หัวข้อในบทที่ 3

- ประโยคและและโปรแกรมบล็อก
- ประโยคเงื่อนไขควบคุมและแบ่งสายงาน
- การวนลูปหรือทำขั้นตอนซ้ำ
- การใช้ประโยคคำสั่ง **goto**
- การใช้ประโยคคำสั่ง **break** และ **continue**

3

ขั้นตอนการทำงานและสายงานควบคุม

3.1 ประโยคและและโปรแกรมบล็อก

ในการวิเคราะห์การทำงานของโปรแกรมรวมถึงการออกแบบสร้างโปรแกรมคอมพิวเตอร์ เรามักจะแบ่งขั้นตอนการทำงานของโปรแกรมออกเป็นส่วนต่างๆตามจุดประสงค์ ซึ่งเราจะเรียกว่า *โปรแกรมบล็อก* (Program Block) โดยเรามองส่วนใดส่วนหนึ่งของโปรแกรมว่าเป็นเสมือนกับ *กล่องดำ* (Black Box) และไม่สนใจว่าภายในกล่องดำจะมีรูปร่างหน้าตาอย่างไร แต่ที่สำคัญกล่องดำนี้จะต้องมีคุณสมบัติและหน้าที่ตามที่เราได้กำหนดไว้อย่างชัดเจนในภายหลัง

เวลาเราออกแบบสร้างโปรแกรม เราอาจจะใช้วิธีวางโครงสร้างของโปรแกรมอย่างคร่าวๆ โดยอาศัยส่วนต่างๆที่มีลักษณะเหมือนกล่องดำ คือมีแต่การกำหนดคุณสมบัติและหน้าที่ (Specification) เท่านั้น ทำให้เรามองภาพการทำงานของโปรแกรมโดยรวมได้ง่ายขึ้น ต่อจากนั้นเราก็เริ่มสร้างองค์ประกอบต่างๆซึ่งเราจะเกี่ยวข้องกับคำถามที่ว่า จะทำอย่างไรให้กล่องดำมีคุณสมบัติตามที่กำหนดไว้ ใช้วิธีการอะไรและอย่างไร ภายในกล่องดำนี้เราอาจจะมีการแบ่งขั้นตอนการทำงานออกเป็นบล็อกย่อยให้เล็กลงไปอีก ถ้าเรายังคิดว่ากล่องดำที่เรากำลังจะสร้างนี้ยังคงมีความซับซ้อน (Complexity) อยู่มาก การแบ่งโปรแกรมออกเป็นส่วนย่อยๆตามหลักของการออกแบบจากบนลงล่าง (Top-Down Design) นอกจากจะทำให้เรามองเห็นภาพการทำงานของโปรแกรมได้ง่ายแล้วยังเปิดโอกาสให้เราสร้างองค์ประกอบเหล่านี้ได้ในเวลาพร้อมกันถ้าองค์ประกอบแต่ละส่วนเหล่านั้นมีอิสระจากกันเมื่อทำงาน ซึ่งจะมีประโยชน์มากเมื่อมีการทำงานเป็นทีม โดยนักเขียนโปรแกรมแต่ละคนสามารถสร้างแต่ละส่วนของโปรแกรมได้ในเวลาพร้อมกัน แต่ถ้าส่วนใดภายในกล่องดำต้องอาศัยส่วนอื่นๆที่เล็กลงไปอีกเราก็จัดให้อยู่ในสายงานเดียวกัน ภายในสายงานเดียวกันเราอาจจะสร้างจากส่วนย่อยแล้วประกอบกันขึ้นเป็นกล่องดำหรือส่วนที่ซับซ้อน

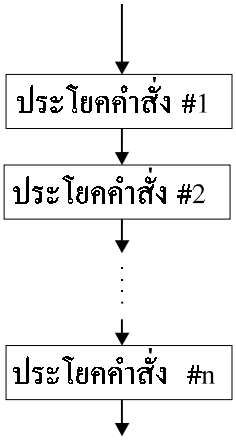
มากขึ้นก็ได้ เราเรียกการออกแบบในลักษณะนี้ว่า การออกแบบจากล่างขึ้นบน (Bottom-Up Design)

ในภาษาซีหรือภาษาระดับสูงอื่นๆ ขั้นตอนการทำงานจะถูกจัดเป็นกลุ่มของคำสั่ง เช่น ประโยคคำสั่งเดี่ยว (Single Statement) หรือเชิงซ้อน (Compound Statement) และฟังก์ชัน ซึ่งเรามองว่าเป็นบล็อกของโปรแกรมและแตกต่างกันไปตามความซับซ้อนของหน้าที่และองค์ประกอบ

ในบทนี้เราจะทำความรู้จักกับโครงสร้างของประโยคและรูปแบบที่ใช้ในการกำหนดเงื่อนไขการทำงานและการแบ่งสายงาน รวมทั้งการทำขั้นตอนซ้ำหรือวนลูป (Looping) ซึ่งถือว่าเป็นหลักการพื้นฐานสำคัญในการเขียนโปรแกรมคอมพิวเตอร์

3.1.1 การกระทำตามลำดับ

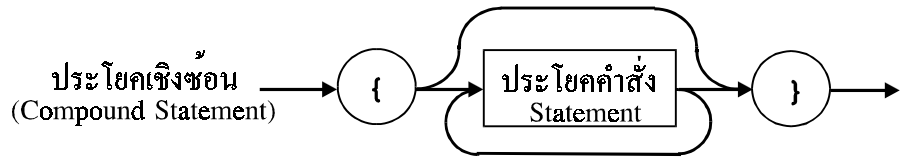
ตามปรกติแล้ว การดำเนินการหรือปฏิบัติตามคำสั่งของคอมพิวเตอร์จะเกิดขึ้นตามลำดับของประโยคคำสั่งที่เราเขียนเรียงต่อกันในโปรแกรม ซึ่งเราเรียกการกระทำในลักษณะนี้ว่า Sequential Execution



รูปภาพที่ 3.1 ผังงานแสดงการกระทำตามลำดับ

เมื่อคอมพิวเตอร์ทำงานก็จะเริ่มกระทำการจากคำสั่งหนึ่งไปสู่อีกคำสั่งหนึ่งถัดไปตามที่เราเขียนในโปรแกรมโค้ดจนกว่าจะมีการสั่งให้โปรแกรมหยุดทำงาน ผังงานข้างบนตามรูปภาพที่ 3.1 มีลักษณะเป็นสายงานเดี่ยว (Flow Line) ในแต่ละช่องสี่เหลี่ยมก็หมายถึงประโยคคำสั่งเดี่ยว หรือเชิงซ้อนก็ได้ ซึ่งเราอาจจะเรียกว่า บล็อก ประโยคคำสั่งเชิงซ้อนในภาษาซีจะมีลักษณะพิเศษที่เรา

สามารถสังเกตได้จากเครื่องหมายวงเล็บปีกกาทั้งสองซึ่งใช้กำหนดขอบเขตของประโยคคำสั่งชนิดนี้



รูปภาพที่ 3.2 โครงสร้างของประโยคเชิงซ้อน

ประโยคคำสั่งเดี่ยวที่ไม่มีนิพจน์ใดๆ มีแต่เพียง White Space (หรือที่ว่าง) และเครื่องหมายเซมิโคลอน(;)ที่ใช้สำหรับแบ่งแยกจากประโยคคำสั่งอื่น หรือประโยคคำสั่งเชิงซ้อนที่ไม่มีข้อความหรือประโยคคำสั่งใดๆระหว่างเครื่องหมายวงเล็บปีกกาเปิดและปิด ({ และ } ตามลำดับ) เราก็เรียกว่า Null Statement

```
;           /* null simple statement */
{ }         /* null compound statement */
```

เราทบทวนอีกครั้งหนึ่งว่า ประโยคคำสั่งใดๆที่เป็นประโยคคำสั่งเดี่ยวจะถูกแบ่งแยกออกจากกันโดยอาศัยเครื่องหมาย ; สำหรับประโยคคำสั่งเชิงซ้อนหรือบล็อกจะอาศัยเครื่องหมายวงเล็บปีกกาเปิดและปิดเป็นคู่ในการกำหนดขอบเขตของประโยค ดังนั้นประโยคเช่นนี้จึงไม่ต้องมีเครื่องหมาย ; อยู่ข้างท้ายเหมือนประโยคคำสั่งเดี่ยว

3.1.2 การเลือกกระทำ

การกระทำแบบตัวเลือกหรือ Selective Execution มีความแตกต่างจากการกระทำแบบแรกคือเราสามารถเลือกสายงานในการทำงาน โดยอาศัยการกำหนดเงื่อนไข (Condition) ขึ้นมาใช้ เช่น ถ้าเงื่อนไขถูกเป็น ‘จริง’ ก็ให้เลือกทำงานในสายงานหนึ่ง แต่ถ้าไม่เป็นไปตามเงื่อนไขที่กำหนดเราก็กำหนดให้ทำานในสายงานอีกสายหนึ่ง หรือไม่ต้องทำขั้นตอนใดๆก็ได้ เงื่อนไขที่เรานำมาใช้ในการกำหนดแบบหรือสายงานสำหรับคำสั่งของคอมพิวเตอร์จะเป็นนิพจน์ทางตรรกศาสตร์ (Boolean Expression) ซึ่งมีหลักในการจำง่ายๆ คือ ถ้านิพจน์เงื่อนไขมีค่าเท่ากับศูนย์ ก็หมายถึงนิพจน์มีค่าเป็น ‘เท็จ’ และถ้ามีค่าเป็นหนึ่ง ก็หมายถึง ‘จริง’

นอกจากนิพจน์ทางตรรกศาสตร์แล้ว เรายังสามารถใช้นิพจน์ใดๆก็ได้ที่ให้ค่าคงที่สำหรับทำหน้าที่เป็นนิพจน์เงื่อนไขในการแบ่งสายงาน ซึ่งในภาษาซีมีการกำหนดไว้ว่า ถ้านิพจน์เงื่อนไขนี้ให้ค่าเท่ากับศูนย์ (0 แบบ `integer` หรือ 0.0 ที่เป็นทศนิยมก็ได้) จะมีค่าทางตรรกศาสตร์เป็น 'เท็จ' และ นิพจน์จะให้ค่าทางตรรกศาสตร์เป็น 'จริง' ก็ต่อเมื่อนิพจน์นี้ให้ค่าใดๆที่ไม่เท่ากับศูนย์

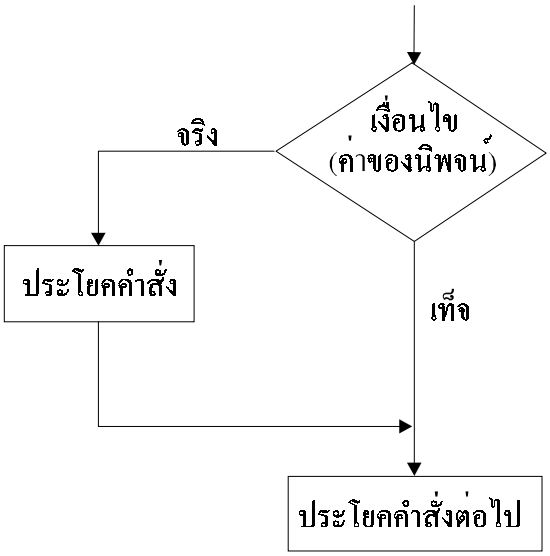
เราได้ทำความรู้จักกับการใช้โอเปอร์เรเตอร์หลายๆตัวในบทที่แล้ว เช่นโอเปอร์เรเตอร์ในทางตรรกศาสตร์ (Logical Operator) โอเปอร์เรเตอร์เชิงเปรียบเทียบหรือหาความสัมพันธ์ระหว่างเลขสองจำนวน (Relational Operator) ในบทนี้เราจะนำโอเปอร์เรเตอร์เหล่านี้มาใช้ในการสร้างนิพจน์เงื่อนไขแบบธรรมดาหรือซับซ้อน ซึ่งเป็นองค์ประกอบที่สำคัญของโครงสร้างเงื่อนไข หรือประโยคคำสั่งที่เราใช้ในการแบ่งขั้นตอนการทำงานของส่วนใดส่วนหนึ่งของโปรแกรมออกเป็นหลายสายงาน เช่น ประโยค `if-else` หรือ `switch-case` เป็นต้น

3.2 ประโยคเงื่อนไขควบคุมและแบ่งสายงาน (Conditional Statement)

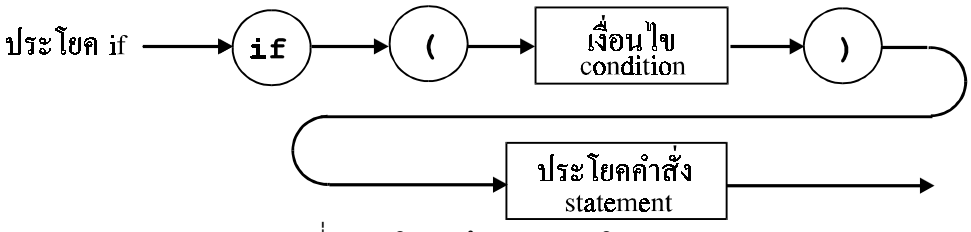
3.2.1 ประโยค `if` แบบง่าย

การสร้างประโยคคำสั่งแบบ `if` มีจุดประสงค์ก็คือ การแบ่งสายงานออกเป็นสองสาย โดยอาศัยนิพจน์เงื่อนไขดังที่กล่าวไปแล้วในการควบคุม หรือกำหนดว่า เมื่อ ใดคอมพิวเตอร์จะทำงานในสายงานใด โดยมีผังงานตามรูปภาพที่ 3.3

จากรูปภาพประกอบ `statement` เป็นได้ทั้งประโยคคำสั่งเดี่ยวและประโยคคำสั่งเชิงซ้อน เราใช้ประโยคคำสั่ง `if` แบบธรรมดานี้เป็นตัวที่กำหนดว่า ถ้าเงื่อนไขของ `if` เป็นจริง ก็ให้กระทำการตามคำสั่งที่อยู่ในโครงสร้างของ `if` แต่ถ้าเงื่อนไขเป็นเท็จก็ไม่ต้องทำตามประโยคคำสั่งนี้ ซึ่งก็คือการข้ามขั้นตอนนี้ไปยังประโยคคำสั่งถัดไป



รูปภาพที่ 3.3 ผังงานแสดงการทำงานของประโยคแบบ `if`



รูปภาพที่ 3.4 โครงสร้างของประโยคแบบ `if`

การใช้ประโยคคำสั่ง `if` มีรูปแบบดังนี้ (เปรียบเทียบกับรูปภาพที่ 3.4)

```
if (condition)
      statement
```

ตัวอย่างการใช้งานประโยคแบบ `if` เช่น ถ้าเรากำหนดว่าตัวแปร `x` เป็นตัวแปรแบบ `float` และเราต้องการทราบว่าในขณะนั้น `x` มีค่าเท่ากับศูนย์หรือไม่ ถ้า `x` มีค่าเท่ากับศูนย์ก็ให้พิมพ์ข้อความออกทางจอภาพโดยใช้คำสั่ง `printf()`

```
if (x == 0.0)
      printf ("x is equal to 0.\n");
```

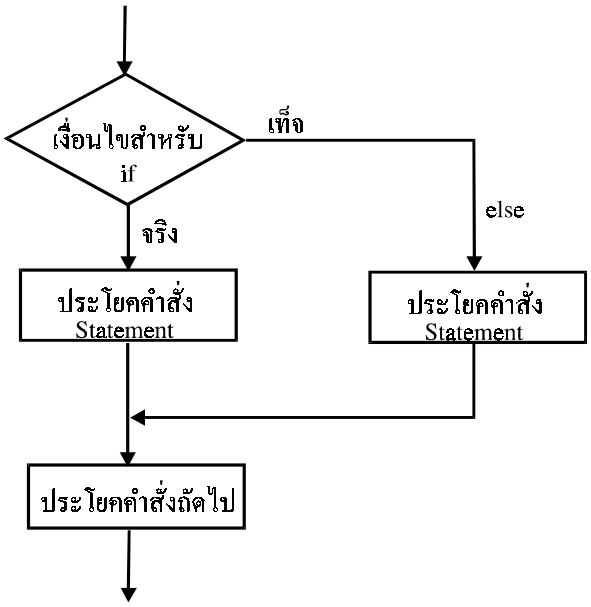
เราจะเห็นได้ว่า คำสั่งที่ใช้ฟังก์ชัน `printf()` เป็นประโยคคำสั่งเดี่ยว แต่ถ้าเราต้องการให้ประโยคที่อยู่หลังเงื่อนไขของ `if` เป็นประโยคเชิงซ้อน เราก็เขียนใหม่ดังนี้โดยอาศัยเครื่องหมายวงเล็บปีกกาเข้ามาช่วย

```
if (x == 0.0)
{
      printf ("x is equal to 0.\n");
}
```

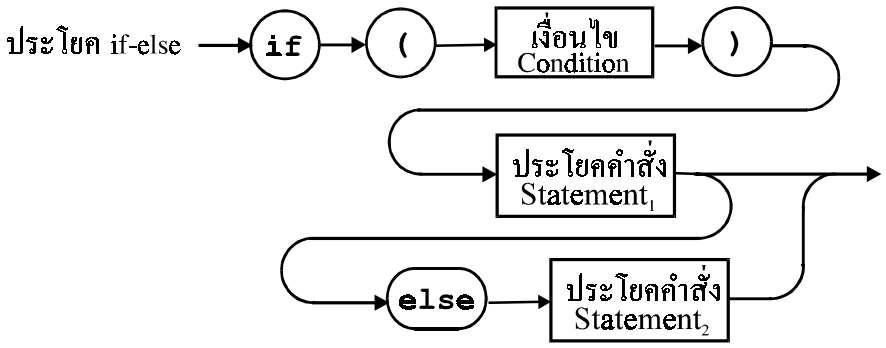
3.2.2 ประโยค if-else แบบสมบูรณ์

ตามหลักแล้วประโยค `if` ธรรมดาเป็นรูปแบบอย่างง่ายที่สุดของประโยค `if-else` ซึ่งเราใช้ในการแบ่งสายงานออกเป็นสองสายซึ่งก็คือเลือกกระทำขั้นตอนใดขั้นตอนหนึ่งที่อยู่ในโครงสร้างของประโยค `if-else` ถ้าเงื่อนไขเป็นจริงก็ให้ทำประโยคคำสั่งที่หนึ่ง ($statement_1$) แต่ถ้าเงื่อนไขเป็นเท็จก็ให้ทำประโยคคำสั่งที่สอง ($statement_2$) รูปแบบการใช้ประโยค `if-else` ก็ไม่แตกต่างจากการใช้ประโยค `if` ธรรมดา เพียงแต่เพิ่มส่วนโครงสร้างของ `else` ต่อท้าย เวลาอ่านประโยค `if-else` นี้เราก็อ่านว่า “ถ้าเงื่อนไข ... เป็นจริงแล้วก็ให้ทำ .. มิฉะนั้นแล้วก็ให้ทำ .... ”

```
if (condition)
      statement1
else
      statement2
```



รูปภาพที่ 3.5 ผังงานแสดงการทำงานของ `if - else`



รูปภาพที่ 3.6 โครงสร้างของประโยคแบบ `if - else`

ดังนั้นประโยค `if` ธรรมดาจึงให้ผลเหมือนกับการแทนที่ $statement_2$ ในส่วนโครงสร้างของ `else` ในประโยคแบบ `if-else` ด้วยประโยคคำสั่งที่เป็น Null Statement

```
if (condition)
      statement1
else
      ;
```

หรือ

```
if (condition)
      statement1
else
{ }
```

โปรดสังเกตว่า $statement_1$ และ $statement_2$ เป็นประโยคคำสั่งใดๆ หรือ บล็อกก็ได้ แต่เพื่อที่จะทำให้โปรแกรมโค้ดอ่านได้ง่ายขึ้น เราควรจะเปลี่ยนประโยคเดี่ยวให้เป็นประโยคเชิงซ้อนภายในโครงสร้างของประโยค `if-else` โดยใช้เครื่องหมายวงเล็บปีกกาในการกำหนดขอบเขต เช่นเดียวกับการกำหนดบล็อก ในภาษาปาสคาสที่เราใช้คำว่า `BEGIN` และ `END` เราลองพิจารณาตัวอย่างข้างล่างนี้

```
if (x == 0.0)
   printf ("x is equal to 0.\n");
else
   printf ("x is greater or less than 0.\n");
```

ตัวอย่างข้างบนใช้แต่ประโยคเดี่ยวในโครงสร้างของ `if-else` หรือถ้าเราเขียนใหม่โดยใช้ประโยคเชิงซ้อน ก็จะทำให้เราเห็นโครงสร้างของประโยคได้ชัดเจนมากขึ้น

```
if (x == 0.0)
{
   printf ("x is equal to 0.\n");
}
else
{
```

```
   printf ("x is greater or less than 0.\n");
}
```

นอกจากจะทำให้เราอ่านโครงสร้างประโยคได้ง่ายขึ้นแล้ว ในบางครั้งเราต้องการเพิ่มเติมประโยคคำสั่งอื่นๆเข้าไปอีกในภายหลัง ถ้าเราเขียนประโยคที่ตามหลังคำว่า `if` และ `else` ให้เป็นประโยคเชิงซ้อนอยู่แล้ว เราก็เพิ่มเติมคำสั่งอื่นๆได้เลย แต่ถ้าเราเขียนเป็นประโยคเดี่ยวเท่านั้น เมื่อต้องการเพิ่มเติมคำสั่งใดๆในแต่ละสายงาน เราก็ต้องเปลี่ยนให้เป็นประโยคเชิงซ้อนก่อนโดยใช้เครื่องหมายปีกกาคู่ แต่ถ้าเราลืมหลักข้อนี้ บางครั้งก็อาจจะทำให้เกิดความผิดพลาดในโปรแกรมของเราได้ ตัวอย่างเช่น เราต้องการจะตรวจสอบว่าค่า `x` มีค่าเท่ากับศูนย์หรือไม่ และกำหนดให้ `y` มีค่าเท่ากับศูนย์เมื่อ `x` มีค่าเท่ากับศูนย์ และ `y` มีค่าเท่ากับหนึ่งเมื่อ `x` ไม่เท่ากับศูนย์ ถ้าเราเขียนว่า

```
if (x == 0.0)
   printf ("x is equal to 0.\n");
   y = 0;
else
   printf ("x is greater or less than 0.\n");
   y = 1;
```

ก็จะผิดหลักไวยกรณ์ในภาษาซี เนื่องจากว่าในประโยคเงื่อนไขแบบ `if-else` จะต้องมีเพียงหนึ่งประโยคคำสั่ง (หรือบล็อก) เท่านั้นที่อยู่ข้างหลังคำว่า `if` และ `else` แต่ในตัวอย่างข้างบนเราใช้ประโยคคำสั่งเดี่ยวสองประโยคตามหลังคำว่า `if` และวางอยู่หน้าคำว่า `else` ดังนั้นจึงผิดหลักไวยากรณ์ในภาษาซี แต่ถ้าเราแก้ไขใหม่เป็น

```
if (x == 0.0)
{
   printf ("x is equal to 0.\n");
   y = 0;
}
else
   printf ("x is greater or less than 0.\n");
   y = 1;
```

ก็จะถูกต้องตามหลักไวยกรณ์ในภาษาซี แต่ทว่ายังคงมีที่ผิดอยู่ เนื่องจากว่าเรายังไม่ได้แก้ไขในส่วนของ `else` เพื่อแปลงให้ประโยคที่ตามมาเป็นประโยคเชิงซ้อน เพราะว่าประโยคคำสั่ง `y = 1;` ตามตัวอย่างนั้นไม่ได้อยู่ในโครงสร้างของ `else` (และอยู่นอกโครงสร้างของประโยค `if-else`) ดังนั้นไม่ว่า `x` จะมีค่าเท่าใด `y` จะมีค่าเป็นหนึ่งเสมอ ซึ่งไม่ถูกต้องตามที่เรากำหนดเอาไว้ และที่ถูกต้องก็คือจะต้องแก้ไขใหม่ดังนี้

```
if (x == 0.0)
{
   printf ("x is equal to 0.\n");
   y = 0;
}
else
{
   printf ("x is greater or less than 0.\n");
   y = 1;
}
```

นอกจากนี้ยังมีความผิดพลาดที่พบได้บ่อยในการใช้ if หรือ if-else แต่ไม่ผิดหลักไวยากรณ์ เช่น

```
if (x == 0.0);
   printf ("x is equal to 0.\n");
```

โปรดสังเกตว่า เครื่องหมาย ; หรือ เซมิโคลอนที่อยู่ท้ายบรรทัดแรกจะให้ผลคือว่า เมื่อ x มีค่าเท่ากับศูนย์ แล้วไม่ต้องทำขั้นตอนใดๆ (แทนที่จะพิมพ์ข้อความเหมือนในตัวอย่างที่แล้ว) เพราะประโยคคำสั่งในโครงสร้างของ if นั้นเป็นประโยคคำสั่งเดี่ยวแบบว่างเปล่า หรือ Null Statement และทำให้ประโยคคำสั่งที่ใช้พิมพ์ข้อความออกทางจอภาพไม่เป็นส่วนหนึ่งของโครงสร้าง if เพราะเครื่องหมาย ; ที่เราเติมเอาไว้ก่อนหน้านี้ (ซึ่งเครื่องหมาย ; ที่เกินมานี้ เราอาจจะเติมโดยมิได้ตั้งใจในขณะที่เราเขียนโปรแกรมโค้ด) ดังนั้นไม่ว่า x จะมีค่าเท่ากับศูนย์หรือไม่ โปรแกรมจะพิมพ์ข้อความว่า x มีค่าเท่ากับศูนย์เสมอ

3.2.3 ประโยค if-else-if

เนื่องจากว่า ประโยค if-else นั้นจัดได้ว่าเป็นประโยคคำสั่งเชิงซ้อนชนิดหนึ่ง เราสามารถใช้ประโยคเหล่านี้ในโครงสร้างของประโยคเงื่อนไขใดๆได้ บางครั้งเราต้องการใช้ประโยค if-else ซ้อนกันหลายๆชั้นเหมือนกับว่าเป็นประโยคเงื่อนไขแบบลูกโซ่ ซึ่งมีรูปแบบการใช้งานดังนี้

```
if (condition1)
      statement1
else
      if (condition2)
            statement2
              .
              .
              .
            if (conditionN)
                  statementN
            else
                  statementN+1
```

ตัวอย่างการใช้งาน เช่น เราต้องการตรวจสอบดูว่าค่าของตัวแปร x อยู่ในช่วงใดต่อไปนี้

```
  x < 0          :   range = 0
  0 ≤ x < 10     :   range = 1
 10 ≤ x < 100    :   range = 2
100 ≤ x          :   range = 3
```

โดยกำหนดให้ x เป็นตัวแปรแบบ float และ range เป็นตัวแปรแบบ int

```
if (x < 0.0)
{
      range = 0;
}
else if (x >= 0.0 && x < 10.0)
{
      range = 1;
}
else if (x >= 10.0 && x < 100.0)
{
      range = 2;
}
else  range = 3;
```

โปรดสังเกตว่า ตัวอย่างข้างบนมีขั้นตอนทำงานถูกต้องตามที่เราต้องการ แต่มีโอเปอร์เรชันที่เรียกได้ว่าไม่จำเป็น โดยเราสามารถแก้ไขใหม่ให้ดีขึ้นได้ดังนี้

```
if (x < 0.0)
{
      range = 0;
}
else if (x < 10.0)
{
      range = 1;
}
else if (x < 100.0)
{
      range = 2;
}
else  range = 3;
```

ขอให้ผู้อ่านลองค้นหาคำตอบด้วยตนเองว่าทำไมในโปรแกรมตัวอย่างมีการใช้โอเปอร์เรเตอร์ && ที่ไม่จำเป็น

การใช้ประโยค if-else-if แบบลูกโซ่นั้นบางครั้งอาจจะทำให้เกิดปัญหาได้โดยเฉพาะเมื่อเราสร้างเงื่อนไขที่แบ่งออกเป็นหลายๆกรณีจนมากเกินไป หรือกล่าวได้ว่า เราซ้อนประโยค if-else มากจนเกินไป นอกจากนี้ควรจะต้องระมัดระวังเรื่องการวางโครงสร้างของประโยคสำหรับส่วน else ด้วย ตัวอย่างเช่น

```
if (x >= 0.0)
      if (y <= 0.0)
            y = x;
else
      y = -x;
```

