Software Security

Language-based Security: *'Safe'* programming languages

Erik Poll

Radboud Universiteit Nijmegen

TRU/e Master in Cyber Security

Language-based security

Security features & guarantees provided by the programming language

- **safety guarantees**, incl. **memory-safety, type-safety, thread-safety**

  NB there are many flavours & levels of 'safety' here. Eg different type systems have different levels of expressivity, and hence of type-safety.

- various forms of **access control**
  - visibility/access restrictions with **`public, private`**
  - **sandboxing** mechanism inside programming language

These features can be interdependent, eg with type-safety relying on memory safety, sandboxing relying on memory & type-safety, ...

This week: safety. See course lecture notes, chapters 2 & 3

Other ways the programming language can help

A programming language can also help security by

- offering good APIs/libraries, eg.
  - APIs with parametrised queries/prepared statements for SQL
  - more secure string libraries for C
- incorporating support for 'external' languages,
  - eg support for SQL and HTML in Wyvern
- offering convenient language features,
  - esp. exceptions, to simplify handling error conditions
- making assurance of the security easier, by
  - being able to understand code in a modular way
  - only having to review the public interface, in a code review

  These properties *require* some form of safety

(Aside: safety vs security)

Common source of confusion!

- safety: protecting a system from *accidental* failures
  (esp. protecting humans from harm)
- security: protecting a system from *active attackers*

Precise border hard to pin down, but what is good for safety is also good for security, so often the distinction is not so relevant.

In Dutch, the confusion is even worse: veiligheid vs beveiliging.

'Safe' programming languages?

You can write insecure programs in ANY programming language.

Eg

- You can forget or screw up forget input validation in any language
- Flaws in the program logic can never be ruled out

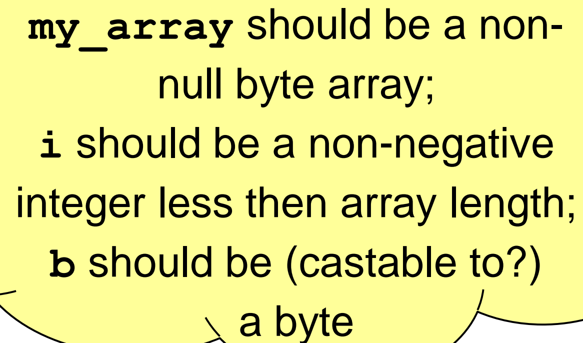
Still...some safety features can be nice

General idea behind safety

Under which conditions does statement

```
my_array[i] = (byte)b;
```

make sense?



Two approaches

- the programmer is responsible for ensuring these conditions
  "unsafe" approach
- the language is responsible for checking this
  "safe" approach

(Heated) debates about the pros & cons highlight tension between
flexibility, speed and control vs safety & security

But note: execution speed ≠ speed of development of secure code

Safe programming languages

Safe programming languages

- impose some discipline or restrictions on the programmer
- offer and enforce some abstractions to the programmer, with associated guarantees

This takes away some freedom & flexibility from the programmer, but hopefully extra safety and clearer understanding makes it worth this.

Attempts at a general definition of safety

A programming language can be considered *safe* if

1. You can trust the abstractions provided by the programming language

   The programming language enforces these abstractions and guarantees that they cannot be broken

   - Eg a **`boolean`** is either **`true`** or **`false`**, and never **`23`** or **`null`**
   - Programmer doesn't have to care if **`true`** is represented as **`0x00`** and **`false`** as **`0xFF`** or vice versa

2. Programs have a precise & well defined semantics (ie. meaning)
   – More generally, leaving things UNDEFINED in any specification is asking for security trouble

3. You can understand the behaviour of programs in a modular way

'safer' & 'unsafer' languages

Warning: this is overly simplistic, as there are many dimensions of safety

Spoiler alert: functional languages such as Haskell are safe because data is immutable (no side-effects)

Dimensions & levels of safety

There are many dimensions of safety

memory-safety, type-safety, thread-safety, arithmetic safety, guarantees about (non)nullness, about immutability, about (absence of) aliasing,...

