University of Glasgow

Memory and Resource Management

Advanced Systems Programming (M)

Lecture 4

Outline

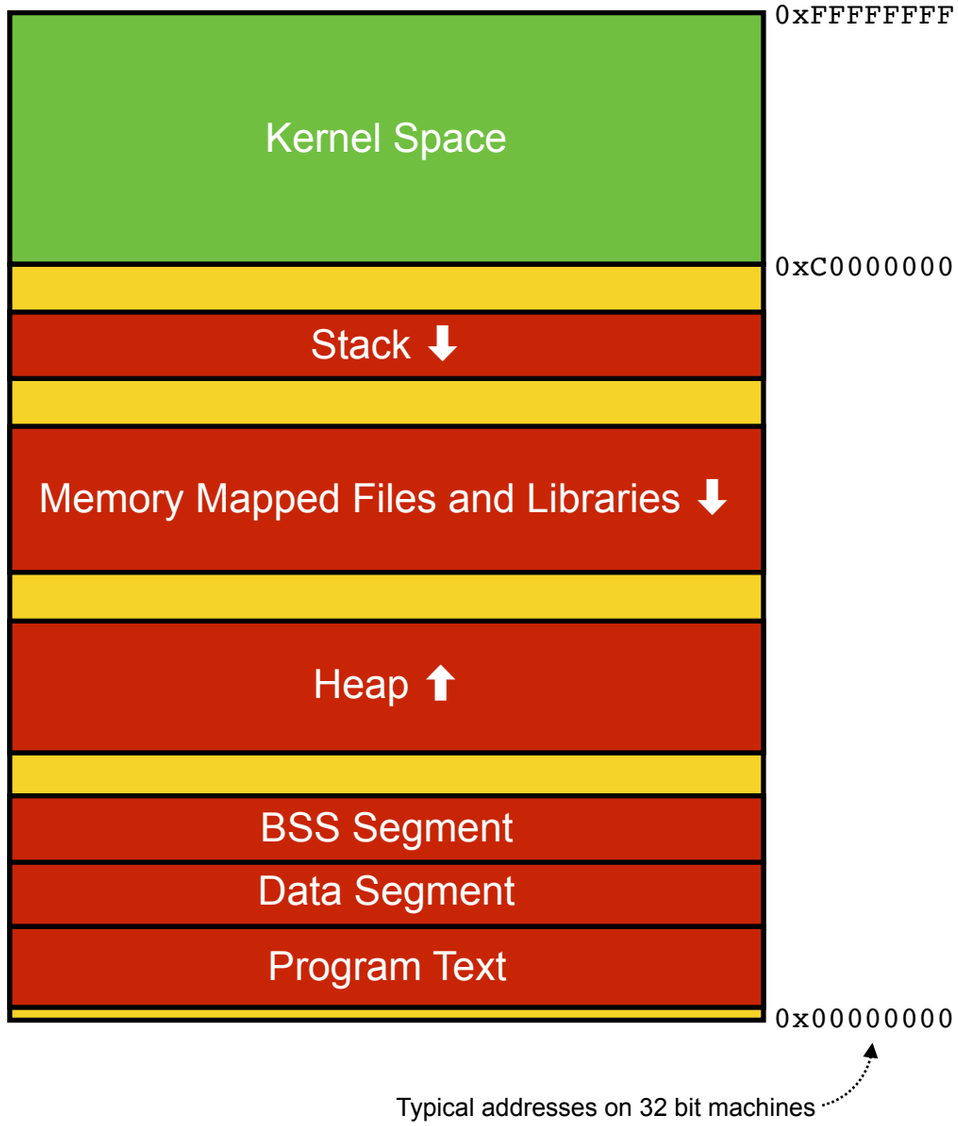
- Memory
  - How is a process stored in memory?
  - What memory has to be managed?
- Memory management
  - Reference counting
  - Region-based memory management
- Resource management

Memory

- How is a process stored in memory?
- What memory has to be managed?

Layout of a Processes in Memory

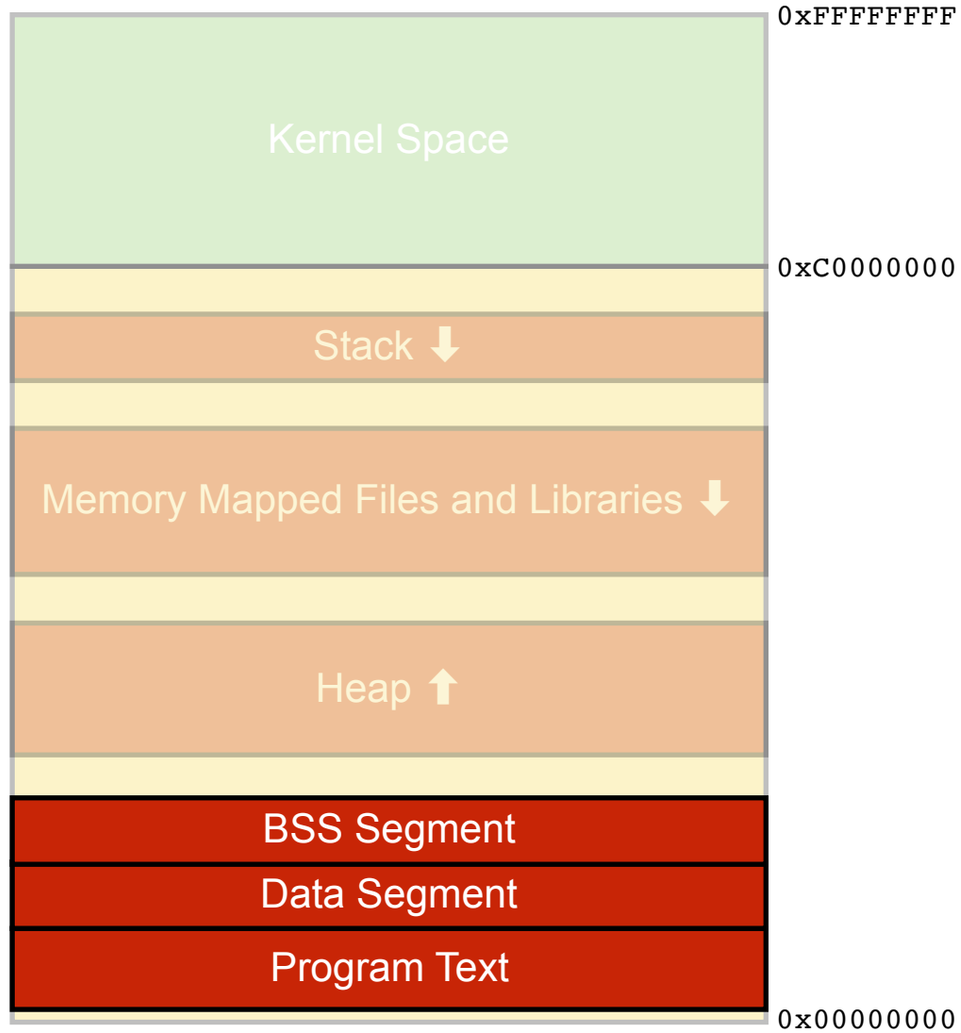
- Layout of process address space:
  - Kernel at top of address space
  - Program text, data, and global variables at bottom of virtual address space
  - Heap allocated upwards, above BSS
  - Stack grows downwards, below kernel
  - Memory mapped files and shared libraries between these