จะให้ผลเหมือนกับประโยค if-else-if ต่อไปนี้

```
if (x >= 0.0)
{
      if (y <= 0.0)
      {
            y = x;
      }
      else
```

```
        {
                y = -x;
        }
}
```

และแตกต่างจากกรณีที่เราเขียนว่า

```
if (x >= 0.0)
{
        if (y <= 0.0)
        {
                y = x;
        }
}
else
{
        y = -x;
}
```

จากตัวอย่างข้างบนเราจะเห็นได้ว่า การใช้เครื่องหมายวงเล็บปีกกานั้น ทำให้เราสามารถอ่านและทำความเข้าใจโครงสร้างของประโยค `if-else-if` ได้ง่ายขึ้น และสามารถตรวจสอบได้ง่ายว่าสิ่งที่เราเขียนขึ้นนั้นถูกต้องตามที่เราต้องการจริงหรือไม่

3.2.4 ประโยค `switch-case-default`

เราจะเห็นได้ว่าการใช้ประโยคคำสั่งแบบ `if-else-if` สำหรับกำหนดเงื่อนไขและแบ่งสายงานออกเป็นหลายๆกรณีนั้นจะดูยุ่งยากและซับซ้อนเกินไปถ้าเราแบ่งออกเป็นหลายกรณีมากเกินไป เราลองพิจารณาตัวอย่างต่อไปนี้ สมมุติว่า เรากำหนดตัวแปร `day` สำหรับเก็บข้อมูลแบบ `int` ซึ่งมีค่าอยู่ระหว่าง 1 ถึง 7 ใช้แทนชื่อของวันในหนึ่งสัปดาห์ โดยมีความหมายต่อไปนี้ ถ้า `day` มีค่าเท่ากับ 1 ก็หมายถึงวันอาทิตย์ 2 หมายถึงวันจันทร์ ไปเรื่อยๆจนถึง 7 ซึ่งหมายถึงวันเสาร์ ถ้าเราต้องการเขียนขั้นตอนที่พิมพ์ข้อ ความหรือชื่อของวันแทนที่ด้วยตัวเลขของตัวแปร `day` นี้ออกทางจอภาพโดยใช้ฟังก์ชัน `printf()` เราก็สามารถเขียนได้ดังนี้

```
if (day == 1)
      printf ("Sunday\n");
else if (day == 2)
      printf ("Monday\n");
else if (day == 3)
      printf ("Tuesday\n");
else if (day ==4)
      printf ("Wednesday\n");
else if (day ==5)
      printf ("Thursday\n");
else if (day ==6)
      printf ("Friday\n");
else if (day ==7)
      printf ("Saturday\n");
else
      printf ("Unknown\n");
```

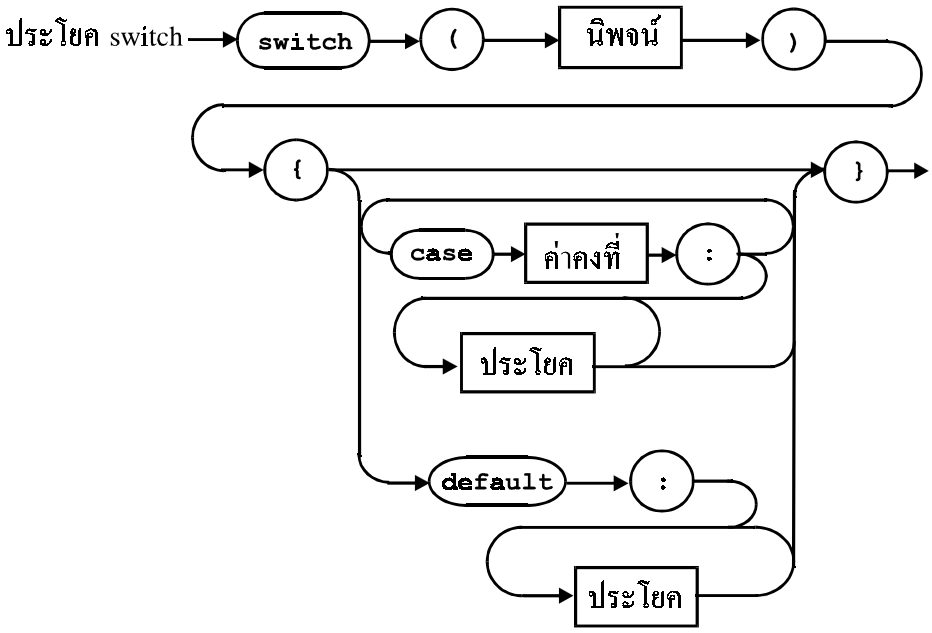
แต่ในภาษาซี ยังมีโครงสร้างของประโยคเงื่อนไขอีกแบบหนึ่ง ที่เรานิยมใช้เมื่อเราต้องการแบ่งการทำงาน ออกเป็นหลายๆสายงานและมีหลายกรณี (หรือตัวเลือก) ประโยคที่เหมาะสมกับหน้าที่เช่นนี้ คือ switch-case ซึ่งมีรูปแบบการใช้งานดังนี้

```
switch (expression)
{
      case constant1 :
            statement_sequence1
            break;
      case constant2 :
            statement_sequence2
            break;

        .
        .
        .
        .

      case constantN :
            statement_sequenceN
            break;
      default :
            statement_sequenceN+1
}
```

โดยที่ expression เป็นนิพจน์ใดๆที่ให้ค่าคงที่ซึ่งมักจะเป็นตัวแปร (ยกเว้นข้อมูลที่เป็นเลขทศนิยมและ ข้อความในภาษาซี) คำว่า case เป็นการกำหนดกรณีหรือตัวเลือกที่แตกต่างกันออกไป ซึ่งจะตามด้วยค่าคงที่และจะต้องเป็นค่าใดค่าหนึ่งที่เท่ากับค่าของ expression ที่เป็นไปได้ เมื่อโปรแกรมดำเนินการมาถึงประโยค switch-case ก็จะเริ่มเปรียบเทียบค่าของ expression กับตัวเลือกที่อยู่ในรายการโดยเป็นไปตามลำดับที่พบอยู่ในโครงสร้างของประโยค ถ้า expression มีค่าเท่ากับค่าคงที่ใดจากกรณี หรือตัวเลือกทั้งหมดที่เราได้กำหนดไว้ ซึ่งก็คือ $constant_1$, $constant_2$, ..., $constant_N$ ที่มีค่าแตกต่างกันไป โปรแกรมก็จะทำตามประโยคคำสั่งที่ตามมาในสายงานนั้น แต่ถ้า expression ให้ค่าที่ไม่ตรงกับกรณีหรือตัวเลือกต่างๆที่เรากำหนดไว้ในรายการ เราก็สามารถใช้คำว่า default แทนกรณีที่นอกเหนือจากกรณีที่กำหนดไว้ได้ซึ่งหมายความว่า ถ้าค่าของ expression ไม่ตรงกับตัวเลือก $constant_1$, $constant_2$, ..., $constant_N$ โปรแกรมก็จะทำงานในสายงานของ default ในบางครั้งเราก็ไม่จำเป็นต้องใช้สายงานของ default ในโครงสร้างของประโยค และถ้าเราตัดสายงานนี้ออกก็จะส่งผลให้โปรแกรมไม่ต้องทำคำสั่งใดๆในประโยค switch-case เมื่อค่าของ expression ไม่ตรงกับค่าของ $constant_1$, $constant_2$, ...,$constant_N$ หรือ เราอาจจะเขียนกรณีของ default ไว้ก็ได้ แต่ประโยคคำสั่งที่ตามมาในสายงานนี้ เราจะเขียนให้เป็น Null Statement ก็ย่อมให้ผลเหมือนกัน



รูปภาพที่ 3.7 โครงสร้างของประโยคแบบ `switch - case`

จากตัวอย่างที่แล้วซึ่งเราได้ใช้ประโยคแบบ `if-else-if` ในการแบ่งแยกกรณี เราสามารถเขียนใหม่โดยใช้ ประโยค `switch-case`

```
switch (day)
{
  case   1 :  printf ("Sunday\n");
              break;
  case   2 :  printf ("Monday\n");
              break;
  case   3 :  printf ("Tuesday\n");
              break;
  case   4 :  printf ("Wednesday\n");
              break;
  case   5 :  printf ("Thursday\n");
              break;
  case   6 :  printf ("Friday\n");
              break;
  case   7 :  printf ("Saturday\n");
              break;
  default :   printf ("Unknown option! \n");
}
```

จะเห็นได้ว่า เราสามารถอ่านโปรแกรมโค้ดในส่วนของประโยค `switch-case` ได้ง่ายขึ้น ในตัวอย่างนี้ เราใช้ `default` เพื่อจัดการกับกรณีที่ค่าของตัวแปร `day` มีค่ามากกว่า 7 หรือน้อยกว่า 1 แต่ถ้าเราไม่ต้อง การให้พิมพ์ข้อความใดๆ เราก็ลบประโยคคำสั่งของฟังก์ชัน `printf()` ออก และเขียนประโยคว่างเปล่า (Null Statement) แทนที่

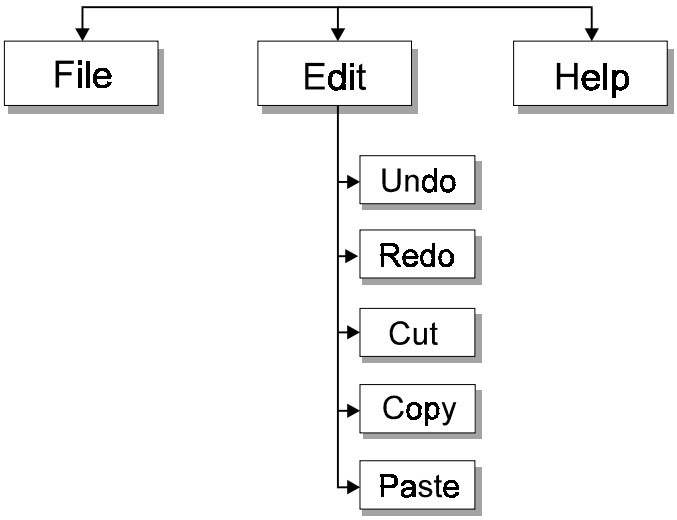
คำสั่ง break ในโครงสร้างของประโยค switch-case นั้นเราใช้เมื่อเราต้องการให้โปรแกรมกระทำขั้นตอนในสายงานที่เราต้องการเท่านั้นและเมื่อโปรแกรมทำงานมาถึงคำสั่ง break ในสายงานดังกล่าวซึ่งเป็นประโยคคำสั่งสุดท้ายของสายงาน โปรแกรมก็จะออกจากโครงสร้างของประโยค switch-case ไปยังประโยคคำสั่งถัดไปในโปรแกรม ถ้าเราไม่เขียนประโยคคำสั่ง break; เป็นคำสั่งท้ายสุดในสายงานของแต่ละตัวเลือก โปรแกรมก็จะกระทำคำสั่งต่างๆในสายงานของตัวเลือกอื่นๆที่อยู่ถัดไปจนถึงท้ายสุดของประโยค switch-case โดยอัตโนมัติ โปรดสังเกตว่า เราไม่จำเป็นต้องใช้คำสั่ง break กับ กรณีตัวเลือกของ default เพราะเป็นสายงานตัวเลือกที่อยู่ท้ายสุดของโครงสร้าง

```
switch (digit)
{
      case 2:
      case 3:
      case 5:
      case 7:
            printf("%d is a prime number.\n", digit);
            break;
      default:
            if (digit >= 0 && digit <= 9)
              printf("%d is not a prime number.\n", digit);
            else
              printf("Invalid option: %d\n", digit);
}
```

ตัวอย่างนี้แสดงให้เห็นการใช้ประโยคคำสั่ง break; ในสายงานตัวเลือกของประโยค switch-case ซึ่ง เราจะเห็นได้ว่าโปรแกรมจะพิมพ์ข้อความแจ้งให้เราทราบว่า ตัวแปร digit เป็นจำนวนเฉพาะเมื่อตัวแปรนี้มีค่าเท่ากับ 2,3,5 หรือ 7 ค่าใดค่าหนึ่ง แต่ถ้า digit มีค่านอกเหนือจากตัวเลือกเหล่านี้ก็จะพิมพ์ข้อความ บอกว่าตัวแปรนี้มิใช่เลขจำนวนเฉพาะ (โดยเรากำหนดไว้ว่า ตัวแปร digit มีค่าอยู่ระหว่าง 0 ถึง 9 เท่า นั้น) เราจะสังเกตได้ว่า ตัวเลือก 2,3,5 และ 7 มีสายงานการทำงานร่วมกัน โดยที่ตัวเลือก 2,3 และ 5 ไม่มีขั้นตอนการทำงานใดๆ แต่เนื่องจากว่าเราไม่ได้เขียนประโยคคำสั่ง break; ไว้ในสายงานของตัวเลือกดังกล่าว ในกรณีที่ digit มีค่าเท่ากับ 2,3 หรือ 5 โปรแกรมก็จะผ่านไปยังสายงานของตัวเลือก 7 โดยไม่ ต้องคำสั่งใดๆเพราะเนื่องจากว่าไม่มีคำสั่งใดๆอยู่ในสายงานเหล่านี้ เมื่อโปรแกรมดำเนินการมาถึงสายงาน ของตัวเลือกที่ 7 ก็จะกระทำคำสั่งซึ่งก็คือคำสั่งที่มีการเรียกใช้ฟังก์ชัน printf() และต่อจากนั้นเมื่อพบ คำสั่ง break; ซึ่งเป็นคำสั่งถัดไปในสายงาน โปรแกรมก็จะออกจากโครงสร้างของประโยค switch-case เป็นอันว่าจบขั้นตอนการทำงานของประโยค ในกรณีที่ตัวแปร digit มีค่าอื่นๆที่ไม่ใช่ 2,3,5 หรือ 7 โปรแกรมก็จะดำเนินการในสายงาน default โดยตรวจสอบเงื่อนไขของประโยค if-else ในสายงานว่าเป็นจริงหรือไม่ ก่อนที่จะเลือกกระทำสายงานตามเงื่อนไขที่กำหนด

