Basic Types and Borrowing

`struct`s, `enum`s, pattern matching and the borrow checker

Cooper Pierce

Carnegie Mellon University

Hello, world!

```rust
fn main() -> () {
    let course: i32 = 98008;
    println!("Welcome to {}!", course);
}
```

Hello, world!

```
fn main() -> () {
    let course: i32 = 98008;
    println!("Welcome to {}!", course);
}
```

- As in C, the entry point of our program is `main`.

Hello, world!

```rust
fn main() -> () {
    let course: i32 = 98008;
    println!("Welcome to {}!", course);
}
```

- As in C, the entry point of our program is `main`.
- Here we're returning unit, `()`, but we can return a couple different types from `main`.

Hello, world!

```
fn main() -> () {
    let course: i32 = 98008;
    println!("Welcome to {}!", course);
}
```

- As in C, the entry point of our program is `main`.
- Here we're returning unit, `()`, but we can return a couple different types from `main`.
- We can introduce variable bindings with `let`.

Hello, world!

```
fn main() -> () {
    let course: i32 = 98008;
    println!("Welcome to {}!", course);
}
```

- As in C, the entry point of our program is `main`.
- Here we're returning unit, `()`, but we can return a couple different types from `main`.
- We can introduce variable bindings with `let`.
- Optionally, we can type annotate these.

Hello, world!

```
fn main() -> () {
    let course: i32 = 98008;
    println!("Welcome to {}!", course);
}
```

- As in C, the entry point of our program is `main`.
- Here we're returning unit, `()`, but we can return a couple different types from `main`.
- We can introduce variable bindings with `let`.
- Optionally, we can type annotate these.
- Function like macros have a `!` at the end when applying them.

Hello, world!

```
fn main() -> () {
    let course: i32 = 98008;
    println!("Welcome to {}!", course);
}
```

- As in C, the entry point of our program is `main`.
- Here we're returning unit, `()`, but we can return a couple different types from `main`.
- We can introduce variable bindings with `let`.
- Optionally, we can type annotate these.
- Function like macros have a `!` at the end when applying them.
- We can print values using `println!` or `print!` in the same way we would with `printf`.

Mutable variables

In most imperative languages, variables are mutable by default.

```
int fact(int n) {
    int ans = 1;
    while (n) {
        ans *= n;
        n--;
    }
    return ans;
}
```

If we want a variable to not be mutable we have to enforce this with a keyword like `const` or similar.

Rust, on the other hand, flips this. If we try the same in Rust:

```rust
fn fact(n: u32) -> u32 {
    let ans = 1;
    while n != 0 {
        ans *= n;
        n -= 1;
    }
    ans
}
```

we'd see an error like

```
error[E0384]: cannot assign to immutable argument `n`
 --> src/lib.rs:5:17
  |
1 |         fn fact(n: u32) -> u32 {
  |                 - help: consider making this binding mutable: `mut n`
...
5 |                 n -= 1;
  |                 ^^^^^^ cannot assign to immutable argument
```

In order to mark a variable as mutable, we need to have `mut` at the binding site.

```
fn fact(mut n: u32) -> u32 {
    let mut ans = 1;
    while n != 0 {
        ans *= n;
        n -= 1;
    }
    ans
}
```

This then permits later assignments to that binding.

Shadowing

```
fn main() {
    let x = 1;
    println!("x is {}", x);
    let x = 98008;
    println!("x is {}", x);
}
```

What about this code? Does it run afoul of our rules about changing variables?

Shadowing

```
fn main() {
    let x = 1;
    println!("x is {}", x);
    let x = 98008;
    println!("x is {}", x);
}
```

What about this code? Does it run afoul of our rules about changing variables?

No! We haven't changed anything here—there just happens to be a second, new variable we've also called `x`.

While at first this might seem similar to mutating the same variable, there are many semantic differences.

While at first this might seem similar to mutating the same variable, there are many semantic differences.

```
let fruits = ["mango", "apple", "banana"];
let fruits = fruits.len();
```

While at first this might seem similar to mutating the same variable, there are many semantic differences.

```
let fruits = ["mango", "apple", "banana"];
let fruits = fruits.len();
```

Here we've declared two variables which happen to have the same name.

