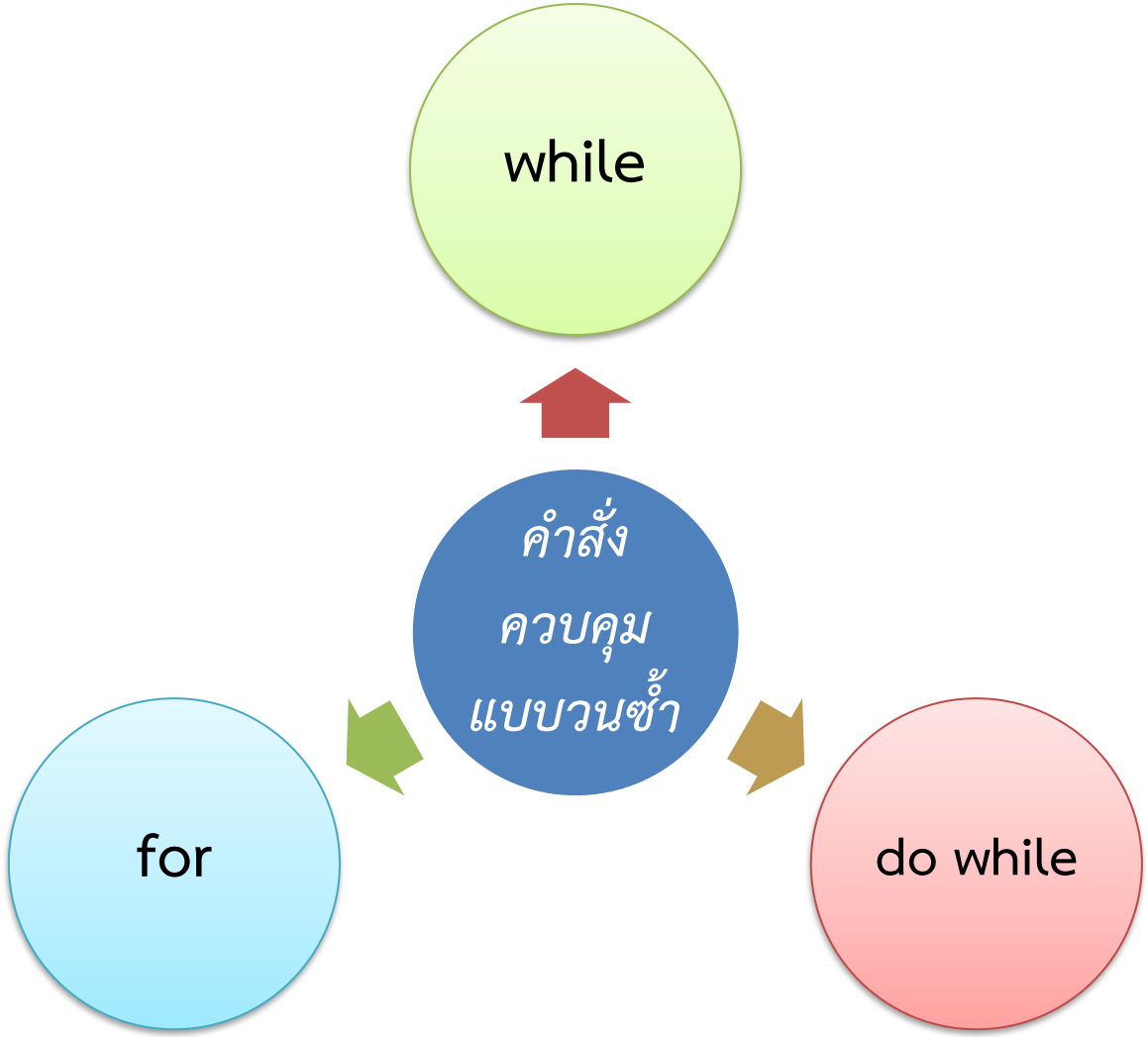
คำสั่งควบคุมการทำซ้ำ

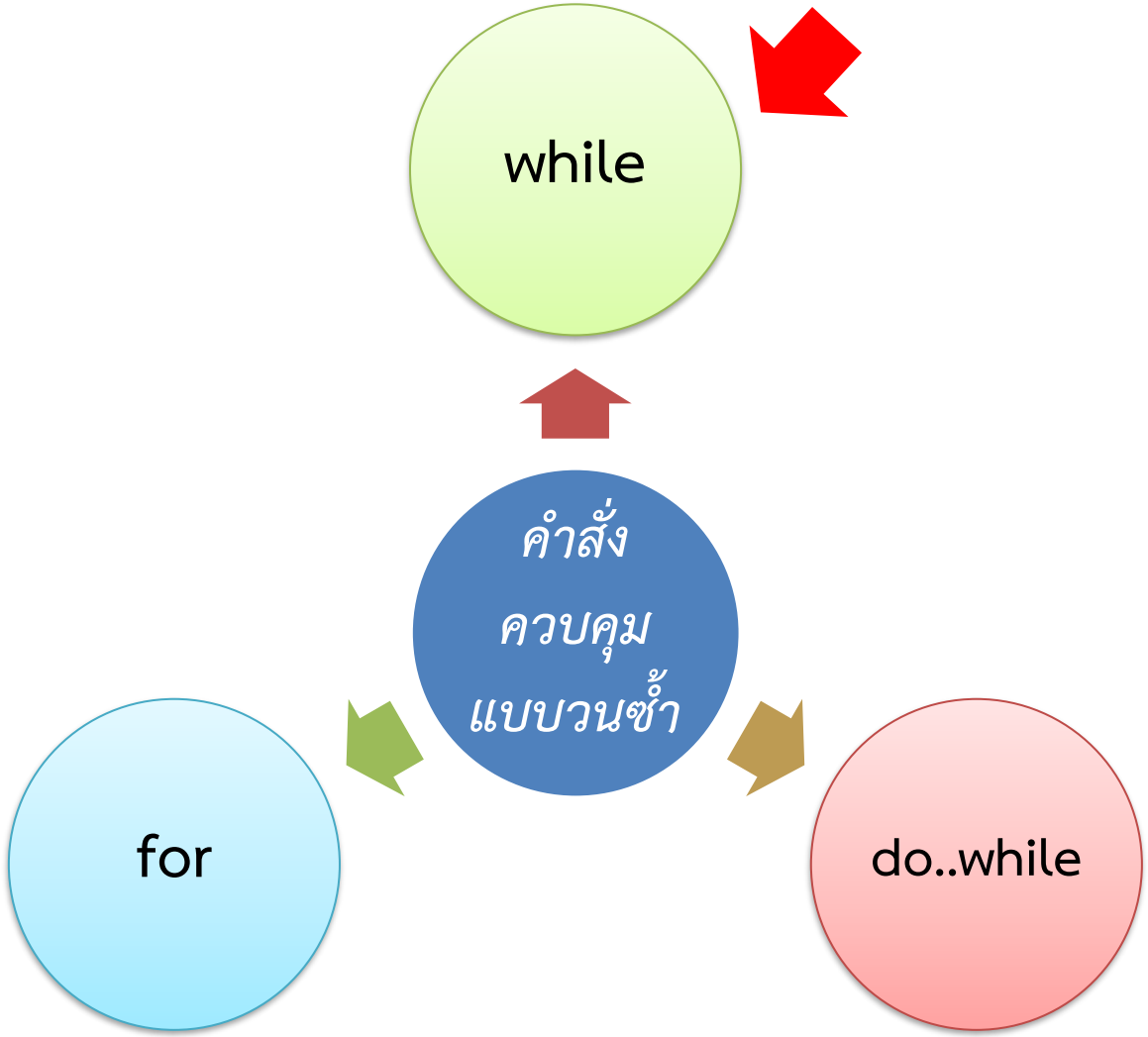


การทำซ้ำ

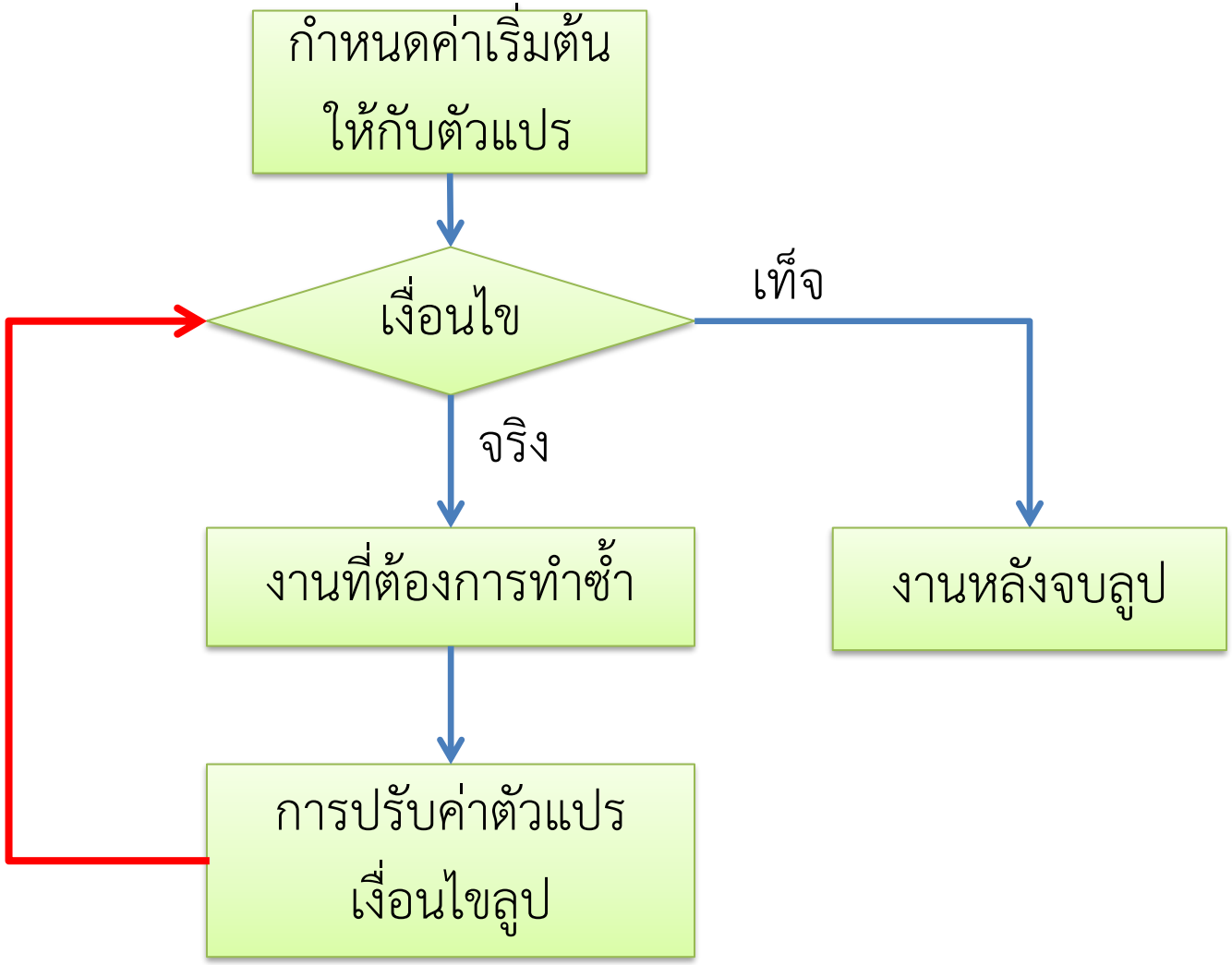
การทำซ้ำหรือการวนลูป จริงแล้วไม่ใช่เรื่องยาก หากเราเข้าใจองค์ประกอบพื้นฐานของการวนลูป ซึ่งมีของอยู่ 4 อย่าง

1. การกำหนดค่าเริ่มต้นของตัวแปรต่าง ๆ ก่อนเข้าลูป
2. เงื่อนไขที่จะให้ทำลูป (จะอยู่ก่อนหรืออยู่ด้านหลังคำสั่งที่อยากทำก็ได้)
3. งานที่ต้องการทำซ้ำ
4. การปรับค่าตัวแปรเงื่อนไขลูป (คือการเปลี่ยนค่าตัวแปรที่เกี่ยวพันกับเงื่อนไขที่จะให้ทำหรือจบลูป)

คำสั่งควบคุมแบบทำซ้ำ



คำสั่ง while ( )


กำหนดค่าเริ่มต้น
ให้กับตัวแปร
