หัวข้อในบทที่ 2

- แบบข้อมูลสำหรับเลขจำนวนเต็ม
- แบบข้อมูลสำหรับเลขทศนิยม

2

แบบของข้อมูล

ในบทนี้เราจะมาทำความรู้จักกับแบบหรือชนิดของข้อมูลที่เราใช้ในภาษาซี เริ่มต้นตั้งแต่ชนิดของข้อมูลพื้นฐานไปจนถึงโครงสร้างของข้อมูลที่ซับซ้อน เพราะว่าการเลือกชนิดของข้อมูลให้เหมาะสมกับงานและปัญหาย่อมมีผลโดยตรงต่อประสิทธิภาพ รวมถึงความถูกต้องในการทำงานของโปรแกรมด้วย เวลาเราเขียนโปรแกรมคอมพิวเตอร์ เราก็มักจะเผชิญกับคำถามที่ว่า เราจะใช้แบบข้อมูลแบบไหนในการเก็บข้อมูลต่างๆ และควรจะเลือกแบบใดจึงจะเหมาะสม ยกตัวอย่างเช่น การหารเลขจำนวนเต็มสองจำนวน ผลลัพธ์จะเป็นได้ทั้งจำนวนเต็มคือตัวเลขทั้งสองสามารถหารกันได้ลงตัวหรือผลหารอาจจะเป็นเลขที่ไม่ใช่เลขจำนวนเต็มก็ได้ ถ้าเราเลือกตัวแปรที่เป็นข้อมูลแบบจำนวนเต็มสำหรับเก็บผลลัพธ์จากการหาร ก็สามารถเก็บค่าได้เฉพาะจำนวนเต็มเท่านั้น ซึ่งไม่สามารถใช้ได้กับผลลัพธ์ที่ไม่ใช่เลขจำนวนเต็มหรือเลขทศนิยมได้ เพราะฉะนั้นเราก็ต้องเลือกตัวแปรแบบ `float` หรือ `double` เท่านั้นจึงจะเหมาะสม (ข้อมูลแบบ `float` และ `double` มีลักษณะอย่างไรเราจะได้เรียนรู้ต่อไป)

ในบางครั้งเราก็จำเป็นต้องรู้ว่าขอบเขตของข้อมูลที่เราใช้นั้นอยู่ในช่วงใด ค่าที่มากที่สุดและน้อยที่สุดสำหรับข้อมูลแบบนี้มีค่าเท่าใด เช่น เราต้องการเก็บค่าตัวเลข 32768 โดยใช้ตัวแปรแบบ `int` ผลที่ตามมาก็คือตัวแปรนี้เก็บค่าที่เราต้องการบันทึกเอาไว้ไม่ถูกต้อง เพราะในเครื่องคอมพิวเตอร์ที่มีขนาดความยาว ของเวิร์ด (WORD) เท่ากับ 16 บิต (ในกรณีนี้ หนึ่งเวิร์ดมีค่าเท่ากับสองไบต์หรือสิบหกบิต) ค่าของตัวแปรจึงมีค่าอยู่ระหว่าง -32768 และ 32767 เท่านั้น เราลองดูตัวอย่างที่แสดงให้เห็นความสำคัญของปัญหานี้

```
#include <stdio.h>

int main()
{
      printf ("%d %d %d\n", 32767, 32767+1, 32767+2);
      return 0;
}
```

ผลของโปรแกรมนี้ก็คือ

```
32767 -32768 -32767
```

แทนที่จะเป็น

```
32767 32768 32769
```

ตามที่เราคาดหวังเอาไว้ เหตุที่เป็นเช่นนี้ก็เพราะว่าค่าของข้อมูล (32768 และ 32769) มีค่าเกินขอบเขตของข้อมูลแบบ `int`

ข้อสังเกต ผู้อ่านบางท่านอาจจะมีความสงสัยหรือไม่เข้าใจการใช้งานฟังก์ชันมาตราฐาน `printf()` ในโปรแกรมตัวอย่าง ก็จะขออธิบายในตอนต้นนี้เพียงคร่าวๆก่อน โปรดสังเกตว่าพารามิเตอร์ตัวแรกของฟังก์ชัน `printf()` จะต้องเป็นข้อความ โดยที่เราสามารถสังเกตได้จากสัญลักษณ์ " ที่อยู่หัวและท้ายเป็นตัวกำหนดขอบเขตของข้อความในภาษาซี ภายในข้อความนี้อาจจะมีกลุ่มของตัวอักขระที่เรียกว่า ลำดับควบคุม (Control Sequence) มีสัญลักษณ์ % นำหน้าตัวอักษร เช่น `%d` ซึ่งจะใช้แทนค่าของพารามิเตอร์ตัวถัดไป หรืออาจจะเรียกได้ว่าเป็นตัวจองที่ในตำแหน่งที่เราต้องการภายในข้อความ เช่น

```
printf ("%d\n", 32767);
```

โดยที่ `%d` จองที่ไว้สำหรับตัวเลข `32767` ซึ่งเป็นพารามิเตอร์ตัวที่สองของฟังก์ชัน และคำสั่งนี้จะให้ผลที่แสดงออกทางจอภาพเหมือนกับคำสั่งต่อไปนี้ซึ่งไม่มีการใช้ลำดับควบคุมใดๆ ดังนั้นจึงมีเพียงพารามิเตอร์ตัวเดียวเท่านั้น

```
printf ("32767\n");
```

แต่สำหรับกรณีทั่วๆไปแล้วเราจำเป็นต้องใช้ลำดับควบคุมแทนที่ค่าของข้อมูลต่างๆ เช่น ค่าของข้อมูลแบบ `int` ที่เรายังไม่ทราบแน่ชัดหรือได้กำหนดไว้อย่างเจาะจง เช่น ค่าของตัวแปรหรือค่าจากฟังก์ชันอื่นๆ ตัวอย่างเช่น

```
printf ("x + y = %d\n", add(x,y));
```

`%d` จองที่ไว้สำหรับค่าของฟังก์ชัน `add()`ซึ่งเป็นผลรวมของตัวแปร `x` และ `y` ส่วนสัญลักษณ์ `\n` จัดเป็นพวก Escape Sequence ส่งผลให้ฟังก์ชัน `printf()` เมื่อทำงานจะขึ้นบรรทัดใหม่ก่อนจะพิมพ์ข้อความใดๆต่อไปถ้ามี เช่น ในกรณีที่เราใช้ฟังก์ชัน `printf()` มากกว่าหนึ่งครั้งตลอดการทำงานของโปรแกรม ดังนั้นจุดประสงค์ของการใช้ `\n` ซึ่งอยู่ตำแหน่งท้ายสุด (ก่อนสัญลักษณ์ ") ในข้อความที่เป็นอาร์กิวเมนต์แรกของฟังก์ชันก็คือการขึ้นบรรทัดใหม่หลังจากที่พิมพ์ข้อความแล้วเหมือนเวลาที่เรากดคีย์ ENTER บนแป้นพิมพ์

เราได้พิจารณาตัวอย่างที่แสดงให้เห็นความสำคัญในการตัดสินใจเลือกแบบหรือชนิดของข้อมูลที่เราจะใช้กับตัวแปรหรือค่าคงที่ต่างๆไปแล้ว ตอนนี้เราลองมาทำความรู้จักกับแบบข้อมูลพื้นฐานในภาษาซี

แบบข้อมูล	คำอธิบาย	หน่วยความจำที่ใช้
`short int`	จำนวนเต็มแบบสั้น	สองไบต์ (16 บิต)
`long int`	จำนวนเต็มแบบยาว	สี่ไบต์ (32 บิต)
`float`	จำนวนจริงที่มีจุดทศนิยม	สี่ไบต์ (32 บิต)
`double`	เหมือนกับ `float` แต่สามารถเก็บค่าได้มากกว่า	แปดไบต์ (64 บิต)
`long double`	เหมือนกับ `float` และ `double` แต่สามารถเก็บค่าได้มากกว่า	สิบไบต์ หรือมากกว่า
`char`	ตัวอักขระ	หนึ่งไบต์ (8 บิต)

ตารางที่ 2.1 แบบข้อมูลพื้นฐานในภาษาซี

2.1 แบบข้อมูลสำหรับเลขจำนวนเต็ม (Integer Data Type)

2.1.1 จำนวนเต็มแบบ `int`

เราใช้ตัวแปรแบบ `int` ในการเก็บค่าของตัวเลขจำนวนเต็มที่มีค่าอยู่ระหว่าง `-32768` และ `32767` คือมีทั้งค่าบวกและค่าลบ (ตามปรกติแล้วข้อมูลแบบ `int` จะหมายถึง `signed int`) ขนาดของตัวแปร แบบ `int` จะมีค่าเท่ากับ 16 บิต หรือสองไบต์สำหรับเครื่องคอมพิวเตอร์แบบพีซีทั่วๆไป ส่วนเครื่องคอมพิวเตอร์ที่มีขนาดของ `int` เท่ากับ 32 บิต (หรืออาจจะมากกว่านี้ก็ได้ เช่น 36 บิต) ก็จะสามารถเก็บค่าของ ตัวเลขจำนวนเต็มที่อยู่ระหว่าง -2^{31} และ $2^{31}-1$ ได้ ค่าคงที่ใดๆที่เป็นจำนวนเต็มและพบอยู่ในโปรแกรมโค้ด เราจะถือว่าเป็นค่าคงที่แบบ `int` (หรือ `signed int`) โดยอัตโนมัติ ถ้าเราต้องการใช้ตัวแปรใดๆแบบ `int` ในโปรแกรมโค้ด เราจะต้องแจ้งการใช้ตัวแปรก่อนโดยมีรูปแบบต่อไปนี้

```
int   variable_name;
```

ตัวอย่างเช่น

```
int  prime_number;
int  Sum;
int  i, j;
```

เราสามารถทราบขนาดของหน่วยความจำสำหรับข้อมูลแบบ int ได้ว่ามีค่าเท่าไร โดยใช้โอเปอร์-เรเตอร์ที่มีชื่อว่า sizeof ตามตัวอย่างข้างล่างนี้

```
#include <stdio.h>

int main()
{
      int x;

      x = -10000;
      printf ("Integer size = %d bits\n", sizeof(x)*8);
      return 0;
}
```

โอเปอร์เรเตอร์ sizeof จะให้ค่าที่เท่ากับความยาวของหน่วยความจำของ x ที่เป็นตัวแปรแบบ int ซึ่งมีหน่วยเป็นไบต์ เมื่อเราคูณด้วยแปดดังตัวอย่างข้างบนก็จะเป็นขนาดของ int ในหน่วยบิต และในตัวอย่างนี้เราจะเห็นได้ว่าตัวเลข -10000 เป็นค่าคงที่แบบ int

ตามที่เราทราบกันข้อมูลที่เก็บในหน่วยความจำของคอมพิวเตอร์จะอยู่ในรูปของบิต(Bit) หรือดิจิต (Digit) คือเลขโดด 0 หรือ 1 การแสดงจำนวนตัวเลขต่างๆก็ต้องแสดงให้อยู่ในรูปของเลขฐานสอง หรือเรียกว่า Binary Number เช่น หนึ่งไบต์ประกอบด้วยแปดบิต ซึ่งเราสามารถใช้แทนตัวเลขจำนวนเต็มที่มี ค่าอยู่ระหว่าง 0 ถึง 255 โปรดสังเกตว่าขอบเขตของข้อมูลแบบหนึ่งๆจะขึ้นอยู่กับขนาดของหน่วยความ จำที่ใช้ เช่น สมมุติว่าเราใช้หน่วยความจำทั้งหมด N บิตในการเก็บค่าของจำนวนเต็มใดๆ เราก็สามารถใช้ เก็บค่าของตัวเลขจำนวนเต็มค่าใดค่าหนึ่งบนเส้นจำนวนจากทั้งหมด 2^N จำนวน ตัวอย่างเช่น ถ้า N มีค่า เท่ากับ 16 เราก็สามารถเขียนช่วงของเลขจำนวนเต็มบนเส้นจำนวนได้ดังนี้

$$Offset \le i \le Offset + 2^N - 1, \quad N = 16$$

i	เป็นเลขจำนวนเต็มใดๆที่เราสามารถเก็บค่าไว้ในหน่วยความจำขนาด 16 บิตได้
Offset	หมายถึงจำนวนเต็มใดๆที่เรากำหนดให้เป็นจำนวนเต็มที่มีค่าน้อยที่สุดของข้อมูลที่เราต้องการเก็บไว้ในหน่วยความจำนี้

ตามปรกติแล้ว สำหรับ `N = 16` ค่าของ `Offset` จะมีค่าเท่ากับ -32768 (= -2^{15}) หรือ 0 ค่าใดค่าหนึ่ง ตารางข้างล่างเป็นตัวอย่างของการเขียนตัวเลขฐานสิบตั้งแต่ 0 ถึง 15 ให้อยู่ในระบบเลขฐานสอง

0	0000 0000	8	0000 1000
1	0000 0001	9	0000 1001
2	0000 0010	10	0000 1010
3	0000 0011	11	0000 1011
4	0000 0100	12	0000 1100
5	0000 0101	13	0000 1101
6	0000 0110	14	0000 1110
7	0000 0111	15	0000 1111

ตารางที่ 2.2 ตัวอย่างการแสดงเลขในฐานสองและฐานสิบ

ขนาดของ `int` ปรกติแล้วจะมีค่าเท่ากับ 16 บิต และมีบิตตัวที่อยู่ริมสุดทางซ้ายมือทำหน้าที่บ่ง บอกว่า จำนวนตัวเลขฐานสองนี้ใช้แทนตัวเลขที่มีค่าบวกหรือลบในเลขฐานสิบ เราเรียกบิตตัวนี้ว่า Sign Bit เช่น 32767 เมื่อเขียนให้อยู่ในฐานสองก็จะมีเลขหนึ่งสิบห้าตัวและมีเลขศูนย์หนึ่งตัวอยู่ตำแหน่งที่สิบ หก (เริ่มนับจากขวาไปซ้าย)

```
32767       0111 1111 1111 1111
```

เมื่อเราบวก 32767 ด้วยหนึ่ง ค่าที่ควรจะเป็นก็คือ 32768

```
32768       1000 0000 0000 0000
```

แต่สำหรับตัวเลขที่เป็นข้อมูลแบบ `int` เลขฐานสองในบรรทัดข้างบนจะหมายถึงค่า -32768 โดยมีหลัก การคำนวณดังนี้ ถ้าบิตซึ่งอยู่ทางซ้ายมือสุดของเลขฐานสองของตัวเลขแบบ `int` เป็นเลขศูนย์ ก็ให้คำนวณ เหมือนที่เราเปลี่ยนเลขฐานสองให้เป็นเลขฐานสิบตามปรกติ แต่ถ้ามีค่าเป็นหนึ่งก็ให้นับผลรวมค่าของบิต ตามปรกติจากบิตตำแหน่งที่หนึ่งไปจนถึงตำแหน่งที่สิบห้า(ยกเว้นตัวที่สิบหก) รวมแล้วก็คือค่าของเลขฐาน สองของบิตสิบห้าตัวแรก (นับจากขวาไปซ้าย) แล้วลบออกด้วย 32768 เช่น

```
1000 0000 0000 0000              0 - 32768 = -32768
1000 0000 0000 0001              1 - 32768 = -32767
1111 1111 1111 1111          32767 - 32768 = -1
```

นอกจากเลขฐานสองแล้ว เราสามารถเขียนเลขจำนวนเต็มให้อยู่ในเลขฐานอื่นๆได้ เช่น ฐานแปด หรือฐาน สิบหก ค่าคงที่แบบ `int` เราสามารถใช้ในภาษาซีได้โดยเขียนอยู่ในระบบเลขฐานสิบ ฐานแปด หรือฐาน สิบหกเท่านั้น โดยมีสัญลักษณ์กำกับไว้ข้างหน้าเพื่อบอกให้คอมไพล์เลอร์ทราบ

ว่า ตัวเลขที่ตามมาเป็นเลข ฐานอะไร (โปรดสังเกตว่า ในภาษาซีเราไม่สามารถเขียนค่าคงที่ให้อยู่ในระบบเลขฐานสองได้) ตัวอย่าง เช่น

165_{10} 245_{8} $A5_{16}$

$245_{8} = (2\times8^{2}) + (4\times8^{1}) + (5\times8^{0})$
$A5_{16} = (10\times16^{1}) + (5\times16^{0})$

4213_{10} 10165_{8} 1075_{16}

$10165_{8} = (1\times8^{4}) + (0\times8^{3}) + (1\times8^{2}) + (6\times8^{1}) + (5\times8^{0})$
$1075_{16} = (1\times16^{3}) + (0\times16^{2}) + (7\times16^{1}) + (5\times16^{0})$

ในภาษาซีเราจะต้องเขียนตัวเลขเหล่านี้ที่เป็นค่าคงที่ให้อยู่ในรูปแบบตามตัวอย่างดังนี้

```
0245            0XA5            0xa5
010165          0X1075          0x1075
```

ถ้าตัวเลขที่เป็นค่าคงที่เริ่มต้นด้วยเลขศูนย์แล้วตัวเลขถัดไปจะต้องเป็นเลขฐานแปด โปรดสังเกตว่าตัวเลข แต่ละตัวของระบบเลขฐานแปดจะเป็นเลข 0 ถึง 7 เท่านั้น ตัวอย่างที่ผิด เช่น

```
0781        008L        02af3
```

สำหรับค่าคงที่ใดๆที่เริ่มต้นด้วย `0x` หรือ `0X` ตัวเลขที่ตามมาจะต้องเป็นเลขฐานสิบหกเท่านั้น (0,1,2,..,9,A, B,C,D,E,F) แต่มีความแตกต่างระหว่าง `0x` และ `0X` อยู่ตรงที่เวลาเราใช้ `0x` (สังเกตว่า `x` เป็นตัวพิมพ์เล็ก) ถ้าตัวเลขฐานสิบหกตัวใดมีค่าตั้งแต่สิบจนถึงสิบห้าแล้วสัญลักษณ์ของเลขเหล่านี้จะต้องใช้ตัวพิมพ์เล็ก

```
a=10  b=11  c=12  d=13  e=14  f=15
```

แต่ถ้าใช้ `0X` ดังนั้นสัญลักษณ์ของเลขเหล่านี้จะต้องเขียนด้วยตัวพิมพ์ใหญ่

```
A=10  B=11  C=12  D=13  E=14  F=15
```

2.1.2 จำนวนเต็มแบบยาว (Long Integer)

จำนวนเต็มแบบยาวหมายถึง การนำตัวเลขแบบ `int` ที่มีขนาด 16 บิต สองตัวมาต่อเข้าด้วยกันทำให้มีความยาวเพิ่มขึ้นเป็น 32 บิต ตัวแปรแบบ `long int` หรือเรียกสั้นๆว่า `long` สามารถแทนตัวเลขที่มีค่าอยู่ระหว่าง -2^{31} = -2,147,483,648 และ $2^{31}- 1$ = 2,147,483,647 สำหรับค่าคงที่ใดๆที่เราต้องการให้เป็นเลขจำนวนเต็มแบบยาวหรือมีขนาด 32 บิต เราก็เติมตัว `L` ต่อท้ายเลขนั้น (จะเป็นตัวพิมพ์เล็กก็ได้ แต่ควรใช้แต่ตัวพิมพ์ใหญ่เท่านั้น) เช่น

```
0L
123L
1802132L
0177L
0x7fL
```

ตัวอย่างต่อไปนี้เป็นตัวอย่างการใช้ข้อมูลแบบ long int

```
#include <stdio.h>

int main()
{
      long int x;

      x = 32767L;
      x++;
      printf ("x = %d\n", x);
      printf ("32767 + 1 = %d\n", 32767L + 1L);
      return 0;
}
```

ผลของโปรแกรมบนจอภาพก็คือ

```
x = 32768
32767 + 1 = 32768
```

ซึ่งให้ผลลัพธ์ถูกต้องตามที่ต้องการ

ในบางครั้งเราต้องการกำหนดใช้ค่าคงที่แบบ long int แต่เราลืมเติมตัว L ต่อท้าย เช่น 10000000 เมื่อทำการคอมไพล์โปรแกรมโค้ด คอมไพล์เลอร์ก็จะเตือนให้เราทราบว่า ค่าคงที่นี้มีค่ามากเกินขอบเขตของข้อมูลแบบ int (10000000 มีค่าเกิน 32767) เพราะถ้าเราไม่เติมตัว L ข้างหลังตัวเลขจำนวนเต็มใดๆ คอมไพล์เลอร์ก็จะถือว่าตัวเลขจำนวนเต็มนี้เป็นข้อมูลแบบ int โดยอัตโนมัติ ขอให้ผู้อ่านทดลองคอมไพล์โปรแกรมต่อไปนี้ แล้วดูว่าคอมไพล์เลอร์จะแจ้งคำเตือนใดให้ทราบ

```
#include <stdio.h>

int main()
{
      printf ("Integer Constant : %u\n", 10000000);
      return 0;
}
```

2.1.3 จำนวนเต็มแบบสั้น (Short Integer)

จำนวนเต็มแบบสั้น `short int` หรือเรียกสั้นๆว่า `short` เป็นการกำหนดขนาดของจำนวนเต็มที่มีความยาวที่แน่นอนคือ 16 บิต โดยไม่ขึ้นกับความยาวของเวิร์ดของเครื่องคอมพิวเตอร์ว่าจะมีค่าเป็นเท่าใด สำหรับคอมพิวเตอร์ที่มีค่าของเวิร์ดเท่ากับ 16 บิตจะไม่มีความแตกต่างระหว่าง `short int` และ `int`

เราสามารถสรุปได้ว่า การจำแนกข้อมูลแบบ `int` ออกเป็น `short` และ `long` ก็เพื่อจะให้แน่ใจว่าข้อมูลแบบ `int` ที่เราต้องการใช้มีขนาดที่แน่นอน ไม่ขึ้นกับความยาวของเวิร์ดของคอมพิวเตอร์แต่ละระบบ ถ้าเราใช้คำว่า `int` เท่านั้น ก็อาจเกิดปัญหาได้เมื่อเราใช้โปรแกรมบนเครื่องที่มีการจัดการของหน่วยความจำที่แตกต่างกันออกไป สำหรับขอบเขตของข้อมูลจำนวนเต็มแบบต่างๆที่คอมไพล์เลอร์ทราบจะถูกนิยามไว้ในไฟล์ชื่อ `<limits.h>` โปรดสังเกตว่า ขนาดของหน่วยความจำแบบ `int` ในตารางข้างล่างนี้มีค่าเท่า กับ 16 บิต

ตัวระบุชื่อ	เลขฐานสิบ	เลขฐานสิบหก	คำอธิบาย
`INT_MIN`	`(-32768)`	`0x7FFF`	ค่าน้อยที่สุดของ `int`
`INT_MAX`	`32767`	`((int)0x8000)`	ค่ามากที่สุดของ `int`
`UINT_MAX`	`65535U`	`0xFFFFU`	ค่ามากที่สุดของ `unsigned int`
`LONG_MIN`	`(-2147483648L)`	`0x7FFFFFFFL`	ค่าน้อยที่สุดของ `long`
`LONG_MAX`	`2147483647L`	`((long)0x80000000L)`	ค่ามากที่สุดของ `long`
`ULONG_MAX`	`4294967295UL`	`0xFFFFFFFFUL`	ค่ามากที่สุดของ `unsigned long`

ตารางที่ 2.3 ขอบเขตของข้อมูลที่เป็นจำนวนเต็มแบบต่างๆ สำหรับเครื่องคอมพิวเตอร์แบบพีซี

2.1.4 จำนวนเต็มที่มากกว่าหรือเท่ากับศูนย์ (Unsigned Integer)

ในบางครั้งเราต้องการใช้ตัวแปรที่มีค่ามากกว่าหรือเท่ากับศูนย์เท่านั้น เช่น ในการนับจำนวนของข้อมูลที่เรามีอยู่ ดังนั้นจึงใช้ตัวเลขที่เป็นจำนวนนับเท่านั้น ซึ่งก็คือจำนวนเต็มที่มากกว่าหรือเท่ากับศูนย์ แต่เนื่องจากว่าจำนวนเต็มแบบ `int` ไม่ว่าจะเป็นแบบสั้นหรือยาวก็ตาม เป็นจำนวนเต็มที่สามารถเป็นได้ทั้งค่าบวกและค่าลบ แต่เราไม่ต้องการใช้จำนวนเต็มที่มีค่าเป็นลบ ถ้าเราใช้ข้อมูลแบบ `int` ธรรมดา เราสามารถใช้ตัวแปรแบบ `int` เก็บค่าของจำนวนนับที่มีค่าอยู่

ระหว่าง 0 และ 32767 (= 2^{15}- 1) เท่านั้นสำหรับจำนวนเต็มแบบสั้นขนาด 16 บิต ในขณะที่ข้อมูลแบบ `int` สามารถใช้แทนค่าใดค่าหนึ่งได้ทั้งหมด 2^{16} = 65536 จำนวนที่แตกต่างกัน ทำให้เราใช้ขอบเขตของข้อมูลแบบ `int` ได้เพียงครึ่งหนึ่งเท่านั้น และเพื่อแก้ไขปัญหานี้เราจะต้องเติมคำว่า `unsigned` ไว้ข้างหน้า `int` (`short` หรือ `long`) ตามรูปแบบข้างล่างนี้

```
unsigned short int          0 ... 2^16-1
unsigned long  int          0 ... 2^32-1
unsigned int
```

หรือเขียนสั้นๆได้ว่า

```
unsigned short
unsigned long
unsigned
```

และเพื่อเป็นการกำหนดแบบข้อมูลขึ้นมาใหม่ที่มีขนาดของหน่วยความจำเท่ากับขนาดของข้อมูลแบบ `int` ธรรมดา แต่สามารถเก็บค่าที่อยู่ในช่วง 0 ถึง 65535 (แทนที่จะเป็น -32768 ถึง 32767) สำหรับจำนวนเต็ม แบบสั้นและ 0 ถึง 4294967295 สำหรับจำนวนเต็มแบบยาวได้ สำหรับค่าคงที่จำนวนเต็มใดๆที่เราต้อง การกำหนดให้เป็นข้อมูลแบบ `unsigned int` เราก็เติมตัว `U` ต่อท้าย เช่น

```
32768U
60000U
0x8f00U
1234567UL
```

2.1.5 การกำหนดค่าของตัวแปรแบบ `int`

ดังที่เคยกล่าวไปแล้วในตอนต้นว่าเราสามารถใช้โอเปอร์เรเตอร์หรือตัวปฏิบัติการที่เรียกว่า Assignment Operator ซึ่งมีสัญลักษณ์เป็นเครื่องหมายเท่ากับและใช้ในการเปลี่ยนแปลงหรือกำหนดค่าของตัวแปร โดยมีหลักการใช้ดังต่อไปนี้ ทางซ้ายมือของเครื่องหมายเท่ากับจะเป็นชื่อของตัวแปรที่เราต้องการจะกำหนดค่าใหม่ให้ตัวแปรตัวนี้เก็บไว้ ส่วนทางขวามือของเครื่องหมายเท่ากับจะเป็นนิพจน์ค่าคงที่ หรือตัวแปรใดๆก็ได้ที่ใช้แทนค่าของข้อมูลแบบเดียวกัน เช่น `x` และ `y` เป็นชื่อของตัวแปรสองตัวแบบ `int` และสมมุติว่า ค่าของตัวแปร `y` ในขณะนั้นมีค่าเท่ากับ `1` ดังนั้นถ้าเราเขียนประโยคคำสั่งต่อไปนี้

```
x = y;
```

ก็จะหมายความว่า ตัวแปร `x` จะมีค่าใหม่เป็น `1` ซึ่งเท่ากับค่าของ `y` นั้นเอง หรือถ้าเราเขียนประโยคคำสั่ง ใหม่เป็น

```
x = (x + y) * 3;
```

โดยเราสมมุติว่า ตอนแรก x มีค่าเท่ากับ 1 และ y มีค่าเป็น -1 ผลก็คือว่า ตัวแปร x ที่อยู่ทางซ้ายมือของ เครื่องหมายเท่ากับจะมีค่าใหม่เป็นศูนย์ ซึ่งคำนวณได้ดังนี้

```
x = (1 + (-1)) · 3
  =  0
```

นอกจากนี้เรายังสามารถผ่านค่าของฟังก์ชันใดๆให้ตัวแปรเก็บไว้ก็ได้ เพียงแต่มีข้อแม้อยู่ว่า ค่าที่เราได้จากฟังก์ชันจะต้องเป็นแบบเดียวกันกับชนิดของข้อมูลที่เก็บไว้ในตัวแปร เช่น ถ้าเป็นตัวแปรแบบ int เราจะต้องผ่านค่าของฟังก์ชันที่เป็นแบบ int เท่านั้นให้กับตัวแปร ตัวอย่างเช่น

```
x = add (1,5);
```

เพราะฟังก์ชัน add ที่เราได้ทำความรู้จักมาแล้วเป็นฟังก์ชันที่ให้ค่าแบบ int ซึ่งเป็นผลรวมระหว่าง 1 และ 5 ดังนั้น ตัวแปร x จึงมีค่าใหม่เท่ากับ 6 ตัวอย่างที่ไม่ถูกต้อง เช่น

```
x = 1.234 + 4;
```

เนื่องจากตัวเลข 1.234 เป็นเลขจำนวนจริงที่ไม่ใช่จำนวนเต็ม และไม่จัดเป็นข้อมูลแบบ int ในขณะที่เรา ได้กำหนดไว้ว่า x เป็นตัวแบบ int ประโยคคำสั่งนี้จึงผิดจุดประสงค์ของการใช้ แต่ไม่ผิดหลักไวยกรณ์ในภาษาซี เพราะเราจะได้เรียนรู้ต่อไปว่า ในภาษาซีมีการแปลงแบบของข้อมูลโดยอัตโนมัติในบางกรณี สำหรับกรณีตัวอย่างนี้ ตัวเลข 1.234 นี้เป็นค่าคงที่แบบ double เมื่อบวกด้วยตัวเลข 4 ซึ่งเป็นค่าคงที่แบบ int แต่จะถูกแปลงแบบเป็น double โดยอัตโนมัติตามกฎเกณฑ์ที่กำหนดไว้ในภาษาซีและได้ผลลัพธ์จากโอเปอร์เรชันการบวกที่อยู่ทางขวามือของเครื่องหมายเท่ากับจะมีค่าเป็น 5.234 (แบบ double) แต่เนื่องจากว่าทางซ้ายมือของเครื่องหมายเท่ากับเป็นตัวแปร x แบบ int ดังนั้นค่าของ 5.234 จะต้องถูกแปลงเป็น 5 (แบบ int) โดยที่ตัวเลขหลังจุดทศนิยมจะถูกตัดทิ้งไป ดังนั้น x จึงเก็บค่าเท่ากับ 5 มิใช่ 5.234

เอกสารเพื่อการศึกษา แจกฟรี

ในบางครั้งเราสามารถใช้เครื่องหมายเท่ากับหลายๆครั้งภายในหนึ่งประโยคคำสั่งสำหรับการกำหนดค่าของตัวแปรมากกว่าหนึ่งตัว เช่น เรากำหนดให้ x, y และ z เป็นตัวแปรแบบ int และเราต้องการกำหนดค่าของตัวแปรทั้งสามตัวให้มีค่าเท่ากับศูนย์ เราก็เขียนประโยคคำสั่งได้ดังต่อไปนี้

```
x = 0;
y = 0;
z = 0;
```

แต่ถ้าเราต้องการประหยัดเนื้อที่ในการเขียนโปรแกรมโค้ด สำหรับในกรณีนี้เราก็เขียนให้อยู่ในประโยคคำสั่งเดียวกันเป็น

```
x = y = z = 0;
```

เพียงแต่มีข้อแตกต่างอยู่ในเรื่องของลำดับการทำงานของแต่ละคำสั่งคือ เวลาเราใช้เครื่องหมายเท่ากับหลายๆครั้งในหนึ่งประโยคคำสั่ง เราจะต้องอ่านจากขวาไปซ้าย (ตามหลักการจัดหมู่ของโอเปอร์เรเตอร์) ดังนั้นจึง หมายถึง

```
{
      z = 0;
      y = z;
      x = y;
}
```