While at first this might seem similar to mutating the same variable, there are many semantic differences.

```
let fruits = ["mango", "apple", "banana"];
let fruits = fruits.len();
```

Here we've declared two variables which happen to have the same name.

```
let mut fruits = ["mango", "apple", "banana"];
fruits = fruits.len();
```

While at first this might seem similar to mutating the same variable, there are many semantic differences.

```
let fruits = ["mango", "apple", "banana"];
let fruits = fruits.len();
```

Here we've declared two variables which happen to have the same name.

```
let mut fruits = ["mango", "apple", "banana"];
fruits = fruits.len();
```

This, to the contrary, results in a compiler error! We can't assign a value of type `usize` to a variable of type `[&str; 3]`.

References and Borrowing

Instead of working directly with pointers (often called “raw” pointers in Rust), we’ll typically use references instead.

```
fn main() {
    let x = 9;
    let y = 2;
    assert_eq!(compute_sum(&x, &y), 11);
}

fn compute_sum(a: &i32, b: &i32) -> i32 {
    a + b
}
```

Mutable References

What if we want to mutate a value through a reference?

```
fn main() {
    let x = 0;
    incr(&x);
    assert_eq!(x, 1);
}
```

```
fn incr(x: &i32) {
    *x += 1
}
```

Mutable References

What if we want to mutate a value through a reference?

```rust
fn main() {
    let x = 0;
    incr(&x);
    assert_eq!(x, 1);
}
```

```rust
fn incr(x: &i32) {
    *x += 1
}
```

Doesn't work!

```
error[E0594]: cannot assign to `*x`, which is behind a `&` reference
 --> src/main.rs:8:13
  |
7 |         fn incr(x: &i32) {
  |                    ---- help: consider changing this to be a mutable reference: `&mut i32`
8 |             *x += 1
  |             ^^^^^^^ `x` is a `&` reference, so the data it refers to cannot be written
```

If we want a mutable reference we need to ask for it explicitly:

```
fn incr(x: &mut i32) {
    *x += 1
}
```

If we want a mutable reference we need to ask for it explicitly:

```
fn incr(x: &mut i32) {
    *x += 1
}
```

and we need to be explicit when borrowing:

```
fn main() {
    let mut x = 0;
    incr(&mut x);
    assert_eq!(x, 1);
}
```

Note that in order to borrow `x` mutably, it has to be mutably bound.

Tuples

One of the simplest types of aggregate data in Rust is a tuple.

```rust
let x: (i32, bool) = (7, true);
```

Tuples

One of the simplest types of aggregate data in Rust is a tuple.

```
let x: (i32, bool) = (7, true);
```

Which we can also destructure into its components via binding:

```
let (i, b) = x;
```

Tuples

One of the simplest types of aggregate data in Rust is a tuple.

```
let x: (i32, bool) = (7, true);
```

Which we can also destructure into its components via binding:

```
let (i, b) = x;
```

or accessed by position:

```
let y = x.0 + 3;
```

Tuples can have many distinct fields, which may themselves be of any type

```
let x = (1, 3e-7, false, "Hello!");
```

and can be returned from functions, or used as arguments

```
fn divmod(n: u32, k: u32) -> (u32, u32) {
    if n < k {
        (0, n)
    } else {
        let (q, d) = divmod(n, n - k);
        (q + 1, d)
    }
}
```

Arrays

Rust also has arrays, which provide for storage for many elements which have the same type. The size of an array must be statically known, and arrays cannot be resized.
We write array types `[T; N]` for an N element array with element type T.

```
let x: [i32; 5] = [0, 1, 2, 3, 4];
let y: [i32; 100] = [0; 100];
```

Arrays

Rust also has arrays, which provide for storage for many elements which have the same type. The size of an array must be statically known, and arrays cannot be resized. We write array types `[T; N]` for an N element array with element type T.

```
let x: [i32; 5] = [0, 1, 2, 3, 4];
let y: [i32; 100] = [0; 100];
```

Accessing an element in the array is fairly standard:

```
y[0] = x[1] + x[3];
assert_eq!(y, 4);
```

What if we index out-of-bounds?

```
let mut x = [1, 2, 3];
x[4] = 7;
```

What if we index out-of-bounds?

```
let mut x = [1, 2, 3];
x[4] = 7;
```

Unlike C, there's no undefined behaviour here! Instead, the program will "panic"—there are some settings for exactly what this means, but by default you'll get a backtrace and the program will terminate.

```
thread 'main' panicked at 'index out of bounds: the len is 1 but the index is 1', src/main.rs:4:5
stack backtrace:
   0: rust_begin_unwind
             at /rustc/db9d1b20bba1968c1ec1fc49616d4742c1725b4b/library/std/src/panicking.rs:498:5
   1: core::panicking::panic_fmt
             at /rustc/db9d1b20bba1968c1ec1fc49616d4742c1725b4b/library/core/src/panicking.rs:107:14
   2: core::panicking::panic_bounds_check
             at /rustc/db9d1b20bba1968c1ec1fc49616d4742c1725b4b/library/core/src/panicking.rs:75:5
   3: playground::main
             at ./src/main.rs:4:5
   4: core::ops::function::FnOnce::call_once
             at /rustc/db9d1b20bba1968c1ec1fc49616d4742c1725b4b/library/core/src/ops/function.rs:227:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
```

Slices

Often in C we might operate on an array by the use of a pointer to its initial element:

```c
int sum(int *x, size_t n) {
    int sum = 0;
    for (size_t i = 0; i < n; ++i) {
        sum += x[i];
    }
    return sum;
}
```

Slices

Often in C we might operate on an array by the use of a pointer to its initial element:

```c
int sum(int *x, size_t n) {
    int sum = 0;
    for (size_t i = 0; i < n; ++i) {
        sum += x[i];
    }
    return sum;
}
```

This is error prone in several ways.

Slices

Often in C we might operate on an array by the use of a pointer to its initial element:

```
int sum(int *x, size_t n) {
    int sum = 0;
    for (size_t i = 0; i < n; ++i) {
        sum += x[i];
    }
    return sum;
}
```

This is error prone in several ways.

- What if `x` is a null pointer?

Slices

Often in C we might operate on an array by the use of a pointer to its initial element:

```
int sum(int *x, size_t n) {
    int sum = 0;
    for (size_t i = 0; i < n; ++i) {
        sum += x[i];
    }
    return sum;
}
```

This is error prone in several ways.

- What if `x` is a null pointer?
- What if `x` doesn't point to `n` elements?

Slices

Often in C we might operate on an array by the use of a pointer to its initial element:

```c
int sum(int *x, size_t n) {
    int sum = 0;
    for (size_t i = 0; i < n; ++i) {
        sum += x[i];
    }
    return sum;
}
```

This is error prone in several ways.

- What if `x` is a null pointer?
- What if `x` doesn't point to `n` elements?
- What if `x` is an otherwise invalid pointer?

We can avoid these issues by using a "slice" type in Rust.

`[T]` is an unsized type representing some contiguous sequence of elements of type `T`—this isn't very useful on its own, because we don't know how big it is!

We can avoid these issues by using a “slice” type in Rust.

`[T]` is an unsized type representing some contiguous sequence of elements of type `T`—this isn't very useful on its own, because we don't know how big it is!

Using a reference, we can get something we do know the size of:

- `&[T]` is the type of shared slices
- `&mut [T]` is the type of mutable/exclusive slices

Both of these will additionally store a length, along with a pointer to the start of the slice.

So if we want to sum an array in Rust, we might instead have:

```rust
fn sum(xs: &[i32]) -> i32 {
    let mut sum = 0;
    for x in xs {
        sum += x;
    }
    sum
}
```

