

Higher-Order Programming

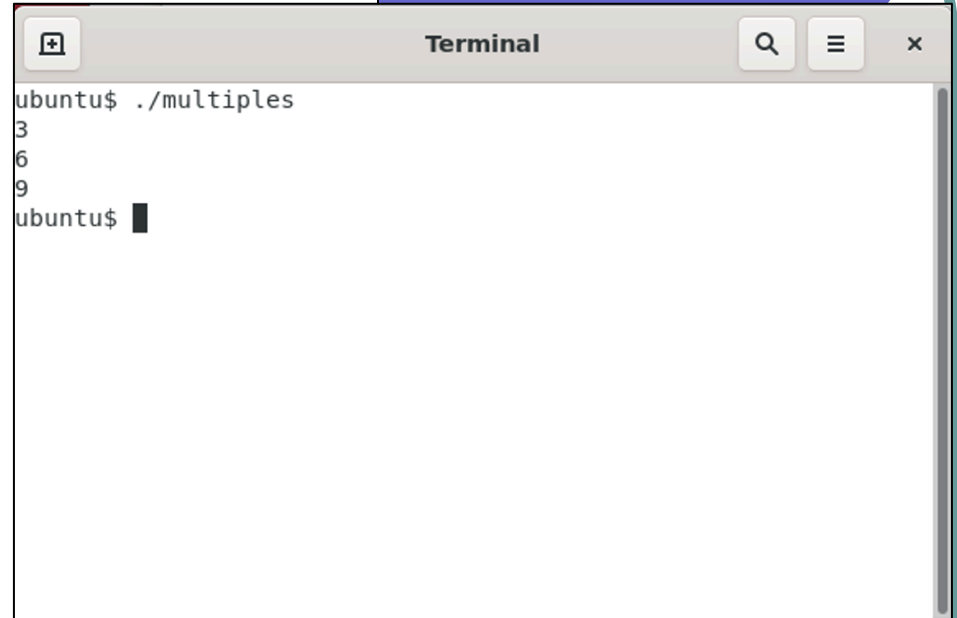
- Definition: In higher-order programming, one can pass functions as arguments to other functions and functions can be the return value of other functions.
- This, just like patterns matching, is a concept that was developed for functional programming languages.
- Actually, the idea of functions accepting functions as parameters and/or returning functions is even older than that...
 - it dates back to the 1930's when Alan Turing and Alonzo 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

```
1 fn unit (x: i64) -> i64 {
2     x
3 }
4
5 fn double (x: i64) -> i64 {
6     2*x
7 }
8
9 fn triple(x: i64) -> i64 {
10    3*x
11 }
12
13 fn multiples(f:fn(i64)->i64, q:i64) -> i64 {
14    f(q)
15 }
16
17 fn main() {
18     println!("{}", multiples(unit, 3));
19     println!("{}", multiples(double, 3));
20     println!("{}", multiples(triple, 3));
21 }
```




A terminal window titled "Terminal" with search, menu, and close buttons. It shows the command `ubuntu$./multiples` being executed, followed by the output `3`, `6`, and `9` on separate lines. The prompt `ubuntu$` is visible at the bottom.

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

Higher-Order Programming

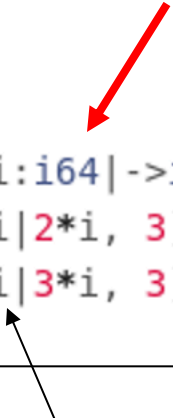
- One consequence of higher-order programming is you will wind up with the definition of a lot of small functions
- Rather than defining a full fledged functions for these simple computation use **closures** (also called lambda functions)
- In Rust closures are fully fledged functions that can be used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

```
1 fn main() {  
2     let result = (|i:i64|->i64{2*i})(3);  
3     println!("{}",result)  
4 }
```

A red arrow points from the top right towards the lambda function syntax `|i:i64|->i64{2*i}` in the second line of the code block.

Higher-Order Programming

```
1  fn multiples(f:fn(i64)->i64, q:i64) -> i64 {
2      f(q)
3  }
4
5  fn main() {
6      println!("{}", multiples(|i:i64|->i64 {i}, 3));
7      println!("{}", multiples(|i|2*i, 3));
8      println!("{}", multiples(|i|3*i, 3));
9  }
```



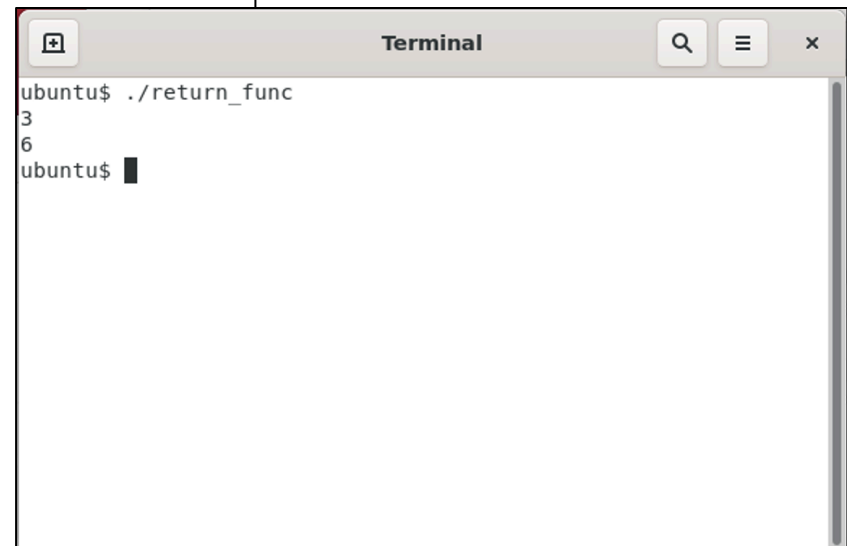
Abbreviated form of a closure

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

```
1 enum Multipliers {  
2     Unit,  
3     Double,  
4 }  
5  
6 fn multiples(what: Multipliers) -> fn(i64)->i64 {  
7     use Multipliers::*;  
8     match what {  
9         Unit => |x:i64|->i64{x},  
10        Double => |x:i64|->i64{2*x},  
11    }  
12 }  
13  
14 fn main() {  
15     use Multipliers::*;  
16  
17     let f = multiples(Unit);  
18     println!("{}", f(3));  
19  
20     println!("{}", multiples(Double)(3));  
21 }
```



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

```
1  fn main() {  
2      let a = [1,2,3];  
3      let a_iter = a.iter();  
4  
5      for e in a_iter {  
6          println!("{}",e);  
7      }  
8  }
```

Iterators

- We can also use iterators to modify the container

```
1  fn main() {  
2      let mut a = [1,2,3];  
3      let a_iter = a.iter_mut();  
4  
5      for e in a_iter {  
6          *e += 1;  
7      }  
8      println!("{:?}", a);  
9  }
```

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.

```
1  fn main() {  
2      let mut a: Vec<i32> = [1, 2, 3].to_vec();  
3  
4      a = a.iter()  
5          .map(|&x| x * 2)  
6          .collect();  
7  
8      println!("{:?}", a);  
9  }
```

Assignments

- Assignment #3