ในตัวอย่างนี้ ลำดับของการกำหนดค่าของตัวแปรนั้นไม่สำคัญ ไม่ว่าเราจะกำหนดค่าของ x, y และ z ให้ เป็นศูนย์ตามลำดับ หรือ เริ่มต้นด้วย z ก่อน ย่อมให้ผลเหมือนกัน เพราะเรากำหนดให้ตัวแปรทั้งสามตัวมี ค่าเป็นศูนย์

ข้อสังเกต เหตุที่เขียนประโยคคำสั่งทั้งสามให้อยู่ระหว่างเครื่องหมายปีกกา ก็เพราะว่า ต้องการให้ผู้อ่าน มองเห็นว่าประโยคคำสั่งประโยคนี้

```
x = y = z = 0;
```

เป็นประโยคคำสั่งที่ประกอบด้วยคำสั่งย่อยๆสามคำสั่ง ซึ่งคอมไพล์เลอร์จะมองว่าเป็นเพียงประโยคคำสั่งประโยคเดียว ถ้าเราจะแยกคำสั่งย่อยๆแต่ละคำสั่งออกให้เป็นประโยคคำสั่ง เราก็ควรจะกำหนดขอบเขตของชุดคำสั่งนี้ให้เป็นเสมือนกับว่า ประโยคคำสั่งเชิงซ้อน (Compound Statement) นี้เป็นประโยคคำสั่งประโยคเดียวโดยใช้เครื่องหมายปีกกาในการกำหนดขอบเขต (Scope) ของประโยคคำสั่งนี้

2.1.6 โอเปอร์เรเตอร์สำหรับข้อมูลแบบ int

เมื่อเราได้กำหนดรูปแบบของค่าคงที่หรือตัวแปรขึ้นมาใช้แล้ว เราก็สามารถนำข้อมูลเหล่านี้มาบวก ลบคูณหาร หรือใช้สำหรับโอเปอร์เรชันพื้นฐานอื่นๆได้ ตามตารางข้างล่างนี้

โอเปอร์เรเตอร์	สัญลักษณ์
การบวก	+
การลบ	-
การคูณ	*
การหาร	/

การหาค่าโมดูโล	%
การเพิ่มค่าขึ้นอีกหนึ่ง	++
การลดค่าลงอีกหนึ่ง	--

ตารางที่ 2.4 โอเปอร์เรเตอร์พื้นฐานเชิงคำนวณสำหรับข้อมูลแบบ int

สำหรับการหารเลขจำนวนเต็มจะมีข้อควรระวังเวลาใช้คือ ถ้าตัวส่วนหารตัวเศษได้ไม่ลงตัว เช่น 13 / 5 ค่าของผลลัพธ์จะเป็น 2 และเศษที่เหลือเท่ากับ 3 จะถูกปัดทิ้งไป ส่วนโมดูโล (Modulo) เป็นการหาค่าเศษที่เหลือจากการหารเลขจำนวนเต็มสองจำนวน เช่น 13 % 5 ผลลัพธ์จะมีค่าเท่ากับ 3

โปรแกรมตัวอย่างการใช้โอเปอร์เรเตอร์สำหรับการหาร และการหาค่าโมดูโล

```
#include <stdio.h>

void main()
{
      int x, y;

      x = 13;
      y = 5;
      printf ("13 is divided by 5 = %d\n",  x / y);
      printf ("Remainder = %d\n", x % y );
}
```

โอเปอร์เรเตอร์ ++ และ -- เรียกว่า Increment และ Decrement Operator ตามลำดับ โอเปอร์เรเตอร์ทั้งสองตัวนี้ใช้ได้กับตัวแปรเท่านั้นไม่สามารถใช้ได้กับค่าคงที่หรือนิพจน์ที่ให้ค่าคงที่ใดๆ ถ้าใช้กับตัวแปรเราสามารถวางไว้ข้างหน้าหรือข้างหลังตัวแปรก็ได้ แต่มีข้อแตกต่างในเรื่องลำดับการทำงานของโอเปอร์เร เตอร์ สมมุติว่า x และ y เป็นตัวแปรแบบ int เราลองมาพิจารณาสองประโยคคำสั่งต่อไปนี้

```
x = ++y;          x = y++;
```

ประโยคคำสั่งแรกหมายถึง การเพิ่มค่าของ y ขึ้นอีกหนึ่ง หลังจากที่ y มีค่าใหม่แล้วก็ผ่านค่านี้ไปให้ตัวแปร x ดังนั้นประโยคคำสั่งต่อไปนี้ จึงให้ผลเหมือนกับเราเขียนประโยคคำสั่งข้างล่างนี้

```
{
      y = y + 1;
      x = y;
}
```

ส่วนประโยคคำสั่งที่สอง ให้ผลแตกต่างจากแบบแรกคือ เราผ่านค่าของ y ให้ x ก่อน แล้วจึงเพิ่มค่าของ y ขึ้นอีกหนึ่งในภายหลัง ซึ่งก็คือ

```
{
      x = y;
      y = y + 1;
}
```

เพราะฉะนั้นการวางโอเปอร์เตอร์หรือตัวดำเนินการ ++ หรือ -- ไว้ข้างหน้าหรือข้างหลังตัวแปรใดๆที่ เราต้องการเพิ่มหรือลดค่าของตัวแปรนี้ จึงให้ผลที่แตกต่างกัน โดยเฉพาะอย่างยิ่งเมื่อเราต้องการผ่านค่าของนิพจน์นี้ไปยังตัวแปรใดๆ หรือใช้เป็นค่าพารามิเตอร์ของฟังก์ชัน ดังนั้นเราต้องแน่ใจว่าขั้นตอนใดจะมาก่อนหรือหลังเมื่อโปรแกรมทำงาน ตัวอย่างเช่น ถ้าเราเขียนประโยคคำสั่งว่า

```
z = add (x++, --y);
```

จะหมายถึง ฟังก์ชัน add ให้ค่าของฟังก์ชันเท่ากับ x+y-1 และผ่านค่านี้ไปให้ตัวแปร z ซึ่งเราสามารถ เขียนขั้นตอนการทำงานที่ให้ผลเหมือนกันได้ดังนี้

```
{
      y = y - 1;
      z = add (x,y);
      x = x + 1;
}
```

ตัวอย่างโปรแกรมที่แสดงวิธีการใช้โอเปอร์เรเตอร์ ++ และ --

```
/* 1 */     #include <stdio.h>
/* 2 */
/* 3 */     int main()
/* 4 */     {
/* 5 */           int x;
/* 6 */
/* 7 */           x = 0;
/* 8 */           printf ("++x = %d \n", ++x);
/* 9 */           x = 0;
/*10 */           printf ("x++ = %d \n", x++);
/*11 */           return 0;
/*12 */     }
```

ผลของโปรแกรมที่แสดงออกทางจอภาพคือ

```
++x = 1
x++ = 0
```

ในประโยคคำสั่งบรรทัดที่ 7 ตัวแปร x มีค่าเป็นศูนย์ ในบรรทัดต่อมาเป็นประโยคคำสั่งที่เรียกใช้ฟังก์ชัน printf() โดยมีนิพจน์ ++x เป็นพารามิเตอร์ตัวที่สองของฟังก์ชัน เมื่อเราวางโอเปอร์เรเตอร์ ++ ไว้ข้างหน้าตัวแปร x เวลาคอมไพล์เลอร์แปลงโปรแกรมโค้ดมาถึงขั้นนี้ ก็จะเขียนคำสั่งในภาษาเครื่องที่เพิ่มค่าของ x ขึ้นอีกหนึ่งคือจากศูนย์เป็นหนึ่ง แล้วจึงผ่านค่าใหม่นี้ให้ฟังก์ชัน

printf() ในบรรทัดที่ 9 เรากำหนดค่าของ x ให้มีค่าเป็นศูนย์อีกครั้ง แต่คราวนี้เราวางโอเปอร์เรเตอร์ ++ ไว้ข้างหลังตัวแปร x ผลที่ได้จึงแตกต่างจากในกรณีแรกคือ เราผ่านค่าของ x ให้ฟังก์ชันก่อนเป็นค่าของพารามิเตอร์ตัวที่สอง แล้วจึงเพิ่มค่าของ x ขึ้นอีกหนึ่งในภายหลัง

การใช้โอเปอร์เรเตอร์ ++ หรือ -- บางครั้งมักจะให้ผลข้างเคียงที่ทำให้โปรแกรมทำงานไม่ถูกต้องตามที่เราต้องการ โดยเฉพาะอย่างยิ่งเมื่อเราใช้โอเปอร์เรเตอร์เหล่านี้กับตัวแปรและใช้ตัวแปรนี้หลายครั้งในหนึ่งประโยคคำสั่ง ตัวอย่างเช่น สมมุติว่า x เป็นตัวแปรแบบ int และมีค่าเริ่มต้นเป็น 1

```
x = 1;
printf ("%d %d %d %d\n", ++x, x, ++x, x++);
```

คำถามก็คือว่า นิพจน์ต่างๆที่ใช้เป็นพารามิเตอร์ของฟังก์ชัน printf() มีค่าเป็นเท่าไร คำตอบของคำถาม นี้ขึ้นอยู่กับทิศทางในการคำนวณค่าของนิพจน์ ซึ่งอาจจะแตกต่างกันไปตามคอมไพล์เลอร์ที่ใช้

ทิศทางการคำนวณ	++x	x	++x	x++	ค่าของ x ภายหลังการเรียกใช้ฟังก์ชัน
จากซ้ายไปขวา	2	2	3	3	4
จากขวาไปซ้าย	4	3	3	1	4

ตารางที่ 2.5 ผลการทำงานของโปรแกรมที่แตกต่างกันไปตามคอมไพล์เลอร์ที่ใช้

สำหรับการคำนวณค่าของนิพจน์ที่เป็นพารามิเตอร์ของฟังก์ชันมีหลักการอยู่ว่า ภายในนิพจน์แต่ละตัวที่เป็นพารามิเตอร์ของฟังก์ชัน จะต้องมีการเปลี่ยนแปลงค่าของตัวแปร x ซึ่งเป็นผลมาจากโอเปอร์เรเตอร์ ++ (หรือ --) ก่อนที่จะคำนวณค่าของนิพจน์ที่เป็นพารามิเตอร์ตัวถัดไป (ตามทิศทางของการคำนวณ) สรุปก็คือว่า ค่าของตัวแปร x ในนิพจน์ตัวถัดไปจะต้องมีค่าใหม่ถ้าหากว่าในนิพจน์ที่เป็นพารามิเตอร์ตัวที่แล้วมีการใช้โอเปอร์เรเตอร์ ++ (หรือ --) กับตัวแปร x จากตารางข้างบนจะเห็นได้ว่าผลลัพธ์ที่ได้นั้นมีค่าแตกต่างกันไปตามทิศทางของการคำนวณ เพราะฉะนั้นการเขียนประโยคคำสั่งในลักษณะนี้ควรระมัดระวังหรือไม่ก็ควรจะหลีกเลี่ยง และตัวอย่างอีกตัวอย่างหนึ่งที่แสดงให้เห็นปัญหาในการใช้โอเปอร์เรเตอร์ ++ และ -- เช่น

```
y = (++x  - (x-- + ++x));
```

ในกรณีนี้เราไม่ได้เกี่ยวข้องกับการคำนวณค่าของนิพจน์ที่เป็นพารามิเตอร์ของฟังก์ชัน เราจึงไม่สามารถทราบได้ว่าค่าของตัวแปร x เมื่อจบประโยคคำสั่งนี้จะมีค่าใหม่เป็นเท่าไร และมีหลักการ

คำนวณอย่างไร นอกเหนือจากคำถามนี้แล้วเราก็ไม่สามารถตอบได้ว่า ค่าของ x แต่ละตัวมีค่าเป็นเท่าไรในการคำนวณแต่ละขั้นตอนย่อยของประโยคคำสั่งนี้ การคำนวณค่าของนิพจน์ทางซ้ายและขวามือของเครื่องหมายลบยังขึ้นอยู่กับทิศทางของการคำนวณแตกต่างกันไปตามคอมไพล์เลอร์ที่ใช้ นอกจากนี้ยังมีข้อสงสัยอยู่อีกว่า ค่าของตัวแปร x แต่ละตัวจะมีค่าเดิม หรือค่าใหม่อันเป็นผลมาจากการใช้โอเปอร์เรเตอร์ ++ และ -- เพราะฉะนั้นการเขียนนิพจน์ในลักษณะเช่นนี้จึงถือว่ากำกวม ควรจะหลีกเลี่ยงอย่างยิ่ง

ข้อสังเกต ทำไมการคำนวณค่าของนิพจน์จึงขึ้นอยู่กับทิศทางการคำนวณที่คอมไพล์เลอร์ใช้ ? คำตอบก็คือว่า ในบางครั้งคอมไพล์เลอร์บางตัวจะพยายามเขียนคำสั่งภาษาเครื่องให้อยู่ในรูปแบบที่มีประสิทธิภาพในการคำนวณมากที่สุด ซึ่งเราเรียกขั้นตอนนี้ว่า Optimization เพราะฉะนั้น ก็ไม่จำเป็นเสมอไปว่า นิพจน์ใดๆที่อยู่ทางซ้ายมือของโอเปอร์เรเตอร์เชิงคำนวณจะต้องถูกดำเนินการก่อนนิพจน์ที่อยู่ทางขวามือ

ข้อควรจำ เราควรจะใช้คำสั่งหรือเขียนนิพจน์ที่อ่านแล้วเข้าใจได้ง่าย ชัดเจน และ ไม่ต้องพิศดารเพื่อหลีกเลี่ยงผลข้างเคียงที่อาจจะเกิดขึ้นได้

ในภาษาซียังมีการใช้โอเปอร์เรเตอร์ที่มีลักษณะแตกต่างจากในภาษาอื่นๆ ซึ่งจะเห็นได้จากตัวอย่างต่อไปนี้ สมมุติว่า x เป็นตัวแปรแบบ `int` มีค่าเท่ากับ 2 ถ้าเราเขียนคำสั่งว่า

```
x = x+3;
```

ก็หมายถึง การบวกค่าของ x ในขณะนั้นซึ่งคือ 2 ด้วยเลข 3 แล้วเก็บผลลัพธ์จากการบวกนี้ไว้ในตัวแปร x ซึ่งทำให้ x จะมีค่าใหม่เป็น 5 เราสังเกตว่า ค่าของตัวแปร x ค่าใหม่จะมากกว่าค่าเดิมอยู่ 3 เมื่อเราเห็นความสัมพันธ์ระหว่างค่าเก่าและค่าใหม่ของตัวแปร x แล้ว เราสามารถเขียนประโยคใหม่ในภาษาซีให้อยู่ในรูปของ

```
x += 3;
```

ซึ่งอ่านว่า “เพิ่มค่าของ x ขึ้นอีก 3 ” สมมุติว่า ถ้าเราต้องการให้ค่าใหม่ของตัวแปร x เป็นสามเท่าของค่าเดิม เราก็เขียนประโยคคำสั่งได้ดังนี้

```
x *= 3;
```

ซึ่งให้ผลเหมือนกับ

```
x = (x * 3);
```

ใครชอบแบบไหนก็สามารถเลือกใช้ได้ตามใจชอบ แต่การใช้โอเปอร์เรเตอร์ เช่น += หรือ *= จะเป็นเอกลักษณ์อย่างหนึ่งของการเขียนโปรแกรมในภาษาซี นอกเหนือจากโอเปอร์เรเตอร์ที่ได้ยกตัวอย่างไปแล้ว ยังมีโอเปอร์เรเตอร์สำหรับหน้าที่อื่นๆ อีกตามตารางข้างล่างนี้

โอเปอร์เรเตอร์	ตัวอย่างการใช้	ตัวอย่างที่ให้ผลเหมือนกัน	ความหมาย
+=	x += 1;	x = x+1;	บวก
-=	x -= 1;	x = x-1;	ลบ
*=	x *= 1;	x = x*1;	คูณ
/=	x /= 1;	x = x/1;	หาร
%=	x %= 1;	x = x%1;	โมดูโล
>>=	x >>=1;	x = x>>1;	เลื่อนบิตไปทางขวา
<<=	x <<=1;	x = x<<1;	เลื่อนบิตไปทางซ้าย
&=	x &= 1;	x = x&1;	เปรียบเทียบบิตแบบ AND
^=	x ^= 1;	x = x^1;	เปรียบเทียบบิตแบบ EXOR
|=	x |= 1;	x = x|1;	เปรียบเทียบบิตแบบ OR