เราจะเห็นได้ว่า ถ้าตัวเลือกหลายๆตัวมีขั้นตอนการทำงานในสายงานของตนที่เหมือนกัน เราก็สามารถเขียนขั้นตอนรวมกันได้ โดยกำหนดให้สายงานของตัวเลือกเหล่านั้นเป็นสายงานว่างเปล่าไม่ต้องมีคำสั่งใดๆ และเขียนขั้นตอนการทำงานไว้ในสายงานของตัวเลือกตัวสุดท้ายของกลุ่ม ตามตัวอย่างที่เราได้พิจารณาไปแล้ว

การใช้ประโยค `switch-case` นั้นเราสามารถใช้ประโยคนี้ซ้อนกันมากกว่าหนึ่งชั้นก็ได้ เราลอง นึกถึง *เมนูคำสั่ง* ของโปรแกรมหรือซอฟต์แวร์ ซึ่งตามหลักการทำงานแล้วเราสามารถสร้างได้โดยใช้ประโยคคำสั่ง `switch-case` โดยผู้ใช้สามารถเลือกคำสั่งหรือการทำงานของโปรแกรมได้จากเมนูคำสั่ง สำหรับคำสั่งที่เป็นตัวเลือกในเมนูคำสั่งนี้ก็อาจจะมีเมนูคำสั่งย่อยลงไปอีก เช่น สมมุติว่า เมนูคำสั่งของโปรแกรมที่เราใช้มีคำสั่งตัวเลือก เช่น `File`, `Edit`, `Help` ถ้าเราเลือกตัวเลือก `Edit` ก็จะมีเมนูย่อยภายใต้คำสั่งตัวเลือก `Edit` อีก เช่น อาจจะประกอบด้วยคำสั่งย่อย `Undo`, `Redo`, `Cut`, `Copy`, `Paste` เป็นต้น



รูปภาพที่ 3.8 ภาพเมนูตัวเลือก

เราลองเขียนส่วนของโปรแกรมอย่างง่ายๆที่แสดงให้เห็นขั้นตอนการเลือกเมนูคำสั่งและการกระทำชุดคำสั่งของตัวเลือกดังกล่าว โดยใช้ประโยคคำสั่ง `switch-case` แต่เราจะไม่คำนึงถึงรายละเอียดต่างๆของโปรแกรมมากนัก ก่อนอื่นเราจะต้องกำหนดค่าคงที่ให้แก่ตัวเลือกต่างๆของเมนูคำสั่ง เช่น เราใช้พรีโปรเซสเซอร์ไดเรคทีฟ ที่มีชื่อว่า `#define` ในการกำหนดค่าของสัญลักษณ์ ตัวอย่างเช่น `FILE_MENU` หมายถึงสัญลักษณ์ในโปรแกรมโค้ดที่ใช้แทนค่าคงที่ `0x100`

```
/* Main Menu */
#define FILE_MENU   0x100
```

```
#define EDIT_MENU   0x200
#define HELP_MENU   0x300

/* Submenu */
#define UNDO        0x201
#define REDO        0x202
#define CUT         0x203
#define COPY        0x204
#define PASTE       0x205
```

ต่อจากนั้นเราก็สร้างโครงสร้างของประโยค switch-case ในการจัดการกับตัวเลือกต่างๆ เพื่อให้ง่ายต่อการทำความเข้าใจเราจะกำหนดใช้ฟังก์ชัน get_choice() ซึ่งเป็นฟังก์ชันที่เราสร้างขึ้นเองสำหรับทำหน้าที่อ่านตัวเลือกของเมนูที่ผู้ใช้ต้องการ เช่น ฟังก์ชันนี้จะพิมพ์ข้อความออกทางจอภาพเพื่อแจ้งให้ผู้ใช้ทราบว่า โปรแกรมมีตัวเลือกใดบ้างที่ผู้ใช้สามารถเลือกได้ จากนั้นก็รอคอยจนกว่าผู้ใช้จะเลือกตัวเลือกที่ต้องการ เมื่อผู้ใช้ได้เลือกตำสั่งใดคำสั่งหนึ่งจากเมนูแล้ว ฟังก์ชันนี้ก็จะให้ค่าของตัวเลือกโดยผ่านเป็นค่าของฟังก์ชันและเก็บไว้ในตัวแปร choice ในกรณีที่ผู้ใช้เลือกตัวเลือก EDIT_MENU ก็จะมีการถามผู้ใช้อีกครั้งว่า ต้องการเลือกเมนูคำสั่งย่อยใด โดยเรียกใช้ฟังก์ชัน get_choice_edit() ซึ่งทำหน้าที่คล้ายกับ get_choice() แต่จะใช้เฉพาะกับตัวเลือกของ EDIT_MENU เท่านั้น

```
choice = get_choice();

switch (choice)
{
      case EDIT_MENU :

                  choice = get_choice_edit();
                  switch (choice)
                  {
                    case UNDO  :  /* Undo.... */
                              undo_submenu();
                              break;
                    case REDO  :  /* Redo.... */
                              redo_submenu();
                              break;
                    case CUT   :  /* Cut..... */
                              cut_submenu();
                              break;
                    case COPY  :  /* Copy.... */
                              copy_submenu();
                              break;
                    case PASTE :  /* Paste... */
                              paste_submenu();
                              break;
                  }
                  break;

      case FILE_MENU : /* File.... */
                  file_menu();
                  break;

      case HELP_MENU : /* Help.... */
                  help_menu();
                  break;
}
```

โปรดสังเกตว่า เราจะไม่พยายามให้รายละเอียดมากนักสำหรับขั้นตอนการทำงานในสายงานย่อยของตัวเลือกต่างๆ แต่ที่เราสนใจคือโครงสร้างการทำงานของประโยค switch-case เท่านั้น ดังนั้นเราจึงเขียนขั้นตอนการทำงานของสายงานต่างๆให้อยู่ในรูปของการเรียกใช้ฟังก์ชัน เช่น

```
undo_submenu()
redo_submenu()
cut_submenu()
copy_submenu()
paste_submenu()
file_menu()
help_menu()
```

โปรดสังเกตว่า นิพจน์ที่เป็นตัวเลือกที่อยู่ถัดจากคำว่า case ในโครงสร้างของประโยคของ swtich-case จะต้องเป็นนิพจน์ค่าคงที่เท่านั้น มิใช่ตัวแปรหรือนิพจน์จากการเรียกใช้ฟังก์ชัน ตัวอย่างที่ผิดเช่น

```
const int ZERO = 0;
int ONE = 1;
int bit;

 .
 .
 .

switch (bit)
{
      case ZERO : printf("0\n"); break;
      case ONE  : printf("1\n"); break;
}
```

ตัวอย่างนี้ไม่ถูกต้องตามหลักไวยกรณ์เพราะตัวเลือกของ case เป็นตัวแปรมิใช่ค่าคงที่แม้ว่า ZERO จะเป็น ตัวแปรแบบ const ก็ตาม

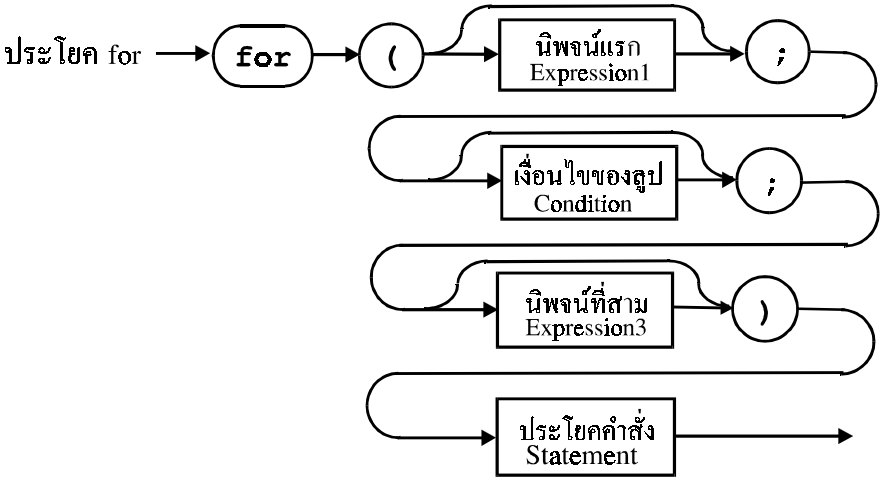
3.3 การวนลูปหรือทำขั้นตอนซ้ำ

ในภาษาซีและภาษาโปรแกรมคอมพิวเตอร์อื่นๆ เราสามารถใช้ประโยคสำหรับการกระทำขั้นตอน หรือชุดคำสั่งซ้ำ (Iteration Statement หรือ Loop) จนกว่าเงื่อนไขที่เรากำหนดไว้สำหรับวงวนนี้ ยังคงเป็น จริง หรือเราอาจจะใช้ประโยคคำสั่งอื่นๆที่ดัดแปลง หรือมีผลต่อกลไกการทำงานของวงวน ประโยคชนิดหลังนี้ เรามักจะเรียกว่า ประโยคคำสั่งกระโดด (Jump Statement) เช่น break, goto, continue และ return

3.3.1 ประโยค for

การวนลูป (Loop) แบบ for เป็นการสร้างกลไกควบคุมที่เราสามารถใช้ในการกระทำขั้นตอนซ้ำ โดยอาศัยการกำหนดนับจำนวนครั้งที่เราต้องการกระทำชุดคำสั่งซ้ำ วงวนแบบนี้จะมีตัวแปรที่ทำหน้าที่ เป็น *ตัวนับ* (Counter) ซึ่งบ่งบอกว่า มีการกระทำขั้นตอนซ้ำกี่ครั้งแล้ว เราจะต้องกำหนดค่าของตัวนับนี้ ก่อนเริ่มการวนลูป หลังจากการกระทำขั้นตอนภายในวงวนแต่ละครั้งจะมีการเปลี่ยนแปลงค่าของตัวนับนี้ การกระทำซ้ำจะเกิดขึ้นนานเท่าที่ *เงื่อนไขของลูป* (Loop Condition) ยังคงมีค่าเป็นจริง และเงื่อนไขควบคุม ของวงวนนี้จะกำหนดให้ค่าของตัวนับของลูปมีค่าอยู่ในช่วงใดช่วงหนึ่งบนเส้นจำนวน ถ้าค่าของตัวนับมี การเปลี่ยนแปลงไปจนอยู่นอกขอบเขตที่ได้กำหนดไว้ กลไกของวงวนก็จะหยุดการวนลูป เรามักจะเรียกวง วนในลักษณะนี้ว่า Counter-Controlled Loop มีลักษณะดังนี้

```
for ( expression1, condition, expression3 )
      statement
```



รูปภาพที่ 3.9 โครงสร้างของประโยคแบบ for

จากคุณสมบัติที่เราได้กล่าวไป ในการสร้างวงวนแบบ for นี้เราจะต้องทราบแน่นอนก่อนว่าจะมีการกระทำขั้นตอนซ้ำกี่ครั้ง สำหรับบางกรณี เราไม่สามารถทราบล่วงหน้าได้ว่าจะต้องทำขั้นตอนซ้ำกี่ครั้ง ดังนั้น เราจะใช้วงวนแบบ while หรือ do-while แทนที่จะใช้วงวนแบบ for ซึ่งเราจะได้ทำความรู้จักต่อไป

การทำงานของวงวนแบบ for เราสามารถอธิบายเป็นขั้นตอนได้ดังนี้

1. `expression`$_1$ (Initialization Expression) หมายถึง นิพจน์ที่ใช้ในการติดตั้งค่า เริ่มต้นของตัวนับ ดังนั้นขั้นตอนนี้จะต้องดำเนินการก่อนที่จะเริ่มวนลูป

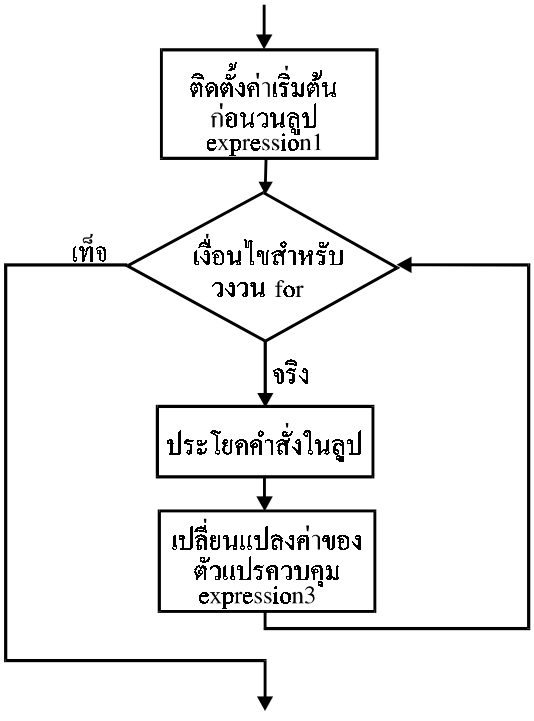
2. คำนวณค่าของนิพจน์ควบคุม `condition` ว่ามีค่าเป็นจริงหรือไม่ ถ้ามีค่าเป็นจริงก็ให้ทำขั้นตอนต่อไปนี้

2.1 ให้กระทำขั้นตอนตามที่กำหนดไว้ใน `statement` ซึ่งเป็นประโยคเดี่ยวหรือ ประโยคเชิงซ้อนก็ได้ แต่จะต้องมีเพียงประโยคเดียวเท่านั้น (ประโยคคำสั่งหลายๆประ-โยคที่อยู่ระหว่างเครื่องหมายวงเล็บปีกกาเปิดและปิดจัดว่าเป็นประโยคเชิงซ้อนประโยคเดียวหรือบล็อก)

2.2 `expression`$_3$ (Increment Expression) หมายถึง นิพจน์ที่ใช้เปลี่ยนแปลง ค่าของตัวนับ ตามปรกติแล้วเราจะเขียนนิพจน์ที่ทำให้ตัวนับมีค่ามากขึ้นทีละหนึ่งใน การวนลูปแต่ละครั้ง

2.3 ย้อนกลับไปทำขั้นตอนที่ 2 อีกครั้ง

ถ้าเงื่อนไขของลูปเป็นเท็จก็ให้หยุดการกระทำขั้นตอนซ้ำของวงวนและไปยังประโยคคำ สั่งต่อไปที่อยู่ถัดจากประโยค `for`



