R Programming Structures: Control Statements - Loops, Looping Over Non-vector Sets, If-Else, Arithmetic and Boolean Operators and values, Default Values for Argument, Return Values - Deciding Whether to explicitly call return, Returning Complex Objects, Functions are Objective, No Pointers in R, Recursion - A Quick sort Implementation, Extended Extended Example: A Binary Search Tree.

Control Statements: The statements in an R program are executed sequentially from the top of the program to the bottom. But some statements are to be executed repetitively, while only executing other statements if certain conditions are met. R has the standard control structures.

Loops: Looping constructs repetitively execute a statement or series of statements until a condition isn't true. These include the for, while and repeat structures with additional clauses break and next.

1) **FOR** :- The for loop executes a statement repetitively until a variable's value is no longer contained in the sequence seq.

- The syntax is ***for(var in sequence)***
  {
  statement
  }
  Here, sequence is a vector and var takes on each of its value during the loop. In each iteration, statement is evaluated.
- for (n in x) { - - - }
  It means that there will be one iteration of the loop for each component of the vector x, with taking on the values of those components—in the first iteration, n = x[1]; in the second iteration, n = x[2]; and so on.
- In this example *for (i in 1:10) print("Hello")* the word Hello is printed 10 times.
- Square of every element in a vector:

```
> x <- c(5,12,13)
> for(n in x) print(n^2)
[1] 25
[1] 144
[1] 169
```

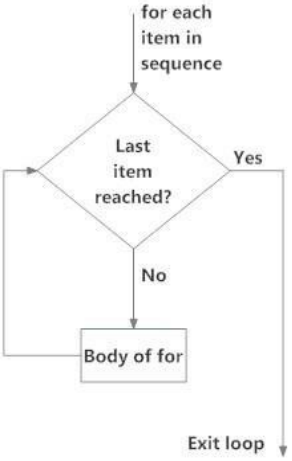


Fig: operation of for loop

- Program to find the multiplication

```
# take input from the user
    num = as.integer(readline(prompt = "Enter a number: "))
# use for loop to iterate 10 times
    for(i in 1:10) {
        print(paste(num,'x', i, '=', num*i)) }
```

2) **WHILE:-** A while loop executes a statement repetitively until the condition is no longer true.

Syntax:

```
while (expression)
{
    statement
}
```

- Here, expression is evaluated and the body of the loop is entered if the result is TRUE.

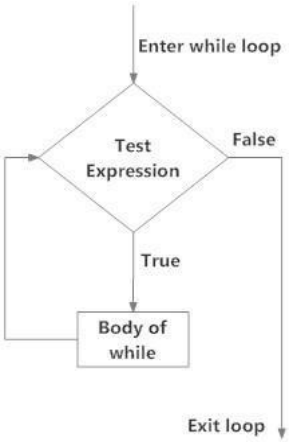


Fig: operation of while loop

- The statements inside the loop are executed and the flow returns to evaluate the expression again.
- This is repeated each time until expression evaluates to FALSE, in which case, the loop exits.
- Example

```
> i <- 1
> while(i<=10)        i <- i+4
> i
[1] 13
```

- Program to find the sum of first n natural numbers

```
sum = 0
# take input from the user
num = as.integer(readline(prompt = "Enter a number: "))
# use while loop to iterate until zero
  while(num > 0)
  {
    sum = sum + num
    num = num - 1
  }
  print(paste("The sum is", sum))
```

Output:

```
Enter a number: 4
[1] "The sum is 10"
```

3) Break statement: A break statement is used inside a loop (repeat, for, while) to stop the iterations and flow the control outside of the loop.In a nested looping situation, where there is a loop inside another loop, this statement exits from the innermost loop that is being evaluated.

Syntax:- **break**

Example

```
x <- 1:5
for (val in x) {
  if (val == 3){
    break
  }
  print(val)
}
```

Output:

```
[1] 1
[1] 2
```

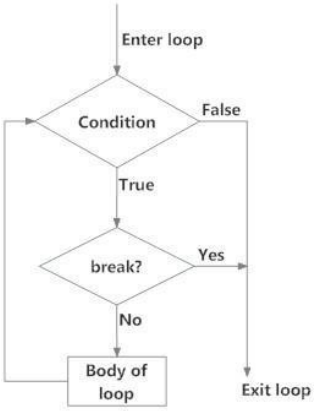


Fig: flowchart of break

4) Next statement:- A next statement is useful when we want to skip the current iteration of a loop without terminating it. On encountering next, the R parser skips further evaluation and starts next iteration of the loop.

