Chapter 12 Operations on Bits

- ภาษา C ได้ถูกพัฒนาขึ้นโดยมีจุดประสงค์เพื่อการเขียนโปรแกรมควบคุมระบบคอมพิวเตอร์ได้โดยตรง
- Pointer เป็นตัวอย่างหนึ่งของภาษา C ที่อนุญาตให้ผู้เขียนโปรแกรมสามารถเข้าถึงและควบคุมหน่วยความจำของคอมพิวเตอร์
- นอกจากนี้ภาษา C ยังอนุญาตให้มีการเข้าถึงและเปลี่ยนแปลงข้อมูลในระดับบิทได้ โดยมี operator ที่จำเป็นสำหรับการกระทำระดับบิทให้
- โดยปกติแล้วข้อมูลหนึ่งไบท์ (Byte: B) ประกอบด้วยข้อมูลย่อยแปดบิท (bit: b) โดยในแต่ละบิทสามารถมีค่าได้เป็น 1 หรือ 0
- สำหรับหน่วยความจำที่ตำแหน่งหนึ่งๆ เราสามารถเก็บข้อมูลได้ 1 ไบท์ หรือ 8 บิท ยกตัวอย่างเช่น 01100100
- โดยบิทด้านขวาสุดของไบท์ จะเรียกว่า Least Significant Bit (LSB) และด้านซ้ายสุดเรียก Most Significant Bit (MSB)
- หากเรากำหนดว่าไบท์นั้นเก็บตัวเลขจำนวนเต็ม บิทด้านขวาสุดจะมีค่า 2^0 หรือ 1 และบิทถัดมาจะเป็น $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, ...
- ดังนั้น 01100100 หากแปลงเป็นเลขจำนวนเต็ม จะได้เท่ากับ $2^6+2^5+2^2 = 64+32+4 = 100$
- ส่วนเลขจำนวนเต็มลบจะเก็บลงในหน่วยความจำในรูป ‘two’s completment’ นั่นคือ MSB จะเป็นบิทสำหรับเครื่องหมาย หาก MSB = 1 จะถือว่าตัวเลขนั้นเป็นเลขจำนวนเต็มลบ และหาก MSB = 0 จะเป็นเลขจำนวนเต็มบวก

- การเขียนเลขจำนวนเต็มลบในรูปแบบ ‘two’s complement’ นั้น เริ่มจากบวกค่า 1 เข้ากับจำนวนเต็มลบ แล้วแปลงค่าสัมบูรณ์ของเลขจำนวนเต็มลบนั้นให้อยู่ในรูปเลขฐานสอง จากนั้นให้ทำ complement กับทุกบิท นั่นคือสลับ 1 เป็น 0 และสลับ 0 เป็นหนึ่ง
- ยกตัวอย่างเช่นเลข -5 แปลงเป็นเลขในรูปแบบ two’s complement โดยบวกเพิ่มขึ้น 1 เป็น -4 จากนั้นแปลงค่าสัมบูรณ์ของ -4 เป็นเลขฐานสองนั่นคือ 00000100 และ complement บิททั้งหมด จะได้ 11111011
- สำหรับการแปลงกลับจาก two’s complement ให้ complement บิททั้งหมด จากนั้นแปลงเลขเป็นเลขฐานสิบ เปลี่ยนเครื่องหมายให้เป็นลบ และลบด้วยหนึ่ง
- การเก็บเลขในรูปแบบ two’s complement จำนวนบวกที่มากที่สุดที่สามารถเก็บได้ในหน่วยความจำ n บิทเท่ากับ 2^{n-1}-1 ในกรณีของ 8 บิทจะเก็บจำนวนบวกมากที่สุดเท่ากับ 2^7-1 หรือ 127 ส่วนจำนวนที่น้อยที่สุดที่สามารถเก็บได้ จะเท่ากับ -2^{n-1} และสำหรับ 8 บิทจะได้ -128
- คอมพิวเตอร์ส่วนมากในปัจจุบันจะเก็บเลขจำนวนเต็มโดยใช้พื้นที่ 4 ไบท์ หรือ 32 บิท ดังนั้นจำนวนบวกที่มากที่สุดจะเท่ากับ 2^{31}-1 เท่ากับ 2,147,483,647 และจำนวนลบที่น้อยที่สุดจะเท่ากับ -2,147,483,648
- ในเรื่องของ*ชนิดของข้อมูลและตัวแปร* เราได้รู้จัก ‘unsigned’ นั่นคือให้ตัวแปรเก็บเลขจำนวนเต็มบวกเท่านั้น ในกรณีนี้ MSB จะไม่ใช้เก็บเครื่องหมาย และนำมาใช้เก็บตัวเลขมากที่สุดได้เพิ่มขึ้นสองเท่า หรือพื้นที่ n บิทสามารถเก็บเลข