ตารางที่ 2.6 โอเปอร์เรเตอร์ที่ใช้ในการกำหนดค่าของตัวแปร

การใช้โอเปอร์เรเตอร์ในการกำหนดค่าของตัวแปรนั้น นอกจากเราได้กำหนดค่าให้ตัวแปรแล้วเรายังได้กำหนดค่าของนิพจน์อีกด้วย ตัวอย่างเช่น

```
/* 1 */     #include <stdio.h>
/* 2 */
/* 3 */     int main()
/* 4 */     {
/* 5 */           int x;
/* 6 */
/* 7 */           x = 0;
/* 8 */           printf ("The value of (x+=2) is %d.\n",(x+=2));
/* 9 */           printf ("x = %d\n", x);
/*10 */           printf ("The value of (x=1) is %d.\n",(x=1));
/*11 */           printf ("x = %d\n", x);
/*12 */           return 0;
/*13 */     }
```

ผลจากโปรแกรมก็คือ

```
The value of (x+=2) is 2.
x = 2
The value of (x=1) is 1.
x = 1
```

ในบรรทัดที่ 7 เรากำหนดให้ตัวแปร x มีค่าเท่ากับศูนย์ ในบรรทัดที่ 8 เราใช้นิพจน์ (x+=2) เป็นพารามิเตอร์ตัวที่สองของฟังก์ชัน printf() เนื่องจาก x มีค่าเริ่มต้นเป็นศูนย์ เมื่อบวกด้วยสองจึงมีค่าใหม่เป็นสอง ในกรณีนี้เราไม่ได้ผ่านค่าของตัวแปร x ให้ฟังก์ชัน แต่เป็นการผ่านค่าของนิพจน์ และค่าของนิพจน์ นี้ก็มีค่าเท่ากับค่าของตัวแปร x ในขณะนั้น ในบรรทัดที่ 10 และ 11 ก็ให้ผลในทำนองเดียวกัน

2.1.7 ลำดับการทำงานของโอเปอร์เรเตอร์

เมื่อเราใช้โอเปอร์เตอร์หลายๆตัว เราก็จำเป็นที่จะต้องทราบลำดับการทำงานของโอเปอร์-เรเตอร์แต่ละตัว เพื่อจะให้แน่ใจว่าการทำงานของโปรแกรมเป็นไปตามลำดับที่เราต้องการจริง

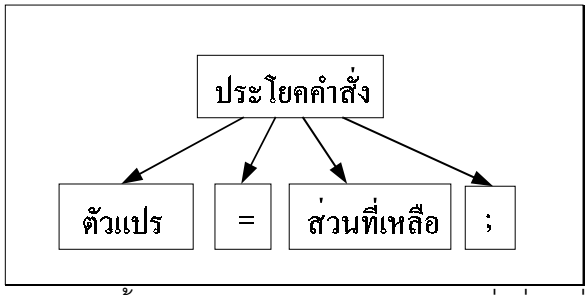
```
z = x ++ + y;
z = - 1 - x / y * 3;
z = 1 - ++ x / y / 2 % 5;
```

ประโยคคำสั่งทั้งสามแสดงให้เห็นความสำคัญของการกำหนดลำดับการทำงานของโอเปอร์เรเตอร์แต่ละตัว ถ้าปราศจากกฎเกณฑ์แล้วเราสามารถตีความหมายของประโยคคำสั่งเหล่านี้ได้หลายๆกรณีดังนั้นเราจึงต้องเข้าใจข้อกำหนดสำหรับลำดับการทำงานของโอเปอร์เรเตอร์ที่คอมไพล์เลอร์ใช้

ลำดับในการทำงาน	โอเปอร์เรเตอร์	ความหมาย	การจัดหมู่
1	++	เพิ่มค่าขึ้นอีกหนึ่ง	จากขวาไปซ้าย
2	--	ลดค่าลงอีกหนึ่ง	จากขวาไปซ้าย
3	*	คูณ	จากซ้ายไปขวา
4	/	หาร	จากซ้ายไปขวา
5	%	หาค่าโมดูโล	จากซ้ายไปขวา
6	+	บวก	จากซ้ายไปขวา
7	-	ลบ	จากซ้ายไปขวา
8	=	กำหนดค่า	จากขวาไปซ้าย

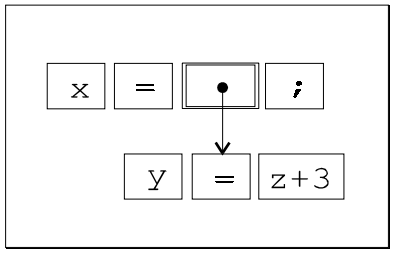
ตารางที่ 2.7 ลำดับการทำงานก่อนหลังของโอเปอร์เรเตอร์

จะเห็นได้ว่าเครื่องหมายเท่ากับที่ใช้ในการกำหนดค่าของตัวแปรจะมีลำดับการทำงานอยู่หลังสุดเมื่อเปรียบเทียบกับโอเปอร์เรเตอร์ตัวอื่นๆในประโยคคำสั่ง ภายในประโยคคำสั่งที่มีเครื่องหมายเท่ากับเราสามารถจำแนกขั้นตอนการทำงานได้ดังต่อไปนี้



รูปภาพที่ 2.1 การจำแนกขั้นตอนการทำงานในประโยคคำสั่งที่มีเครื่องหมายเท่ากับ

ตัวอย่างเช่น `x = y = z+3;`



รูปภาพที่ 2.2 การคำนวณค่าของนิพจน์ `x = y = z+3`

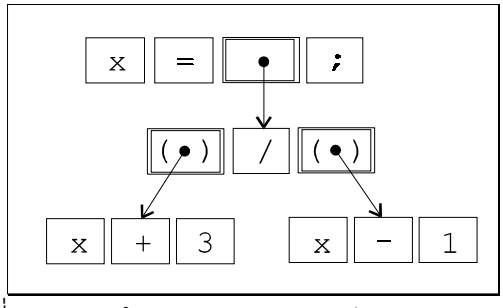
หลังจากที่เราได้จำแนกขั้นตอนการทำงานของประโยคคำสั่งแล้วเราจะอ่านจากข้างล่างขึ้นข้างบน ซึ่งเหมือนกับเวลาที่คอมพิวเตอร์ทำตามคำสั่งเหล่านี้

(1) บวกค่าของ z ด้วย 3 แล้วเก็บค่าผลรวมไว้ในตัวแปร y

(2) กำหนดค่าของตัวแปร x ให้มีค่าเหมือนกับค่าของ y

(3) จบประโยคคำสั่ง

ตัวอย่างที่สอง `x = (x+3)/(x-1);`



รูปภาพที่ 2.3 การคำนวณค่าของนิพจน์ `x = (x+3)/(x-1)`

(1) หาค่าผลบวกของ x และ 3 และ หาค่าของผลลบของ x และ 1

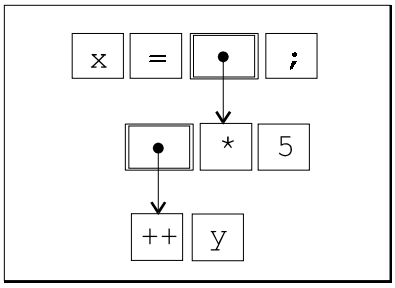
(2) หาค่าผลหารจากตัวเศษและส่วนที่คำนวณได้จากขั้นตอนที่หนึ่ง

(3) กำหนด x ให้มีค่าเท่ากับผลหารจากขั้นตอนที่สอง

(4) จบประโยคคำสั่ง

ตัวอย่างที่สาม `x = ++y * 5;`

ในประโยคคำสั่งนี้มีสามโอเปอร์เรเตอร์ คือ = ++ และ * จากตาราง โอเปอร์เรเตอร์ = มีลำดับการทำ งานหลังสุด และ ++ มีลำดับการทำงานมาก่อน * ดังนั้นเราสามารถเขียนขั้นตอนการทำงานได้ดังนี้



รูปภาพที่ 2.4 การคำนวณค่าของนิพจน์ `x = ++y * 5`

(1) เพิ่มค่าของตัวแปร y ขึ้นอีกหนึ่ง

(2) คูณผลลัพธ์จากขั้นที่หนึ่ง ด้วย 5

(3) กำหนดค่าของ x ให้มีค่าเท่ากับผลลัพธ์จากขั้นที่สอง

(4) จบประโยคคำสั่ง

ถ้าเราไม่แน่ใจว่า โอเปอร์เรเตอร์ใดจะทำงานก่อนหรือหลัง เราก็สามารถใช้เครื่องหมายวงเล็บเข้ามาช่วยกำหนด และยังช่วยให้เราอ่านโปรแกรมโค้ดได้ง่ายขึ้น เช่น

```
z = (x++) + y;
z = - 1 - ((x/y)*3);
z = 1 - ((((++x)/y) /2) %5);
```

2.1.8 โอเปอร์เรเตอร์เชิงเปรียบเทียบ (Relational Operator)

โอเปอร์เรเตอร์ประเภทนี้มักใช้ในการตรวจสอบค่าความจริงของนิพจน์ทางตรรกศาสตร์ และในการเขียนโปรแกรม เราจะใช้นิพจน์ทางตรรกศาสตร์เหล่านี้สำหรับโครงสร้างหรือประโยคที่เป็นเงื่อนไข (Conditional Structure) เช่น โครงสร้างเงื่อนไขแบบ if - else และ switch - case เป็นต้น

โอเปอร์เรเตอร์	ความหมาย
==	เท่ากับ
!=	ไม่เท่ากับ
<	น้อยกว่า
<=	น้อยกว่าหรือเท่ากับ
>	มากกว่า
>=	มากกว่าหรือเท่ากับ

ตารางที่ 2.8 ตารางโอเปอร์เรเตอร์เชิงเปรียบเทียบ

เราสามารถกำหนดเงื่อนไขหรือประพจน์ขึ้นมาใช้โดยอาศัยโอเปอร์เรเตอร์เชิงเปรียบเทียบเหล่านี้ เพื่อหาความสัมพันธ์ระหว่างค่าของตัวเลขสองตัว เช่น เราสามารถตรวจสอบได้ว่าตัวเลขจำนวนเต็มสองจำนวนมีค่ามากกว่า น้อยกว่า หรือเท่ากัน ในภาษาซีค่าของประพจน์ทางตรรก-ศาสตร์หรือนิพจน์เงื่อนไขจะมีค่าเท่ากับศูนย์หรือหนึ่งเท่านั้น โดยที่ศูนย์หมายถึง ‘เท็จ’ และหนึ่งหมายถึง ‘จริง’

นิพจน์เงื่อนไข	ค่าของนิพจน์เงื่อนไข	ความหมาย
(1 == 1)	1	จริง
(1 != 1)	0	เท็จ
(1 >= 2)	0	เท็จ
(1 <= 2)	1	จริง
(1 < 2)	1	จริง
(1 > 2)	0	เท็จ

ตารางที่ 2.9 ตัวอย่างการใช้โอเปอร์เรเตอร์เชิงเปรียบเทียบในภาษาซี

ตัวอย่างโปรแกรมสำหรับการใช้โอเปอร์เรเตอร์เชิงเปรียบเทียบ

```
#include <stdio.h>

int main()
{
      int x, y;

      x = 0;  y = 1;
      printf ("The value of (x >= y) is %d.\n", (x >= y));
      printf ("The value of ((x >= y) != 1) is %d.\n",
              (x >= y) != 1);
      printf ("The value of (((++x >= y)+1) < 3) is %d.\n",
              (((++x >= y)+1) < 3));
      return 0;
}
```

ผลของโปรแกรมคือ

```
The value of (x >= y) is 0.
The value of ((x >= y) != 1) is 1.
The value of (((++x >= y)+1) < 3) is 1.
```

ขอให้ผู้อ่านลองคำนวณค่าของนิพจน์ในตัวอย่างที่แล้วทีละขั้นด้วยตนเองและเปรียบเทียบกับผลของโปรแกรม โปรดสังเกตว่า โอเปอร์เรชันเชิงเปรียบเทียบจะให้ค่าเท่ากับศูนย์หรือหนึ่งเท่านั้น

2.1.9 โอเปอร์เรเตอร์ทางตรรกศาสตร์ (Boolean Operator)

เมื่อเราสามารถกำหนดเงื่อนไขหรือประพจน์ขึ้นมาใช้ได้แล้ว โดยอาศัยโอเปอร์เรเตอร์เชิงเปรียบดัง ที่กล่าวไปในหัวข้อที่แล้ว บางครั้งเราก็ต้องการสร้างเงื่อนไขที่ซับซ้อนกว่าเดิม การใช้โอเปอร์เรเตอร์ทางตรรกศาสตร์จึงเปิดโอกาสให้เรานำเงื่อนไขพื้นฐานเหล่านี้หลายๆเงื่อนไขมาประกอบเข้าด้วยกันขึ้นเป็นเงื่อนไขใหม่และซับซ้อนกว่าเดิม

โอเปอร์เรเตอร์	รูปแบบการใช้งาน	ความหมาย
!	!(*expression*)	ปฏิเสธ
&&	(*expression*$_1$ && *expression*$_2$)	และ
||	(*expression*$_1$ || *expression*$_2$)	หรือ

ตารางที่ 2.10 โอเปอร์เรเตอร์เชิงตรรกศาสตร์ในภาษาซี โดยเรียงตามลำดับการทำงานของโอเปอร์เรเตอร์

นิพจน์	ค่าของนิพจน์
`!0`	1
`!1`	0
`!0xf9ff`	0
`!(-101)`	0
`!!(-101)`	1
`!!!!!!!(!0)`	0
`(1 && 100)`	1
`((10 < 1) || (1==1))`	1
`((5 * 3/2) && 0)`	0
`((x=0x0001) && (x<=1))`	1
`((x=1) && (x--<1) || ++x)`	1

ตารางที่ 2.11 ตัวอย่างการใช้โอเปอร์เรเตอร์ในทางตรรกศาสตร์

โปรดสังเกตว่า โอเปอร์เรชันเชิงตรรกศาสตร์ในภาษาซีจะให้ค่าเท่ากับศูนย์หรือหนึ่งเท่านั้น เช่น นิพจน์ `!(!(-101))` มีค่าเท่ากับ `1` ไม่ใช่เท่ากับ `-101`

เรามาลองเขียนโปรแกรมตัวอย่างแบบง่ายๆในการใช้โอเปอร์เรเตอร์ทางตรรกศาสตร์ เนื่องจากว่าเรายังไม่ได้ทำความรู้จักกับโครงสร้างหรือประโยคที่เป็นเงื่อนไขในภาษาซีมากเท่าที่ควร เราจึงพิจารณาตัวอย่างง่ายๆก่อน

```
/* 1 */     #include <stdio.h>
/* 2 */     int main()
/* 3 */     {
/* 4 */        int x, y, result;
/* 5 */        x = 0; y = 1;
/* 6 */        printf("x = %d, y = %d\n", x, y);
/* 7 */        result = ((x==y) || (x > y));
/* 8 */        printf("((x==y) || (x > y)) is %d\n\n", result);
/* 9 */
/*10 */        printf("x = %d, y = %d\n", x, y);
/*11 */        result = (!(++x && y--) && (x || !y));
/*12 */        printf("(!(++x && y--) && (x || !y)) is %d\n",
/*13 */                        result);
/*14 */        printf("x = %d, y = %d\n", x, y);
/*15 */        return 0;
/*16 */     }
```

