Design Patterns

- Specific numeric types
- Enumerations for alternatives

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

Numeric Types

- Is a value really a **`double`** or **`int`**, or does it have some meaning?
  - Temperature in degrees celsius
  - Speed in miles per hour
  - UserID
  - Packet sequence number
  - ...
- Encode the meaning as a type, so the compiler checks for consistent usage
  - Operations that mix types should fail, or perform safe unit conversions if possible
  - Operations that are inappropriate for a type shouldn't be possible



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

Numeric Types – Strong Typing

- Weakly typed – programmer knows `c` and `f` are different types, but the compiler does not
- Program silently calculates the wrong answer

```
fn main() {
    let c = 15.0;  // Celsius
    let f = 50.0;  // Fahrenheit

    let t = c + f;

    println!("{:?}", t); // 65.0
}
```

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

Numeric Types – Strong Typing

- Strongly typed – **`struct`** with single unnamed field wraps numeric value
  - **`derive`** and **`impl`** standard operations
    https://crates.io/crates/newtype_derive has macros to auto-generate `impl` blocks
  - Resulting code won't compile → types are mismatched

```
> cargo build
   Compiling foo v0.1.0 (/Users/csp/tmp/foo)
error[E0308]: mismatched types
  --> src/main.rs:28:17
   |
28 |     let t = c + f;
   |                 ^ expected struct `Celsius`, found struct `Fahrenheit`
   |
   = note: expected type `Celsius`
              found type `Fahrenheit`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0308`.
error: Could not compile `foo`.

To learn more, run the command again with --verbose.
>
```

```
use std::ops::Add;

#[derive(Debug,PartialEq,PartialOrd)]
struct Celsius(f32);

#[derive(Debug,PartialEq,PartialOrd)]
struct Fahrenheit(f32);

impl Add for Celsius {
    type Output = Celsius;

    fn add(self, other : Celsius) -> Self::Output {
        Celsius(self.0 + other.0)
    }
}

impl Add for Fahrenheit {
    type Output = Fahrenheit;

    fn add(self, other : Fahrenheit) -> Self::Output {
        Fahrenheit(self.0 + other.0)
    }
}

fn main() {
    let c = Celsius(15.0);
    let f = Fahrenheit(50.0);
    let t = c + f;

    println!("{:?}", t);
}
```

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

Numeric Types – Conversion

- We can add implementations that perform unit conversion

```
impl Add<Fahrenheit> for Celsius {
    type Output = Celsius;

    fn add(self, other: Fahrenheit) -> Self::Output {
        Celsius(self.0 + ((other.0 - 32.0) * 5.0 / 9.0))
    }
}

fn main() {
    let c = Celsius(15.0);
    let f = Fahrenheit(50.0);

    let t = c + f;

    println!("{:?}", t);   // Celsius(42.7) = Fahrenheit(109)
}
```

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

Numeric Types – Operations

- Do all the standard operations make sense for the type?
  - It's reasonable to compare **`Celsius`** values:

```
fn is_freezing(temp: Celsius) -> bool {
    temp < Celsius(0.0)
}
```

    so you'd implement the **`Ord`** trait that provides these operations

  - But this might not make sense for a **`UserID`** type
    - You likely want to be able to compare two **`UserID`** values for equality (the `Eq` trait), but adding two **`UserID`** values or comparing to see which is largest might not be meaningful
    - Not all standard operations need to be implemented for a type

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

Numeric Types – No Runtime Cost

- Wrapping value inside **`struct`** adds zero runtime overhead in Rust
  - Programmer must implement standard operations – some extra code, but no runtime cost
    - https://crates.io/crates/newtype_derive provides macros to implement the common cases
  - Why no runtime cost?
    - No information added to the **`struct`**, so same size
    - Passed in the same way – not automatically boxed on the heap
    - Optimiser will recognise that the code collapses down to operations on primitive types, and generate the code to do so
    - All the additions are a compile-time model of the ways the data can be used, they don't affect the compiled code
    - *(Equivalent C++ code has the same properties)*

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

Alternative Types: `enum`

- Use **`enum`** types and pattern matching to model alternatives and options
  - **`Option<T>`**
  - **`Result<T,E>`**
  - Features and response codes
  - Flags

- The compiler can check that options, alternatives, features, and flags are handled correctly – debug before running the code

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

Optional Values: `Option<T>`

- If a value might not exist, use **`Option<T>`**
  - As function return value, pattern matching on result:

```
fn lookup(self, user : Username) -> Option<User>
```