เงื่อนไข
เท็จ
จริง
งานที่ต้องการทำซ้ำ
งานหลังจบลูป
การปรับค่าตัวแปร
เงื่อนไขลูป

รูปแบบทั่วไปของลูป while ( ) { … }

```
การกำหนดค่าเริ่มต้นตัวแปรก่อนเข้าลูป
while ( เงื่อนไข ) {
    งานที่ต้องการทำซ้ำ
    การปรับค่าตัวแปรเงื่อนไขลูป
}
… งานหลังจบลูป …
```

- <u>**สิ่งไหนที่จะให้ทำซ้ำบ่อย ๆ ต้องอยู่ในลูป สิ่งไหนไม่ต้องทำซ้ำอยู่ข้างนอก**</u>
- หลักการทำงานก็คือว่า ถ้าเงื่อนไขของลูปเป็นจริง โปรแกรมจะทำสิ่งที่อยู่ภายในลูป ซึ่งก็คือ ‘งานที่จะให้ทำ’ และ ‘การแก้ไขตัวแปรเงื่อนไขลูป’
- เป็นไปได้ที่เงื่อนไขของลูปจะไม่เป็นจริงตั้งแต่แรก ทำให้ไม่มีการทำงานใด ๆ ภายในลูปเลยแม้แต่ครั้งเดียว

