The Evolution of Memory and Resource Management

- Jeff Walker "Code Ranger"
- .NET Web Developer (C#)
- Jeff@WalkerCodeRanger.com
- @WalkerCodeRangr
- WalkerCodeRanger.com
- Interested in languages
- Worked on design of own programming language
- Attended Columbus Rust Society meet up since Aug 2015

Approach

- From the programmer's perspective
- Ignore hardware and OS issues
  - Partitions
  - Extended Memory
  - Virtual Memory
  - Pagining
- Won't be entirely chronological

Kinds of Memory Management

- Absolute Address
- Static Allocation
- Stack
- Manual Memory Mangement
- Smart Pointers
- Reference Counting
- Garbage Collection
- Borrow Checker

Absolute Address

- Simple
- Used in first computers
- Only option for very limited memory
- Very limiting

Static Allocation

- Only option in older versions of COBOL and Fortran
- Can't support recursive function/subroutine calls
- All data structure size known at compile time

Stack

- Enables recursion
- Fast allocation and deallocation
- Avoids fragmentation
- Often limited to fixed size allocations

Manual Memory Management

- Aka heap allocation
- **`malloc()`**, **`free()`**, **`realloc()`** in C
- **`new`**, **`delete`** in C++

Manual Memory Management Issues

- Easy for developer to make mistakes
- Use after free
- Double free
- Memory Leak
- Use before alloc
- Not allocating enough
- Reading uninitalized memory
- Heap fragmentation

Reference Counting

- Increment when adding reference
- Decrement when removing reference
- Releases resources as soon as possible
- Will fail to release cycles
- May need weak references to handle cycles
- May be manual or somewhat automatic
- Updating the count is inefficent, causes cache issues and may require locking

Automatic Reference Counting

- Used in Objective-C and Swift
- Compiler inserts calls to adjust reference counts

```
// Strong reference, cannot be nil
var strongReference: MyClass
// Strong reference, can be nil
var strongNilReference: MyClass?
// Weak reference, can be nil
weak var weakReference: MyClass?
// Weak reference, cannot be nil
unowned var unownedReference: MyClass
```

Smart Pointers

- Types that act like pointers but add memory management
- C++
  - **`auto_ptr`**
  - **`unique_ptr`** (C++11)
  - **`shared_ptr`** and **`weak_ptr`** (C++11)
- Not always safe (i.e. **`auto_ptr`** could move ownership and old pointer was null)

Garbage Collection

- All memory that is not "reachable" is garbage
- Tracing
- Tri-color
- Compacting vs. non-compacting
- Stop-the-world vs. incremental vs. concurrent
- Generational

Garbage Collection Issues

- Performance is an issue but has greatly improved
- Generally uses and needs more memory
- Very complex
- Serious performance issues may be rare, but very difficult to address
- Can still "leak" by holding a reference
- Disposable pattern and finalizers

Garbage Collection Just for Memory

- Require disposable pattern using finalizers
- Finalizers
  - Difficult to write correctly
  - May not be called for a long time
  - May never be called
  - Order of finalization not determined and may be concurrent
  - Possible resurrection
  - Poor performance – GC must revisit
  - Don't handle exceptions well (Java ignores, C# terminates)

Dispose Pattern

C#

```
using (Font font1 = new Font("Arial",
                                   10.0f))
{
    byte charset = font1.GdiCharSet;
}
```

Java (Prior to Java SE 7, you can use a **`finally`** block)

```
try(BufferedReader br = new
   BufferedReader(new FileReader(path)))
{
    return br.readLine();
}
```

Borrow Checker

- Compile time checking of memory allocation
- Deterministic and Safe
- Introduced by the Rust Programming Language

Rules

- Only one owner at a time
- Owner is only one that can access data (unless they lend it out)
- Ownership can be transferred (move semantics)
- Any borrow must last for a scope no greater than that of the owner
- Have one or the other of these two kinds of borrows, but not both at the same time:
  - one or more references (&T) to a resource
  - exactly one mutable reference (&mut T)
- Owner limited while borrowed

Ownership

- Memory freed at end of scope

```
struct Point
{
    x: i32,
    y: i32
}

fn main()
{
    let p = Point {x: 2, y: 4};
}
```

Move Semantics

```
let p = Point {x: 2, y: 4};
let q = p;
println!("p.x: {}", p.x);
```

```
error[E0382]: use of moved value: `p.x`
  --> <anon>:11:25
   |
10 |     let q = p;
   |         - value moved here
11 |     println!("p.x: {}", p.x);
   |                         ^^^ value used here after move
   |
   = note: move occurs because `p` has type `Point`, which
does not implement the `Copy` trait
```

