Higher-Order Programming

- Definition: In higher-order programming, one can pass functions as arguments to other functions and functions can be the return value of other functions.
- This, just like patterns matching, is a concept that was developed for functional programming languages.
- Actually, the idea of functions accepting functions as parameters and/or returning functions is even older than that…
  - it dates back to the 1930's when Alan Turing and Alfonso Church explored what computation actually is. Turing did this with Turing machines and Church did this with lambda-calculus – a computational model which had functions as its primitive notion rather than a state machine as in the Turing machines.
  - The lambda calculus can be considered a precursor to our modern notion of functional programming languages.

Higher-Order Programming

- We can use higher-order programming to customize the behavior of functions
- Consider a function that computes multiples
  - We can customize the behavior of this function by passing in functions with specific behavior

Higher-Order Programming

```rust
fn unit (x: i64) -> i64 {
    x
}

fn double (x: i64) -> i64 {
    2*x
}

fn triple(x: i64) -> i64 {
    3*x
}

fn multiples(f:fn(i64)->i64, q:i64) -> i64 {
    f(q)
}

fn main() {
    println!("{}", multiples(unit, 3));
    println!("{}", multiples(double, 3));
    println!("{}", multiples(triple, 3));
}
```

Terminal

```
ubuntu$ ./multiples
3
6
9
ubuntu$
```

Higher-Order Programming

- One consequence of higher-order programming is you will wind up with the definition of a lot of small functions
- Rather than defining a full fledge function for these simple computation use **closures** (also called lambda functions)
- In Rust closures are themselves functions that can be used to compute values:

```
fn main() {
    let result = (|i:i64|->i64{2*i})(3);
    println!("{}",result)
}
```

Higher-Order Programming

```
fn multiples(f:fn(i64)->i64, q:i64) -> i64 {
    f(q)
}

fn main() {
    println!("{}", multiples(|i:i64|->i64 {i}, 3));
    println!("{}", multiples(|i|2*i, 3));
    println!("{}", multiples(|i|3*i, 3));
}
```

Abbreviated form of a closure

Higher-Order Programming

- As the paradigm definition suggests we can also **return functions from functions**
- This again allows us to write general code that can be specialized using higher-order functions
- Consider a function that returns a specific computation based on some input parameters.

Higher-Order Programming

```rust
enum Multipliers {
    Unit,
    Double,
}

fn multiples(what: Multipliers) -> fn(i64)->i64 {
    use Multipliers::*;
    match what {
        Unit => |x:i64|->i64{x},
        Double => |x:i64|->i64{2*x},
    }
}

fn main() {
    use Multipliers::*;

    let f = multiples(Unit);
    println!("{}",f(3));

    println!("{}",multiples(Double)(3));
}
```

Terminal

```
ubuntu$ ./return_func
3
6
ubuntu$
```

Higher-Order Programming

- In the functional programming language context higher-order programming also gives us access to **partially evaluated functions**
- Unfortunately this is not available in Rust but we will see this when we discuss Haskell
- Partially evaluated functions are a powerful tool when customizing code from a given library
- An interesting application of higher-order programming are iterators

Iterators

- Definition: An iterator is an object that enables a programmer to traverse a container, particularly lists.

Iterators

- Typically we classify iterators into two classes:
  - **Internal iterators**: Internal iterators are higher order functions implementing the traversal across a container, applying the given function to every element in turn.
  - **External iterators**: An external iterator may be thought of as a type of pointer that has two primary operations: referencing one particular element in a collection (called *element access*) and modifying itself so it points to the next element (called *element traversal*).

Iterators

- Here is a basic Rust external iterator:
  - Element access
  - Forward to next element

```
fn main() {
    let a = [1,2,3];
    let a_iter = a.iter();

    for e in a_iter {
        println!("{}",e);
    }
}
```

Iterators

- We can also use iterators to modify the container

```rust
fn main() {
    let mut a = [1,2,3];
    let a_iter = a.iter_mut();

    for e in a_iter {
        *e += 1;
    }
    println!("{:?}",a);
}
```

Iterators

- In Rust internal iterators sit on top of external iterators
  - This is not true for many other languages
  - We will see true internal iterators in Haskell

Iterators

- A basic internal iterator program
- Notice, no explicit iteration needed
- Note: we needed to switch from arrays to vectors because ‘collect’ only supports vectors.

```
fn main() {
    let mut a: Vec<i32> = [1, 2, 3].to_vec();

    a = a.iter()
        .map(|&x| x * 2)
        .collect();

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

Rust Summary

- Rust is truly a modern programming language combining various programming paradigms
  - Objects with Traits no inheritance
  - Generics (bounded parametric polymorphism)
  - Higher-order programming
  - Pattern matching
- Rust also implements a novel way to view memory management alleviating some of the pains in C/C++ programming yet it still remains competitive in terms of runtime speeds.