5

พรีโปรเซสเซอร์

ในบทนี้เราจะได้ทำความรู้จักกับสิ่งที่เรียกว่า *พรีโปรเซสเซอร์* ซึ่งเป็นเครื่องมือช่วยอย่างหนึ่งที่ทำให้เราเขียนโปรแกรมคอมพิวเตอร์ในภาษาซีได้ง่ายและมีประสิทธิภาพมากขึ้น

ดังที่ได้กล่าวไปแล้วในบทที่หนึ่ง ก่อนที่คอมไพล์เลอร์จะเริ่มแปลงโปรแกรมโค้ดใดๆ ให้เป็นชุดคำสั่งในภาษาเครื่องหรือที่เราเรียกว่าออปเจคโค้ดนั้น คอมไพล์เลอร์จะเรียกโปรแกรมชนิดหนึ่งขึ้นมาทำงานก่อนโดยอัตโนมัติ เราเรียกโปรแกรมชนิดนี้ว่า *พรีโปรเซสเซอร์* (Preprocessor) หรือ *ตัวดำเนินการก่อน* โดยผ่านโปรแกรมโค้ด(สำเนาของไฟล์) ไปยังพรีโปรเซสเซอร์นี้ เพื่อจัดการแก้ไขหรือดัดแปลงก่อนที่จะเริ่มทำการคอมไพล์โปรแกรมโค้ดจริงในขั้นตอนต่อไป

พรีโปรเซสเซอร์จะอาศัยสิ่งที่เราเรียกว่า *พรีโปรเซสเซอร์ ไดเรคทีฟ* ในการจำแนกว่าในส่วนใดของโปรแกรมโค้ดเป็นประโยคคำสั่งสำหรับพรีโปรเซสเซอร์ซึ่งประโยคเหล่านี้มีหลักไวยกรณ์แบบเน้นบรรทัด (Line-oriented Syntax)

5.1 การใช้พรีโปรเซสเซอร์ไดเรคทีฟ

พรีโปรเซสเซอร์มีหน้าที่ลบคำอธิบายทั้งหมดที่มีอยู่ในไฟล์สำเนาของโปรแกรมโค้ด (เพราะคำอธิบายเหล่านี้ไม่มีผลต่อขั้นตอนการทำงานของโปรแกรม เพียงแต่ช่วยให้ผู้ที่เขียนหรืออ่านโปรแกรมโค้ดเข้าใจคำสั่งหรือการทำงานของโปรแกรมได้ง่ายขึ้นเท่านั้น) และนอกจากนี้ยังมีหน้าที่อื่นๆที่กำหนดไว้แล้วอย่างชัดเจนโดยอาศัยบรรทัดของคำสั่งที่เรียกว่า *พรีโปรเซสเซอร์ ไดเรคทีฟ* (Preprocessor Directive) เป็นตัวกำหนด ซึ่งก็คือตัวระบุชื่อที่เริ่มต้นด้วยสัญลักษณ์ # เช่น

#include #define #ifdef เหล่านี้เป็นต้น ดังนั้นเมื่อพรีโปรเซสเซอร์ตรวจพบว่า ในบรรทัดใดมีไดเรคทีฟดังกล่าวอยู่ในตำแหน่งเริ่มต้นของแต่ละบรรทัด ซึ่งหมายถึงบรรทัดที่มีสัญลักษณ์ # เป็นตัวแรกของข้อความในบรรทัดนั้น แต่อาจจะมีที่ว่างอยู่นำหน้าได้ ก็จะถือว่าข้อความทั้งบรรทัดนั้นทั้งหมดเป็นบรรทัดคำสั่งของไดเรคทีฟ และพรีโปรเซสเซอร์ก็จะตีความหมายของคำสั่งดังกล่าวและดำเนินการตามความหมายของคำสั่งนี้ที่ได้กำหนดไว้ จากนั้นก็ตรวจดูบรรทัดต่อไปในโปรแกรมโค้ดว่ายังมีบรรทัดใดที่เริ่มต้นด้วยไดเรคทีฟอยู่อีกหรือไม่ ถ้ามีก็ดำเนินการก่อนไปเรื่อยๆจนถึงบรรทัดสุดท้ายของโปรแกรมโค้ด เมื่อพรีโปรเซสเซอร์ทำงานเสร็จก็ผ่านผลของโปรแกรมโค้ดที่ถูกดัดแปลงแก้ไขแล้วซึ่งเป็นโปรแกรมโค้ดที่เสร็จสมบรูณ์ ไปยังคอมไพล์เลอร์อีกครั้งเพื่อจะได้ดำเนินการคอมไพล์ ดังนั้นพรีโปรเซสเซอร์ไดเรคทีฟจะมีผลเฉพาะตอนที่เราทำการคอมไพล์โปรแกรมโค้ดเท่านั้น

โปรดสังเกตว่า ถ้าเราจะใช้พรีโปรเซสเซอร์ ไดเรคทีฟ ในบรรทัดใด แล้วตลอดทั้งบรรทัดนั้นจะต้องมีเพียงคำสั่งไดเรคทีฟเดียวเท่านั้น ตัวอย่างที่ผิด เช่น

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

พรีโปรเซสเซอร์ ไดเรคทีฟที่เราสามารถเลือกใช้ได้มีดังต่อไปนี้

```
#if         #ifdef          #ifndef
#else       #elif           #endif
#include    #define         #undef
#line       #error          #pragma
```

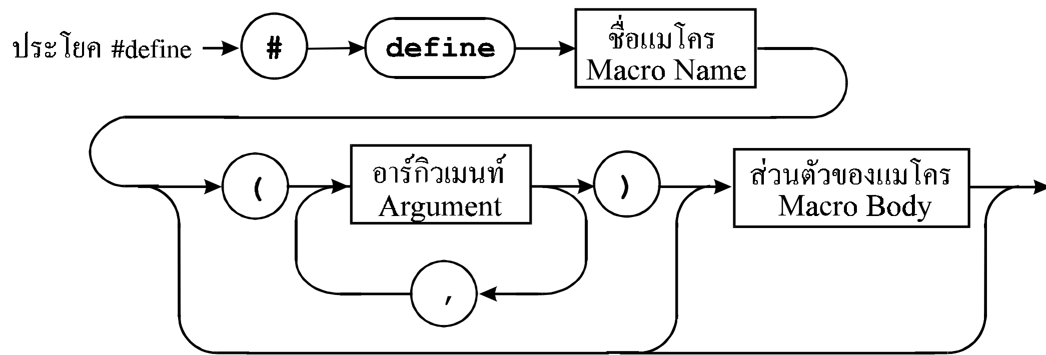
5.2 คำสั่ง #define

ประโยคคำสั่งไดเรคทีฟแรกที่เราจะทำความเข้าใจก็คือ #define ซึ่งเป็นการนิยามสัญลักษณ์ของลำดับของตัวอักษรหรือโทเคน (Token) ใดๆ เรามักจะเรียกว่า *แมโคร* (Macro) หรือ *ค่าคงที่เชิงสัญลักษณ์* (Symbolic Constant) และมีรูปแบบการใช้งานอย่างง่ายดังต่อไปนี้

```
#define   identifier  replacement_token
```