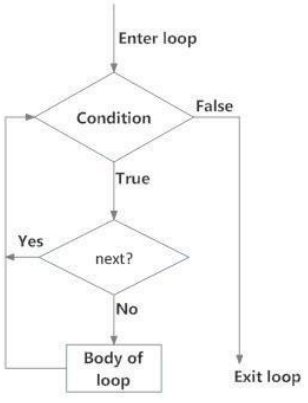
Syntax:- **next**

Example

```
x <- 1:5
for (val in x) {
  if (val == 3){
    next
  }
  print(val)
}
```

Output:

```
[1] 1
[1] 2
[1] 4
[1] 5
```

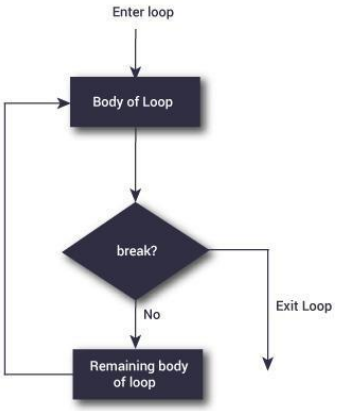


5) Repeat:- Repeat loop is used to iterate over a block of code multiple number of times. There is no condition check in repeat loop to exit the loop.

We must ourselves put a condition explicitly inside the body of the loop and use the break statement to exit the loop. Failing to do so will result into an infinite loop.

Syntax:

```
repeat
{
        statement
}
```



Example:

```
x <- 1
repeat
{
        print(x)
        x = x+1
        if (x == 6)
                break
}
```

Output:

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

Looping Over Non-vector Sets:- R does not directly support iteration over nonvector sets, but there are a couple of indirect yet easy ways to accomplish it:

- ***apply( ):-*** Applies on 2D arrays (matrices), data frames to find aggregate functions like sum, mean, median, standard deviation.
  syntax:- apply(matrix,margin,fun,...)
  margin = 1 indicates row
  = 2 indicates col

```
> x <- matrix(1:20,nrow =  4,ncol=5)
> x
     [,1] [,2] [,3] [,4]  [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
> apply(x,2,sum)
[1] 10 26 42 58 74
```

- Use ***lapply( )***, assuming that the iterations of the loop are independent of each other, thus allowing them to be performed in any order. Lapply( ) can be applies on dataframes,lists and vectors and return a list. lapply returns a list of the same length as X, each element of which is the result of applying FUN to the corresponding element of X.

  Syntax: lapply(X, FUN, ..)

```
> x <- matrix(1:4,2,2)
> x
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> lapply(x,sqrt)
[[1]]
[1] 1

[[2]]
[1] 1.414214

[[3]]
[1] 1.732051
```

lapply( ) function is applied on every elements of the object.

```
[[4]]
[1] 2
```

- Use ***get( ),*** As its name implies, this function takes as an argument a character string representing the name of some object and returns the object of that name. It sounds simple, but get() is a very powerful function.

Syntax: get("character string")

```
> get("sum")
  function (..., na.rm = FALSE) .Primitive("sum")
> get("g")
    function(x)
    {
      return(x+1)
    }
> get("num")
  [1] "45"
```

Note:- Reserved words in R programming are a set of words that have special meaning and cannot be used as an identifier (variable name, function name etc.).This list can be viewed by typing help(reserved) or ?reserved at the R command prompt.

Reserved words in R

if	*else*	*repeat*	*while*	*function*
for	*in*	*next*	*break*	*TRUE*
FALSE	*NULL*	*Inf*	*NaN*	*NA*
*NA_integer_*	*NA_real_*	*NA_complex_*	*NA_character_*	*...*

If -Else:- The if-else control structure executes a statement if a given condition is true. Optionally, a different statement is executed if the condition is false.

