Region-based Memory Management

- Concepts and Rationale
- Application to Rust

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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 (→ lecture 5) has unpredictable timing, high overhead
- 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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Stack-based Memory Management (1/4)

- Automatic management of **stack variables** is common and efficient:

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

static double pi = 3.14159;

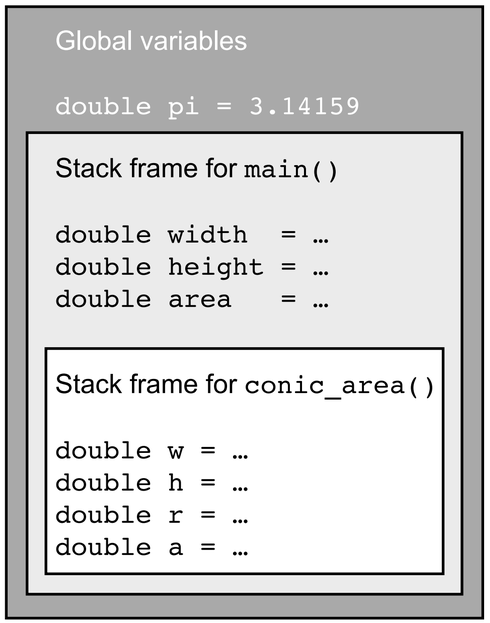
static double conic_area(double w, double h) {
  double r = w / 2.0;
  double a = pi * r * (r + sqrt(h*h + r*r));

  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;
}
```



Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

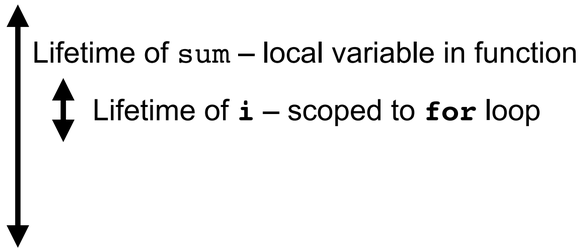
Stack-based Memory Management (2/4)

- 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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

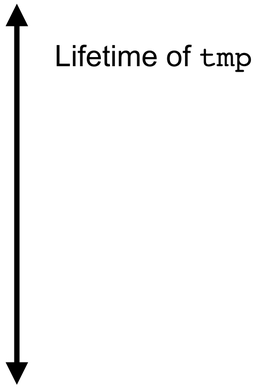
Stack-based Memory Management (3/4)

- 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

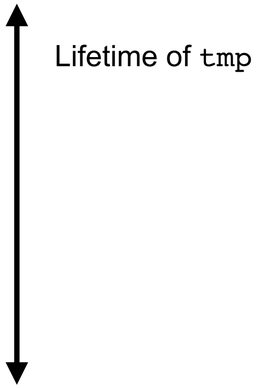
Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Stack-based Memory Management (4/4)

```
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;
}
```



- Could, in principle, track **pointers to the heap** and call **`free()`** when the pointer goes out of scope
- But how to manage returning pointers to heap allocated memory?

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Region-based Memory Management (1/4)

- Region-based memory management extends the stack-based approach to keep track of **changes in ownership** of values:

```
const PI: f64 = 3.14159;

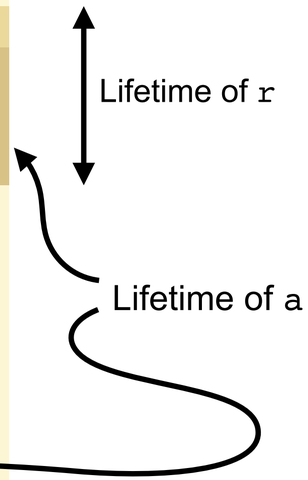
fn area_of_cone(w : f64, h : f64) -> f64 {
    let r = w / 2.0;
    let a = PI * r * (r + (h*h + r*r).sqrt());

    return a;
}

fn main() {
    let width  = 3.0;
    let height = 2.0;

    let area = area_of_cone(width, height);

    println!("area = {}", area);
}
```



The **region** in which an object is active is tracked and can extend beyond a single stack frame

Stack memory is managed as normal; object lifetimes are also tracked

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Region-based Memory Management (2/4)

- Region-based memory management extends the stack-based approach to keep track of **changes in ownership** of values and **references to the heap:**
  - Once the compiler tracks object ownership across function calls, it can track when pointers are returned from functions and free the referenced memory when the pointer is deallocated
  - In Rust, this is managed using the **`Box<T>`** type – 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 variable allocated on the stack
  - The heap allocated **`T`** has lifetime matching the **`Box<T>`** – when the `Box` goes out of scope, the referenced heap memory is freed

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Region-based Memory Management (3/4)

- This is **region-based memory management** as used in Rust:
  - Objects are stored on the stack but may reference heap allocated data via **`Box<T>`**
  - The compiler tracks object ownership and **changes of ownership:**
    - Every object has a single owner at all times
    - Ownership of objects can change as they are passed to, and returned from, functions
  - Objects are deallocated when they go out of scope
    - When a **`Box<T>`** is deallocated, the reference heap allocated object of type **`T`** is also deallocated

- The rules around ownership tracking and references in Rust enforce this region-based approach

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Region-based Memory Management (4/4)

- Region-based memory management is safe, low-overhead, has predictable timing, and no run-time costs
  - More efficient than reference counting or garbage collection
  - More predictable than garbage collection

- **But** – this comes at the cost of complexity (multiple pointer types, ownership rules) and restrictions on the data structures that can be supported (single ownership, no cyclic data structures)

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Region-based Memory Management

- Concepts and Rationale
- Memory Management in Rust

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow