บทที่ 6

==========

โครงสร้างการควบคุมการทำงานของโปรแกรม
(Program control structures)

วัตถุประสงค์

1) เพื่อให้ผู้เรียนได้เข้าใจโครงสร้างต่างๆ ที่เกี่ยวข้องกับการควบคุมลำดับการทำงานของโปรแกรม
2) เพื่อให้เข้าใจแนวคิดในการออกแบบโครงสร้างการกำหนดนิพจน์และกำหนดค่าให้กับตัวแปร
3) เพื่อให้รู้จักโครงสร้างต่างๆ ที่ผู้ออกแบบภาษาใช้ในการรวมกลุ่มคำสั่ง รวมถึงเข้าใจตรรกะของการจัดกลุ่มคำสั่ง
4) เพื่อให้เรียนรู้รูปแบบต่างๆ ของโครงสร้างคำสั่งที่ทำงานแบบมีเงื่อนไข
5) เพื่อให้เข้าใจตรรกะและรูปแบบโครงสร้างคำสั่งวนรอบ เพื่อการทำงานซ้ำหลายครั้ง
6) เพื่อให้รู้จักแนวคิดของการโปรแกรมเชิงโครงสร้างและแนวคิดการแบ่งการทำงานเป็นโปรแกรมย่อย
7) เพื่อให้รู้จักวิธีการออกแบบโปรแกรมที่ทนต่อข้อบกพร่องด้วยการใช้โครงสร้างของภาษาจัดการกับกรณียกเว้น หรือเหตุการณ์ผิดปกติทั้งหลาย

การศึกษาเกี่ยวกับโครงสร้างการควบคุมการทำงานของโปรแกรม เป็นการศึกษารูปแบบต่างๆ ของคำสั่งเพื่อการควบคุมลำดับการทำงานของโปรแกรมให้ตรงตามที่โปรแกรมเมอร์ต้องการ ในกลุ่มภาษาเชิงคำสั่งและภาษาเชิงวัตถุ โครงสร้างการควบคุมจะประกอบด้วยโครงสร้างของคำสั่งกำหนดค่า คำสั่งรวมกลุ่มการทำงาน คำสั่งแบบมีเงื่อนไข และคำสั่งทำซ้ำ คำสั่งทั้งสี่ประเภทนี้เป็นพื้นฐานหลักในการทำโปรแกรม นอกจากนี้ภาษาคอมพิวเตอร์รุ่นใหม่มักจะมีโครงสร้างเพื่อการทำโปรแกรมย่อย และโครงสร้างเพื่อการจัดการกรณียกเว้น เพื่อให้การทำโปรแกรมเป็นโครงสร้างที่ดีและเป็นโปรแกรมที่เชื่อถือได้ในด้านความทนทานต่อข้อบกพร่อง

6.1 นิพจน์

(Expression)

ส่วนประกอบพื้นฐานของคำสั่งแทบทุกคำสั่งคือนิพจน์ นิพจน์เป็นส่วนที่ผลิตค่าให้กับคำสั่ง ตัวอย่างของนิพจน์ได้แก่ 1, 57.732, 1+(3-1.77)+4.8, x+11*y นิพจน์อาจจะเป็นค่าตัวเลขเดี่ยวๆ (เรียกว่า literal) เช่น 4, 57.7 หรือเป็นนิพจน์ที่ประกอบขึ้นจากค่าและตัวแปรต่างๆ มากระทำกันด้วยตัวดำเนินการหรือโอเปอเรเตอร์ (operator) เช่น x+11*y/2 โอเปอเรเตอร์ที่ต้องการโอเปอแรนด์ (operand) ตัวเดียว เช่น –b เรียกว่า โมนาดิก(monadic) หรือ ยูนารี(unary) โอเปอเรเตอร์ที่ต้องการโอเปอแรนด์สองตัว เช่น โอเปอเรเตอร์ +, -, *, / เรียกว่า ไดอาดิก(dyadic) หรือ ไบนารี(binary)

รูปแบบการเขียนนิพจน์ที่ใช้ในภาษาคอมพิวเตอร์ มีด้วยกัน 3 รูปแบบคือ รูปแบบอินฟิกซ์ (infix notation), รูปแบบพรีฟิกซ์ (prefix notation) และ รูปแบบโพสฟิกซ์ (postfix notation)

<u>รูปแบบอินฟิกซ์</u>

นิพจน์ในรูปแบบอินฟิกซ์ เขียนโอเปอเรเตอร์อยู่ตรงกลางระหว่างโอเปอแรนด์ทางซ้ายและทางขวา เช่น x+y ในกรณีที่นิพจน์มีมากกว่าหนึ่งโอเปอเรเตอร์ และโอเปอเรเตอร์เหล่านั้นมีความสำคัญเท่าเทียมกัน เช่น x+y+z, 1+3-2+x, a=b=c=3 ผู้ออกแบบภาษาจะต้องกำหนดทิศทางการคำนวณว่า จะทำจากซ้ายไปขวา หรือจากขวาไปซ้าย ทิศทางในการคำนวณนี้เรียกว่า แอสโซซิเอทีฟ (associative) ถ้าการคำนวณทำจากซ้ายไปขวาจะเรียกว่า left-associative เช่น 1+3-2+x จะมีลำดับการทำงานเป็น (((1+3)-2)+x) แต่ถ้าการคำนวณทำจากขวาไปซ้าย จะเรียกว่า right-associative เช่น a=b=c=3 จะมีลำดับการทำงานเป็น (a=(b=(c=3)))

ในกรณีที่นิพจน์มีโอเปอเรเตอร์ที่ความสำคัญไม่เท่ากัน เช่น x+p*q ผู้ออกแบบภาษาจะต้องกำหนดลำดับความสำคัญก่อนหลังเรียกว่า การทำก่อน หรือ พรีซีเดนซ์ (precedence) เช่นถ้าการคูณสำคัญกว่าการบวก x+p*q จะมี ลำดับการทำงานเป็น (x+(p*q)) รูปที่ 6.1 แสดงการกำหนดลำดับความสำคัญของโอเปอเรเตอร์ในภาษา C จากความสำคัญสูงสุดลงไปถึงความสำคัญต่ำสุด

ระดับความสำคัญ	เครื่องหมายดำเนินการ	ชื่อตัวดำเนินการ
1	tokens, a[k], f( ), ., ->	literals, subscript, function call, selection
2	++, --	postfix increment/ decrement
3**	++, --, ~, -, sizeof, !, &, *	prefix increment/ decrement, unary operators, storage, logical negation, indirection
4	(typename)	cast
5	*, /, %	multiplicative operators
6	+, -	additive operators
7	<<, >>	shift
8	<, >, <=, >=	relational
9	= =, !=	equality
10	&	bitwise and
11	Λ	bitwise xor
12	|	bitwise or
13	&&	logical and
14	||	logical or
15**	?:	conditional
16**	=, +=, -=, *=, /=, %=, <<=, >>=, &=, Λ=, |=	assignment
17	,	sequential evaluation