รูปภาพที่ 3.10 ผังงานของประโยคแบบ `for`

ในการกำหนดจำนวนครั้งของการกระทำขั้นตอนซ้ำ เราจะใช้วิธีการเหมือนการนับเลขโดยใช้ตัวนับ ซึ่งจะ ต้องมีการกำหนดค่าเริ่มต้นก่อน จากนั้นก็เปลี่ยนแปลงค่าของตัวนับให้เพิ่มขึ้นหรือลดลงทีละหนึ่ง (หรืออาจ จะเปลี่ยนแปลงมากกว่าหนึ่งในแต่ละครั้งก็ได้) และเราจะต้องกำหนดค่าที่เป็นขอบเขตของการนับด้วย เช่น เริ่มนับตั้งแต่หนึ่งไปจนถึงหนึ่งร้อย เป็นต้น

1) การนับที่ตัวนับมีค่าเพิ่มขึ้น

ตัวนับ (counter) : `initial_value` ≤ `counter` ≤ `limit_value`

2) การนับที่ตัวนับมีค่าลดลง

ตัวนับ (counter) : `initial_value` ≥ `counter` ≥ `limit_value`

เพื่อที่จะให้เห็นวิธีการกำหนดกลไลควบคุมและการสร้างลูปแบบ `for` เราจะเขียนแบบการใช้งานในลักษณะง่ายๆดังนี้ และแบ่งออกได้เป็นสองกรณี โดยเรากำหนดให้ `counter` เป็นตัวแปรแบบ `int` ในขณะที่ `initail_value` และ `limit_value` เป็นค่าคงที่หรือตัวแปรแบบ `int`

1) **การนับที่ตัวนับมีค่าเพิ่มขึ้นทีละหนึ่ง**

```
for ( counter = initial_value ;
      counter <= limit_value  ;
     counter++ )
          statement
```

จากโครงสร้างของประโยค `for` แบบนี้ เราจะเห็นได้ว่าค่าของตัวนับจะมีมากขึ้นเรื่อยๆหลังจากที่มีการวนลูปแต่ละครั้ง เมื่อใดที่ตัวนับมีค่าเกินขอบเขตที่เรากำหนดเอาไว้ตามเงื่อนไขของลูป โปรแกรมก็จะออกจาก วงวนนี้ เราคำนวณได้ว่า

จำนวนครั้งของการวนลูป = `limit_value - initial_value + 1`

2) **การนับที่ตัวนับมีค่าลดลงทีละหนึ่ง**

```
for ( counter = initial_value ;
      counter >= limit_value  ;
       counter-- )
          statement
```

วิธีการทำงานของวงวนแบบนี้ก็ไม่แตกต่างจากโครงสร้างแบบแรก เพียงแต่แทนที่จะเพิ่มค่าของตัวนับก็ลดค่าลงทีละหนึ่งหลังจากที่กระทำคำสั่งภายใน statement และเราคำนวณได้ว่า

จำนวนครั้งของการวนลูป = initial_value - limit_value + 1

ตัวอย่างแรกสำหรับการใช้ประโยค for จะเป็นการพิมพ์ข้อความและตัวเลขตั้งแต่ 1 ถึง 5 ออกทางจอภาพ โดยเราจะใช้ตัวแปร i แบบ int เป็นตัวนับที่ใช้ควบคุมการวนลูป ร่วมกับเงื่อนไขของวงวน

```
#include <stdio.h>

int main()
{
      int i;

      for (i=1; i <= 5 ; ++i)
      {
            printf("Loop i = %d\n", i);
      }
      printf("\nLoop terminates at i = %d\n", i);

      return 0;
}
```

โปรแกรมนี้จะพิมพ์ข้อความออกทางจอภาพดังนี้

```
Loop i = 1
Loop i = 2
Loop i = 3
Loop i = 4
Loop i = 5

Loop terminates at i = 6
```

จะเห็นได้ว่าโปรแกรมพิมพ์ข้อความตั้งแต่ 1 ถึง 5 เท่านั้นและออกจากวงวนเมื่อ i มีค่าเท่ากับ 6

เราอาจจะดัดแปลงโครงสร้างและขั้นตอนการทำงานของประโยค for ใหม่โดยเราย้ายคำสั่งแรกที่ใช้ติดตั้งค่าของตัวนับ i ก่อนเริ่มการวนลูปไปไว้ภายนอกโครงสร้างของประโยค นอกจากนี้เราก็ย้ายคำสั่งที่ใช้เพิ่มค่าของตัวนับหลังจากที่มีการวนลูปแต่ละครั้งไปไว้ในส่วนของประโยคคำสั่งภายในบล็อกของประโยค for ตามแบบข้างล่างนี้

```
i = 1;
for (  ; i <= 5 ;  )
{
      printf("Loop i = %d\n", i++);
}
printf("\nLoop terminates at i = %d\n", i);
```

ให้ผู้อ่านลองเปลี่ยนแปลงแก้ไขโปรแกรมตัวอย่างใหม่ แล้วคอมไพล์และรันโปรแกรมเพื่อที่จะตรวจสอบว่า โปรแกรมโค้ดที่ได้แก้ไขตามรูปแบบข้างบนให้ผลที่แสดงออกทางจอภาพเหมือนกับโปรแกรมโค้ดเดิมก่อนที่เราจะเปลี่ยนแปลงแก้ไข

สำหรับตัวอย่างการใช้ประโยค for ตัวอย่างต่อไป เราจะย้อนกลับไปพิจารณาปัญหาในบทที่หนึ่ง คือการ หาค่าของผลรวมของเลขจำนวนนับตั้งแต่ 1 ถึง 100 และเราจะแก้ปัญหาดังกล่าวโดยการเขียนประโยค for ตามโปรแกรมข้างล่างนี้

```
#include <stdio.h>

int main ()
{
      int  i;                 /* counter variable */
      int  sum = 0;
      const int N = 100;      /* limit value */

      for (i = 1; i <= N; i++)
      {
            sum += i;
      }

      printf ("1 + 2 + ... + %d = %d\n", N, sum);
      return 0;
}
```

ผลของโปรแกรมก็คือ

```
1 + 2 + ... + 100 = 5050
```

แต่ถ้าเราใช้วิธีการที่ตัวนับมีค่าน้อยลง เราก็เขียนประโยค for ใหม่ได้ดังนี้ ซึ่งย่อมให้ผลท้ายสุดเหมือนกัน

```
      for (i = N; i >= 1; i--)
      {
            sum += i;
      }
```

ข้อควรระวัง

เราจะเห็นได้ว่า ตัวแปรที่ทำหน้าที่เป็นตัวนับของลูปนั้นจะใช้ในกลไกควบคุม และค่าของตัวนับจะ เปลี่ยนแปลงไปโดยอัตโนมัติตามที่เรากำหนดไว้ในนิพจน์ภายในประโยค ดังนั้นเราไม่ควรจะเขียนคำสั่ง ใดๆใน statement ที่ทำให้ค่าของตัวนับนี้เปลี่ยนแปลงไปซึ่งจะทำให้กลไกการวนลูปไม่ถูกต้องตามเราที่ต้องการ เราลองมาพิจารณาดูตัวอย่างต่อไปนี้

```
#include <stdio.h>

int main ()
{
      int  i;                 /* counter variable */
      const int N = 10;       /* limit value */

      for (i = 1; i <= N; i++)
      {
            printf ("i = %d\n", i);
            i = N+1;
      }
      printf ("---------------\n", i);
      return 0;
}
```

ผลของโปรแกรมบนจอภาพ

```
i = 1
---------------
```

เราจะเห็นได้ว่า แทนที่จะพิมพ์ข้อความทั้งหมดสิบครั้งโดยที่ตัวแปร `i` มีค่าตั้งแต่ 1 ถึง 10 ตามลำดับ โปรแกรมก็พิมพ์ข้อความที่เป็นผลจากฟังก์ชัน `printf()` ภายในลูปเพียงครั้งเดียว ก็เพราะว่าเราได้เขียนประโยคคำสั่ง

```
i = N+1;
```

ที่มีผลต่อกลไกควบคุมของลูป ทำให้ `i` มีค่ามากกว่า `N` ดังนั้นเมื่อโปรแกรมเริ่มวนลูป เงื่อนไขของลูปก็จะเป็นเท็จ ทำให้ไม่มีการวนลูปและไปยังคำสั่งที่อยู่ถัดไป นอกจากนี้เราก็ไม่ควรเปลี่ยนแปลงค่าที่ใช้เป็นขอบ เขตของการนับเมื่อมีการวนลูปในแต่ละครั้ง

ตัวอย่างต่อไปนี้แสดงให้เห็นผลที่เกิดขึ้นเนื่องจากขั้นตอนภายในวงวนที่เปลี่ยนแปลงค่าของตัวแปร `N` ที่เราใช้เป็นค่าจำกัดในการนับ

```
#include <stdio.h>

int main ()
{
      int i;                  /* counter variable */
      int N = 5;              /* limit value */

      for (i = 1; i <= N; i++)
      {
            printf ("i = %d, N = %d\n", i, N);
            N++;
      }
      return 0;
}
```

ผลจากการรันโปรแกรม

```
i = 1, N = 5
i = 2, N = 6
i = 3, N = 7
i = 4, N = 8
i = 5, N = 9
i = 6, N = 10
  ...
  ...
```

ซึ่งโปรแกรมจะพิมพ์ข้อความออกทางจอภาพไปเรื่อยๆ แทนที่จะพิมพ์ข้อความแค่ห้าครั้งคือสำหรับตัวเลข 1 ถึง 5 เหตุที่เป็นเช่นนี้ก็เพราะประโยคคำสั่ง

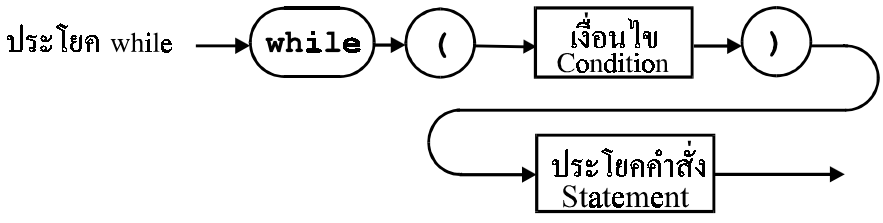
```
N++;
```

เปลี่ยนแปลงค่าของตัวแปร N ในแต่ละครั้ง และเราจะเห็นได้ว่าค่าของ N จะเพิ่มขึ้นทีละหนึ่งทุกครั้งที่มีการ วนลูป และค่าของ N จะมากกว่าค่าของ i เสมอ (จนกว่าค่าของ N จะเกิน 32767 ซึ่งต่อจากนั้นจะกลายเป็น ค่าลบ -32768 ทำให้เงื่อนไขเป็นเท็จ และหยุดการวนลูป)

3.3.2 ประโยค while

เราใช้โครงสร้างประโยคคำสั่ง while ในการกระทำตามประโยคคำสั่งหรือลำดับของประโยคคำสั่งซ้ำหลายๆครั้ง ตราบเท่าที่เงื่อนไขที่เรากำหนดยังคงมีค่าทางตรรกศาสตร์เป็นจริง ประโยคคำสั่ง while มีรูปแบบดังต่อไปนี้

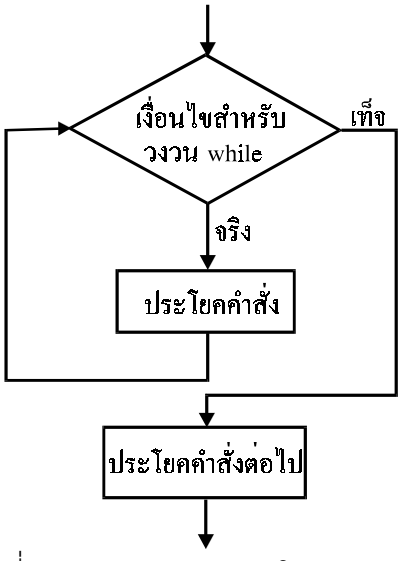
```
while (condition)
      statement
```



รูปภาพที่ 3.11 โครงสร้างของประโยคแบบ while

condition หมายถึง การกำหนดเงื่อนไขของการกระทำซ้ำ ซึ่งเป็นนิพจน์ใดๆที่ให้ค่าคงที่เป็นตัวเลขหรือค่าคงที่ในทางตรรกศาสตร์ เราจะอ่านได้ว่า “กระทำตามคำสั่งในประโยคที่กำหนดโดย statement นานเท่าที่เงื่อนไขยังคงเป็นจริง และออกจากวงวน (Loop) นี้เมื่อเงื่อนไขเป็นเท็จ”

`statement` เป็นประโยคคำสั่ง (และมีเพียงประโยคเดียวเท่านั้นที่อยู่ถัดจากเงื่อนไขของวงวนภายในโครงสร้างของประโยค `while`) อาจจะเป็นประโยคคำสั่งเดี่ยวหรือประโยคคำสั่งเชิงซ้อนก็ได้



รูปภาพที่ 3.12 ผังงานของประโยคแบบ `while`

ลำดับการทำงานภายในโครงสร้างของประโยค `while`

1. เงื่อนไขสำหรับวงวน (Loop Condition) แบบ `while` นี้จะถูกคำนวณก่อน
2. ถ้าเงื่อนไขในขั้นตอนที่หนึ่งเป็นจริง ให้ดำเนินการดังต่อไปนี้
    2.1 กระทำคำสั่ง (Statement) ที่อยู่ในโครงสร้างของประโยค `while`
    2.2 ย้อนกลับไปทำขั้นตอนที่หนึ่งซ้ำ

แต่ถ้าเงื่อนไขเป็นเท็จ ก็ให้ออกจากวงวนไปยังคำสั่งที่อยู่ถัดจากประโยค `while` ต่อไป

โปรดสังเกตว่า เมื่อเข้าสู่โครงสร้างของ `while` เราจะต้องตรวจความถูกต้อง (จริงหรือเท็จ) ของเงื่อนไขก่อนที่จะกระทำคำสั่งในโครงสร้างของ `while` ดังนั้นถ้าเงื่อนไขเป็นเท็จตั้งแต่ต้น ก็จะไม่มีการคำสั่งของโครงสร้างแบบนี้ และไปยังคำสั่งที่อยู่ถัดจากประโยค `while` ต่อไป โครงสร้างของประโยค `while` เรามัก จะเรียกว่า Pretest Loop หรือ Test-At-The-Top Loop หมายถึงตรวจสอบเงื่อนไขตอนต้นก่อนการกระทำ ตัวอย่างเช่น

```
i = 0;
while (i)
{
      printf ("While Loop\n");
}
```

