Dynamic Dispatch, Garbage Collection, and Rust

CIS352 — Fall 2024

Kris Micinski

- In this lecture, we'll talk about three final topics, in varying levels of detail
  - Dynamic Dispatch (implementing objects in C++)
  - Garbage Collection
  - Rust (borrow checker, etc…)
- So far in class, I've focused on Racket / lambda calculus—this time, I'll mostly use other languages to show how some of the ideas from the class apply

Closures

- In Racket/lambda calculus/etc…: returning a lambda *allocates* a closure. The closure *captures* ("closes over") the free variables that would be in scope:
  - `(let ([x 5]) (lambda (y) (+ x y)))`
    - Returns a closure of `(lambda (y) (+ x y))` and the captured environment, `{x ↦ 5}`
    - Need the environment to interpret x
      - Could use substitution—but that's slow!

Objects

- A very similar thing happens in OO languages: new allocates an object, calling the constructor to store arguments in fields
  - In FP, the environment is implicitly captured via a lambda, but in OOP, the constructor explicitly captures fields
- In OOP, the primitive notion of a function is a *method*, which may syntactically reference *instance* variables and *arguments*
- In FP, the closure allows you to reference any free variables (the contents of the environment stored by the closure) and arguments

```
class B
{
    virtual int f() { return 1; }
};
class A : public B
{
    virtual int f() { return 2; }
};


B* a = new A(); // Get a pointer to an A obj
std::cout << a->f() << std::endl;
```

// 2 is printed out, because A is the runtime class

Function pointers

```
int add1(int x) { return x+1; }
```

In stored-program machines, all code sits somewhere in memory.

In C/C++, you can obtain pointers to functions at run-time,
and invoke them! The pointer for `add1` can be obtained with:

```
&add1
```

```
int add1(int x) { return x+1; }

int main()
{
    int (*f)(int) = &add1;

    // …

    int four = (*f)(3);
}
```

A function pointer, `cmp`, passed to `sort` as an argument.

```
int sort(int* x, int len, bool (*cmp)(int,int))
{
    // …

        // …
            if ((*cmp)(*x,*y))
            {
                swap(*x,*y);

                // …
            }

    // …
}
```

The function pointer, `cmp`, dereferenced and invoked.

```
{
    // …

    sort(buff, length, &lessthan);

    // …
}
```

A pointer to function `lessthan` **is passed into** `sort`**.**

A function pointer, `cmp`, type `int x int -> bool`, is a template parameter to `sort`.

```
template <bool (*cmp)(int,int)>
int sort(int* x, int len)
{
        // …
            if ((*cmp)(*x,*y))
            {
                swap(*x,*y);
                // …
```

Templated function `sort` is invoked with a template parameter like so: `sort<…>(…)`

```
int main()
{
    // …
    sort<&lessthan>(buff, length);
```

C++ dynamic dispatch: class polymorphism

```
class Cmp
{
    virtual bool cmp(int x, int y) = 0;
};
class LessThan : public Cmp
{
    virtual bool cmp(int x, int y)
    { return x < y; }
};
class GreaterThan : public Cmp
{
    virtual bool cmp(int x, int y)
    { return x > y; }
};
```

An instance of type `Cmp`, `cmp`, has overloaded method `cmp`.