** เครื่องหมายดำเนินการที่มีทิศทางการดำเนินการจากขวาไปซ้าย (right associative)

รูปที่ 6.1 ลำดับการทำก่อนของตัวดำเนินการในภาษา C

ในภาษา C จัดให้เครื่องหมายกำหนดค่าให้ตัวแปร (เครื่องหมาย =) เป็นโอเปอเรเตอร์ แต่ในบางภาษาเช่น ภาษา Ada ไม่จัดเครื่องหมายกำหนดค่า (เครื่องหมาย :=) เป็นโอเปอเรเตอร์จึงไม่สามารถเขียน := อยู่ภายในนิพจน์ได้ (เช่นไม่สามารถเขียน *a:=b:=c:=3*) นอกจากนี้ในบางภาษาอาจให้เครื่องหมายกำหนดค่าเป็นโอเปอเรเตอร์ แต่ไม่สามารถเขียนให้สัมพันธ์ต่อเนื่องกันได้ เรียกว่า non-associative ในกรณีเช่นนี้ จะต้องเขียนโอเปอเรเตอร์แยกเป็นแต่ละคำสั่ง เช่น *a=b=c=3* จะต้องเขียนเป็น 3 คำสั่งคือ

$$c = 3$$
$$b = c$$
$$a = b$$

<u>รูปแบบพรีฟิกซ์ และ โพสฟิกซ์</u>

รูปแบบพรีฟิกซ์บางครั้งเรียกว่า รูปแบบโพลิช (Polish notation) เนื่องจากผู้คิดรูปแบบนี้เป็นนักคณิตศาสตร์ชาวโปแลนด์ รูปแบบนี้ถูกสร้างขึ้นเพื่อให้การเขียนนิพจน์คณิตศาสตร์ไม่จำเป็นต้องใช้วงเล็บเพื่อระบุลำดับก่อนหลังของการคำนวณ รูปแบบพรีฟิกซ์จะเขียนโอเปอเรเตอร์ไว้ก่อนหน้าโอเปอแรนด์ เช่น นิพจน์ 4+(5*6) จะเขียนอยู่ในรูปแบบพรีฟิกซ์ได้เป็น +4*56 และนิพจน์ (4+5)*6 จะเขียนอยู่ในรูปแบบพรีฟิกซ์ได้เป็น *+456 นิยามของรูปแบบนิพจน์พรีฟิกซ์แสดงได้ดังนี้

```
<expression> ::= <0-operand operator>
               | <1-operand operator> < expression>
               | <2-operand operator> < expression>< expression>
               | …
```

รูปแบบโพสฟิกซ์มีลักษณะตรงข้ามกับรูปแบบพรีฟิกซ์ บางครั้งจึงเรียกว่า รูปแบบโพลิชสลับลำดับ (reverse Polish notation) รูปแบบโพสฟิกซ์จะเขียนโอเปอเรเตอร์ไว้หลังโอเปอแรนด์ เช่น นิพจน์ 4+(5*6) จะเขียนอยู่ในรูปแบบโพสฟิกซ์ได้เป็น 456*+ และนิพจน์ (4+5)*6 จะเขียนอยู่ในรูปแบบโพสฟิกซ์ได้เป็น 45+6*นิยามของการเขียนนิพจน์รูปแบบโพสฟิกซ์แสดงได้ดังนี้

```
<expression> ::= <0-operand operator>
               | < expression><1-operand operator>
               | < expression>< expression><2-operand operator>
               | …
```

วิธีการเขียนนิพจน์ในทั้งสามรูปแบบ แสดงตัวอย่างเปรียบเทียบได้ดังรูปที่ 6.2

นิพจน์อินฟิกซ์	นิพจน์พรีฟิกซ์	นิพจน์โพสฟิกซ์
3+4	+34	34+
3+4*5	+3*45	345*+
(3+4)*5	*+345	34+5*
(3+4)*(5-6)	*+34-56	34+56-*

รูปที่ 6.2 ตัวอย่างเปรียบเทียบวิธีการเขียนนิพจน์ในรูปแบบอินฟิกซ์, พรีฟิกซ์ และโพสฟิกซ์

ในทางคณิตศาสตร์การเขียนนิพจน์ส่วนใหญ่ จะใช้รูปแบบอินฟิกซ์เช่น 2+3*4/6 แต่มีการดำเนินการบางอย่างที่ใช้รูปแบบพรีฟิกซ์ (เช่น -4) และรูปแบบโพสฟิกซ์ (เช่น 5!) ในภาษาคอมพิวเตอร์ เช่น ภาษา C การเขียนนิพจน์ส่วนใหญ่จะใช้รูปแบบอินฟิกซ์ แต่มีโอเปอเรเตอร์ หรือ เครื่องหมายดำเนินการบางอย่างที่ใช้รูปแบบพรีฟิกซ์ (เช่น ++x) และโพสฟิกซ์ (เช่น x- -)

<u>การคำนวณผลลัพธ์ของนิพจน์</u>

นิพจน์คณิตศาสตร์เช่น 2*7 สามารถคำนวณผลลัพธ์ได้เป็น 14 ด้วยวิธีการคูณเลขจำนวนเต็ม และนิพจน์ 2.5*4.326 คำนวณได้ผลลัพธ์เป็น 10.8150 ด้วยวิธีการคูณเลขทศนิยม จะเห็นได้ว่าเครื่องหมาย * ใช้เป็นโอเปอเรเตอร์ในการคูณเลขจำนวนเต็ม และในการคูณเลขทศนิยม การใช้เครื่องหมายโอเปอเรเตอร์เดียวทำงานมากกว่าหนึ่งแบบ จะเรียกว่า การโอเวอร์โหลดโอเปอเรเตอร์ (operator overloading) ซึ่งเป็นการกระทำที่เกิดขึ้นบ่อยมากในภาษาคอมพิวเตอร์ เช่น ภาษา C++, Java, C, Ada ทั้งนี้เพื่อลดจำนวนเครื่องหมายที่ใช้ในภาษาคอมพิวเตอร์นั้นๆ การมีเครื่องหมายแทนโอเปอเรเตอร์มากเกินไป จะทำให้การเรียนรู้และจดจำรูปแบบคำสั่งในภาษานั้นๆ ทำได้ยาก

การคำนวณผลลัพธ์ของนิพจน์ที่เขียนอยู่ในรูปแบบอินฟิกซ์ จะต้องใช้ข้อมูลเกี่ยวกับ precedence และ association ประกอบการพิจารณาลำดับการคำนวณ เช่น นิพจน์ในภาษา Pascal (ภาษา Pascal ใช้เครื่องหมาย = แทนการเปรียบเทียบค่าเท่ากับ ซึ่งจะตรงกับเครื่องหมาย = = ในภาษา C)

```
a = b < c
```

ภาษา Pascal กำหนดให้ = และ < มี precedence ระดับเดียวกัน และเป็น left-association ถ้ากำหนดค่าของ a, b, c เป็นค่า FALSE และในภาษา Pascal ค่า FALSE มีค่าน้อยกว่าค่า TRUE ดังนั้นนิพจน์ข้างต้นคำนวณค่าผลลัพธ์ได้เป็น

```
((FALSE = FALSE) < FALSE)   => (TRUE <FALSE)
                            => FALSE
```