จำนวนเต็มมากที่สุดได้เท่ากับ 2^n-1 หากมี 32 บิทก็จะเก็บเลขค่ามากที่สุดได้เท่ากับ 4,294,967,296

12.1 การกระทำระดับบิท

- ภาษา C มี operator สำหรับการกระทำในระดับบิทดังนี้

&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Ones complement
<<	Left shift
>>	Right shift

- โดยการกระทำระดับบิทสามารถใช้กับจำนวนเต็มชนิดใดๆก็ได้ เช่น short, long, long long, signed, unsigned และตัวอักษร แต่ไม่สามารถทำกับเลขทศนิยม

Bitwise AND

- เมื่อจำนวนเต็มสองจำนวนมา AND กัน เลขฐานสองของจำนวนเต็มคู่นั้นจะถูกเปรียบเทียบบิทต่อบิท ผลลัพธ์ในแต่ละบิทจะเป็นไปตาม truth table นี้

b1	b2	b1 & b2
0	0	0
0	1	0
1	0	0
1	1	1

- เช่นถ้า w1 และ w2 เป็นเลขจำนวนเต็มชนิด short int และ w1 เท่ากับ 25 และ w2 เท่ากับ 77 ดังนั้น w3 = w1 & w2; จะทำให้ w3 มีค่าเท่ากับ 9 ซึ่งเป็นไปตามการคำนวณดังนี้

w1	0000000000011001	25
w2	0000000001001101	77
w3	0000000000001001	9

- ระวังอย่าสับสน bitwise AND operator (&) กับ logical AND operator (&&)
- logical AND operator จะใช้กับเรื่องตรรกะเพื่อให้ผลลัพธ์เป็น true/false ส่วน bitwise AND operator จะกระทำการ and ข้อมูลในระดับบิท
- Bitwise AND จะใช้มากสำหรับการ mask ข้อมูล คือทำให้ข้อมูลเป็น 0 เฉพาะบางบิท เช่น w3 = w1 & 3; เป็นการคัดลอกข้อมูลเฉพาะสองบิทแรกใน w1 ไปไว้ใน w3 และให้บิทที่เหลือใน w3 เท่ากับ 0
- เราสามารถเขียน bitwise AND ในรูป &= ได้ เช่น

  word = word & 15; เขียนได้เป็น

  word &= 15;

  และเป็นการทำให้บิททั้งหมดเท่ากับ 0 ยกเว้นสี่บิทด้านขวา
- เรานิยมใช้เลขฐานแปดหรือสิบหกในการเขียนโปรแกรมสำหรับการกระทำในระดับบิท เนื่องจากเลขฐานแปดหนึ่งหลักเขียนได้เป็นเลขฐานสองสามหลัก และเลขฐานสิบหกหนึ่งหลักเขียนได้เป็นเลขฐานสองสี่หลัก

Program 12.1 The Bitwise AND Operator

```
// Program to demonstrate the bitwise AND operator
#include <stdio.h>
int main (void)
{
    unsigned int word1 = 077u, word2 = 0150u, word3 = 0210u;

    printf("%o ", word1 & word2);
    printf("%o ", word1 & word1);
    printf("%o ", word1 & word2 & word3);
    printf("%o\n", word1 & 1);

    return 0;
}
```

```
50 77 10 1
```

