Intro to Memory Management

计算机系统导论 (Class 9)

老师: 汪小林
助教: 陈东武

北京大学 信息科学技术学院

2024 年 12 月 04 日

Overview

Manual Management

- Memory management is a critical aspect of programming that involves the allocation, use, and deallocation of memory in software.
- Different programming languages provide diverse approaches to memory management, balancing ease of use, performance, and control.

Manual Management

- Memory management is a critical aspect of programming that involves the allocation, use, and deallocation of memory in software.
- Different programming languages provide diverse approaches to memory management, balancing ease of use, performance, and control.
- Some languages use explicit control, requiring careful planning to avoid memory leaks and dangling pointers.
  - **C**: Memory is managed using functions like `malloc`, `calloc`, and `free`.
  - Developers are responsible for releasing memory.

Manual Management

- Memory management is a critical aspect of programming that involves the allocation, use, and deallocation of memory in software.
- Different programming languages provide diverse approaches to memory management, balancing ease of use, performance, and control.
- Some languages use explicit control, requiring careful planning to avoid memory leaks and dangling pointers.
  - **C**: Memory is managed using functions like `malloc`, `calloc`, and `free`.
  - Developers are responsible for releasing memory.
  - **C++**: Builds on C by introducing features like constructors and destructors,
  - but still requires explicit calls to `new` and `delete`.
  - Smart pointers provide safer alternatives.

Automatic Memory Management

- Languages with automatic memory management use mechanisms like *garbage collection* to relieve developers of manual memory handling.

Automatic Memory Management

- Languages with automatic memory management use mechanisms like *garbage collection* to relieve developers of manual memory handling.
- **Mark and Sweep** algorithm:
  - Mark: mark all reachable objects recursively.
  - Sweep: free all unreachable objects.

Automatic Memory Management

- Languages with automatic memory management use mechanisms like *garbage collection* to relieve developers of manual memory handling.
- **Mark and Sweep** algorithm:
  - Mark: mark all reachable objects recursively.
  - Sweep: free all unreachable objects.
- **Reference Counting**:
  - Maintain a counter for each referenced object.
  - Free objects when their reference count drops to zero.
  - Problem: cyclic referencing!

Automatic Memory Management

- Languages with automatic memory management use mechanisms like *garbage collection* to relieve developers of manual memory handling.
- **Mark and Sweep** algorithm:
  - Mark: mark all reachable objects recursively.
  - Sweep: free all unreachable objects.
- **Reference Counting**:
  - Maintain a counter for each referenced object.
  - Free objects when their reference count drops to zero.
  - Problem: cyclic referencing!
- **Generational GC**:
  - Divide objects into generations based on their age.
  - Collect younger generations more frequently.
  - Young objects may be promoted to older generations.

Automatic Memory Management

- **Java**: Uses a garbage collector to reclaim memory.
  - Developers allocate objects on the heap using `new`,
  - but the GC automatically deallocates unreachable objects.

Automatic Memory Management

- **Java**: Uses a garbage collector to reclaim memory.
  - Developers allocate objects on the heap using `new`,
  - but the GC automatically deallocates unreachable objects.
- **Python**: Relies on reference counting and garbage collection.
  - It frees objects when their reference count drops to zero,
  - and periodically handles cycles using GC.

Deterministic Management

- Some languages use deterministic strategies, where memory is deallocated predictably at a certain scope or point in execution.
- **Swift**: Uses Automatic Reference Counting (ARC) to manage memory.
  - Objects are deallocated when their reference count drops to zero,
  - similar to reference counting in Python, but with stricter guarantees.

Deterministic Management

- Some languages use deterministic strategies, where memory is deallocated predictably at a certain scope or point in execution.
- **Swift**: Uses Automatic Reference Counting (ARC) to manage memory.
  - Objects are deallocated when their reference count drops to zero,
  - similar to reference counting in Python, but with stricter guarantees.
- **Rust**: Employs *ownership*, *borrowing*, and *lifetimes* at compile time,
  - ensuring safety without a garbage collector.
  - Memory is freed automatically when a variable goes out of scope.

Rust Safety Rules

Ownership

- In Rust, each value has a variable that owns it.
- When the owner goes out of scope, the value is dropped.

  -
    ```rust
    {
      let x = String::from("Hello");
    } // `x` is dropped here.
    ```

Ownership

- In Rust, each value has a variable that owns it.
- When the owner goes out of scope, the value is dropped.

  -
    ```rust
    {
      let x = String::from("Hello");
    } // `x` is dropped here.
    ```
- Ownership can be transferred using `move` semantics.

  -
    ```rust
    let x = String::from("Hello");
    let y = x; // Ownership moves to `y`.
    // println!("{}", x); // Error: `x` is no longer valid.
    ```

Ownership

- In Rust, each value has a variable that owns it.
- When the owner goes out of scope, the value is dropped.

  -
    ```rust
    {
      let x = String::from("Hello");
    } // `x` is dropped here.
    ```
- Ownership can be transferred using `move` semantics.

  -
    ```rust
    let x = String::from("Hello");
    let y = x; // Ownership moves to `y`.
    // println!("{}", x); // Error: `x` is no longer valid.
    ```
- If you want to keep both variables, you can explicitly clone the value.

  -
    ```rust
    let x = String::from("Hello");
    let y = x.clone(); // Deep copy.
    println!("{}", x); // Works fine.
    ```

Borrowing

- Borrowing allows a variable to let others temporarily access its value without transferring ownership.
- Rust ensures borrowing rules prevent unsafe memory access.
- **Immutable Borrowing:**

  -
    ```rust
    let s = String::from("Hello");
    let len = calculate_length(&s); // Borrow `s` immutably.
    println!("Length is {}", len); // `s` can still be used here.
    fn calculate_length(s: &String) -> usize { s.len() }
    ```
- **Borrowing Rules Enforcement:**

  -
    ```rust
    let mut s = String::from("Hello");
    let r1 = &s; // Immutable borrow.
    let r2 = &s; // Another immutable borrow.
    // let r3 = &mut s;
    // Error: Cannot borrow as mutable while immutably borrowed.
    ```

#thanks