

School of Computing Science



# Region-based Memory Management

Advanced Operating Systems
Lecture 6



#### **Lecture Outline**

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



#### Rationale

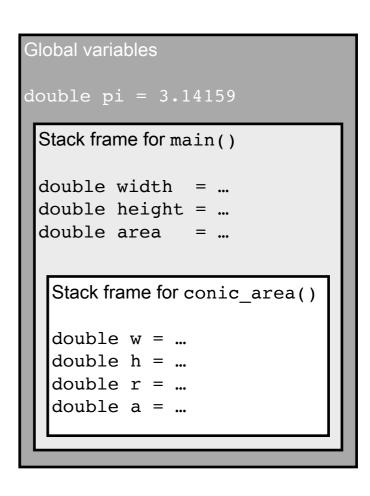
- Garbage collection tends to have unpredictable timing and high memory overhead
  - Real-time collectors exist, but are uncommon and have significant design implications for applications using them
- Manual memory management is too error prone
- Region-based memory management aims for a middle ground 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:

```
#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;
}</pre>
Lifetime of sum - local variable in function

Lifetime of i - lexically scoped
```

 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;
}
```

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
    - This is RAII, to C++ programmers
- 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:

```
Lifetime of local variables
const PI: f64 = 3.14159;
                                                             in area of cone
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() {
                                           Lifetime of a
    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 heapallocated 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

```
% cat consume.rs
fn consume(mut x : Vec<u32>) {
  x.push(1);
                       Lifetime of a
fn main() {
  let mut a = Vec::new();
  a.push(1);
  a.push(2);
                  Ownership of a transferred to consume()
  consume(a);
  println!("a.len() = {}", a.len());
% rustc consume.rs
```

- 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 compile time

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

#### **Borrowing Data**

```
% cat borrow.rs
fn borrow(mut x : &mut Vec<u32>) {
  x.push(1);
                           A mutable reference
fn main() {
  let mut a = Vec::new();
  a.push(1);
  a.push(2);
  borrow(&mut a);
  println!("a.len() = {}", a.len());
% rustc borrow.rs
% ./borrow
a.len() = 3
કૃ
```

- 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

```
% cat borrow.rs
fn borrow(mut x : &mut Vec<u32>) {
  x.push(1);
fn main() {
  let mut a = Vec::new();
  a.push(1);
  a.push(2);
  borrow(&mut a);
 println!("a.len() = {}", a.len());
% rustc borrow.rs
% ./borrow
a.len() = 3
엉
```

- 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

# 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:

```
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 compiletime 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:



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:
  - D. Grossman et al., "Region-based memory management in Cyclone", Proc. ACM PLDI, Berlin, Germany, June 2002. DOI:10.1145/512529.512563
  - You are not expected to read/understand section 4
  - What was Cyclone? Did the project's goals make sense?
  - How does the region-based memory management system described differ from that outlined in the lecture?
  - Interactions with the garbage collector?
  - Other features added to C?
  - Ease of porting C code? Performance?
  - Does it make sense to try to extend C with region-based memory management?

#### Region-Based Memory Management in Cyclone \*

Dan Grossman Greg Morrisett Trevor Jimi Michael Hicks Yanling Wang James Chene

omputer Science Department ornell University aca, NY 14853

<sup>†</sup>AT&T Labs Research 180 Park Avenue Florham Park, NJ 07932 trevor@research.att.co

#### ABSTRA

Cyclone is a type-safe programming language derived from C. The primary design goal of Cyclone is to let program mers control data representation and memory managemen without sacrificing type-safety. In this paper, we focus on the region-based memory management of Cyclone and it static typing discipline. The design incorporates several advancements, including support for region subtyping and a vancements, including support for region subtyping and coherent integration with stack allocation and a garbage collector. To support separate compilation, Cyclone require programmers to write some explicit region annotations, bu a combination of default annotations, local type inference and a novel treatment of region effects reduces this burden As a result, we integrate C idioms in a region-based frame work. In our experience, porting legacy C to Cyclone ha required altering about 5% of the code, of the changes, onl-6% (of the 8%) were region annotations.

#### Categories and Subject Descriptors

D.3.3 [Programming Languages]: Language Constr and Features—dynamic storage management

#### Genera

Languages

#### 1. INTRODUCTION

Many software systems, including operating systems, de vice drivers, file servers, and databases require fine-grained

"This research was supported in part by Sloan grant BR 3734: NSF grant 987536: AFOSR grant 879620-01-0198, F49629-01-1098, F49629-01-1-0298, F49629-01-1-0298, P49629-01-1-0299, and F49629-01-1-0312; ONR grant NO0014-01-1-0908; and NSF Graduat Fellowships. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not reflect the views of these agencies.

reminsion of mane ougan to make open of an or part of this work.

personal or classroom use is granted without fee provided that copies
personal or classroom the feet of the copies
bear this notice and the full citation on the first page. To copy otherwise
republish, to post on servers or tredistribute to lists, requires prior spec
permission and/or a fee.

PLDTO2\_How IT-119\_2002, Berlin, Germany.

control over data representation (e.g., field layout) and to source management (e.g., memory management). The defacts language for coding such systems is C. However, it providing low-level control, C. admits a wide class of danger ons — and extremely common — safety violations, such a incorrect type casts, buffer overrus, dangling-pointer deceerences, and space leaks. As a result, building large system in C. especially ones including third-party extensions, is per ilous. Higher-level, type-sell languages awould these draw the control needed in low-level systems. Moroover, portin or extending legacy code is often prohibitively expensive. Therefore, as alse language at the Clevel of abstraction, wit

Toward this end, we have developed Cyclone [6, 19], a language designed to be very close to C, but also safe. We have written or ported over 110,000 lines of Cyclone code, including the Cyclone compiler, an extensive library, lexer and parser generators, compression utilities, device drivers, and many smaller benchmarks. In the process, we identified many common C idioms that are usually safe, but which the language with modern features and types so that programmers can still use the idioms, but have safety varantees.

For example, to reduce the need for type \$2.85, Cyclans has features like parametric polymorphism, susying, and tagged unions. To prevent bounds violations without making liddlen data-representation changes, Cyclone has a variety of pointer types with different compile-time invariants and associated nun-time checks. Other projects aimed at making legacy C code safe have addressed these issues with somewhat different approaches, and discussed these issues with somewhat different approaches, and discussed these issues with somewhat different approaches, and discussed these issues with concern the control of the cont

- Sound: Programs never dereference dangling pointers
- Static: Dereferencing a dangling pointer is a compitime error. No run-time checks are needed to det mine if memory has been deallocated.
- Convenient: We minimize the need for explicit p grammer annotations while supporting many C ioms. In particular, many uses of the addresses of lo