University of Glasgow

Types and Systems Programming

Advanced Systems Programming (M)

Lecture 2

Lecture Outline

- Strongly Typed Languages
  - What is a strongly typed language?
  - Why is strong typing desirable?
  - Types for systems programming
- Introducing the Rust programming language
  - Basic operations and types
  - Arrays, vectors, tuples, strings
  - Structures and traits
  - Enumerated types and pattern matching
  - Memory allocation and boxes
  - Why is Rust interesting?

Strongly Typed Languages

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

What is a Type?

- A type describes *what* an item of data represents
  - Is it an integer? floating point value? file? sequence number? username?
  - What, conceptually, is the data?
  - How is it represented?

- Types are very familiar in programming:

```
int    x;
double y;
char  *hello = "Hello, world";

struct sockaddr_in {
  uint8_t          sin_len;
  sa_family_t      sin_family;
  in_port_t        sin_port;
  struct in_addr   sin_addr;
  char             sin_pad[16];
};
```

Declaring variables and specifying their type

Declaring a new type

What is a Type System?

- A type system is a set of rules constraining how types can be used:
  - What operations can be performed on a type?
  - What operations can be performed with a type?
  - How does a type compose with other types of data?
- A type system proves the absence of certain program behaviours
  - It doesn’t guarantee the program is correct
  - It does guarantee that *some* incorrect behaviours do not occur
    - A good type system eliminates common classes of bug, without adding too much complexity
    - A bad type system adds complexity to the language, but doesn't prevent many bugs
  - Type-related checks can happen at compile time, at run time, or both
    - e.g., array bounds checks are a property of an array type, checked at run time

Static and Dynamic Types (1/2)

- In a language with static types, the type of a variable is fixed when the variable is created:
  - Some require types to be explicitly declared; others can infer types from context
    - C and Java requires the types to be explicitly stated in all cases
    - Haskell, Rust, OCaml, ... can infer from the context
  - Just because the language can infer the type does not mean the type is dynamic:

```
> cat src/main.rs
fn main() {
    let x = 6;
    x += 4.2;
    println!("{}", x);
}
> cargo build
   Compiling hello v0.1.0 (/Users/csp/tmp/hello)
error[E0277]: cannot add-assign `{float}` to `{integer}`
 --> src/main.rs:3:7
  |
3 |     x += 4.2;
  |       ^^ no implementation for `{integer} += {float}`
  |
  = help: the trait `std::ops::AddAssign<{float}>` is not implemented for `{integer}`

error: aborting due to previous error
```

    - The Rust compiler infers that x is an integer and won't let us add a floating point 4.2 to it, since that would require changing its type

Static and Dynamic Types (2/2)

- In a language with dynamic types, the type of a variable can change during its lifetime

```
> python3
Python 3.6.2 (v3.6.2:5fd33b5926, Jul 16 2017, 20:11:06)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> x = 6
>>> type(x)
<class 'int'>
>>> x += 4.2
>>> type(x)
<class 'float'>
>>>
```

- Dynamically typed languages tend to be lower performance, but offer more flexibility
  - They have to store the type as well as its value, which takes additional memory
  - They can make fewer optimisation based on the type of a variable, since that type can change
- Systems languages generally have static types, and be compiled ahead of time, since they tend to be performance sensitive

Strong and Weak Types (1/2)

- In a language with strong types, every operation must conform to the type system
  - If the compiler and/or run-time cannot prove that the operation is legal according to the type rules, the operation is not permitted
- Other languages have weaker types, and provide ways of circumventing the type checker:
  - This might be automatic safe conversions between types:

```
float  x = 6.0;
double y = 5.0;

double z = x + y;
```

C has static types, but allows lower precision values to be assigned to variables with higher precision types – there's no data loss

  - Or it might be an open-ended cast:

```
char *buffer[BUFLEN];
int   fd = socket(…);
…
if (recv(fd, buffer, BUFLEN, 0) > 0) {
  struct rtp_packet *p = (struct rtp_packet *) buf;
  …
}
```

Common C programming idiom: casting between types using pointers to evade the type system

Strong and Weak Types (2/2)

- Sometimes clearer to consider *safe* and *unsafe* languages, rather than strong or weak types
  - *“A safe language is one that protects its own abstractions”* [Pierce]
  - A safe language – whether static or dynamic – knows the types of all variables, and only allows legal operations on those values
  - An unsafe language allows the types to be circumvented – to perform operations that the programmer believes are correct, but the type system can’t prove so

Why is Strong Typing Desirable?