- ในกรณีที่ตัวเลขจำนวนเต็มนำหน้าด้วย 0 จะถือว่าเป็นเลขฐานแปด
- ดังนั้นเลขจำนวนเต็มใน word1, word2, word3 มีค่า 077, 0150, และ 0210 ในฐานแปด
- หากมี u หรือ U ตามหลังเลขจำนวนเต็ม จะถือว่าเป็นชนิด unsigned
- การคำนวณ word1 & word2 เป็นดังนี้

word1	…	000 111 111	077
word2	…	001 101 000	0150
	…	000 101 000	050

- การคำนวณ word1 & word1 จะได้ผลลัพธ์เท่ากับ word1
- การคำนวณ word1 & word2 & word3 ผลลัพธ์ที่ได้จะไม่แตกต่างกันไม่ว่าจะทำคู่ไหนก่อน แต่ในภาษา C การคำนวณจะเป็นลำดับจากซ้ายไปขวาเสมอ

- word1 & 1 เป็นการหาค่า LSB ของ word1 ซึ่งอาจนำไปใช้เพื่อทดสอบว่าตัวเลขจำนวนเต็มนั้นเป็นเลขคู่หรือเลขคี่ได้ เนื่องจากถ้าเป็นเลขคี่ LSB จะเป็น 1 และถ้าเป็นเลขคู่ LSB จะเป็น 0 โดยอาจเขียนเงื่อนไขได้เป็น

  ```
  if (word1 & 1){…}
  else{…}
  ```
  เมื่อ word1 เป็นเลขคี่ ผลลัพธ์จะเป็น true

Bitwise OR

- เมื่อจำนวนเต็มสองจำนวนมา OR กัน เลขฐานสองของจำนวนเต็มคู่นั้นจะถูกเปรียบเทียบบิทต่อบิท ผลลัพธ์ในแต่ละบิทจะเป็นไปตาม truth table นี้

b1	b2	b1 | b2
0	0	0
0	1	1
1	0	1
1	1	1

- เช่นถ้า w1 เป็น unsigned int มีค่า 0431 และ w2 มีค่า 0152 การทำ w1 | w2 จะได้ผลลัพธ์เท่ากับ 0573 ดังแสดง

```
w1    …  100 011 001    0431
w2    …  001 101 010    0152
---------------------------
      …  101 111 011    0573
```

- ระวังอย่าสับสนระหว่าง bitwise OR operator | กับ logical OR operator ||

- ตัวอย่างการใช้งาน bitwise OR คือการเซ็ทให้เฉพาะบางบิทเป็น 1 เช่น w1 = w1 | 07; เป็นการทำให้สามบิทด้านขวาของ w1 เท่ากับ 1 โดยไม่สนใจค่าเดิม
- เราสามารถเขียน bitwise OR ในรูปย่อได้ เช่น w1 = w1 | 07; เขียนได้เป็น w1 |= 07;

Bitwise XOR

- Truth table ของ XOR (^) หรือ exclusive OR เป็นดังนี้

b1	b2	b1 ^ b2
0	0	0
0	1	1
1	0	1
1	1	0

- เช่นถ้า w1 เป็น unsigned int มีค่า 0536 และ w2 มีค่า 0266 การทำ w1 ^ w2 จะได้ผลลัพธ์เท่ากับ 0750 ดังแสดง

```
w1      …   101 011 110      0536
w2      …   010 110 110      0266
----------------------------------
        …   111 101 000      0750
```

- จำนวนเต็มใดๆที่ทำ XOR กับตัวมันเองจะได้ผลลัพธ์เป็น 0 ซึ่งวิธีนี้ถูกใช้ในภาษา assembly เพื่อเซ็ทค่าให้เป็น 0 หรือใช้เปรียบเทียบจำนวนสองจำนวนว่าเท่ากันหรือไม่

