Region-based Memory Management

Advanced Operating Systems
Lecture 10
Lecture Outline

• Rationale

• Stack-based memory management

• Region-based memory management
  • Ownership
  • Borrowing
  • Benefits and limitations
Rationale

• Garbage collection tends to have unpredictable timing and high memory overhead
  • Real-time collectors exist, but are uncommon and have significant design implications for applications using them

• Manual memory management is too error prone

• Region-based memory management aims for a middle ground between the two approaches
  • Safe, predictable timing
  • Limited impact on application design
Stack-based Memory Management

• Automatic allocation/deallocation of variables on the stack is common and efficient:

```c
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

static double pi = 3.14159;

static double conic_area(double w, double h) {
    double r = w / 2.0;
    double a = pi * r * (r + sqrt(h*h + r*r));

    return a;
}

int main() {
    double width  = 3;
    double height = 2;
    double area   = conic_area(width, height);

    printf("area of cone = %f\n", area);

    return 0;
}
```

Global variables

- `pi = 3.14159`
  - `double pi = 3.14159`

Stack frame for `main()`

- `double width = ...`
- `double height = ...`
- `double area = ...`

Stack frame for `conic_area()`

- `double w = ...`
- `double h = ...`
- `double r = ...`
- `double a = ...`
Stack-based Memory Management

• Hierarchy of regions corresponding to call stack:
  • Global variables
  • Local variables in each function
  • Lexically scoped variables within functions

```c
double vector_avg(double *vec, int len) {
    double sum = 0;
    for (int i = 0; i < len; i++) {
        sum += vec[i];
    }
    return sum / len;
}
```

• Variables live within regions, and are deallocated at end of region scope
Stack-based Memory Management

• Limitation: requires data to be allocated on stack

• Example:

```c
int hostname_matches(char *requested, char *host, char *domain) {
    char *tmp = malloc(strlen(host) + strlen(domain) + 2);
    sprintf(tmp, "%s.%s", host, domain);
    if (strcmp(requested, host) == 0) {
        return 1;
    }
    if (strcmp(requested, tmp) == 0) {
        return 1;
    }
    return 0;
}
```

The local variable `tmp` (pointer of type `char *`) is freed when the function returns; the allocated memory is not freed

• Heap storage has to be managed manually
Region-based Memory Allocation

- Stack allocation effective within a narrow domain – can we extend the ideas to manage the heap?
  - Create pointers to heap allocated memory
  - Pointers are stored on the stack and have lifetime matching the stack frame – pointers have type `Box<T>` for a pointer to heap allocated `T`
  - The heap allocation has lifetime matching that of the `Box` – when the `Box` goes out of scope, the heap memory it references is freed
    - i.e., the destructor of the `Box<T>` frees the heap allocated `T`

- Efficient, but loses generality of heap allocation since ties heap allocation to stack frames
Region-based Memory Management

- For effective region-based memory management:
  - Allocate objects with lifetimes corresponding to regions
  - Track object ownership, and *changes of ownership*:
    - What region owns each object at any time
    - Ownership of objects can move between regions
  - Deallocate objects at the end of the lifetime of their owning region
    - Use scoping rules to ensure objects are not referenced after deallocation

- Example: the Rust programming language
  - Builds on previous research with Cyclone language (AT&T/Cornell)
  - Somewhat similar ideas in Microsoft’s Singularity operating system
Returning Ownership of Data

- Returning data from a function causes it to outlive the region in which it was created:

```rust
const PI: f64 = 3.14159;

fn area_of_cone(w : f64, h : f64) -> f64 {
    let r = w / 2.0;
    let a = PI * r * (r + (h*h + r*r).sqrt());
    return a;
}

fn main() {
    let width  = 3.0;
    let height = 2.0;
    let area = area_of_cone(width, height);
    println!("area = ", area);
}
```
Returning Ownership of Data

- Runtime must track changes in ownership as data is returned
  - Copies made of stack-allocated local variables; original deallocated, copy has lifetime of stack-allocated local variable in calling function
  - Allows us to return a copy of a `Box<T>` that references a heap allocated value of type `T`
    - Effective with a reference counting implementation
    - Creating the new `Box<T>` temporarily increases the reference count on the heap-allocated `T`
    - The original box is then immediately deallocated, reducing the reference count again
    - (An optimised runtime can eliminate the changes to the reference count)
    - The heap-allocated `T` is deallocated when the box goes out of scope of the outer region