เนื่องจากว่า เรากำหนดให้ i เป็นเงื่อนไขของประโยค while แต่ตัวแปร i มีค่าเท่ากับศูนย์ ดังนั้นเงื่อนไขจึงมีค่าเป็นเท็จ ทำให้ประโยคคำสั่งในโครงสร้างของ while ไม่มีผลแต่อย่างใดในการทำงาน

เราลองมาเขียนโปรแกรมที่ใช้โครงสร้างประโยคแบบ while โดยพิจารณาตัวอย่างในบทที่ 1 คือ การหาผลรวมของเลขจำนวนนับตั้งแต่ 1 ถึง 100 แต่สำหรับตัวอย่างต่อไปนี้ เราจะหาผลรวมของเลขจำนวน นับแค่สิบตัวแรกคือตั้งแต่ หนึ่งถึงสิบเท่านั้น เรากำหนดให้ตัวแปร N แบบ int มีค่าเท่ากับ 10 และตัว แปร sum ใช้เก็บค่าของผลรวมในแต่ละขั้น โดยมีตัวแปร i บ่งบอกว่าเรานับถึงเลขใดแล้วและมีค่าเริ่มต้นเป็นหนึ่ง การวนลูปนี้ ก็คือการกระทำขั้นตอนซ้ำโดยการวนลูปในแต่ละครั้งเราจะเพิ่มค่าของตัวแปรจากค่า เดิมขึ้นอีกหนึ่งและการวนซ้ำนี้จะหยุดเมื่อเงื่อนไขเป็นเท็จ และเงื่อนไขก็คือ ตัวแปร i จะต้องมีค่าน้อยกว่า หรือเท่ากับ N

```
#include <stdio.h>

int main()
{
      const int N = 10;
      int i = 1, sum = 0;

      printf("\t   i  \t   sum\n");
      while (i <= N)
      {
            sum += i;
            printf("\t %3d \t %4d\n", i, sum);
            i++;
      }
      printf ("----------------------\n");
      printf ("1 + 2 + ... + %d = %d\n", N, sum);
      return 0;
}
```

ผลของโปรแกรมบนจอภาพก็คือ

```
            i       sum
            1        1
            2        3
            3        6
            4       10
            5       15
            6       21
            7       28
            8       36
            9       45
           10       55
----------------------
1 + 2 + ... + 10 = 55
```

ในตัวอย่างเราได้กำหนดให้ตัวแปร i มีการเพิ่มค่าของตัวแปรขึ้นทีละหนึ่ง โดยมีค่าเริ่มต้นเท่ากับ 1 ดังนั้น เมื่อมีการวนลูปในโครงสร้างของ while ค่าของตัวแปร i จึงมีค่ามากขึ้นตามลำดับ นอกจากนี้เรายังได้กำหนดเงื่อนไขของการกระทำไว้ว่า i จะต้องมีค่าไม่เกิน N คือ 10 ดังนั้นเราจึงแน่ใจว่า หลังจากที่มีการกระทำซ้ำตามจำนวนครั้งที่เหมาะสมแล้วก็จะมีการหยุดการกระทำซ้ำ เพราะเรารู้ว่า ค่าของตัวแปร i จะมี ค่ามากกว่า N หลังจากที่วนลูปไปแล้วหลายครั้ง เปรียบได้กับว่านักวิ่งระยะ 1600 ม. เมื่อวนรอบสนามที่มีความยาวรอบเท่า 400 ม. ครบ 4 รอบแล้ว เมื่อขึ้นรอบที่ 5 นักวิ่งคนนี้จึงหยุดวิ่ง

นอกจากนี้เราสามารถกำหนดเงื่อนไขและวิธีการนับได้อีกลักษณะหนึ่งก็คือ แทนที่เราจะกำหนดให้ตัวแปร i มีค่าเริ่มต้นเท่ากับ 1 แล้วก็เพิ่มค่ามากขึ้นเรื่อยๆ เราก็กำหนดใหม่ให้ i มีค่าเริ่มต้นเป็น N แล้วก็ ลดค่าของ i ลงทีละหนึ่ง นอกจากนี้เรายังต้องแก้ไขเงื่อนไขใหม่เป็นดังนี้ ทำขั้นตอนซ้ำเมื่อ i ยังคงมีค่า มากกว่าศูนย์ เราเขียนโปรแกรมใหม่ได้ดังนี้

```
#include <stdio.h>

int main()
{
      const int N = 10;
      int i = N, sum = 0;

      printf ("\t   i  \t   sum\n");
      while (i > 0)
      {
            sum += i;
            printf("\t %3d \t %4d\n", i, sum);
            i--;
      }

      printf ("----------------------\n");
      printf ("%d + %d + ... + 1 = %d\n", N, N-1, sum);
      return 0;
}
```

ผลของโปรแกรมบนจอภาพคือ

```
            i       sum
           10       10
            9       19
            8       27
            7       34
            6       40
            5       45
            4       49
            3       52
            2       54
            1       55
----------------------
10 + 9 + ... + 1 = 55
```

ตัวอย่างต่อไป เราจะใช้ฟังก์ชันมาตราฐาน `getchar()` ในการอ่านข้อมูลที่เป็นตัวอักขระ ซึ่งก็คือการอ่านข้อมูลจากการกดแป้นพิมพ์แต่ละครั้ง ฟังก์ชันนี้จะอ่านค่าของข้อมูลที่ตรงกับคีย์ต่างๆที่เรากด และเราจะสร้างวงวนแบบ `while` ที่อ่านข้อมูลจากแป้นพิมพ์จนกว่าผู้ใช้โปรแกรมจะกดแป้นตัวอักษร `'Y'` และแจ้งว่า มีการป้อนข้อมูลจากผู้ใช้เป็นตัวอักขระทั้งหมดกี่ตัว ถ้าเรากดแป้น ENTER ฟังก์ชันจะให้ค่าเท่ากับ 10 ซึ่งเป็นผลมาจากการขึ้นบรรทัดใหม่ แต่เราจะไม่นับตัวอักขระตัวนี้ ดังนั้นเราจะไม่เพิ่มค่าของ `counter` ถ้า `ch` มีค่าเท่ากับ 10 ซึ่งเราใช้ตัวแปร `counter` เป็นตัวนับว่าเราได้กดตัวขระจากแป้นพิมพ์ไปกี่ตัวแล้ว

```
#include <stdio.h>

int main()
{
      int ch;
      int counter = 0;
      while ((ch = getchar()) != 'Y')
      {
            if (ch != 10) counter++;
      }
      printf("%d characters\n", counter);
      return 0;
}
```

จะเห็นได้ว่า ประโยคแบบ `while` นั้นใช้ได้กับการทำขั้นตอนซ้ำที่เราทราบจำนวนครั้งของการวนลูปและในกรณีที่เราไม่ทราบจำนวนครั้งล่วงหน้า แต่อาศัยการรอคอยเหตุการณ์ที่ทำให้เงื่อนไขของการวนลูปเป็นเท็จ ประโยคแบบ `for` เราสามารถเขียนโดยใช้ประโยค `while` แทนได้ดังนี้

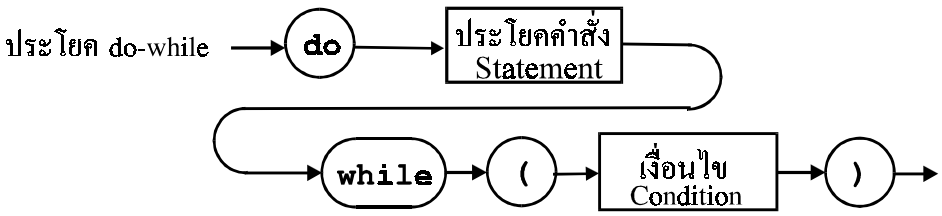
```
for ( expression1, condition, expression3 )
      statement


expression1
while (condition)
{
      statement
      expression3
}
```

3.2.3 ประโยค `do-while`

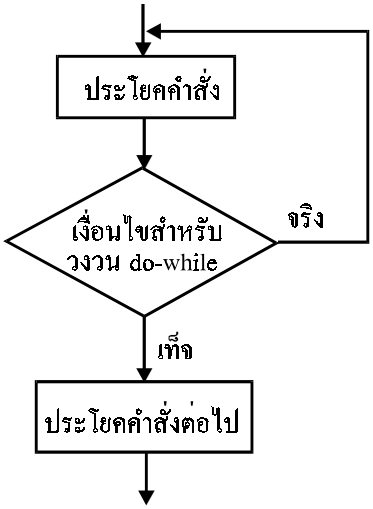
ประโยค `do-while` นั้นใช้สำหรับจุดประสงค์เดียวกับประโยค `while` แต่มีโครงสร้างของประโยคที่แตกต่างกันดังนี้

```
do
      statement
while (condition);
```



รูปภาพที่ 3.13 โครงสร้างของประโยคแบบ do - while

โดยที่ประโยคคำสั่ง statement (ประโยคเดี่ยวหรือเชิงซ้อนก็ได้แต่ต้องมีเพียงประโยคเดียวเท่านั้น) จะต้องถูกดำเนินการก่อนที่จะมีการตรวจสอบค่าความเป็นจริงของเงื่อนไข condition ในขณะที่ประโยค while จะตรวจสอบเงื่อนไขก่อนแล้วจึงกระทำคำสั่ง ดังนั้นสำหรับประโยค while-do ไม่ว่าเงื่อนไขจะมีค่าทางตรรกศาสตร์เป็นเท่าใดจะต้องมีการกระทำคำสั่ง statement อย่างน้อยหนึ่งครั้ง เราเรียกโครงสร้างวงวนแบบ do-while นี้ว่า Posttest Loop หรือ Test-At-The-Bottom Loop



รูปภาพที่ 3.14 ผังงานของประโยคแบบ do - while

ลำดับการทำงานภายในโครงสร้างของประโยค do-while

1. กระทำตามคำสั่งที่ระบุไว้ใน statement
2. คำนวณหาค่าของนิพจน์เงื่อนไข

3. ถ้าเงื่อนไขมีค่าเป็นจริงก็ให้ย้อนกลับไปทำขั้นตอนที่หนึ่งซ้ำ
แต่ถ้าเงื่อนไขเป็นเท็จก็ให้ไปทำคำสั่งในประโยคที่อยู่ถัดไป

ตัวอย่างการใช้งานประโยค do-while

```
#include <stdio.h>

int main()
{
      int ch, vowels = 0, lines = 1;
      const int CTRL_D =  4;
      const int ENTER  = 13;

      printf("\n>");
      do
      {
            ch = getch();
            if ((ch == ENTER) || (ch == '\n'))
            {
                ch = '\n';
                printf("\n>");
            }
            else
              if (ch >= 32 && ch < 128)
                printf ("%c", ch);

            switch (ch)
            {
              case 'A'  :
              case 'E'  :
              case 'I'  :
              case 'O'  :
              case 'U'  :
              case 'a'  :
              case 'e'  :
              case 'i'  :
              case 'o'  :
              case 'u'  :
                   vowels++; break;
              case '\n' :
                   lines++;  break;
              default   :
                   ;  /* nothing */

            }
      }
      while (ch != CTRL_D);

      printf("\nNumbers of vowels = %3d\n", vowels);
      printf("Numbers of lines  = %3d\n", lines);

      return 0;
}
```

ผลของโปรแกรมตัวอย่าง เมื่อพิมพ์คำว่า Hello World!

```
>Hello World!
>Ctrl+D
Numbers of vowels =  3
Numbers of lines  =  2
```

Ctrl+D หมายถึงการกดปุ่ม `Ctrl` พร้อมกับคีย์ `D` ของแป้นพิมพ์

3.2.4 การวนลูปแบบไม่รู้จบ

การวนลูปแบบไม่รู้จบ (Infinite Loop) นั้นหมายถึง การสร้างวงวนแบบต่างๆ ไม่ว่าจะเป็น `for` `while` หรือ `do-while` ที่มีการกระทำคำสั่งภายในวงวนแบบไม่รู้จบ อันเป็นผลมาจากการที่เงื่อนไขของวงวนเป็นจริงเสมอ การสร้างลูปในลักษณะนี้แบ่งออกได้เป็นสองกรณี กรณีแรกคือเราตั้งใจที่จะกำหนดใช้วงวนแบบไม่รู้จบนี้ และกรณีที่สองคือเราไม่ต้องการให้เกิดวงวนในลักษณะนี้ แต่เป็นผลมาจากการเขียน วิธีการทำงานของโปรแกรมที่ไม่ถูกต้อง (Logical Error) ซึ่งเกิดขึ้นได้เสมอถ้าผู้ที่เขียนโปรแกรมไม่ระมัดระวัง สำหรับการสร้างวงวนแบบไม่รู้จบ เราสามารถเขียนโครงสร้างของประโยคได้ดังต่อไปนี้

```
for (; ; )
      statement

while (1)
      statement

do
      statement
while (1);
```

ส่วนการสร้างลูปที่ทำงานไม่รู้จบซึ่งเกิดขึ้นโดยที่ผู้เขียนโปรแกรมมิได้ตั้งใจนั้นมีอยู่หลายๆลักษณะ เราลอง พิจารณาความผิดพลาดที่พบเห็นได้เป็นประจำ

ตัวอย่างที่หนึ่ง

```
int counter = 1;
while (counter < 10)
{
      printf ("%d\n", counter);
      counter--;
}
```

แทนที่ตัวนับจะมีค่าเพิ่มขึ้นแต่กลับมีค่าน้อยลงทีละหนึ่งซึ่งเป็นผลมาจากโอเปอร์เรเตอร์ `--` ดังนั้นค่าของตัว นับจึงน้อยกว่า 10 เสมอ (จนกว่าจะเกิด Underflow ซึ่งตัวแปรแบบ `int` ที่มีค่าเท่ากับ -