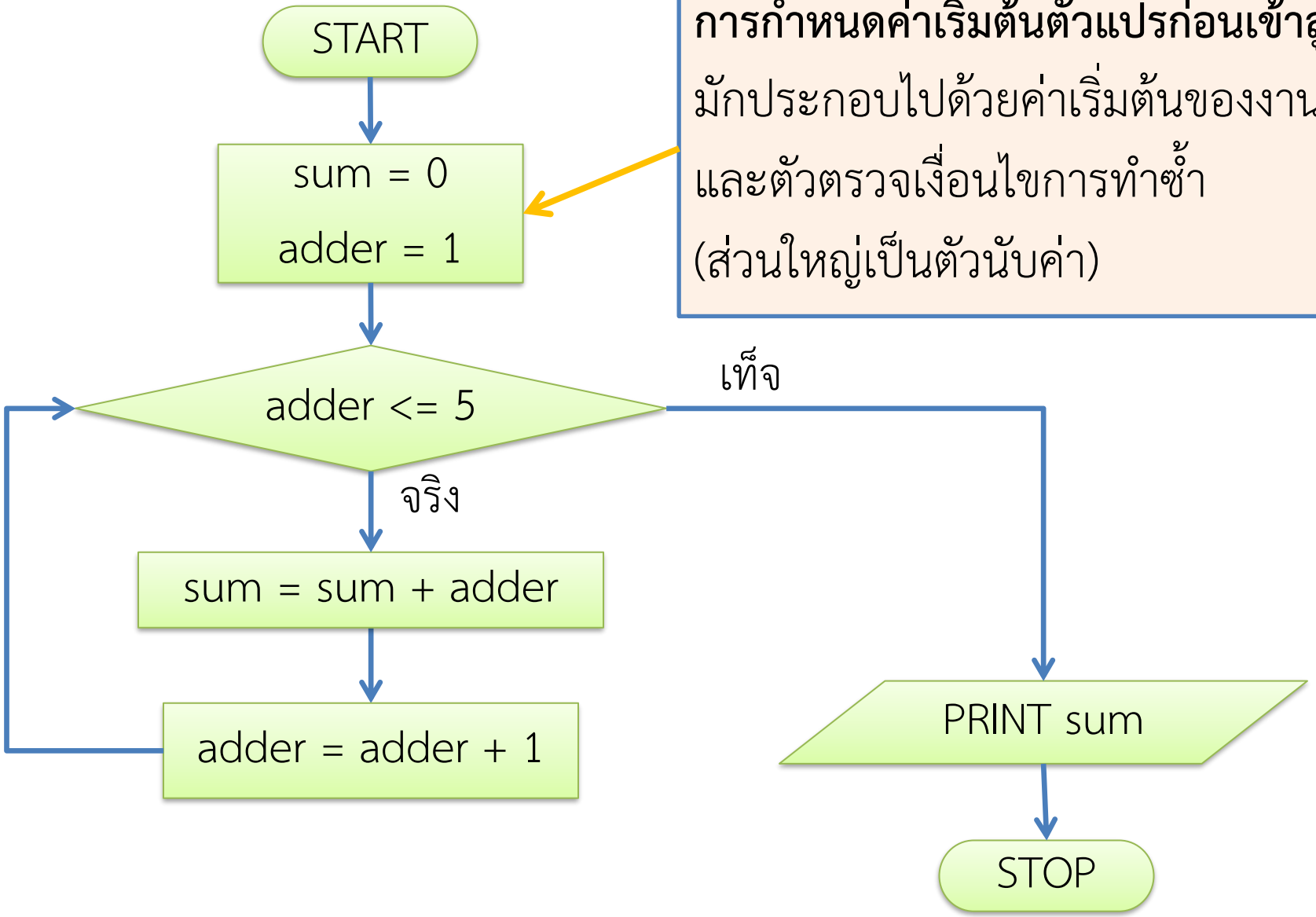
ตัวอย่างการทำงานของลูป

โจทย์ จงเขียนโฟลวชาร์ตและโค้ดภาษาซีสำหรับการหาผลบวกของเลขจำนวนเต็มที่มีค่าอยู่ในช่วงปิด 1 ถึง 5 (ช่วงปิดจะรวมเลข 1 และ 5 ด้วย) จากนั้นพิมพ์ผลลัพธ์ออกมาทางจอภาพ (บังคับให้ใช้ลูปค่อย ๆ บวกเลขทีละค่า)

วิเคราะห์

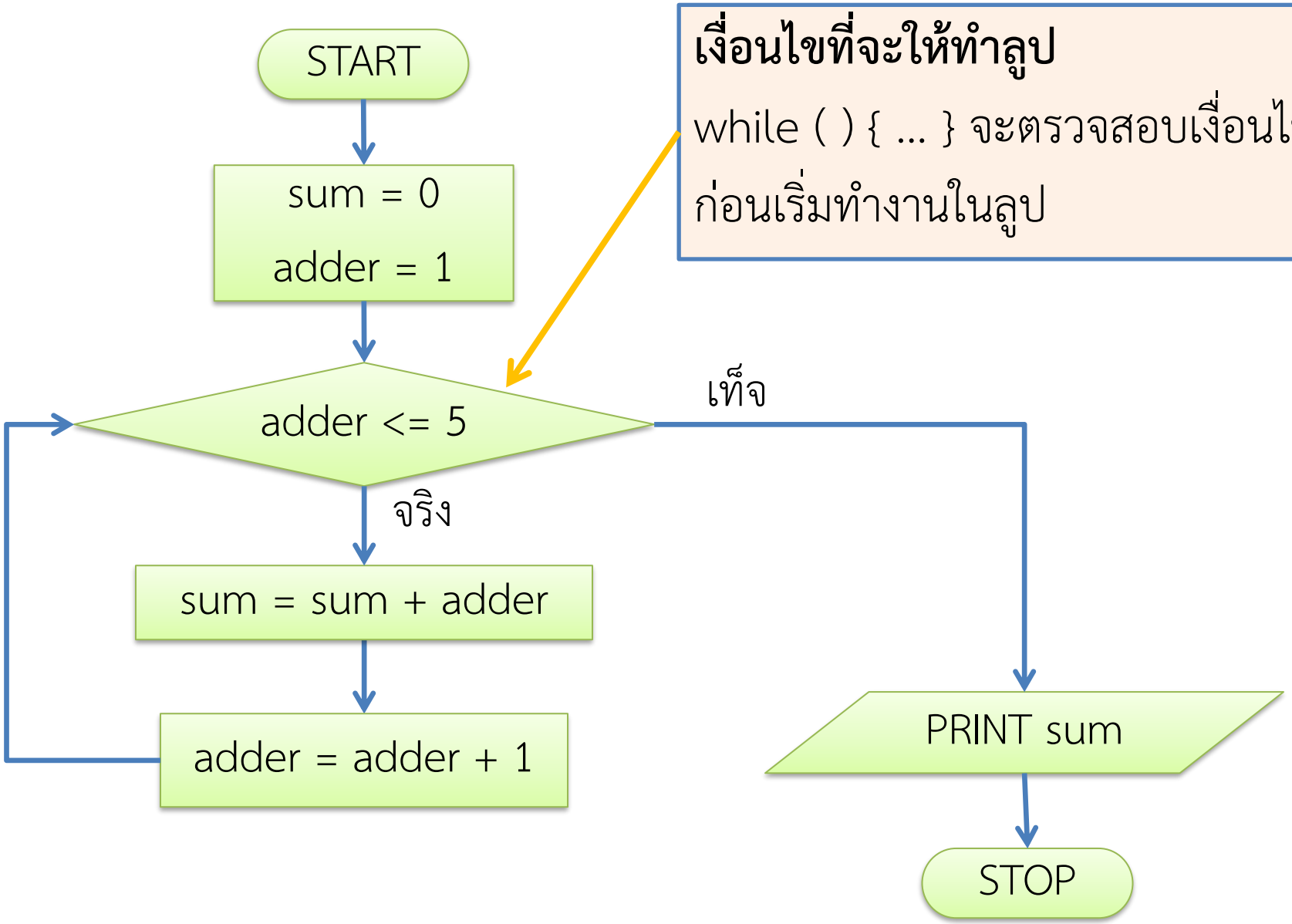
1. ไม่มีการรับข้อมูลเข้าจากผู้ใช้ แต่จะต้องสร้างตัวเลขขึ้นมาเอง
2. <u>งานที่ต้อง**ทำซ้ำ**แน่ ๆ คือการบวกเลข</u>
3. ต้องมีการนับเลขที่จะบวกเพิ่มขึ้นเรื่อย ๆ เพื่อให้เปลี่ยนตัวบวกจาก 1 ไปเป็น 2, 3, 4 และ 5 ได้
4. เงื่อนไขที่ควรใช้ในการทำงานคือ ‘ตัวบวกต้องอยู่ในช่วง 1 ถึง 5’

โฟลวชาร์ต การวนซ้ำบวกเลข



การกำหนดค่าเริ่มต้นตัวแปรก่อนเข้าลูป
มักประกอบไปด้วยค่าเริ่มต้นของงาน
และตัวตรวจเงื่อนไขการทำซ้ำ
(ส่วนใหญ่เป็นตัวนับค่า)

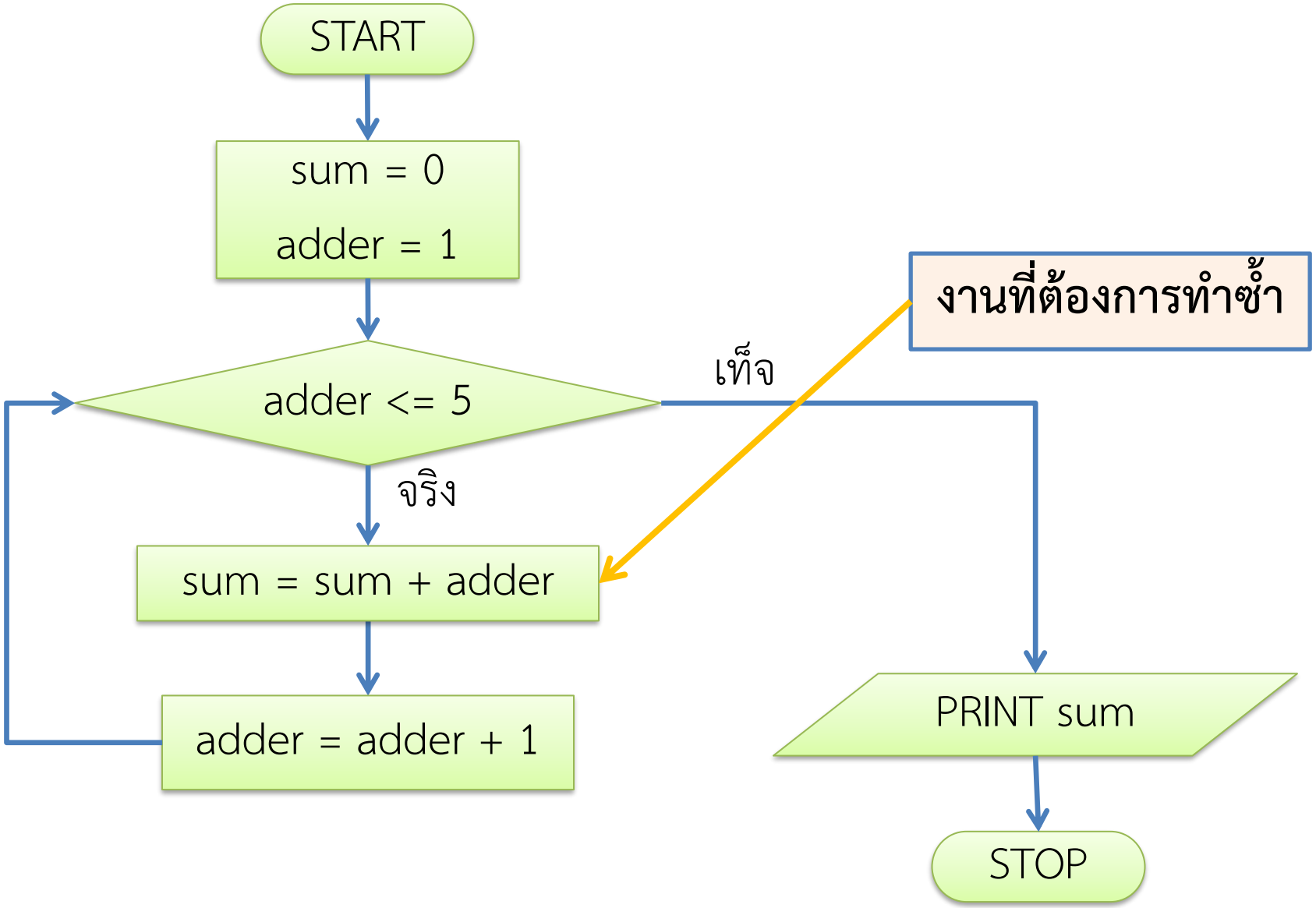
โฟลวชาร์ต การวนซ้ำบวกเลข (2)