For each dimension, there can be many levels of safety

Eg, in increasing level of safety, going outside array bounds may:

1. let an attacker inject arbitrary code
2. *possibly* crash the program (or else corrupt some data)
3. *definitely* crash the program
4. throw an exception, which the program can catch to handle the issue gracefully
5. be ruled out at compile-time

Items 1–2: 'unsafe'; some undefined semactics

Items 3–5: 'safe'

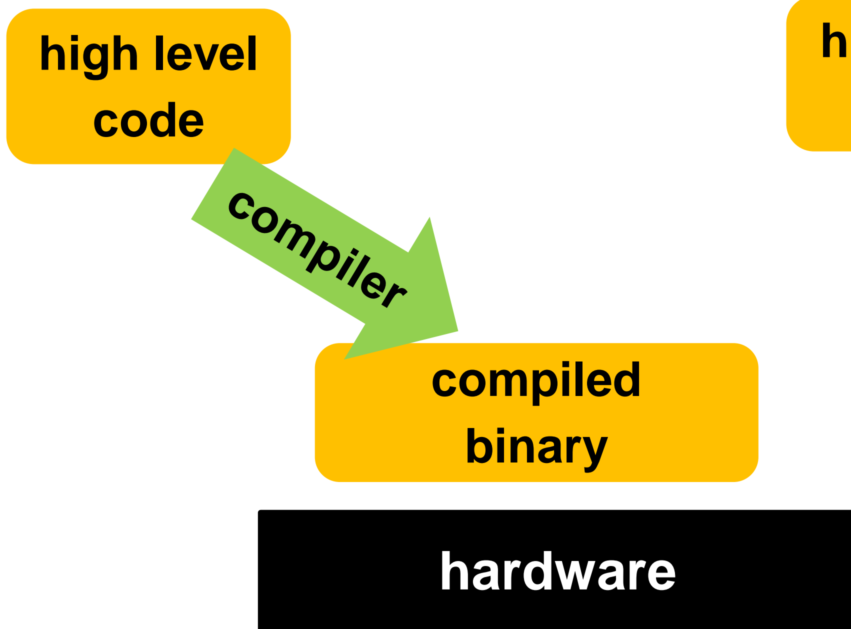
Safety: how?

Mechanisms to provide safety include

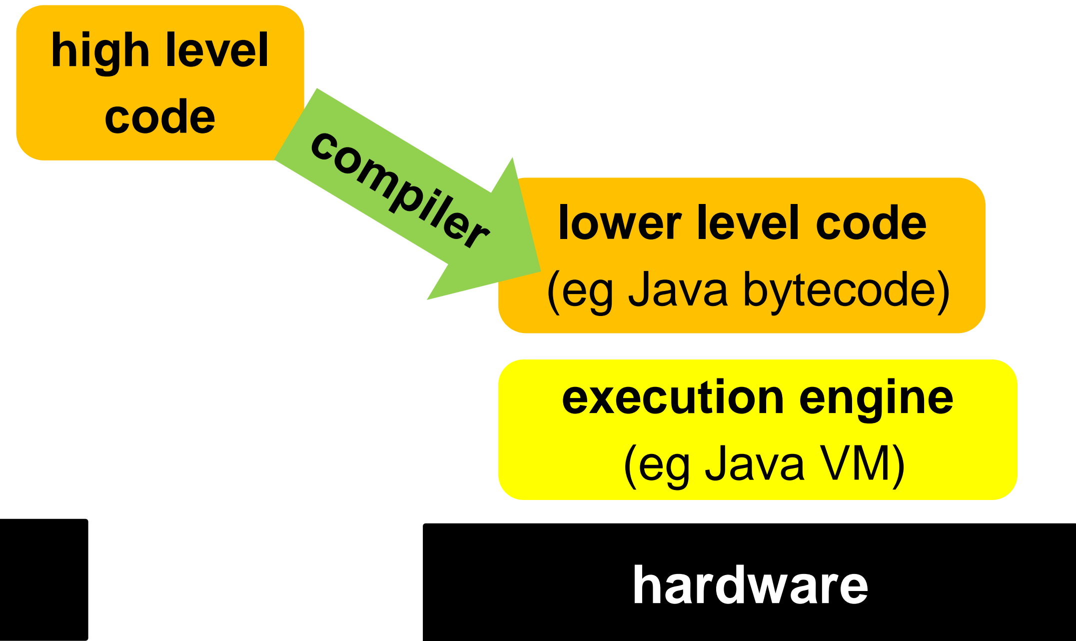
- compile time checks, eg type checking
- runtime checks, eg array bounds checks, checks for null-ness, runtime type checks, ...
- automated memory management using a garbage collector
  - so the programmer does not have to `free()` heap-allocated data
- using an execution engine, to do the things above
  - eg the Java Virtual Machine (VM), which runs the bytecode verifier to type-check code, perfoms some runtime checks, and periodically invokes the garbage collector

Compiled binaries vs execution engines

Compiled binary runs on bare hardware



Execution engine (aka 'runtime') isolates code from hardware



Any defensive measures have to be compiled into the code.

