

Computer Resources

Computation

- CPU

Memory:

- Persistent memory
 - Hard driver or SSD
- Volatile memory
 - RAM

RAM

- Stack
 - Function Arguments
 - Local Variables
 - Known size at compile time
 - size: Dynamic / Fixed upper limit
 - Cleanup: Automatic / When function returns

Heap

- Values that live beyond a function's lifetime
- Values accessed by multiple threads
- Large values
- Unknown size at compile time
- Size: Dynamic
- Lifetime: Determined by programmer
- Cleanup: Manual

Static

- Program's binary
- Static variables
- String literals
- Size: fixed
- Lifetime: Lifetime of program
- Cleanup: automatic When program terminates

Heap Memory Managing

Managing Memory on the Heap



Manual (C)

- Pros
 - Full control
 - Efficient
- Cons
 - Tedious
 - Error prone

RAII (C++) / OBRM (Rust)

- Pros
 - Full control
 - Efficient
 - Mostly error free
- Cons
 - Somewhat tedious

Automatic (Java, C#, ect.)

- Pros
 - Easy
 - Error free
- Cons
 - No control
 - Not efficient

RAII

利用析构函数 + 对象 (Wrapper了原始对象) 分配到栈上(不使用new)

What is a resource

What is a resource?

- Something with a finite supply that requires management
- Examples
 - Heap allocated memory
 - Network sockets
 - File handles
 - Database handles
 - Mutexes
 - ect.



```
class Car {};\n\nvoid memory_example()\n{\n    Car* car = new Car;           // allocate memory on the heap\n    function_that_can_throw();   // memory leak if exception is thrown\n    if(!should_continue()) return; // memory leak if early return\n    delete car;                 // clean up memory on the heap\n}
```

```
void file_example()\n{\n    ofstream file("example.txt"); // acquire file handle\n    function_that_can_throw();   // file is never closed if exception is thrown\n    if(!should_continue()) return; // file is never closed if early return\n    file.close();               // close file handle\n}\n\n在函数的最后，我们关闭文件句柄。
```

Solution

Use objects (constructors/destructors) to manage resources.

```
class CarManager\n{\nprivate:\n    Car* p; // pointer to a Car\npublic:\n    CarManager(Car* p) : p(p) {}\n    ~CarManager()\n    {\n        // clean up memory on the heap\n        delete p;\n    }\n};
```

```
void memory_example_2()
{
    CarManager car = CarManager(new Car);
    function_that_can_throw();
    if(!should_continue()) return;
}
```

因为car manager是在堆栈上分配的，

Because car manager is allocated on the stack,

```
class File
{
private:
    ofstream file; // file handle
public:
    File(string file_name) {
        file = ofstream(file_name);
    }
    ~File()
    {
        // close file handle
        file.close();
    }
};
```

- only one owner

```
void memory_example_3()
{
    unique_ptr<Car> car = make_unique<Car>();
    function_that_can_throw();
    if(!should_continue()) return;
}
```

除了不用汽车管理器。
except instead of using car manager

- shared

```
shared_ptr<Car> car = make_shared<Car>();
shared_ptr<Car> car2 = car;
function_that_can_throw();
if(!should_continue()) return;
```

共享指针允许您共享资源的所有权。
Shared pointers allow you to share ownership of a resource.

Ownership Based Resource Management (OBRM)

Similar to RAII but instead of being a best-practice/pattern it's a built-in language feature.

Ownership rules are checked at compile time

- Each value in Rust has a variable that's called its owner.
 - There can only be one owner at a time.
 - When the owner goes out of scope, the value will be dropped.
- Rust OBRM VS C++ RAII

Rust中ownership不仅用于内存管理，而且用于资源管理,如文件句柄、网络socket等

```

G+ raii.cpp > memory_example()
class Car {};

void memory_example()
{
    unique_ptr<Car> car =
        make_unique<Car>();
    function_that_can_throw();
    if(!should_continue()) return;
}

void file_example()
{
    File file = File("example.txt");
    function_that_can_throw();
    if(!should_continue()) return;
}

```

```

src > @ obrm.rs > memory_example
11 struct Car {}
12
13 fn memory_example() {
14     let car = Box::new(Car {});
15     let my_string = String::from("LGR");
16     function_that_can_panic();
17     if !should_continue() { return; }
18 }
19
20 fn file_example() {
21     let path = Path::new("example.txt");
22     let file = File::open(&path).unwrap();
23     function_that_can_panic();
24     if !should_continue() { return; }
25 }
26
27
28

```

- 所有权转移对比

```

class Car {};

void memory_example()
{
    unique_ptr<Car> car =
        make_unique<Car>();
    unique_ptr<Car> car2 = move(car);
    function_that_can_throw();
    if(!should_continue()) return;
}

void file_example()
{
    File file = File("example.txt");
    function_that_can_throw();
    if(!should_continue()) return;
}

```

```

11 struct Car {}
12
13 fn memory_example() {
14     let car = Box::new(Car {});
15     let car2 = car;           // 注意，使用Rust，我们不会得到任何错误，这是因为移动语义是隐式的。
16     let my_string = String::from("LGR");
17     function_that_can_panic();
18     if !should_continue() { return; }
19 }
20
21 fn file_example() {
22     let path = Path::new("example.txt");
23     let file = File::open(&path).unwrap()
24     function_that_can_panic();
25     if !should_continue() { return; }
26 }

```

注意，使用Rust，我们不会得到任何错误，这是因为移动语义是隐式的。

Notice that with Rust, we don't get any errors, and that's because move semantics are implicit.

- 共享所有权对比

```

class Car {};

void memory_example()
{
    shared_ptr<Car> car =
        make_shared<Car>();
    shared_ptr<Car> car2 = car;
    function_that_can_throw();
    if(!should_continue()) return;
}

void file_example()
{
    File file = File("example.txt");
    function_that_can_throw();
    if(!should_continue()) return;
}

```

```

11 struct Car {}
12
13 fn memory_example() {
14     let car = Rc::new(Car {});
15     let car2 = car.clone();
16     let my_string = String::from("LGR");
17     function_that_can_panic();
18     if !should_continue() { return; }
19 }
20
21 fn file_example() {
22     let path = Path::new("example.txt");
23     let file = File::open(&path).unwrap();
24     function_that_can_panic();
25     if !should_continue() { return; }
26 }
27
28

```

在RC智能指针上调用Clone将产生一个指向相同内存位置的新智能指针。

Calling Clone on an RC SmartPointer will produce a new SmartPointer pointing to the same memory location.

Cons

Problems Ownership Solves

- Memory/Resource leaks*
- Double free
- Use after free

Tips

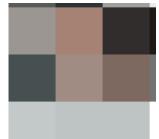
- primitives that are entirely stored on stack, 即 primitive 类型的变量赋值给其他变量 或传递到函数中不会发生move, 而是clone
- 赋值发生move: 一个变量赋值给另一个会发生所有权转移
- 所有权转移到函数: 将变量传递到函数中同样会发生所有权转移
- 所有权转移出函数: 函数返回值情况, 如函数返回 String

Borrowing

Borrowing

- The act of creating a reference
 - References are pointers with rules/restrictions
 - References do not take ownership
- Why borrow?
 - Performance
 - When ownership is not needed/desired

Borrowing Rules



- Rules
 - At any given time, you can have either one mutable reference or any number of immutable references.
 - References must always be valid.
- Problems these rules solve
 - Data races
 - Dangling references

