# Standard Library

## CIS 198 Lecture 5

# String Types

- Rust strings are complicated.
  - Sequences of Unicode values encoded in UTF-8.
  - Not null-terminated and may contain null bytes.
- There are two kinds: `&str` and `String`.

# `&str`

- `&str` is a string slice (like array slice).
- `"string literals"` are of type `&str`.[1]
- `&str`s are statically-allocated and fixed-size.
- May not be indexed with `some_str[i]`, as each character may be multiple bytes due to Unicode.
- Instead, iterate with `chars()`:
  - `for c in "1234".chars() { ... }`
- As with all Rust references, they have an associated lifetime.

[1]More specifically, they have the type `&'static str`.

# `String`

- `String`s are heap-allocated, and are dynamically growable.
  - Like `Vec`s in that regard.
  - In fact, `String` is just a wrapper over `Vec<u8>`!
- Cannot be indexed either.
  - You can select characters with `s.nth(i)`.
- May be coerced into an `&str` by taking a reference to the `String`.

```
let s0: String = String::new();
let s1: String = "foo".to_string();
let s2: String = String::from("bar");
let and_s: &str = &s0;
```

# `str`

- If `&str` is the second string type, what exactly is `str`?
- An `Unsized` type, meaning the size is unknown at compile time.
    - You can't have bindings to `str`s directly, only references.

# String Concatenation

- A `String` and an `&str` may be concatenated with `+`:

```
let course_code = "CIS".to_string();
let course_name = course_code + " 198";
```

- Concatenating two `String`s requires coercing one to `&str`:

```
let course_code = String::from("CIS");
let course_num  = String::from(" 198");
let course_name = course_code + &course_num;
```

- You can't concatenate two `&str`s.

```
let course_name = "CIS " + "198"; // Err!
```

# String Conversion

- However, *actually* converting a `String` into an `&str` requires a dereference:

```rust
use std::net::TcpStream;

TcpStream::connect("192.168.0.1:3000"); // &str
let addr = "192.168.0.1:3000".to_string();
TcpStream::connect(&*addr);
```

- This doesn't automatically coerce because `TcpStream` doesn't take an argument of type `&str`, but a Trait bounded type:
    - `TcpStream::connect<A: ToSocketAddr>(addr: A);`

# Aside: `Deref` Coercions

- Rust's automatic dereferencing behavior works *between* types as well.

```
pub trait Deref {
    type Target: ?Sized;
    fn deref(&self) -> &Self::Target;
}
```

- Since `String` implements `Deref<Target=str>`, so values of `&String` will automatically be dereferenced to `&str` when possible.

# `String` & `&str`: Why?

- Like slices for `Vec`s, `&str`s are useful for passing a view into a `String`.
- It's expensive to copy a `String` around, and lending an entire `String` out may be overkill.
- `&str` therefore allows you to pass portions of a `String` around, saving memory.

# `String` & `&str`: Why?

- Generally, if you want to do more than use string literals, use `String`.
  - You can then lend out `&str`s easily.

# `Option<T>`

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

- Provides a concrete type to the concept of *nothingness*.
- Use this instead of returning `NaN`, `-1`, `null`, etc. from a function.
- No restrictions on what `T` may be.

# `Option::unwrap()`

- The pattern where None values are ignored is pretty common:

```
// fn foo() -> Option<i32>

match foo() {
    None => None,
    Some(value) => {
        bar(value)
        // ...
    },
}
```

# Option::unwrap()

- What if we extracted the pattern match into a separate function to simplify it?

```
fn unwrap<T>(&self) -> T { // 🎁!
    match *self {
        None => panic!("Called `Option::unwrap()` on a `None` value")
        Some(value) => value,
    }
}

let x = foo().unwrap();
let y = bar(x);
// ...
```

- Unfortunately, `panic!`ing on `None` values makes this abstraction inflexible.
- Better: use `expect(&self, msg: String) -> T` instead.
    - `panic!`s with a custom error message if a `None` value is found.