See also http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/

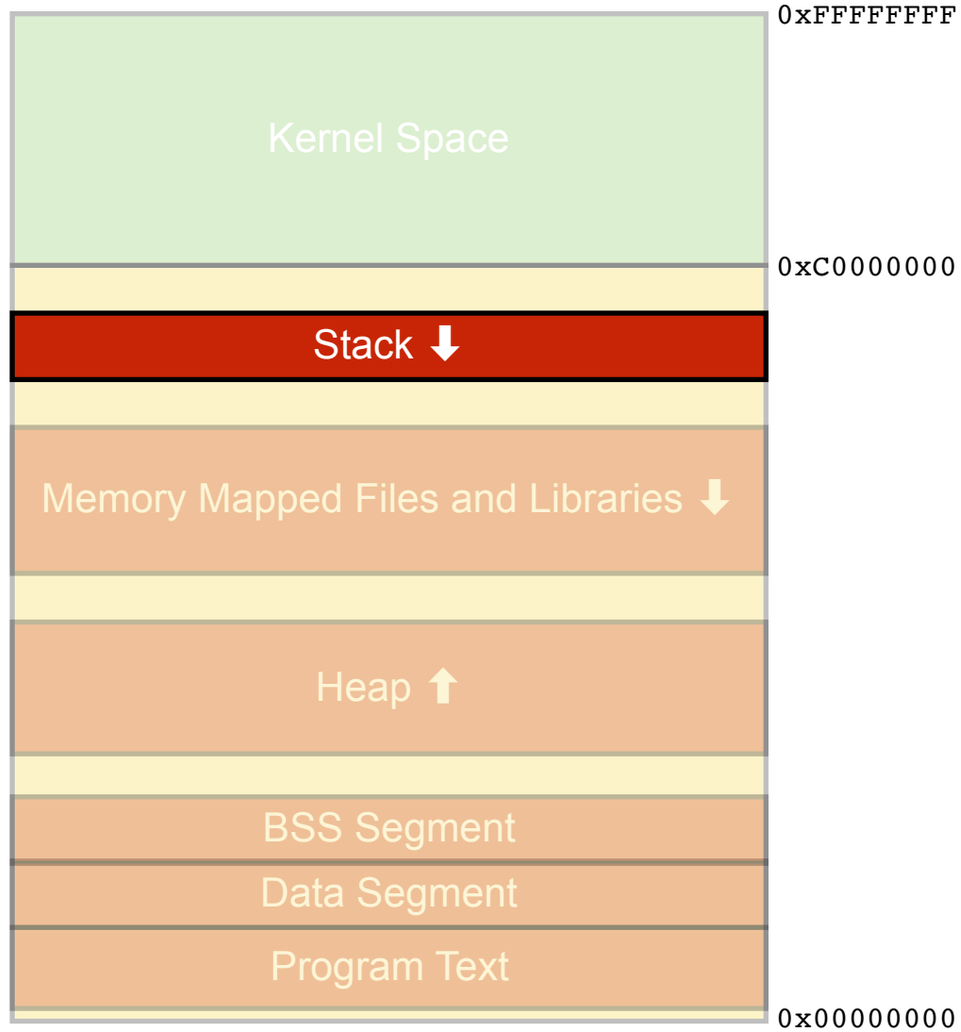
Program Text, Data, and BSS

- Program and static data occupies bottom of address space
  - Lowest few pages above address zero reserved to trap null-pointer dereferences
  - **Program Text** is compiled machine code of program
  - **Data segment** is variables initialised in source code
    - String literals, initialised **`static`** global variables in C
    - Known at compile time, loaded along with program text
- **BSS segment** is reserved space for uninitialised **`static`** global variables
  - "block started by symbol" – name is historical relic
  - Initialised to zero by runtime when the program loads