ตัวอย่างการใช้งาน เช่น

```
/* 1 */    #define  PI          3.141592654
/* 2 */    #define  TRUE        1
/* 3 */    #define  FALSE       0
```



รูปภาพที่ 5.1 โครงสร้างประโยคของ #define ที่มีพารามิเตอร์

คำสั่งไดเรคทีฟในบรรทัดที่หนึ่งส่งผลให้โทเคนใดๆในโปรแกรมโค้ด (ยกเว้นโทเคนที่เป็นส่วนหนึ่งของข้อความแบบ String) ที่ตรงกับคำว่า PI จะถูกแทนที่ด้วยตัวเลขค่าคงที่ 3.141592654 ในบรรทัดที่สองและสามเป็นการกำหนดค่าคงที่เชิงสัญลักษณ์ TRUE และ FALSE ที่มีค่าเท่ากับ 1 และ 0 ตามลำดับ ตัวอย่างการใช้ #define เช่น

```
/* 1 */     #include <stdio.h>
/* 2 */     #include <math.h>
/* 3 */     #define   PI  3.141592654
/* 4 */
/* 5 */     int main()
/* 6 */     {
/* 7 */        double radius = 10.0;
/* 8 */        double PI_2   = PI / 2;
/* 9 */        double area   = PI * radius * radius;
/*10 */
/*11 */        printf ("sin(90°) = sin(PI/2) = %lf\n",
/*12 */                 sin(PI_2));
/*13 */        printf ("Area = %lf\n", area);
/*14 */        return 0;
/*15 */     }
```

เราจะเห็นได้ว่าในบรรทัดที่ 3 เรานิยามคำว่า PI โดยใช้แทนตัวเลข(จำนวนจริง) 3.141592654 ดังนั้นในบรรทัดที่ 8 และ 9 ก็จะหมายถึง

```
double PI_2   = 3.141592654 /2;
double area   = 3.141592654 * radius * radius;
```

ซึ่ง PI จะถูกแทนที่ด้วย 3.141592654 โดยพรีโปรเซสเซอร์เมื่อเราทำการคอมไพล์โปรแกรมโค้ดนี้

โปรดสังเกตว่า PI_2 นั้นแตกต่างจาก PI ซึ่ง PI_2 เป็นตัวแปรแบบ double ในขณะที่ PI เป็น ค่าคงที่เชิงสัญลักษณ์หรือแมโคร ดังนั้นจึงไม่มีการแทนที่ PI ที่เป็นส่วนหนึ่งของชื่อตัวแปร PI_2 นี้ นอกจากนั้นเราจะเห็นได้ว่าในบรรทัดที่ 11 มีคำว่า PI อยู่ แต่เนื่องจากว่าคำนี้เป็นส่วนหนึ่งของข้อความหรือสายตัวอักขระ (String) ในภาษาซี ซึ่งมีสัญลักษณ์ " อยู่หัวและท้ายข้อความ

```
"sin(90°) = sin(PI/2) = %lf\n"
```

ดังนั้น PI ในข้อความนี้จะไม่ถูกแทนที่ด้วยตัวเลขที่เราได้นิยามไว้โดยแมโคร PI

ถ้าเราใช้สัญลักษณ์ PI แทนที่จะเขียนตัวเลขลงไปโดยตรงในจุดต่างๆของโปรแกรมโค้ด เมื่อเราต้องการจะเปลี่ยนแปลงค่าของ PI ใหม่ เช่น จาก 3.141592654 เป็น 3.14159 ดังนั้น ถ้าเราไม่ใช้วิธีนิยามสัญลักษณ์ PI ขึ้นมาใช้ในโปรแกรมโค้ด แต่เขียนตัวเลขดังกล่าวแทนตามตัวอย่างข้างล่าง

```
double radius = 10.0, PI_2 = 3.141592654 /2;
double area = 3.141592654 * radius * radius;

printf ("PI = %lf\n", 3.141592654);
```

เมื่อเราจะเปลี่ยนค่าของตัวเลข 3.141592654 เป็น 3.14159 เราก็ต้องแก้ไขตัวเลขนี้ในหลายๆตำแหน่งของข้อความด้วยตนเอง แต่ถ้าเรานิยามสัญลักษณ์ PI ไว้ตอนต้นและใช้แทนตัวเลขดังกล่าว เวลาเราต้องการเปลี่ยนแปลงค่าของตัวเลขนี้ เราก็แก้ไขเฉพาะที่บรรทัดคำสั่งของไดเรคทีฟเท่านั้น ซึ่งเขียนใหม่ได้ดังนี้

```
#define PI  3.14159
```

ทำให้ง่ายต่อการเปลี่ยนแก้ไขในโปรแกรมโค้ด

นอกจากนี้ เราสามารถนิยามแมโครใดๆที่มีแต่ตัวระบุชื่อ แต่ไม่มีส่วนตัวของแมโคร ซึ่งถือว่าเป็นแมโครว่างเปล่า ดังนั้นแมโครลักษณะนี้เราจะไม่ใช้ในการแทนที่โทเคนหรือข้อความใดๆ แต่จะมีผลเฉพาะสำหรับไดเรคทีฟคำสั่ง #ifdef หรือ #ifndef ซึ่งใช้ในการสร้างเงื่อนไขในการคอมไพล์ส่วนของโปรแกรมโค้ด เช่น

```
#define DEBUG

#ifdef DEBUG
      printf("Debugging code begins...\n")
#endif
```

ในกรณีตัวอย่างนี้ เราได้นิยามแมโครว่างเปล่าชื่อ DEBUG ดังนั้นจึงให้ค่าของเงื่อนไขเป็นจริงและคอมไพล์เลอร์ก็จะคอมไพล์ส่วนของโปรแกรมโค้ดที่อยู่ระหว่างไดเรคทีฟ #ifdef และ #endif ถ้าไม่มีการนิยามแมโคร DEBUG มาก่อนหน้านี้ ก็จะไม่มีการคอมไพล์คำสั่ง printf() ตามตัวอย่าง

