# Resource Management

- **Borrowing Memory**
- Managing Resources


Colin Perkins | https://csperkins.org/ 

# 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 take **references** to data
  - Does **not** move ownership of the data, it borrows it – moves ownership of the reference, not the referenced value
- 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 contents of 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` – shared reference to immutable object
  - `&mut T` – unique reference to mutable object
- The compiler and runtime control reference 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
  - Existence of an `&T` reference to mutable data makes the data 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 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*

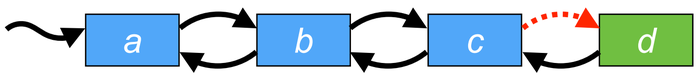
  - Enforced by the compiler

# Benefits

- Type system tracks ownership, turning run-time bugs into compile-time errors:
  - Prevents 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 a correct 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 in safe Rust:




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)
  - Fails a run-time exception if there could be a race, 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

# Resource Management

- Borrowing Memory
- **Managing Resources**

# Resource Management: Deterministic Cleanup

- Rust deterministically frees (“drops”) memory when reference goes out of scope
- Types can implement **`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`**

  - 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 4

```
impl UnauthenticatedConnection {
  fn login(self, c: Credentials) -> Result<AuthenticatedConnection, ...)> {
    ...
  }

  fn disconnect(self) -> NotConnected {
    ...
  }
}
```

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