32768 เมื่อลด ค่าลงอีกหนึ่งจะมีค่าเป็น 32767 ทำให้เงื่อนไขเป็นเท็จ) และถ้าจะแก้ไขให้ถูกต้องตามหลักการเพื่อมิให้เกิด การวนซ้ำแบบไม่รู้จบ โดยที่เราต้องการจะให้ตัวนับมีค่ามากขึ้นทีละหนึ่ง มิใช่ลดลง ดังนั้นจะต้องใช้ โอเปอร์เรเตอร์ ++ ถึงจะถูกต้อง

ตัวอย่างที่สอง

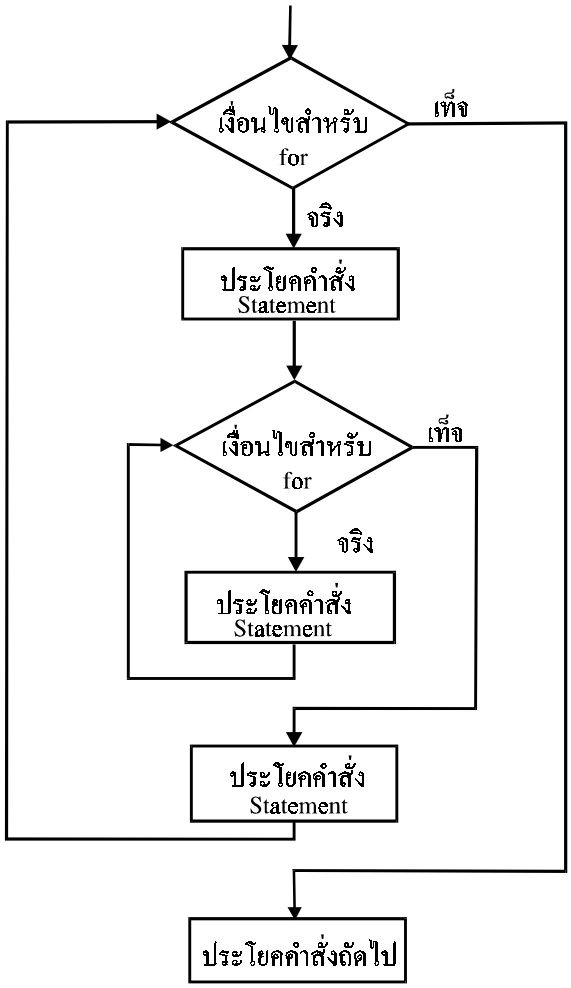
สมมุติว่า เรากำหนดเงื่อนไขของลูปไว้ว่า ถ้าค่าของตัวแปร `i` ซึ่งทำหน้าที่เป็นตัวนับจะต้องมีค่าไม่ เท่ากับ 0 โดยที่เราแน่ใจว่า ค่าของตัวนับนี้จะลดลงทีละหนึ่งและเมื่อมีค่าเท่ากับ 0 เงื่อนไขก็จะเป็นเท็จและหยุดการวนลูป

```
int i = 100;
do
{
      printf ("%d\n", i);
      i--;
}
while (!(i = 0))
```

แต่ถ้าเราเขียนโอเปอร์เรเตอร์เชิงเปรียบเทียบ == แทนที่จะเป็น = ตามตัวอย่างข้างบนซึ่งอาจจะมิได้ตั้งใจ ดังนั้นแทนที่จะเป็นการเปรียบเทียบก็กลายเป็นการกำหนดค่าของตัวนับและส่งผลให้นิพจน์ `!(i = 0)` ซึ่งเป็นนิพจน์เงื่อนไขของลูปมีค่าเป็นจริงเสมอ ทำให้เกิดการทำขั้นตอนซ้ำแบบไม่รู้จบ

3.3.5 การสร้างวงวนซ้อนหลายชั้น

ในบางกรณีเราจำเป็นต้องสร้างวงวนหลายๆวงที่ทำงานซ้อนกันตามลำดับ เมื่อโปรแกรมทำงานก็จะ เริ่มจากวงวนที่อยู่นอกสุดแล้วเข้าสู่วงวนที่อยู่ลึกลงไปตามลำดับจนถึงวงวนที่อยู่ชั้นในสุด แต่โปรแกรมจะต้องดำเนินการขั้นตอนของวงวนในให้เสร็จก่อนแล้วจึงย้อนกลับขึ้นมาตามลำดับจนถึงวงวนนอกสุด ตามปรกติแล้วการกระทำซ้ำอันเป็นผลมาจากวงวนที่ซ้อนกันนี้จะหยุดทำงาน ก็ต่อเมื่อวงวนชั้นนอกสุดได้ทำหน้าที่จนครบแล้ว โปรดดู รูปผังการทำงานประกอบซึ่งแสดงให้เห็นการทำงานของวงวนแบบ `for` สองวงที่ ซ้อนกัน



รูปภาพที่ 3.15 ผังงานของตัวอย่างการสร้างวงซ้อน

สำหรับตัวอย่างการใช้วงวนตามแผนการทำงาน เราลองมาพิจารณาปัญหาต่อไปนี้ สมมุติว่า เราต้องการจะนับเลขเริ่มตั้งแต่หนึ่งไปเรื่อยจนถึง N และกำหนดให้เป็นตัวแปรนี้มีค่าเท่ากับ 9 ในการนับแต่ละครั้งก็ให้พิมพ์ลำดับของเลขที่กำลังนับในขณะนั้นโดยให้มีจำนวนเท่ากับค่าของตัวเลขในขณะนั้น ซึ่งมีลักษณะดังนี้

```
1
22
333
4444
55555
....
```

โดยพิมพ์ไปเรื่อยๆจนกว่าจะนับถึง N มีที่ค่าเท่ากับ 9 และกำหนดให้พิมพ์ตัวอักขระที่เป็นเลขโดดนี้โดยใช้ ฟังก์ชันมาตราฐาน `printf()` ดังนั้นวิธีการง่ายๆที่จะเขียนโปรแกรมที่แก้ปัญหานี้คือการใช้วงวนซ้อนกันสองครั้งแบ่งออกเป็นวงวนนอก และวงวนนอก โดยที่เราใช้วงวนนอกสำหรับการนับตัวเลขจากหนึ่งถึงเก้า และวงวนในสำหรับการพิมพ์ตัวเลขโดดออกทางจอภาพให้มีจำนวนของเลขโดดเท่ากับค่าของเลขจำนวนนับในตอนนั้น เราจะสร้างวงวนแบบ `for` เพราะเหมาะสมกับ

ปัญหานี้เพราะเราทราบจำนวนครั้งของขั้นตอนที่จะต้องทำซ้ำ และกำหนดให้ `i` และ `j` เป็นตัวแปรแบบ `int` ที่ทำหน้าที่เป็นตัวนับสำหรับวงวนนอก และวงวนในตามลำดับ (โปรดสังเกตว่า สำหรับวงวนแต่ละวงเราจะต้องใช้ตัวแปรที่แตกต่างกันออกไปสำ-หรับใช้เป็นตัวนับของวงวนดังกล่าว)

```
#include <stdio.h>

int main()
{
      int i, j;
      int N = 9, function_calls = 0;

      for (i = 1; i <= N; i++)
      {
            for (j = 1; j <= i; j++)
            {
                  printf ("%d", i);
                  function_calls++;
            }
            printf ("\n", i);
      }
      printf ("Number of function calls : %d\n",
               function_calls);
      return 0;
}
```

เราสามารถคำนวณได้ว่า มีการเรียกใช้ฟังก์ชัน `printf()` ทั้งหมดกี่ครั้งภายในวงวน (นับเฉพาะการเรียกใช้ฟังก์ชันที่พิมพ์ตัวเลขโดดออกทางจอภาพและไม่รวมฟังก์ชันที่ทำหน้าที่ขึ้นบรรทัดใหม่ของข้อความ) ซึ่งคำนวณได้ดังนี้

จำนวนครั้งของการเรียกใช้ฟังก์ชัน = 1 + 2 + 3 + ... + 9 = 45

ตัวอย่างต่อไป เราลองมาเขียนโปรแกรมที่ใช้คำนวณค่าของนิพจน์ทางคณิตศาสตร์ต่อไปนี้

$$S = \sum_{i=1}^{N} \sum_{j=i}^{M} a^i b^{i-j}$$

เช่น เรากำหนดค่าต่างๆไว้ดังนี้

ตัวแปร	แบบข้อมูล	คำอธิบายเพิ่มเติม
`i`	`int`	ตัวนับสำหรับวงวนชั้นนอก (วงวนแรก)
`j`	`int`	ตัวนับสำหรับวงวนชั้นใน (วงวนที่สอง)
`N`	`int`	กำหนดให้มีค่าเท่ากับ `10`
`M`	`int`	กำหนดให้มีค่าเท่ากับ `10`

a	double	กำหนดให้มีค่าเท่ากับ 0.1
b	double	กำหนดให้มีค่าเท่ากับ 2.0
S	double	เป็นค่าของนิพจน์ที่เราจะต้องคำนวณ

```
#include <stdio.h>
#include <math.h>

int main()
{
      int i, j;
      int N = 10, M = 10;
      double S = 0.0, a = 0.1, b = 2.0;

      for (i = 1; i <= N; i++)
      {
            for (j = i; j <= M; j++)
            {
                  S += pow(a,i) * pow(b,i-j);
            }
      }
      printf ("S = %10.9lf\n", S);

      return 0;
}
```

ในตัวอย่างนี้ เราใช้วงวนแบบ for ซ้อนกันสองวง และเราใช้ฟังก์ชันมาตราฐาน pow() ของ math.h ใน การคำนวณค่าของเลขยกกำลัง

3.4 การใช้ประโยคคำสั่ง goto

เราใช้คำสั่ง goto ในการบังคับให้โปรแกรมไปยังขั้นตอนใดๆที่เราต้องการภายในฟังก์ชัน โดยเราต้องกำหนดก่อนว่า เราต้องการจะให้โปรแกรมไปทำงานในตำแหน่งใดต่อไป ซึ่งมีรูปแบบการใช้ดังนี้

```
goto  label;
```

เมื่อโปรแกรมทำงานมาถึงคำสั่งดังกล่าว ก็จะข้ามไปยังขั้นตอนที่เราเขียนกำกับไว้ด้วยตัวระบุ *label* ตาม ด้วยเครื่องหมายจุดคู่ :

```
label :
```

ซึ่งจะว่างไว้ตำแหน่งใดก็ได้ในฟังก์ชันใดฟังก์ชันหนึ่ง (เราไม่สามารถบังคับให้โปรแกรมกระโดดข้ามไปมาระหว่างฟังก์ชันได้โดยใช้คำสั่ง goto) ตัวอย่างการใช้งาน เช่น เราสร้างวงวนโดยใช้ประโยคคำสั่ง goto

```
#include <stdio.h>

int main()
{
      i   = 1;
      sum = 0;

      loop :
            sum += i;
            i++;
            if (i <= 100) goto loop;

      printf("Sum = %d", sum);
      return 0;
}
```

เมื่อตัวแปร i ซึ่งทำหน้าที่เป็นตัวนับ ยังคงมีค่าน้อยกว่าหรือเท่ากับ 100 โปรแกรมก็จะย้อนกลับไปยังขั้น ตอนที่อยู่ในตำแหน่งที่เราเขียนคำว่า loop: ไว้ในโปรแกรมโค้ดของเรา เมื่อค่าของตัวนับมีค่ามากขึ้นจนเกิน 100 เงื่อนของประโยค if จึงเป็นเท็จ ทำให้ไม่มีการใช้คำสั่ง goto ดังนั้นจึงออกจากการวนลูป

ตามปรกติแล้ว เราจะไม่นิยมใช้คำสั่ง goto เพราะการใช้คำสั่งนี้มากเกินไปในโปรแกรมโค้ดทำให้ยากต่อการทำความเข้าใจ และทำให้การทำงานของโปรแกรมมีลักษณะกระโดดไปกระโดดมาอันเป็นผลโดยตรงจากคุณสมบัติของคำสั่ง goto

3.5 การใช้ประโยคคำสั่ง break และ continue

สำหรับการทำงานโดยใช้วงวนแต่ละแบบนั้น ตามปรกติแล้วโปรแกรมจะหยุดการทำขั้นตอนซ้ำก็ต่อเมื่อเงื่อนไขของลูปเป็นเท็จเท่านั้น แต่ในบางครั้งเราต้องการที่จะหยุดการทำงานของวงวนนี้ในเวลาใดก็ได้เมื่อเราต้องการ โดยไม่จำเป็นต้องคำนึงถึงเงื่อนไขของวงวนว่าจะมีค่าเป็นเท่าไหรในขณะนั้นหรือเราอาจจะกล่าวได้ว่า เราต้องการจะหยุดการทำงานของวงวนก่อนจะถึงเวลาที่เงื่อนไขของวงวนนี้จะมีค่าเป็นเท็จ ในภาษาซีเราจะใช้คำสั่ง break ในการบังคับให้โปรแกรมออกจากวงวนที่มีประโยคคำสั่งนี้ซึ่งโปรแกรมกำลังทำงานอยู่ในวงวนขณะนั้น นอกจาก

นี้เรายังใช้คำสั่ง break ในโครงสร้างของประโยค swtich-case ตามที่เราได้เรียนรู้ไปแล้ว แต่ถ้าเราต้องการให้โปรแกรมข้ามขั้นตอนบางส่วนภายในวงวนและไปยังจุดเริ่มต้นของวงวนนี้ ซึ่งมิใช่หยุดการทำงานของวงวนเหมือนเวลาเราใช้คำสั่ง break เราก็สามารถใช้ประโยค continue ได้ การใช้ประโยค break และ continue นี้เปิดโอกาสให้เราสามารถดัดแปลงกลไลของวงวนได้ ตามปรกติแล้วเรามักจะใช้ประโยคทั้งสองนี้รวมกับโครงสร้างเงื่อนไขสำหรับการเลือกกระทำขั้นตอน เช่น if-else หรือ switch เป็นต้น ที่อยู่ภายในวงวนใดๆ โดยมีการกำหนดเงื่อนไขซึ่งเป็นตัวบ่งบอกว่า เมื่อไหรเราจะใช้ประโยคคำสั่ง break หรือ continue เพื่อหยุดการทำงานของวงวน หรือข้ามขั้นตอนที่เหลือซึ่งอยู่ถัดไปภายในวงวนและขึ้นรอบใหม่ของการทำขั้นตอนซ้ำเหมือนการขึ้นรอบใหม่ตามปรกติจนกว่าเงื่อนไขจะเป็นเท็จ