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

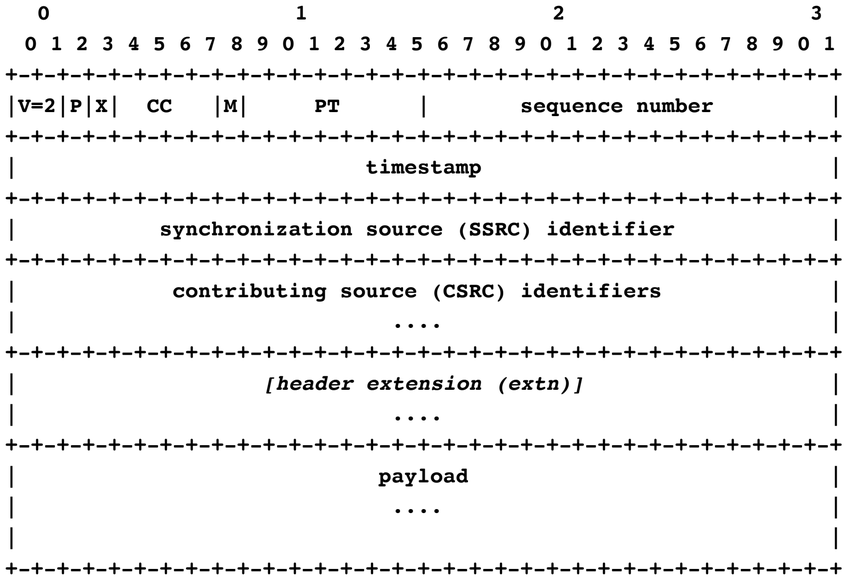
Optional Values: `Option<T>`

- If a value might not exist, use **`Option<T>`**
  - As function return value, pattern matching on result:

```
fn lookup(self, user : Username) -> Option<User>
```

  - In **`struct`** definition:

```
struct RtpHeader {
  v       : Version,
  pt      : PayloadType,
  seq     : SequenceNumber,
  ts      : Timestamp,
  ssrc    : SourceId,
  csrc    : Vec<SourceId>,
  extn    : Option<HeaderExtension>,
  payload : RtpPayload
}
```



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

Optional Values: `Option<T>`

- If a value might not exist, use **`Option<T>`**
  - As function return value, pattern matching on result:

```
fn lookup(self, user : Username) -> Option<User>
```

  - In **`struct`** definition:

```
struct RtpHeader {
  v       : Version,
  pt      : PayloadType,
  seq     : SequenceNumber,
  ts      : Timestamp,
  ssrc    : SourceId,
  csrc    : Vec<SourceId>,
  extn    : Option<HeaderExtension>,
  payload : RtpPayload
}
```

  - Compiler enforces that both variants of **`Option<T>`** are handled
    - **`Some(T)`**, **`None`**
    - Can't accidentally write code that assumes the value is present and crashes otherwise

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

Results: `Result<T,E>`

- The **`Result<T,E>`** type represents results that can fail
- Used as a result type for a function:

```
fn load_document() -> Result<Document, DatabaseError> {
    let db = open_database()?;
    db.load("document_1")?
}

...
match load_document() {
  Ok(doc) => println!(doc), // success
  Err(e)  => ...            // failed
}
```

Use of ? operator for early return on error

Custom error types can be defined: https://doc.rust-lang.org/rust-by-example/error/multiple_error_types/define_error_type.html

- C code frequently uses a signal value to indicate errors
  - e.g., **`socket()`** returns -1 on error, file descriptor >0 on success
  - Success and failure have the same type – easy to forget to check error codes
- With **`Result<T,E>`**, must pattern match to distinguish success and failure type; compiler will check that all error codes are handled

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

Features and Response Codes

- Anti-pattern #1: string typing
  - Method parameters that are strings, rather than some more appropriate type
  - Strings returned from network function (e.g., HTTP response codes) directly used, rather than converted to appropriate type

- Use **`enum`** to represent values that can be one of several alternatives
  - Exhaustiveness checking – catches bugs if new codes introduced
  - Ease of refactoring – decouples code from external representation
  - Make nonsensical values unrepresentable
  - Types are machine checkable documentation – strings are not

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

Flags

- Anti-pattern #2: boolean flags
  - Use of boolean flags obscures code meaning – compare:

```
let f = File.Open(“foo.txt”, true, false);
```

and

```
let f = File.Open(“foo.txt”, FileType::TextMode, FileMode::ReadOnly);
```

  - Further, the **`enum`**-based version won’t compile if the arguments are passed in the wrong order; the **`bool`**-based version will compile, run, and give the wrong results
  - See also:
    - https://www.luu.io/posts/dont-use-booleans/
    - https://wiki.c2.com/?UseEnumsNotBooleans

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

Design Patterns

- Use the type system to describe features of the system design, so the compiler can check for correctness
- There is an up-front cost: you must define the types
- The benefit is that fixing compilation errors is easier than fixing silent data corruption
  - For small systems, the cost may outweigh the benefit
  - For large systems, compiler enforced consistency checks due to use of types can be a significant win

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

Design Patterns

- Specific numeric types
- Enumerations for alternatives

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