ตัวอย่างเพิ่มเติมสำหรับการใช้ #define

```
#define Yes          1
#define No           0
#define MIN(x, y)   ((x) > (y) ? (y) : (x))
#define MAX(x, y)   ((x) > (y) ? (x) : (y))
#define ABS(x)      ((x) >= 0 ? (x) : -(x))
#define USAGE       "talk <user_name> [<tty>]"
```

เราสามารถใช้แมโครที่นิยามไว้ได้ดังตัวอย่างต่อไปนี้

```
printf ("The absolute value of %d : %d\n", -10,
         ABS(-10) );

if ( MIN(x,0)==x )
     x = -x;

printf ("Usage : %s\n", USAGE);
```

ตัวอย่างเหล่านี้จะถูกแปลงโดยพรีโปรเซสเซอร์ให้เป็น

```
printf ("The absolute value of %d : %d\n",-10,
         ((-10) >= 0 ? (-10) : -(-10)) );

if ( ((x) > (0) ? (0) : (x))==x )
     x = -x;

printf ("Usage : %s\n", "talk <user_name> [<tty>]");
```

โปรดจำไว้ว่า เราสามารถใช้พรีโปรเซสเซอร์ ไดเรคทีฟ ในตำแหน่งใดๆของโปรแกรมโค้ดได้ โดยเฉพาะ ถ้าเราใช้ #define ในการนิยามแมโครขึ้นมาใช้ แมโครนี้ก็จะเริ่มมีผลใช้ตั้งแต่บรรทัดที่เรานิยามแมโครนี้เป็นต้นไปจนสิ้นสุดโปรแกรมโค้ดหรือจนกว่าเราจะได้ใช้ไดเรคทีฟอีกตัวหนึ่งคือ #undef ในการยกเลิกการนิยามแมโครดังกล่าว

จากตัวอย่างข้างบน เราเห็นได้ว่าเราสามารถนิยามแมโครที่มีลักษณะคล้ายฟังก์ชัน คือ มีอาร์กิวเมนท์ที่อยู่ระหว่างเครื่องหมายวงเล็บเปิดปิด และสามารถมีอาร์กิวเมนท์ได้มากกว่าหนึ่งตัว

ทำไมเราจึงใช้แมโครที่มีลักษณะคล้ายฟังก์ชัน เช่น MIN(x,y) MAX(x,y) หรือ ABS(x) แทนที่จะเขียนให้อยู่ในรูปของฟังก์ชัน ? เหตุผลก็คือว่า ถ้าเราไม่ใช้แมโครลักษณะนี้แต่เราเขียนฟังก์ชันขึ้นใช้แทน เช่น

```
int MIN (int x, int y)
{
     return (x > y) ? y : x;
}

int MAX (int x, int y)
{
     return (x > y) ? x : y;
}

int ABS (int x)
{
     return (x >= 0) ? x : -x;
}
```

ซึ่งเรานำไปใช้งานในลักษณะเดียวกันกับที่เราใช้แมโครได้ แต่มีข้อแตกต่างคือถ้าเราเขียนเป็นฟังก์ชันเราจะใช้ได้เฉพาะกับตัวแปรที่เป็น int เท่านั้น(ตามตัวอย่างฟังก์ชันทั้งสาม) เมื่อเราต้องการใช้กับตัวแปรที่เป็นเลขจำนวนจริงเช่น double หรือ float เราจะต้องเขียนฟังก์ชันเพิ่มเติมขึ้นมาที่ใช้กับตัวแปรแบบนี้ แต่ถ้าเราใช้แมโครเราสามารถใช้ได้กับแบบของข้อมูลหลายๆแบบ เพราะแมโครเป็นการแทนที่สัญลักษณ์ ด้วยโทเคนที่เราได้นิยามไว้ เช่น เราใช้ได้ทั้งข้อมูลแบบ int และ double

```
MIN(3.1, 1)
MAX(1.1, -1.55)
ABS(3-1.332)
ABS(-3.4)
```

นอกจากนี้ แมโครยังมีข้อดีเหนือการใช้ฟังก์ชันในเรื่องของความเร็วในการทำงานของโปรแกรม เพราะการใช้แมโครเป็นการแทนที่ส่วนของโปรแกรมโค้ดหรือโทเคนต่างๆโดยตรง เมื่อเปรียบเทียบกับการเรียกใช้ฟังก์ชันแล้ว เราจะเห็นได้ว่าโปรแกรมจำเป็นต้องจัดทำสำเนาของพารามิเตอร์แต่ละตัวของฟังก์ชันก่อนที่จะเรียกส่วนของฟังก์ชันขึ้นมาทำงานและเมื่อฟังก์ชันจบการทำงานก็ต้องทำลายหน่วยความจำของสำเนาเหล่านั้น ซึ่งรวมแล้วทำให้ใช้เวลามากกว่าในกรณีที่เราใช้แมโคร

ถ้าฟังก์ชันมีขนาดเล็กและสั้น เราก็มักจะเขียนให้อยู่ในรูปของแมโคร โปรดสังเกตว่า เราไม่ควร เขียนแมโครที่มีขนาดใหญ่หรือยาวเกินไป เพราะดังที่กล่าวไปการใช้แมโครเป็นการแทนที่ข้อความโดยตรง แม้ว่าโปรแกรมจะทำงานได้เร็ว แต่มีข้อเสียคือขนาดของโปรแกรมโดยรวมจะมีขนาดใหญ่ขึ้นมากกว่าในกรณีที่เราเขียนโดยใช้ฟังก์ชัน โดยเฉพาะเมื่อมีการใช้แมโครแทนที่โทเคนใดๆที่มีขนาดยาวหลายๆครั้งในโปรแกรมโค้ดเพราะตลอดทั้งโปรแกรม เราเขียนโปรแกรมโค้ดของฟังก์ชันเพียงครั้งเดียวและเขียนแต่คำสั่งเรียกใช้ฟังก์ชันเท่านั้นในการใช้แต่ละครั้ง ในขณะที่การเรียกใช้แมโครแต่ละครั้งเป็นการแทนที่ชุดคำสั่งการทำงานทั้งหมดที่นิยามไว้ในแมโคร

สรุปได้ว่า ถ้าฟังก์ชันมีขนาดเล็กและสั้น และเราต้องการใช้หลายๆครั้ง เราสามารถเขียนให้อยู่ในรูปของแมโครได้

ปัญหาสำคัญที่มักจะเกิดขึ้นได้บ่อยๆเมื่อมีการนิยามแมโครในเชิงฟังก์ชันจะเกี่ยวข้องกับความสำคัญของการใช้เครื่องหมายวงเล็บเปิดปิดในแมโครสำหรับส่วนที่เป็นพารามิเตอร์ เราสังเกตได้จากตัวอย่างต่อไปนี้ ถ้าเราเขียนแมโครสำหรับ ABS(x) ใหม่เป็น

```
#define ABS(x)       (x >= 0 ? x : -x)
```

