บทที่ 4
โครงสร้างการทำงานซ้ำ

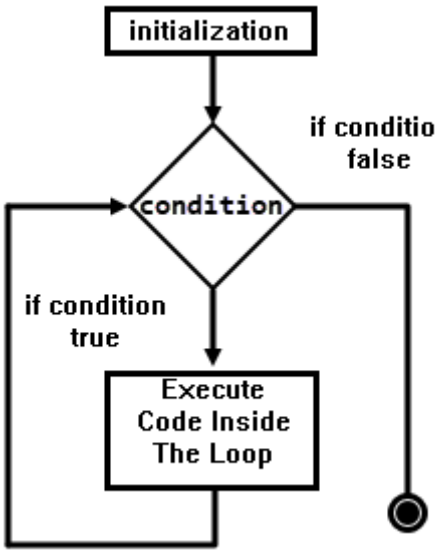
การวนซ้ำ เป็นรูปแบบของการเขียนโปรแกรมสั่งให้คอมพิวเตอร์ทำงานซ้ำตั้งแต่ 1 ครั้งขึ้นไป ทั้งนี้ขึ้นอยู่กับเงื่อนไขในการทำงาน ซึ่งจะช่วยให้การเขียนโปรแกรมได้ง่ายสะดวก ไม่ต้องเขียนข้อความคำสั่งเดิมหลายครั้ง ทำให้โปรแกรมมีความกระชับ สามารถตรวจสอบความผิดพลาดได้ง่าย โครงสร้างการทำงานซ้ำ (repetition control structure) ในภาษา R ประกอบด้วย

1) การทำงานซ้ำ for
2) การทำงานซ้ำ while
3) การทำงานซ้ำ repeat
4) คำสั่งควบคุมลูป (loop)
5) ฟังก์ชันลูป (loop function)

โดยแต่ละโครงสร้างคำสั่ง มีรูปแบบและวิธีการใช้งานที่แตกต่างกัน ซึ่งผู้เขียนโปรแกรมสามารถเลือกใช้ได้ตามความเหมาะสมกับลักษณะการใช้งานในโปรแกรม

4.1 การทำงานซ้ำ for

คำสั่ง for เป็นคำสั่งที่สั่งให้คอมพิวเตอร์ประมวลผลคำสั่ง หรือชุดคำสั่งวนซ้ำได้หลายรอบ โดยต้องกำหนดจำนวนรอบให้การวนซ้ำที่แน่นอน



ขั้นตอนการทำงานของลูป for

1. กำหนดค่าเริ่มต้น โดยจะกำหนดค่าเริ่มต้นให้กับตัวแปรสำหรับนับรอบ

2. ตรวจสอบเงื่อนไข คอมไพเลอร์จะตรวจสอบรายการในเวกเตอร์และถ้ามีรายการข้อมูล (True) จะดำเนินการประมวลผลคำสั่งภายในลูป แต่ถ้าหากไม่มีรายการข้อมูล (False) จะออกจากลูป

3. หลังจากประมวลผลเสร็จ คอมไพเลอร์จะดำเนินการข้อมูลรายการถัดไปในเวกเตอร์

4. จากนั้นจะทำการตรวจสอบเงื่อนไขในข้อ 2 อีกรอบ

โดยสามารถเขียนคำสั่งตามรูปแบบดังต่อไปนี้

รูปแบบ :

```
for (value in vector) {
        statements
}
```

ตัวอย่าง 1: การแสดงผลข้อมูลในเวกเตอร์ v

```
v<- c(1,2,3,4,5)
for( i in v){
        print(i)
}
```

ตัวอย่าง 2: การวนลูปแสดงชื่อเดือนใน 1 ปี

```
for(i in month.name){
        print(i)
}
```

ตัวอย่าง 3: การวนลูปแสดงตัวอักษรพิมพ์ใหญ่ในภาษาอังกฤษ

```
for(i in LETTERS){
        print(i)
```

```
}
```