# `Option::map()`

- Let's make the pattern a little better.
- We'll take an `Option`, change the value if it exists, and return an `Option`.
    - Instead of failing on `None`, we'll keep it as `None`.

```rust
fn map<U, F>(self, f: F) -> Option<U>
        where F: FnOnce(T) -> U {
    match self {
        None => None,
        Some(x) => Some(f(x))
    }
}

// fn foo() -> Option<i32>

let x = foo().map(|x| bar(x));
```

# Option::and_then()

- There's a similar function `and_then`:

```
fn and_then<U, F>(self, f: F) -> Option<U>
      where F: FnOnce(T) -> Option<U> {
    match self {
        Some(x) => f(x),
        None => None,
    }
}

// fn foo() -> Option<i32>

let x = foo().and_then(|x| Some(bar(x)));
```

- Notice the type of `f` changes from `T -> U` to `T -> Some(U)`.

# Option::unwrap_or()

- If we don't want to operate on an `Option` value, but it has a sensible default value, there's `unwrap_or`.

```
impl<T> Option<T> {
    fn unwrap_or<T>(&self, default: T) -> T {
      match *self {
          None => default,
          Some(value) => value,
      }
    }
}
```

# Option::unwrap_or_else()

- If you don't have a static default value, but you can write a closure to compute one:

```
impl<T> Option<T> {
    fn unwrap_or_else<T>(&self, f: F) -> T
            where F: FnOnce() -> T {
        match *self {
            None => f(),
            Some(value) => value,
        }
    }
}
```

# Other

- Some other methods provided by Option:
- `fn is_some(&self) -> bool`
- `fn is_none(&self) -> bool`
- `fn map_or<U, F>(self, default: U, f: F) -> U`
  - `where F: FnOnce(T) -> U`
  - A default value`: U`.
- `fn map_or_else<U, D, F>(self, default: D, f: F) -> U`
  - `where D: FnOnce() -> U, F: FnOnce(T) -> U`
  - A default-generating closure`: D`.

# Other

- `fn ok_or(self, err: E) -> Result<T, E>`
- `fn ok_or_else(self, default: F) -> Result<T, E>`
  - `where F: FnOnce() -> E`
  - Similar to `unwrap_or` but returns a `Result` with a default `Err` or closure.
- `fn and<U>(self, optb: Option<U>) -> Option<U>`
  - Returns `None` if `self` is `None`, else `optb`
- `fn or(self, optb: Option<T>) -> Option<T>`
  - returns `self` if `self` is `Some(_)`, else `optb`

# Result

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

- `Result` is like `Option`, but it also encodes an `Err` type.
- Also defines `unwrap()` and `expect()` methods.
- Can be converted to an `Option` using `ok()` or `err()`.
    - Takes either `Ok` or `Err` and discards the other as `None`.
- Can be operated on in almost all the same ways as `Option`
    - `and`, `or`, `unwrap`, etc.

# Result

- Unlike `Option`, a `Result` should *always* be consumed.
  - If a function returns a `Result`, you should be sure to `unwrap`/`expect` it, or otherwise handle the `Ok`/`Err` in a meaningful way.
  - The compiler warns you if you don't.
  - Not using a result could result (ha) in your program unintentionally crashing!

# Custom Result Aliases

- A common pattern is to define a type alias for Result which uses your libary's custom Error type.

```
use std::io::Error;

type Result<T> = Result<T, Error>;
```

- Typically a convenience alias; other than fixing `E = Error`, this is identical to `std::Result`.
- Users of this type should namespace it:

```
use std::io;

fn foo() -> io::Result {
    // ...
}
```

# Result - `try!`

- `try!` is a macro, which means it generates Rust's code at compile-time.
    - This means it can actually expand to pattern matching syntax patterns.
- The code that `try!` generates looks roughly like this:

```
macro_rules! try {
    ($e:expr) => (match $e {
        Ok(val) => val,
        Err(err) => return Err(err),
    });
}
```

# `try!`

- `try!` is a concise way to implement early returns when encountering errors.

```
let socket1: TcpStream = try!(TcpStream::connect("127.0.0.1:8000"));

// Is equivalent to...
let maybe_socket: Result<TcpStream> =
    TcpStream::connect("127.0.0.1:8000");
let socket2: TcpStream =
    match maybe_socket {
        Ok(val) => val,
        Err(err) => { return Err(err) }
    };
```

- This is actually a *slight* simplification.
    - Actual `try!` has some associated trait ~~magic~~logic.