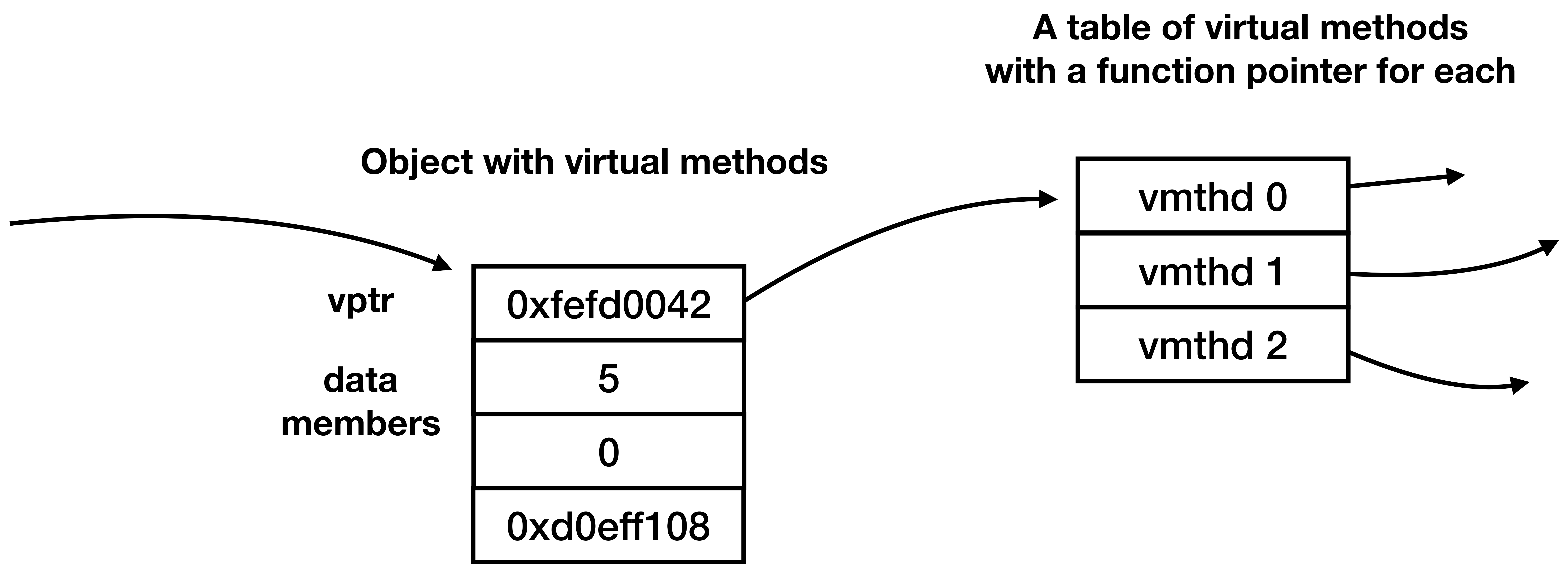
```
int sort(int* x, int len, const Cmp& cmp)
{
        // …
            if (cmp.cmp(*x,*y))
            {
                swap(*x,*y);
                // …
```

Pass in object `lessthan` by reference to polymorphic type Cmp supporting the Cmp::cmp(int, int) member.

```
int main()
{
    // …
    LessThan lessthan;
    sort(buff, length, lessthan);
```

Virtual Tables (vtables)