- การใช้งานอีกอย่างของ XOR คือใช้สลับที่ของจำนวนเต็มสองจำนวนโดยไม่ต้องใช้หน่วยความจำเพิ่ม โดยปกติเราสลับค่าใน i1 และ i2 ตามลำดับนี้

  ```
  temp = i1;
  i1 = i2;
  i2 = temp;
  ```

  ซึ่งถ้าใช้ XOR ช่วย จะเขียนได้ดังนี้

  ```
  i1 ^= i2;
  i2 ^= i1;
  i1 ^= 12;
  ```

<u>Complement operator</u>

- Complement operator (~) เป็นการกลับบิท 0 เป็น 1 และ 1 เป็น 0
- เช่นหาก w1 เป็นตัวแปรชนิด short int ที่มี 16 บิท และมีค่าเท่ากับ 0122457 ทำ w2 = ~w1; จะได้

w1	001 010 010 100 101 111	0122457
w2	000 101 101 011 010 000	0055320

- ระวังสับสนกับ logical negation operator (!) ซึ่งใช้กลับ true เป็น false และ false เป็น true
- ตัวอย่างถัดไปเป็นการใช้งาน bitwise operator ต่างๆที่ได้กล่าวมาแล้ว
- หากใช้งาน operator เหล่านี้ร่วมกัน ลำดับความสำคัญจะเรียงจากมากไปน้อยดังนี้ Complement, AND, XOR, OR

Program 12.2 Illustrate Bitwise Operators

```
/* Program to illustrate bitwise operators */
#include <stdio.h>
int main (void)
{
    unsigned int w1 = 0525u, w2 = 0707u, w3 = 0122u;

    printf("%o %o %o\n", w1 & w2, w1 | w2, w1 ^ w2);
    printf("%o %o %o\n", ~w1, ~w2, ~w3);
    printf("%o %o %o\n", w1 ^ w1, w1 & ~w2, w1 | w2 | w3);
    printf("%o %o\n", w1 | w2 & w3, w1 | w2 & ~w3);
    printf("%o %o\n", ~(~w1 & ~w2), ~(~w1 | ~w2));

    w1 ^= w2;
    w2 ^= w1;
    w1 ^= w2;
    printf ("w1 = %o, w2 = %o\n", w1, w2);
    return 0;
}
```

```
505 727 222
37777777252 37777777070 37777777655
0 20 727
527 725
727 505
w1 = 707, w2 = 525
```

Left shift operator

- Left shift operator หรือ << ใช้เลื่อนบิททั้งหมดไปทางซ้ายเท่ากับจำนวนบิทที่ระบุเอาไว้ในคำสั่ง บิทด้านซ้ายสุดที่ถูกเลื่อนออกจะหายไป และบิท 0 จะถูกเลื่อนเข้ามาแทนที่ทางด้านขวามือ
- เช่นในกรณีที่ w1 มีค่าเท่ากับ 3 การทำ w1 = w1 << 1; หรือ w1 <<= 1; เป็นการเลื่อนค่าใน w1 ไปด้านซ้าย 1 บิท มีผลทำให้ w1 มีค่าเท่ากับ 6

w1 000 011 03
w1 << 1 000 110 06

- ในการนี้ ค่าที่อยู่ทางด้านซ้ายของเครื่องหมาย << คือค่าที่ต้องการเลื่อนและค่าที่อยู่ทางด้านขวาคือจำนวนบิทที่ต้องการเลื่อน
- หาก shift w1 ไปทางซ้ายอีกหนึ่งบิท จะได้ผลลัพธ์เป็น 014

w1 000 110 06
w1 << 1 001 100 014

- จริงๆแล้วการเลื่อนค่าใดๆไปทางซ้ายทีละบิท มีผลเท่ากับการคูณค่านั้นด้วยสอง และในบางระบบเมื่อต้องคูณด้วยสอง จะใช้การเลื่อนบิทแทนเนื่องจากการเลื่อนทำได้ไวกว่าการคูณมาก