- "Well-typed programs don't go wrong" – Robin Milner
  - The result is well-defined – although not necessarily correct
    - The type system ensures results are consistent with the rules of the language, but cannot check if you calculated the right result
    - A strongly-typed system will only ever perform operations on a type that are legal – there is no undefined behaviour
  - Types help model the problem, check for consistency, and eliminate common classes of bug

Segmentation fault (core dumped)

Segmentation faults should never happen:

- Compiler and runtime should strongly enforce type rules
- If program violates them, it should be terminated cleanly
- Security vulnerabilities – e.g., buffer overflow attacks – come from undefined behaviour after type violations

```
Segmentation fault (core dumped)
```

3.4.3

1 **undefined behavior**

behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

2 NOTE Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

C has 193 kinds of undefined behaviour

Appendix J of the C standard https://www.iso.org/standard/74528.html ($$$) or http://www.open-std.org/jtc1/sc22/wg14/www/abq/c17_updated_proposed_fdis.pdf

Each leads to *entirely unpredictable* results
→ https://blog.regehr.org/archives/213

A language should specify behaviour of each operation

The behavior is undefined in the following circumstances:

— A "shall" or "shall not" requirement that appears outside of a constraint is violated (clause 4).

— A nonempty source file does not end in a new-line character which is not immediately preceded by a backslash character or ends in a partial preprocessing token or comment (5.1.1.2).

— Token concatenation produces a character sequence matching the syntax of a universal character name (5.1.1.2).

— A program in a hosted environment does not define a function named `main` using one of the specified forms (5.1.2.2.1).

— The execution of a program contains a data race (5.1.2.4).

— A character not in the basic source character set is encountered in a source file, except in an identifier, a character constant, a string literal, a header name, a comment, or a preprocessing token that is never converted to a token (5.2.1).

— An identifier, comment, string literal, character constant, or header name contains an invalid multibyte character or does not begin and end in the initial shift state (5.2.1.2).

— The same identifier has both internal and external linkage in the same translation unit (6.2.2).

— An object is referred to outside of its lifetime (6.2.4).

— The value of a pointer to an object whose lifetime has ended is used (6.2.4).

— The value of an object with automatic storage duration is used while it is indeterminate (6.2.4, 6.7.9, 6.8).

— A trap representation is read by an lvalue expression that does not have character type (6.2.6.1).

— A trap representation is produced by a side effect that modifies any part of the object using an lvalue expression that does not have character type (6.2.6.1).

— The operands to certain operators are such that they could produce a negative zero result, but the implementation does not support negative zeros (6.2.6.2).

— Two declarations of the same object or function specify types that are not compatible (6.2.7).

— A program requires the formation of a composite type from a variable length array type whose size is specified by an expression that is not evaluated (6.2.7).

— Conversion to or from an integer type produces a value outside the range that can be represented (6.3.1.4).

— Demotion of one real floating type to another produces a value outside the range that can be represented (6.3.1.5).

— An lvalue does not designate an object when evaluated (6.3.2.1).

— A non-array lvalue with an incomplete type is used in a context that requires the value of the designated object (6.3.2.1).

— An lvalue designating an object of automatic storage duration that could have been declared with the `register` storage class is used in a context that requires the value of the designated object, but the object is uninitialized. (6.3.2.1).

— An lvalue having array type is converted to a pointer to the initial element of the array, and the array object has register storage class (6.3.2.1).

Types for Systems Programming

- C is weakly typed and widely used for systems programming
  - Why is this?
  - Can systems programming languages be strongly typed?
  - What are the challenges in strongly typed systems programming?

Why is C Weakly Typed?

- Mostly, historical reasons:
  - The original designers of C were not type theorists
  - The original machines on which C was developed didn’t have the resources to perform complex type checks
  - Type theory was not particularly advanced in the early 1970s – we didn’t know how to do better?

Is Strongly-typed Systems Programming Feasible?