# Collections

# Vec<T>

- Nothing new here.

# VecDeque<T>

- An efficient double-ended `Vec`.
- Implemented as a ring buffer.

# `LinkedList<T>`

- A doubly-linked list.
- Even if you want this, you probably don't want this.
  - Seriously, did you even read *any* of Gankro's book?

# `HashMap<K,V>`/`BTreeMap<K,V>`

- Map/dictionary types.
- `HashMap<K, V>` is useful when you want a basic map.
  - Requires that `K: Hash + Eq`.
  - Uses "linear probing with Robin Hood bucket stealing".
- `BTreeMap<K, V>` is a sorted map (with slightly worse performance).
  - Requires that `K: Ord`.
  - Uses a B-tree under the hood (surprise surprise).

# `HashSet<T>`/`BTreeSet<T>`

- Sets for storing unique values.
- `HashSet<T>` and `BTreeSet<T>` are literally struct wrappers for `HashMap<T, ()>` and `BTreeMap<T, ()>`.
- Same tradeoffs and requirements as their Map variants.

# `BinaryHeap<T>`

- A priority queue implemented with a binary max-heap.

# Aside: Rust Nursery

- Useful "stdlib-ish" crates that are community-developed, but not official-official.
- Contains things like:
  - Bindings to `libc`
  - A `rand` library
  - Regex support
  - Serialization
  - UUID generation

# Iterators

- You've seen these in HW3!

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;

    // More fields omitted
}
```

- A Trait with an associated type, `Item`, and a method `next` which yields that type.
- Other methods (consumers and adapters) are implemented on `Iterator` as default methods using `next`.

# Iterators

- Like everything else, there are three types of iteration:
  - `into_iter()`, yielding `T`s.
  - `iter()`, yielding `&T`s.
  - `iter_mut()`, yielding `&mut T`s.
- A collection may provide some or all of these.

# Iterators

- Iterators provide syntactic sugar for for loops:

```
let values = vec![1, 2, 3, 4, 5];
{
    let result = match values.into_iter() {
        mut iter => loop {
            match iter.next() {
                Some(x) => { /* loop body */ },
                None => break,
            }
        },
    };
    result
}
```

- `into_iter()` is provided by the trait `IntoIterator`.
    - Automatically implemented by anything with the Trait `Iterator`.

# IntoIterator

```
pub trait IntoIterator where Self::IntoIter::Item == Self::Item {
    type Item;
    type IntoIter: Iterator;

    fn into_iter(self) -> Self::IntoIter;
}
```

- As you did in HW3, you can implement `IntoIterator` on a `&T` to iterate over a collection by reference.
  - Or on `&mut T` to iterate by mutable reference.
- This allows this syntax:

```
let ones = vec![1, 1, 1, 1, 1, 1];

for one in &ones {
    // Doesn't move any values.
    // Also, why are you doing this?
}
```

# Iterator Consumers

- Consumers operate on an iterator and return one or more values.
- There are like a billion of these, so let's look at a few.

# Preface: Type Transformations

- Many iterator manipulators take an `Iterator` and return some other type.
  - e.g. `map` returns a `Map`, `filter` returns a `Filter`.
- These types are just structs which themselves implement `Iterator`.
  - Don't worry about the internal state.
- The type transformations are used mostly to enforce type safety.

# collect

- `collect()` rolls a (lazy) iterator back into an actual collection.
- The target collection must define the `FromIterator` trait for the `Item` inside the `Iterator`.
- `collect()` sometimes needs a type hint to properly compile.
  - The output type can be practically any collection.

```
fn collect<B>(self) -> B where B: FromIterator<Self::Item>

let vs = vec![1,2,3,4];
// What type is this?
let set = vs.iter().collect();
// Hint to `collect` that we want a HashSet back.
// Note the lack of an explicit <i32>.
let set: HashSet<_> = vs.iter().collect();
// Alternate syntax! The "turbofish" ::<>
let set = vs.iter().collect::<HashSet<_>>();
```

# `fold`