The syntax is

```
if(cond)
{
        statements
}
```

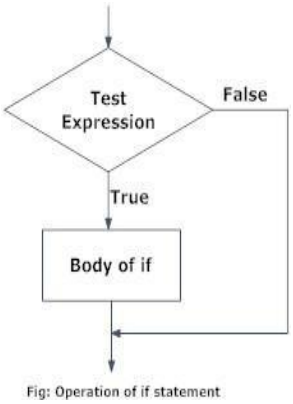


Fig: Operation of if statement

```
if (cond)
{
        statement1
} else
{
        statement2
}
```

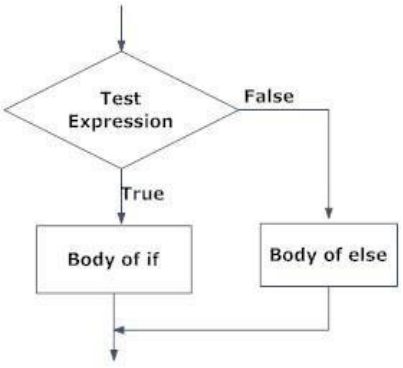


Fig: Operation of if...else statement

```
x <- 8
if(x>3)  {y <- 10
} else  {y<-0}
print(y)
```

Output:
[1] 10

```
y <- ifelse(x>3, 10, 0)
y
```

Output:
[1] 0

```
x <- 4
if(x==4)
  x <- 1
else
{
  x <- 3
  y <- 4
}
```

Output: Error.
The right brace before the else is used by the R parser to deduce that this is an if-else rather than just an if.

An if-else statement works as a function call, and as such, it returns the last value assigned.

v <- if (cond) expression1 else expression2

This will set v to the result of expression1 or expression2, depending on whether cond is true. You can use this fact to compact your code. Here's a simple example:

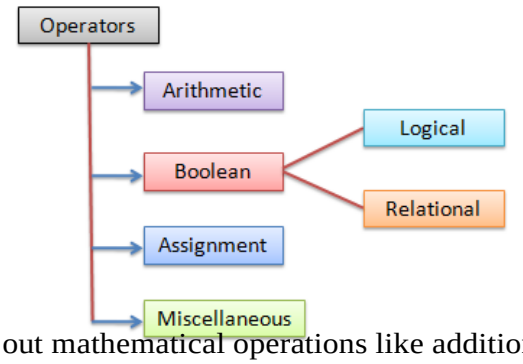
```
> x <- 2
> y <- if(x == 2)  x      else   x+1
> y
[1] 2

> x <- 2
>if(x == 2) y <- x   else y <- x+1
> y
[1] 2
```

Operators:- R has many operators to carry out different mathematical and logical operations.

Types of operators

1. Arithmetic operators.
2. Relational operators.
3. Logical operators.
4. Assignment operators.
5. Miscellaneous Operators



1.Arithmetic operators:- These operators are used to carry out mathematical operations like addition and multiplication. Here is a list of arithmetic operators available in R.

Operator	Description	Example
+	Adds two vectors	v <- c(2,5.5,6) t <- c(8, 3, 4) print(v+t) it produces the following result − [1] 10.0 8.5 10.0
–	Subtracts second vector from the first	v <- c(2,5.5,6) t <- c(8, 3, 4) print(v-t) it produces the following result −

		[1] -6.0 2.5 2.0
*	Multiplies both vectors	v <- c(2,5.5,6) t <- c(8, 3, 4) print(v*t) it produces the following result − [1] 16.0 16.5 24.0
/	Divide the first vector with the second	v <- c(2,5.5,6) t <- c(8, 3, 4) print(v/t) When we execute the above code, it produces the following result − [1] 0.250000 1.833333 1.500000
%%	Give the remainder of the first vector with the second	v <- c(2,5.5,6) t <- c(8, 3, 4) print(v%%t) it produces the following result − [1] 2.0 2.5 2.0
%/%	The result of division of first vector with second (quotient)	v <- c(2,5.5,6) t <- c(8, 3, 4) print(v%/%t) it produces the following result − [1] 0 1 1
^	The first vector raised to the exponent of second vector	v <- c(2,5.5,6) t <- c(8, 3, 4) print(v^t) it produces the following result − [1] 256.000 166.375 1296.000

2.Relational Operator:- Relational operators are used to compare between values.Each element of the first vector is compared with the corresponding element of the second vector. The result of comparison is a Boolean value.