การทำงานของประโยคคำสั่ง break นั้นจะมีลักษณะคล้ายกับการทำงานของประโยคคำสั่ง goto แต่จะใช้เจาะจงกับวงวนเท่านั้น ส่วนประโยค goto เราจะใช้เมื่อไหรและตำแหน่งใดก็ได้ในแต่ละฟังก์ชัน และโปรดสังเกตว่า ประโยคคำสั่ง break และ continue จะมีผลต่อวงวนที่มีคำสั่งนี้อยู่และเราจะต้องใช้ประโยค continue ภายในวงวนเท่านั้น ถ้าเรามีการซ้อนลูปหลายๆชั้น คำสั่งทั้งสองนี้จะไม่มีผลใดๆต่อลูปชั้นนอกที่อยู่ถัดไปแต่จะมีผลต่อวงวนที่มีประโยคคำสั่งนี้เท่านั้น

ตัวอย่างการใช้งานประโยคคำสั่ง break และ continue

```
#include <stdio.h>

int main()
{
      int i = 0;

      for ( ; ; )
        if (i > 10)
            break;
        else
        {
            printf ("%d*%d = %d\n", i, i ,i*i);
            i++;
        }

      return 0;
}
```

สำหรับตัวอย่างนี้เราจะเห็นได้ว่า ประโยค for เป็นการวนลูปซ้ำแบบไม่รู้จบ แต่ทว่า เราได้สร้างเงื่อนไขไว้ว่า ถ้าตัวแปร i มีค่ามากกว่า 10 แล้วเราจะใช้คำสั่ง break ซึ่งส่งผลให้โปรแกรมหยุดการทำงานของวงวน

ตัวอย่างต่อไป เราจะใช้ทั้งคำสั่ง break และ continue ภายในวงวนแบบ do-while ซึ่งโปรแกรมจะอ่านข้อมูลจากผู้ใช้เมื่อกดคีย์ต่างๆบนแป้นพิมพ์แล้วเก็บไว้ในตัวแปร ch แบบ char โดยใช้ฟังก์ชันมาตราฐาน getchar() ถ้า ch เป็นตัวขระใดๆที่อยู่ในช่วงตั้งแต่ '0' ถึง '9' ก็ให้เพิ่มค่าของตัวแปร sum ขึ้น อีกเป็นจำนวนเท่ากับค่าของเลขโดดในขณะนั้น ถ้าค่าของ ch ไม่เท่ากับ '.' ก็ให้ไปเริ่มทำงานต่อที่จุดเริ่มต้นของวงวน แต่ถ้า ch เท่ากับ '.' หรือ ตัวแปร i มีค่ามากกว่า 999 โปรแกรมก็จะออกจากวงวน

```
#include <stdio.h>

int main()
{
      int i = 0, sum = 0;
      char ch;

      do
      {
        ch = (char)getchar();
        if ('0' <= ch && ch <= '9')
        {
            sum += (int)(ch - '0');
            i++;
        }

        if (ch != '.')
            continue;

        break;

      } while (i < 999);

      printf ("sum = %d\n", sum);
      return 0;
}
```

แบบฝึกหัดท้ายบท

1. จงอธิบายการทำงานของวงวนต่อไปนี้

```
int N = 5, a = 2;
for (i=0, x=1; i < N; i++, x*= a)
                ;
```

โดยที่ `i,x,a` และ `N` เป็นตัวแปรแบบ `int` และหาความสัมพันธ์ระหว่างค่าของตัวแปร `x`, `a` และ `N` หลังจากที่จบการทำงานของวงวน

2. จงหาค่าของตัวแปร `i` หลังจากที่โปรแกรมกระทำขั้นตอนต่อไปนี้ โดยกำหนดให้ `i` และ `N` เป็นตัวแปร แบบ `int`

1)
```
for (i=0, N = 1371; N &= N - 1; i++)
                    ;
```

2)
```
for (i=0, N = 1371 ; ; i++)
{
    N &= N - 1;
    if (!N)
       break;
}
```

และอธิบายว่าทำไมขั้นตอนการทำงานทั้งสองแบบจึงให้ผลเหมือนกัน

3. จงเขียนฟังก์ชันในภาษาซีที่หาค่าของ S ในสมการทางคณิตศาสตร์ต่อไปนี้

$$S = \sum_{i=0}^{N} \frac{1}{i!}$$

$$S = 4 - 8 \cdot \sum_{k=1}^{N} \frac{1}{(16k^2 - 1)}$$

$$S = \sum_{i=0}^{N} \sum_{j=i}^{M} (-1)^i \cdot 2^j$$

โดยกำหนดให้ใช้ประโยค `for` และ `do-while` รวมทั้งเลือกตัวแปรและแบบข้อมูลให้เหมาะสม

4. จงอธิบายการทำงานของโปรแกรมต่อไปนี้

```
#include <stdio.h>
int main ()
```

```
{
      int i, j, k, x = 0;

      for (i=0; i < 5; i++)
        for (j=0; j < i; j++)
        {
           k = (i+j-1);
           if (k % 2 == 0)
              x += k;
           else
             if (k % 3 == 0)
                x+= k - 2;
           printf ("%d", x);
        }

      printf ("\nx = %d", x);

      return 0;
}
```

5. จงพิจารณาขั้นตอนการทำงานต่อไปนี้

```
void long2bin (long dec)
{
      char bit;
      int i, width = 8*sizeof(long);

      for (i = 0; i < width; i++)
      {
         bit = ((dec >> (width-i-1)) & 1UL) ? '1' : '0';
         if (i==16)
           printf (" ");
         printf ("%c", bit);
      }

      printf ("\n");
}
```

ลองเขียนโปรแกรมภาษาซีง่ายๆที่มีการเรียกใช้ฟังก์ชัน `long2bin()` เช่น

```
long2bin (32768L);
long2bin (-1L);
long2bin (0xffff);
long2bin (04771L);
```

และหลังจากที่ทำการรันโปรแกรมแล้วให้สังเกตผลที่แสดงออกทางจอภาพ

6. จงเขียนโปรแกรมภาษาซี (หรือฟังก์ชัน) ที่ใช้หาค่าของนิพจน์ต่อไปนี้

$$val = X^P \bmod N$$

โดยที่เรากำหนดใช้ตัวระบุ `X, P, N` และ `val` เป็นชื่อของตัวแปรแบบ `long` และจงใช้โปรแกรมที่ได้ เขียนขึ้นหาค่าต่อไปนี้

1) $23^{503} \bmod 29$

2) $1009^{3000017} \bmod 1013$

ลองทำการคอมไพล์โปรแกรมต่อไปนี้ และสังเกตความเร็วในการทำงานของโปรแกรมรวมทั้งผลลัพธ์ที่ได้

```
#include <stdio.h>

int main()
{
      int i;
      long X, P, N;
      long j, power, val;

      X = 1009;   P = 1013;   N = 3000017L;

      power = X;
      val   = 1L;
      for (i=0; i < 8*sizeof(N); i++)
      {
        if ((N >> i) & 1L)
        {
             val = (val*power) % P;
             printf ("i = %4d, val = %5ld, power = %5ld\n",
                     i, val, power);
        }
        power = (power * power) % P;
      }

      printf ("pow(%ld,%ld) %% %ld = %ld\n", X, N, P, val);

      return 0;
}
```

7. ในการหาค่าของตัวแปร x ที่ทำให้สมการต่อไปนี้เป็นจริง

$$ax^2 + bx + c \quad = \quad 0, \qquad a \neq 0$$

เรามักจะใช้สูตร

$$x \quad = \quad -\frac{b}{2} \quad \pm \quad \sqrt{(\frac{b}{2})^2 - ac}$$

7.1 จงเขียนฟังก์ชันในภาษาซีที่พิมพ์คำตอบของสมการออกทางจอภาพ โดยให้ชื่อฟังก์ชันว่า solve_eqn และกำหนดให้ a,b และ c เป็นพารามิเตอร์แบบ double ของฟังก์ชัน โดยมีส่วนหัวของฟังก์ชันที่อยู่ใน รูปแบบต่อไปนี้

```
void solve_eqn (double a, double b, double c);
```

คำแนะนำ ฟังก์ชันมาตราฐานที่เราใช้ในการสร้างฟังก์ชัน solve_eqn() เช่น sqrt() สำหรับหาราก ที่สองของเลขทศนิยมจำนวนจริง จะมีส่วนหัวของฟังก์ชันที่นิยามไว้ในไฟล์ math.h ในลักษณะนี้

```
double  sqrt (double x);
```

นอกจากนี้จะต้องคำนึงถึงกรณีที่คำตอบของสมการสามารถเป็นได้ทั้งเลขจำนวนจริงเท่านั้นหรือเป็นเลขจำ-นวนเชิงซ้อนก็ได้

7.2 ลองเขียนโปรแกรมที่ใช้ฟังก์ชัน `solve_eqn()` ในการแก้สมการต่อไปนี้

```
x² - 1000x + 1 = 0
x² + 4x + 5    = 0
x² + 8x + 25   = 0
```

8. ในการหาค่าของราก $a^{1/m}$ เมื่อ a เป็นเลขจำนวนบวกใดๆ เราสามารถใช้สูตรต่อไปนี้ได้

$$x_0 = 1, \quad n = 0$$

$$x_n = \frac{1}{m}\left[(m-1)\cdot x_{n-1} + \frac{a}{x_{n-1}^{(m-1)}} \right], \quad n = 1,\ 2,\ 3,\ldots.$$

$$x_{n\to\infty} = \sqrt[m]{a}$$

สูตรข้างบนเป็นการคำนวณโดยใช้หลักของ Newton Iteration สำหรับสมการที่อยู่ในรูปของ

$$x^m - a = 0$$

และเราสามารถหาค่าประมาณของคำตอบ x ของสมการได้โดยคำนวณค่าของ x_N เช่น N มีค่าเท่ากับ 100

$$\sqrt[m]{a} \approx x_N, \quad N = 100$$

8.1 จงเขียนฟังก์ชันที่ใช้สูตรข้างบนในการแก้สมการต่อไปนี้

```
x² - 2= 0
x¹⁰ - 2      = 0
x⁵ - 10      = 0
```

และกำหนดให้ฟังก์ชันมีรูปแบบดังต่อไปนี้

```
double mth_root_of_a (double a, int m);
```

และถ้า a หรือ m มีค่าเป็นศูนย์หรือน้อยกว่าศูนย์ ให้ฟังก์ชันคืนค่าเป็น `-1.0` เพื่อเป็นการแจ้งให้ทราบว่า ผู้ใช้ผ่านค่าพารามิเตอร์ที่ไม่ถูกต้องตามเงื่อนไขของฟังก์ชัน

8.2 การหาค่าของ $a^{1/m}$ อีกในวิธีหนึ่ง เราสามารถคำนวณได้จากการใช้สูตรต่อไปนี้

$$a^{1/m} = e^{\ln(a)/m}$$

จงสร้างฟังก์ชันจากสมการข้างบนในการหาค่าของ $a^{1/m}$ โดยใช้ฟังก์ชันมาตราฐาน exp() และ log() จากไฟล์ math.h

```
double exp (double x);
double log (double x);
```

9. จงสร้างฟังก์ชันชื่อ Exp() สำหรับหาค่าโดยประมาณของ exp(x) โดยใช้สูตรต่อไปนี้

$$\exp(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}$$

ซึ่ง n เป็นจำนวนนับใดๆที่มากกว่าหรือเท่ากับศูนย์ และกำหนดให้ฟังก์ชัน Exp() มีรูปแบบดังต่อไปนี้

```
double  Exp (double x, int n);
```

10. จงใช้ฟังก์ชัน Exp() ที่สร้างขึ้นในแบบฝึกหัดข้อที่แล้ว ในการคำนวณหาค่าของ

```
exp(0.0)    exp(4.1)    exp(10.0)   exp(-0.05)
```

โดยกำหนดให้ค่า n เท่ากับ 10 และ 100 ตามลำดับ และลองเปรียบเทียบความแตกต่างระหว่างทั้งสองกรณี

11. อนุกรมไฟโบนาชชี (Fibonacci Series) เป็นอนุกรมที่มีลักษณะดังต่อไปนี้

```
1,1,2,3,5,8,13,21,...
```

ซึ่งเราสามารถเขียนความสัมพันธ์ทางคณิตศาสตร์ระหว่างตัวเลขแต่ละตัวได้ตามสมการข้างล่างนี้

$$\begin{aligned} f_1 &= 1 \\ f_2 &= 1 \\ f_3 &= f_2 + f_1 = 2 \\ f_4 &= f_3 + f_2 = 3 \\ &\cdots\cdots\cdots\cdots \\ f_n &= f_{n-1} + f_{n-2}, \quad n > 2 \end{aligned}$$

จงเขียนโปรแกรมในภาษาซีที่แสดงตัวเลขต่างๆของอนุกรมไฟโบนาชชีเมื่อ n มีค่าต่อไปนี้

```
n = 2       n = 7      n = 15      n = 23
```

12. จงเขียนโปรแกรมที่แสดงผลทางกราฟิกบนจอภาพอย่างง่ายๆสำหรับค่าของ y ในสมการต่อไปนี้

$$y = 30\, e^{-0.2x} \sin(2.0x)$$

โดยกำหนดให้ตัวแปร x มีค่าอยู่ระหว่าง 0.0 ถึง 10.0 และระยะห่างของจุด (x,y) แต่ละจุดเท่ากับ 0.25 เพื่อ ที่จะแสดงจุดต่างๆออกทางจอภาพ เราจะกำหนดใช้สัญลักษณ์ `'*'` แทนจุด (x,y) ในกราฟ โดยมีแกน x อยู่ในแนวตั้งจากบนลงล่างและแกน y อยู่ในแนวนอนตามจอภาพ โดยที่แต่ละบรรทัดจะมีเพียงจุดๆเดียว เท่านั้น เนื่องจากว่าค่าของ y เราจะคำนวณโดยใช้ข้อมูลแบบ `double` ดังนั้นเราจะต้องแปลงค่าของตัว แปร y จาก `double` ให้เป็นข้อมูลแบบ `int` ซึ่งทำได้ง่ายโดยตัดตัวเลขหลังจุดทศนิยมทิ้งไป โปรดดูตัว อย่างของผลลัพธ์ที่แสดงออกบนจอภาพ