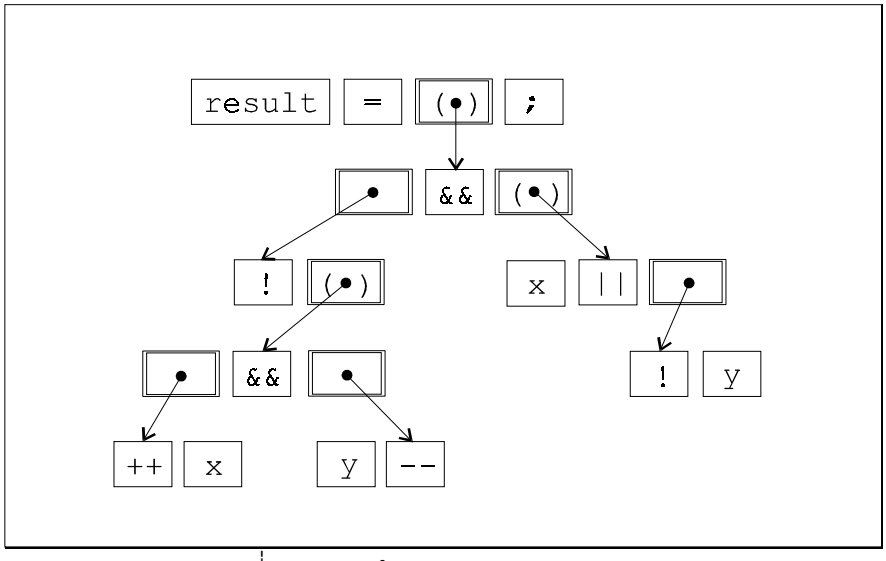
ผลของโปรแกรมที่แสดงออกทางจอภาพ คือ

```
x = 0, y = 1
((x==y) || (x > y)) is 0
```

```
x = 0, y = 1
(!(++x && y--) && (x || !y)) is 0
x = 1, y = 0
```

ในบรรทัดที่ 7 ตัวแปร result มีค่าเท่ากับ 0 เพราะประพจน์ “x มีค่าเท่ากับ y หรือ x มีค่ามากกว่า y” มีค่าเป็น‘เท็จ’ หรือศูนย์ เพราะตัวแปร x และ y มีค่าในขณะนั้นเท่ากับ 0 และ 1 ตามลำดับ ประโยคคำสั่งในบรรทัดที่ 11 นั้นซับซ้อนมากกว่าในบรรทัดที่ 7 ซึ่งเราสามารถจำแนกขั้นตอนการหาค่าความจริงของ ประพจน์ได้ดังนี้

```
result = (!(++x && y--) && (x || !y));
```



รูปภาพที่ 2.5 การคำนวณค่าของตัวแปร result

เวลาเราคำนวณค่าของนิพจน์ทางตรรกศาสตร์ เราจะเริ่มจากซ้ายไปขวาและในวงเล็บก่อน ดังนั้นตามภาพประกอบข้างบนเราจะเริ่มคำนวณจากนิพจน์ที่อยู่ทางซ้ายมือของโอเปอร์เรเตอร์ && หรือ || ก่อน โดยแบ่งออกเป็นสองกรณีดังนี้

1) โอเปอร์เรเตอร์ ‘และ’ (&&)

ให้เริ่มหาค่าของนิพจน์ที่อยู่ทางซ้ายมือของโอเปอร์เรเตอร์ก่อน ถ้านิพจน์ให้ค่าเท่ากับหนึ่ง (หรือไม่เท่ากับศูนย์ เนื่องจากว่าในภาษาซี เรากำหนดให้ศูนย์มีค่าความจริงเป็น ‘เท็จ’ ในเชิงตรรก - ศาสตร์ และ ‘ส่วนที่เหลือ’ มีค่าความจริงเป็น ‘จริง’ ซึ่งก็คือค่าที่ไม่เท่ากับศูนย์) นิพจน์นี้จึงให้ค่าเท่ากับ ‘จริง’ ดังนั้น เราต้องหาค่าของนิพจน์ทางขวามือของโอเปอร์เรเตอร์ต่อไป แต่ถ้าค่าของนิพจน์ทางซ้ายมือเป็น ‘เท็จ’ หรือศูนย์ เราก็ไม่ต้องคำนวณหรือปฏิบัติตามคำสั่งใดๆทางขวามือของโอเปอร์เรเตอร์อีกต่อไป เพราะการเปรียบเทียบแบบ ‘และ’ ถ้ามีตัวใดตัวหนึ่งที่เป็นองค์ประกอบ

ของประพจน์มีค่าเป็นเท็จ ค่าของประพจน์สำหรับการเปรียบแบบ ‘และ’ จึงเป็นเท็จเสมอในเชิงตรรกศาสตร์

2) โอเปอร์เรเตอร์ ‘หรือ’ (||)

ให้เริ่มหาค่าของนิพจน์ที่อยู่ทางซ้ายมือก่อน ถ้านิพจน์นี้ให้ค่าเท่ากับศูนย์ นิพจน์นี้จึงให้ค่าเป็น ‘เท็จ’ ดังนั้นจึงต้องหาค่าของนิพจน์ทางขวามือของโอเปอร์เรเตอร์ต่อไป แต่ถ้าค่าของนิพจน์ทางซ้ายมือเป็น ‘จริง’ หรือไม่เท่ากับศูนย์ เราก็ไม่ต้องปฏิบัติตามคำสั่งใดๆทางขวามือของโอเปอร์-เรเตอร์อีกต่อไป เพราะการเปรียบเทียบแบบ ‘หรือ’ ถ้ามีตัวใดตัวหนึ่งเป็นจริง ค่าของประพจน์ของการเปรียบแบบ ‘หรือ’ จึงเป็นจริงเสมอ

เมื่อเราได้เรียนรู้หลักการคำนวณทั้งสองข้อแล้ว เราก็เริ่มโดยตรวจดูว่า ++ ให้ค่า x เท่ากับศูนย์หรือไม่ ในตัวอย่างของเราบรรทัดที่ 11 นิพจน์ ++x ให้ค่าเป็นหนึ่ง และ x ก็มีค่าใหม่เป็นหนึ่งด้วย ดังนั้นเราก็ต้องคำนวณค่าที่อยู่ทางขวามือต่อไป y-- ให้ค่าเท่ากับหนึ่งเพราะเนื่องจากว่าโอเปอร์เรเตอร์ -- อยู่ข้างหลัง (โปรดสังเกตว่า -- และ ++ มีลำดับการทำงานก่อน && และ ||) ดังนั้น !(x++ && y--) จึงให้ค่าเป็นศูนย์ เมื่อทางซ้ายมือของโอเปอร์เรเตอร์ && มีค่าเท่ากับศูนย์หรือเท็จ ดังนั้นเราก็ไม่ต้องคำนวณค่าของนิพจน์ (x || !y) อีกต่อไป (ซึ่งจะได้ค่าเป็นหนึ่งถ้าเราต้องคำนวณจริงๆ โปรดสังเกตว่าตัวแปร y ในนิพจน์นี้มีค่าเท่ากับศูนย์และไม่ใช่หนึ่งเพราะเป็นผลมาจากนิพจน์ y-- ก่อนหน้านี้) ดังนั้นเราจึงได้ค่าของ (!(++x && y--) && (x || !y)) เท่ากับศูนย์เป็นคำตอบ

2.1.10 โอเปอร์เรเตอร์ในการคำนวณแบบบิต (Bitwise Operator)

เราใช้โอเปอร์เรเตอร์ชนิดนี้สำหรับการคำนวณแบบบิตต่อบิต เช่น การเปรียบเทียบบิตแบบต่างๆใน เชิงตรรกศาสตร์ การเลื่อนแถวบิตไปทางซ้ายหรือขวา หรือใช้ในการเปลี่ยนแปลงแก้ไขค่าของจำนวนเต็มใดๆแบบบิตต่อบิต เป็นต้น ตามตารางข้างล่างนี้

โอเปอร์เรเตอร์	สัญลักษณ์	การจัดหมู่
บิตคอมพลีเมนท์	~	จากขวาไปซ้าย
เลื่อนแถวบิตไปทางซ้าย	<<	จากซ้ายไปขวา
เลื่อนแถวบิตไปทางขวา	>>	จากซ้ายไปขวา
การเปรียบเทียบแบบ AND	&	จากซ้ายไปขวา

การเปรียบเทียบแบบ XOR	^	จากซ้ายไปขวา
การเปรียบเทียบแบบ OR	|	จากซ้ายไปขวา
การกำหนดค่าของแถวบิต	&= |= ^= <<= >>=	จากขวาไปซ้าย

ตารางที่ 2.12 โอเปอร์เรเตอร์ต่างๆสำหรับการคำนวณแบบบิต เรียงตามลำดับการทำงานก่อนหลัง

ค่าของโอเปอร์เรชันต่างๆเราสามารถอ่านได้จากตารางข้างล่างนี้ โดยที่ A และ B คือ บิตเดี่ยวหรือเลขโดดฐานสองมีค่าเท่ากับหนึ่งหรือศูนย์

A	B	~A	~B	A & B	A | B	A ^ B	~(A & B)	~(A | B)
0	0	1	1	0	0	0	1	1
0	1	1	0	0	1	1	1	0
1	0	0	1	0	1	1	1	0
1	1	0	0	1	1	0	0	0

ตารางที่ 2.13 ตัวอย่างการคำนวณนิพจน์แบบบิต

การเปรียบเทียบแบบ AND นั้นจะให้ค่าเป็นหนึ่งก็ต่อเมื่อนิพจน์(บิต) ทางซ้ายมือและขวามือของโอเปอร์เรเตอร์มีค่าเป็นหนึ่ง ในกรณีอื่นการเปรียบเทียบแบบนี้ก็จะให้ค่าเป็นศูนย์ การเปรียบเทียบแบบ OR เมื่อ บิตตัวใดตัวหนึ่งมีค่าเท่ากับหนึ่งก็จะให้ค่าเป็นหนึ่ง แต่ถ้าบิตทั้งสองตัวมีค่าเท่ากับศูนย์ก็จะให้ค่าเป็นศูนย์ การเปรียบเทียบแบบ XOR นั้นมีหลักการคำนวณอยู่ว่า ถ้าบิตทั้งสองตัวที่เราต้องการเปรียบเทียบกันมีค่าแตกต่างกัน ซึ่งก็คือว่าถ้าบิตตัวหนึ่งมีค่าเป็นศูนย์และบิตอีกตัวหนึ่งมีค่าเป็นหนึ่ง ผลลัพธ์ที่ได้จากการ เปรียบเทียบแบบ XOR จะให้ค่าเป็นหนึ่ง และ ถ้าบิตทั้งสองตัวมีค่าเท่ากันก็จะให้ค่าเป็นศูนย์

จากตารางข้างบน A และ B เป็นเลขโดด (หนึ่งบิต) ในเลขฐานสอง หรือเราอาจจะมองว่าเลขโดดเหล่านี้เป็นแถวบิตก็ได้แต่มีความยาวเท่ากับหนึ่งบิต การใช้โอเปอร์เรเตอร์สำหรับการคำนวณแบบบิตต่อบิตเราสามารถใช้ได้กับแถวบิตที่มีความยาวตั้งแต่หนึ่งบิตขึ้นไปซึ่งใช้แสดงค่าของเลขจำนวนเต็มแบบ `int` ในระบบเลขฐานสอง การเปรียบเทียบระหว่างแถวบิตก็ไม่แตกต่างอะไรจากการเปรียบเทียบระหว่างบิตแต่ละตัวเพียงแต่ตำแหน่งของบิตจากแถวบิตแต่ละแถวจะต้องตรงกันเท่านั้น ตัวอย่างการใช้โอเปอร์เรเตอร์สำหรับแถวบิต A และ B ที่มีค่าตามตารางข้างล่างนี้

นิพจน์ (แถวบิต)	ค่าของนิพจน์ (แถวบิต)
A	01001010
B	11011011
~A	10110101
~B	00100100
A & B	01001010

A | B	11011011
A ^ B	10010001
~(A & B)	10110101
~(A | B)	00100100
~(A ^ B)	01101110
~A | ~B	10110101
~A & ~B	00100100
~A ^ ~B	10010001

ตารางที่ 2.14 ตัวอย่างการคำนวณนิพจน์สำหรับแถวบิต

2.1.11 โอเปอร์เรเตอร์สำหรับการเลื่อนแถวบิต (Shift Operator)

การเลื่อนแถวบิตจัดว่าเป็นโอเปอร์เรชันอย่างหนึ่งสำหรับการประมวลผลข้อมูลแบบบิตต่อบิต โดยเราสามารถเลื่อนบิตของข้อมูลที่แสดงให้อยู่ในระบบเลขฐานสองได้ โดยเลื่อนบิตทุกๆตัวของข้อมูลไปทางซ้ายหรือขวา โอเปอร์เรเตอร์ที่ใช้สำหรับหน้าที่นี้ก็คือ << และ >> ตามลำดับ และมีรูปแบบการใช้ดังนี้

expression$_1$ **<<** *expression*$_2$
expression$_1$ **>>** *expression*$_2$

expression$_1$ เป็นนิพจน์ใดๆที่เราต้องการเลื่อนแถวบิตของข้อมูล โดยเลื่อนไปทางซ้ายหรือขวาทั้งหมด expression$_2$ ตำแหน่ง ดังนั้น expression$_2$ จึงเป็นนิพจน์ที่ให้ค่าคงที่ๆเป็นจำนวนเต็มเท่านั้น และต้องมีค่าเป็นบวกหรือเท่ากับศูนย์เท่านั้นและมีค่าไม่เกินความยาวของแถวบิต ในกรณีที่ expression$_2$ มีค่าเท่ากับศูนย์ก็จะไม่มีการเลื่อนบิตใดๆ หรืออาจจะกล่าวได้ว่า เลื่อนไปทางซ้ายหรือขวาทั้งหมดศูนย์ตำแหน่ง

การเลื่อนแถวบิตของข้อมูลก็เปรียบเสมือนกับว่า เรามีที่อยู่ทั้งหมด สมมุติว่า 16 ที่ แต่ละที่มีหมาย เลขกำกับอยู่ตั้งแต่ 0 ถึง 15 และสามารถให้นักเรียนหนึ่งคนขึ้นไปยืนได้ในแต่ละที่ และถ้าเรากำหนดว่า ทุกๆที่จะต้องมีนักเรียนชายหรือหญิงยืนอยู่หนึ่งคน ดังนั้นจึงมีนักเรียนยืนอยู่ทั้งหมดสิบหกคน ถ้าตำแหน่งใดมีนักเรียนชายยืนอยู่ เราก็กำหนดให้ที่นั้นมีค่าเป็นศูนย์ และถ้าตำแหน่งใดมีนักเรียนหญิงยืนอยู่ก็ให้ที่นั้นมีค่าเป็นหนึ่ง คราวนี้เราบอกให้นักเรียนทุกคนก้าวไปทางขวาหนึ่งตำแหน่งคนที่อยู่ริมขวาสุด ก็จะหล่นจากที่ดังกล่าวไป และที่ทางซ้ายสุดก็ว่างหนึ่งที่ เราอาจจะให้นักเรียนชายอีกหนึ่งคนขึ้นไปยืนบนที่ว่างนั้น แต่ถ้าเราบอกให้นักเรียนทุกคนขยับไปทางซ้ายหนึ่งที่ คนที่อยู่ซ้ายสุดก็จะต้องออกจากที่ของตน และที่ทางขวามือริมสุดก็ว่าง เราก็ให้นักเรียนชายอีกคนขึ้นไปแทนที่ๆว่างนั้น ดังนั้นแถวของนักเรียนชายหญิงจึงใช้แทนที่ค่าของเลขฐานสองได้ ซึ่งขึ้นอยู่กับว่าในตำแหน่งใดมีนักเรียนชายหรือหญิงยืนอยู่

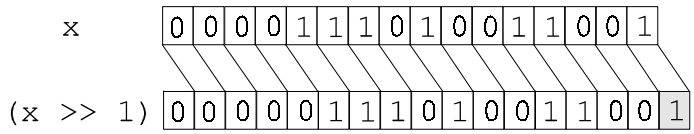
การเลื่อนแถวบิตของข้อมูลก็ไม่แตกต่างจากตัวอย่างข้างบนที่บอกให้นักเรียนในแถวขยับไปทางซ้ายหรือขวา และขยับไปกี่ตำแหน่ง ตำแหน่งที่ว่างก็ให้นักเรียนคนอื่นขึ้นไปแทนที่ จากนั้นก็

ดูว่า มีนักเรียนชาย (แทนสัญลักษณ์ด้วยศูนย์) หรือ หญิง (แทนสัญลักษณ์ด้วยหนึ่ง) ยืนอยู่ในที่ใดบ้าง เราลองมาดู ตัวอย่างการใช้โอเปอร์เรเตอร์ในการเลื่อนแถวบิตต่อไปนี้

