University of Glasgow

Types and Systems Programming

Advanced Systems Programming (M)

Lecture 2

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow | This work is licensed under the Creative Commons Attribution-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/4.0/ or send a letter to Creative Commons, PO Box 1866,

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
  - Pattern matching
  - Memory management
  - Why is Rust interesting?

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

Strongly Typed Languages

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

CC BY ND

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, **`struct sockaddr_in`**

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

What is a Type System? (1/2)

- 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?

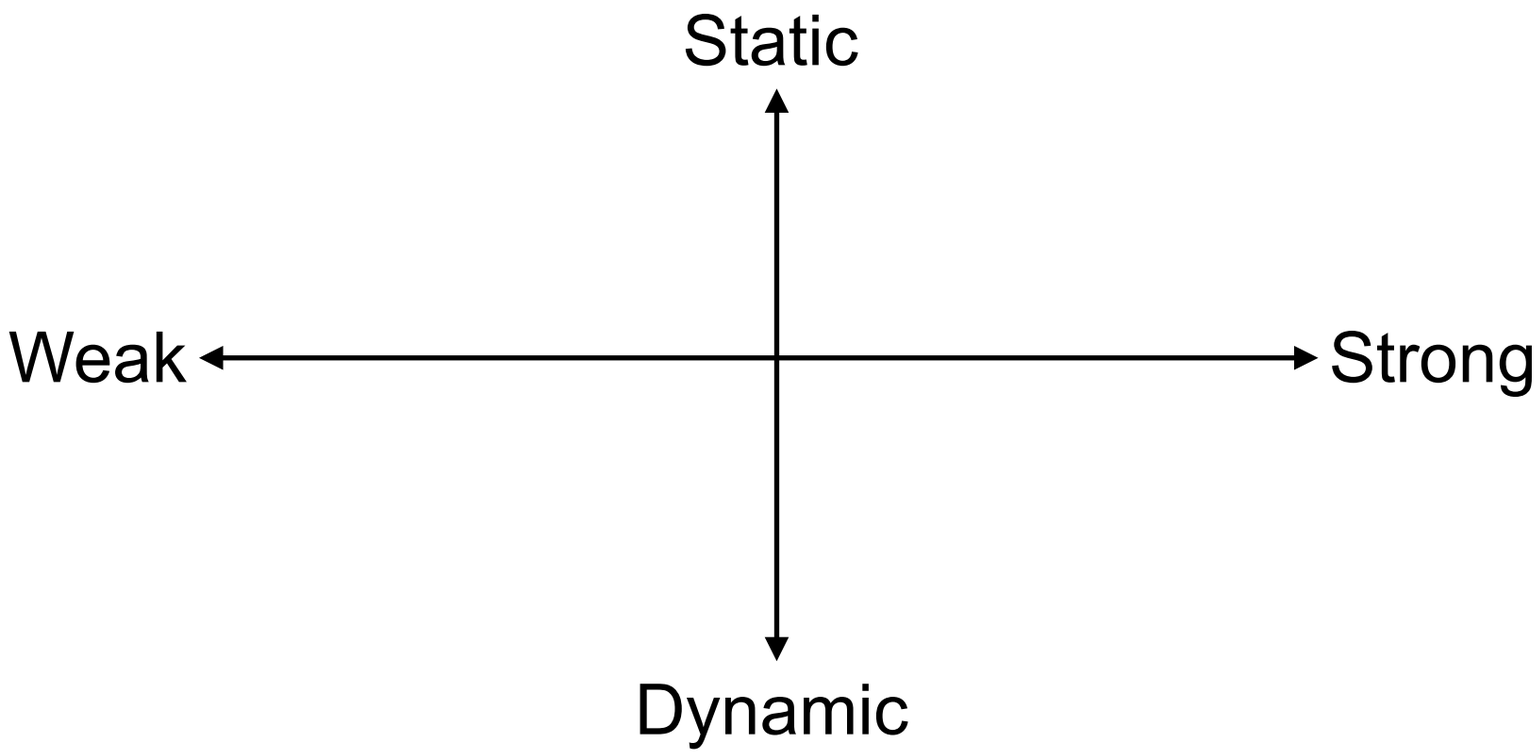
Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

What is a Type System? (2/2)

- 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

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

Types of Type Systems



- Can objects changes their type?
- How strictly are the typing rules enforced?

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

Static and Dynamic Types (1/2)

- In a language with static types, the type of a variable is fixed:
  - 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 value to it, since that would require changing its type

cc BY ND

Static and Dynamic Types (2/2)

- In a language with dynamic types, the type of a variable can change:

```
> 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

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

Strong and Weak Types (1/2)

- In a language with strong types, every operation must conform to the type system
  - Operations that cannot be proved to conform to the typing rules are not permitted

- Weakly typed languages 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;
```

- 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

CC BY ND

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*"

Benjamin C. Pierce, "Types and Programming Languages", MIT Press 2002, https://mitpress.mit.edu/books/types-and-programming-languages

- Safe languages, whether static or dynamic, know the types of all variables and only allows legal operations on those values
- Unsafe languages allow the types to be circumvented to perform operations the programmer believes correct, but the type system can't prove so

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

Why is Strong Typing Desirable?

