# Introducing Rust

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

# References (1/2)

- References are explicit – like pointers in C
  - Create a variable binding:

```
let x = 10;
```

```
int x = 10;
```

  - Take a reference (pointer) to that binding:

```
let r = &x;
```

```
int *r = &x;
```

# References (1/2)

- References are explicit – like pointers in C
  - Create a variable binding:

```
let x = 10;
```

```
int x = 10;
```

  - Take a reference (pointer) to that binding:

```
let r = &x;
```

```
int *r = &x;
```

  - Explicitly dereference to access value:

```
let s = *r
```

```
s = *r;
```

# References (1/2)

- References are explicit – like pointers in C
  - Create a variable binding:

```
let x = 10;
```

```
int x = 10;
```

  - Take a reference (pointer) to that binding:

```
let r = &x;
```

```
int *r = &x;
```

  - Explicitly dereference to access value:

```
let s = *r
```

```
s = *r;
```

- Functions can take parameters by reference:

```
fn calculate_length(b: &Buffer) -> usize {
    // …
}
```

```
size_t calculate_length(buffer *b) {
  // …
}
```

# References (2/2)

- Immutable references: **`&`**

```
fn main() {
    let x = 10;
    let r = &x;

    *r = 15;

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

**immutable reference** – can't be changed

compile error: cannot assign to `*r` which is behind an `&` reference

# References (2/2)

- Immutable references: **`&`**

```
fn main() {
    let x = 10;
    let r = &x;

    *r = 15;

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

**immutable reference** – can't be changed

compile error: cannot assign to `*r` which is behind an `&` reference

- Mutable references: **`&mut`**

```
fn main() {
    let mut x = 10;
    let r = &mut x;

    *r = 15;

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

**mutable reference** – referenced value *can* change

# Constraints on References

- References can never be **`null`** – they always point to a valid object
  - **`Option<T>`** indicates an optional value of type **`T`** where C would use a potentially null pointer

# Constraints on References

- References can never be **`null`** – they always point to a valid object
- There can be many immutable references (**`&`**) to an object in scope at once, but there cannot be a mutable reference (**`&mut`**) to the same object in scope
  - An object becomes immutable while immutable references to it are in scope

# Constraints on References

- References can never be **`null`** – they always point to a valid object
- There can be many immutable references (**`&`**) to an object in scope at once, but there cannot be a mutable reference (**`&mut`**) to the same object in scope
- There can be at most *one* mutable reference (**`&mut`**) to an object in scope and there can be no immutable references (**`&`**) to the object while the mutable reference exists
  - An object is inaccessible to its owner while the mutable reference exists

# Constraints on References

- References can never be **`null`** – they always point to a valid object
- There can be many immutable references (**`&`**) to an object in scope at once, but there cannot be a mutable reference (**`&mut`**) to the same object in scope
- There can be at most *one* mutable reference (**`&mut`**) to an object in scope and there can be no immutable references (**`&`**) to the object while the mutable reference exists

- These rules are enforced at compile time and prevent null pointer exceptions, iterator invalidation, data races between threads → Lecture 5

# Memory Allocation and Boxes (1/2)

- A **`Box<T>`** is a smart pointer that refers to memory allocated on the heap:

```
fn box_test() {
    let b = Box::new(5);
    println!("b = {}", b);
}
```

```
void box_test() {
  int *b = malloc(sizeof(int));
  *b = 5;
  printf("b = %d\n", *b);
  free(b);
}
```

# Memory Allocation and Boxes (1/2)

- A **`Box<T>`** is a smart pointer that refers to memory allocated on the heap:

```
fn box_test() {
    let b = Box::new(5);
    println!("b = {}", b);
}
```

```
void box_test() {
  int *b = malloc(sizeof(int));
  *b = 5;
  printf("b = %d\n", *b);
  free(b);
}
```

- Rust makes guarantees about memory allocation:
  - The value returned by Box::new() is guaranteed to be initialised
  - The allocated memory is guaranteed to match the size of the type it is to store
  - Rust guarantees that the memory will be automatically deallocated when the box goes out of scope

# Memory Allocation and Boxes (2/2)

- Boxes own and, if bound as **`mut`**, may change the data they store on the heap

```
fn main() {
    let mut b = Box::new(5);
    *b = 6;
    println!("b = {}", b);
}
```

# Memory Allocation and Boxes (2/2)

- Boxes own and, if bound as **`mut`**, may change the data they store on the heap

```
fn main() {
    let mut b = Box::new(5);
    *b = 6;
    println!("b = {}", b);
}
```

- Boxes do *not* implement the standard **`Copy`** trait; can pass boxes around, but only one copy of each box can exist – again, to avoid data races between threads
  - A **`Box<T>`** is a pointer to the heap allocated memory; if it were possible to copy the box, we could get multiple mutable references to that memory

# Strings

- Strings are Unicode text encoded in UTF-8 format
- A **`str`** is an immutable string slice, always accessed via an **`&str`** reference

```
let s1 = "Hello, World!";
```

String literals are of type **`&str`**

- The **`&str`** type is built-in to the language

# Strings

- Strings are Unicode text encoded in UTF-8 format
- A **`str`** is an immutable string slice, always accessed via an **`&str`** reference
- A **`String`** is a mutable string buffer type, implemented in the standard library

```
let s2 = String::new();
s2.push_str("Hello, World");
s2.push('!');
```

```
let s3 = String::from("Hello, World");
s3.push('!');
```

# Strings

- Strings are Unicode text encoded in UTF-8 format
- A **`str`** is an immutable string slice, always accessed via an **`&str`** reference
- A **`String`** is a mutable string buffer type, implemented in the standard library

```
let s2 = String::new();
s2.push_str("Hello, World");
s2.push('!');
```

```
let s3 = String::from("Hello, World");
s3.push('!');
```

- The **`string`** type implements the **`Deref<Target=str>`** trait, so taking a reference to a **`String`** results actually returns an **`&str`**

```
let s = String::from(“test");
let r = &s;
let t : &str = &s;
```

**`s`** has type **`String`**
**`r`** has type **`&String`**
**`t`** has type **`&str`**

- This conversion has zero cost, so functions that don't need to mutate the string tend to be only implemented for **`&str`** and not on **`String`** values

# 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

# 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

# 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

# Why is Rust interesting? (2/3)

- A type system that can model data and resource ownership
  - Deterministic automatic memory management → lectures 5 and 6
    - Prevents iterator invalidation
    - Prevents use-after-free bugs
    - Prevents most memory leaks
  - Rules around references and ownership prevent data races in concurrent code → lecture 7
    - Enforces the design patterns common in well-written C programs

# Why is Rust interesting? (3/3)

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

# Summary

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