- Yes – many examples of operating systems written in strongly-typed languages
  - Old versions of macOS written in Pascal
  - Project Oberon http://www.projectoberon.com
  - US DoD and the Ada programming language
    - Aerospace, military, air traffic control
- Popularity of Unix and C has led to a belief that operating systems require unsafe code
  - True only at the very lowest levels
  - Most systems code, including device drivers, can be written in strongly typed, safe, languages
  - Rust is a modern attempt to provide a type-safe language suited to systems programming

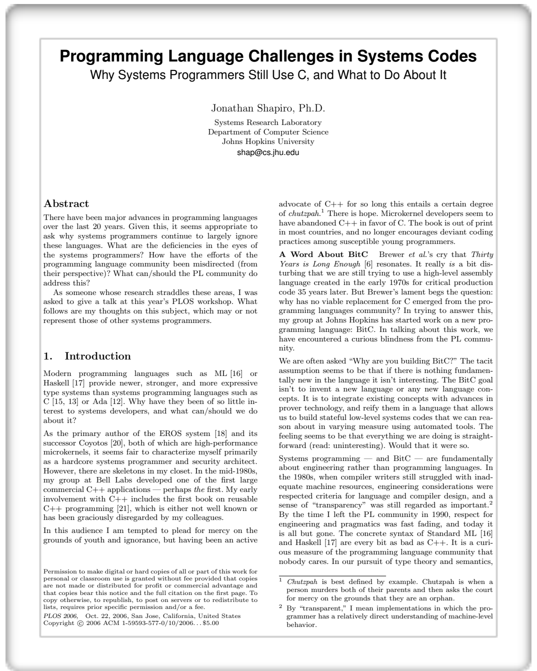
```
type ErrorType   is range 0..15;
type UnitSelType is range 0..7;
type ResType     is range 0..7;
type DevFunc     is range 0..3;
type Flag        is (Set, NotSet);
type ControlRegister is
record
    errors    : ErrorType;
    busy      : Flag;
    unitSel   : UnitSelType;
    done      : Flag;
    irqEnable : Flag;
    reserved  : ResType;
    devFunc   : DevFunc;
    devEnable : Flag;
end record;

for ControlRegister use
record
    errors    at 0*Word range 12..15;
    busy      at 0*Word range 11..11;
    unitSel   at 0*Word range  8..10;
    done      at 0*Word range  7.. 7;
    irqEnable at 0*Word range  6.. 6;
    reserved  at 0*Word range  3.. 5;
    devFunc   at 0*word range  1.. 2;
    devEnable at 0*Word range  0.. 0;
end record;

for ControlRegister'Size use 16;
for ControlRegister'Alignment use Word;
for ControlRegister'Bit_order use Low_Order_First;
...
```

Challenges in Strongly-typed Systems Programming

- Four fallacies:
  - Factors of 1.5x to 2x in performance don't matter
  - Boxed representation can be optimised away
  - The optimiser can fix it
  - The legacy problem is insurmountable
- Four challenges:
  - Application constraint checking
  - Idiomatic manual storage management
  - Control over data representation
  - Managing shared state



J. Shapiro, "Programming language challenges in systems codes: why systems programmers still use C, and what to do about it", Workshop on Programming Languages and Operating Systems, San Jose, CA, October 2006. DOI:10.1145/1215995.1216004

- Many good ideas in research languages and operating systems – only recently that these issues have been considered to make *practical* tools

Introducing Rust

- What is Rust?
  - Basic operations and types
  - Arrays, vectors, tuples, strings
  - Structures and traits
  - Enumerated types and pattern matching
  - Memory allocation and boxes
- Why is it interesting?

The Rust Programming Language

- Initially developed by Graydon Hoare as a side project, starting 2006
- Sponsored by Mozilla since 2009
- Rust v1.0 released in 2015
- Rust v1.31 “Rust 2018 Edition” released December 2018
  - Backwards compatible – but tidies up the language
  - https://blog.rust-lang.org/2018/12/06/Rust-1.31-and-rust-2018.html
- New releases made every six weeks – strong backwards compatibility policy

Basic Features

```
fn main() {
    println!("Hello, world!”);
}
```

Function definition; macro expansion; string literal

```
fn gcd(mut n: u64, mut m: u64) -> u64 {
    assert!(n != 0 && m != 0);
    while m!=0 {
        if m < n {
            let t = m;
            m = n;
            n = t;
        }
        m = m % n;
    }
    n
}

fn main() {
  let m = 12;
  let n = 16;
  let r = gcd(m, n);
  println!("gcd({}, {}) = {}", m, n, r);
}
```

Function arguments and return type; mutable vs immutable

Control flow: `while` and `if` statements

Local variable definition (`let` binding); type is inferred

Implicitly returns value of final expression (can return from function early using `return` statement)