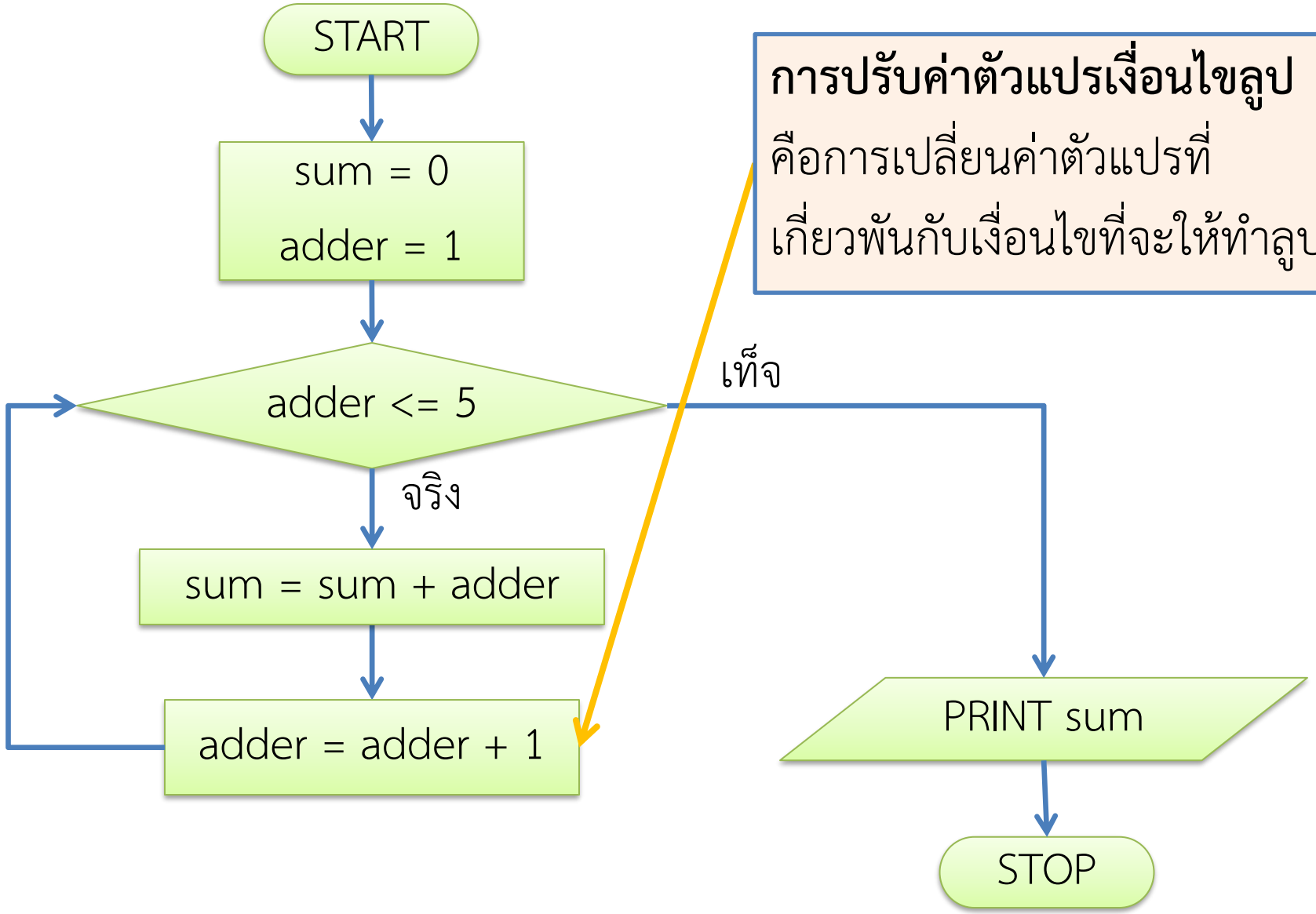
เงื่อนไขที่จะให้ทำลูป

while ( ) { ... } จะตรวจสอบเงื่อนไขก่อนเริ่มทำงานในลูป

โฟลวชาร์ต การวนซ้ำบวกเลข (3)



โฟลวชาร์ต การวนซ้ำบวกเลข (4)



การปรับค่าตัวแปรเงื่อนไขลูป
คือการเปลี่ยนค่าตัวแปรที่
เกี่ยวพันกับเงื่อนไขที่จะให้ทำลูป

โค้ดภาษาซี

```
void main() {
    int sum = 0;
    int adder = 1;

    while (adder <= 5) {
        sum = sum + adder;
        adder = adder + 1;
    }

    printf("%d", sum);
}
```