The programming language still exists at runtime, and the execution engine can provide checks at runtime

Memory-safety

Memory-safety – two different flavours

A programming language is memory-safe if it guarantees that

1. programs can never access unallocated or de-allocated memory
   - hence also: no segmentation faults at runtime
2. maybe also: program can never access *uninitialised* memory

Here

1. means we could switch off OS access control to memory.
   Assuming there are no bugs in our execution engine...
2. means we don't have to zero out memory before de-allocating it to
   avoid information leaks (within the same program).
   Again, assuming there are no bugs in our execution engine...

Memory safety

Unsafe language features that break memory safety

- no array bounds checks
- pointer arithmetic
- null pointers, *but only if these cause undefined behaviour*

Null pointers in C

Common (and incorrect!) folklore:

dereferencing a NULL pointer will crash the program.

But, the C standard only guarantees

the result of dereferencing a null pointer is undefined.

So it *may* crash the program, but ANYTHING ELSE *might happen*

See the CERT Secure Coding guidelines for C

https://www.securecoding.cert.org/confluence/display/c/EXP34-C.+Do+not+dereference+null+pointers

for discussion of a security vulnerability in a PNG libary caused by a null dereference that didn't crash (on ARM processors).

Excerpts of C11 standard

"If an invalid value has been assigned to the pointer, the behavior of the unary * operator is undefined.[102]

[102]: Among the invalid values for dereferencing a pointer by the unary * operator are a null pointer,[...]"

"A null-pointer constant is either an integral constant expression that evaluates to zero (such as **`0`** or **`0L`**) , or a value of type nullptr_t (such as **`nullptr`**)."

Memory safety

Unsafe language features that break memory safety

- no array bounds checks
- pointer arithmetic
- null pointers, *but only if these cause undefined behaviour*
- manual memory management
  - esp. manual de-allocation, eg with **`free()`** in C;
    this causes *dangling pointers, use-after-free* and *double-free* bugs

Manual memory management can be avoided by

- not using the heap at all (eg in MISRA C), or
- automating it with a garbage collector
  - Garbage collection first used in LISP in 1959,
    and went mainstream with Java in 1995

Type-safety

Types

- Types assert certain invariant properties of program elements. Eg
  - This variable will always hold an integer
  - This function will always return an object of class X (or one of its subclasses)
  - This array will never store more than 10 items

  NB there is a *wide range of expressivity* in type systems!

- Type checking verifies these assertions. This can be done
  - at compile time (static typing) or
  - at runtime (dynamic typing)

  or a combination.

- Type soundness (aka type safety or strong typing)
  A language is type sound if the assertions are guaranteed to hold at run-time

Type information

```
public class Demo{
  static private string greeting = "Hello";
  final static int CONST = 43;

  static void  Main (string[]  args){
     foreach (string name in args){
         Console.Writeline(sayHello(name));
     }
  }

  public static string sayHello(string name){
     return greeting + name;
  }
}
```

`greeting` only accessible in class Demo

`CONST` will *always* be 43

sayHello will always return a string

sayHello will always be called with 1 parameter of type string

Type-safety

Type-safety programming language guarantees that programs that pass the type-checker can only manipulate data in ways allowed by their types

- So you cannot multiply booleans, dereference an integer, take the square root of reference, etc.

  NB: this removes lots of room for undefined behaviour

- For OO languages: no “Method not found” errors at runtime

Combinations of memory & type safety

Programming languages can be

- memory-safe, typed, and type sound:
  - Java, C#, Rust, Go
    - though some of these have loopholes to allow unsafety
  - Functional languages such as Haskell, ML, Clean, F#
- memory-safe and untyped
  - LISP, Prolog, many interpreted languages
- memory-unsafe, typed, and type-unsafe
  - C, C++

  Not type sound: using pointer arithmetic in C, you can break any guarantees the type system could possibly make

More generally: without any memory safety, ensuring type safety is impossible.