Basic Types

C	Rust
`int`	`isize`

`int8_t, signed char`
`int16_t`
`int32_t`
`int64_t`
	`i8`
	`i16`
	`i32`
	`i64`

`float` `double`	`f32` `f64`
`_Bool` *`int`*	`bool`
	`char` *(32 bit unicode scalar value)*

C	Rust
`unsigned`	`usize`

`uint8_t, unsigned char`
`uint16_t`
`uint32_t`
`uint64_t`
	`u8`
	`u16`
	`u32`
	`u64`

https://doc.rust-lang.org/book/ch03-02-data-types.html

- Basic types have close to direct mapping from C to Rust
- Rust has a native `bool` type, C uses `int` to represent boolean (C99 has `_Bool`)
- In C, a `char` is defined as a single byte, implementation defined whether signed, no character set specified; Rust `char` is a 32-bit Unicode scalar value
  - Unicode scalar value ≠ code point ≠ grapheme cluster ≠ “character”
  - e.g., ü is *two* scalar values “Latin small letter U (U+0075)” + “combining diaeresis (U+0308)”, but *one* grapheme cluster (https://crates.io/crates/unicode-segmentation – text is *hard)*

Arrays and Vectors

```
fn main() {
    let a = [1, 2, 3, 4, 5];
    let b = a[2];
    println!("b={}", b);
}
```

Arrays work as expected
Types are inferred

```
fn main() {
    let v = vec![1, 2, 3, 4, 5];
}
```

Vectors are the dynamically sized equivalent
`vec![...]` macro creates vector literals

```
fn main() {
    let mut v = Vec::new();
    v.push(1);
    v.push(2);
    v.push(3);
    v.push(4);
    v.push(5);
}
```

Vectors are implemented internally as the equivalent of a C program that uses `malloc()` to allocate space for an array, then `realloc()` to grow the space when it gets close to full.

They implement the `Deref<Target=&[T]>` trait, so they can be passed to functions that expect a reference to an array of the same type – gives pointer to array implementing the vector

Tuples

```
fn main() {
    let tup = (500, 6.4, 1);

    let (x, y, z) = tup;

    println!("The value of y is: {}", y);
    println!("The 2nd element is {}”, tup.1)
}
```

Tuples are collections of unnamed values; each element can be a different type

`let` bindings can de-structure tuples

Tuple elements can be accessed by index

```
()
```

An empty tuple is the unit type (like `void` in C)

Structure Types (1/2)

```
struct Rectangle {
    width: u32,
    height: u32
}

fn area(rectangle: Rectangle) -> u32 {
    rectangle.width * rectangle.height
}

fn main() {
    let rect = Rectangle { width: 30, height: 50 };

    println!("Area of rectangle is {}", area(rect));
}
```

Structs are collections of named values; each element can have a different type

https://doc.rust-lang.org/book/ch05-00-structs.html

Access fields in struct using dot notation

Create a struct, specifying the values for each field

Structure Types (2/2)

```
struct Point(i32, i32, i32);

let origin = Point(0, 0, 0);
```

Tuple structs are tuples with a type name
useful for type aliases

```
struct Marker;
```

Unit-like structs have no elements and take up no space
useful as markers or type parameters

Methods

- Rust doesn't have *objects* in the traditional way, but you can implement methods on structs

```
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect = Rectangle { width: 30, height: 50 };

    println!("Area of rectangle is {}", rect.area());
}
```

Methods defined in `impl` block

Methods and instance variables use explicit `self` references, like Python

Method call uses dot notation

Traits (1/5)

- Traits describe features that types can implement
  - Methods that must be provided, and associated types that must be specified, by types that implement the trait – but not instance variables or data
  - Similar to type classes in Haskell or interfaces in Java
  - https://doc.rust-lang.org/book/ch10-02-traits.html

```
trait Area {
    fn area(&self) -> u32;
}
```

Define a trait with a single method that must be implemented

```
struct Rectangle {
    width: u32,
    height: u32,
}

impl Area for Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

Implement that trait for the `Rectangle` type

Traits (2/5)

```
trait Area {
    fn area(&self) -> u32;
}

struct Rectangle {
    width: u32,
    height: u32,
}

impl Area for Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

```
struct Circle {
    radius: u32
}

impl Area for Circle {
    fn area(&self) -> u32 {
        PI * self.radius * self.radius
    }
}
```

A trait can be implemented by multiple types

Traits are an important tool for abstraction in Rust – similar role to sub-typing in many languages

Traits (3/5): Generic Functions

- Rust uses traits instead of classes and inheritance
  - Define a trait:

```
trait Summary {
    fn summarize(&self) -> String;
}
```