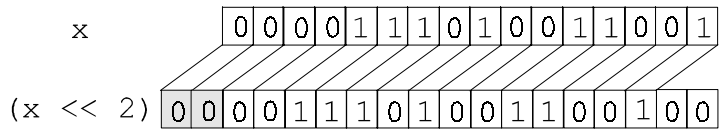
กำหนดให้ x เป็นตัวแปรแบบ `unsigned int` ขนาด 16 บิต ซึ่งมีค่าในขณะนั้นเท่ากับ `3737` เมื่อเราเขียนให้อยู่ในเลขฐานสองก็จะได้ดังนี้

$x = 3737_{10} = 0000111010011001_2$

ถ้าเราเลื่อนแถวบิตไปทางขวาหนึ่งตำแหน่ง เราก็จะอ่านค่าของ `x >> 1` ได้ตามรูปภาพข้างล่างนี้



หรือเลื่อนไปทางซ้ายสองตำแหน่ง ซึ่งก็คือ `x << 2`



จากตัวอย่างข้างบน เราสามารถเขียนค่าต่างๆจากการเลื่อนแถวบิตสรุปลงในตารางได้ดังนี้

I	(x << i)	(x >> i)
0	0000111010011001	0000111010011001
1	0001110100110010	0000011101001100
2	0011101001100100	0000001110100110
3	0111010011001000	0000000111010011
4	1110100110010000	0000000011101001
..	.....	.....
14	0100000000000000	0000000000000000
15	1000000000000000	0000000000000000

ตารางที่ 2.15 ตัวอย่างการใช้โอเปอร์เรเตอร์ในการเลื่อนแถวบิต

จากตารางแสดงลำดับการทำงานของโอเปอร์เรเตอร์ต่างๆ โอเปอร์เรเตอร์สำหรับการเลื่อนแถวบิต `<<` และ `>>` มีการจัดหมู่จากซ้ายไปขวา ดังนั้นในตัวอย่างนี้ ซึ่ง `A,B` และ `C` เป็นตัวแปรแบบ `unsigned int`

```
A =  1234;
B =  A << 2 << 5;
C =  A >> 3 << 2;
```

เราสามารถเขียนประโยคคำสั่งใหม่ที่ให้ผลเหมือนกันได้ดังนี้

```
A =  1234;
B =  (A << 2) << 5;        /* B == A << (2+5) */
```

```
C =  (A >> 3) << 2;        /* C != A >> (3-2) */
```

สำหรับค่าของ C ในประโยคคำสั่งข้างบนจะไม่เท่ากับประโยคคำสั่งข้างล่างนี้

```
C =  A >> (3-2);
```

ซึ่งถ้าเราเลื่อนแถวบิตไปทางขวา 3 ตำแหน่ง และเลื่อนกลับไปทางซ้าย 2 ตำแหน่งก็มิได้หมายความว่า จะเหมือนกับการเลื่อนบิตไปทางขวาเพียง 1 ตำแหน่ง ตัวอย่างเช่น

```
A  = 211;          /* 11010011 */
C0 = (A  >> 3);    /* 00011010 */
C1 = (C0 << 2);    /* 01101000 */
C2 = (A  >> 1);    /* 01101001 */
```

เราจะเห็นได้ว่า ค่าของ C1 และ C2 มีค่าแตกต่างกัน

ข้อสังเกต การเลื่อนแถวบิตไปทางขวาทั้งหมด i ตำแหน่งเป็นการหารค่าของข้อมูลในแถวบิตด้วย 2^i ส่วนการเลื่อนบิตไปทางซ้ายเป็นการคูณค่าของข้อมูลในแถวบิตด้วย 2^i ตัวอย่างเช่น เรากำหนดให้ x มีค่าเท่ากับ 55

```
 x       ≡ 00110111₂  =  55
(x >> 1) ≡ 00011011₂  = (55 / 2) =  27
(x << 1) ≡ 01101110₂  = (55 * 2) =  110
```

จากตัวอย่างที่ผ่านมาเราได้ใช้แต่จำนวนเต็มแบบ unsigned int (short หรือ long) เท่านั้นในการเลื่อนแถวบิต แต่ถ้าเราใช้ข้อมูลแบบ int ธรรมดา ผลที่ได้จะแตกต่างออกไปในบางกรณี เพราะฉะนั้นควรจะใช้แต่ข้อมูลแบบ unsigned int เท่านั้นในการเลื่อนแถวบิต

โอเปอร์เรเตอร์	ความหมาย	การจัดหมู่
!	การปฏิเสธในเชิงตรรกศาสตร์	จากขวาไปซ้าย
~	บิตคอมพลีเมนท์	จากขวาไปซ้าย
++	การเพิ่มค่าขึ้นอีกหนึ่ง	จากขวาไปซ้าย
--	การลดค่าลงอีกหนึ่ง	จากขวาไปซ้าย
*	การคูณ	จากซ้ายไปขวา
/	การหาร	จากซ้ายไปขวา
%	การหาค่าโมดูโล	จากซ้ายไปขวา
+	การบวก	จากซ้ายไปขวา
-	การลบ	จากซ้ายไปขวา
<<	การเลื่อนแถวบิตไปทางซ้าย	จากซ้ายไปขวา

>>	การเลื่อนแถวบิตไปทางขวา	จากซ้ายไปขวา
<	การเปรียบเทียบ ‘น้อยกว่า’	จากซ้ายไปขวา
<=	การเปรียบเทียบ ‘น้อยกว่า หรือเท่ากับ’	จากซ้ายไปขวา
>	การเปรียบเทียบ ‘มากกว่า’	จากซ้ายไปขวา
>=	การเปรียบเทียบ ‘มากกว่า หรือเท่ากับ’	จากซ้ายไปขวา
==	การเปรียบเทียบ ‘เท่ากับ’	จากซ้ายไปขวา
!=	การเปรียบเทียบ ‘ไม่เท่ากับ’	จากซ้ายไปขวา
&	การเปรียบเทียบบิตแบบ ‘AND’	จากซ้ายไปขวา
^	การเปรียบเทียบบิตแบบ ‘ XOR’	จากซ้ายไปขวา
|	การเปรียบเทียบบิตแบบ ‘OR’	จากซ้ายไปขวา
&&	การผูกเงื่อนไขเชิงตรรกศาสตร์แบบ ‘และ’	จากซ้ายไปขวา
||	การผูกเงื่อนไขเชิงตรรกศาสตร์แบบ ‘หรือ’	จากซ้ายไปขวา
? :	การสร้างนิพจน์เงื่อนไขสำหรับตัวเลือก	จากขวาไปซ้าย
=	การกำหนดค่า	จากขวาไปซ้าย

ตารางที่ 2.16 ลำดับการทำงานก่อนหลังของโอเปอร์เรเตอร์สำหรับข้อมูลแบบ `int`

2.1.12 การพิมพ์ข้อมูลเลขจำนวนเต็มออกทางจอภาพ

หลายๆโปรแกรมตัวอย่างที่เราได้ทำความรู้จักในบทนี้ มีการใช้ฟังก์ชันมาตราฐาน `printf()` ในการพิมพ์ข้อความออกทางจอภาพ สำหรับข้อมูลแบบ `int` (`short`, `long` หรือ `unsigned`) เราสามารถพิมพ์ค่าของข้อมูลเหล่านี้ออกทางจอภาพให้อยู่ในรูปแบบที่เราต้องการได้ โดยอาศัยลำดับควบคุมต่างๆต่อไปนี้ซึ่งเราจะใช้ร่วมกับฟังก์ชัน `printf()`

ลำดับควบคุม	ความหมาย
%d	ใช้แทนข้อมูลเลขจำนวนเต็มและแสดงผลในฐานสิบ
%u	ใช้แทนข้อมูลแบบ `unsigned` และแสดงผลในฐานสิบ
%o	ใช้แทนข้อมูลเลขจำนวนเต็มและแสดงผลในฐานแปด
%x หรือ %X	ใช้แทนข้อมูลเลขจำนวนเต็มและแสดงผลในฐานสิบหก
%hd	ใช้แทนข้อมูลแบบ `short` และแสดงผลในฐานสิบ

%ld	ใช้แทนข้อมูลแบบ long และแสดงผลในฐานสิบ
%hu	ใช้แทนข้อมูลแบบ unsigned short และแสดงผลในฐานสิบ
%lu	ใช้แทนข้อมูลแบบ unsigned long และแสดงผลในฐานสิบ

ตารางที่ 2.17 ลำดับควบคุมที่ใช้แทนข้อมูลแบบต่างๆเมื่อต้องการแสดงค่าโดยใช้ printf()

โปรดสังเกตว่า การแสดงข้อมูลออกทางจอภาพให้เป็นเลขฐานแปด และสิบหกนั้น จะไม่มีความแตกต่างระหว่าง (signed) int และ unsigned int

ตัวอย่างการใช้ลำดับควบคุมสำหรับข้อมูลที่เป็นเลขจำนวนเต็มภายในฟังก์ชัน printf()

```
#include <stdio.h>

int main()
{
      int       a;
      short     b;
      long      c;
      unsigned  d;

      a = -1000;
      printf ("%d %o %x %X\n", a, a, a, a);
      b = 0x00fa;
      printf ("%d %o %x %X\n", b, b, b, b);
      c = 12345678L;
      printf ("%ld %lo %lx %lX\n", c, c, c, c);
      d = 32768U;
      printf ("%u %o %x %X\n", d, d, d, d);
      return 0;
}
```

เราจะเห็นได้ว่า พารามิเตอร์ตัวแรกของฟังก์ชันจะเป็นข้อความที่มีลำดับควบคุมต่างๆเป็นส่วนหนึ่งข้อความนี้ (สังเกตได้จากสัญลักษณ์ %) และโปรดสังเกตว่า จำนวนของลำดับควบคุมทั้งหมดที่เราใช้นี้จะต้องเท่ากับจำนวนของพารามิเตอร์ที่ตามมา โดยเรียงตามลำดับที่พบ

ผลของโปรแกรมคือ

```
-1000 176030 fc18 FC18
250 372 fa FA
12345678 57060516 bc614e BC614E
32768 100000 8000 8000
```

นอกจากนี้เราสามารถจัดรูปแบบของตัวเลขให้สวยงามและเป็นระเบียบมากขึ้นโดยใช้สัญลักษณ์ต่อไปนี้ร่วมกับลำดับควบคุมที่แสดงอยู่ในตารางที่แล้ว แต่เราจะต้องกำหนดความยาวของข้อความ

ที่เป็นตัวเลขก่อน (ข้อความที่ประกอบด้วยเลขโดดและเครื่องหมายบวกหรือลบเท่านั้น) โดยเติมตัวเลขเท่ากับความยาวของข้อความหรือจำนวนหลักของตัวเลขที่ต้องการถัดจาก % ในลำดับควมคุม ตัวอย่างเช่น %8d หมายถึง เราต้องการพิมพ์ค่าของตัวเลขแบบ int ให้มีความยาวเท่ากับ 8 หลัก เนื่องจากว่า ค่าของข้อมูลแบบ int (16 บิต) มีอย่างมากไม่เกิน 5 หลัก ดังนั้น 3 หลักที่เหลือ (หรือมากกว่านี้ถ้าค่าของตัวเลขมีน้อยกว่า 5 หลัก) ก็จะเป็นที่ว่าง (Space) เพื่อทำให้ครบ 8 หลัก ถ้าเรากำหนดความยาวของข้อความหรือจำนวนหลักของตัวเลขในลำดับควบคุมแล้ว เราสามารถใช้สัญลักษณ์ต่อไปนี้ โดยใส่เพิ่มเติมเข้าไปเพื่อปรับแต่งรูปแบบของข้อความบนจอภาพให้ดีขึ้น

สัญลักษณ์	ความหมาย
-	พิมพ์ข้อความที่เป็นตัวเลขชิดขอบซ้าย
+	พิมพ์เครื่องหมายบวกหรือลบนำหน้าข้อความที่เป็นตัวเลข ถ้าปราศจากสัญลักษณ์นี้ในลำดับควบคุมก็จะพิมพ์เครื่องหมายลบนำหน้าตัวเลขที่มีค่าเป็นลบเท่านั้น ส่วนตัวเลขที่มีค่าบวกก็จะไม่มีเครื่องหมายบวกนำหน้า
0	เติมศูนย์ข้างหน้าในที่ว่างให้เต็ม

ตารางที่ 2.18 สัญลักษณ์เพิ่มเติมเพื่อปรับแต่งรูปแบบข้อมูลเมื่อใช้กับ printf()

ตัวอย่างการใช้เช่น

```
#include <stdio.h>

int main()
{
      int       a;
      long      b;
      unsigned  c;

      a = -1000;
      printf ("[%d]\n", a);
      printf ("[%8d]\n", a);
      printf ("[%-8d]\n", a);
      b = 0x00fa;
      printf ("[%ld]\n", b);
      printf ("[%10ld]\n", b);
      printf ("[%010ld]\n", b);
      printf ("[%+10ld]\n", b);

      c = 50000U;
      printf ("[%u]\n", c);
      printf ("[%8u]\n", c);
      printf ("[%08u]\n", c);
      printf ("[%08X]\n", c);
      return 0;
}
```

ผลจากโปรแกรมเป็นดังนี้

```
[-1000]
[   -1000]
[-1000   ]
[250]
[       250]
[0000000250]
[      +250]
[50000]
[   50000]
[00050000]
[0005C350]
```

2.2 แบบข้อมูลสำหรับเลขทศนิยม (Floating-Point Number)

ข้อมูลต่างๆในชีวิตประจำวันมักจะเป็นเลขทศนิยม เช่น อุณหภูมิเฉลี่ยที่วัดได้ในหนึ่งสัปดาห์มีค่าเท่ากับ 32.5 องศา ค่าของ sin (60°) มีค่าเท่ากับ $\sqrt{3}$ / 2 หรือ ประมาณ 0.866025 เหล่านี้เป็นต้น ดังนั้นเพื่อให้ใช้ข้อมูลลักษณะนี้ได้ในโปรแกรมคอมพิวเตอร์ เราจำเป็นต้องมีแบบข้อมูลสำหรับเลขที่ไม่ใช่จำนวนเต็ม ในภาษาซีแบบข้อมูลสำหรับเลขทศนิยม คือ `float` และ `double` ซึ่งมีขนาดตามปรกติแล้วเท่ากับ 32 บิต และ 64 บิตตามลำดับ

2.2.1 เลขทศนิยมที่เป็นค่าคงที่ในภาษาซี

ในภาษาซีเราสามารถเขียนตัวเลขทศนิยมต่างๆที่เป็นค่าคงที่ได้หลายๆรูปแบบตามตัวอย่างใน ตารางข้างล่างนี้

รูปแบบของค่าคงที่ในภาษาซี	รูปแบบของค่าคงที่ในทางคณิตศาสตร์
1.0	1.0
1234.	1234.0
-1234.567	-1234.567
1.234567e+3	1.234567×10^{3}
+6.022E23	6.022×10^{23}
1.66e-27	1.66×10^{-27}
-0.00340E+5	-340.0

ตารางที่ 2.19

ในภาษาซีเราจะใช้สัญลักษณ์ e หรือ E ในการแทนที่เลขกำลังฐานสิบ โปรดสังเกตว่า ค่าคงที่ใดๆที่เรากำหนดให้เป็นเลขทศนิยมในภาษาซีจะต้องมีจุดทศนิยมเสมอไม่ว่าค่าคงที่นั้นจะมีตัว

เลขหลังจุดทศนิยมหรือไม่ก็ตาม เช่น `1234.` และ `1234` แม้ว่าค่าคงที่ทั้งสองจะมีค่าเท่ากันแต่มีชนิดของข้อมูลที่แตกต่าง กันคือ `double` และ `int` ตามลำดับ

2.2.2 การเก็บเลขทศนิยมในหน่วยความจำของคอมพิวเตอร์

ใช่ว่าเราจะสามารถใช้ตัวแปรแบบ `double` หรือ `float` ในการเก็บค่าของเลขทศนิยมที่มีค่าเท่าไร ก็ได้ เพราะเนื่องจากขนาดของหน่วยความจำของตัวแปรหรือค่าคงที่นั้นมีขนาดที่แน่นอนและจำกัด ยกตัวอย่างเช่น ถ้าเราต้องการจะเก็บค่าของ `1/3` ให้เป็นค่าคงที่แบบ `double` แต่ตัวเลขจำนวนนี้เป็นเลขตรรกยะในรูปของเลขเศษส่วนและเราจะต้องแปลงให้เป็นเลขทศนิยมก่อน