The Stack

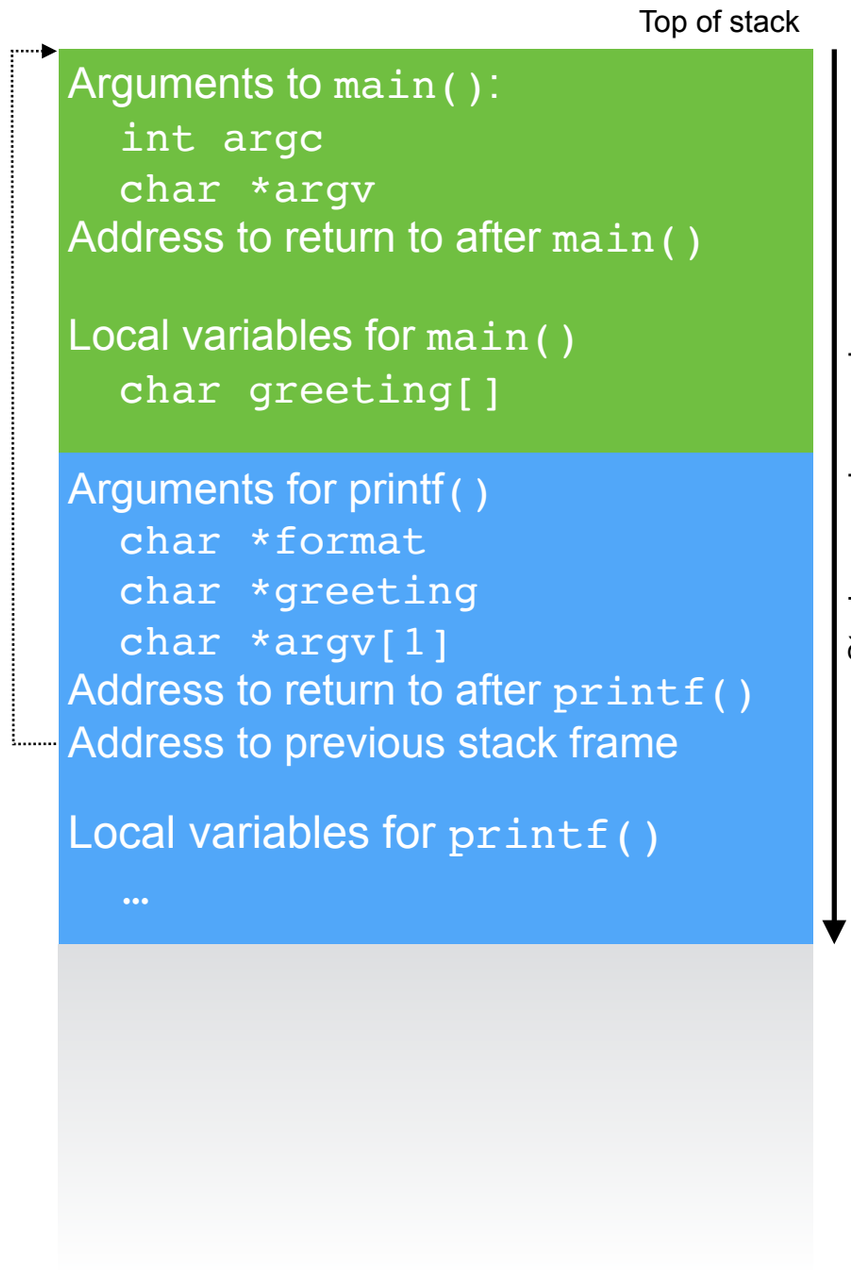
- **The stack** holds function parameters, return address, and local variables
  - Function calls push data onto stack, growing down
    - Parameters for the function; return address; pointer to previous stack frame; local variables
  - Data removed, stack shrinks, when function returns – the stack is managed automatically
    - Compiler generates code to manage the stack as part of the compiled program
    - The calling convention for functions – how parameters are pushed onto the stack – is standardised for a given processor and programming language
  - The operating system generates the stack frame for **`main()`** when the program starts
- Ownership of stack memory follows function invocation



Function Calling Conventions

- Example: code and contents of stack while calling **`printf()`** in code below:

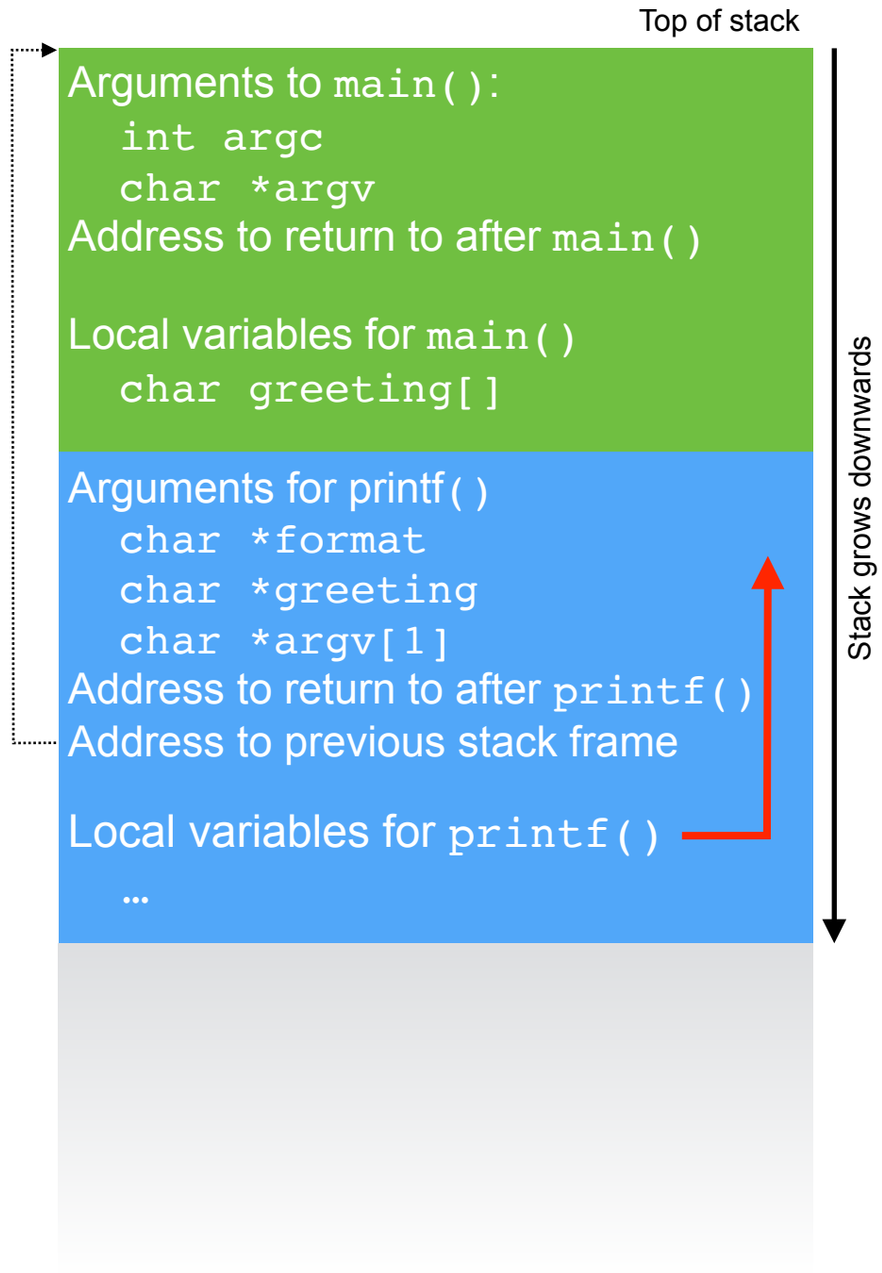
```
#include <stdio.h>

int
main(int argc, char *argv[])
{
  char greeting[] = "Hello";

  if (argc == 2) {
    printf("%s, %s\n", greeting, argv[1]);
    return 0;
  } else {
    printf("usage: %s <name>\n", argv[0]);
    return 1;
  }
}
```



- Address of the previous stack frame is stored for ease of debugging, so stack trace can be printed, so it can easily be restored when function returns