Example – breaking type soundness in C++

```
class DiskQuota {
  private:
      int MinBytes;
      int MaxBytes;
};

void EvilCode(DiskQuota* quota) {
    // use pointer arithmetic to access
    // the quota object in any way we like!
    ((int*)quota)[1] = MAX_INT;

}
```

NB For a C(++) program we can make *no guarantees whatsoever* in the presence of untrusted code.

Hence, in a code review we have to look at *all code* to make guarantees

Ruling out buffer overflows in Java or C#

Ruled out at language-level, by combination of

- compile-time typechecking (static checks)
  - or at load-time, by bytecode verifier (bcv) rather than compile time
- runtime checks (dynamic checks)

What runtime checks are performed when executing the code below?

```
public class A extends Super{
  protected int[] d;
  private A next;

  public A() { d = new int[3]; }
  public void  m(int j) { d[0] = j; }
  public setNext(Object s)
        next = (A)s;
  }
}
```

runtime checks for
1) non-nullness of d,
and 2) array bound

runtime check for
type (down)cast

Remaining buffer overflow issues in Java or C#

Buffer overflows can still exist, namely:

1. in native code
2. for C#, in code blocks declared as `unsafe`
3. through bugs in the Virtual Machine (VM) implementation, which is typically written in C++....
4. through bugs in the implementation of the type checker, or worse, bugs in the type system (unsoundness)

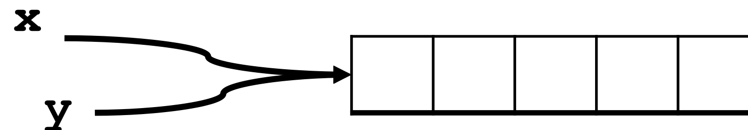
The VM (incl. the type checker aka byte code verifier) is part of the ***Trusted Computing Base (TCB)*** *for memory and type-safety*,

Hence 3 & 4: bugs in it can break these properties.

Breaking type safety?

Type safety is an extremely fragile property:
one tiny flaw brings the whole type system crashing down

In the end, values and objects are just blocks of memory.
If we can create type confusion, by having two references with different types to the same block of memory, then *all* guarantees provided by the type system are gone.



Eg, type confusion attack on Java in Netscape 3.0:

```
public class A[]{ ... }
```

Netscape's Java execution engine confused this type **`A[]`** with the type array of **`A`**

Root cause: [and] should not be allowed in class names

– So this is another input validation problem!

Type confusion attacks

```
public class A{
    public Object x;
     ...
}
```

What if we compile B against A but we run it against A?

We can do pointer arithmetic again!

If Java Virtual Machine would allow such so-called *binary incompatible* classes to be loaded, the whole type system would break.

```
public class A{
  public int x;
  ...
}

public class B{
 void setX(A a) {
  a.x = 12;
 }
}
```

How do we know a type system is sound? (1)

- Representation independence (for booleans)

  it does not matter if we represent true as 0 and false as 1 (or FF), or vice versa

  - ie. if we execute a given program with either representation, the result is guaranteed to be the same

- We could test this, or try to prove it.

  Given a formal mathematical definition of the programming language, we could prove that it does not matter how true and false are represented for all programs

- Similar properties should hold for all datatypes.

How do we know type system is sound? (2)

Give two formal definitions of the programming language

- a typed operational semantics, which records and checks type information at runtime
- an untyped operational semantics, which does not

and prove their equivalence for all well-typed programs.

Or, in other words, prove the equivalence of

- a defensive execution engine (which records and checks *all* type information at runtime) and
- a normal execution engine which does not

for any program that passes the type checker.

People have formalised the semantics and type system of eg Java, using theorem provers (Coq, Isabelle/HOL), to prove such results.

Ongoing evolution to richer type systems

Many ways to enrich type systems further, eg

- distinguishing non-null and possibly-null types

```
public @nonNull String hello = "hello";
```

  to

  - improve efficiency

  - prevent null pointer bugs or catch them earlier, at compile time

- alias control
  restrict possible interferences between modules due to aliasing

- information flow
  controlling on the way tainted information flows through an implementation.
  More on type systems for information flow in later lecture.