ดังนั้นถ้าเราใช้แมโครดังตัวอย่างต่อไปนี้ในโปรแกรม เช่น

```
int x = -3;
int y = ABS(x+2);
```

ก็จะหมายถึง

```
int x = -3;
int y = (x+2 >= 0 ? x+2 : -x+2);
```

และตัวแปร y จะมีค่าเท่ากับ 5 ซึ่งเป็นค่าที่ไม่ถูกต้องตามที่เราต้องการ และที่ถูกต้องควรจะเป็น 1 เพราะค่าสัมบรูณ์ของผลรวมของ -3 และ 2 คือ 1 ดังนั้นการนิยามแมโครควรจะใส่เครื่องหมายวงเล็บให้กับพารามิเตอร์ที่เกี่ยวข้องกับจำนวนตัวเลขต่างๆ และแมโครที่ถูกต้องจะต้องเขียนให้อยู่ในรูปแบบนี้ (โปรดสังเกตเครื่องหมายวงเล็บที่ได้ใส่เพิ่มเติม)

```
#define ABS(x)       ((x) >= 0 ? (x) : -(x))
```

และอีกปัญหาหนึ่งเกี่ยวกับการใช้แมโครคือ การใช้โอเปอร์เรเตอร์ ++ และ -- ร่วมกับแมโคร จะทำให้เกิดผลข้างเคียงที่ทำให้ได้ผลลัพธ์ไม่ถูกต้อง เช่น

```
#define ABS(x)       ((x) >= 0 ? (x) : -(x))

int x,y;
  .
  .
  .
y = ABS(x++);
```

ดังนั้นประโยคที่เรียกใช้แมโครจะให้ผลลัพธ์ดังนี้

```
y = ((x++) >= 0 ? (x++) : -(x++));
```

จะเห็นได้ว่า มีการใช้โอเปอร์เรเตอร์ ++ สองครั้งไม่ว่าเงื่อนไขจะเป็นจริงหรือเท็จก็ตาม เพื่อให้เห็นความแตกต่างเราลองสมมุติว่า x มีค่าเริ่มต้นเท่ากับ -5 และเราต้องการหาค่าสัมบรูณ์ของตัวแปร x และหลังจากนั้นก็เพิ่มค่าของตัวแปรนี้ขึ้นอีกหนึ่ง ถ้าเราใช้ฟังก์ชัน

```
int ABS (int x)
{
     return (x > 0) ? x : -x;
}
```

เราจะได้ค่าของตัวแปร y

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

เป็น 5 และ x จะมีค่าใหม่เป็น -4 แต่ถ้าเราใช้แมโคร

```
#define ABS(x)      ((x) >= 0 ? (x) : -(x))
```

ตัวแปร y จะมีค่าเท่ากับ 4 และ x จะมีค่าใหม่เป็น -3

จากตัวอย่างนี้เราเห็นได้ว่า การใช้แมโครนั้นมีความแตกต่างจากการเรียกใช้ฟังก์ชันอย่างยิ่ง ดังนั้นควรจะระมัดระวังการใช้แมโคร #define ที่ใช้งานคล้ายกับฟังก์ชัน (แต่มิได้หมายถึงว่าเราควรจะหลีกเลี่ยงการใช้แมโครในลักษณะนี้)

เมื่อเรานิยามแมโครที่มีอาร์กิวเมนท์ เช่น ABS(X) เราจะต้องเขียนชื่อของแมโครติดกับวงเล็บเปิด ห้ามเว้นที่ว่างไว้ ตัวอย่างที่ผิด เช่น

```
#define ABS (x)   ((x) >= 0 ? (x) : -(x))
```

ในตัวอย่างนี้ พรีโปรเซสเซอร์จะเข้าใจว่า แมโคร ABS ไม่มีอาร์กิวเมนท์ใดๆ

ตัวอย่างต่อไปแสดงให้เห็นวิธีการเปลี่ยนฟังก์ชันสั้นๆให้เป็นแมโครโดยใช้คำสั่ง #define เช่น ฟังก์ชันของผู้ใช้ print_usage() ซึ่งนิยามไว้ดังนี้

```
void print_usage()
{
   printf ("Usage : %s\n", "talk <user_name> [<tty>]");
}
```

เราก็เขียนให้อยู่ในรูปของแมโคร

```
#define PRINT_USAGE() \
{ \
   printf("--------------------------------------\n"); \
   printf("Usage: %s\n","talk <user_name> [<tty>]\n"); \
```

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

เวลาเราเรียกใช้แมโครนี้ เราก็เขียนได้อย่างง่ายเหมือนกับเวลาเราเรียกใช้ฟังก์ชัน เช่น

```
PRINT_USAGE();
```

เนื่องจากว่าแมโคร PRINT_USAGE() ใช้แทนประโยคคำสั่งเชิงซ้อน ดังนั้นเครื่องหมาย ; ที่อยู่ต่อท้ายก็ไม่จำเป็นต้องใช้ แต่ถ้าเขียนไว้ก็ไม่ผิดอะไร (สำหรับในกรณีตัวอย่างนี้เท่านั้น) เพียงแต่เราต้องเข้าใจว่าเราเขียนประโยคคำสั่งสองประโยคต่อท้ายกัน ประโยคหนึ่งเป็นประโยคเชิงซ้อน อีกประโยคหนึ่งเป็นประโยคว่างเปล่า

```
{
   printf("--------------------------------------\n");
   printf("Usage: %s\n","talk <user_name> [<tty>]\n");
   printf("--------------------------------------\n");
};
```

โปรดสังเกตว่า เครื่องหมาย \ หรือ backslash ที่อยู่ท้ายบรรทัดของประโยคคำสั่งไดเรคทีฟ นั้นทำหน้าที่ แจ้งให้พรีโปรเซสเซอร์ทราบว่า ข้อความต่างๆในบรรทัดที่ตามมา (บรรทัดถัดไปในโปรแกรมโค้ด) นั้นเป็นข้อความที่ต่อเนื่องกันจากบรรทัดในขณะนั้นคือ อยู่ในประโยคเดียวกัน

ตัวอย่างต่อไปคือการเขียนแมโครที่ใช้สลับค่าของเลขจำนวนเต็มสองจำนวนที่เก็บไว้ในตัวแปรแบบ int

```
#define INT_SWAP(X,Y) \
{ int tmp = X; X = Y; Y = tmp; }
```