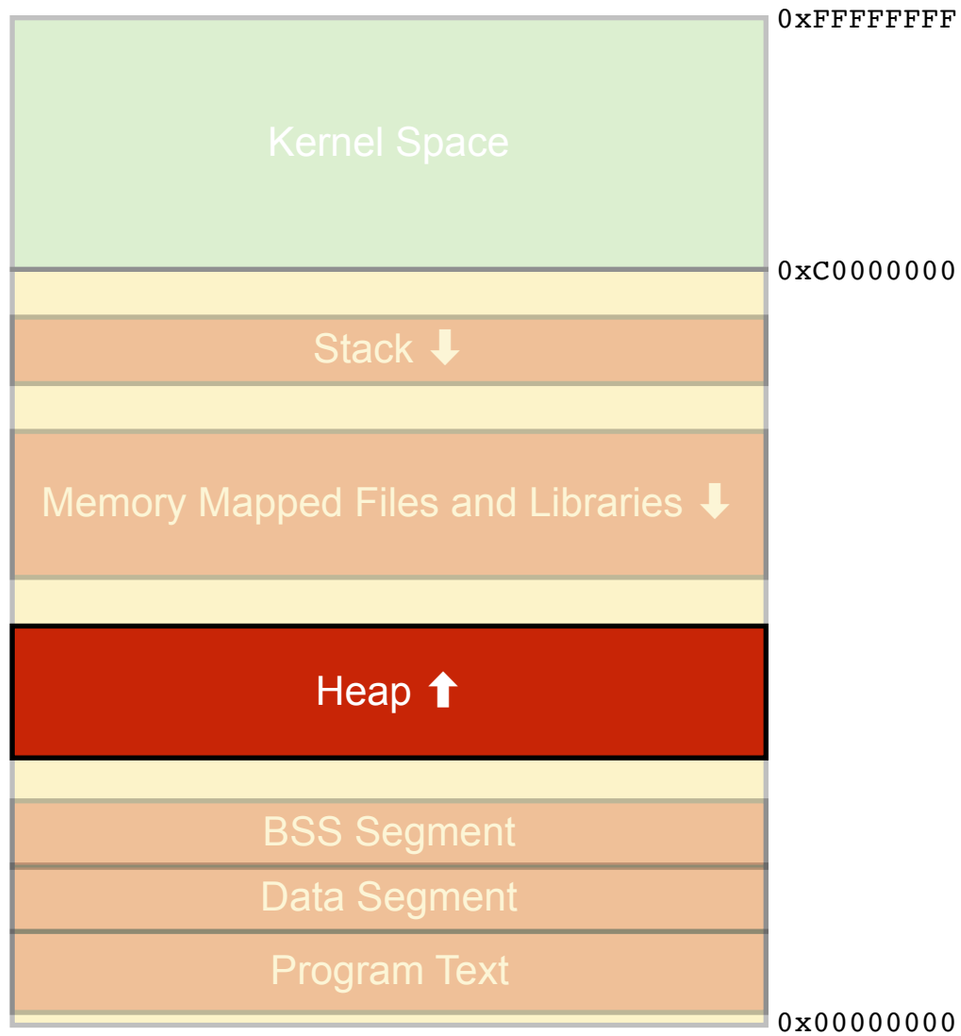
Buffer Overflow Attacks

- Classic buffer overflow attack:
  - Language not type safe, doesn't enforce abstractions
  - Write past array bounds → overflows space allocated to local variables, overwrites function return address, and following data
  - Contents valid machine code; the overwritten function return address is made to point to that code
  - When function returns, code written during overflow is executed
- Workarounds:
  - Marks stack as non-executable
  - Randomise top of stack address each program run
  - Various more complex buffer overflow attacks still possible – e.g., see "return-oriented programming"
- Solution: use a language that is type safe and enforces array bounds checks



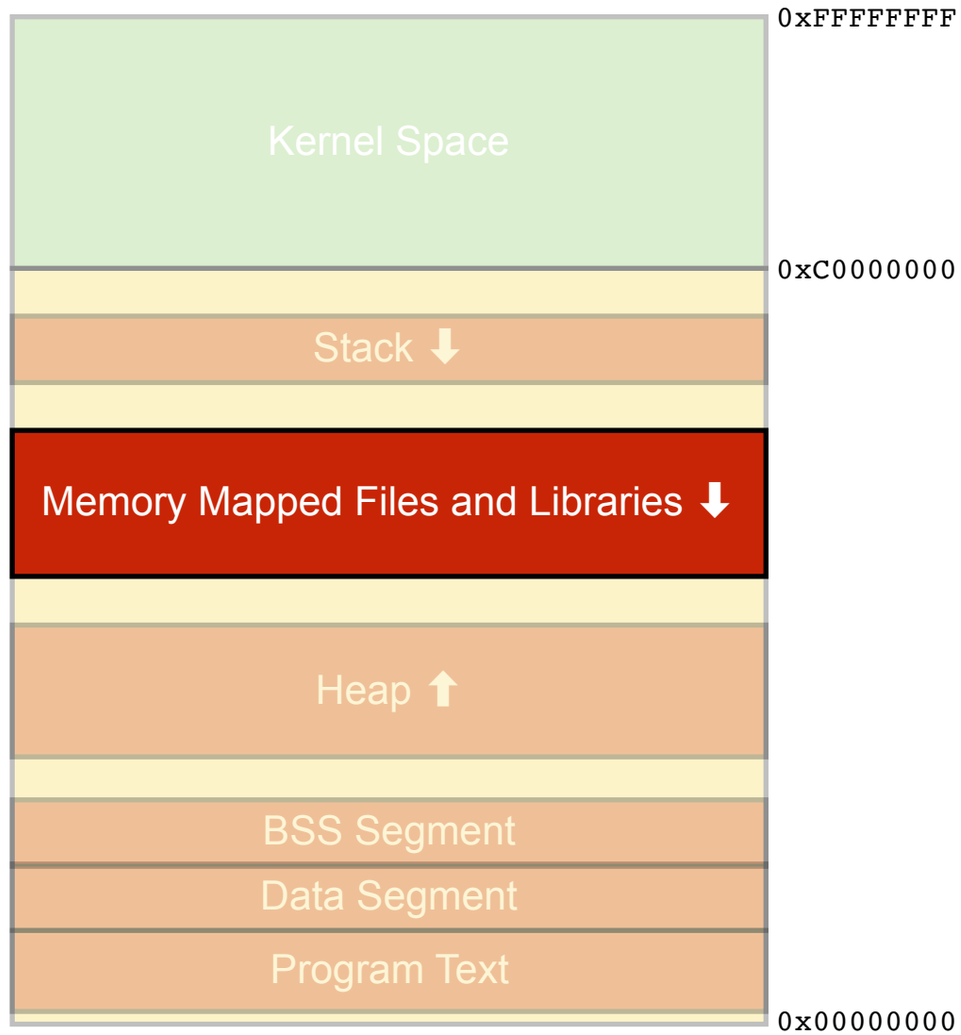
The Heap

- **The heap** holds explicitly allocated memory
  - Allocated using **`malloc()`**/**`calloc()`** in C
  - Starts at a low address in memory; later allocations follow in consecutive addresses
    - Sometimes padded to align to a 32 or 64 bit boundary, depending on processor
    - Modern **`malloc()`** implementations are thread aware, split heap into different parts different threads to avoid cache sharing
  - Memory management is primarily concerned with reclaiming heap memory
    - Manually, using **`free()`**
    - Automatically via reference counting/garbage collection
    - Automatically based on regions and lifetime analysis