Other language-based guarantees

- visibility: public, private, etc
  - eg private fields not accessible from outside a class
- immutability
  - of primitive values (ie constants)
    - in Java : `final int i = 5;`
    - in C(++) : `const int BUF_SIZE = 128;`

    Beware: meaning of `const` get confusing for C(++) pointers and objects!
  - of objects
    - In Java, for example `String` objects are constants

Scala and Rust provides a more systematic distinction between mutable and immutable data.

Safe arithmetic

What happens if `i=i+1;` overflows?

What would be unsafe or safe(r) approaches?

1. *Unsafest approach*: leaving this as undefined behavior
   - eg C and C++
2. *Safer approach*: specifying how over/underflow behaves
   - eg based on 32 or 64 bit two-complements behaviour
   - eg Java and C#
3. *Safer still*: integer overflow results in an exception
   - eg checked mode in C#
4. *Safest*: have infinite precision integers & reals, so overflow never happens
   - Some experiments in functional programming languages

Thread-safety

Problems with threads (ie. lack of thread safety)

- Two concurrent execution threads both execute the statement

```
x = x+1;
```

where **x** initially has the value 0.

What is the value of **x** *in the end?*

– Answer: **x** can have value 2 or 1

- The root cause of the problem is a **data race**:
  x = x+1 is *not* an atomic operation, but happens in two steps - reading **x** and assigning it the new value - which may be interleaved in unexpected ways

- Why can this lead to security problems?

Think of internet banking, and running two simultaneous sessions with the same bank account… *Do try this at home!* ☺

Weird thread behaviour in Java

```
class A {
    private int i ;
    A() { i = 5 ;}
    int geti() { return i; }
}
```

Can **geti()** ever return something else than 5?
Yes!

Thread 1, initialising x

```
static A x = new A();
```

Thread 2, accessing x

```
j = x.geti();
```

You'd think that here **x.geti()** returns 5 or throws an exception, depending on whether thread 1 has initialised **x**

Hence: **x.geti()** in thread 2 can return 0 instead of 5

Execution of thread 1 takes in 3 steps

1. allocate new object m
2. m.i = 5;
3. x = m;

the compiler or VM is allowed to swap the order of these statements, because they don't affect each other

Weird thread behaviour in Java

```
class A {
    private final int i ;
    A() { i = 5 ;}
    int geti() { return i;}
}
```

Now **geti()** always return 5.

Declaring a private field as final fixes this particular problem

- due to ad-hoc restrictions on the initialisation of final fields

- In a revision of the Java Memory Model, which specifies how compilers & VM (incl. underlying hardware) can deal with concurrency, in 2004.
- The API implementation of String was only fixed in Java 2 (aka 1.5)

Data races and thread-safety

- A program contains a data race if two threads simultaneously access the same variable, where at least one of these accesses is a write
  - NB data races are highly non-deterministic, and a pain to debug
- thread-safety = the behaviour of a program consisting of several threads can be understood as an interleaving of those threads
- In Java, the semantics of a program with data races is effectively undefined, ie. only programs without data races are thread safe

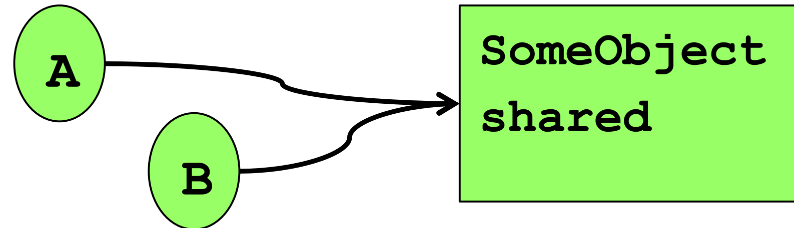
  Moral of the story:

  Even purportedly "safe" programming languages can have very weird behaviour in presence of concurrency
- The programming language Rust aims to guarantee the absence of data races, ad thread-safety, at the language level

Why things often break in C(++), Java, C#, ...