  - Write functions that work on types that implement that trait:

```
fn notify<T: Summary>(item: T) {
    println!("Breaking news! {}", item.summarize());
}
```

  Type parameter in angle brackets: `T` is any type that implement the `Summary` trait

- Allows generic code – functions or methods that can work with *any* type that implements a particular trait

Traits (4/5): Deriving Common Traits

- The `derive` attribute makes compiler automatically generate implementations of some common traits:

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}
```

- Generates `impl` block with standard implementation of methods for derived trait
- Compiler implements this for many traits in the standard library that are always implemented the same way: https://doc.rust-lang.org/book/appendix-03-derivable-traits.html
- Can also be implemented for other traits:
  - Only useful if every implementation of the trait will follow the exact same structure
  - https://doc.rust-lang.org/book/ch19-06-macros.html#how-to-write-a-custom-derive-macro

Traits (5/5): Associated Types

- Traits can also specify associated types – types that must be specified when a trait is implemented
- Example: `for` loops operate on iterators

```
fn main() {
    let a = [42, 43, 44, 45, 46];

    for x in a.iter() {
        println!("x={}", x);
    }
}
```

`a.iter()` returns an iterator over the array

- An iterator is something that implements the `Iterator` trait:

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
    // more...
}
```

The `impl` of the trait has to specify the type, `item`, as well as the methods

Enumerated Types (1/2)

```
enum TimeUnit {
    Years, Months, Days, Hours, Minutes, Seconds
}
```

Basic enums work just like in C

https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html

Enums also generalise to store tuple-like variants:

```
enum RoughTime {
    InThePast(TimeUnit, u32),
    JustNow,
    InTheFuture(TimeUnit, u32)
}

let when = RoughTime::InThePast(TimeUnit::Years, 4*20 + 7);
```

...and struct-like variants:

```
enum Shape {
    Sphere {center: Point3d, radius: f32},
    Cuboid {corner1: Point3d, corner2: Point3d}
}

let unit_sphere = Shape::Sphere{center: ORIGIN, radius: 1.0};
```

Enumerated Types (2/2)

- Enums indicates that a *type* can be one of several alternatives
  - They can have type parameters that must be defined when the enum is instantiated:

```
enum Result<T, E> {
    Ok(T),
    Err(E)
}
```

  - They can also implement methods – same as for structs

- Enums are useful to model data that can take one of a set of related values

`Option` and `Result`

- Rust implements two extremely useful standard enums
- The `option` type represents optional values
  - In C, one might write a function to lookup a key in a database:

```
value *lookup(struct db*self, key *k) {
  // ...
}
```

  this returns a pointer to the value, or `null` if the key doesn't exist

  - In Rust, the equivalent function returns an optional value:

```
fn lookup(self, key : Key) -> Option<Value> {
  // ...
}
```

```
enum Option<T> {
    Some(T),
    None
}
```

- The `result` type similarly encodes success or failure:

```
fn recv(self) -> Result<Message, NetworkError> {
  // ...
}
```

```
enum Result<T, E> {
    Ok(T),
    Err(E)
}
```

  - Easy to ignore errors or missing values in C – Rust uses *pattern matching* on `Option`/`Result` types to encourage error handling; no concept of exceptions

Pattern Matching (1/4)

- Rust `match` expressions generalise the C `switch` statement
  - https://doc.rust-lang.org/book/ch06-02-match.html

- Match against constant expressions and wildcards:

```
match meadow.count_rabbits() {
    0 => {} // nothing to say
    1 => println!("A rabbit is nosing around in the clover."),
    n => println!("There are {} rabbits hopping about in the meadow", n)
}
```

  - The value of `meadow.count_rabbits()` is matched against the alternatives
  - If matches the constants `0` or `1`, the corresponding branch executes
  - If none match, the value is stored in the variable `n` and that branch executes
    - Matching against _ gives a wildcard without assigning to a variable

Pattern Matching (2/4)

- Patterns can be any type, not just integers

```
let calendar = match settings.get_string("calendar") {
    "gregorian" => Calendar::Gregorian,
    "chinese"   => Calendar::Chinese,
    "ethiopian" => Calendar::Ethiopian,
    other       => return parse_error("calendar", other)
};
```

- The `match` expression evaluates to the value of the chosen branch
  - Allows, e.g., use in `let` bindings, as shown

Pattern Matching (3/4)