การกำหนดค่าตัวแปรเริ่มต้นก่อนเข้าลูป

เงื่อนไขที่จะให้ทำลูป

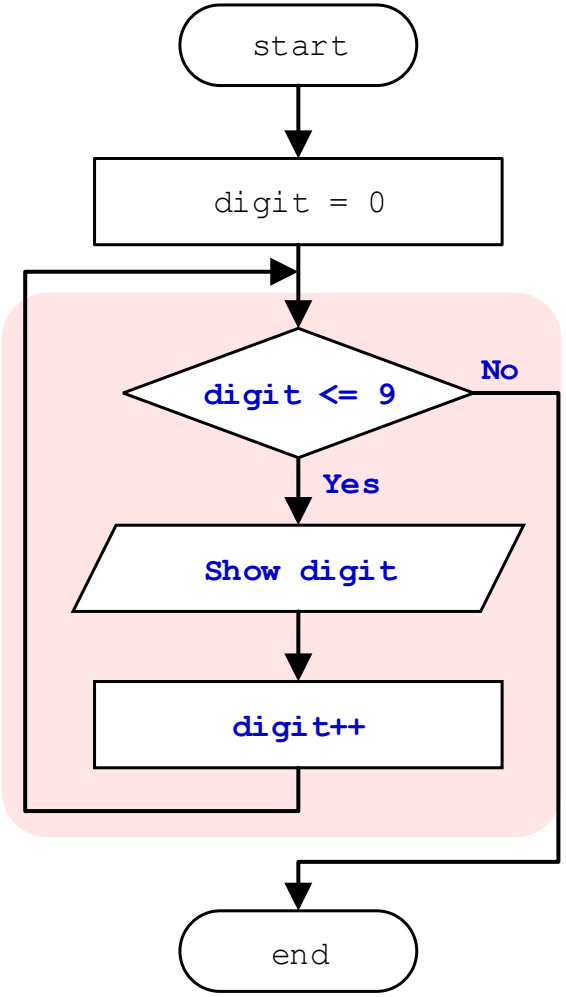
งานที่ต้องการทำซ้ำ

การปรับค่าตัวแปรเงื่อนไขลูป

งานหลังจบลูป

แยกให้ออกด้วยว่า งานไหนที่ต้องทำซ้ำ งานไหนที่ไม่ต้องทำซ้ำ

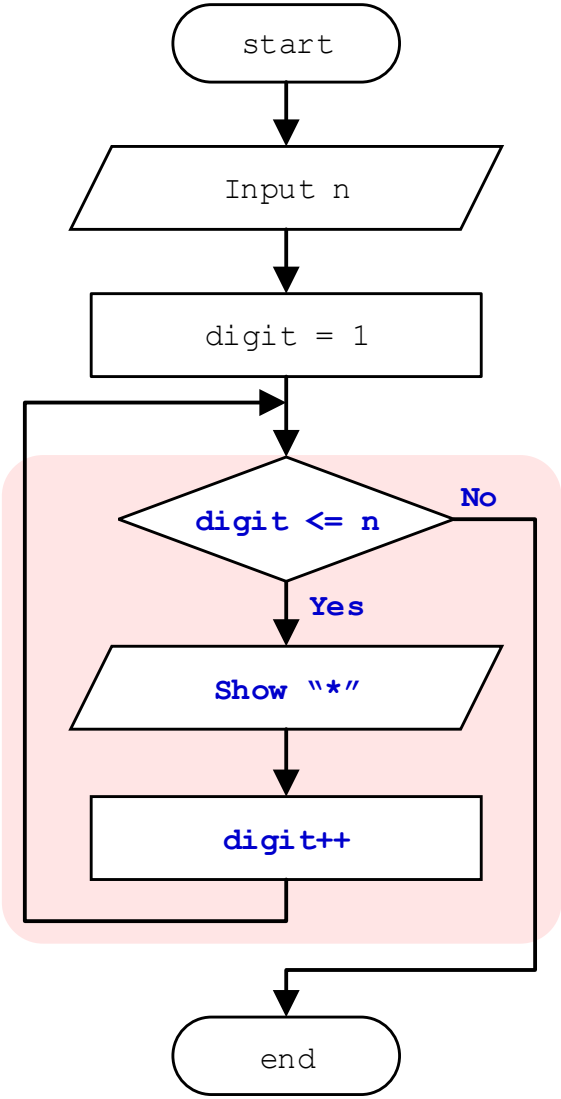
คำสั่ง while




```
void main()
{       int digit = 0;
        while(digit <= 9) {
                printf("%d", digit);
                digit++;
        }
}
```

ผลลัพธ์ที่ได้คือ ...

คำสั่ง while




```
void main()
{       int n;
        scanf("%d", &n);
        int digit = 1;
        while(digit <= n) {
                printf("*");
                digit++;
        }

}
```

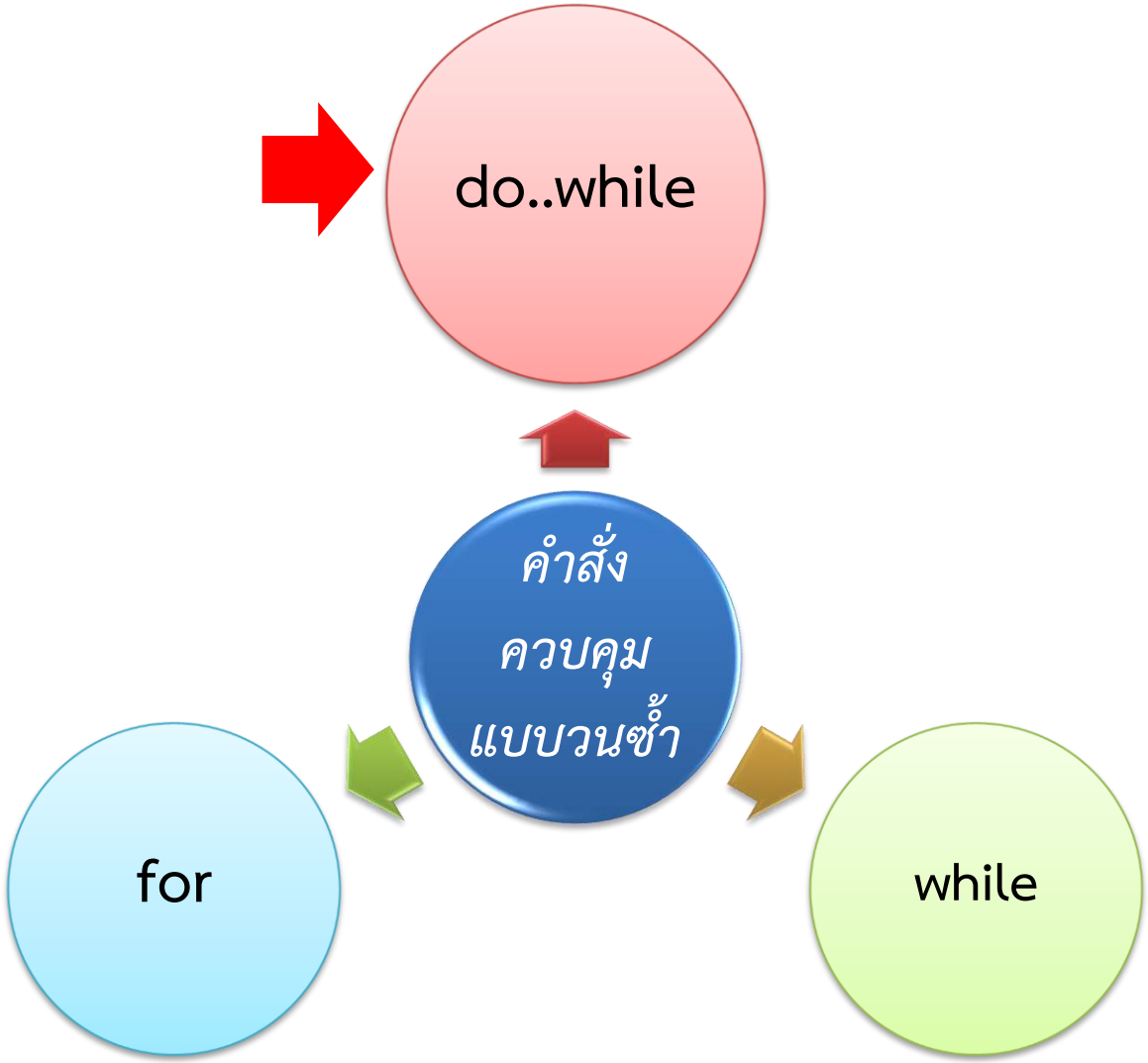
ผลลัพธ์ที่ได้คือ ? ถ้า n=5

คำสั่ง while

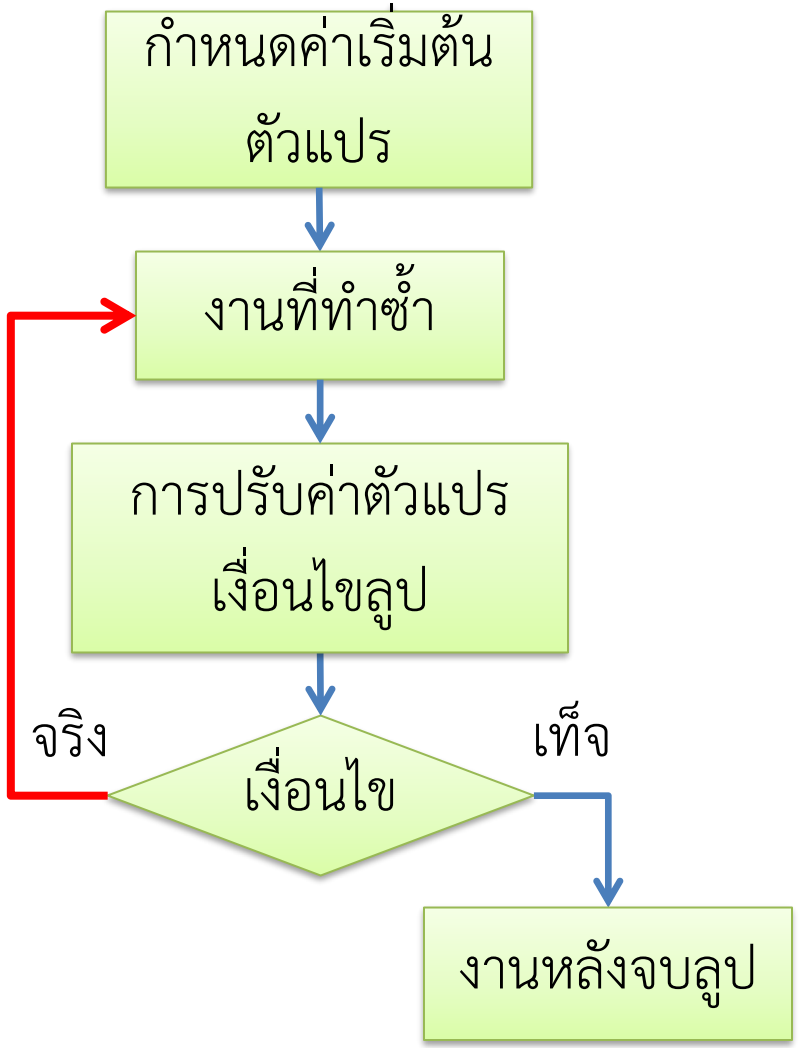
```
void main()
{       int n, fac = 1;

        printf("Enter positive number");
        scanf("%d",&n);
        while(n>=1)
        {
          fac = fac*n;
          n = n-1;
        }
        printf("%d",fac);
}
```