Operator	Description	Example
>	Checks if each element of the first vector is greater than the corresponding element of the second vector.	v <- c(2,5.5,6,9) t <- c(8,2.5,14,9) print(v>t) it produces the following result − [1] FALSE TRUE FALSE FALSE
<	Checks if each element of the first vector is less than the corresponding element of the second vector.	v <- c(2,5.5,6,9) t <- c(8,2.5,14,9) print(v < t) it produces the following result − [1] TRUE FALSE TRUE FALSE

==	Checks if each element of the first vector is equal to the corresponding element of the second vector.	v <- c(2,5.5,6,9) t <- c(8,2.5,14,9) print(v == t) it produces the following result − [1] FALSE FALSE FALSE TRUE
<=	Checks if each element of the first vector is less than or equal to the corresponding element of the second vector.	v <- c(2,5.5,6,9) t <- c(8,2.5,14,9) print(v<=t) it produces the following result − [1] TRUE FALSE TRUE TRUE
>=	Checks if each element of the first vector is greater than or equal to the corresponding element of the second vector.	v <- c(2,5.5,6,9) t <- c(8,2.5,14,9) print(v>=t) it produces the following result − [1] FALSE TRUE FALSE TRUE
!=	Checks if each element of the first vector is unequal to the corresponding element of the second vector.	v <- c(2,5.5,6,9) t <- c(8,2.5,14,9) print(v!=t) it produces the following result − [1] TRUE TRUE TRUE FALSE

3)Logical Operators:- It is applicable only to vectors of type logical, numeric or complex. Zero is considered FALSE and non-zero numbers are taken as TRUE. Each element of the first vector is compared with the corresponding element of the second vector. The result of comparison is a Boolean value.

Operator	Description	Example
&	It is called Element-wise Logical AND operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if both the elements are TRUE.	v <- c(3,1,TRUE,2+3i) t <- c(4,1,FALSE,2+3i) print(v&t) it produces the following result − [1] TRUE TRUE FALSE TRUE
|	It is called Element-wise Logical OR operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if one the elements is TRUE.	v <- c(3,0,TRUE,2+2i) t <- c(4,0,FALSE,2+3i) print(v|t) it produces the following result − [1] TRUE FALSE TRUE TRUE
!	It is called Logical NOT operator. Takes each element of the vector and gives the opposite logical value.	v <- c(3,0,TRUE,2+2i) print(!v) it produces the following result − [1] FALSE TRUE FALSE FALSE

The logical operator && and || considers only the first element of the vectors and give a vector of single element as output.

Operator	Description	Example
&&	Called Logical AND operator. Takes first element of both the vectors and gives the TRUE only if both are TRUE.	v <- c(3,0,TRUE,2+2i) t <- c(1,3,TRUE,2+3i) print(v&&t) it produces the following result − [1] TRUE
||	Called Logical OR operator. Takes first element of both the vectors and gives the TRUE if one of them is TRUE.	v <- c(0,0,TRUE,2+2i) t <- c(0,3,TRUE,2+3i) print(v||t) it produces the following result −[1] FALSE

4) Assignment Operators:- These operators are used to assign values to vectors.

Operator	Description	Example
<– or = or <<–	Called Left Assignment	v1 <- c(3,1,TRUE,2+3i) v2 <<- c(3,1,TRUE,2+3i) v3 = c(3,1,TRUE,2+3i) print(v1) print(v2) print(v3) it produces the following result − [1] 3+0i 1+0i 1+0i 2+3i [1] 3+0i 1+0i 1+0i 2+3i [1] 3+0i 1+0i 1+0i 2+3i
–> or –>>	Called Right Assignment	c(3,1,TRUE,2+3i) -> v1 c(3,1,TRUE,2+3i) ->> v2 print(v1) print(v2) it produces the following result − [1] 3+0i 1+0i 1+0i 2+3i [1] 3+0i 1+0i 1+0i 2+3i

5) Miscellaneous Operators:- These operators are used to for specific purpose and not general mathematical or logical computation.