Virtual Tables (vtables)



```
class Animal
{
    virtual const char* name() = 0;
    virtual int weight() const = 0;
    virtual void eat(Animal* prey)
    {
        if (this->weight()
                < 2 * prey->weight())
            return;
        delete prey;
        std::cout << prey->name()
                  << “ was eaten!\n”;
    }
};
```

```
class Mouse : public Animal
{
    int grams;

    Mouse(int grams)
        : grams(grams) {}

    virtual const char* name()
    {
        return “Mouse”;
    }

    virtual int weight() const
    {
        return this->grams;
    }
};
```

```
class Cat : public Animal
{
    Cat() {}

    virtual const char* name()
    {
        return “Cat”;
    }

    virtual int weight() const
    {
        return 4260;
    }
};
```

```
class Giraffe : public Animal
{
    virtual const char* name()
    {
        return “Giraffe”;
    }
    virtual int weight() const
    {
        return 1570000;
    }
    virtual void eat(Animal* prey)
    {
        std::cout << this->name()
                  << “ wont eat that.\n”;
    }
};
```

```
// vtable struct for Animal subclasses
struct AnimalVTable
{
    const char* (*name)(void*);
    int (*weight)(const void*);
    void (*eat)(void*,void*);

    AnimalVTable(const char* (*name)(void*),
                 int (*weight)(const void*),
                 void (*eat)(void*,void*))
      : name(name), weight(weight), eat(eat)
    {}
};

// Allocate a vtable for each concrete Animal
AnimalVTable mouse_vtable(&nameMouse,
                          &weightMouse,
                          &eatAnimal);
```

```c
// Class Mouse compiled to a struct
struct Mouse
{
   AnimalVTable* vptr;
   int grams;
};


// An allocator/constructor for Mouse
Mouse* newMouse(int grams)
{
    Mouse* m = (Mouse*)malloc(sizeof(Mouse));
    m->vptr = &mouse_vtable;
    m->grams = grams;
    return m;
}
```

```
// A name method for Mouse instances
const char* nameMouse(void* _ths)
{
    return “Mouse”;
}


// A weight method for Mouse instances
int weightMouse(const void* _ths)
{
    const Mouse* ths = (const Mouse*)_ths;
    return ths->grams;
}
```

```
// Looks up the vtable for an object
VTable* vtable(void* obj)
{
    return (VTable*)((void**) obj)[0];
}
```

```
{
    // To call a member function f:
    // e.g., obj->f(arg0, arg1, …);

    vtable(obj)->f(obj, arg0, arg1, …);
}
```

```
// Looks up the vtable for an Animal object
AnimalVTable* vtable(void* obj)
{
    return (AnimalVTable*)((void**) obj)[0];
}

// A default eat method for Animals
void eatAnimal(void* ths, void* prey)
{
    if (vtable(ths)->weight(ths)
            < 2 * vtable(prey)->weight(prey))
        return;
    delete prey; // vtable(prey)->~Animal…
    std::cout << vtable(prey)->name(prey)
              << “ was eaten!\n”;
}
```

Try an example:

How do you define the constructor for Giraffe?

```
// Class Giraffe compiled to a struct
struct Giraffe
{
   AnimalVTable* vptr;
   // No data members
};

AnimalVTable giraffe_vtable(&nameGiraffe,
                            &weightGiraffe,
                            &eatGiraffe);

// An allocator/constructor for Giraffe
Giraffe* newGiraffe()
{
    Giraffe* g = new Giraffe();
    g->vptr = giraffe_vtable;
    return g;
}
```

Try an example:

How do you define the virtual member functions for Giraffe?

```
const char* nameGiraffe(void* _ths)
{
    return “Giraffe”;
}

int weightGiraffe(const void* _ths)
{
    return 1570000;
}

void eatGiraffe(void* _ths)
{
    Giraffe* ths = (Giraffe*)_ths;
    std::cout << vtable(ths)->name(ths)
              << “ wont eat that.\n”;
}
```

Manual Memory Management (C/C++/…)

- In C/C++, all memory is **manually managed**. This is a real problem, because the reality is that—even for good programmers—it is very tricky to ensure that memory does not:
  - Get freed twice (double free)
  - Get leaked (reference falls out of scope without free)
  - Get corrupted (pointers go out of bounds, SIGSEGV)

In practice, each of the above can / do lead to potentially-serious security vulnerabilities in systems of sufficient complexity.

Garbage Collection

- In contrast to manual memory management, Racket/Java/JS/… use *automatic memory management* (via garbage collection)
  - **No** explicit memory allocation, **no** explicit pointers even (no pointer arithmetic!), only *references*
  - The garbage collector (GC) runs occasionally, in the background
  - This allows temporary waste (unreachable/dead objects) until a GC pause (“stop the world”), at which point we throw away everything which we know must be trash.
- GC is very good/fast now, and a hallmark of modern languages!
  - Java, C#, JS, Python, many other languages you probably use