Dangerous combination: **aliasing** & **mutation**

Eg threads or objects **`A`** and **`B`** both have a reference to a mutable object **`shared`**



This is the root cause of many problems, not just with concurrency

1. in concurrent (multi-threaded) context: data races
   - Locking objects (eg **`synchronized`** methods in Java) can help, but: expensive & risk of deadlock
2. in single-threaded context: dangling pointers
   - Who is responsible for free-ing **`shared`** ? **`A`** or **`B`** ?
3. in single-threaded context: broken assumptions
   - If **`A`** *changes* the **`shared`** object, this may break **`B`**'s code, because **`B`**'s assumptions about **`shared`** are broken

Rust

A programming language aimed at low-level 'system' programming

- typed & type-safe
  - non-nullness and immutability by default
- memory-safe
  - without a garbage collection, but using ownership
  - inspired by safe conventions in writing C++ code,
    such as RAII (Resource Aquisition Is Initialisation) style
- thread-safe
  - by ownership & explicitly tracking (im)mutability

- Some inspiration from functional programming,
  eg pattern-matching, type inference, traits
- Supported by Mozilla. More info at https://www.rust-lang.org.
  Beware: older online material (< May 2015) elsewhere may be out of date.

non-null by default

References cannot be null. If you want to allow for undefinedness, you have to explicitly use the **`Option`** enumeration type.

The type **`Option<T>`** has two options

- **`None`** , to indicate failure or a lack of any value
- **`Some(value)`** , for some **`value`** of type **`T`**

Pattern matching is used to distinguish these, eg

```
fn print(x:Some<String>) {
  match x {
     None =>  { ... // handle the 'null' case}
     Some(s) => { ... // use s }
}
```

immutability

Variables and objects are immutable by default.

– To allow mutation, you have to explicitly enable it.

```
fn increase (n:i32) -> i32 {
                    // i32 is the type of 32 bit integer
   let i = n; // The type i32 of i is inferred
   i = i+1;   // Error: i not been declared as mutable
   let mut j = n;
   j = j+1;   // ok
   return j;
}
```

ownership & automated memory management

- A variable binding ('assignment') takes ownership of data
  - A piece of data can only have one owner at a time
- When a variable binding goed out of scope, the data it owns is automatically released; if this is heap-allocated data, it is de-allocated
  - Data must be guaranteed to outlive its references, or this would be unsafe

```
fn example(){
  let mut v = vec![1,2,3];
       // Vector allocated and ownership given to v
       // v's type Vec<i32> is inferred
  v.pop();
  v.push(25);
  println!("", v[2]);
} // As v goes out of scope, the vector is de-allocated
```

move semantics

Assignment moves ownership

- because a piece of data can only have one owner.

```
let mut v = vec![1,2,3];
v.push(25);
let v2 = v;
       // Ownership of the vector given to v2,
       // Variable binding of v is no longer valid.
println!("{}", v[2]);
      // Error, as v no longer owns the vector.
```

borrowing

When calling functions, you could pass ownership back and forth. Alternatively, a variable's data can be borrowed, by taking a reference to it. The original variable retains ownership, but cannot transfer ownership while borrow lasts.
Borrowers are only allowed to read (to prevent data races)

```
let v = vec![1,2,3];
{ let v_ref = &v; // v_ref is a reference to v
  let len  = length(v_ref); // Reference to v passed to length
                 // You could also write this as length(&v)
  let v2 = v ;  // Disallowed, as v's data is still borrowed
} // v_ref goes out of scope, so borrow ends
let v3 = v;  // Allowed, as there are no references to v
```

Here the function **`length`** has type **`&Vec<32> -> i32`**

References are also immutable by default. To allow **`v_ref`** to change, it would have to be declared as **`mut`**

mutable borrowing

For a function to modify an argument, the data has to be borrowed mutably.

```
fn length(v_ref: &Vec<i32> ): i32 { ... }
fn push(v_ref: &mut Vec<i32> , i:i32) { ... }

fn main(){
  let v = vec![1,2,3];
  let v_ref = &mut v;
  push(v_ref, 24);
}
```

To prevent data races, an object of type `T` can have

- *many immutable* references to it (of type `&T`)
- XOR *exactly one mutable* reference to it (of type `&mut T`)

preventing use-after-free errors

```
let v_ref : &Vec<i32>;
{ let v = vec![1,2,3];
   v_ref = &v ; // Error: v does not live long enough
}
// Without this error, v_ref would be dangling here
```

So dangling pointer error (use-after-free) results in compile time error

Pros & cons of ownership

Pros

- automated memory management, without the overhead of a garbage collector
- easy way to safely program concurrency

Cons: ownership rules make some things impossible: you cannot

- implement a doubly-linked list, or any circular data structures in general
- call C libraries

Hence, there is the `unsafe` construct

```
unsafe {
     ...
 }
```

which relaxes some rules