ตัวอย่าง 4: การหาผลรวมของข้อมูลในเวกเตอร์ v

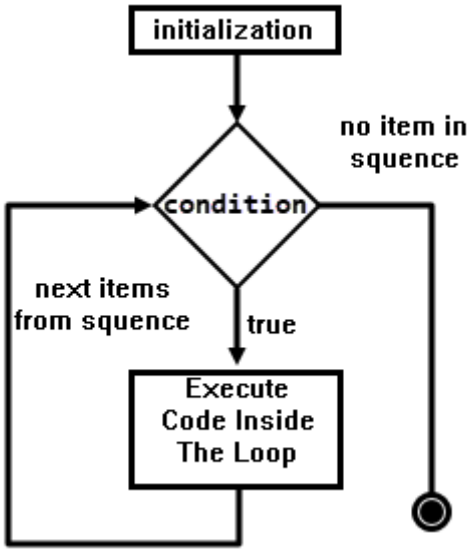
```
N=c(2,3,4,7,12,30)
total=0
for(i in N){
        total=total+i
}
print(paste("total=",total))
```

ตัวอย่าง 5: การแสดงผลการตัดเกรดจากข้อมูลในเวกเตอร์ v

```
N=c(60,65,70,80)
for(i in N){
        if(i>=80){
                G<-"A"
        }else if(i>=70){
                G<-"B"
        }else if(i>=60){
                G<-"C"
        }else if(i>=50){
                G<-"D"
        }else {
                 G<-"E"
         }
        print(G)
}#for
```

4.2 การทำงานซ้ำ while

การทำงานของ while จะคล้ายกับการทำงานของ for แต่จะแตกต่างกันตรงที่ for จะมีการเพิ่มค่าจำนวนรอบโดยอัตโนมัติ ซึ่งลำดับขั้นตอนในการทำงานของลูป while สามารถอธิบายได้ดังภาพที่แสดงด้านล้างต่อไปนี้



ขั้นตอนการทำงานของ while มีรายละเอียดดังนี้

1. กำหนดค่าเริ่มต้นให้กับตัวแปรสำหรับใช้ในการกำหนดจำนวนรอบการทำงาน

2. ทำการตรวจสอบเงื่อนไข ถ้าเงื่อนไขเป็นจริงก็จะทำงานคำสั่งภายใต้ลูป

3. เมื่อประมวลผลคำสั่งเสร็จ จะทำการเพิ่มค่าตัวแปรนับรอบในการทำงานซึ่งผู้ใช้จะเป็นคนกำหนดค่าเอง

4. จากนั้นจะกลับมาตรวจสอบเงื่อนไขอีกครั้ง แล้วทำคำสั่งซ้ำอีกรอบ และจะทำแบบนี้ไปเรื่อย ๆ จนกว่าเงื่อนไขจะเป็นเท็จจึงจะออกจากลูป

โดยรูปแบบคำสั่งสามารถเขียนได้ดังนี้

รูปแบบ :

```
while (test_expression) {
        statement
}
```

ตัวอย่าง 1: การแสดงข้อมูลตามจำนวนรอบที่กำหนดในเวกเตอร์ v

```
i<- 1
while (i <6) {
print(1:i)
        i = i + 1
}
```

ตัวอย่าง 2: การแสดงข้อมูลตามจำนวนรอบที่กำหนดในเวกเตอร์ v

```
v <- c("Hello","while loop")
i <- 1
while (i <= 7) {
        print(paste(i,v))
        i = i + 1
}
```

ตัวอย่าง 3: การแสดงข้อมูลในเวกเตอร์ X

```
X <- c(2,3,4,5,6)
i<- 1
while (i <6) {
        print(X[i])
         i = i + 1
}
```

ตัวอย่าง 4: การแสดงข้อมูลในเวกเตอร์ H

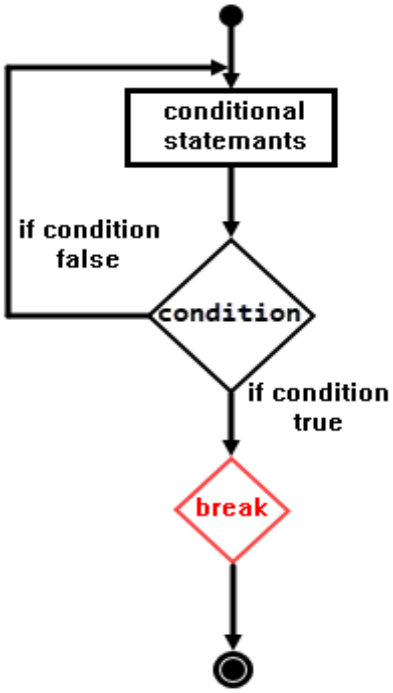
```
H <- c('h','e','l','l','o')
i<- 1
while (i <6) {
          print(H[1:i])
         i = i + 1
}
```