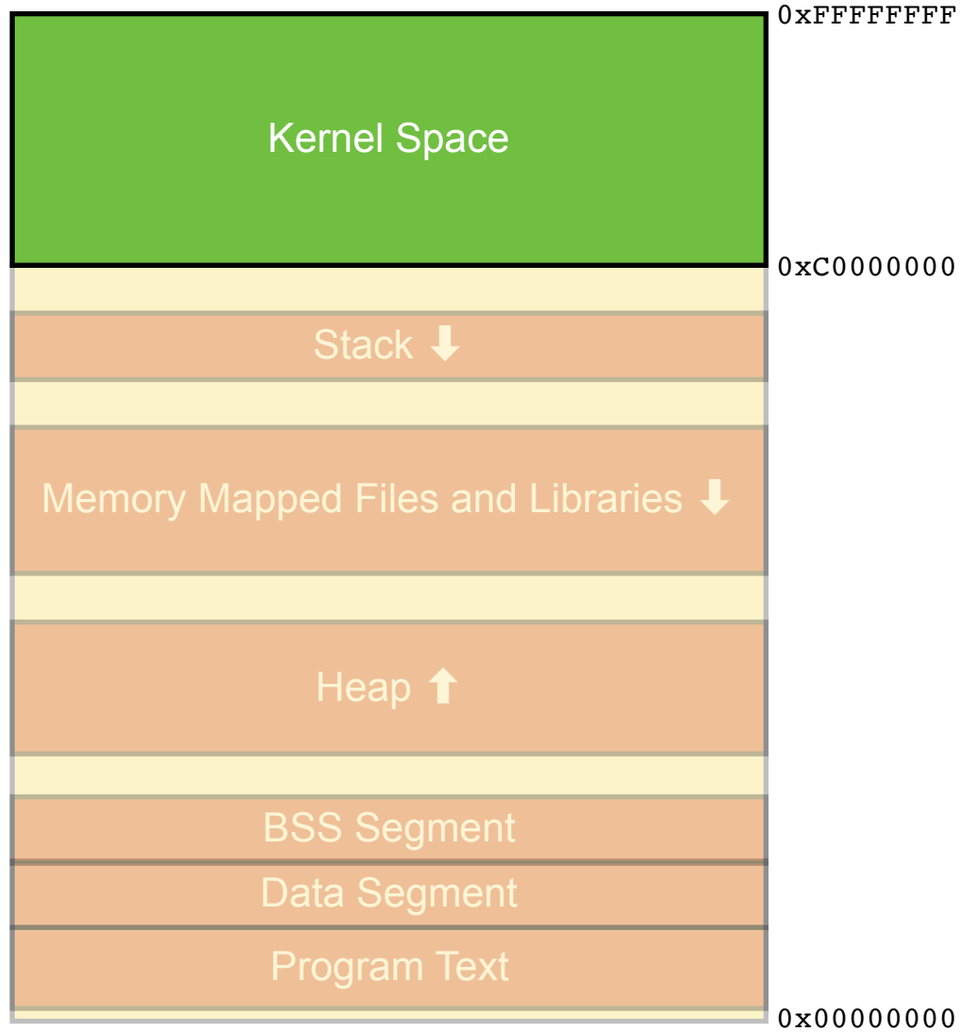
Memory Mapped Files and Shared Libraries

- **Memory mapped files** allow data on disk to be directly mapped into address space
  - Mappings created using **`mmap()`** system call
    - Returns a pointer to a memory address that acts as a proxy for the start of the file
    - Reads from/writes to subsequent addresses acts on the underlying file
    - File is demand paged from/to disk as needed – only the parts of the file that are accessed are read into memory (granularity depends on virtual memory system – often 4k pages)
    - Useful for random access to parts of files
- Used to map **shared libraries** into memory



The Kernel

- Operating system **kernel** resides at top of the address space
  - Not directly accessible to user-space programs
    - Attempt to access kernel → segmentation violation
    - The **`syscall`** instruction in x86_64 assembler calls into the kernel after permission check
  - Kernel can read/write memory of user processes



Memory Management

- Concepts
- Reference counting
- Region-based memory management

Automatic Memory Management

- Automatic memory management distrusted by systems programmers
  - Perceived high processor and memory overheads, unpredictable timing
  - But, memory management problems are common:
    - Unpredictable performance
      - Calls to **`malloc()`**/**`free()`** can vary in execution time by several orders of magnitude
    - Memory leaks
    - Memory corruption and buffer overflows
    - Use-after-free
    - Iterator invalidation
- New automatic memory management schemes solve many problems
  - Garbage collectors → lower overhead, more predictable
    - Also system performance improvements made overhead more acceptable
  - Region-based memory management → predictability, compile time guarantees

Automatic Memory Management

- Memory allocation/deallocation can be manual or automatic
  - Stack memory always managed automatically:
    - In the example, memory for **`di`** is automatically allocated when the function executes; freed on completion
    - Simple and efficient for languages like C/C++ that have complex value types
    - Useless for Java-like languages, where objects are always allocated on the heap

```
int saveDataForKey(char *key, FILE *outf)
{
    struct DataItem di;

    if (findData(&di, key)) {
        saveData(&di, outf);
        return 1;
    }
    return 0;
}
```

  - Heap memory is managed (semi-)manually
    - Allocation using, e.g., **`malloc()`**
    - Deallocation using explicit **`free()`**, automatically reclaimed when no longer referenced
      - Automatic reclamation doesn't remove need to think about object lifetime
      - Automatic reclamation doesn't prevent memory leaks