ผลลัพธ์ที่ได้คือ ? ถ้า n=4

คำสั่งควบคุมแบบวนซ้ำ



คำสั่ง Do .. While( );



รูปแบบทั่วไปของลูป do { ... } while ( );

```
การกำหนดค่าเริ่มต้นตัวแปรก่อนเข้าลูป
do {
    งานที่ต้องการทำซ้ำ
    การปรับค่าตัวแปรเงื่อนไขลูป
} while ( เงื่อนไข );
... งานหลังจบลูป ...
```

- เป็นลูปที่มีเอกลักษณ์เฉพาะตัว คือมีลำดับการทำงานที่แตกต่างจากคนอื่น เพราะการตรวจเงื่อนไขถูกกระทำที่ด้านท้ายของลูป แทนที่จะเป็นด้านบน
- ส่วนตรงกลางของลูปยังไงก็ต้องถูกทำอย่างน้อยหนึ่งครั้ง เพราะไม่มีเงื่อนไขใดจะไปขัดขวางมันได้ (เว้นแต่จะโดนคำสั่ง break;)

ตัวอย่างการใช้ลูป do .. while();

โจทย์ จงเขียนโค้ดภาษาซีสำหรับการหาผลบวกของเลขจำนวนเต็มที่มีค่าอยู่ในช่วงปิด 1 ถึง 5 (ช่วงปิดจะรวมเลข 1 และ 5 ด้วย) จากนั้นพิมพ์ผลลัพธ์ออกมาทางจอภาพ ให้เขียนด้วยการใช้ลูป do..while

วิเคราะห์

1. <u>งานที่ต้อง**ทำซ้ำ**แน่ ๆ คือการบวกเลข</u>
2. งานทำซ้ำนี้ต้องทำอย่างน้อยหนึ่งครั้ง ดังนั้นเราสามารถใช้ลูป do while ได้

ตัวอย่างโค้ดลูป do .. while();

```
void main() {
    int sum = 0;
    int i = 1;
    do {
        sum = sum + i;
        ++i;
    } while(i <= 5);

    printf("%d", sum);
}
```

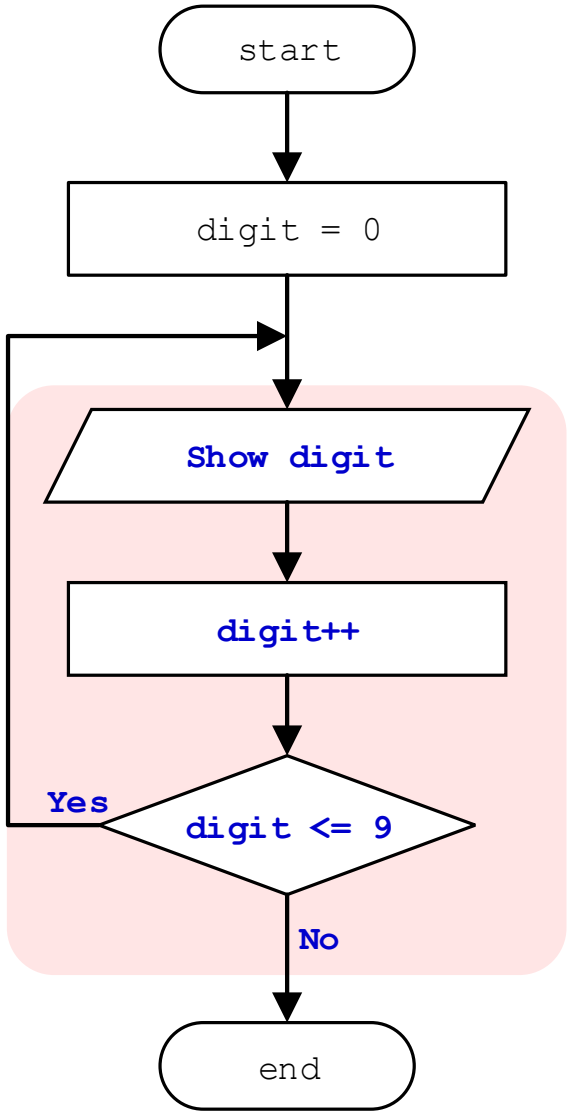
ใส่วงเล็บ { } ไว้หลังคำว่า do

ระวังลืมใส่เครื่องหมาย ;

เกร็ดเรื่องลูป do .. while();

- เงื่อนไขลูปอยู่กับคำว่า while เช่นเดิมและอยู่ในวงเล็บด้วย
- ต้องมีเครื่องหมายเซมิโคลอนตามหลังวงเล็บเงื่อนไข (ลักษณะเฉพาะ)
- ความนิยมของลูป do .. while( ); จะมีน้อยกว่าลูป while และ for
  เพราะบังคับให้ต้องทำงานอย่างน้อยหนึ่งครั้ง ในขณะที่ while กับ for มีอิสระมากกว่า
- ถึงความนิยมจะน้อยกว่า แต่อย่าลืมว่าของพวกนี้มันถูกคิดขึ้นมาเพื่อให้สอดคล้องกับธรรมชาติในการคิดที่หลากหลายของโปรแกรมเมอร์

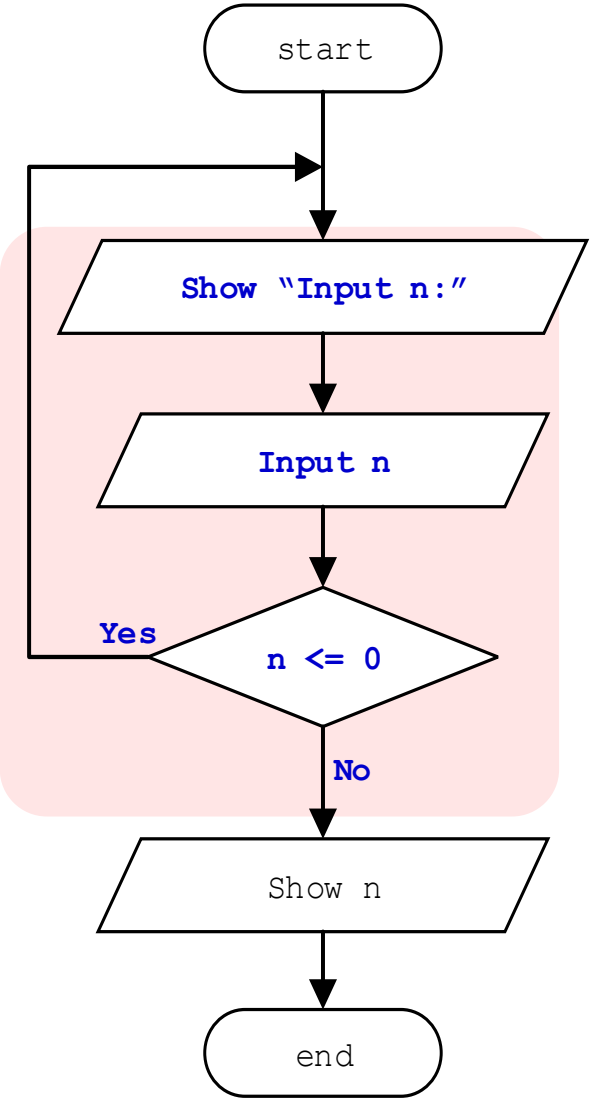
คำสั่ง do...while




```
void main()
{       int digit = 0;
        do
         {
            printf("%d", digit);
            digit++;
        } while(digit <= 9);
}
```

ผลลัพธ์ที่ได้คือ ...