ตัวอย่าง 5: การหาผลรวมของข้อมูลในเวกเตอร์ v

```
H <- c(2,3,4,5,10)
i<- 1
sum<-0
while (i <6) {
          sum=sum+H[i]
          i = i + 1
}
print(sum)
```

4.3 การทำงานซ้ำ Repeat

การทำงานของ Repeat จะมีลักษณะการทำงานซ้ำคล้ายกับการทำงาน while แต่จะแตกต่างกันตรงที่ การทำงานของ Repeat เป็นการลักษณะทำจนกระทั่ง คือ เงื่อนไขในการหลุดออกจากลูปของ Repeat ตอนสุดท้าย เป็นเงื่อนไขเป็นจริง ซึ่งลำดับขั้นตอนในการทำงานของลูป Repeat สามารถอธิบายได้ดังภาพที่แสดงด้านล้างต่อไปนี้



ขั้นตอนการทำงานของ repeat

1. อันดับแรกกำหนดค่าเริ่มต้นให้กับตัวแปรสำหรับใช้ในการนับจำนวนของรอบ จากนั้นเข้าสู่การทำงานของลูป repeat
2. ขั้นตอนถัดมาเป็นการประมวลผลคำสั่งในลูป
3. เพิ่มค่าตัวแปรที่ใช้ในการนับจำนวนรอบของลูป
4. ทำการตรวจสอบเงื่อนไข ถ้าหากเงื่อนไขเป็นจริง จะประมวลผลคำสั่ง Break เพื่อออกจากลูป แต่ถ้าเงื่อนไขเป็นเท็จก็จะวนไปทำงานในขั้นตอนที่ 2 อีกครั้ง

โดยรูปแบบการเขียนคำสั่งดังนี้

รูปแบบ :

```
repeat {
        commands
        if(condition) {
                break
        }
}
```

ตัวอย่าง 1: การพิมพ์ตัวเลขตั้งแต่ 1 ถึง 10

```
a = 1
repeat {
        print(a)
        a = a+1
        if(a>=11){
                break
        }
}
```

ตัวอย่าง 2: การหาผลรวมของข้อมูลในเวกเตอร์ v

```
H <- c(2,3,4,5,10)
i<- 1
sum<-0
repeat{
        sum=sum+H[i]
        i = i + 1
        if(i>5){
                break
        }
}
print(sum)
```

ตัวอย่าง 3: การแสดงผลข้อความในเวกเตอร์ v

```
v <- c("Test","Repeat loop","R Programming")
C <- 1
repeat {
        print(v[C])
        C <- C+1
        if(C > 5) {
                break
        }
}
```

ตัวอย่าง 4: การตัดเกรดของนักศึกษาจำนวน 5 คน

```
score <- c(65,80,75,85,50)
i<-1
repeat{
        if(score[i]>=80){ G<-"A"
        }else if(score[i]>=70){ G<-"B"
        }else if(score[i]>=60){ G<-"C"
        }else if(score[i]>=50){ G<-"D"
        }else { G<-"E" }
        print(G)
        i=i+1
        if(i>5){
                break
        }
 }
```

ตัวอย่าง 5: โปรแกรมสุ่มทายตัวเลข

```
readinteger <- function(){
        n <- readline(prompt="Please, enter your ANSWER: ")
}
repeat {
        exit_me <- as.integer(readinteger());
        if (exit_me == 35) {
                print("Well done!");
                break
        }else print("Sorry, the answer to whatever the question MUST be 35");
}
```

4.4 คำสั่งควบคุมลูป (loop control statement)

ใช้สำหรับควบคุมการทำงานของลูปเพื่อให้เป็นไปตามความต้องการของการเปลี่ยนโปรแกรม โดยเมื่อออกจากขอบเขตการทำงาน ออปเจ็คแบบอัตโนมัติที่ถูกสร้างขึ้นทั้งหมดจะถูกยกเลิกไป โดยภาษา R สนับสนุนการทำงานของคำสั่งควบคุมต่อไปนี้

คำสั่ง	คำอธิบาย
คำสั่ง break	ใช้สำหรับจบการทำงานของลูปและย้ายไปทำงานยังคำสั่งถัดไปทันที
คำสั่ง next	ใช้สำหรับย้ายไปทำงานคำสั่งถัดไป คล้ายการทำงานของคำสั่ง switch ในภาษา R