แต่แมโครนี้ใช้ได้กับตัวแปรสองตัวแบบ int เท่านั้น ถ้าเราต้องการจะเขียนแมโครให้ใช้ได้กับตัวแปรสอง ตัวสำหรับหลายๆแบบข้อมูล (ตัวแปรทั้งสองต้องมีแบบข้อมูลเหมือนกัน) เช่น int, float, double เราก็ ใช้วิธีการดังต่อไปนี้

```
#define SWAP(X,Y,TYPE) \
{ TYPE tmp = X; X = Y; Y = tmp; }
```

ตัวอย่างการเรียกใช้แมโครทั้งสองในโปรแกรมก็เช่น

```
#include <stdio.h>

#define INT_SWAP(X,Y) \
{ int tmp = X; X = Y; Y = tmp; }

#define SWAP(X,Y,TYPE) \
{ TYPE tmp = X; X = Y; Y = tmp; }
```

```
int main()
{
      int   x=5, y=-10;
      float i=5.0, j=-10.0;

      printf ("%5d %5d\n", x, y);
      INT_SWAP(x, y);
      printf ("%5d %5d\n\n", x, y);

      printf ("%5.1f %5.1f\n", i, j);
      SWAP(i, j, float);
      printf ("%5.1f %5.1f\n", i, j);

      return 0;
}
```

นอกจากนี้ยังมีแมโครที่ได้นิยามไว้แล้วโดยคอมไพล์เลอร์ ซึ่งเรียกว่า Built-In Macro คือ

แมโคร	ความหมาย
__LINE__	แทนที่หมายเลขของบรรทัดภายในโปรแกรมโค้ดในขณะที่เรียกใช้
__FILE__	แทนที่ชื่อของไฟล์ที่เก็บโปรแกรมโค้ดในขณะนั้น
__TIME__	แทนที่เวลาที่ได้ทำการคอมไพล์โปรแกรมโค้ด
__DATE__	แทนที่วันเดือนปีที่ได้ทำการคอมไพล์โปรแกรมโค้ด
__STDC__	แทนที่ค่าคงที่ 1 ถ้าคอมไพล์เลอร์สนับสนุนมาตราฐาน ANSI

ตัวอย่างการใช้ เช่น

```
#include <stdio.h>

int main()
{
      printf("This program was compiled on %s at %s\n",
             __DATE__, __TIME__);
      printf("Source code file : %s\n", __FILE__);
      return 0;
}
```

นอกจากนี้เรายังสามารถนิยามแมโครสำหรักการสร้างแถวตัวอักขระ (String) ได้โดยใช้สัญลักษณ์ # เช่น

```
#define STRING(s)     #s
```

เป็นการเปลี่ยนโทเคนซึ่งเป็นอาร์กิวเมนท์ของแมโคร STRING ให้เป็นแถวตัวอักขระในภาษาซีโดยพรีโปรเซสเซอร์ ซึ่งจะเติมเครื่องหมาย " ที่หัวเลขท้ายอาร์กิวเมนท์ของแมโคร ตัวอย่างเช่น

```
printf("%s\n", STRING (Hello World !));
```

จะให้ผลเหมือนคำสั่ง

```
printf("%s\n", "Hello World !");
```

ตัวอย่างอีกตัวอย่างหนึ่งแสดงให้เห็นความสามารถในการเปลี่ยนอาร์กิวเมนท์ของแมโครให้เป็นแถวตัวอักขระ

```
#define ASSERT(condition) \
      if (!(condition)) { \
            printf("Condition failed : %s\n", \
                  #condition); \
            exit(-1); \
      }
```

แมโครนี้มีลักษณะการทำงานคล้ายฟังก์ชัน โดยมี condition เป็นอาร์กิวเมนท์ และแมโครนี้ใช้แทนประโยคเชิงซ้อนซึ่งเป็นโครงสร้างของประโยคเงื่อนไขแบบ if และใช้ในการตรวจเช็คเงื่อนไขซึ่งกำหนดโดย condition ตัวอย่างการใช้งาน เช่น สมมุติว่า x เป็นตัวแปร ถ้าเราต้องการกำหนดให้ x มีค่าเป็นจำนวนเต็มบวกหรือศูนย์เท่านั้นซึ่งใช้เป็นเงื่อนไขสำหรับค่าของ x ถ้าค่าของตัวแปรนี้ไม่เป็นไปตามเงื่อนไขก็ให้หยุดการทำงานของโปรแกรมและพิมพ์เงื่อนไขดังกล่าวออกทางจอภาพโดยใช้คำสั่ง printf()

```
ASSERT(x >= 0);
```

ถ้า x มีค่าน้อยกว่า 0 เช่น x มีค่าเท่ากับ -1 ขั้นตอนข้างบนก็จะพิมพ์ข้อความในลักษณะนี้ออกทางจอภาพ

```
Condition failed : x >= 0
```

ตามมาตราฐาน ANSI เราสามารถนิยามแมโครที่สามารถใช้ในการประกอบโทเคนที่เป็นอาร์กิว-เมนท์ของแมโครนี้เข้าด้วยกัน โดยใช้สัญลักษณ์ ## ซึ่งเป็นโอเปอร์เรเตอร์หนึ่งของพรีโปรเซสเซอร์ ตัวอย่างการใช้งาน เช่น

```
#define PASTE(arg1,arg2)  arg1 ## arg2
```

ดังนั้นถ้าเราเขียนคำสั่งว่า

```
PASTE(Hello world   ,   !)
```

ก็จะหมายถึง

```
Hello world!
```

เป็นการต่อโทเคนทั้งสองเข้าด้วยกัน โปรดสังเกตว่าที่ว่างที่อยู่ข้างหน้าและข้างหลังอาร์กิวเมนท์แต่ละตัวจะถูกตัดทิ้งไป (คำว่า Hello และ World ถือว่าอยู่ในอาร์กิวเมนท์เดียวกัน)

แต่ถ้าเราต้องการจะต่อโทเคนทั้งสองเข้าด้วยกันและเปลี่ยนให้เป็นแถวตัวอักขระ เราก็ทำได้ดังนี้

```
#define STRING2(arg1,arg2)        #arg1 ## #arg2
#define STRING3(arg1,arg2,arg3)   #arg1 ## #arg2 ## #arg3
```

ดังนั้นเมื่อเราเรียกใช้แมโคร STRING2 (tmp, 0001) ก็จะหมายถึง

```
"tmp0001"
```

สำหรับแมโคร STRING2 หรือ STRING3 ที่ได้นิยามตามตัวอย่างนั้น เราจะต้องผ่านอาร์กิวเมนท์ที่เป็นโทเคนและจะต้องไม่ใช่แถวตัวอักขระที่มีเครื่องหมาย " อยู่หัวและท้าย ดังนั้นถ้าเราเขียนว่า

```
STRING2 ("tmp", 0001)
STRING2 (tmp, "0001")
STRING2 ("tmp", "0001")
```