- Patterns can match against enum values:

```
enum RoughTime {
    InThePast(TimeUnit, u32),
    JustNow,
    InTheFuture(TimeUnit, u32)
}

let when = RoughTime::InThePast(TimeUnit::Years, 4*20 + 7);
```

```
match rt {
    RoughTime::InThePast(units, count) => format!("{} {} ago", count, units.plural()),
    RoughTime::JustNow => format!("just now"),
    RoughTime::InTheFuture(units, count) => format!("{} {} from now", count, units.plural())
}
```

- Selects from different types of data, expressed as `enum` variants
- Variables can be bound against values stored in `enum` variants
- Must match against all possible variants of the `enum`, or include a wildcard – else compile error

Patterns Matching (4/4)

- C functions often return pointer to value, or `null` if the value doesn't exist
- Easy to forget the `null` check when using the value:

```
customer *get_user(struct db *db, char *username) {
  // ...
}

customer *c = get_user(db, customer_name);
book_ticket(c, event);
```

  - Program crashes with null pointer dereference at run-time if user is not found
- Equivalent Rust code returns an `Option<>` type and pattern matches on result:

```
fn get_user(self, username : String) -> Option<Customer> {
  // ...
}

match db.get_user(customer_name) {
  Some(customer) => book_ticket(customer, event),
  None           => handle_error()
}
```

```
enum Option<T> {
    Some(T),
    None
}
```

https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html#the-option-enum-and-its-advantages-over-null-values

  - Why is this better? Won't compile unless `match` against both variants; documents the optional nature of the result in a machine checkable way in the type
  - Can't force meaningful error handling, but Rust compiler tells you if you forget to handle errors

References (1/3)

- 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:

```
assert!(*r == 10);
```

```
assert(*r == 10);
```

- Functions can take parameters by reference:

```
fn calculate_length(s: &String) -> usize {
    s.len()
}
```

References (2/3)

- Rust has *two* types of reference:
  - Immutable references: `&`

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

    *r = 15;

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

immutable reference – referenced value *cannot* be changed, but several immutable references can refer to the same value

compile error: cannot assign to `*r` which is behind a `&` 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, but the mutable reference *must* be unique

References (3/3)

- Constraints on references:
  - References can never be `null` – they always point to a valid object
    - Use `option<T>` to indicate an optional value of type `T` where C would use a potentially null pointer
  - 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
  - There can be at most *one* mutable reference (`&mut`) to an object in scope, but there cannot be any immutable references (`&`) to the object while that mutable reference exists
    - An object is inaccessible to its owner while the mutable reference exists
  - These *ownership* and *borrowing* rules are enforced at compile time → lecture 4

- These restrictions prevent:
  - Null pointer exceptions, iterator invalidation, data races between threads
  - → lectures 4 and 6 for details

Memory Allocation and Boxes

- 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);
}
```

- Note: boxes implement the standard `Display` trait so can be printed without dereferencing
- Memory allocated to the box is freed when the box goes out of scope; we must explicitly call `free()` in C
- 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 implemented as a struct that has a private pointer to 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`

  - `&str` is like `char *` in C, except contents guaranteed to be immutable UTF-8 text
  - `&str` is built in to the language
- 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;
```

`r` is of 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 a traditional imperative systems programming language
  - Basic types, control flow, data structures are very familiar
- Key innovations:
  - Enumerated types and pattern matching
    - `Option` and `Result`
  - Structure types and traits as an alternative to object oriented programming
  - Ownership, borrowing, and multiple reference types
  - *Little of this is novel* – adopting many good ideas from research languages:
    - Syntax is a mixture of C, Standard ML, and Pascal
    - Basic data types are heavily influenced by C and C++
    - Enumerated types and pattern matching are adapted from Standard ML
    - Traits are adapted from Haskell type classes
    - The ownership and borrowing rules, and the way references are handled, are built on ideas developed in Cyclone
    - Many ideas from C++, if often of the form "see how C++ does it, and do the opposite"

Why is Rust interesting?

- A modern type system and runtime
  - No concept of undefined behaviour
    - Buffer overflows, dangling pointers, null pointer dereferences
  - No-cost abstractions for modelling problem space and checking consistency of solutions → lecture 3
- A type system that can model data and resource ownership:
  - Deterministic automatic memory management → lectures 4 and 5
    - Avoids iterator invalidation and use-after-free bugs, most memory leaks
  - Rules around references, data ownership, and borrowing prevent data races in concurrent code → lecture 6
    - Enforces the design patterns common in well-written C programs

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