Automatic Memory Management: Managing the Heap

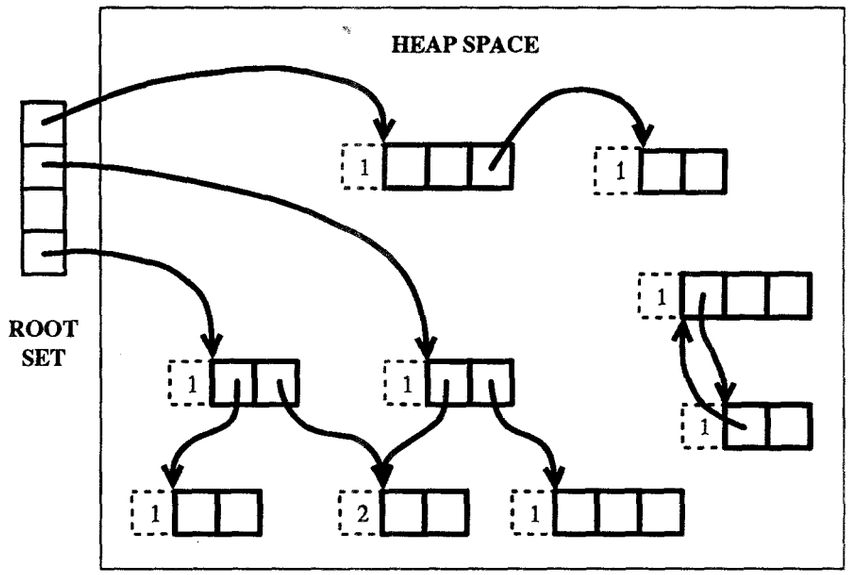
- Aim is to find objects that are no longer used, and make their space available for reuse
  - An object is no longer used (ready for reclamation) if it is not reachable by the running program via any path of pointer traversals
  - Any object that is *potentially* reachable is preserved – better to waste memory than deallocate an object that's in use

- Approaches to automatic heap management:
  - Reference counting
  - Region-based lifetime tracking
  - Garbage collection → lecture 5

Reference Counting

Reference Counting

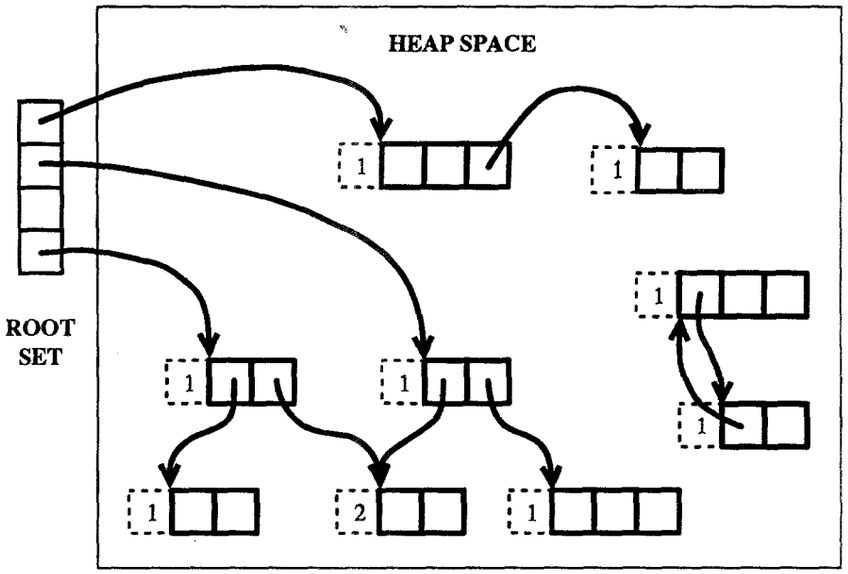
- Simplest automatic heap management
- Each allocation also allocates space for an additional **reference count**
  - An extra **`int`** is allocated along with every object
  - Counts number of references to the object
    - Increased when new reference to the object is created
    - Decremented when a reference is removed
  - When reference count reaches zero, there are no references to the object, and it may be reclaimed
    - Reclaiming object removes references to other objects
    - May reduce their reference count to zero, so triggering further reclamation



Source: P. Wilson, “Uniprocessor garbage collection techniques”, Proc IWMM’92, DOI:10.1007/BFb0017182

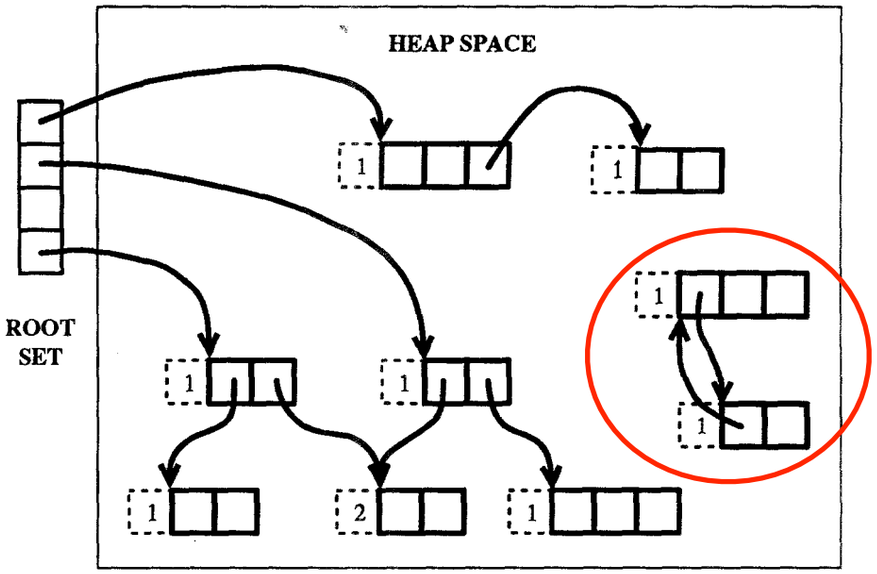
Reference Counting: Benefits

- Incremental operation – memory reclaimed in small bursts
- Predictable and understandable
  - Easy to explain
  - Easy to understand when memory is reclaimed
  - Easy to understand overheads and costs
  - Follows programmer intuition



Reference Counting: Costs

- Cyclic data structures give mutual references
  - Objects all reference each other – never reclaimed, since reference count doesn't go to zero
  - Memory leaks unless cycle explicitly broken – needs programmer action
- Stores additional **`int`** along with each object to hold the reference count
  - Maybe also a mutex if concurrent access possible
  - Per-object overhead is significant for small objects; wastes memory
- Processor cost of updating references can be significant for short-lived objects