- "Well-typed programs don't go wrong" – Robin Milner

- Results of strongly typed programs are well-defined
  - The type system ensures results are consistent with the rules of the language
  - A strongly-typed program 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
  - Strong typing *cannot* prove that a program is correct
  - Strong typing *can* prove the absence of certain classes of bug

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

`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

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

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).

CC BY ND

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?

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

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?

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

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

```
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;
...
```

- 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

cc BY ND

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



Programming Language Challenges in Systems Codes
Why Systems Programmers Still Use C, and What to Do About It

Jonathan Shapiro, Ph.D.
Systems Research Laboratory
Department of Computer Science
Johns Hopkins University
shap@cs.jhu.edu

Abstract

There have been major advances in programming languages over the last 20 years. Given this, it seems appropriate to ask why systems programmers continue to largely ignore these languages. What are the deficiencies in the eyes of the systems programmers? How have the efforts of the programming language community been misdirected (from their perspective)? What can/should the PL community do address this?

As someone whose research straddles these areas, I was asked to give a talk at this year's PLOS workshop. What follows are my thoughts on this subject, which may or not represent those of other systems programmers.

1. Introduction

Modern programming languages such as ML [16] or Haskell [17] provide newer, stronger, and more expressive type systems than systems programming languages such as C [15, 13] or Ada [12]. Why have they been of so little interest to systems developers, and what can/should we do about it?

As the primary author of the EROS system [18] and its successor Coyotos [20], both of which are high-performance microkernels, it seems fair to characterize myself primarily as a hardcore systems programmer and security architect. However, there are skeletons in my closet. In the mid-1980s, my group at Bell Labs developed one of the first large commercial C++ applications — perhaps *the* first. My early involvement with C++ includes the first book on reusable C++ programming [21], which is either not well known or has been graciously disregarded by my colleagues.

In this audience I am tempted to plead for mercy on the grounds of youth and ignorance, but having been an active advocate of C++ for so long this entails a certain degree of *chutzpah*.[1] There is hope. Microkernel developers seem to have abandoned C++ in favor of C. The book is out of print in most countries, and no longer encourages deviant coding practices among susceptible young programmers.

A Word About BitC Brewer *et al.*'s cry that *Thirty Years is Long Enough* [6] resonates. It really *is* a bit disturbing that we are still trying to use a high-level assembly language created in the early 1970s for critical production code 35 years later. But Brewer's lament begs the question: why has no viable replacement for C emerged from the programming languages community? In trying to answer this, my group at Johns Hopkins has started work on a new programming language: BitC. In talking about this work, we have encountered a curious blindness from the PL community.

We are often asked "Why are you building BitC?" The tacit assumption seems to be that if there is nothing fundamentally new in the language it isn't interesting. The BitC goal isn't to invent a new language or any new language concepts. It is to integrate existing concepts with advances in prover technology, and reify them in a language that allows us to build stateful low-level systems codes that we can reason about in varying measure using automated tools. The feeling seems to be that everything we are doing is straightforward (read: uninteresting). Would that it were so.

Systems programming — and BitC — are fundamentally about engineering rather than programming languages. In the 1980s, when compiler writers still struggled with inadequate machine resources, engineering considerations were respected criteria for language and compiler design, and a sense of "transparency" was still regarded as important.[2] By the time I left the PL community in 1990, respect for engineering and pragmatics was fast fading, and today it is all but gone. The concrete syntax of Standard ML [16] and Haskell [17] are every bit as bad as C++. It is a curious measure of the programming language community that nobody cares. In our pursuit of type theory and semantics,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLOS 2006, Oct. 22, 2006, San Jose, California, United States
Copyright © 2006 ACM 1-59593-577-0/10/2006…$5.00

[1] *Chutzpah* is best defined by example. Chutzpah is when a person murders both of their parents and then asks the court for mercy on the grounds that they are an orphan.

[2] By "transparent," I mean implementations in which the programmer has a relatively direct understanding of machine-level behavior.