4.4.1 คำสั่ง break

คำสั่ง break ในโปรแกรมภาษา R ถูกใช้เพื่อวัตถุประสงค์ 2 อย่างดังนี้

1) ใช้สำหรับจบการทำงานภายในลูป จากนั้นต้องการย้ายไปทำงานยังคำสั่งถัดไปทันที
2) ใช้สำหรับยุติการทำงานภายใต้ case ในคำสั่ง switch ได้

รูปแบบ :

```
break
```

ตัวอย่าง : การวนลูปแสดงคำว่าว Hello loop จำนวน 5 ครั้ง

```
v <- c("Hello","loop")
cnt <- 2
repeat{
   print(v)
   cnt <- cnt+1
   if(cnt > 5){
      break
   }
}
```

4.4.2 คำสั่ง next

คำสั่ง next ในการเขียนโปรแกรมภาษา R จะเป็นประโยชน์มาก เมื่อเราต้องการข้ามลูป (loop) ปัจจุบันโดยไม่ต้องยกเลิกการทำงานของลูป เมื่อพบคำสั่ง next โปรแกรมจะข้ามการทำงานและเริ่มทำงานลูปรอบถัดไปทันที

รูปแบบ :

```
next
```

ตัวอย่าง : การวนลูปแสดงตัวอักษรภาษาอังกฤษ A-F

```
v <- LETTERS[1:6]
for ( i in v){
   if (i == "D"){
      next
   }
   print(i)
}
```

4.5 ฟังก์ชันลูป (loop function)

4.5.1 ฟังก์ชัน lapply() เป็นฟังก์ชันที่ใช้ในการวนลูป (loop) การทำงานในรายการข้อมูล ซึ่งมีขั้นตอนการทำงานดังนี้:

1) วนลูปทำงานผ่านลิสต์ (list) การวนลูปจะวนจนครบทุกรายการข้อมูลในลิสต์

2) ใช้ฟังก์ชันกับแต่ละรายการของลิสต์ (list) (ฟังก์ชันที่ระบุ)

3) และส่งค่ากลับเป็นลิสต์ (list)

ฟังก์ชันนี้ใช้เวลาสามอาร์กิวเมนต์: (1) ตัวแปรลิสต์ (list) ; (2) ฟังก์ชัน (หรือชื่อของฟังก์ชัน) FUN; (3) อาร์กิวเมนต์อื่น ๆ ผ่านอาร์กิวเมนต์ กรณีที่ตัวแปรในข้อ (1) ไม่ใช่ลิสต์จะถูกบังคับให้เป็นลิสต์ (list) ด้วยฟังก์ชัน as.list () ตัวอย่างของฟังก์ชัน lapply () สามารถแสดงตัวอย่างการทำงานได้ดังนี้

ตัวอย่าง 1 : การหาค่ายกกำลังสองของตัวเลขที่กำหนด

```
lapply(1:3, function(x) x^2)
```

ตัวอย่าง 2 : การหาค่ายกกำลังสองของตัวเลขที่กำหนดและส่งค่าออกเป็นข้อมูลเวกเตอร์

```
unlist(lapply(1:3, function(x) x^2))
```

ตัวอย่าง 3 : การหาค่าเฉลี่ยในตัวแปรลิสต์ x ด้วยฟังก์ชัน lapply

```
x <- list(a = 1:5, b = rnorm(10))
 lapply(x, mean)
```

4.5.2 ฟังก์ชัน sapply() เป็นฟังก์ชั่นมีคุณสมบัติการทำงานคล้ายกับ lapply () แต่ความแตกต่างกันเพียงอย่างเดียวคือการส่งค่ากลับคืนในฟังก์ชัน sapply() จะพยายามลดความซับซ้อนของฟังก์ชัน lapply() ถ้าเป็นไปได้ เป็นหลักการทำงานของฟังก์ชัน sapply() มีขั้นตอนการทำงานดังต่อไปนี้:

1) ถ้าผลลัพธ์ (output) เป็นลิสต์ (list) ที่มีความยาวค่าเป็น 1 จะส่งค่ากลับเป็นเวกเตอร์

2) ถ้าผลลัพธ์เป็นลิสต์ ที่ทุกรายการเป็นเวกเตอร์ที่มีความยาวเท่ากัน (> 1) จะส่งค่ากลับเป็นเมทริกซ์

3) ถ้าไม่สามารถประมวลผลข้อมูลได้ จะมีการส่งค่ากลับเป็นลิสต์

ดังแสดงในตัวอย่างการเขียนโปรแกรมด้านล่างต่อไปนี้

ตัวอย่าง 1: การใช้ฟังก์ชัน sapply วนลูปพิมพ์แสดงข้อความ Hello จำนวน 5 ครั้ง

```
x = sapply(1:5, function(x) print("Hello"))
```