Right shift operator

- Right shift operator หรือ >> ใช้เลื่อนค่าไปทางขวา ทำให้บิทด้านขวาสุดหายไป และบิท 0 เลื่อนเข้ามาแทนที่ในด้านซ้าย
- สำหรับกรณีที่ค่านั้นมีบิทเครื่องหมายกำกับในด้านซ้ายสุด ถ้าบิทเครื่องหมายเป็น 0 (หรือค่านั้นเป็นบวก) บิท 0 จะถูกเลื่อนเข้ามาเติม แต่หากบิทเครื่องหมายเป็น 1 (ค่าติดลบ) ค่าที่เลื่อนเข้ามาแทนที่จะเป็น 0 หรือ 1 ขึ้นอยู่กับระบบที่ใช้งาน
- ให้ w1 เป็น unsigned int ที่มี 32 บิทและเก็บค่าเริ่มต้น F777EE22 ดังนั้นการ shift w1 ไปทางขวาหนึ่งบิท หรือ w1 >>= 1; จะได้ 7BBBF711

w1 1111 0111 0111 0111 1110 1110 0010 0010 F777EE22
w1 >> 1 0111 1011 1011 1011 1111 0111 0001 0001 7BBBF711

- เมื่อมีการเลื่อนค่าไปทางซ้ายหรือขวามากกว่าหรือเท่ากับจำนวนบิทที่เก็บข้อมูล ภาษา C จะให้ผลลัพธ์ที่ไม่แน่นอน เช่นหากมีเลขจำนวนเต็มขนาด 32 บิทแล้วสั่งเลื่อนไปทางซ้ายหรือขวามากกว่า 32 บิท จะไม่สามารถบอกผลลัพธ์ได้
- เช่นเดียวกัน การเลื่อนค่าใดๆด้วยจำนวนลบ เช่น w1 <<= -1; จะไม่นิยาม

<u>ตัวอย่างฟังก์ชันที่ใช้ shift operator</u>

- โปรแกรมต่อไปนี้สร้างฟังก์ชันขึ้นมาใหม่เพื่อเลื่อนค่าไปทางซ้ายหรือขวา ขึ้นอยู่กับเครื่องหมายของจำนวนบิทที่ต้องการเลื่อนโดยจะเลื่อนไปทางซ้ายถ้าเป็นบวกและเลื่อนไปทางขวาถ้าเป็นลบ

Program 12.3 Implementing a Shift Function

```
// Function to shift an unsigned int left if
// the count is positive, and right if negative

#include <stdio.h>
unsigned int shift(unsigned int value, int n)
{
    if( n > 0 )
        value <<= n;
    else
        value >>= -n;
    return value;
}

int main (void)
{
    unsigned int w1 = 0177777u, w2 = 0444u;
    printf("%o\t%o\n", shift(w1, 5), w1 << 5);
    printf("%o\t%o\n", shift(w1, -6), w1 >> 6);
    printf("%o\t%o\n", shift(w2, 0), w2 >> 0);
    printf("%o\n", shift(shift(w1, -3), 3));
    return 0;
}
```

- ฟังก์ชัน shift คืนค่าออกเป็น unsigned int และรับ argument สองตัวคือค่าที่ต้องการเลื่อนเป็น unsigned int และจำนวนบิทที่ต้องการเลื่อนเป็น int
- ถ้าจำนวนบิท n มากกว่าศูนย์ ค่าจะถูกเลื่อนไปทางซ้าย n บิท แต่ถ้าจำนวนบิทน้อยกว่าศูนย์ ค่าจะถูกเลื่อนไปทางขวา -n บิท
- ฟังก์ชันถูกเรียกใช้งานหลายครั้งในโปรแกรม โดยแต่ละครั้งมีลักษณะการเลื่อนที่แตกต่างออกไป