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

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

Introducing Rust

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

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

The Rust Programming Language

- A modern systems programming language with a strong static type system
- 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



CC BY ND

Basic Features and Types

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

Basic Features and Types (1/10)

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

Function definition: **`main()`** takes no arguments, returns noting

Macro expansion: **`println!()`** with string literal as argument

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

Basic Features and Types (2/10)

```
use std::env;

fn main() {
    for arg in env::args() {
        println!("{:?}", arg);
    }
}
```

Import **`env`** module from standard library

Use **`for`** loop to iterate over command line arguments

```
$ rustc args.rs
$ ./args 1 2 3
"./args"
"1"
"2"
"3"
$
```

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

Basic Features and Types (3/10)

```
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

Function implicitly returns value of final expression (**`return`** statement allows early return)

Function call, assigning result to local variable

```
$ rustc gcd.rs
$ ./gcd
gcd(12, 16) = 4
$
```

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

Basic Features and Types (4/10)

C	Rust
`int`	`isize`
`int8_t, signed char`	`i8`
`int16_t`	`i16`
`int32_t`	`i32`
`int64_t`	`i64`
`float`	`f32`
`double`	`f64`
`int`	`bool`
`char`	

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

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

- Primitive types map closely to those in C
- Rust has a native **`bool`** type, C uses **`int`** to represent booleans
- In C, a **`char`** is one byte, implementation defined if signed, character set unspecified
- In Rust, a **`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)*

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

Basic Features and Types (5/10)

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

Arrays work as expected; elements are all the same type; number of elements fixed
Type of the array is inferred from type of elements

Array types are written `[T]`, where `T` is the type of the elements

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

Basic Features and Types (6/10)

```
fn main() {
    let mut v : Vec<u32> = Vec::new();
    v.push(1);
    v.push(2);
    v.push(3);
    v.push(4);
    v.push(5);
}
```

Vectors, **`Vec<T>`**, are the variable sized equivalent of arrays
The type parameter **`<T>`** specifies element

The **`: Vec<u32>`** modifier specifies the type for variable **`v`** and can usually be omitted, allowing compiler to infer type:

```
let mut v = Vec::new();
```

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

The **`vec![...]`** macro is a shortcut to create vector literals

Vectors implemented 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.

A vector, **`Vec<T>`**, can be passed to a function that expects a reference to an array **`[T]`**
(`Vec<T>` implements the trait `Deref<Target=&[T]>` that defines the conversion)

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

Basic Features and Types (7/10)

```
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)

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

Basic Features and Types (8/10)

```
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

Creates a struct, specifying field values

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

Basic Features and Types (9/10)

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

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

Elements of a struct can be unnamed: known as tuple structs
useful as type aliases

```
struct Marker;
```

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

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

Basic Features and Types (10/10)

```
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

Explicit **`self`** references, like Python

Method call uses dot notation

Methods can be implemented on structs – somewhat similar to objects, but Rust does not support inheritance or sub-types

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

Abstraction

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

Traits (1/5)

- Traits describe functionality 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;
}

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

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

Define a trait with a single method that must be implemented

Define a struct type, **`Rectangle`**

Implement the **`Area`** trait on the **`Rectangle`** type

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

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
    }
}
```

A trait can be implemented by multiple types – here we also implement the **`Area`** trait for the **`Circle`** type

```
struct Circle {
    radius: u32
}

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

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

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

Traits (3/5): Generic Functions

- Rust uses traits instead of classes and inheritance to define generic functions or methods that work with any type that implements a particular trait

- 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

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

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,
}
```

The **`#[derive(T)]`** annotation makes compiler generate **`impl`** block with standard implementation of methods for derived trait **`T`**

- Compiler implements this for 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 user-defined 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

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

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);
    }
}
```

The **`a.iter()`** function call 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...
}
```

An **`impl`** of **`Iterator`** must define the type of **`item`**, as well as implementing the methods

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

Enumerated Types (1/3)

```
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};
```

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

Enumerated Types (2/3)

- An **`enum`** is used when a variable, parameter, or result can have on of several possible types
  - An **`enum`** defines alternative types for a type
  - An **`enum`** can have type parameters that must be defined when the enum is instantiated:

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

  - An **`enum`** can have methods and can implement traits

- Use **`enum`** to model data that can take one of a set of related values

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

Enumerated Types (3/3)

- Standard library has two extremely useful standard **`enum`** types
- 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 would return **`Option<Value>`**:

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

Defined in the standard library:

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

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

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

Defined in the standard library:

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

BY ND

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

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

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,
    _           => 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

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

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

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

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

- Rust function return an **`Option`** and pattern match 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()
}
```

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? All enum variants must be handled, so won't compile if you forget to check the error case

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

Memory Management

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

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()
}
```

BY ND

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 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, but the mutable reference *must* be unique

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

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

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

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);
}
```

`Box` implements the trait **`Display`** from the standard library so can be printed without dereferencing

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

- 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

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

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

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

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 in Rust 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 influences from C++, generally "see how C++ does it, and do the opposite"

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

Why is Rust interesting?

- A modern type system and runtime
  - No concept of undefined behaviour
    - Buffer overflows, dangling pointers, null pointer dereferences
  - Zero 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

cc BY ND

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