ตัวอย่าง 2:การใช้ฟังก์ชัน sapply วนลูปพิมพ์แสดงข้อความ Hello จำนวน 5 ครั้ง แบบที่ 2

```
f = function(x) print("Hello")
x = sapply(1:5, f)
```

ตัวอย่าง 3: การใช้ฟังก์ชัน sapply วนลูปแสดงค่ายกกำลังสอง

```
sapply(1:3, function(x) x^2)
```

ตัวอย่าง 4: การหาค่ากลางของแต่ละควอไทน์ของแต่ละอิลิเมนต์

```
x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FALSE,TRUE))
sapply(x, quantile)
```

4.5.3 ฟังก์ชัน apply เป็นฟังก์ชันที่ใช้ในการวนลูปทำงานในอาร์เรย์ ส่วนใหญ่มักประยุกต์ใช้จัดการกับข้อมูลแถวและคอลัมน์ของข้อมูลอาร์เรย์ 2 มิติเท่านั้น แต่อย่างไรก็ตามยังสามารถประยุกต์ใช้กับข้อมูลอาร์เรย์ทั่วไปได้ เช่น ใช้ในการหาค่าเฉลี่ยของอาร์เรย์ของเมทริกซ์ได้ การทำงานของฟังก์ชัน apply ไม่ได้เร็วกว่าการใช้งาน loop แต่มันสามารถที่จะเขียนคำสั่งภายในบรรทัดเดียว ซึ่งกระทัดรัดมาก สามารถแสดงตัวอย่างการทำงานดังต่อไปนี้

รูปแบบ :

apply (X, MARGIN, FUN, ...)

อาร์กิวเมนต์ของฟังก์ชัน apply ประกอบด้วย

1) X หมายถึง ข้อมูลอาร์เรย์หรือเมทริกซ์

2) MARGIN หมายถึง เวกเตอร์จำนวนที่จะบุขอบเขตอย่างชัดเจน

3) FUN หมายถึง ฟังก์ชันที่จะประยุกต์ใช้งาน

4) ... หมายถึง อาร์กิวเมนต์ที่จะถูกส่งผ่านไปยังฟังก์ชัน

ข้อมูลตั้งต้น

mymat<-matrix(rep(seq(5), 4), ncol = 5)

```
> mymat
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    4    3    2
[2,]    2    1    5    4    3
[3,]    3    2    1    5    4
[4,]    4    3    2    1    5
>
```

ตัวอย่าง 2: การหาผลรวมในแนวแถวทุกแถว

apply(mymat, 1, sum)

ตัวอย่าง 3: หาค่าเฉลี่ยในแนวแถวทุกแถว

rowSums = apply(x, 1, mean)

ตัวอย่าง 4: การหาผลรวมในแนวคอลัมน์ทุกแถว

```
apply(mymat, 2, sum)
```

ตัวอย่าง 5: หาค่าเฉลี่ยในแนวคอลัมน์ทุกแถว

```
rowSums = apply(x, 2, mean)
```

ตัวอย่าง 6: หาผลรวมในแนวแถวทุกแถวจากนั้นเพิ่มค่า y=4.5 เข้าไป

```
apply(mymat, 1, function(x, y) sum(x) + y, y=4.5)
```

ตัวอย่าง 7: หาผลรวมในแนวคอลัมน์แต่ละคอลัมน์ด้วยฟังก์ชัน summary

```
apply(mymat, 2, function(x, y) summary(mymat))
```

ตัวอย่าง 8: การคำนวณค่าควอไทน์ในแถวของเมทริกซ์ด้วยฟังก์ชันควอไทน์

```
apply(x, 1, quantile, probs = c(0.25, 0.75))
```

4.5.4 ฟังก์ชัน tapply() เป็นฟังก์ชันที่ใช้ทำงานกับข้อมูลรายการของเวกเตอร์ โดยจะส่งค่ากลับเป็นเป็นเวกเตอร์ หรือแฟคเตอร์ได้

รูปแบบ :

```
tapply(X, INDEX, FUN, ...)
```

อาร์กิวเมนต์สำหรับการทำงานของ tapply ประกอบด้วย

1) X หมายถึง ข้อมูลอาร์เรย์หรือเมทริกซ์ที่ใช้ในการคำนวณ

2) INDEX หมายถึง รายการของแฟคเตอร์

3) FUN หมายถึง ฟังก์ชันที่จะเรียกใช้งาน

4) ... หมายถึง อาร์กิวเมนต์ที่จะถูกส่งผ่านไปยังฟังก์ชัน (ถ้ามี)