<u>การหมุน Bit</u>

- ตัวอย่างโปรแกรมถัดไปเป็นการเขียนฟังก์ชันเพื่อหมุนค่าบิทซึ่งคล้ายกับการเลื่อน แต่การหมุนบิท เมื่อบิทถูกหมุนออกไปทางด้านซ้ายแล้วจะถูกใส่กลับเข้ามาทางด้านขวา เช่นเดียวกันเมื่อบิทถูกหมุนออกไปทางด้านขวาก็จะถูกใส่กลับเข้ามาทางด้านซ้าย
- เช่นถ้าหากมีเลขฐานสิบหกเท่ากับ 80000000 หมุนไปทางซ้ายหนึ่งบิทจะได้ค่า 00000001
- ดังนั้นฟังก์ชันที่จะเขียน รับ argument สองค่าคือค่าที่ต้องการหมุน และจำนวนบิทที่ต้องการหมุน หากเป็นบวกจะหมุนไปทางซ้ายและหากเป็นลบจะหมุนไปทางขวา
- กำหนดให้ค่าที่ต้องการหมุนเป็นชนิด int และมีขนาด 32 บิท

Program 12.4 Implementing a Rotate Function

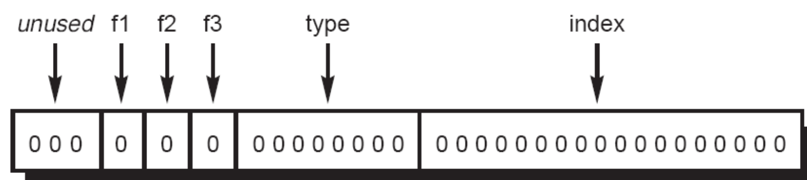
```
// Program to illustrate rotation of integers
#include <stdio.h>
int main(void)
{
    unsigned int w1 = 0xabcdef00u, w2 = 0xffff1122u;
    unsigned int rotate(unsigned int value, int n);

    printf("%x\n", rotate(w1, 8));
    printf("%x\n", rotate(w1, -16));
    printf("%x\n", rotate(w2, 4));
    printf("%x\n", rotate(w2, -2));
    printf("%x\n", rotate(w1, 0));
    printf("%x\n", rotate(w1, 44));
    return 0;
}

// Function to rotate an unsigned int left or right
unsigned int rotate(unsigned int value, int n)
{
    unsigned int result, bits;
    // scale down the shift count to a defined range
    if( n > 0 )
        n = n % 32;
    else
        n = -(-n % 32);

    if( n == 0 )
        result = value;
    else if( n > 0 )
    {
        bits = value >> (32 - n);
        result = value << n | bits;
    }
    else
    {
        n = -n;
        bits = value << (32 - n);
        result = value >> n | bits;
    }
    return result;
}
```

```
cdef00ab
ef00abcd
fff1122f
bfffc448
abcdef00
def00abc
```

- if-else คู่แรกในฟังก์ชัน rotate จะปรับจำนวนบิทที่ต้องการหมนุหรือ n ให้อยู่ในช่วง -31 ถึง 31 ก่อน โดยใช้ modulus หรือหารเอาเศษเข้ามาช่วย โดยถ้าค่า n เป็นบวกให้หารเอาเศษกับ 32 ได้เลย และหาก n เป็นลบ ให้ทำให้เป็นบวกก่อนแล้วหารเอาเศษกับ 32 จากนั้นใส่เครื่องหมายลบคืนให้ n
- if-else คู่หลังเป็นการหมุนบิท แบ่งเป็นสามกรณีคือกรณีที่ n เท่ากับศูนย์ จะไม่มีการเปลี่ยนแปลง กรณีที่ n มากกว่าศูนย์จะหมุนบิทไปทางซ้าย และกรณีที่ n น้อยกว่าศูนย์ จะหมุนบิทไปทางขวา
- การหมุนบิทไปทางซ้าย มีขั้นตอนดังนี้ เลื่อนค่าใน value ไปทางขวาเป็นจำนวน 32-n บิทแล้วนำไปเก็บไว้ในตัวแปรชื่อ bits จากนั้นเลื่อนค่าใน value ไปทางซ้าย n บิท แล้วนำค่าที่ได้ไป OR กับค่าในตัวแปร bits จะได้ผลลัพธ์เท่ากับการหมุนไปทางซ้าย n บิท
- ในฟังก์ชัน main เขียนเลขจำนวนเต็มในรูปของเลขฐานสิบหก และมีการเรียกใช้ฟังก์ชัน rotate เพื่อหมุนบิทของ w1 และ w2 ด้วยค่าต่างๆกัน