Reference Counting

- Widely used in scripting languages
  - Python, Ruby, etc.
  - Memory and processor overhead not significant in interpreted runtime

- Used on small scale for systems programming
  - e.g., Objective C runtime on iOS
  - Ease of understanding is important
  - Tends to be for large, long-lived, data – reduces overheads
  - Not typically used in kernel code, high-performance systems

Region-based Memory Management

Region-based Memory Management: Rationale

- Reference counting has high overheads
  - Memory overhead to store the reference count
  - Processor time to update the reference counts
- Garbage collection tends to have unpredictable timing and high memory overhead

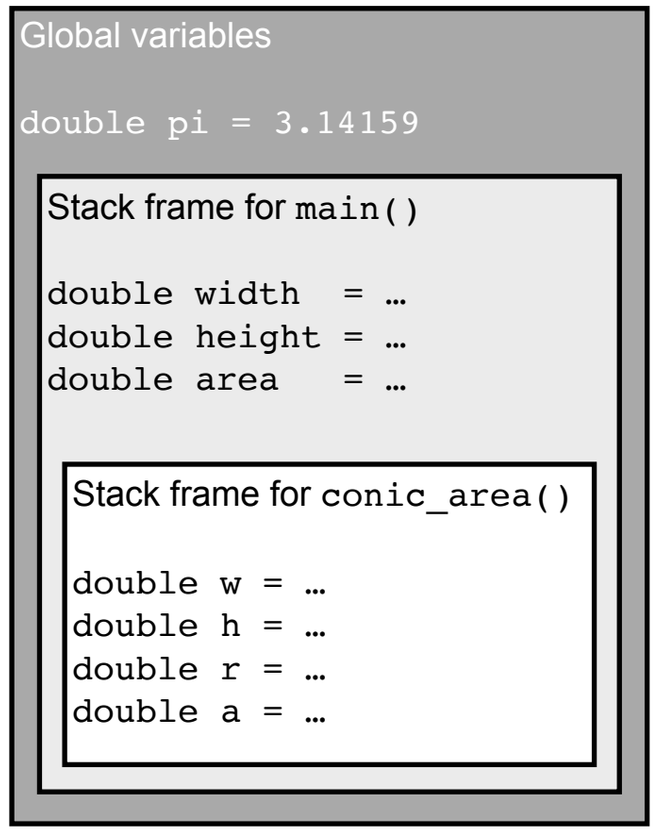
  ⟶ lecture 5
- Manual memory management is too error prone

- Region-based memory management aims for a middle ground between the these approaches
  - Safe, predictable timing – no run-time cost
  - Limited impact on application design

Stack-based Memory Management

- Automatic management of stack variable common and efficient:

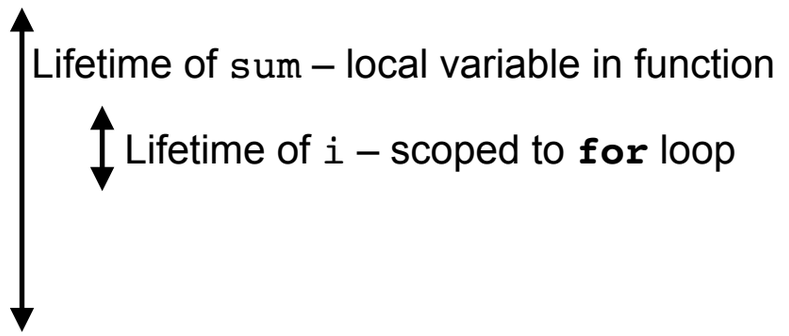
```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

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;
}
```

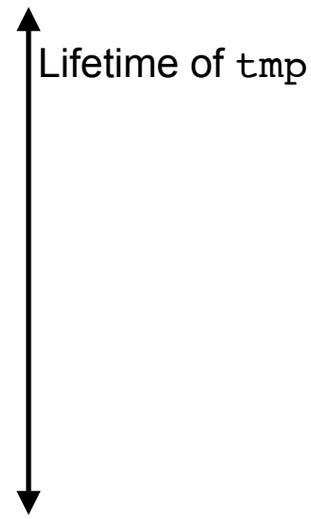


- Variables live within regions, and are deallocated at end of region scope

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);
}
```

  - i.e., it's a pointer to a **`T`**
  - 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

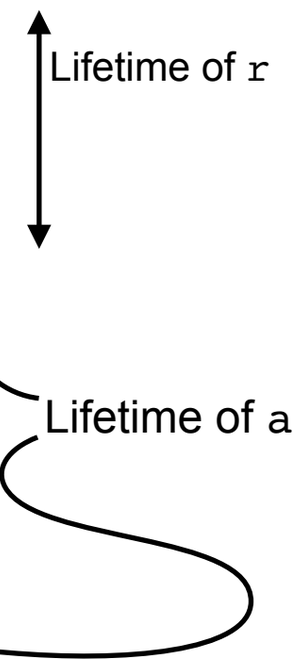
- For effective region-based memory management:
  - Allocate objects with lifetimes corresponding to regions
  - Track object ownership, and *changes of ownership:*
    - What region owns each object at any time
    - Ownership of objects can move between regions
  - Deallocate objects at the end of 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

- Compiler tracks changes in 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 copy of 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

Returning Ownership of Data: 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 value for
to be borrowed from
  = help: consider giving it a 'static lifetime
```

```
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

Returning Ownership of Data: No Use-After-Free

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

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

```
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
```

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

```
#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

Giving 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());
}
```

Lifetime of **a**

Ownership of a transferred to **consume()**

- Ownership of data 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

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

Borrowing Data

```
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
%
```

- Functions can *borrow* **references** to data
  - Does *not* move ownership of the data
  - Borrowed value not accessible by called for duration of the borrow
  - Naïvely safe to use, since borrowed data lives longer than the function
- Functions can also return references to borrowed input parameters
  - The parameters are borrowed from the calling function, so safe to return them to it

Problems with Naïve Borrowing – Iterator Invalidation

```
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
%
```

- In this example, **`borrow()`** 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 one reference of type `&mut T`, but not both
  - Cannot get an `&mut T` reference to data of type `T` that is marked as immutable (i.e., via an `&T` reference)
- 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 own 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

Iterator Invalidation: Example