แบบใดแบบหนึ่ง จึงถือว่าผิดจุดประสงค์ (แต่ไม่ผิดหลักไวยกรณ์ของภาษาซี) ผลที่จะได้คือ

```
"\"tmp\"0001"
"tmp\"0001\""
"\"tmp\"\"0001\""
```

5.3 คำสั่ง #include

การใช้ประโยคคำสั่งไดเรคทีฟ #include นั้นเราได้ทำความรู้จักไปบ้างแล้วในหลายๆ โปรแกรมตัวอย่าง

#include หมายถึง การแจ้งให้คอมไพล์เลอร์ (ถ้าจะกล่าวให้ถูกต้อง เราจะหมายถึงพรีโปรเซสเซอร์) ทราบว่า จะต้องแทรกไฟล์ใดเข้าในโปรแกรมโค้ด ดังนั้นข้อความต่างๆที่อยู่ในไฟล์ดังกล่าวก็จะถูกแทรกเข้าในโปรแกรมโค้ดในบรรทัดที่มีไดเรคทีฟ #include นี้อยู่ วิธีการนี้มีข้อดีคือ แทนที่เราจะเขียนโปรแกรมโค้ดที่มีจำนวนของบรรทัดมากๆ เราสามารถแบ่งโปรแกรมโค้ดออกเป็นหลายๆส่วน โดยเก็บไว้ในแต่ละไฟล์แยกกัน ทำให้ไฟล์ย่อยเหล่านี้มีขนาดเล็ก ซึ่งจะดีกว่าเขียนโปรแกรมโค้ดทั้งหมดให้อยู่ในไฟล์เดียวกันและมีขนาดใหญ่ ตัวอย่างการใช้ เช่น

```
#include <stdio.h>
#include "myheader.h"
```

ถ้าชื่อของไฟล์เขียนอยู่ระหว่างเครื่องหมาย < > พรีโปรเซสเซอร์ก็จะค้นหาไฟล์ดังกล่าวในไดเรคทรอรี่ที่กำหนดไว้โดยคอมไพล์เลอร์ แต่ถ้าเขียนชื่อไฟล์ที่มีเครื่องหมาย " ปิดหัวและท้าย พรีโปรเซสเซอร์ก็จะค้นหาไฟล์ในไดเรคทรอรี่ที่เรากำลังทำงานอยู่ (Working directory) ก่อน ถ้าพบว่าไม่มีไฟล์ที่ต้องการอยู่ในไดเรคทรอรี่ดังกล่าว ก็จะค้นหาไฟล์ในไดเรคทรอรี่มาตราฐานของคอมไพล์เลอร์ต่อไป ถ้าไม่พบไฟล์ที่ต้องการคอมไพล์เลอร์ก็จะหยุดการทำงานและแจ้งความผิดพลาดดังกล่าวให้เราทราบ

สำหรับระบบปฏิบัติการแบบยูนิกซ์ (UNIX) ไฟล์ส่วนหัว (Header Files) ที่เรามักจะแทรกเข้าในโปรแกรมโค้ดของเราเสมอๆ มักจะพบอยู่ในไดเรคทรอรี่ต่อไปนี้ แต่อย่างไรก็ตามก็ขึ้นอยู่กับเครื่องคอมพิวเตอร์หรือเวิรค์สเตชัน (Workstation) แต่ละยี่ห้อที่ใช้ซึ่งมีโครงสร้างของระบบไฟล์ข้อมูลสำหรับยูนิกซ์ที่แตกต่างกันออกไป

```
/usr/include
/usr/local/include
```

5.4 การเลือกแปลบางส่วนของโปรแกรมโค้ด

ในบางครั้งเราต้องการเลือกแปลโปรแกรมโค้ดของเราเฉพาะบางส่วนเท่านั้น เช่น ในช่วงเวลาที่โปรแกรมยังไม่เสร็จเรียบร้อย และต้องมีการตรวจสอบการทำงานของโปรแกรมที่เขียนขึ้นก่อน โดยมีการเขียนคำสั่งเพิ่มเติมไว้ตั้งแต่แรกที่ช่วยในการตรวจสอบการทำงานของโปรแกรม เช่น รายงานการทำงานของโปรแกรมให้ทราบเป็นระยะๆ พิมพ์ข้อความออกทางจอภาพเวลาโปรแกรมกำลังทำงานเพื่อให้ทราบว่าโปรแกรมกำลังดำเนินการในขั้นตอนใดหรือมีปัญหาอย่างไร เป็นต้น หลังจากที่ได้ตรวจสอบแล้ว เมื่อนำโปรแกรมนี้ไปใช้งานจริง คำสั่งที่รายงานข้อมูลเกี่ยวกับการทำงานของโปรแกรมก็ไม่จำเป็น เพราะขั้นตอนหรือคำสั่งเหล่านั้นใช้เฉพาะสำหรับช่วยในการตรวจสอบการทำงานของโปรแกรมเท่านั้น ดังนั้นก็จะเป็นการดีถ้าหากสามารถกำหนดได้ว่า เมื่อไหรคอมไพล์เลอร์ควรจะคอมไพล์ขั้นตอนใดส่วนใดของโปรแกรมโค้ด เช่น ถ้าไม่ต้องการคอมไพล์บางคำสั่งหรือขั้นตอนการทำงานใด เราก็สามารถเลือกกำหนดได้ หรือกำหนดเงื่อนไขว่า ถ้าเงื่อนไขเป็นจริงก็ให้คอมไพล์โค้ดส่วนหนึ่ง ถ้าเงื่อนไขเป็นเท็จก็ให้คอมไพล์อีกส่วนหนึ่งโดยไม่จำเป็นต้องลบคำสั่งหรือขั้นตอนเหล่านั้นออกจากโปรแกรมโค้ด ในภาษาซีมีแมโครหลาย ตัวสำหรับหน้าที่ดังกล่าว

5.4.1 คำสั่ง #if, #else, #elif, #endif

ไดเรคทีฟเหล่านี้เราใช้ในการกำหนดว่า เมื่อใดเราจะเลือกส่วนของโปรแกรมโค้ดใดสำหรับการคอมไพล์ โดยอาศัยเงื่อนไขเป็นตัวกำหนด รูปแบบอย่างง่ายที่สุดของการคอมไพล์แบบตัวเลือกคือ #if #endif

```
#if condition_expression
      statement_sequence
#endif
```