คำสั่ง do...while




```
void main()
{       int n;
        do
        {
            printf("Input n: ");
            scanf("%d", &n);
        } while(n <= 0);
        printf("n = %d",n);
}
```

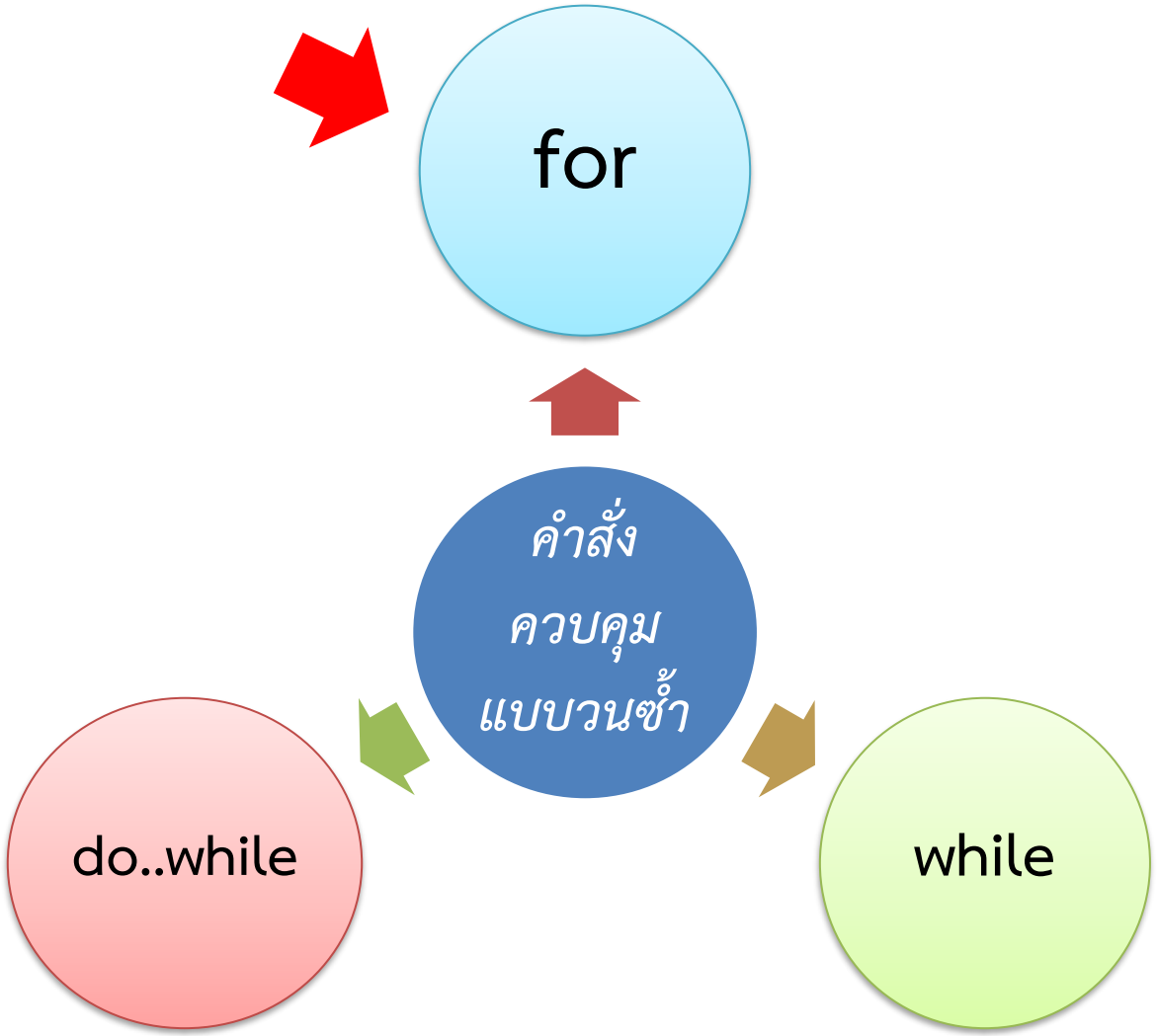
ผลลัพธ์ที่ได้คือ ? ถ้า n= 1

ผลลัพธ์ที่ได้คือ ? ถ้า n= -1

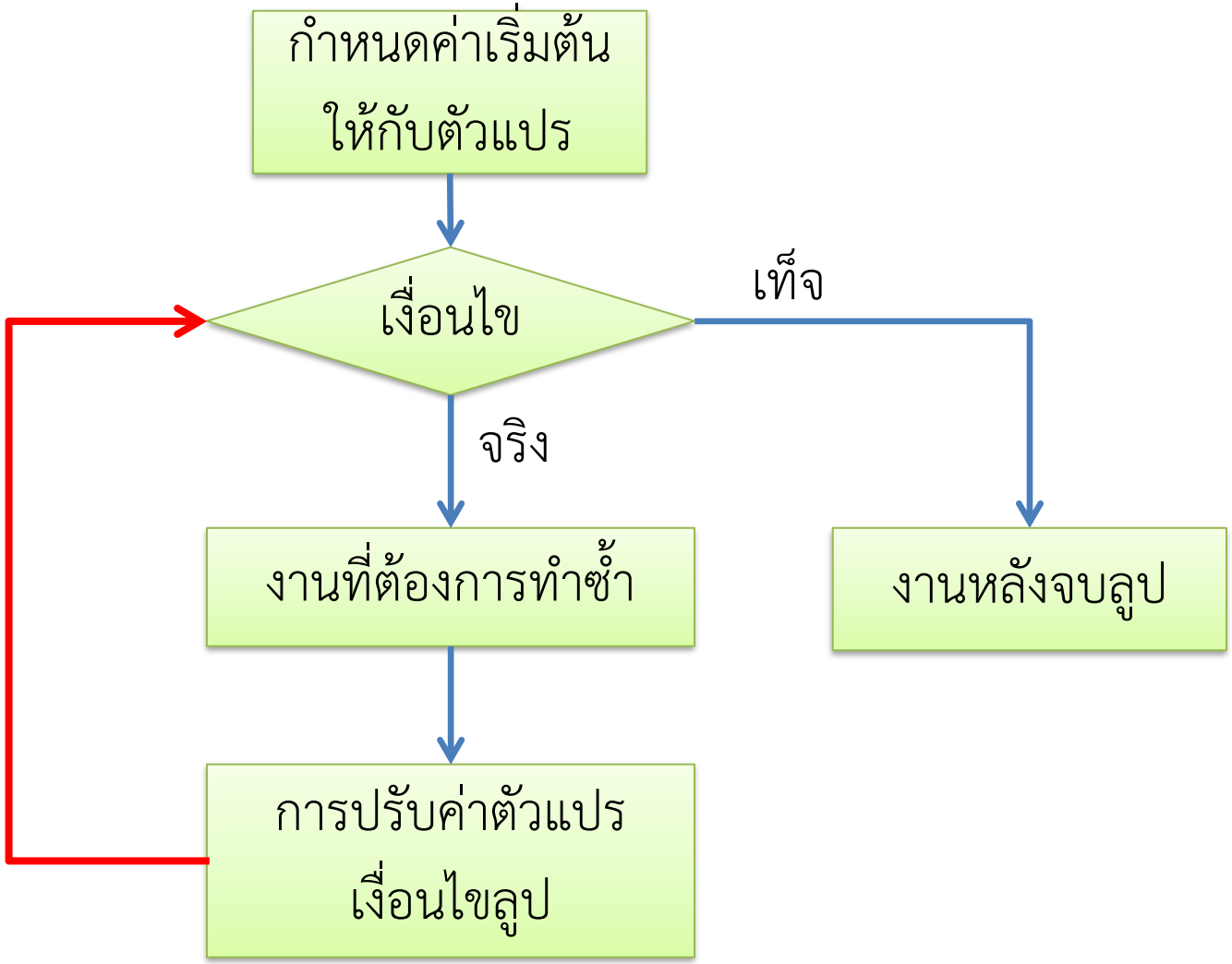
คำสั่ง do...while

```
void main()
{       int n, fac = 1;
        printf("Enter positive number");
        scanf("%d",&n);
        do
        {
            fac = fac*n;
            n = n-1;
        } while(n>=1)
        printf("%d",fac);
}
```