12.2 Bit Fields

- โดยปกติตัวแปรชนิด _Bool ซึ่งใช้พื้นที่จริงเพียง 1 บิทเพื่อเก็บ true หรือ false จะจองพื้นที่จริงในหน่วยความจำ 8 บิท หรือ 1 ไบท์ หากเรามีตัวแปร _Bool มาก ก็จะเสียพื้นที่ในหน่วยความจำไปโดยเปล่าประโยชน์
- เช่นเดียวกันหากเรามีข้อมูลอื่นๆ เช่นค่าตัวเลขจาก 0-10 ซึ่งใช้พื้นที่เก็บเพียง 4 บิท การประกาศตัวแปรเป็น int หรือ short int ก็จะจองหน่วยความจำเกินความจำเป็น
- เพื่อประหยัดเนื้อที่ในหน่วยความจำเราสามารถจัดเก็บข้อมูลหลายๆตัวลงในระดับบิทได้โดยไม่เสียพื้นที่หน่วยความจำ เรียกว่า bit field
- อย่างเช่น เราต้องการเก็บข้อมูลห้าตัวลงในตัวแปรตัวเดียว และข้อมูลสามตัวแรกเป็น boolean ชื่อ f1, f2, และ f3 และข้อมูลตัวที่สี่เป็นเลขจำนวนเต็มชื่อ type ซึ่งเก็บค่าจาก 1 ถึง 255 และข้อมูลตัวสุดท้ายคือ index ซึ่งมีค่าจาก 0 ถึง 100,000
- f1, f2, f3 ต้องการเพียงสามบิทเพื่อเก็บข้อมูล และ type ต้องการ 8 บิท ส่วน index ต้องการ 18 บิท ดังนั้นเราต้องการตัวแปรที่เก็บได้อย่างน้อย 29 บิท



- เราสามารถสร้าง structure ที่ประกอบด้วย bit fields ดังนี้

```
struct packed_struct
{
    unsigned int :3;
    unsigned int f1:1;
    unsigned int f2:1;
    unsigned int f3:1;
    unsigned int type:8;
    unsigned int index:18;
};

struct packed_struct packed_data;
```

- structure นี้ประกอบด้วยสมาชิกเป็น bit field หกตัว โดยสมาชิกตัวแรกไม่มีชื่อ ใช้พื้นที่ 3 บิท และสมาชิกตัวถัดมา f1, f2, f3 แต่ละตัวใช้พื้นที่ 1 บิท และ type ใช้พื้นที่ 8 บิท ส่วน index ใช้ 18 บิท สมาชิกทั้งหมดสำหรับเก็บเลขจำนวนเต็มค่าบวก
- เราสามารถเข้าถึงตัวแปรย่อยได้เหมือนตัวแปร structure ทั่วไป เช่น

  packed_data.type = 7;

  เป็นการใส่ค่า 7 ลงในตัวแปรย่อย type ของ packed_data

  หรือ

  n = packed_data.type;
  i = packed_data.index / 5 + 1;
  if(packed_data.f2){…}
- เราสามารถรวมข้อมูลปกติลงไปใน structure ที่มี bit fields ได้ เช่น

```
struct table_entry
{
    int count;
    char c;
    unsigned int f1:1;
    unsigned int f2:1;
};
```

- สำหรับ bit fields จะต้องเป็นชนิดเลขจำนวนเต็ม (int, unsigned int, short int, …) หรือ _Bool เท่านั้น
- bit field ไม่สามารถประกาศเป็น array อย่างเช่น flag:1[5] ได้
- ไม่สามารถใช้ pointer กับ bit field ได้