Mark/Sweep Collectors

- The most basic kind of collector is a mark/sweep collector
- Occasionally, the interpreter is paused
- Mark everything as dead
- Start with a "root set," which comprises all *definitely reachable / alive* data (typically pointers on the stack, registers, globals, etc…)
  - Everything in the root set becomes alive
- For every data structure, inspect its pointers—mark all of the pointers as reachable / alive
- Repeat this process until you've found everything reachable
- Recall this is the ***transitive closure*** algorithm from **project 2**

Issues with Mark/Sweep

- Stop-the-world nature
  - Undesirable in a real-time setting (safety-critical software, etc…)
- In practice, mark/sweep garbage collection is very slow—it examines the whole heap at every GC pause
  - Overall system throughput goes down, expensive, bad for interactive apps
- Fragmentation
  - Can ameliorate this via a ***copying/compacting*** collector
- Challenging to make concurrent / parallel
- Lots of cache eviction
- Etc…

Generational GC

- Mark/sweep is popular, but modern GCs use a mix of insights
- One insight is that—if an object has been around a while (multiple GC cycles)—it will likely remain around a while
- ***Generational*** GC partitions objects into *generations* based on when they are allocated:
  - Allocate objects into a ***minor*** heap which is GCd frequently
  - Once they have been alive a while, move them into **major** heap
  - Minor GC only needs to look at the most-recently-allocated objects (relatively small compared to the major heap often)
  - Run major GC cycle once in a while
    - Always possible to delay GC, you may just waste memory
- Lower latency, better throughput, now in Java, OCaml, …

Concurrent GC, etc..

- On multi-core machines, “stop the world” collectors really kill throughput (Amdahl’s law); in a parallel setting, synchronization almost always translates into a throughput hit
- Many implementations now use ***concurrent*** GC:
  - Incremental marking
  - Break mark/sweep tasks into small chunks
- E.g., V8 (JavaScript for Chrome, Firefox) incremental GC:
  - Stop-the-world for the minor heap, concurrent techniques for the major heap. Make minor heap collection very fast (even if sequential); Major heap collected concurrently, higher latency but better overall throughput

Custom-Purpose Allocators

- If you program in a managed language (C#, JavaScript, Java, …) you'll have some kind of GC, and the dynamics of the GC may matter, especially in situations with high data loads and lots of concurrent actions occurring
  - If you use these languages in an enterprise setting, may want to read more about the specific GC your runtime engine uses
- If you use a native language (Rust, C++, C, …) then you need to manage memory yourself
  - Modern C++ / Rust provides some features which provide quasi-automatic memory management, e.g., auto pointers
- Also, may want to use custom allocators, e.g., slab allocator:
  - Big array of chunks of data of a specific size, very fast
    - E.g., Chez Scheme uses a slab allocator for cons cells

Reference Counting

- Garbage collection is fairly heavyweight—*requires* a runtime system (typically seen in "managed" languages)
- Reference counting is even simpler:
  - Every heap-allocated object gets an associated *reference count*
  - Every time the pointer is copied, the reference count is bumped
  - When pointer goes out of scope, reference count decremented
  - When reference count goes to zero, free the associated object

x is initialized, object is allocated on heap, constructor called (initializes fields, etc…), reference count set to 1

```
{
  Foo *x = new Foo();
  // …
  {
    Foo *y = x;
    // …
  } // y goes out of scope
} // x goes out of scope
```

1	Foo’s fields, etc…

Pointer to x copied into y, underlying object's reference count bumped to 2

```
{
  Foo *x = new Foo();
  // …
  {
    Foo *y = x;
    // …
  } // y goes out of scope
} // x goes out of scope
```

2	Foo's fields, etc…

y goes out of scope, pointer is now unreachable,
decrement count back to 1

```
{
  Foo *x = new Foo();
  // …
  {
    Foo *y = x;
    // …
  } // y goes out of scope
} // x goes out of scope
```