ตัวอย่างการใช้งานเช่น

```
/* 1 */    #include <stdio.h>
/* 2 */
/* 3 */    #define VERSION  1.1
/* 4 */
/* 5 */    int main()
/* 6 */    {
/* 7 */      #if (VERSION <= 1.2)
/* 8 */         printf("Program Version %3.1f\n", VERSION);
/* 9 */         #define VERSION 1.2
/*10 */      #endif
/*11 */         return 0;
/*12 */    }
```

โปรแกรมตัวอย่างนี้มีการใช้ไดเรคทีฟ #if #endif ถ้าแมโคร VERSION มีค่าน้อยกว่าหรือเท่ากับ 1.2 แล้วคอมไพล์เลอร์จะแปลคำสั่งในบรรทัดที่ 8 และ 9 ถ้าเงื่อนไขเป็นเท็จก็จะไม่มีการคอมไพล์คำสั่งในบรรทัดทั้งสอง

นอกจากนี้เราสามารถใช้ไดเรครีฟ #else ในโครงสร้างประโยคของ #if #endif ดังนี้

```
#if condition_expression
      statement_sequence1
#else
      statement_sequence2
#endif
```

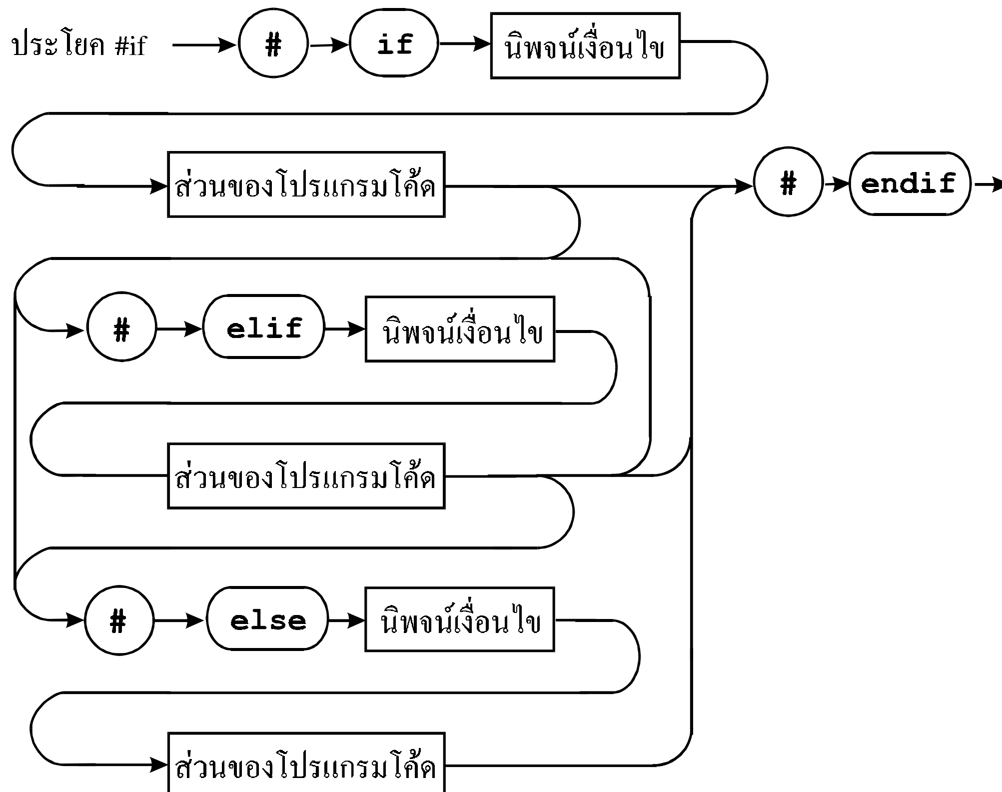
หรือถ้าเราต้องการสร้างประโยคเงื่อนไขที่ซ้อนกันหลายชั้น เราก็สามารถทำได้ โดยใช้ #elif เพิ่มเติม เช่น

```
#if condition_expression1
    statement_sequence1
#elif condition_expression2
       statement_sequence2
```

```
#endif
```

หรือ

```
#if condition_expression1
      statement_sequence1
#elif condition_expression2
      statement_sequence2
#else
      statement_sequence3
#endif
```



รูปภาพที่ 5.2 โครงสร้างประโยคของ #if - #elsif - #else

ตัวอย่างการใช้งานเช่น

```
#define MS_DOS    0
#define UNIX      1

#define OS   UNIX

#if (OS == UNIX)
  #define  PROGRAM_DIR  "/home/user01/Program"
#elif (OS == MS_DOS)
  #define  PROGRAM_DIR  "C:\HOME\USER01\PROGRAM"
#else
  #define  PROGRAM_DIR  ""
#endif
```

5.4.2 คำสั่ง #ifdef, #ifndef

ไดเรคทีฟทั้งสองเป็นไดเรคทีฟที่มีหน้าที่เฉพาะเจาะจงคือ ใช้ในการตรวจสอบว่าได้มีการนิยามแมโครสำหรับชื่อที่กำหนดแล้วหรือไม่ ซึ่งมีรูปแบบการใช้คล้ายกับประโยคคำสั่ง #if ดังนี้

```
#ifdef macro_name
      statement_sequence
#endif

#ifdef macro_name
      statement_sequence1
#else
      statement_sequence2
#endif


#ifndef macro_name
      statement_sequence
#endif

#ifndef macro_name
      statement_sequence1
#else
      statement_sequence2
#endif
```

เราสามารถเขียนไดเรคทีฟ #ifdef และ #ifndef โดยใช้ #if ได้

```
#if defined macro_name

#if !defined macro_name
```

หมายถึง

```
#ifdef macro_name

#ifndef macro_name
```

ซึ่งจะต้องใช้ร่วมกับคำว่า defined และ !defined ตามลำดับ

5.5 คำสั่ง #undef

แมโครใดๆที่ถูกนิยามไว้โดยคำสั่ง #define เราสามารถยกเลิกได้โดยใช้คำสั่ง #undef ตามปรกติแล้วแมโครได้ที่ได้นิยามแล้วครั้งหนึ่ง ถ้าเราจะนิยามอีกครั้งที่ใช้แทนข้อความหรือชุดคำ

สั่งใดๆจะต้องใช้คำสั่ง #undef ยกเลิกการนิยามแมโครครั้งก่อนแล้วจึงสามารถใช้คำสั่ง #define นิยามแมโครสำหรับตัวระบุชื่อที่ต้องการใหม่อีกครั้งได้

```
#undef macro_name
```

