Memory Models

Gunnar Bergmann

30. November 2015

Table of contents

1. Manual memory management
2. RAII
3. Smart Pointers
4. Rust's Ownership
5. Conclusion

Manual memory management

Problems caused by manual memory management:

- Memory leaks
- use after free
- double delete
- repeated resource release on all paths
- separation of allocation and release
- exceptions only with finally

Garbage Collection

- memory overhead
- unpredictable collection cycle
- restricted to memory:
  - no deterministic destruction order
  - no immediate destruction

$\Longrightarrow$ Not always useful

Local variables

- memory automatically reclaimed after end of scope
- stack allocated
- returned by copying the value
- pointers become invalid after end of scope

RAII

- Classes contain constructor and destructor
- Constructor allocates resource
- Destructor frees resource
- *Resource Acquisition Is Initialization*

```
class String {
private:
    char* data; // pointer to a character
public:
    // Constructor
    String(const char* s) {
        data = new char[strlen(s)+1];
        strcpy(data, s);
    }

    // disable copying
    String(const String&) = delete;

    // Destructor
    ~String() {
        delete[] data;
    }
};
```

Add a member function for appending strings:

```
concat(const char* s) {
    char* old = data;
    int len = strlen(old)+strlen(s)+1;
    data = new char[len]; // more memory
    strcpy(data, old);
    strcat(data, s);
    delete[] old;     // free old memory
}
```

- Automatic destruction at end of lifetime
- Destructors of members and base classes automatically called
  $\Longrightarrow$ simple composition of classes
- Immediate destructor call
- Allows other resources than memory:

```
{
    lock_guard<mutex> guard(some_mutex);
    // ... code inside the mutex
} // some_mutex automatically unlocked
```

Destruction Order

- Destroy local variables at end of scope
- First run code of the destructor
  then destroy members and base classes
- Reverse of construction order
- *Stack unwind* on raised exception:
  Go back the call stack and destroy all objects.

Already solved problems:

- Memory leaks
- double delete
- repeated resource release on all paths
- separation of allocation and release
- exceptions only with finally

Remaining and new problems:

- Use after free
- Strict hierarchies required; no cyclic references

Containers

- can store multiple objects of the same type
- use RAII to call all destructors
- can move objects internally
  $\Longrightarrow$ can cause dangling references / iterators

Smart Pointers

Definition (Owning pointer)

Pointer that may need to free its memory.

Definition (Raw pointer)

Pointer like in C, for example *int**.

Definition (Smart pointer)

Pointer-like object that manages its own memory.

Unique Pointer

C++: `unique_ptr`
Rust: `Box`

- unique ownership
- hold a reference
- automatically deallocate it

Shared Pointer

C++: `shared_ptr`
Rust: `Rc`

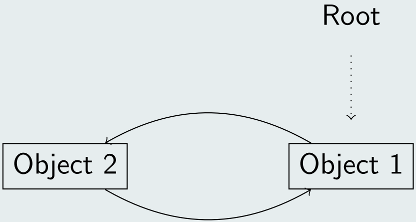
- shared ownership
- uses reference counting
- increases counter in copy operation
- decreases counter in destructor and maybe destroys object
- reference cycles can cause leaks

Weak Pointer

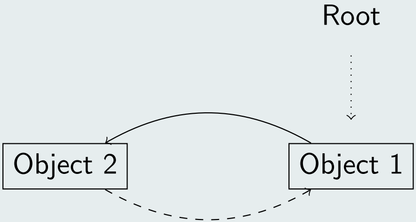
C++: `weak_ptr`
Rust: `Weak`

- no ownership, pointer can dangle
- can be upgraded to a shared pointer
- used to break reference cycles

without weak pointer



with weak pointer



RAII in garbage collected languages

- traditionally gc'ed languages use finally
- finally is more verbose than RAII and can be forgotten easily
- D uses garbage collector for memory and RAII for other resources
- Some languages provide RAII-like behavior

RAII in garbage collected languages

For example Python:

```
with open(”test.file”) as f:
    content = f.read()
```

Rust

Rust aims at memory safety, abstractions without overhead and multithreading.

- detect problems at compile time
- no dangling references
- thread safety
- no overhead

It uses ownership, borrowing and lifetimes.

Ownership

- only a single variable can access a value
- ownership can be transferred to another variable, e.g. inside a function
- can not use a variable after move
- some exceptions when transferring and copying is equal

Ownership

```
let a = vec![1, 2, 3];
let b = a;

// does not work
a.push(4); // append 4
```

Borrowing

- temporarily borrow instead of transferring ownership
- like references
- move borrowed value is forbidden (no dangling references)
- similar to read-write-lock
- just one reference can mutate the variable
  $\Longrightarrow$ no dangling iterators

Borrowing

```
fn foo() -> &i32 {
    let a = 12;
    return &a;
}
```

```
let x = vec![1,2,3];
let reference = &mut x;
println!("{}", x);
```

Borrowing is not threadsafe but can be used for it:

```
{
    let guard = mutex.lock().unwrap();
    modify_value(&mut guard);
}
```

Lifetimes

- mechanism for implementing ownership and borrowing
- every type has a lifetime
- can not take a reference with a larger lifetimes
- object can not be moved while a reference exists
- often they can be automatically generated

Lifetimes

```
struct Foo {
    x: i32,
}

// the lifetime 'a is a generic parameter
// it returns a reference with the same
// livetime as the input
fn first_x<'a>(first: &'a Foo, second: &Foo)
     -> &'a i32 {
    &first.x
}
```

Anonymous functions

▸ skip

- lambda functions
- can access local variable
- use ownership model for checks
- simplest type borrows environment
  $\Longrightarrow$ can not be returned
- second type takes ownership
  $\Longrightarrow$ variables can not be used from outside

Anonymous functions

Some can be called only once.

- ownership is transferred into the called functions
- destroyed at end of call
- function can consume data
- same concept available for methods

Anonymous functions in C++

- no checks by the compiler
- individual capture per value
- allows more complex expressions to be captured (C++14)
- more flexibility, less safety
- no self consuming functions in C++

Anonymous functions

```
let f = move || {
    // ...
};
```

```
fn do_something(self) {
// ...
} // self is destroyed
```

```
[ x,  // by value
  &r, // by reference
  p = make_unique<int>(0)
      // generalized capture
] (auto parameter1) {
    // the code in the function
}
```

Ownership in C++

- ownership was already used for reasoning about code
- just the checks are new
- At CppCon 2015 a tool for checks similar to Rusts' was announced.

Conclusion

RAII:

- safe building blocks for resource handling
- good program structure
- dangling references
- manually break cyclic references (not always possible)
- need to choose adequate type

RAII is general solution for resources without cycles

$\Longleftrightarrow$

GC is general solution for memory handling

Conclusion

Ownership:

- prevents dangling references
- improves safety
- does not solve all issues
- requires lots of annotations

Thank you for listening.
Questions?