นิพจน์เดียวกันนี้ ถ้าเขียนในรูปแบบภาษา C จะเป็นดังนี้

a = = b < c

ในภาษา C เครื่องหมาย < มีระดับ precedence ต่ำกว่าเครื่องหมาย = = และในภาษา C ไม่มีค่า TRUE / FALSE แต่ใช้ตัวเลข 0 แทนค่า FALSE และตัวเลขที่ไม่ใช่ 0 (มักนิยมใช้เลข 1) แทนค่า TRUE ดังนั้นนิพจน์ข้างต้นคำนวณค่าผลลัพธ์ได้เป็น

((a = = b) < c) => ((0 = = 0) < 0)
=> (1 < 0)
=> 0 (แทนความหมาย FALSE)

การกำหนดค่าของนิพจน์ให้ตัวแปรในภาษา Pascal ใช้เครื่องหมาย := แทนการกำหนดค่า เช่น x:=3; ข้อความนี้จัดเป็นคำสั่งกำหนดค่า (assignment statement) แต่ในภาษา C ข้อความสั่งกำหนดค่า เช่น x =3; จัดเป็นนิพจน์โดยเครื่องหมาย = จัดเป็นโอเปอเรเตอร์ที่มีลำดับการทำงานจากขวาไปซ้าย ผลลัพธ์ของนิพจน์จะเป็นค่าของโอเปอแรนด์ที่ปรากฏข้างซ้ายมือของเครื่องหมาย = ดังนั้นข้อความ x =3; จะได้ผลลัพธ์เป็น 3 และข้อความต่อไปนี้

a = b = c = 0;

เป็นทั้งคำสั่งและนิพจน์ โดยลำดับการคำนวณค่าของนิพจน์จะเป็น

(a = (b = (c = 0)))

นั่นคือผลลัพธ์ของนิพจน์เป็น 0 ซึ่งเป็นค่าของตัวแปร (หรือโอเปอแรนด์) a และเมื่อเครื่องหมายกำหนดค่าเป็นโอเปอเรเตอร์และคำสั่งกำหนดค่าสามารถพิจารณาได้ว่าเป็นนิพจน์ในภาษา C เราจึงเขียนคำสั่งต่อไปนี้ได้

*while ((*p++ = *q++)!=0) { };*

นิพจน์ **p++ = *q++* มีความหมายว่าให้ copy ค่าจากหน่วยความจำตำแหน่งที่ชี้โดย q ไปไว้ที่หน่วยความจำตำแหน่งที่ชี้โดย p หลังจากนั้นเลื่อนตัวชี้ p และ q ไปหนึ่งตำแหน่ง ค่าที่เป็นผลลัพธ์ของนิพจน์นี้คือค่าที่ถูก copy ซึ่งจะถูกนำไปเปรียบเทียบค่าว่าถ้าไม่ใช่ศูนย์ เงื่อนไขของคำสั่ง while จะเป็นจริงส่งผลให้มีการทำคำสั่ง { }; ซึ่งเป็นคำสั่งว่าง หมายถึงไม่มีการกระทำใด ต่อจากนั้นจะมีการวนลูปซ้ำซึ่งเริ่มด้วยการตรวจสอบค่าของนิพจน์ (**p++ = *q++*) ว่าเป็นศูนย์หรือไม่ การวนลูปจะสิ้นสุดเมื่อนิพจน์ให้ค่าที่เป็นศูนย์

จากตัวอย่างคำสั่งวนลูปข้างต้น นิพจน์ (*p++ = *q++) ทำหน้าที่เป็นนิพจน์ตรวจสอบเงื่อนไขว่าเป็นจริงหรือเท็จให้กับคำสั่ง while แต่ผลข้างเคียง (side effect) ที่เกิดขึ้นคือทำให้เกิดการ copy ค่าจากตำแหน่งที่ชี้โดย q ไปยังตำแหน่งที่ชี้โดย p และตำแหน่งที่มีการ copy ค่าจะเลื่อนไปตามลำดับ (ด้วยการใช้โอเปอเรเตอร์ ++) จนกว่าค่าที่ copy จะเป็นศูนย์

การเขียนนิพจน์ในภาษา C นอกจากใช้เครื่องหมาย = แล้ว ยังสามารถใช้เครื่องหมาย ?: ที่เป็นโอเปอเรเตอร์หมายถึงการทำงานแบบมีเงื่อนไข if.. then.. else.. (เรียกว่า เทอนารีโอเปอเรเตอร์ หรือ ternary operator เพราะใช้ 3 โอเปอแรนด์) เช่น x!=0? 1/x:0 หมายถึง if (x!=0) then return 1/x else return 0 นิพจน์ที่ใช้โอเปอเรเตอร์ ?: และให้ผลลัพธ์เป็นค่า (value) สามารถปรากฏทางขวาของเครื่องหมาย = ถ้าให้ผลลัพธ์เป็นตำแหน่ง (address)สามารถปรากฏทางซ้ายของเครื่องหมาย = ดังตัวอย่างต่อไปนี้

q = (x!=0) ? 1/x : 0;
*(p>r) ? *p: *r = 0;*

ข้อความสั่งบางประเภทมีการใช้นิพจน์ย่อยหลายนิพจน์ การคำนวณผลลัพธ์ของนิพจน์ย่อยเหล่านั้นไม่จำเป็นต้องกระทำทั้งหมดในคราวเดียว เช่น ถ้าในโปรแกรมมีการประกาศอาร์เรย์ a[1..10] และมีการใช้คำสั่งต่อไปนี้

IF (i <=10) AND (a[i] > 0) THEN ...

นิพจน์ย่อยในคำสั่งนี้คือ (i <=10) และ (a[i] > 0) ในกรณีที่ i มีค่ามากกว่า 10 นิพจน์ย่อยลำดับแรกจะเป็นเท็จ ซึ่งส่งผลให้เงื่อนไขทั้งหมดของคำสั่ง IF เป็นเท็จ โดยไม่จำเป็นต้องพิจารณาค่าของ (a[i] > 0) ดังนั้นการประมวลผลในกรณีเช่นนี้จะทำได้เร็วขึ้นถ้าละเว้นการคำนวณ (a[i] > 0) เมื่อ (i <=10) ให้ผลลัพธ์เป็นเท็จ (แต่ถ้า i <=10 เป็นจริงเราจำเป็นต้องคำนวณ a[i] > 0) การชะลอการคำนวณผลลัพธ์นิพจน์ไว้จนกว่าจะมีความจำเป็นต้องใช้ค่าของนิพจน์นั้นเรียกว่า การคำนวณแบบเกียจคร้าน (lazy evaluation) ลักษณะการ

ทำงานแบบนี้ใช้มากในภาษาที่ใช้ในการทำโปรแกรมเชิงหน้าที่ทั้งนี้เพื่อเพิ่มความเร็วในการประมวลผลโปรแกรม

ในภาษา C มีการกำหนดวิธีการทำงานของบางโอเปอเรเตอร์ เช่น &&,|| ให้ใช้การคำนวณแบบเกียจคร้าน และเรียกโอเปอเรเตอร์พวกนี้ว่า short-cut operators หรือ short-circuit operators ภาษา Ada เป็นภาษารุ่นแรกๆ ที่ใช้โอเปอเรเตอร์ short-cut เหล่านี้ แต่วิธีเขียนคำสั่งในภาษา Ada จะต่างออกไปดังนี้

```
if (I <= 10) and then (A[I] >0) then … end if ;
```