ตัวอย่างการใช้งาน เช่น

```
#define MAX_SIZE  1024

#undef  MAX_SIZE
#define MAX_SIZE  512
```

5.6 คำสั่ง #error

ไดเรคทีฟตัวนี้บังคับให้คอมไพล์เลอร์หยุดการทำงาน พร้อมกับพิมพ์ข้อความ หรือเหตุผลของการหยุดการคอมไพล์โปรแกรมโค้ด ตามปรกติแล้วไดเรคทีฟ #error จะใช้กับประโยคไดเรคทีฟเงื่อนไข เช่น #if #ifdef หรือ #ifndef เป็นต้น รูปแบบการใช้ไดเรคทีฟ #error

```
#error error_message
```

ซึ่งจะส่งผลให้คอมไพล์เลอร์พิมพ์ข้อความ

```
Error directive: error_message
```

ออกทางจอภาพในระหว่างเวลาของการคอมไพล์ ตัวอย่างเช่น สมมุติว่า เราต้องการจะตรวจสอบว่า ค่าคงที่เชิงสัญลักษณ์ MAX_LENGTH มีค่ามากกว่า 1024 หรือไม่ ถ้าเงื่อนไขไม่ถูกต้องเราก็ใช้ประโยค #error ในการหยุดการคอมไพล์ และเพื่อแจ้งให้ผู้ใช้ทราบว่ามีการนิยามค่าของ MAX_LENGTH ที่ไม่ถูกตามเงื่อนไขดังกล่าว

```
#if (MAX_LENGTH > 1024)
#error MAX_LENGTH must be less than 1024.
#endif
```

5.7 คำสั่ง #line

ไดเรคทีฟ #line ใช้ในการกำหนดแมโครชนิดหนึ่งที่ใช้แทนหมายเลขบรรทัดในโปรแกรมโค้ด (__LINE__) ซึ่งมีประโยชน์มากในการตรวจสอบการทำงานของโปรแกรม รูปแบบการใช้งานเป็นดังนี้

```
#line   line_number
```

ตัวอย่างการใช้งาน เช่น

```
/*1*/ #include <stdio.h>
/*2*/ int main()
/*3*/ {
/*4*/       printf ("Current Line : %d\n", __LINE__);
/*5*/ #line 100
/*6*/       printf ("Current Line : %d\n", __LINE__);
/*7*/
/*8*/ #line 200
/*9*/       printf ("Current Line : %d\n", __LINE__);
/*10*/            return 0;
/*11*/      }
```

ผลของโปรแกรมคือ

```
Current Line : 4
Current Line : 100
Current Line : 200
```

เราจะเห็นได้ว่า ค่าของ __LINE__ ในบรรทัดที่ 4 มีค่าเท่ากับ 4 เพราะเป็นค่าของเลขบรรทัดในขณะนั้น ซึ่งตามปรกติแล้ว __LINE__ จะมีค่าเพิ่มขึ้นทีละหนึ่งเมื่อเรานับบรรทัดของโปรแกรมโค้ดจากบนลงล่าง ในบรรทัดที่ 5 และ 8 เราใช้ไดเรคทีฟ `#line` ในการกำหนดค่าของแมโคร __LINE__ ใหม่ โดยค่าของ แมโครนี้จะมีค่าเท่ากับ 100 และ 200 ตามลำดับ เมื่อเราตั้งค่าของเลขบรรทัดใหม่โดยใช้ ไดเรคทีฟ `#line` ในบรรทัดถัดไป ค่าของ __LINE__ จะมีค่าเท่ากับค่าใหม่และในบรรทัดต่อไปค่าของ __LINE__ ก็จะเพิ่มขึ้นทีละหนึ่งตามปรกติ จนกว่าจะมีการใช้ไดเรคทีฟ `#line` ในการตั้งค่าของตัวนับสำหรับเลขบรรทัดใหม่อีกครั้ง

แบบฝึกหัดท้ายบท

1. จงอธิบายความแตกต่างระหว่างแมโครที่นิยามไว้สำหรับสามกรณีต่อไปนี้

(1) `#define SQUARE(a)   a*a`

(2) `#define SQUARE(a)  (a)*(a)`

(3) `#define SQUARE(a)  ((a)*(a))`

และลองคำนวณค่าของนิพจน์ต่อไปนี้ สำหรับทั้งสามกรณี

```
SQUARE(-2+3)/5
```

2. จงเขียนแมโคร F, G, H และ I ที่มีอาร์กิวเมนท์ X, Y และ Z ตามรายละเอียดการทำงานของแมโครที่ กำหนดไว้ในตารางข้างล่างนี้โดยใช้ #define

แมโคร	การทำงานของแมโคร
F(X,Y,Z)	(X AND Y) OR (NOT X AND Z)
G(X,Y,Z)	(X AND Z) OR (Y AND NOT Z)
H(X,Y,Z)	X XOR Y XOR Z
I(X,Y,Z)	Y XOR (X OR NOT Z)

3. จงเขียนแมโครโดยใช้ #define สำหรับหน้าที่ต่อไปนี้

แมโครใช้ในการเลื่อนบิตของแถวบิตขนาด 32 บิตไปทางซ้ายมือ เป็นจำนวนตำแหน่งใดๆที่ต้องการซึ่งกำหนดโดยอาร์กิวเมนต์ และบิตที่อยู่ทางซ้ายมือที่ตามปรกติแล้วจะถูกตัดทิ้งไปเมื่อเลื่อนแถวบิตจะต้องย้ายไปอยู่ในตำแหน่งทางขวามือ เสมือนกับว่าหัวและท้ายของแถวบิตต่อกันเป็นวงกลม เช่น สมมุติว่าเรากำหนดให้แมโครมีชื่อและอาร์กิวเมนท์ดังต่อไปนี้

```
ROTATE_LEFT(x, n)
```

ถ้า x มีค่าเท่ากับเลขที่อยู่ในรูปของระบบฐานสองขนาด 32 บิต เช่น

```
10011100 11011101 00000101 00000110
```

และ n มีค่าเท่ากับ 1 ผลที่ได้จะต้องเป็นดังนี้

```
00111001 10111010 00001010 00001101
```

จงเขียนแมโครในลักษณะดังข้างต้นอีกครั้งแต่ใช้สำหรับเลื่อนบิตไปทางขวา

4. จงเขียนแมโครที่ใช้ในการตรวจสอบว่า บิตหมายเลข N ที่นับจากขวาไปซ้ายในแถวบิตของค่าคงที่ X ซึ่งเป็นเลขจำนวนเต็มมีค่าเป็นหนึ่งหรือศูนย์ ซึ่งแมโครจะมีรูปแบบนี้ `IsBitSet(X,N)`