`1/3 = 0.3333333333.....` ≈ `0.3333333333`

ซึ่งเป็นเลขทศนิยมที่มีเลขสามอยู่หลังจุดทศนิยมซ้ำนับไม่ถ้วน จำนวนเช่นนี้แม้ว่าจะมีค่าไม่มากก็จริงแต่ เราก็ไม่สามารถเก็บค่าของตัวเลขนี้ไว้ในหน่วยความจำของคอมพิวเตอร์ได้ทั้งหมดและถูกต้องตามที่ควรจะ เป็น เราจะต้องประมาณค่าของ 1/3 ก่อนให้มีค่าใกล้เคียงกับค่าที่ถูกต้องมากที่สุด และต้องสามารถเก็บไว้ ในหน่วยความจำของตัวแปรแบบ `double` ได้ด้วย คือเราจะต้องเขียนให้อยู่ในรูปแบบมาตราฐานขนาด 64 บิตได้

เราจะเห็นได้ว่าสำหรับเลขหลายๆจำนวนไม่ว่าจะเป็นเลขตรรกยะหรืออตรรกยะก็ตาม เราไม่สามารถเก็บค่าที่ถูกต้องของมันไว้ในหน่วยความจำได้ แต่จะเป็นค่าประมาณเท่านั้น สมมุติว่าเราสามารถเก็บตัวเลขหลังจุดทศนิยมได้ไม่เกิน 10 ตำแหน่งเท่านั้น จากค่าของ 1/3 เราจึงประมาณค่าให้เท่ากับ 1/3 ≈ 0.3333333333 โปรดสังเกตว่าในกรณีนี้ค่าประมาณของ 1/3 มีค่าน้อยกว่าค่าจริง สำหรับเลขจำนวนจริงใดๆ เราสามารถเขียนให้อยู่ในเลขทศนิยมได้ และดังที่กล่าวไปในข้างต้น บางจำนวนเราจะต้องประมาณค่าก่อนจึงจะใช้เป็นข้อมูลแบบ `float` หรือ `double` ในภาษาซีได้ สำหรับการทำงานของคอมพิวเตอร์ ทุกๆตัวเลขจะต้องแปลงหรือเขียนให้อยู่ในเลขฐานสอง ในกรณีของเลขจำนวนเต็มเราได้ทำความรู้จักกับการเขียนเลขฐานสิบให้เป็นเลขฐานสองหรือฐานอื่นๆแล้ว เช่น ฐานแปดหรือฐานสิบหก คราวนี้เราจะมาลองเขียนเลขทศนิยมฐานสิบให้อยู่ในระบบเลขฐานสอง ตัวอย่างเช่น เลขทศนิยม 39.75 เราสามารถเขียนให้อยู่ในเลขฐานสิบและฐานสองแบบกระจายได้ดังนี้

$$
\begin{aligned}
39.75 &= (3 \times 10^{1}) + (9 \times 10^{0}) + (7 \times 10^{-1}) + (5 \times 10^{-2}) \\
&= 39 + 0.50 + 0.25 = 32 + 4 + 2 + 1 + 1/2 + 1/4 \\
&= (1 \times 2^{5}) + (1 \times 2^{2}) + (1 \times 2^{1}) + (1 \times 2^{0}) + (1 \times 2^{-1}) + (1 \times 2^{-2}) \\
&= 100111.11_{2}
\end{aligned}
$$

จากตัวอย่างข้างบน เรากล่าวได้ว่าเลขทศนิยมใดๆจะต้องสามารถแสดงให้อยู่ในรูปแบบต่อไปนี้ได้

$$\pm \left(\sum_{i=0}^{N} a_i B^i + \sum_{i=1}^{P} b_i B^{-i} \right)$$

- B เป็นเลขฐาน เช่น B = 2 หมายถึงเลขฐานสอง
- N และ P เป็นเลขจำนวนเต็ม โดยที่ N+1 บอกถึงจำนวนของตัวเลขทั้งหมดที่อยู่หน้าจุดทศนิยม และ P คือจำนวนของตัวเลขทั้งหมดที่อยู่หลังจุดทศนิยม
- a_i และ b_i เป็นเลขโดด (จำนวนเต็ม) ที่มีค่าอยู่ระหว่าง 0, 1,...., B-1

ดังนั้น 100111.11_2 จึงแสดงให้อยู่ในรูปแบบมาตราฐานได้ โดยที่ B=2, N=5 และ P=2

อย่างไรก็ตาม เราสามารถเลื่อนจุดทศนิยมไปทางซ้ายหรือขวาก็ได้ แต่เราจะต้องคูณด้วยค่าคงที่ (อยู่ในรูปของเลขยกกำลังฐานสอง) ที่เหมาะสมเพื่อทำให้ค่าของเลขทศนิยมนี้ไม่เปลี่ยนแปลง โปรดดูภาพประ-กอบข้างล่าง ในแต่ละกรณีจะมีรูปแบบที่แตกต่างกันแต่ให้ค่าเท่ากัน

1	0	0	1	1	1	.	1	1	$\times 2^0$

1	0	0	1	1	.	1	1	1	$\times 2^1$

1	.	0	0	1	1	1	1	1	$\times 2^5$

.	1	0	0	1	1	1	1	1	$\times 2^6$

การกำหนดรูปแบบของข้อมูลแบบ `float` หรือ `double` ให้เป็นมาตราฐานนั้น อาจจะแตกต่างกันไป โดยขึ้นอยู่กับโครงสร้างและวิธีการทำงานของคอมพิวเตอร์แต่ละชนิด แต่เราจะกล่าวถึงเฉพาะรูปแบบมาตราฐานที่กำหนดโดย IEEE (Institute of Electrical and Electronic Engineers) เท่านั้น เมื่อเราเขียนเลขทศนิยมให้เป็นเลขฐานสองแล้ว เราจะต้องแปลงให้อยู่ในรูปแบบมาตราฐานดังนี้

$$\text{floating point} = (-1)^S \cdot (1.M)_2 \cdot 2^{(e-E)}$$

$$\text{floating point} = \text{significand} \times \text{radix}^{\text{exponent}}$$

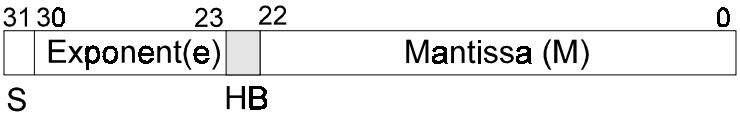
คำอธิบาย

- S คือ ตัวเลขที่กำหนดว่ารูปแบบข้างบนใช้แทนค่าของเลขทศนิยมที่มีค่าเป็นบวกหรือลบ ถ้า S = 0 หมายถึง เลขที่มีค่าเป็นบวก (หรือศูนย์) และถ้า S = 1 หมายถึงเลขที่มีค่าเป็นลบ
- M เป็นค่าแมนทิสซ่า (Mantissa) ซึ่งเป็นเลขจำนวนเต็มที่มีค่าบวกหรือศูนย์เท่านั้น
- e - E เป็นเลขกำลังของฐานสอง เนื่องจากได้มีการกำหนดให้ e เป็นเลขจำนวนเต็มบวก หรือศูนย์เท่านั้น ดังนั้นเพื่อที่จะสามารถใช้กับเลขกำลังที่มีค่าเป็นลบได้เราจึงต้องลบออกด้วย E ซึ่งเป็นค่าคงที่มีค่ามากกว่าศูนย์ เช่น สำหรับข้อมูลแบบ `float` ขนาด 32 บิต ค่าของ E ตามมาตราฐานของ IEEE จะเท่ากับ 127 (เรียกว่า 127-excess) และ e มีขนาด 8 บิต ส่งผลให้เลขกำลัง (e - E) มีค่าอยู่ระหว่าง

$$-E \le e - E \le e_{max} - E$$

แต่มีข้อยกเว้นสำหรับเลขทศนิยมที่มีค่าเท่ากับศูนย์ ซึ่งเรากำหนดให้ s = 0, M = 0 และ e = 0 ตามลำดับ

ตัวอย่างเช่น เรากำหนดให้ 39.75 เป็นข้อมูลแบบ `float` ขนาด 32 บิต โครงสร้างของข้อมูลแบบ `float` จะเป็นดังนี้



คำอธิบาย

M	เป็นค่าของแมนทิสซ่าขนาด 24 บิต โดยมีบิตหนึ่งตัวเป็น *บิตซ่อน* หรือ Hidden Bit (HB) อยู่ทางซ้ายสุดในแถวบิตของแมนทิสซ่า
HB	มีค่าเป็นหนึ่งเสมอ ยกเว้นในกรณีที่เลขทศนิยมมีค่าเป็นศูนย์ HB จะมีค่าเป็นศูนย์ (และ M จะถูกกำหนดให้มีค่าเท่ากับศูนย์ด้วย)
S	เป็นบิตที่บอกว่า เลขทศนิยมมีเครื่องหมายบวกหรือลบอยู่ข้างหน้า
e	เป็นเลขกำลังของฐานสองขนาด 8 บิต

จะเห็นได้ว่า เราใช้หน่วยความจำรวมทั้งหมด 33 บิต แต่ในทางปฏิบัติเราจะไม่บันทึก HB บิตลงในหน่วยความจำทำให้จำนวนบิตลดลงเหลือ 32 สำหรับข้อมูลแบบ `float` เหตุผลก็คือว่าเรา

สามารถเขียนเลขทศนิยมทุกๆจำนวนยกเว้นศูนย์ ให้อยู่ในรูปแบบมาตราฐานที่มีเลขหนึ่งอยู่ในตำแหน่งแรกหน้าจุดทศนิยมคูณ ด้วยเลขกำลังฐานสองได้ ตัวอย่างเช่น

$+11.01100110011_2 = +1.10110011011_2 \times 2^1$
$-0.00101001_2 = -1.01001_2 \times 2^{-3}$

ดังนั้นบิตนี้เราจึงไม่จำเป็นต้องบันทึกลงในหน่วยความจำ แต่เลขทศนิยมที่มีค่าเท่ากับศูนย์จัดเป็นข้อยกเว้นเพราะเหตุที่ว่า ตัวเลขทุกๆตัวเป็นศูนย์จึงไม่สามารถเขียนให้มีเลขหนึ่งอยู่ข้างหน้าได้ ดังนั้นจึงได้มีการกำหนดให้แมนทิสซ่าและค่าของเลขกำลังให้มีค่าเป็นศูนย์พร้อมกัน

เราสามารถหาค่าแมนทิสซ่า (หรือที่เรียกว่า significand) และเลขกำลัง (exponent) ของฐานสอง (radix = 2) ของข้อมูลแบบ `double` ได้โดยใช้ฟังก์ชันมาตราฐานชื่อ `frexp()` ซึ่งมีส่วนหัวของฟังก์ชันอยู่ในไฟล์ `<math.h>` มีลักษณะดังนี้

```
double frexp (double x, int *exponent);
```

โดยที่ `x` เป็นตัวแปรแบบ `double` และเราต้องการหาค่าแมนทิสซ่าของ `x` ในเลขฐานสิบซึ่งเป็นค่าของ ฟังก์ชันแบบ `double` เนื่องจากว่าเราต้องการหาค่าของเลขกำลัง (exponent) ด้วย แต่ฟังก์ชันให้ค่า เป็นข้อมูลแบบ `double` สำหรับแมนทิสซ่าเท่านั้น ดังนั้นจึงต้องหาวิธีการที่จะผ่านตัวแปรให้ฟังก์ชันเพื่อ ที่จะได้ใช้เก็บค่าของเลขกำลังที่คำนวณได้ในฟังก์ชัน และวิธีการที่ง่ายก็คือการใช้พอยน์เตอร์กับตัว แปร `exponent` (ทำไมจึงต้องใช้พอยน์เตอร์? เราจะตอบคำถามนี้เมื่อเราเริ่มต้นทำความเข้าใจการทำงานของพอยน์เตอร์อย่างละเอียดในบทต่อๆไป)

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

int main()
{
  double realNumber, significand;
  int    exponent;

  realNumber = 1.75;
  significand = frexp (realNumber, &exponent);
  printf ("%13.10lf * 2^%d = %lf\n", significand,
    exponent, significand * pow(2,exponent));

  realNumber = -0.0005;
  significand = frexp (realNumber, &exponent);
  printf ("%13.10lf * 2^%d = %lf\n", significand,
    exponent, significand * pow(2,exponent));

  realNumber = +39.75;
  significand = frexp (realNumber, &exponent);
  printf ("%13.10lf * 2^%d = %lf\n", significand,
    exponent, significand * pow(2,exponent));

  return 0;
}
```

ในโปรแกรมตัวอย่างข้างบน เราใช้ฟังก์ชันมาตราฐานชื่อ `pow()` ในการหาค่าของเลขยกกำลังฐานสอง ซึ่งค่าของจำนวนจริงแบบ `double` เราสามารถคำนวณได้ตามสูตรข้างล่างนี้โดยที่ `significand` เป็นเลข ทศนิยมฐานสิบซึ่งมีค่าอยู่ระหว่าง -1 และ 1 (ไม่รวม 1 และ -1) เท่านั้น

$$\text{significand} \times 2^{\text{exponent}} = \text{floating-point number}$$

ผลของโปรแกรมคือ

```
 0.8750000000 * 2^1 = 1.750000
-0.5120000000 * 2^-10 = -0.000500
 0.6210937500 * 2^6 = 39.750000
```

2.2.3 โอเปอร์เรเตอร์สำหรับเลขทศนิยม

โอเปอร์เรเตอร์สำหรับเลขทศนิยมนั้นมีบางตัวที่แตกต่างจากในกรณีของเลขจำนวนเต็ม โดยเฉพาะโอเปอร์เรเตอร์เชิงคณิต เช่น โอเปอร์เรเตอร์ในการหาค่าโมดูโลจะไม่มีสำหรับเลขทศนิยม นอกจากนี้การหารระหว่างเลขทศนิยมก็เป็นไปตามปรกติที่เราเข้าใจ คือไม่มีการปัดเศษใดๆทิ้งเหมือนกับการหารระหว่างเลขจำนวนเต็ม ส่วนโอเปอร์เรเตอร์อื่นๆ เช่น โอเปอร์เรเตอร์เชิงเปรียบเทียบหรือเชิงตรรกศาสตร์เราก็ สามารถใช้กับเลขทศนิยมได้เหมือนกับที่เราใช้กับเลขจำนวนเต็ม ยกเว้นการเลื่อนบิตและการคำนวณแบบ บิตต่อบิตเราไม่สามารถใช้กับเลขทศนิยมได้

ในบางครั้ง เราต้องการเขียนโปรแกรมที่ใช้สำหรับการคำนวณทางคณิตศาสตร์ เช่น หาค่าของเลขยกกำลัง การหารากที่สองของเลขจำนวนจริงใดๆ การหาค่าทางตรีโกณมิติ เหล่านี้เป็นต้น ในภาษาซีจะไม่มีโอเปอร์เรเตอร์ที่ทำหน้าที่ในลักษณะนี้ แต่เราจะเรียกใช้ฟังก์ชันมาตราฐานทางตณิตศาสตร์ในภาษาซีซึ่งมีการกำหนดส่วนหัวของฟังก์ชันทางตณิตศาสตร์เหล่านี้ไว้ในไฟล์ชื่อ `<math.h>` และได้มีการสร้างส่วนตัวของฟังก์ชันที่ถูกคอมไพล์แล้วเก็บไว้ในคลังฟังก์ชัน ถ้าเป็นระบบยูนิกซ์ จะเก็บไว้ในไฟล์ชื่อ `libm.a` ดังนั้นเวลาใช้คำสั่ง `cc` สำหรับการแปลโปรแกรมโค้ดจะต้องเติมพารามิเตอร์ `-lm` เช่น