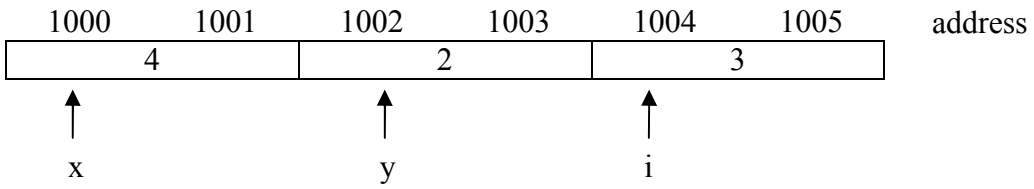
6.2 คำสั่งกำหนดค่า

(Assignment statement)

คำสั่งกำหนดค่าเป็นคำสั่งที่ใช้ในการสร้างหรือเปลี่ยนแปลงค่าให้กับตัวแปร เช่นคำสั่งในภาษา C

i = (x+y) / 2;

จะมีผลให้เกิดการคำนวณผลลัพธ์ของนิพจน์ (x+y) /2 และค่าที่ได้จะถูกกำหนดให้กับตัวแปร i นิพจน์ (x+y)/2 จะเรียกว่าแหล่ง (source) ของค่า และตัวแปร i จะเรียกว่า เป้าหมาย (target or destination) ของการกำหนดค่า ถ้าพิจารณาถึงโครงสร้างภายในหน่วยความจำซึ่ง สมมุติให้ตำแหน่งเก็บข้อมูลเป็นดังรูปที่ 6.3 และตัวแปรทุกตัวเป็นชนิด int



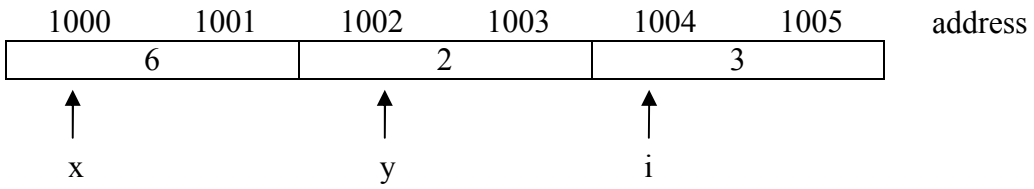
รูปที่ 6.3 โครงสร้างภายในหน่วยความจำของตัวแปร x, y, i

นิพจน์ (x+y) /2 ที่ปรากฏอยู่ทางขวามือของเครื่องหมายกำหนดค่า (เครื่องหมาย = ในภาษา C และเครื่องหมาย := ในภาษา Pascal และ Ada) จะเรียกว่าค่าทางขวามือ หรือ r-value ส่วนตัวแปร i ที่ปรากฏอยู่ทางซ้ายมือของเครื่องหมายกำหนดค่าจะเรียกว่าค่าทางซ้ายมือ หรือ l-value ค่า r-value จะหมายถึงค่าที่เก็บอยู่ในหน่วยความจำตำแหน่งต่างๆ ส่วน l-value จะหมายถึงตำแหน่งหรือแอดเดรสของหน่วยความจำ ดังนั้นชื่อตัวแปร x และ y ในนิพจน์ ซึ่งปรากฏเป็น r-value จึงหมายถึงค่า 4 และ 2 ตามลำดับ (ตามรูปที่ 6.3) และเมื่อนิพจน์ได้รับการคำนวณค่าแล้วได้ผลลัพธ์เป็น 3 ค่านี้ถูกกำหนดให้ตัวแปร i ที่เป็น l-value การอ้างถึงชื่อตัวแปร i จึงหมายถึงตำแหน่ง 1004 ในหน่วยความจำ ซึ่งเป็นตำแหน่งที่จองไว้เพื่อเก็บค่าของ i ผลของการทำงานของคำสั่งนี้ จึงเป็นการบรรจุค่า 3 ลงที่ตำแหน่ง 1004

จากตัวอย่างข้างต้น ถ้าต่อมามีคำสั่งเพิ่มเติมดังนี้

*x = y*i ;*

ในคำสั่งนี้ y และ i เป็น r-values จึงหมายถึงค่า 2 และ 3 ตามลำดับ แต่ x เป็น l-value จึงหมายถึงตำแหน่งที่ 1000 เมื่อมีการคำนวณค่าของนิพจน์ y*i จึงได้ค่าใหม่ของ x เป็น 6 เก็บไว้ที่ตำแหน่ง 1000 (ตามรูปที่ 6.4)



รูปที่ 6.4 โครงสร้างภายในหน่วยความจำภายหลังทำคำสั่ง *x = y*i ;*

กระบวนการประมวลผลคำสั่งกำหนดค่า จะเริ่มต้นด้วยการคำนวณค่าผลลัพธ์ของนิพจน์ที่เป็น r-value เมื่อได้ค่านิพจน์แล้วจึงจะบรรจุค่าให้กับตัวแปรที่เป็น l-value ขั้นตอนการทำงานเช่นนี้ จึงทำให้การเขียนคำสั่ง i = i+1; เป็นคำสั่งที่ถูกต้องตามไวยากรณ์ของภาษาคอมพิวเตอร์ เพราะหมายถึงการกำหนดค่าใหม่ให้กับตัวแปร i ในขณะที่ทางคณิตศาสตร์การใช้เครื่องหมาย = จะเป็นการเปรียบเทียบค่า เช่น 1+1 = 2 หรือ x*2 = x+x จะเห็นได้ว่าในทางคณิตศาสตร์ นิพจน์ทางซ้ายมือ และทางขวามือของเครื่องหมาย = เป็น r-values ทั้งคู่ การเขียนคำสั่ง x = x+1 จึงผิด เมื่อพิจารณาตามหลักการของคณิตศาสตร์ เนื่องจากคณิตศาสตร์ไม่มีแนวคิดในเรื่อง l-value

รูปแบบการเขียนคำสั่งกำหนดค่าในภาษาต่างๆ แสดงได้ดังนี้

รูปแบบคำสั่ง	ภาษา
A := B	Pascal, Ada
A = B	C, C++, FORTRAN, PL/I, Java, ML, SNOBOL4, Prolog
MOVE B TO A	COBOL
A ← B	APL
(SETQ A B)	LISP

6.3 คำสั่งเป็นกลุ่ม

(Compound statement)

คำสั่งกำหนดค่าเป็นคำสั่งที่ส่งผลให้การทำงานของคอมพิวเตอร์เกิดการเปลี่ยนสถานะ (state) เช่น จากตัวอย่างในรูปที่ 6.3 และ 6.4 คำสั่ง x = y*i; ทำให้เกิดการเปลี่ยนสถานะจาก {<x,4>, <y,2>, <i,3>} เป็นสถานะใหม่ {<x,6>, <y,2>, <i,3>} และเมื่อพิจารณาร่วมกับคำสั่งก่อนหน้า คือ คำสั่ง i = (x+y) / 2; จะเห็นลำดับของการเปลี่ยนสถานะดังต่อไปนี้

