Introducing Rust

- What is Rust?
  - Basic operations and types
  - Abstraction: Traits, Enumerated types and pattern matching
  - Memory allocation and boxes
  - **Ownership**
- Why is it interesting?

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

Ownership

- Systems programs care about ownership of resources
  - To control memory management, close files, etc.
  - To model state machines
- Programmer maintains a mental model of what part of the code owns each resource
  - What function is responsible for calling **`free()`**, **`close()`**, etc.
  - Garbage collected languages still require understanding of ownership – but make **`free()`** call automatic when lifetime ends
  - C++ and Python tie resource ownership to scoping:

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

    gives automatic resource clean-up at end of scope

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

Ownership in Rust

- Rust tracks ownership of data – enforces that every value has a single owner
- Function calls explicitly manage ownership of values
  - Take explicit ownership of a value

```
fn consume(r : Resource) {
  ...
}
```

Function *takes ownership* of parameter passed by value

No longer accessible to caller; freed at end of function

  - Borrow a value

```
fn borrow(r : &Resource) {
  ...
}
```

Function *borrows* the parameter passed via reference

Ownership remains with caller

  - Return ownership of a value

```
fn generate() -> Resource {
  ...
}
```

Function *passes ownership* of return value to caller

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

Ownership in Rust: Example

```
struct Resource {
  value : u32
}
```

```
fn consume(r : Resource) {
  println!("consumed");
}

fn main() {
  let r = Resource{value: 42};
  consume(r);
  println!("{}", r.value);
}
```

Function *takes ownership* of parameter passed by value

No longer accessible to caller; freed at end of function

- The **`consume()`** function takes ownership of the resource – doesn't return it to the caller
- Above code won't compile: **`println!()`** cannot access **`r.value`**, since **`main()`** no longer has access to **`r`** because it gave ownership to **`consume()`**

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

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;

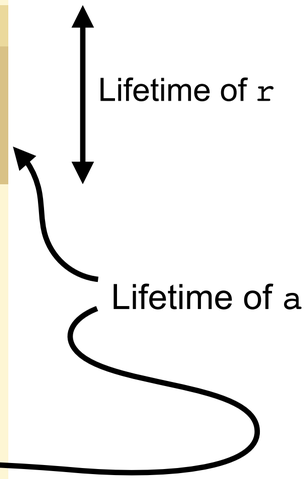
fn area_of_cone(w : f64, h : f64) -> f64 {
    let r = w / 2.0;
    let a = PI * r * (r + (h*h + r*r).sqrt());

    return a;
}

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

    let area = area_of_cone(width, height);

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



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

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

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

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

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

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

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

Taking Ownership of Data

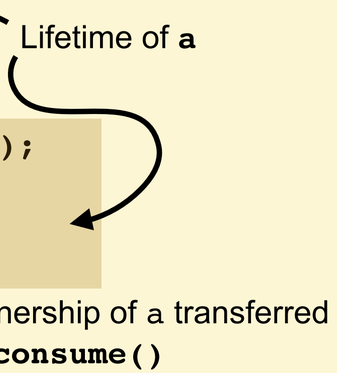
```
fn consume(mut x : Vec<u32>) {
  x.push(1);
}

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());
                                         ^
```

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

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

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

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**

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

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

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

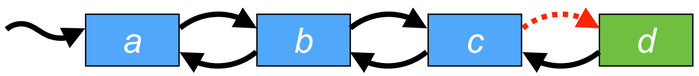
Benefits

- 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

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

Limitations

- 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
- Forces consideration of object ownership early and explicitl

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

Rust – Key Points

- Largely traditional systems programming language: basic types, control flow, and data structures are very familiar
- Key innovations in a systems language:
  - Enumerated types and pattern matching
    - **`Option`** and **`Result`**
  - Structure types and traits as an alternative to object oriented programming
  - Multiple reference types and ownership

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

Rust – Key Points

- Largely traditional systems programming language: basic types, control flow, and data structures are very familiar
- Key innovations in a systems language:
  - Enumerated types and pattern matching
    - `Option` and `Result`
  - Structure types and traits as an alternative to object oriented programming
  - Multiple reference types and ownership

- **Little in Rust is novel**
  - Rust adopts ideas from research languages:
    - Syntax is a mixture of C and Standard ML
    - Basic data types are heavily influenced by C and C++
    - Enumerated types and pattern matching adapted from Standard ML
    - Traits adapted from Haskell type classes
  - Many influences from C++, but generally “see how C++ does it, and do the opposite”
  - References and ownership rules extend ideas originally developed in Cyclone – this is where Rust has new ideas

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

Why is Rust interesting? (1/3)

- A modern type system and runtime
  - No concept of undefined behaviour
  - Memory safe
    - No buffer overflows
    - No dangling pointers
    - No null pointer dereferences
  - Zero cost abstractions to model problem space and check consistency of design → lecture 4

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

Why is Rust interesting? (2/3)

- A type system that can model data and resource ownership
  - Deterministic automatic memory management
    - Prevents iterator invalidation
    - Prevents use-after-free bugs
    - Prevents most memory leaks
  - Rules around references and ownership prevent data races in concurrent code
    - Enforces the design patterns common in well-written C programs

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

Why is Rust interesting? (3/3)

- A systems programming language that eliminates many **classes** of bug that are common in C and C++ programs

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

Summary

- What is a strongly typed language?
- Why is strong typing desirable?
- Types for systems programming
- Introduction to Rust

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