Operator	Description	Example
:	Colon operator. It creates the series of numbers in sequence for a vector.	v <- 2:8 print(v) it produces the following result − [1] 2 3 4 5 6 7 8
%in%	This operator is used to identify if	v1 <- 8 v2 <- 12 t <- 1:10 print(v1 %in% t)

	an element belongs to a vector.	print(v2 %in% t) it produces the following result – [1] TRUE [1] FALSE
%*%	This operator is used to multiply a matrix with its transpose.	M = matrix(c(2,6,5,1,10,4), nrow = 2,ncol = 3,byrow = TRUE) t = M %*% t(M) print(t) it produces the following result – [,1] [,2] [1,] 65 82 [2,] 82 117

Functions:- A function is a block or chunck of code having a specific structure, which is often singular or atomic nature, and can be reused to accomplish a specific nature. A function helps to divide a large program into modules to enhance readability and code reuse. or A function is a group of instructions that takes inputs, uses them to compute other values, and returns a result.

Built-in functions in R

structure of a function

```
function_name <- function(arguments)
{
        statements
}
```

- The word ‗function' is a keyword which is used to specify the statements enclosed within the curly braces are part of the function.
- Function_name is used to identify the function
- Function consists of formal arguments and body
- The function is called using the following statement:

function_name(arguments) Example:

```
say.hello <- function()
{
        print("Hello, World!")
}
say.hello()
  [1] "Hello, World!"
```

```
g <- function(x)
{
  x<- x+1
  return(x)
}
g(2)
  [1] 3
```

- ✓ Functions are assigned to objects just like any other variable, using <-operator.
- ✓ Function are a set of parentheses that can either be empty – not have any arguments – or contain any number of arguments.
- ✓ The body of the function is enclosed in curly braces ({ and }).This is not necessary if the function contains only one line.
- ✓ A semicolon(;) can be used to indicate the end of the line but is not necessary.

```
# counts the number of odd integers in x
        > oddcount <- function(x)
        {
          k <- 0                          # assign 0 to k
          for (n in x) {
          if (n %% 2 == 1) k <- k+1       # %% is the modulo operator
          }
```

```
  return(k)
}
> oddcount(c(1,3,5))
    [1] 3
> oddcount(c(1,2,3,7,9))
    [1] 4
```

Variables created outside functions are global and are available within functions as well. Example:

```
> f <- function(x) return(x+y)
> y <- 3
> f(5)
[1] 8
```

Here y is a global variable. A global variable can be written to from within a function by using R's superassignment operator, <<-.

Default Arguments:- R also makes frequent use of *default arguments*. Consider a function definition like this:

```
> g <- function(x,y=2,z=T) { ... }
```

Here y will be initialized to 2 if the programmer does not specify y in the call. Similarly, z will have the default value TRUE. Now consider this call:

```
> g(12,z=FALSE)
```

Here, the value 12 is the actual argument for x, and we accept the default value of 2 for y, but we override the default for z, setting its value to FALSE.

Default Values for Arguments:-

```
> my_matrix<- matrix (1:12,4,3,byrow= TRUE)
```

The argument *byrow= TRUE* tells R that the matrix should be filled in row wise. In this example the default argument is byrow = FALSE, the matrix is filled in column wise.

Lazy Evaluation of Function:- Arguments to functions are evaluated lazily, which means so they are evaluated only when needed by the function body.

```
# Create a function with arguments.
new.function <- function(a, b) {
  print(a^2)
  print(a)
  print(b)
}
# Evaluate the function without supplying one of the arguments.
new.function(6)
```

When we execute the above code, it produces the following result

– [1] 36
[1] 6
Error in print(b) : argument "b" is missing, with no default

Return Values:- Functions are generally used for computing some value, so they need a mechanism to supply that value back to the caller.This is called returning.