สถานะก่อนทำคำสั่ง	คำสั่ง	สถานะหลังทำคำสั่ง
{<x,4>, <y,2>, <i,undefine>	*i = (x+y) / 2;*	{<x,4>, <y,2>, <i,3>}
{<x,4>, <y,2>, <i,3>}	*x = y*i;*	{<x,6>, <y,2>, <i,3>}

การเปลี่ยนสถานะเป็นลำดับต่อเนื่องกันไปเช่นนี้ เป็นผลจากการใช้คำสั่งควบคุมการทำงานของโปรแกรมประเภทที่เรียกว่า คำสั่งต่อเนื่อง หรือ sequence ซึ่งปรากฏในโปรแกรมด้วยรูปแบบดังนี้

```
statement_1;
statement_2;
statement_3;
…
statement_n;
```

คำสั่งต่อเนื่องเหล่านี้สามารถถูกจัดให้เป็นชุดเดียวกันเรียกว่า คำสั่งเป็นกลุ่ม (compound statement) ด้วยการใช้เครื่องหมายวงเล็บ {…} ตามรูปแบบของภาษา C, C++, Java หรือใช้ข้อความ begin .. end ตามรูปแบบของภาษา Pascal, Ada

<u>รูปแบบภาษา C/ C++/ Java</u>

```
{ statement_1;
  statement_2;
  …
  statement_n;
}
```

<u>รูปแบบภาษา Pascal/ Ada</u>

```
begin
    statement_1;
    statement_2;
    …
    statement_n;
end
```

คำสั่งต่อเนื่องเป็นโครงสร้างพื้นฐานที่ใช้สร้างคำสั่งเป็นกลุ่ม ในภาษา C/ C++/ Java คำสั่งที่ปรากฏต่อเนื่องกันนั้น ทุกคำสั่งต้องจบด้วยเครื่องหมาย ; แต่ในภาษา Pascal/ Ada คำสั่งต่อเนื่องที่ปรากฏเป็นกลุ่มคำสั่ง ใช้เครื่องหมาย ; เพื่อคั่นระหว่างคำสั่ง ดังนั้นคำสั่งสุดท้าย (statement_n) จึงอาจปรากฏเครื่องหมาย ; หรือไม่ปรากฏก็ได้

6.4 คำสั่งแบบมีเงื่อนไข

(Conditional statement)

คำสั่งแบบมีเงื่อนไขหรือ บางครั้งเรียกว่าคำสั่งมีการเลือก (selection) ใช้ในการควบคุมลำดับการทำงานของโปรแกรมให้เลือกทำกลุ่มคำสั่งเพียงบางกลุ่ม โดยการเลือกว่าจะทำกลุ่มคำสั่งใดใช้เงื่อนไขเป็นตัวกำหนด

ภาษาคอมพิวเตอร์โดยทั่วไปจะมีโครงสร้างคำสั่ง IF-THEN ให้ใช้ในกรณีต้องการเลือกหรือไม่เลือกทำกลุ่มคำสั่ง และใช้โครงสร้าง IF-THEN-ELSE ในกรณีเลือกทำกลุ่มคำสั่งที่ 1 หรือกลุ่มคำสั่งที่ 2 ในกรณีมีการเลือกกลุ่มคำสั่งมากกว่าสองกลุ่ม อาจใช้คำสั่ง IF หลายคำสั่งซ้อนกัน หรือใช้คำสั่ง switch (ภาษา Pascal เรียกคำสั่ง case) ตัวอย่างต่อไปนี้แสดงรูปแบบการเขียนคำสั่งมีการเลือกเปรียบเทียบระหว่างภาษา C และภาษา Ada

<u>ภาษา C</u>

ตัวอย่างที่ 1:

```
if (x>x_max) {
        x=x_max;
}
```

ตัวอย่างที่ 2:

```
if (x<0) {
        y=-x;
}
else {
        y=x;
}
```

ตัวอย่างที่ 3:

```
switch (n) {
        case 0:  printf("empty");
                 break;
        case 1:  printf("single");
                 break;
        default: printf("multiple");
                 break;
}
```

<u>ภาษา Ada</u>

ตัวอย่างที่ 1:

```
if X > X_Max then
        X := X_Max;
end if;
```

ตัวอย่างที่ 2:

```
if X < 0 then
        Y := -X;
else
        Y := X;
end if;
```

ตัวอย่างที่ 3:

```
case N is
        when 0 => Put ("empty");
        when 1 => Put ("single");
        when others => Put ("multiple");
end case;
```

6.5 คำสั่งทำซ้ำ

(Iteration statement)

การทำซ้ำ (iteration, repetition or loop) เป็นวิธีการควบคุมลำดับการทำงานของโปรแกรมให้วนทำงานกับกลุ่มคำสั่งบางกลุ่มซ้ำหลายครั้งเรียก การทำงานแบบนี้ว่า การวนลูป กลุ่มคำสั่งที่ถูกทำงานซ้ำๆ เรียกว่า บอดี้ของลูป (loop body) และจะมีส่วนควบคุมการวนลูป เรียกว่า ตัวแปรควบคุม (control variable) เพื่อควบคุมการทำซ้ำให้ได้จำนวนรอบตามที่ต้องการ

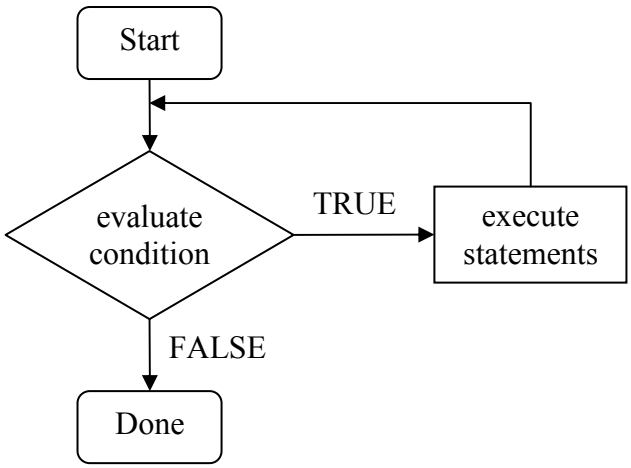
คำสั่งในการทำซ้ำของภาษาคอมพิวเตอร์ปัจจุบัน เช่นภาษา C, Pascal ประกอบด้วยคำสั่ง while, for, และ do_while (ภาษา Pascal ใช้ repeat-until) รูปแบบของคำสั่งเป็นดังนี้

	while repeat_condition **do** body_statements
(ภาษา Pascal)	**for** control_var := lower_bound **to** upper_bound **do** body_statements
(ภาษา C)	**for (**initialization; repeat_condition; increment**)** {body_statements}
(ภาษา Pascal)	**repeat** body_statements **until** stop_ condition
(ภาษา C)	**do** body_statements **while** repeat_condition