```
cc  prog01.c -o prog01  -lm
```

ซึ่งเวลาคอมไพล์เลอร์ประกอบส่วนต่างๆของโปรแกรมเข้าด้วยกันก็จะค้นหาส่วนหนึ่งในไฟล์ชื่อ `libm.a` ซึ่งมักจะพบอยู่ที่ `/usr/lib/` สำหรับผู้ที่ใช้ชุดคอมไพล์เลอร์ของ Borland ก็ไม่ต้องกังวล

เกี่ยวกับเรื่องนี้เพราะคอมไพล์เลอร์จะจัดการขั้นตอนเหล่านี้โดยอัตโนมัติ เพียงแค่กดปุ่มเมาส์เท่านั้น เราก็จะได้โปรแกรมสำเร็จรูปที่ทำงานได้

2.2.4 การแสดงข้อมูลสำหรับเลขทศนิยมบนจอภาพ

ในการแสดงข้อมูลแบบ `float` หรือ `double` ออกทางจอภาพ เรานิยมใช้ฟังก์ชันมาตราฐาน `printf()` เช่นเดียวกับเวลาเราใช้กับข้อมูลแบบ `int` สำหรับข้อมูลแบบ `int` นี้เราใช้สัญลักษณ์ `%d` หรือที่เราเรียกว่า ลำดับควบคุม (Control Sequence) ในการแทนที่ตัวเลขแบบ `int` (`short` หรือ `long`) หรือลำดับควบคุม `%u` สำหรับข้อมูลแบบ `unsigned int` (`short` หรือ `long`) แต่ถ้าเป็นข้อมูลที่เป็นเลขทศนิยมแบบ `float` และ `double` เราจะใช้ลำดับควบคุม `%f` และ `%lf` ตามลำดับ

นอกจากนี้เราสามารถกำหนดความยาวของข้อความที่ใช้แสดงค่าของเลขทศนิยมได้ เช่น กำหนดให้แสดงเลขทศนิยมมีตำแหน่งหรือดิจิตอยู่ข้างหน้าและหลังจุดทศนิยมกี่ตำแหน่ง ตัวอย่างเช่น

```
#include <stdio.h>

int main()
{
      double  value;

      value = 1079.12345;
      printf ("Value = %8.3lf\n",  value);
      printf ("Value = %10.4lf\n", value);
      printf ("Value = %11.7lf\n", value);
      return 0;
}
```

ซึ่งแสดงผลออกทางจอภาพดังนี้

```
Value = 1079.123
Value =  1079.1235
Value = 1079.1234500
```

จะเห็นได้ว่าจำนวนของตัวเลขข้างหน้าและข้างหลังจุดทศนิยมแตกต่างกันไปโดยขึ้นอยู่กับลำดับควบคุมที่เราใช้ซึ่งมีตัวเลขกำกับตามรูปแบบทั่วไปดังนี้

```
%n.mlf      %.mlf       %n.lf
%n.mf       %.mf        %n.f
```

ซึ่ง `n` หมายถึงจำนวนของตัวเลขหรือตำแหน่งทั้งหมดที่อยู่ทั้งข้างหน้าและข้างหลังจุดทศนิยม รวมทั้งจุด ทศนิยมด้วย `m` หมายถึงจำนวนของตัวเลขที่อยู่หลังจุดทศนิยม ดังนั้น `n-m-1` จึงหมายถึง

จำนวนของ ตัวเลขที่อยู่ข้างหน้าจุดทศนิยม ถ้าจำนวนของตัวเลขทศนิยมมีน้อยกว่าค่าของ n-m-1 และ m ฟังก์ชัน printf() ก็จะเติมที่ว่าง (space) ข้างหน้าในแถวตัวเลขหรือเลขศูนย์ต่อท้ายให้ครบ (โปรดสังเกตว่า ฟังก์ชันจะแสดงข้อความที่ไม่ถูกต้องถ้า m มีค่ามากเกินไป เช่น m = 20) แต่ถ้า m มีค่าน้อยกว่าจำนวนของตัวเลขข้างหลังจุดทศนิยม ฟังก์ชัน printf() ก็จะประมาณค่าของตัวเลขข้างหลังจุดทศนิยมที่เกินมาหรือปัดทิ้งไป เช่น เราต้องการให้ฟังก์ชันแสดงผลของเลขทศนิยม 1079.12345 ที่มี จำนวนสี่หลักเท่านั้น หลังจุดทศนิยม เนื่องจากตัวเลขในหลักที่ห้ามีค่าเป็น 5 เราจึงปัดขึ้น กลายเป็น 1079.1235 โปรดสังเกต ว่า ถ้าตัวเลขข้างหน้าจุดทศนิยมมีมากกว่า n-m-1 ก็จะไม่มีผลใดๆ

เพื่อให้เกิดความเข้าใจมากขึ้น ขอให้ผู้อ่านลองคอมไพล์และรันโปรแกรมตัวอย่างข้างบน และ เปลี่ยนค่าของ n และ m เพื่อที่จะดูว่าผลที่แสดงบนจอภาพมีการเปลี่ยนแปลงอย่างไร เช่น

```
printf ("Value = %9.3lf\n",  value);
printf ("Value = %10.4lf\n", value);
printf ("Value = %11.7lf\n", value);
printf ("Value = %.5lf\n",   value);
printf ("Value = %11.0lf\n", value);
printf ("Value = %5.2lf\n",  value);
printf ("Value = %7lf\n",    value);
printf ("Value = %7.lf\n",   value);
printf ("Value = %lf\n",     value);
printf ("Value = %f\n",      value);
printf ("Value = %.20lf\n",  value);
```

ถ้าเราใช้ลำดับควบคุม %f หรือ %lf โดยไม่มีตัวเลขกำกับ ฟังก์ชัน printf() ก็จะกำหนดให้แสดง ข้อมูลที่มีจำนวนของตัวเลขหลังจุดทศนิยมเท่ากับ 6 หลักเท่านั้น

นอกจากนี้เรายังสามารถแสดงผลของเลขทศนิยมให้อยู่ในรูปของแมนทิสซ่าคูณด้วยเลขกำลังฐานสิบ โดยใช้ลำดับควบคุม %e หรือ %E ภายในข้อความที่เป็นพารามิเตอร์แรกของฟังก์ชัน printf() ตัวอย่างเช่น

```
printf ("%e\n",  12345.6789);
printf ("%E\n",  .00123456789);
printf ("%.10E\n", 0.00123456789);
```

2.2.5 ความแตกต่างระหว่างข้อมูลแบบ float และ double

นอกจากข้อมูลแบบ float และ double จะมีความแตกต่างในเรื่องของขนาดของหน่วยความจำที่ใช้แล้วยังมีความแตกต่างในเรื่องของความแม่นยำถูกต้อง (Precision) ในการคำนวณซึ่งเห็นได้จากตัวอย่างต่อไปนี้

```
#include <stdio.h>

int main()
{
      float  float_number;
      double double_number;

      float_number   = 123.45 / 3.30;
      float_number  *= 3.30;
      double_number  = 123.45 / 3.30;
      double_number *= 3.30;
      printf ("%f\n", float_number);
      printf ("%lf\n", double_number);
      return 0;
}
```

ผลของโปรแกรมคือ

```
123.449997
123.450000
```

จากตัวอย่างข้างบน เราจะเห็นได้ว่านิพจน์แบบเดียวกันให้ผลที่แตกต่างกัน คำตอบที่ถูกต้องจะต้องเท่ากับ 123.45 ซึ่งตัวแปรแบบ `double` เท่านั้นที่ให้ผลที่ถูกต้อง ส่วนตัวแปรแบบ `float` ให้ค่าที่ใกล้เคียงเท่านั้น ดังนั้นถ้าเราต้องการใช้ตัวแปรในการคำนวณที่มีความแม่นยำสูง เราก็ใช้ตัวแปรแบบ `double` หรือ `long double` (double-precision variable)

ในไฟล์ชื่อ `<float.h>` ได้มีการนิยามค่าคงที่ต่างๆไว้ที่ให้ข้อมูลเกี่ยวแบบข้อมูลสำหรับเลขจำนวนจริง เช่น จำนวนตำแหน่งของตัวเลขหลังจุดทศนิยมที่มากที่สุดสำหรับข้อมูลแบบต่างๆ เลขกำลังฐานสอง (exponent) ที่มากที่สุดค่าความแม่นยำของข้อมูลหรือที่เรียกว่า Machine Epsilon เป็นต้น ตารางข้างล่างเหล่านี้แสดงค่าคงที่ต่างๆที่นิยามไว้ใน `<float.h>` (จากซอฟต์แวร์คอมไพล์เลอร์ภาษาซีของ Borland International, Inc.)

ตารางที่ 2.20 รายละเอียดเพิ่มเติมสำหรับแบบข้อมูลต่างๆในภาษาซี

ตัวระบุชื่อ	จำนวนดิจิตมากที่สุด (ฐานสิบ) หลังจุดทศนิยม
FLT_DIG	7
DBL_DIG	15
LDBL_DIG	19

ตัวระบุชื่อ	จำนวนบิตที่ใช้เก็บค่าแมนทิสซ่า
FLT_MANT_DIG	24
DBL_MANT_DIG	53
LDBL_MANT_DIG	64

ตัวระบุชื่อ	ค่าความแม่นยำของข้อมูล (Machine Epsilon)	
FLT_EPSILON	1.19209290E-07	(=2^{-23})
DBL_EPSILON	2.2204460492503131E-16	(=2^{-52})
LDBL_EPSILON	1.084202172485504E-19	(=2^{-63})

ตัวระบุชื่อ	ค่าของจำนวนจริง(บวก)ที่น้อยที่สุด	
FLT_MIN	1.17549435E-38	(=(2-2^{-23})*2^{-127})
DBL_MIN	2.2250738585072014E-308	(=(2-2^{-52})*2^{-1023})

โปรดสังเกตว่าข้อมูลแบบ `float` และ `double` จะใช้ 127-excess และ 1023-excess ตามลำดับ

ตัวระบุชื่อ	ค่าของเลขกำลัง (exponent) ฐานสอง ที่มากและน้อยที่สุด
FLT_MAX_EXP	+128
DBL_MAX_EXP	+1024
LDBL_MAX_EXP	+16384
FLT_MIN_EXP	-125
DBL_MIN_EXP	-1021
LDBL_MIN_EXP	-16381

ตัวระบุชื่อ	ค่าของเลขกำลัง (exponent) ฐานสิบที่มากและน้อยที่สุด
FLT_MAX_10_EXP	+38
DBL_MAX_10_EXP	+308
LDBL_MAX_10_EXP	+4932
FLT_MIN_10_EXP	-37
DBL_MIN_10_EXP	-307
LDBL_MIN_10_EXP	-4931

แบบฝึกหัดท้ายบท

1. จงคำนวณค่าของนิพจน์ต่อไปนี้

   1.1) `13 / 4`

   1.2) `13 % 4`

   1.3) `-13 * 4`

   1.4) `-13 * 4 + 8`

   1.5) `3 * ++ 3 / 4`

   1.6) `10 * 7 / 4`

   1.7) `10 * (7 / 4)`

   1.8) `5.0 * (6 % 4)`

   1.9) `1.9 + '8'`

2. จงเปลี่ยนเลขฐานต่อไปนี้ในภาษาซีให้เป็นเลขฐานสิบ

   2.1) `011`

   2.2) `0xb7`

   2.3) `0XA`

   2.4) `0x1aL`

   2.5) `0644U`

   2.6) `0x7ffffUL`

3. จงค่าของนิพจน์ต่อไปนี้ โดยที่ `x` เป็นตัวแปรแบบ `int`

   3.1) `((5 <= x) <= 10)`

   3.2) `((5 <= x) && (x <= 10))`

   3.3) `(((x = 5.0) && 1) || ++x) - x`

   3.4) `((x=5) ? ++x : --x)`

   3.5) `(1U << 8)`

4. จงเขียนโปรแกรมที่ใช้โอเปอร์เรเตอร์ `sizeof` ในการหาขนาดของหน่วยความจำของข้อมูลแบบ `char, unsigned char, int, unsigned, long, short, float, double, long double`

5. จงเขียนโปรแกรมที่พิมพ์ค่าของจำนวนเต็มต่อไปนี้ออกทางจอภาพโดยให้อยู่ในเลขฐานแปดและสิบหก

```
1234
-32767
0xfa01
037
'%'
```

6. จงเขียนฟังก์ชันที่ใช้ตรวจสอบว่า บิตของข้อมูลแบบ `unsigned` ในตำแหน่งที่เรากำหนดมีค่าเป็นศูนย์หรือไม่ และให้ผ่านค่าของบิต (หมายถึงค่าเท่ากับศูนย์หรือหนึ่ง) ในตำแหน่งนี้เป็นค่าของฟังก์ชันแบบ `int` ซึ่งกำหนดให้มีส่วนหัวของฟังก์ชันดังนี้

```
int getBit(unsigned int data, int index);
```

โดยที่พารามิเตอร์ `index` เป็นตำแหน่งของบิต ซึ่งมีค่าระหว่าง `0` และ `sizeof(unsigned int)`

7. จงหาค่าของนิพจน์ต่อไปนี้

```
(int) 3.99999
(int)-123.45
(int)-3.99999
(char)3.99999
(unsigned char)-1.1
(unsigned char)-1
(x = (int)1.5 * 5)
(x = (int)((char)1.5 * 5))
```

8. ประโยคคำสั่งต่อไปนี้เป็นการใช้ฟังก์ชันมาตราฐาน `printf()` แต่มีการใช้แบบข้อมูลที่เป็นพารามิเตอร์ของฟังก์ชันไม่ถูกต้องตามที่ได้กำหนดไว้โดยลำดับควบคุมในข้อความที่เป็นพารามิเตอร์ตัวแรก

```
printf ("%d %lf \n", 1.0, 1);
printf ("%d %u %ld %f \n", 'A', 'A', 'A', 'A');
```

ขอให้ผู้อ่านลองเขียนโปรแกรมสั้นๆที่แสดงผลของประโยคคำสั่งทั้งสองบนจอภาพ และสังเกตว่ามีอะไรไม่ถูกต้องตามที่คาดหวังไว้หรือไม่

9. จงอธิบายความแตกต่างระหว่างนิพจน์ต่อไปนี้

```
(x && 0x01)
(x &  0x01)
(x || 0x000f)
(x |  0x000f)
```

และหาค่าของนิพจน์เหล่านี้เมื่อ x มีค่าเท่ากับ `0,1,-10` ตามลำดับ

10. จงคอมไพล์และรันโปรแกรมต่อไปนี้

```
int main()
{
      int      x = 0xdc0b;    /* 1101 1100 0000 1011 */
      unsigned y = 0xdc0bU;   /* 1101 1100 0000 1011 */

      printf ("x = %d, y = %u\n", x, y);
      printf ("The value of (((x >> 2) & 0x8000) >> 15) is %d.\n",
                  ((x >> 2) & 0x8000) >> 15);
      printf ("The value of (((y >> 2) & 0x8000) >> 15) is %d.\n",
                  ((y >> 2) & 0x8000) >> 15);
      return 0;
}
```

และสังเกตผลของโปรแกรมที่แสดงออกทางจอภาพ

11. จงพิจารณาประโยคคำสั่งต่อไปนี้

```
X ^= Y;
Y ^= X;
X ^= Y;
```

โดยที่กำหนดให้ตัวแปร X และ Y เป็นข้อมูลแบบ int เช่น สมมุติว่า เราสามารถเขียนค่าเริ่มต้นของ X และ Y คือ 10110111_2 และ 10011101_2 ตามลำดับในระบบฐานสอง ลองทำตามประโยคแต่ละขั้นแล้วดูว่า ค่าของ X และ Y หลังจากที่ดำเนินการแล้วมีค่าใหม่เป็นเท่าไร

12. จงให้เหตุผลว่า ทำไมประโยคคำสั่งต่อไปนี้จึงผิดหลักไวยกรณ์ในภาษาซี

```
x++ = 1;
x   = ++(x-1);
x/2 = (x-- -1);
```

13. จงเติมช่องว่างในตารางข้างล่างนี้ให้สมบูรณ์และถูกต้อง

X	Y	X<Y	X<=Y	X>Y	X>=Y	X==Y	X!=Y	X<<Y	X>>Y	X==Y
3	3									
3	4									
4	3									

X	Y	X && Y	X || Y	!X	!Y	X & Y	X | Y	X != 0 && Y !=0
0	0							
0	7							
5	0							
5	7							
8	7							