- Allows data to be passed around, if it always has a single owner
Giving Ownership of Data

Ownership of parameters passed to a function is transferred to that function

- DEALLOCATED WHEN FUNCTION ENDS, UNLESS IT RETURNS THE DATA
- DATA CANNOT BE LATER USED BY THE CALLING FUNCTION – ENFORCED AT COMPILATION TIME

% cat consume.rs
fn consume(mut x : Vec<u32>) {
    x.push(1);
}

fn main() {
    let mut a = Vec::new();
a.push(1);
a.push(2);
consume(a);
    println!("a.len() = ", a.len());
}

% rustc consume.rs
consume.rs:15 println!("a.len() = ", a.len());
Borrowing Data

% cat borrow.rs
fn borrow(mut x : &mut Vec<u32>) {
    x.push(1);
}

fn main() {
    let mut a = &mut Vec::new();

    a.push(1);
    a.push(2);

    borrow(a);

    println!("a.len() = {}", a.len());
}

% rustc borrow.rs
% ./borrow
a.len() = 3
%

• Functions can borrow references to data owned by an enclosing scope
  • Does not transfer ownership of the data
  • Naïvely safe to use, since live longer than the function

• Can also return references to input parameters passed as references
  • Safe, since these references must live longer than the function
Problems with Naïve Borrowing

- The `borrow()` function changes the contents of the vector
- But – it cannot know whether it is safe to do so
  - In this example, it is safe
  - If `main()` was iterating over the contents of the vector, changing the contents might lead to elements being skipped or duplicated, or to a result to be calculated with inconsistent data
  - Known as *iterator invalidation*

```rust
% cat borrow.rs
fn borrow(mut x : &mut Vec<u32>) {
    x.push(1);
}

fn main() {
    let mut a = &mut Vec::new();
    a.push(1);
    a.push(2);
    borrow(a);
    println!("a.len() = {}", a.len());
}

% rustc borrow.rs
% ./borrow
a.len() = 3
%```
Safe Borrowing

- Rust has two kinds of pointer:
  - &T – a shared reference to an immutable object of type T
  - &mut T – a unique reference to a mutable object of type T

- Runtime system controls pointer ownership and use
  - An object of type T can be referenced by one or more references of type &T, or by exactly 1 reference of type &mut T, but not both
  - Cannot get an &mut T reference to data of type T that is marked as immutable

- Allows functions to safely borrow objects – without needing to give away ownership

- To change an object:
  - You either own the object, and it is not marked as immutable; or
  - You have the only &mut reference to it

- Prevents iterator invalidation
  - The iterator requires an &T reference, so other code can’t get a mutable reference to the contents to change them:

```rust
fn main() {
    let mut data = vec![1, 2, 3, 4, 5, 6];
    for x in &data {
        data.push(2 * x);
    }
}
```

fails, since push takes an &mut reference enforced at compile time
Benefits

• Type system tracks ownership, turning run-time bugs into compile-time errors:
  • Prevents memory leaks and use-after-free bugs
  • Prevents iterator invalidation
  • Prevents race conditions with multiple threads – borrowing rules prevent two threads from getting references to a mutable object
  • Efficient run-time behaviour – timing and memory usage are predictable
Limitations of Region-based Systems

- Can’t express cyclic data structures
  - E.g., can’t build a doubly linked list:

    ![Diagram of cyclic data structure].

    Can’t get mutable reference to c to add the link to d, since already referenced by b.

  - Many languages offer an escape hatch from the ownership rules to allow these data structures (e.g., raw pointers and unsafe in Rust)

- Can’t express shared ownership of mutable data
  - Usually a good thing, since avoids race conditions
  - Rust has RefCell<T> that dynamically enforces the borrowing rules (i.e., allows upgrading a shared reference to an immutable object into a unique reference to a mutable object, if it was the only such shared reference)
  - Raises a run-time exception if there could be a race condition, rather than preventing it at compile time
Limitations of Region-based Systems

- Forces programmer to consider object ownership early and explicitly
  - Generally good practice, but increases conceptual load early in design process – may hinder exploratory programming
Summary

- Region-based memory management with strong ownership and borrowing rules
- Efficient and predictable behaviour
- Strong correctness guarantees prevent many common bugs
- Constrains the type of programs that can be written

Further reading:
- You’re not expected to read/understand section 4
- Will be discussed in tutorial 5