โครงสร้างการทำงานของคำสั่งทำซ้ำ แสดงในลักษณะของผังงานได้ดังต่อไปนี้

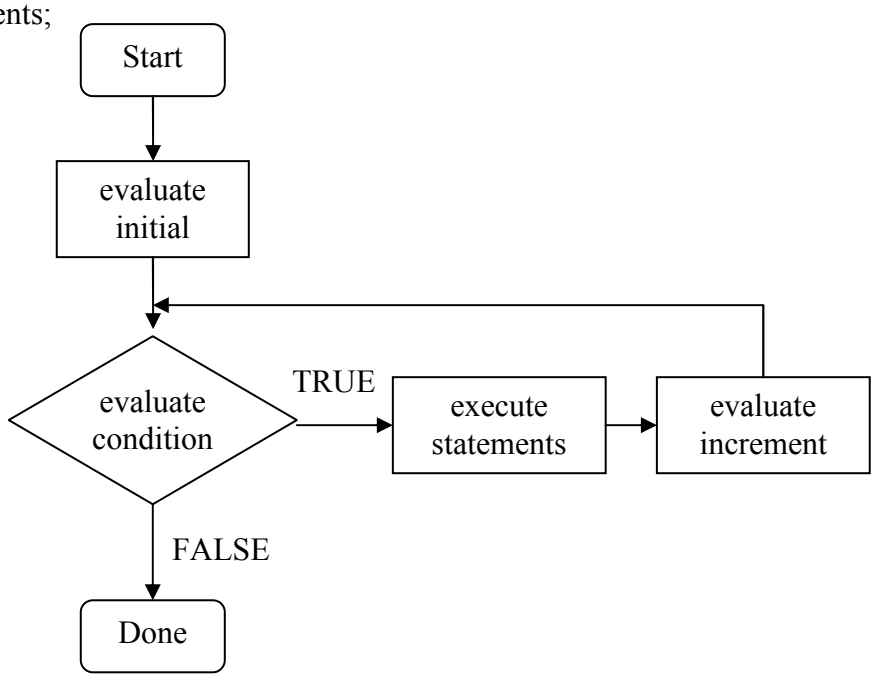
<u>คำสั่ง while</u>

```
while (condition)
        statements;
```



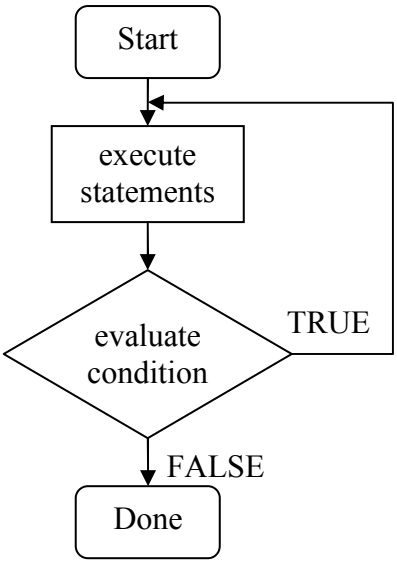
<u>คำสั่ง for</u>

```
for (initial; condition; increment)
        statements;
```



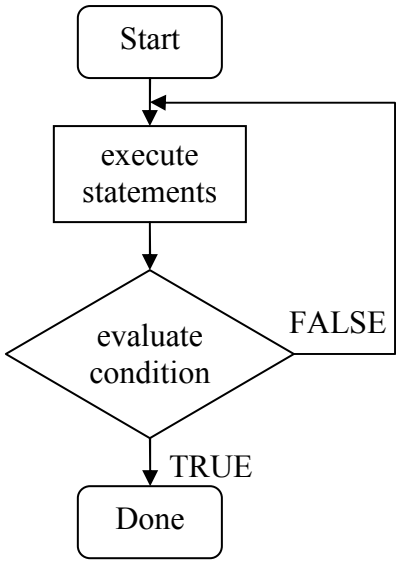
<u>คำสั่ง do-while</u> (ภาษา C)

```
do statements;
while (condition);
```



<u>คำสั่ง repeat-until</u> (ภาษา Pascal)

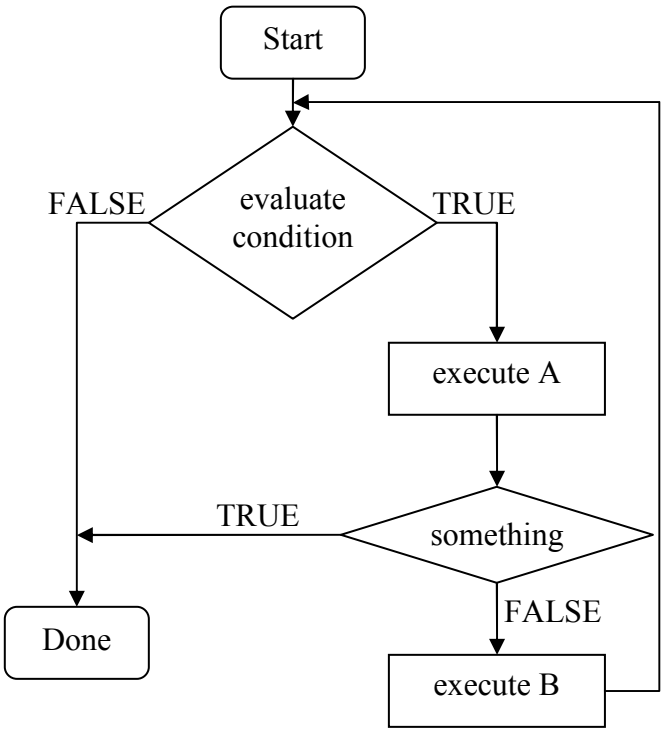
```
repeat statements
until (condition)
```



ในกรณีมีเหตุการณ์บางอย่างเกิดขึ้นและต้องการหยุดการทำซ้ำโดยทันที ภาษา C/C++ มีคำสั่ง break ให้ใช้ในกรณีเช่นนี้ โครงสร้างการทำงานของคำสั่งแสดงได้ดังนี้

<u>คำสั่ง break</u>

```
While (condition)
{
        A;
        if (something) break;
        B;
}
```



6.6 โปรแกรมย่อย

(Subprogram)

โปรแกรมย่อย หรือ รูทีน (routine) คือ กลุ่มของคำสั่งในโปรแกรมที่ถูกแยกออกเป็นโมดูลเพื่อทำหน้าที่เฉพาะอย่าง มีชื่อของโมดูล และมีพารามิเตอร์หรือ อาร์กิวเมนต์ เพื่อใช้ในการติดต่อรับส่งค่าระหว่างโปรแกรมกับโมดูล โปรแกรมย่อยแบ่งออกเป็นสองประเภทคือ โปรแกรมย่อยที่มีการส่งค่ากลับ เรียกว่า ฟังก์ชัน (function) และโปรแกรมย่อยที่ไม่มีการส่งค่ากลับ เรียกว่า โพรซีเจอร์ (procedure) ในภาษา Pascal และภาษา Ada มีการใช้ฟังก์ชันและโพรซีเจอร์ แต่ภาษา C/C++/Java ใช้ฟังก์ชันเพียงอย่างเดียว โดยฟังก์ชันที่ไม่มีการส่งค่ากลับ (ระบุค่าเป็น void) จะทำหน้าที่เสมือนเป็นโพรซีเจอร์ ตัวอย่างต่อไปนี้แสดงการเขียนโพรซีเจอร์และฟังก์ชัน ในรูปแบบภาษา C และภาษา Ada

<u>โพรซีเจอร์ในรูปแบบภาษา C</u>

```
void print_square (int n) {
    printf("%d" , n*n);
}
```

