Region-based Memory Management

- Concepts and Rationale
- Memory Management in Rust

Region-based Memory Management: Rationale

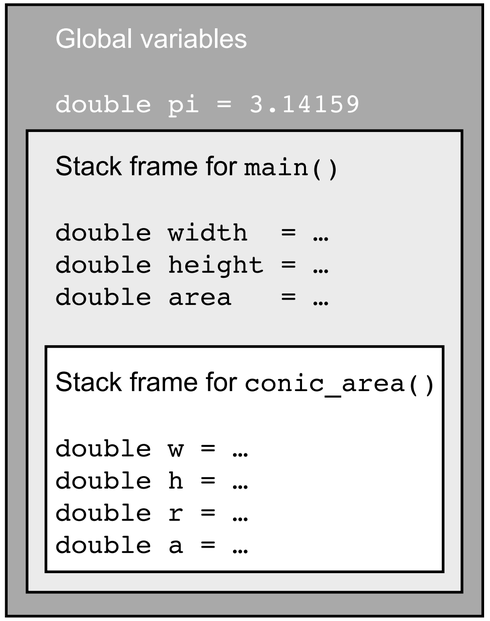
- Reference counting has relatively high overhead
  - Memory overhead to store the reference count
  - Processor time to update the reference counts
- Garbage collection has unpredictable timing, high overhead → lecture 6
- Manual memory management is too error prone

- Region-based memory management aims for middle ground:
  - Safe, predictable timing – no run-time cost
  - Accepts some impact on application design

Stack-based Memory Management

- Automatic management of stack variables common and efficient:

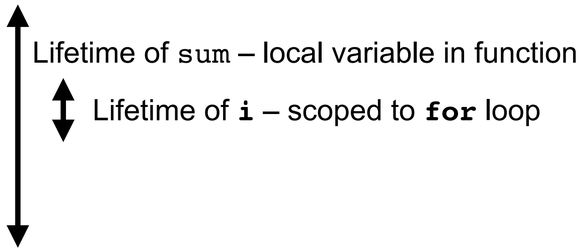
```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

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;
}
```



Stack-based Memory Management

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

```
double vector_avg(double *vec, int len) {
  double sum = 0;

  for (int i = 0; i < len; i++) {
    sum += vec[i];
  }

  return sum / len;
}
```

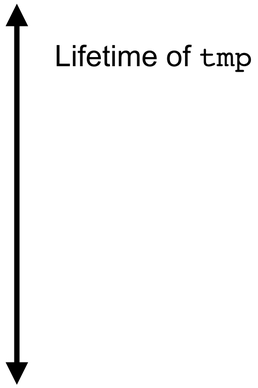


- Variables live within regions, and are deallocated at end of region scope

Stack-based Memory Management

- Limitation: requires data to be allocated on stack
  - Example:

```
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;
}
```



  - Local variable **`tmp`** stored on the stack, freed when function returns
  - Memory allocated by **`malloc()`** is not freed – memory leak

From Stack-to Region-based Memory Management

- Stack-based memory management effective, but limited applicability – can we extend to manage the heap?
  - Track lifetime of data – **values on the stack** and **references to the heap**
  - A **`Box<T>`** is a value stored on the stack that holds a reference to data of type **`T`** allocated on the heap:

```
fn main() {
    let b = Box::new(5);
    println!("b = {}", b);
}
```

`b` is a pointer to heap allocated memory, holding integer value 5

  - The **`Box<T>`** is a normal local variables with lifetime matching the stack frame
  - The heap allocated **`T`** has lifetime matching the **`Box<T>`** – when the `Box` goes out of scope, the referenced heap memory is freed
    - i.e., the destructor of the **`Box<T>`** frees the heap allocated **`T`**
    - This is RAII, to C++ programmers
  - Efficient, but loses generality of heap allocation since heap lifetime tied to stack frame lifetime

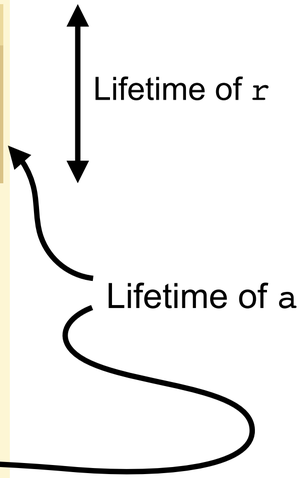
Region-based Memory Management

- 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:

```
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

- Ownership of return value is *moved* to the calling function
  - The value is moved into the calling function's stack frame
  - Original value, in the called function's stack frame, is deallocated
  - Allows us to return a **`Box<T>`** that references a heap allocated value of type **`T`**
    - The **`Box<T>`** is moved, but the referenced **`T`** on the heap is not
- Variables not returned by a function go out of scope and are reclaimed
  - The heap-allocated **`T`** is deallocated when the **`Box<T>`** goes out of scope and is reclaimed
  - i.e., the compiler generates to equivalent of a call to **`free()`** when the **`Box<T>`** goes out of scope

No Dangling References

```
fn foo() -> &i32 {
  let n = 42;
  &n
}
```

- Lifetime of local variable ends when function returns
- Can’t return a reference to an object that doesn’t exist

```
% rustc test.rs
error[E0106]: missing lifetime specifier
 --> test.rs:1:13
  |
1 | fn foo() -> &i32 {
  |             ^ expected lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but there is no
  = help: value for it to be borrowed from
```

```
int *foo() {
  int n = 42;
  return &n;
}
```

- Equivalent C code will compile but crash at runtime
  - Good compilers give a warning for many, *but not all*, cases

No Use-After-Free

```
use std::mem::drop; // free() equivalent

fn main() {
  let x = "Hello".to_string();
  drop(x);
  println!("{}", x);
}
```

- Similarly – once memory is freed, it cannot be accessed
  - Explicit **`drop()`** is equivalent of **`free()`** in C

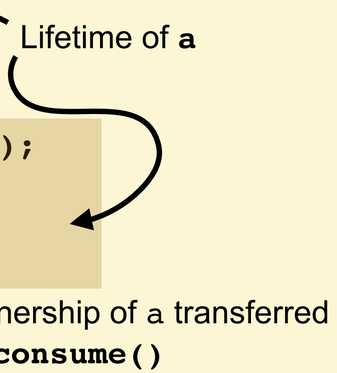
```
error[E0382]: use of moved value: `x`
 --> test.rs:6:18
  |
5 |   drop(x);
  |        - value moved here
6 |   println!("{}", x);
  |                  ^ value used here after move
  |
  = note: move occurs because `x` has type `std::string::String`, which does not implement the `Copy` trait
```

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

int main() {
  char *x = malloc(14);
  sprintf(x, "Hello, world!");
  free(x);
  printf("%s\n", x);
}
```

- Equivalent C program compiles and runs, but has undefined behaviour

Taking Ownership of Data

```
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());
}
```



- Ownership of data passed to a function is transferred to that function
  - Deallocated when function ends, unless data is returned by the function
  - Data cannot be later used by the caller function – enforced at compile time

```
% rustc consume.rs
consume.rs:15:28: 15:29 error: use of moved value: `a` [E0382]
consume.rs:15   println!("a.len() = {}", a.len());
                                         ^
```

Region-based Memory Management

- Concepts and Rationale
- Memory Management in Rust