Move into Function

```
fn take(p: Point)
{
    // not important
}

let p = Point {x: 2, y: 4};
take(p);
println!("p.x: {}", p.x);

error[E0382]: use of moved value: `p.x`
```

Limitations of Ownership

```
fn foo(p: Point) -> Point
{
    // do stuff with p

    // hand back ownership
    return p;
}
```

Borrowing References

```
fn distance_from_origin(p: &Point)
                                 -> f64
{
  return ((p.x*p.x+p.y*p.y)
                      as f64).sqrt();
}

let p = Point {x: 2, y: 4};
let d = distance_from_origin(&p);
println!("d: {}", d);
println!("p.x: {}", p.x);

d: 4.47213595499958
p.x: 2
```

Immutability is the Default

```
let p = Point {x: 2, y: 4};
p.x = 5;

error: cannot assign to immutable
field `p.x`
  --> <anon>:14:5
   |
14 |     p.x = 5;
   |     ^^^^^^^
```

Immutability is the Default

```
let mut p = Point {x: 2, y: 4};
p.x = 5;
```

Mutable References

```
fn double(p: &Point)
{
    p.x *= 2;
    p.y *= 2;
}
```

```
error: cannot assign to immutable field
`p.x`

error: cannot assign to immutable field
`p.y`
```

Mutable References

```
fn double(p: &mut Point)
{
    p.x *= 2;
    p.y *= 2;
}

let mut p = Point {x: 2, y: 4};
double(&mut p);
```

Scopes

```
fn main()
{
   let mut x = 5;
   let y = &mut x;      // + &mut borrow
   *y += 1;             // |
   println!("{}", x);   // | try to borrow
                        // + borrow ends
}
```

```
error[E0502]: cannot borrow `x` as
immutable because it is also
borrowed as mutable
```

Scopes (fixed)

```
fn main()
{
  let mut x = 5;
  {
    let y = &mut x; // + &mut borrow
    *y += 1;        // |
  }                 // + borrow ends

  println!("{}", x); // borrow
}
```

Why?

- Safe data structures
- Iterator invalidation
- Prevent use after free
- Compile time checked
- Performant, no runtime overhead
- Resource management

Resources

- Files
- Networking
- Instead of disposible pattern, use Resource Acquisition is Initialization (RAII) pattern
- Implement `` `Drop` `` trait
- Don't need something like C# `` `using` `` statement

Lifetimes

```
fn skip_prefix(line: &str, prefix: &str) -> &str
{
    // ...
    line
}
fn main()
{
    let line = "lang:en=Hello World!";
    let v = skip_prefix(line, "lang:en=");
    println!("{}", v);
}
```

```
error[E0106]: missing lifetime specifier
  |
1 | fn skip_prefix(line: &str, prefix: &str) -> &str
  |                                            ^ expected lifetime
parameter
  |
  = help: this function's return type contains a borrowed
Value, but the signature does not say whether it is borrowed
From line` or `prefix`
```

Lifetimes

```
fn skip_prefix<'a, 'b>
    (line: &'a str, prefix: &'b str)
    -> &'a str
{
    // ...
    line
}
```

Reflections

- Thinking about Rust can clarify memory management in other languages
- Borrow checker can be very restrictive
- There is room to grow in memory management still

Dyon

- Dynamically typed scripting language, designed for game engines and interactive applications

```
// `a` outlives `b`
fn put(a: 'b, mut b) {
    b[0] = a
}

fn main() {
    a := [2, 3]  // - lifetime of `a`
                 // |
    b := [[]]    // |  - lifetime of `b`
                 // |  |
    put(a, mut b)// |  |
}
```

Adamant

```
public async Main
      (console: mut Console, args: string[]) -> Promise
{
  let results = mut new List<~own Promise<int>>();

  foreach(let file in args)
  {
    console.WriteLine("Begin processing file: "
                                                + file);
    results.Add(ProcessFile(file));
  }

  let lengths = await Promise.WhenAll(results);
  Console.WriteLine("Total Length of File(s): "
                                       + lengths.Sum());
}
```

Adamant

```
public async ProcessFile
            (fileName: string) -> Promise<int>
{
  let file =
            mut new FileReader.Open(fileName);
  let contents = await file.ReadToEnd();

  // simulate work
  await Promise.Wait(
                     new TimeSpan.Seconds(5));
  return contents.Length;
}
```

Questions?

Resources

- rust-lang.org
- doc.rust-lang.org/book
- rustbyexample.com
- www.piston.rs/dyon-tutorial
- adamant-lang.org

Contact Me

- Jeff Walker "Code Ranger"
- Jeff@WalkerCodeRanger.com
- @WalkerCodeRangr
- WalkerCodeRanger.com