So if we want to sum an array in Rust, we might instead have:

```
fn sum(xs: &[i32]) -> i32 {
    let mut sum = 0;
    for x in xs {
        sum += x;
    }
    sum
}
```

which we could use like so:

```
let x = [1, 2, 3, 4];

assert_eq!(sum(&x[ .. ]), 10);
assert_eq!(sum(&x[1.. ]),  9);
assert_eq!(sum(&x[ ..2]),  3);
```

structs

Like many other languages, Rust supports structs.
We can have traditional, C-style structs:

```
struct Student {
    andrewid: [u8; 8],
    name: String,
    section: char,
}
```

structs

Like many other languages, Rust supports structs.
We can have traditional, C-style structs:

```
struct Student {
    andrewid: [u8; 8],
    name: String,
    section: char,
}
```

or named tuple style structs:

```
struct Fraction(u32, u32);
```

`struct`s

Like many other languages, Rust supports structs.
We can have traditional, C-style structs:

```
struct Student {
    andrewid: [u8; 8],
    name: String,
    section: char,
}
```

or named tuple style structs:

```
struct Fraction(u32, u32);
```

or unit structs:

```
struct Refl;
```

Every field of a struct must be assigned a value when initialising it.

```
let jack = Student {
    andrewid: [b'j', b'r', b'd', b'u', b'v', b'a', b'l', b'l'],
    name: "Jack Duvall",
    section: 'A',
};
```

Every field of a struct must be assigned a value when initialising it.

```
let jack = Student {
    andrewid: [b'j', b'r', b'd', b'u', b'v', b'a', b'l', b'l'],
    name: "Jack Duvall",
    section: 'A',
};
```

If there are local variables with the same name, we can shortcut this somewhat:

```
// Dereference because this gives a slice
let andrewid = *b"cppierce";
let name = "Cooper Pierce";
let section = 'A';
let cooper = Student { andrewid, name, student };
```

Member access for structs is similar to C, with the exception of eliminating `->`.

```
assert_ne(cooper.andrewid, jack.andrewid);

let s = &cooper;
assert_eq(cooper.name, s.name);
```

enums

Rust also has enums. C-style "named constants" like

```
enum Weekday {
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday
}
```

enums

Rust also has enums. C-style “named constants” like

```
enum Weekday {
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday
}
```

There are kept in their own namespace:

```
let today = Weekday::Wednesday;
```

And also more functionally-inspiried ones with data:

```
enum Number {
    Rational { numer: u32, denom: u32, sign: bool }
    Float(f64),
    Int(i32),
    Infinity,
}
```

And also more functionally-inspiried ones with data:

```
enum Number {
    Rational { numer: u32, denom: u32, sign: bool }
    Float(f64),
    Int(i32),
    Infinity,
}
```

Which we can use similarily:

```
let f = Number::Float(1.6);
let r = Number::Rational { numer: 3, denom: 8, sign: true };
```

And also more functionally-inspiried ones with data:

```
enum Number {
    Rational { numer: u32, denom: u32, sign: bool }
    Float(f64),
    Int(i32),
    Infinity,
}
```

Which we can use similarily:

```
let f = Number::Float(1.6);
let r = Number::Rational { numer: 3, denom: 8, sign: true };
```

What if we used an enum for `sign`?

impl blocks

We can add associated functions and methods to a struct or enum we've defined by using an impl block.

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

```
impl Rectangle {
    fn unit() -> Self {
        Self { width: 1, height: 1 }
    }

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

Invoking an associated function is done by qualifying it with the type

```rust
let unit_square = Rectangle::unit();
```

Invoking an associated function is done by qualifying it with the type

```rust
let unit_square = Rectangle::unit();
```

and methods are typically invoked using a dot:

```rust
let r = Rectangle { width: 4, height: 7 };
assert_eq!(unit_square.area(), 1);
assert_eq!(r.area(), 28);
```

`if` expressions

Similar to many functional programming languages, `if` does not introduce a statement, but instead an expression.

`if` expressions

Similar to many functional programming languages, `if` does not introduce a statement, but instead an expression.

So while we can do

```
let x;
if some_condition {
    x = 7;
} else {
    x = 9
}
```

You'd typically see

```
let x = if some_condition { 7 } else { 9 };
```

If we omit the else branch the if branch must evaluate to unit—`()`

```
if user.is_admin() {
    println!("Hello administator!");
}
```

If we omit the else branch the if branch must evaluate to unit—`()`

```
if user.is_admin() {
    println!("Hello administator!");
}
```

Note that any expression followed by a semicolon will be an expression which discards the result and evaluates to unit.

while loops

We have the typical while loop:

```
fn exp(mut n: i32) -> i32 {
    let mut b = 2;
    let mut x = 1;
    while n > 1 {
        if n % 2 == 1 {
            x = x * b;
        }
        b *= b;
        n /= 2;
    }
    x * b
}
```

`for` loops

and iterator-based for loops:

```
let nums = [1, 2, 3, 4, 5];
for n in nums {
    println!("{}", n);
}
```

for loops

and iterator-based for loops:

```
let nums = [1, 2, 3, 4, 5];
for n in nums {
    println!("{}", n);
}
```

Range types are often useful here:

```
for i in 0..n {
    println("{} squared is {}", i, i * i);
}
```

`loop` loops

In addition, we also have an unconditional loop construct:

```
loop {
    println!("Hi again!");
}
```

`loop` loops

In addition, we also have an unconditional loop construct:

```
loop {
    println!("Hi again!");
}
```

This is more useful when using `break`

```
let prime = loop {
    let p = gen_random_number();
    if miller_rabin(p) {
        break p;
    }
};
```

match expressions

What if we want to deal with many possible branching choices for an expression?

```
fn fib(n: u32) -> u32 {
    match n {
        0 | 1 => 0,
        n => fib(n - 1) + fib(n - 2),
    }
}
```

This is a bit more useful when dealing with enums

```rust
enum Coin { Penny, Nickel, Dime, Quarter }

impl Coin {
    fn value(&self) -> u32 {
        match self {
            Coin::Penny   =>  1,
            Coin::Nickel  =>  5,
            Coin::Dime    => 10,
            Coin::Quarter => 25,
        }
    }
}
```

Most of all when the enum has data

```
enum Transmission {
    Incoming(String)
    Done,
}

fn listen(&mut p: Port) {
    loop {
        match p.receive() {
            Transmission::Incoming(s) => {
                println!(s);
            }
            Done => return,
        }
    }
}
```

Sometimes we can employ more specific pattern matching constructs to simplfy code.

Sometimes we can employ more specific pattern matching constructs to simplfy code.

```
enum Transmission {
    Incoming(String)
    Done,
}

fn listen(&mut p: Port) {
    while let Transmission::Incoming(s) = p.receive() {
        println!(s);
    }
}
```

Likewise, there's also `if let`. However, you'll essentially always want to use `match` if you have two or more things to do.

Reference Pitfalls

In many other languages with references (e.g., C++) there are a number of potential pitfalls:

```
int main() {
    auto v = std::vector<int>{1, 2, 3, 4};
    auto x = &v[1];
    v.push_back(5);
    *x = 0;
    std::cout << v[1] << std::endl;
    return 0;
}
```

What's wrong?

Reference Pitfalls

In many other languages with references (e.g., C++) there are a number of potential pitfalls:

```
int main() {
    auto v = std::vector<int>{1, 2, 3, 4};
    auto x = &v[1];
    v.push_back(5);
    *x = 0;
    std::cout << v[1] << std::endl;
    return 0;
}
```

What's wrong?

By changing `v`, we invalidate the reference `x`!

Rules for Borrowing

In Rust, this *cannot happen*, because borrowing has restrictions:

Rules for Borrowing

In Rust, this *cannot happen*, because borrowing has restrictions:

- Every value has an “owner”.

Rules for Borrowing

In Rust, this *cannot happen*, because borrowing has restrictions:

- Every value has an "owner".
- There can only be one owner.

Rules for Borrowing

In Rust, this *cannot happen*, because borrowing has restrictions:

- Every value has an "owner".
- There can only be one owner.
- When ownership of the value ends, the value will be "dropped" (think deallocated/destructed).

Rules for Borrowing

In Rust, this *cannot happen*, because borrowing has restrictions:

- Every value has an "owner".
- There can only be one owner.
- When ownership of the value ends, the value will be "dropped" (think deallocated/destructed).
- You can have as many shared borrows (`&`) as you want, all at the same time . . .

Rules for Borrowing

In Rust, this *cannot happen*, because borrowing has restrictions:

- Every value has an "owner".
- There can only be one owner.
- When ownership of the value ends, the value will be "dropped" (think deallocated/destructed).
- You can have as many shared borrows (`&`) as you want, all at the same time . . .
- . . . but, you can only have one exclusive borrow (`&mut`), and not at the same time as any shared borrow.