- The return value of a function can be any R object. Although the return value is often a list, it could even be another function.
- You can transmit a value back to the caller by explicitly calling return(). Without this call, the value of the last executed statement will be returned by default.
- If the last statement in the call function is a for( ) statement, which returns the value NULL.

```
 # First build it without an explicit return
num<-function(x)
{
   x*2
}
> num(5)
[1] 10
```

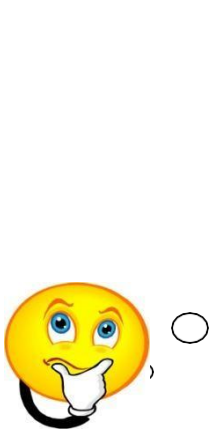
```
# Now build it with an explicit return
num <- function(x)
{
    return(x*2)
}
> num(5)
[1] 10
```

```
# build it again, this time with another argument after the explicit return
num <- function(x)
{
    return(x*2)
        #below here is not executed because the return function already exists.
    print("VISHNU")
    return(17)
}
> num(5)
[1] 10
```

```
# if the last statement is for loop or any empty statement then it return NULL.
num <- function(x)
{  }
> num(5)
[1] NULL
```

Deciding Whether to Explicitly Call return():-The R idiom is to avoid explicit calls to return(). One of the reasons cited for this approach is that calling that function lengthens execution time. However, unless the function is very short, the time saved is negligible, so this might not be the most compelling reason to refrain from using return(). But it usually isn't needed.

```
#Example to count the odd numbers with no return statement
oddcount <- function(x) {
 k <- 0
 for (n in x) {
  if (n %% 2 == 1) k <- k+1
 }
 k
}
> oddcount(c(12,2,5,9,7))
[1] 3
```




```
#Example to count the odd numbers with return statement
oddcount<-function(x){
 k <- 0
 for (n in x) {
  if (n %% 2 == 1) k <- k+1
 }
return(k)
}
> oddcount(c(12,2,5,9,7))
[1] 3
```

Good software design, can glance through a function's code and immediately spot the various points at which control is returned to the caller. The easiest way to accomplish this is to use an explicit return() call in all lines in the middle of the code that cause a return.

Returning Complex Objects:- The return value can be any R object, you can return complex objects. Here is an example of a function being returned:

```
g<-function() {
        x<- 3
        t <- function(x) return(x^2)
        return(t)
    }

    > g()
function(x) return(x^2)
<environment: 0x16779d58>
```

If your function has multiple return values, place them in a list or other container.

Functions are Objective:- R functions are first-class objects (of the class "function"), meaning that they can be used for the most part just like other objects. This is seen in the syntax of function creation:

```
g <- function(x)
{
    return(x+1)
}
```

function( ) is a built-in R function whose job is to create functions.On the right-hand side, there are really two arguments to function(): The first is the formal argument list for the function i.e, x and the second is the body of that function return(x+1). That second argument must be of class "expression". So, the point is that the right-hand side creates a function object, which is then assigned to g.

> ?"{" Its job is the make a single unit of what could be several statements.

- ✓ formals( ) :- Get or set the formal arguments of a function

```
> formals(g)          # g is a function with formal arguments –x‖
  $x
```

- ✓ body( ) :- Get or set the body of a function

```
> body(g)           # g is a function
{
   x <- x + 1
   return(x)
}
```

- ✓ Replacing body of the function: quote( ) is used to substitute expression

```
> g <- function(x) return(x+1)
> body(g) <- quote(2*x+3)
> g
function (x)
2 * x + 3
```

- ✓ Typing the name of an object results in printing that object to the screen which is similar to all objects

```
> g
function(x)
{
   return(x+1)
}
```

- ✓ Printing out a function is also useful if you are not quite sure what an R library function does. Code of a function is displayed by typing the built-in function name.

```
> sd
function (x, na.rm = FALSE)
sqrt(var(if (is.vector(x) || is.factor(x)) x else as.double(x),
   na.rm = na.rm))
<bytecode: 0x17b49740> <environment: namespace:stats>
```

- ✓ Some of R's most fundamental built-in functions are written directly in C, and thus they are not viewable in this manner.

```
> sum
function (..., na.rm = FALSE) .Primitive("sum")
```

- ✓ Since functions are objects, you can also assign them, use them as arguments to other functions, and so on.

```
> f1 <- function(a,b) return(a+b)
> f2 <- function(a,b) return(a-b)
> f <- f1                    # Assigning function object to other object
> f(3,2)
[1] 5
> g <-function(h,a,b)h(a,b)  # passing function object as an arguments
> g(f1,3,2)
[1] 5
> g(f2,3,2)
[1] 1
```

No Pointers in R:- R does not have variables corresponding to pointers or references like C language. This can make programming more difficult in some cases.

