- Closures allow us to trap auxiliary arguments or data so that they don't need to be specified by the user
- more elegant than and less confusing than parameters of a function with a default value.
- But where they really become essential is when our function can update the value of the "hidden" variables
- When the function has state that gets updated across separate calls to that function (instance)

- For those of you who know C, closures have a similarity to static variables within a routine.
  - value persists across calls to routine, and any changes persist.

```
int
numTimesCalled()
{
  static int count = 0;
  count = count + 1;
  return(count);
}
```

Dynamic Programming

- Use results from previous computations when doing subsequent computations

- Consider, e.g., the Fibonacci sequence
  $F(n) = F(n-1) + F(n-2)$
  $F(0) = 1, F(1) = 1$

```
generator =
function()
{
  .Fibonacci<- c(1, 1)
   function(n) {
      curMax = length(.Fibonacci)
      if(curMax >= n)
        return(.Fibonacci[n])

      for(i in seq(curMax + 1, n))
        .Fibonacci[i] <<- .Fibonacci[i - 1]  + .Fibonacci[i - 2]

      .Fibonacci[n]
    }
}
fibonacci = generator()
```

```
fib = generator()

fib(4)
fib(8)

environment(fib)$.Fibonacci
[1]  1  1  2  3  5  8 13 21
```

Processing big XML files

- Want to get the times of change for each topic in Wikipedia

- Wikipedia is an 11G XML file
  Read into memory as a tree, then use XPath
  Very slow, may not even be feasible.

- So we need to use a different approach that doesn't create the tree in memory, but instead reads the XML as a stream reporting "events" such as
  - at the start of an element
  - closing an element
  - reading text
  - comment...

SAX - Simple API for XML

- This is an "event" driven style of programming.
  We give the XML function a collection of functions
  associated with the different events.

- The XML parser calls the appropriate function when
  each event occurs, passing information about the event,
  e.g. the name and attributes of the node
  the content of the "text"
  name of the

```
create =
function(verbose = FALSE)
{
  times <- character()

  inTotalTimeElement = FALSE

   # called for start of <Total_Time> element, so always set inTotalTime to TRUE
  tt = function(name, ...)  {

     if(verbose) cat("In tt\n")

     inTotalTimeElement <<- TRUE
  }

       # If we are in a Total_Time element, put the x into the times vector
  collect =  function(x) {
     if(verbose) cat("in collect\n")

     if(inTotalTimeElement)
         times <<- c(times, x)
  }

      # Need to turn inTotalTimeElement off if it is on.
  endElement = function(name, ...) {

     if(name == "Total_Time")
       inTotalTimeElement <<- FALSE
  }

     # return a list of functions which are used by the SAX parser and also .result
which gives us the values
  list("Total_Time" = tt,
       .endElement = endElement,
       .text = collect,
       .result = function()
                   as.numeric(times))
}
```

- h = create()
- o = xmlEventParse("http://eeyore.ucdavis.edu/itunes-short.xml", handlers = h, saxVersion = 2)
- h$.result()

- Analogous to classes with methods in Python, Java, C++
  - A class defines the nature of each instance
    - data (variables) and
    - methods which can access and update those and those updates are available in calls to subsequent methods.
- In these languages, we are always dealing with a reference to a variable within object, not a copy.
- But in R, we always deal with copies of values but closures allow us to have mutable state.

```
createNormal =
function(mu = 0, sd = 1)
{

   sample =
     function(num = 1)
        rnorm(num, mu, sd)

   shift =
     function(to)
        mu <<- to

   scale =
     function(by)
         sd <<- by

   list(sample = sample, shift = shift, scale = scale)
}
```

```
funs = createNormal()

funs$sample()
funs$sample(10)

funs$shift(-10)
mean(funs$sample(1000))

funs$scale(10)
sd(funs$sample(1000))
```