```rust
fn fold<B, F>(self, init: B, f: F) -> B
    where F: FnMut(B, Self::Item) -> B;

let vs = vec![1,2,3,4,5];
let sum = vs.iter().fold(0, |acc, &x| acc + x);
assert_eq!(sum, 15);
```

- `fold` "folds up" an iterator into a single value.
  - Sometimes called `reduce` or `inject` in other languages.
- `fold` takes two arguments:
  - An initial value or "accumulator" (`acc` above) of type `B`.
  - A function that takes a `B` and the type inside the iterator (`Item`) and returns a `B`.
- Rust doesn't do tail-recursion, so `fold` is implemented iteratively.
  - See here if you're interested why.

# `filter`

```
fn filter<P>(self, predicate: P) -> Filter<Self, P>
    where P: FnMut(&Self::Item) -> bool;
```

- `filter` takes a predicate function `P` and removes anything that doesn't pass the predicate.
- `filter` returns a `Filter<Self, P>`, so you need to `collect` it to get a new collection.

# `find` & `position`

```
fn find<P>(&mut self, predicate: P) -> Option<Self::Item>
    where P: FnMut(Self::Item) -> bool;

fn position<P>(&mut self, predicate: P) -> Option<usize>
    where P: FnMut(Self::Item) -> bool;
```

- Try to find the first item in the iterator that matches the `predicate` function.
- `find` returns the item itself.
- `position` returns the item's index.
- On failure, both return a `None`.

# `skip`

```
fn skip(self, n: usize) -> Skip<Self>;
```

- Creates an iterator that skips its first `n` elements.

# `zip`

```rust
fn zip<U>(self, other: U) -> Zip<Self, U::IntoIter>
    where U: IntoIterator;
```

- Takes two iterators and zips them into a single iterator.
- Invoked like `a.iter().zip(b.iter())`.
  - Returns pairs of items like `(ai, bi)`.
- The shorter iterator of the two wins for stopping iteration.

# `any` & `all`

```
fn any<F>(&mut self, f: F) -> bool
    where F: FnMut(Self::Item) -> bool;

fn all<F>(&mut self, f: F) -> bool
    where F: FnMut(Self::Item) -> bool;
```

- `any` tests if any element in the iterator matches the input function
- `all` tests all elements in the iterator match the input function
- Logical OR vs. logical AND.

# `enumerate`

```
fn enumerate(self) -> Enumerate<Self>;
```

- Want to iterate over a collection by item and index?
- Use `enumerate`!
- This iterator returns `(index, value)` pairs.
    - `index` is the `usize` index of `value` in the collection.

# Iterator Adapters

- Adapters operate on an iterator and return a new iterator.
- Adapters are often *lazy* -- they don't evaluate unless you force them to!
- You must explicitly call some iterator consumer on an adapter or use it in a `for` loop to cause it to evaluate.

# map

```
fn map<B, F>(self, f: F) -> Map<Self, F>
    where F: FnMut(Self::Item) -> B;

let vs = vec![1,2,3,4,5];
let twice_vs: Vec<_> = vs.iter().map(|x| x * 2).collect();
```

- `map` takes a function and creates an iterator that calls the function on each element
- Abstractly, it takes a `Collection<A>` and a function of `A -> B` and returns a `Collection<B>`
    - (`Collection` is not a real type)

# take & take_while

```
fn take(self, n: usize) -> Take<Self>;

fn take_while<P>(self, predicate: P) -> TakeWhile<Self, P>
    where P: FnMut(&Self::Item) -> bool;
```

- `take` creates an iterator that yields its first `n` elements.
- `take_while` takes a closure as an argument, and iterates until the closure returns `false`.
- Can be used on infinite ranges to produce finite enumerations:

```
for i in (0..).take(5) {
    println!("{}", i); // Prints 0 1 2 3 4
}
```

# `cloned`

```
fn cloned<'a, T>(self) -> Cloned<Self>
    where T: 'a + Clone, Self: Iterator<Item=&'a T>;
```

- Creates an iterator which calls `clone` on all of its elements.
- Abstracts the common pattern `vs.iter().map(|v| v.clone())`.
- Useful when you have an iterator over `&T`, but need one over `T`.

# `drain`

- Not actually an `Iterator` method, but is very similar.
- Calling `drain()` on a collection removes and returns some or all elements.
- e.g. `Vec::drain(&mut self, range: R)` removes and returns a range out of a vector.

# Iterators

- There are many more `Iterator` methods we didn't cover.
- Take a look at the docs for the rest.