ตัวอย่าง 1: การคำนวณค่าควอไทน์ในแถวของเมทริกซ์ด้วยฟังก์ชันควอไทน์

```
x <- 1:20
y <- factor(rep(letters[1:5], each = 4))
tapply(x, y, sum)
```

ตัวอย่าง 2: การคำนวณค่าในข้อมูล iris

```
data(iris)  # Load the dataset iris
str(iris)  # Structure of the dataset

mean(iris$Sepal.Length)
tapply(iris$Sepal.Length, iris$Species, mean)
```

ตัวอย่าง 3: การคำนวณค่าในข้อมูล mtcars

```
data(mtcars)
str(mtcars)

tapply(mtcars$mpg, list(mtcars$cyl, mtcars$am), mean)
```

4.5.5 ฟังก์ชัน mapply() เป็นฟังก์ชัน lapply เวอร์ชันหลายตัวแบบใช้สำหรับ เป็นการประยุกต์ใช้การเรียงลำดับหลายตัวแปร การเรียกใช้งาน lapply เป็นการเรียกใช้เพียงรอบเดียวเท่านั้น แต่ฟังก์ชัน mapply จะทำงานวนรอบหลายออบเจ็ค (object) แบบขนาน

รูปแบบ :

```
mapply (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
```

อาร์กิวเมนต์สำหรับใช้ในฟังก์ชัน mapply ประกอบด้วย

1) FUN หมายถึง ฟังก์ชันที่จะเรียกใช้งาน เช่น ฟังก์ชันทางคณิตศาสตร์ ฟังก์ชันที่ผู้ใช้สร้างขึ้นมาใช้งานเอง

2) ... หมายถึง ออบเจ็คลิสต์ (list) ในภาษา R ที่ต้องการเรียกใช้

3) MoreArgs หมายถึง การระบุรายการอาร์กิวเมนต์ที่ถูกเรียกใช้ในฟังก์ชัน

4) SIMPLIFY หมายถึง การระบุการลดความซับซ้อนของผลลัพธ์หรือไม่ ซึ่งจะระบุเป็นข้อมูลตรรกศาสตร์ (logical) หรือข้อความ

5) USE.NAMES หมายถึง การระบุให้แสดงชื่อของอาร์กิวเมนต์แรกหรือไม่

ตัวอย่างการทำงานของฟังก์ชัน mapply สามารถแสดงดังรายการตัวอย่างด้านล่างดังต่อไปนี้

ตัวอย่าง 1: การเรียกใช้ฟังก์ชัน mapply

```
mapply(rep, 1:4, 4:1)
```

ตัวอย่าง 2: การเรียกใช้ฟังก์ชัน mapply แบบมีการระบุจำนวนครั้งในการทำงาน

```
mapply(rep, times = 1:4, x = 4:1)
```

ตัวอย่าง 3: การเรียกใช้ฟังก์ชัน mapply แบบมีการเรียกใช้ฟังก์ชันที่ผู้ใช้เขียนขึ้นมาใช้เอง

```
mapply(function(x,y){x^y},x=c(2,3),y=c(3,4))
```

ตัวอย่าง 4: การเรียกใช้ฟังก์ชัน mapply แบบไม่แสดงชื่อของเวกเตอร์

```
mapply(function(x,y){x^y},c(a=2,b=3),c(A=3,B=4),USE.NAMES=FALSE)
```

ตัวอย่าง 5: การเรียกใช้ฟังก์ชัน mapply แบบมีการระบุ MoreArgs

```
mapply(function(x,y,z,k){(x+k)^(y+z)},c(a=2,b=3),c(A=3,B=4),MoreArgs=list(1,2))
```

ตัวอย่าง 6: การเรียกใช้ฟังก์ชัน mapply แบบมีการระบุ MoreArgs

```
mapply(rep, times = 1:4, MoreArgs = list(x = 42))
```

ตัวอย่าง 7: การเรียกใช้ฟังก์ชัน mapply แบบแสดงชื่อข้อมูลจากเวกเตอร์แรก

```
mapply(function(x, y) seq_len(x) + y,
c(a =  1, b = 2, c = 3),  # names from first
c(A = 10, B = 0, C = -10))
```