1	Foo’s fields, etc…

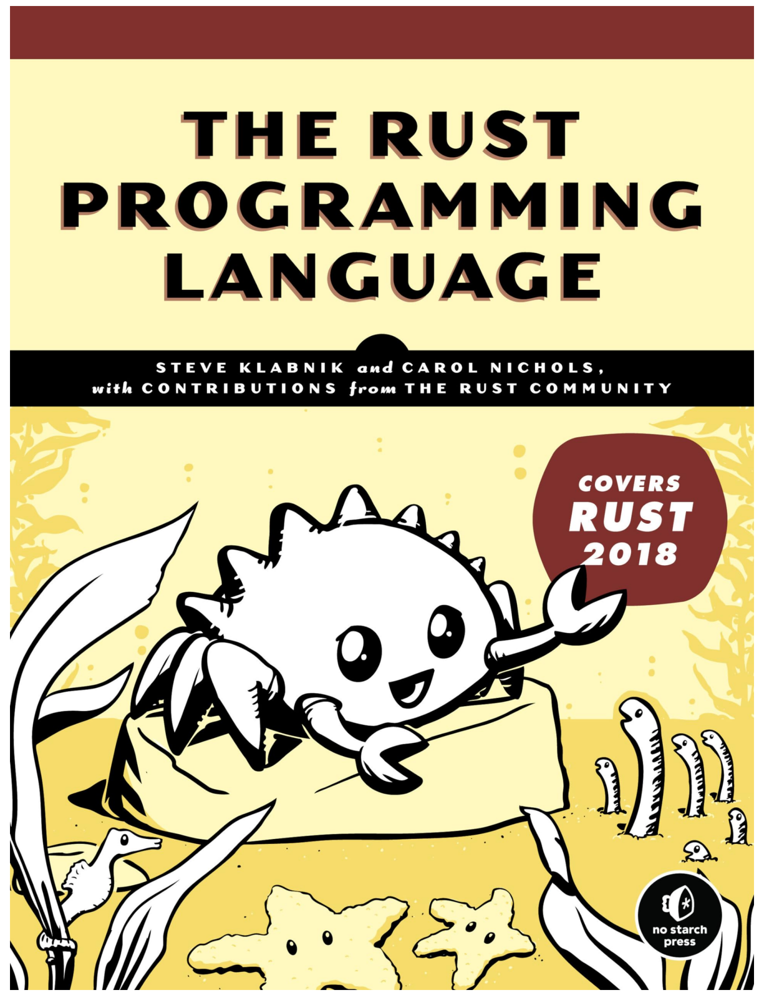
Finally, x goes out of scope: reference count goes down to 0 which triggers destruction and deallocation (freeing)


```
{
  Foo *x = new Foo();
  // …
  {
    Foo *y = x;
    // …
  } // y goes out of scope
} // x goes out of scope
```

- Reference counting offers many of the benefits of automatic memory management without the need for a dedicated (stop the world) garbage collector or runtime system
- Runtime systems may be too expensive (memory / time-intensive) in power-constrained settings
- Implementations in many native languages
  - C++’s std::auto_ptr, Rusts’s Rc<…>

Rust

- Relatively new (2015) systems-focused programming language
- Replaces (mostly) C, C++, etc…
- No runtime system, low-level access to memory layout (mostly)
- Memory safe by design
  - Unlike C/C++, no concern over massive memory bugs (segfaults)
- Type system takes inspiration from Haskell/OCaml, etc…
  - Includes a **borrowing** system to ensure that memory access invariants are maintained
- Values are immutable by default, defined with `let`
- Can make values mutable by using `let mut`



println! is a **macro**, which generates code at compile-time. Rust has a powerful macro system, similar to Racket's

```
// Example 1: Ownership and Borrowing
fn main() {
    let s = String::from("hello");
    let len = calculate_length(&s);
    println!("The length of '{}' is {}.", s, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

calculate_length borrows a *reference* to a String

This avoids *copying*, which is (in general) costly