<u>โพรซีเจอร์ในภาษา Ada</u>

```
prodedure Print_Square (N: Integer) is
begin
    Put(N*N);
end Print_ Square;
```

<u>ฟังก์ชันในภาษา C</u>

```
int square (int n) {
    return n*n;
}
```

<u>ฟังก์ชันในภาษา Ada</u>

```
function Square (N: Integer)
    return Integer is
begin
    return N*N;
end Square;
```

การมีโปรแกรมย่อยในลักษณะของฟังก์ชันให้ใช้งาน ช่วยทำให้การเขียนโปรแกรมในลักษณะเวียนบังเกิด (recursion) สามารถทำได้ ดังตัวอย่างต่อไปนี้

```
int factorial (int N)  {
    int result;
    if (N<3) result = N;
    else {
            result = factorial (N-1);
            result = N*result;
    }
    return result;
}
```

6.7 การจัดการกรณียกเว้น

(Exception handling)

การทำงานของโปรแกรมมีโอกาสที่จะเกิดเหตุผิดปกติได้มากมายเช่น การระบุอินเด็กซ์ของอาร์เรย์อาจจะเกินขอบเขตเนื้อที่ของอาร์เรย์, การคำนวณของนิพจน์คณิตศาสตร์อาจจะเกิดกรณีตัวหารเป็นค่าศูนย์, ฟังก์ชันที่คำนวณค่ารากที่สองอาจจะได้รับอาร์กิวเมนต์ที่เป็นค่าลบ, การจัดสรรเนื้อที่หน่วยความจำขณะรันโปรแกรมอาจเกิดกรณีเนื้อที่ว่างในหน่วยความจำมีไม่พอ การเขียนโปรแกรมให้สามารถจัดการกับกรณีผิดปกติทั้งหลายที่อาจเกิดขึ้นจะทำให้โปรแกรมนั้นมีความน่าเชื่อถือมากขึ้น ดังนั้นภาษาคอมพิวเตอร์ที่ดีจะต้องได้รับการออกแบบให้มีโครงสร้างคำสั่งที่ช่วยให้โปรแกรมเมอร์สามารถระบุประเภทของเหตุผิดปกติต่างๆ, สามารถสั่งการตรวจจับเหตุผิดปกติ และสามารถกำหนดชุดคำสั่งให้ทำงานเมื่อเกิดเหตุผิดปกติแต่ละประเภท (เรียกชุดคำสั่งนี้ว่า exception handler)

กรณียกเว้น (exceptions) หมายถึงเหตุผิดปกติทั้งหลายที่อาจเกิดขึ้นขณะรันโปรแกรม การเขียนโปรแกรมจะต้องแยกคำสั่งออกเป็นสองกลุ่ม คือชุดคำสั่งที่ทำงานในกรณีปกติ และชุดคำสั่งที่ทำงานเมื่อมีเหตุผิดปกติ พร้อมทั้งจำแนกประเภทของเหตุการณ์ผิดปกติ เพื่อส่งมอบการควบคุมไปยังชุดคำสั่งที่ถูกต้อง

ภาษา Ada เป็นภาษาคอมพิวเตอร์ภาษาแรกที่เริ่มกำหนดการจัดการกรณียกเว้นให้เป็นโครงสร้างส่วนหนึ่งของภาษา โดยกำหนดกรณียกเว้นไว้ 4 ประเภทได้แก่ Constraint_Error, Program_Error, Storage_Error, Tasking_Error และอนุญาตให้ผู้ใช้กำหนดกรณียกเว้นอื่นๆ ได้เพิ่มเติม รูปแบบการเขียนคำสั่งโดยมีการจัดการกรณียกเว้น แสดงตัวอย่างได้ดังนี้

```
begin - - this is a block with exception handlers
    … Statements …
    exception when Constraint_Error   =>  handler for exception Constraint_Error,
                                          which might be raised  by a division by zero;
    when others =>          handler for any other exception that is not Constraint_Error;
end;
```

ภาษา C++ และ Java ใช้คำสั่ง try … catch ในการทำงานเพื่อตรวจจับ และจัดการเมื่อมีกรณียกเว้นเกิดขึ้น โครงสร้างของคำสั่งแสดงได้ดังต่อไปนี้

```
try
    ... statements ...
    catch (exception_type_1)
            handler_1;
    ...
    catch (exception_type_n)
            handler_n;
    finally
            final_block;
```

ตัวอย่างต่อไปนี้แสดงคำสั่งในภาษา Java เพื่อรับจำนวนเลขที่ผู้ใช้พิมพ์ผ่านแป้นพิมพ์แล้วเปลี่ยนอักขระที่พิมพ์ให้อยู่ในรูปแบบของจำนวนเลขที่ถูกต้อง ถ้าอักขระที่พิมพ์ไม่ใช่จำนวนเลขจะถือเป็นกรณียกเว้น และการทำงานจะข้ามไปที่ส่วนจัดการกรณียกเว้น (คำสั่ง catch)

```
public static void main (String [ ] arg)  {
        BufferedReader in = new BufferedReader (new InputStreamReader (System.in));
        while (true)  {
                try {
                        System.out.print("Enter number:  ");
                        number = Integer.parseInt (in.realLine( ));
                        break;
                        }
                catch (NumberFormatException e) {
                        System.out.println ("Invalid number, please reenter.");
                        }
                catch (IO Exception e)  {
                        System.out.println ("Input error occurred, please reenter.");
                        }
                } //try
        } //while
} //main
```

6.8 สรุป

ในการออกแบบภาษาคอมพิวเตอร์ ผู้ออกแบบจะกำหนดรูปแบบและวิธีการทำงานของโครงสร้างคำสั่งต่างๆ ที่จะช่วยควบคุมลำดับการทำงานของโปรแกรม ลำดับการทำงานของโปรแกรมโดยทั่วไปจะมีอยู่ 3 ลักษณะ คือ การทำงานต่อเนื่อง (sequencing), การเลือกทำ (selection) และ การทำซ้ำ (iteration)

นอกจากโครงสร้างควบคุมลำดับการทำงานภายในโปรแกรมหรือภายในบล็อกของคำสั่งแล้ว ยังต้องมีโครงสร้างเพื่อการทำงานกับโปรแกรมย่อย รวมถึงการติดต่อระหว่างคำสั่งที่เรียกใช้โปรแกรมย่อยกับตัวโปรแกรมย่อย แนวคิดของการมีโปรแกรมย่อยก็เพื่อสนับสนุนการเขียนโปรแกรมอย่างมีโครงสร้าง นั่นคือโปรแกรมควรแบ่งออกเป็นส่วนย่อย แต่ละส่วนย่อยมีวัตถุประสงค์เพื่อการทำงานที่ชัดเจนเพียงอย่างเดียว การมีโครงสร้างที่ดีจะทำให้โปรแกรมมีระเบียบ อ่านได้ง่าย แต่การจัดโครงสร้างที่ดีเพียงอย่างเดียวยังไม่เพียงพอต่อการทำให้โปรแกรมน่าเชื่อถือและสามารถทนทานต่อเหตุผิดปกติต่างๆ ภาษาคอมพิวเตอร์ที่ดีจะต้องมีโครงสร้างที่สนับสนุนการตรวจจับและการจัดการกับกรณีผิดปกติต่างๆ ได้อย่างมีประสิทธิภาพ