The fundamental thought is to create a class constructor and have every instantiation of the class be its own environment. One can then pass the object/condition into a function and it will be passed by reference instead of by value, because unlike other R objects, environments are not copied when passed to functions. Changes to the object in the function will change the object in the calling frame. In this way, one can operate on the object and change internal elements without having to create a copy of the object when the function is called, nor pass the entire object back from the function. For large objects, this saves memory and time.

For example, you cannot write a function that directly changes its arguments.

```
> x <- c(12,45,6)
> sort(x)
[1] 6 12 45
> x
[1] 12 45 6
```

The argument to sort() does not change. If we do want x to change in this R code, the solution is to reassign the arguments:

```
> x <- sort(x)
> x
[1] 6 12 45
```

If a function has several output then a solution is to gather them together into a list, call the function with this list as an argument, have the function return the list, and then reassign to the original list. An example is the following function, which determines the indices of odd and even numbers in a vector of integers:

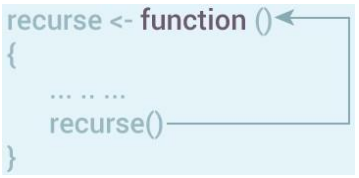
```
> y <-function(v){
    odds <- which(v %% 2 == 1)
    evens <- which(v %% 2 == 0)
    list(o=odds,e=evens)
   }
> y(c(2,34,1,5))
$o
[1] 3 4

$e
[1] 1 2
```

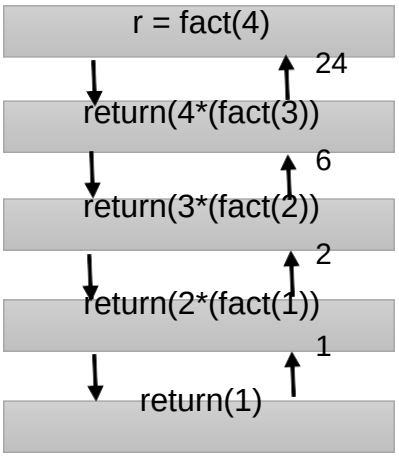
Recursion:- Recursion is a programming technique in which, a function calls itself repeatedly for some input.

To solve a problem of type X by writing a recursive function f():

1. Break the original problem of type X into one or more smaller problems of type X.
2. Within f(), call f() on each of the smaller problems.
3. Within f(), piece together the results of (b) to solve the original problem.




```
# Recursive function to find factorial
recursive.factorial <- function(x)
{
 if (x == 0)   return (1)
 else        return (x * recursive.factorial(x-1))
}
> recursive.factorial(5)
[1] 120
```



A Quicksort Implementation:- Quick sort is also known as Partition-Exchange sort and is based on Divide and conquer Algorithm design method. This was proposed by C.A.R Hoare. The basic idea of quick sort is very simple. We consider one element at a time (pivot element). We have to move the pivot element to the final position that it should occupy in the final sorted list. While identifying this position, we arrange the elements, such that the elements to the left of the pivot element will be less than pivot element & elements to the right of the pivot element will be greater than pivot element. There by dividing the list by 2 parts. We have to apply quick sort on these 2 parts recursively until the entire list is sorted.

