely, what they do, is eliminate certain classes of vulnerability. When we write networked, security critical code in C, we have to be continually watching for buffer overflows for example.

If we write our code in rust, buffer overflows cant happen, and we've eliminated this class of vulnerability. This is the benefit of memory safe and strongly typed languages, they get rid of some types of security vulnerabilities, and let us focus on the rest.

We're never going to get rid of security vulnerabilities entirely, theres no magic silver bullet here.

3.6 Liability and Ethics

As programmers, we need to think about how we write code, if we look at the ACM code of ethics, one of the key things is to avoid harm. For civil engineers, this might mean building bridges that dont fall over, for software engineers, it means making sure out code doesnt cause harm.

Security vulnerabilities and software failures do routinely cause harm, so we need to consider the best practices to avoid this.