แบบฝึกหัดท้ายบทที่ 6

คำถามปรนัย: ให้เลือกคำตอบที่ถูกต้องที่สุด

1. คุณสมบัติของภาษาคอมพิวเตอร์ในข้อใดที่ช่วยเพิ่มความถูกต้องให้กับการทำงานของโปรแกรม ?
   - ก. exception handling และ orthogonality
   - ข. exception handling และ type checking
   - ค. type checking และ overloading
   - ง. type coercion และ orthogonality

2. ลักษณะใดถือเป็นลักษณะของภาษาคอมพิวเตอร์ที่ไม่ดี ?
   - ก. การมีตัวแปร boolean และการมีคำสั่ง goto
   - ข. การมี exception และการทำ aliasing
   - ค. การมีคำสั่ง goto และการทำ aliasing
   - ง. การมี exception และการทำ overloading

3. ภาษาคอมพิวเตอร์ที่ดีจะต้องถูกออกแบบให้สามารถตรวจสอบความถูกต้องของชนิดข้อมูลได้ ความสามารถเช่นนี้เรียกว่าอะไร ?
   - ก. parameter checking
   - ข. type checking
   - ค. orthogonality
   - ง. functionality

4. การใช้คำสั่ง A + fun(A) จะก่อให้เกิดปัญหาได้ในกรณีใด ?
   - ก. เมื่อชนิดข้อมูลของ A และ fun(A) ไม่ตรงกัน
   - ข. เมื่อ fun(A) เปลี่ยนค่าของ A
   - ค. เมื่อ fun(A) ทำงานก่อนที่จะทำการบวก
   - ง. เมื่อ fun(A) ส่งข้อมูลกลับเป็นค่าเดียวกับ A

5. การใช้เครื่องหมาย + ให้ทำหน้าที่ทั้งการบวกและการเชื่อมต่อข้อความ เรียกการกระทำหลายหน้าที่นี้ว่าอะไร ?
   - ก. mixed mode arithmetic
   - ข. operator polymorphism
   - ค. global side effect
   - ง. operator overloading

6. พารามิเตอร์ที่กำหนดไว้ที่ส่วนหัวของฟังก์ชัน เรียกว่าอะไร ?
   - ก. prototype
   - ข. header parameter
   - ค. formal parameter
   - ง. actual parameter

7. พารามิเตอร์ที่กำหนดไว้ที่ส่วนหัวของฟังก์ชัน เรียกว่าอะไร ?
   - ก. prototype
   - ข. header parameter
   - ค. formal parameter
   - ง. actual parameter

8. พารามิเตอร์ที่ปรากฏในคำสั่งเรียกใช้ฟังก์ชัน เรียกว่าอะไร ?

ก. prototype parameter ข. header parameter

ค. formal parameter ง. actual parameter

9. ฟังก์ชันในภาษา C และ C++ สามารถทำให้เป็น procedure เหมือนในภาษา Pascal ได้อย่างไร ?

ก. โดยการเรียกใช้ฟังก์ชันและฟังก์ชันที่เรียกไม่ต้องส่งค่าพารามิเตอร์มาให้

ข. โดยการเขียนฟังก์ชันและกำหนดให้ส่งค่ากลับเป็น void

ค. โดยการแทรกฟังก์ชันไว้ใน expression

ง. โดยการเขียนฟังก์ชันไว้ทางขวามือของเครื่องหมาย =

10. ถ้าพารามิเตอร์ถูกส่งด้วยวิธี pass-by-value จะมีการทำงานใดเกิดขึ้น ?

ก. actual parameter จะถูกเปลี่ยนชื่อให้ตรงกับตัวแปรในฟังก์ชัน

ข. actual parameter จะถูกเปลี่ยนเป็นตัวชี้ตำแหน่งในหน่วยความจำ

ค. formal parameter จะทำหน้าที่เป็นตัวแปร local ในฟังก์ชัน

ง. formal parameter จะถูกเปลี่ยนชื่อให้ตรงกับ actual parameter

11. ข้อเสียของวิธีการส่งผ่านค่าพารามิเตอร์แบบ pass-by-value คืออะไร ?

ก. ต้องเสียเนื้อที่หน่วยความจำเพื่อเก็บค่า formal parameter

ข. ต้องเสียเนื้อที่หน่วยความจำเพื่อเก็บค่า actual parameter

ค. ไม่สามารถตรวจสอบชนิดของตัวแปรได้

ง. ไม่สามารถทำ recursive ได้

12. การส่งผ่านพารามิเตอร์แบบใดที่เป็นการส่งค่าตำแหน่งในหน่วยความจำ ?

ก. pass-by-reference ข. pass-by-value

ค. pass-by-value-result ง. pass-by-name

กำหนดซูโดโค้ดที่มีไวยากรณ์คล้ายภาษา C ให้ดังต่อไปนี้ (ใช้ตอบคำถามข้อ 13-15)

```
void  f(int x, int y)  { x = 1; y = 3; }
void  main()  {
      int  i, a[2];
      i = 0;  a[0] = 0;  a[1] = 0;
     f(i, a[i]);
     printf(" %d  %d  %d \n", i , a[0], a[1] );
   }
```

13. ถ้าการส่งผ่านค่าระหว่างฟังก์ชัน main() และฟังก์ชัน f() เป็นแบบ pass-by-reference ผลลัพธ์ที่ได้จากคำสั่ง printf() คือค่าใด ?

ก. 0 0 0 ข. 1 0 3

ค. 1 3 0 ง. 0 1 3

14. ถ้าการส่งผ่านค่าระหว่างฟังก์ชัน main() และฟังก์ชัน f() เป็นแบบ pass-by-name ผลลัพธ์ที่ได้จากคำสั่ง printf() คือค่าใด ?

    ก. 0 0 0 ข. 1 0 3

    ค. 1 3 0 ง. 0 1 3

15. ถ้าการส่งผ่านค่าระหว่างฟังก์ชัน main() และฟังก์ชัน f() เป็นแบบ pass-by-value ผลลัพธ์ที่ได้จากคำสั่ง printf() คือค่าใด ?

    ก. 0 0 0 ข. 1 0 3

    ค. 1 3 0 ง. 0 1 3

16. ข้อใดเป็นลักษณะสำคัญของภาษาในกลุ่ม imperative?

    ก. การใช้คำสั่ง package, exception handling และ switch

    ข. การใช้คำสั่ง assignment, selection และ iteration

    ค. การใช้คำสั่ง assignment, exception handling และ garbage collection

    ง. การใช้คำสั่ง assignment, recursive และ selection

17. ข้อใดเป็นลักษณะเด่นของภาษาคอมพิวเตอร์ในกลุ่ม procedural ?

    ก. มีลักษณะ orthogonality สูง แต่มี reliability ต่ำ

    ข. มีการใช้ตัวแปร การใช้คำสั่งกำหนดค่า และการทำงานซ้ำ

    ค. มีการเรียกตัวเองซ้ำ (recursion) การทำ aliasing และการเรียกใช้ฟังก์ชัน

    ง. มีคำสั่ง goto คำสั่ง switch และการใช้ pointer