```rust
// Example 2: Pattern Matching with Enums
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}

fn main() {
    let coin = Coin::Quarter;
    println!("The value of the coin is {} cents.", value_in_cents(coin));
}
```

Rust has pattern matching, enums are enumerated types given by a specific list of constructors

Here we use Result to signal that the return type is ***either*** a string or an error. The ? says roughly "if this operation fails, return an error"

```
// Example 3: Error Handling with Result
use std::fs::File;
use std::io::Read;

fn read_file_content(filename: &str) -> Result<String, std::io::Error> {
    let mut file = File::open(filename)?;
    let mut content = String::new();
    file.read_to_string(&mut content)?;
    Ok(content)
}

fn main() {
    match read_file_content("example.txt") {
        Ok(content) => println!("File content:\n{}", content),
        Err(e) => println!("Error reading file: {}", e),
    }
}
```

If we get to the end, we return Ok(…), which coerces the string into a Result<…> type

Just like Racket, Rust has closures (`|x| …` is a lambda) which work with iterators

vec! is a macro which builds an (immutable) vector

```
// Example 4: Iterators and Closures
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    let doubled: Vec<i32> = numbers.iter().map(|x| x * 2).collect();
    println!("Original: {:?}, Doubled: {:?}", numbers, doubled);

    let sum: i32 = numbers.iter().sum();
    println!("Sum of numbers: {}", sum);
}
```

```rust
// Example 5: Structs and Traits
struct Rectangle {
    width: u32,
    height: u32,
}

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

fn main() {
    let rect = Rectangle { width: 10, height: 20 };
    println!("The area of the rectangle is {} square units.", rect.area());
}
```

Rust has *structs*, which are objects with fields and methods

The impl block defines methods callable on Rectangles

Now let's walk through an extended example: translating exercise 3 into Rust

We use a Box<…>, which is a smart pointer (with ownership) to an Expr

Environments use HashMap<String,Value>

```
use std::collections::HashMap;

#[derive(Debug, Clone)]
enum Expr {
    Number(i64),
    Add(Box<Expr>, Box<Expr>),
    Var(String),
    Lambda(String, Box<Expr>),
    App(Box<Expr>, Box<Expr>),
}

#[derive(Debug, Clone)]
enum Value {
    Number(i64),
    Closure(String, Box<Expr>, Environment),
}

type Environment = HashMap<String, Value>;
```

We use *e0/e1 to get the box's underlying value

```
fn interp(expr: Expr, env: &Environment) -> Value {
    println!("At expression: {:?}, env: {:?}", expr, env);
    match expr {
        Expr::Number(n) => Value::Number(n),
        Expr::Add(e0, e1) => {
            if let Value::Number(v0) = interp(*e0, env) {
                if let Value::Number(v1) = interp(*e1, env) {
                    Value::Number(v0 + v1)
                } else {
                    panic!("Expected a number in addition");
                }
            } else {
                panic!("Expected a number in addition");
            }
        }
        Expr::Var(x) => env.get(&x).cloned().unwrap_or_else(|| {
            panic!("Unknown variable: {}", x);
        }),
        Expr::Lambda(param, body) => Value::Closure(param, body, env.clone()),
        Expr::App(e0, e1) => {
            let v0 = interp(*e0, env);
            let v1 = interp(*e1, env);

            match v0 {
                Value::Closure(param, body, mut closure_env) => {
                    closure_env.insert(param, v1);
                    interp(*body, &closure_env)
                }
                _ => panic!("Tried to apply {:?}, but it is not a closure", v0),
            }  }  }  }
```

Summary

- Objects and closures offer similar mechanisms for bundling code + data together, objects with fields, closures w/ captured variables
- Managed languages typically employ automatic memory management in the form of garbage collection
  - GC runs in the background, cleans up unreachable memory
- Rust is a new systems-focused language
  - Not managed, but still memory safe
  - Type system, borrow checker, designed to ensure memory access is safe without necessitating a runtime system (GC, etc…)
  - Modern replacement for C/C++, which are often riddled by tons of tricky memory errors that lead to vulnerable / hard-to-debug code