```
fn push_all(from: &Vec<i32>, to: &mut Vec<i32>) {
  for elem in from.iter() {
    to.push(*elem);
  }
}

fn main() {
  let mut vec = Vec::new();
  push_all(&vec, &mut vec);
}
```

- Common bug in C++ and Java
  - Modify an iterator while iterating
  - Typically ends in null pointer deference or data corruption – follows reference to element that no longer exists
  - Does not compile in Rust, because of borrowing rules

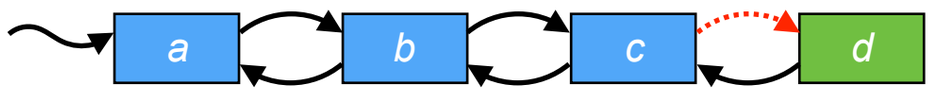
```
error[E0502]: cannot borrow `vec` as mutable because it is also borrowed as immutable
 --> test.rs:9:23
  |
9 |   push_all(&vec, &mut vec);
  |            ---       ^^^- immutable borrow ends here
  |            |         |
  |            |         mutable borrow occurs here
  |            immutable borrow occurs here

error: aborting due to previous error
```

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
  - Generates **exactly the same code** as a correctly written program using **`malloc()`** and **`free()`**
  - Timing and memory usage are as predictable as correct a C program
    - Deterministic when memory allocated
    - Deterministic when memory freed

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 consideration of object ownership early and explicitly
  - Generally good practice, but increases conceptual load early in design process – may hinder exploratory programming

Region-based Memory Management: 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 and used in Rust?
  - 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 Jim†
Michael Hicks Yanling Wang James Cheney

Computer Science Department
Cornell University
Ithaca, NY 14853
{danieljg,jgm,mhicks,wangyl,jcheney}@cs.cornell.edu

†AT&T Labs Research
180 Park Avenue
Florham Park, NJ 07932
trevor@research.att.com

ABSTRACT

Cyclone is a type-safe programming language derived from C. The primary design goal of Cyclone is to let programmers control data representation and memory management without sacrificing type-safety. In this paper, we focus on the region-based memory management of Cyclone and its static typing discipline. The design incorporates several advancements, including support for region subtyping and a coherent integration with stack allocation and a garbage collector. To support separate compilation, Cyclone requires programmers to write some explicit region annotations, but 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 framework. In our experience, porting legacy C to Cyclone has required altering about 8% of the code; of the changes, only 6% (of the 8%) were region annotations.

Categories and Subject Descriptors

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

General Terms

Languages

1. INTRODUCTION

Many software systems, including operating systems, device drivers, file servers, and databases require fine-grained control over data representation (e.g., field layout) and resource management (e.g., memory management). The *de facto* language for coding such systems is C. However, in providing low-level control, C admits a wide class of dangerous — and extremely common — safety violations, such as incorrect type casts, buffer overruns, dangling-pointer dereferences, and space leaks. As a result, building large systems in C, especially ones including third-party extensions, is perilous. Higher-level, type-safe languages avoid these drawbacks, but in so doing, they often fail to give programmers the control needed in low-level systems. Moreover, porting or extending legacy code is often prohibitively expensive. Therefore, a safe language at the C level of abstraction, with an easy porting path, would be an attractive option.

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, a multimedia distribution overlay network, a web server, and many smaller benchmarks. In the process, we identified many common C idioms that are usually safe, but which the C type system is too weak to verify. We then augmented the language with modern features and types so that programmers can still use the idioms, but have safety guarantees.

For example, to reduce the need for type casts, Cyclone has features like parametric polymorphism, subtyping, and tagged unions. To prevent bounds violations without making hidden data-representation changes, Cyclone has a variety of pointer types with different compile-time invariants and associated run-time checks. Other projects aimed at making legacy C code safe have addressed these issues with somewhat different approaches, as discussed in Section 7.

In this paper, we focus on the most novel aspect of Cyclone: its system for preventing dangling-pointer dereferences and space leaks. The design addresses several seemingly conflicting goals. Specifically, the system is:

- *Sound:* Programs never dereference dangling pointers.
- *Static:* Dereferencing a dangling pointer is a compile-time error. No run-time checks are needed to determine if memory has been deallocated.
- *Convenient:* We minimize the need for explicit programmer annotations while supporting many C idioms. In particular, many uses of the addresses of local variables require no modification.

*This research was supported in part by Sloan grant BR-3734; NSF grant 9875536; AFOSR grants F49620-00-1-0198, F49620-01-1-0298, F49620-00-1-0209, and F49620-01-1-0312; ONR grant N00014-01-1-0968; and NSF Graduate 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.

PLDI'02, June 17-19, 2002, Berlin, Germany.

Resource Management

Resource Management: Deterministic Cleanup

- Rust deterministically frees memory when data goes out of scope – known as *dropping* the data
- Types can implement the **`Drop`** trait to get custom destructors

```
impl Drop for MyType {
    fn drop(&mut self) {
        ...
    }
}
```

```
pub trait Drop {
    fn drop(&mut self);
}
```

Definition of **`std::ops::Drop`** from Rust standard library

- Dropping is deterministic → clean-up resource ownership
  - Garbage collected languages typically give no guarantee when the destructor runs
- e.g., the **`File`** class uses custom **`drop()`** implementation to close the file when it goes out of scope
- Python has special syntax for this:

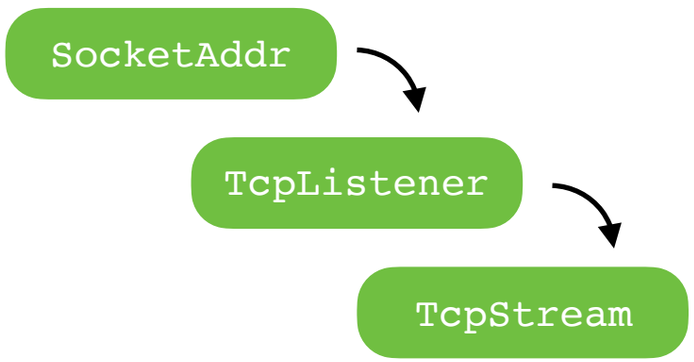
```
with open(filename) as file:
    data = file.read()
    ...
```

unnecessary in Rust – cleanup happens naturally

Resource Management: Ownership and States

- Use ownership transfer between different types to model resource states
  - **`struct`**-based state machine → lecture 3

```
let listener = TcpListener::bind(socket);
match listener.accept() {
  Ok(connection) => …
  Err(error) => …
}
```



- Manage the different states of a resource
- Make illegal operations compile time errors

See also: https://blog.systems.ethz.ch/blog/2018/a-hammer-you-can-only-hold-by-the-handle.html

Memory and Resource Management

- Memory
- Memory management
  - Reference counting
  - Lifetimes and region-based management
- Resource management