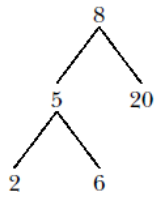
For instance, suppose we wish to sort the vector (5,4,12,13,3,8,88). We first compare everything to the first element, 5, to form two subvectors: one consisting of the elements less than 5 and the other consisting of the elements greater than or equal to 5. That gives us subvectors (4,3) and (12,13,8,88). We then call the function on the subvectors, returning (3,4) and (8,12,13,88). We string those together with the 5, yielding (3,4,5,8,12,13,88), as desired. R's vector-filtering capability and its c() function make implementation of Quicksort quite easy.

```
# Quicksort recursive function
qs <- function(x) {
                    if (length(x) <= 1) return(x)
                    pivot <- x[1]
                    therest <- x[-1]
                    sv1 <- therest[therest < pivot]
                    sv2 <- therest[therest >= pivot]
                    sv1 <- qs(sv1)
                    sv2 <- qs(sv2)
                    return(c(sv1,pivot,sv2)) }
```

```
> qs(c(12,6,7,34,3))
[1] 3 6 7 12 34
```

Binary search tree:- The nature of binary search trees implies that at any node, all of the elements in the node's left subtree are less than or equal to the value stored in this node, while the right subtree stores the elements that are larger than the value in this mode. In our example tree, where the root node contains 8, all of the values in the left subtree-5, 2 and 6-are less than 8, while 20 is greater than 8.

Here is an example:



The code follows. Note that it includes only routines to insert new items and to traverse the tree. # storage is in a matrix, say m, one row per node of the tree; a link i in the tree means the vector
#m[i,] = (u,v,w); u and v are the left and right links, and w is the stored value; null links have the value #NA; the matrix is referred to as the list (m,nxt,inc), where m is the matrix, nxt is the next empty row to #be used, and inc is the number of rows of expansion to be allocated when the matrix becomes full

```
# initializes a storage matrix, with initial stored value firstval
newtree <- function(firstval,inc) {
 m <- matrix(rep(NA,inc*3),nrow=inc,ncol=3)
 m[1,3] <- firstval
 return(list(mat=m,nxt=2,inc=inc))
}

# inserts newval into nonempty tree whose head is index hdidx in the storage space treeloc; note that
#return value must be reassigned to tree; inc is as in newtree() above
ins <- function(hdidx,tree,newval,inc) {
 tr <- tree
 # check for room to add a new element
 tr$nxt <- tr$nxt + 1
 if (tr$nxt > nrow(tr$mat))
  tr$mat <- rbind(tr$mat,matrix(rep(NA,inc*3),nrow=inc,ncol=3))
 newidx <- tr$nxt # where we'll put the new tree node
 tr$mat[newidx,3] <- newval
 idx <- hdidx # marks our current place in the tree
 node <- tr$mat[idx,]
 nodeval <- node[3]
 while (TRUE) {
  # which direction to descend, left or right?
  if (newval <= nodeval) dir <- 1 else dir <-
  2 # descend
  # null link?
  if (is.na(node[dir])) {
   tr$mat[idx,dir] <- newidx
   break
  } else {
   idx <- node[dir]
   node <- tr$mat[idx,]
   nodeval <- node[3]
  }
 }
 return(tr)
}
```

```
# print sorted tree via inorder traversal
printtree <- function(hdidx,tree) {
  left  <- tree$mat[hdidx,1]
  if (!is.na(left)) printtree(left,tree)
  print(tree$mat[hdidx,3])
  right <- tree$mat[hdidx,2]
  if (!is.na(right)) printtree(right,tree)
}
```

sapply( ):- sapply is wrapper class to lapply with difference being it returns vector or matrix instead of list object.

Syntax: `sapply(X, FUN, ...,)`

```
    # create a list with 2 elements
     x = (a=1:10,b=11:20) # mean of values using sapply
sapply(x,
  mean) a     b
 5.5 15.5
```

tapply( ):- tapply() applies a function or operation on subset of the vector broken down by a given factor variable.

To understand this, imagine we have ages of 20 people (male/females), and we need to know the average age of males and females from this sample. To start with we can group ages by the gender (male or female), ages of 12 males, and ages of 8 females, and later calculate the average age for males and females.

Syntax of tapply: ***tapply(X, INDEX, FUN, ...)***

X = a vector, INDEX = list of one or more factor, FUN = Function or operation that needs to be applied, … optional arguments for the function

```
> ages <- c(25,26,55,37,21,42)
> affils <- c("R","D","D","R","U","D")
> tapply(ages,affils,mean)
        D R U
        41 31 21
```