คำสั่งควบคุมแบบวนซ้ำ



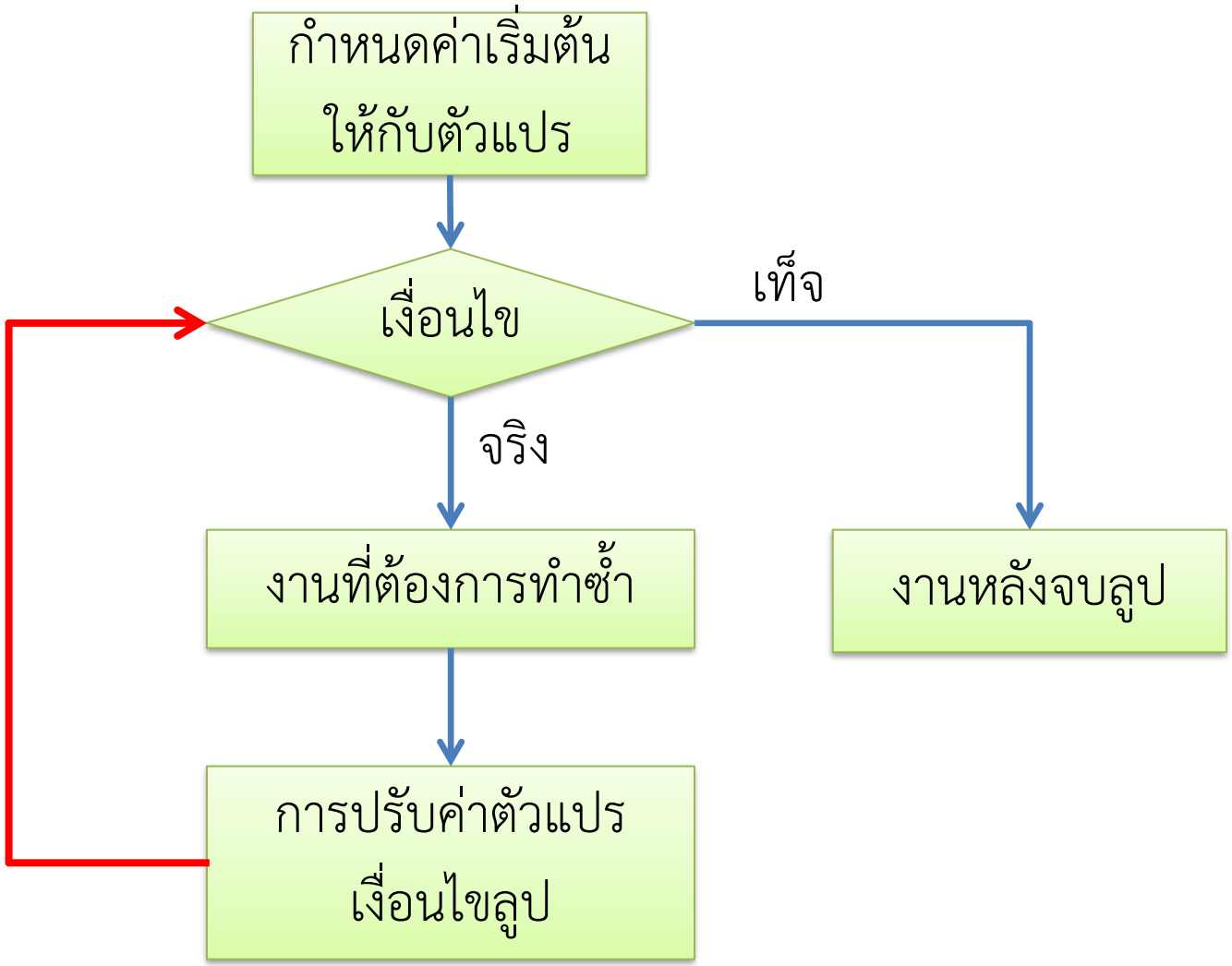
คำสั่ง for ( )


กำหนดค่าเริ่มต้น
ให้กับตัวแปร
เงื่อนไข
เท็จ
จริง
งานที่ต้องการทำซ้ำ
งานหลังจบลูป
การปรับค่าตัวแปร
เงื่อนไขลูป

การทำซ้ำด้วย for Loop

1. มีลักษณะเทียบเท่ากับ While Loop ทุกประการ
2. แต่งานที่มันถนัดก็คือการวนทำซ้ำจากค่า i = 0 ถึง n
   - แบบนี้ for loop จะได้โค้ดที่กะทัดรัด ดูดีกว่า
3. ถ้าเป็นงานแบบอื่นคิดด้วย for loop แล้วอาจจะชวนงงกว่า while loop
   เช่น ถ้าเงื่อนไขการจบลูปคือ x < 0 เป็นต้น การใช้ for loop อาจจะไม่ช่วยอะไรให้ดีขึ้น แถมชวนงงน่าสงสัยด้วย
   - คำสั่งทั้งสองทดแทนกันได้เสมอ แต่การเลือกใช้ให้เข้ากับปัญหาจะนำไปสู่โค้ดที่ดูเป็นธรรมชาติและเข้าใจง่ายกว่า
4. for loop มีลูกเล่นชวนงงมากกว่า แต่มักให้โค้ดที่สั้นกว่า
   (โค้ดที่สั้นกว่า ไม่ได้หมายความว่าดีกว่า)

แนวคิดลูปแบบ while ( ) { ... } และ for ( ) { ... }



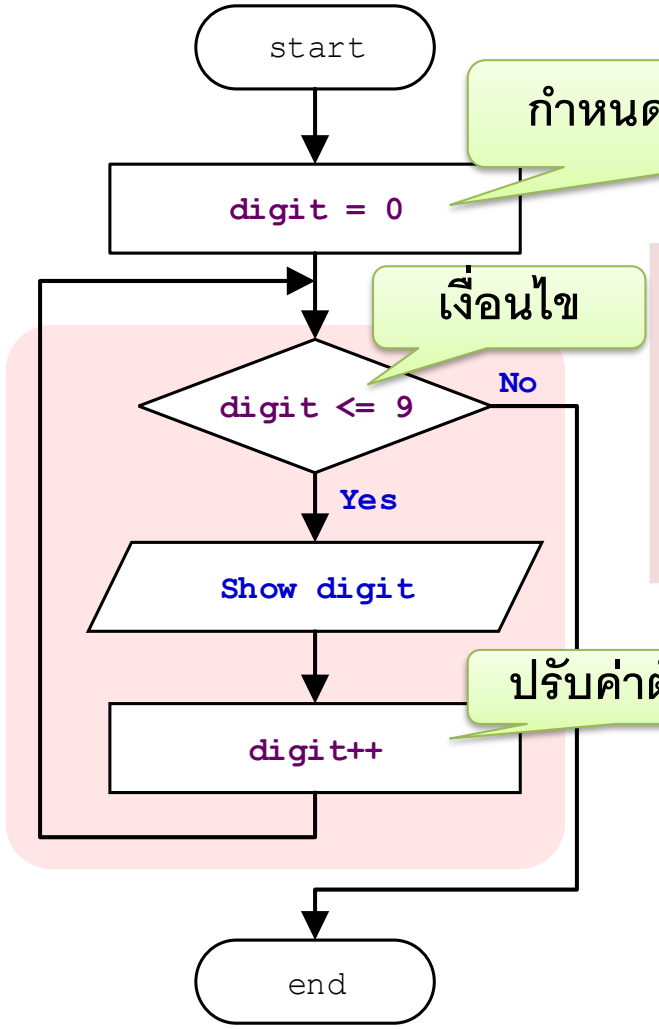
องค์ประกอบของ for loop ในภาษาซี

มีอยู่สี่ส่วนเหมือน while loop เพียงแต่มีการจัดเรียงตำแหน่งที่ต่างกัน

```
กำหนดค่าเริ่มต้นตัวแปรก่อนเข้าลูป
while ( เงื่อนไข ) {
    งานที่ต้องการทำซ้ำ
    ปรับค่าตัวแปรเงื่อนไขลูป
}
... งานหลังจบลูป ...
```

```
for (กำหนดค่าเริ่มต้นตัวแปรก่อนเข้าลูป; เงื่อนไข; ปรับค่าตัวแปรเงื่อนไขลูป) {
    งานที่ต้องการทำซ้ำ
}
... งานหลังจบลูป ...
```

คำสั่ง for



กำหนดค่าเริ่มต้นตัวแปร

เงื่อนไข

ปรับค่าตัวแปร

```
void main()
{       int digit;
        for(digit=0; digit<=9; digit++)
            printf("%d", digit);

}
```

ผลลัพธ์ที่ได้คือ ...

ตัวอย่างการทำงานของลูป

โจทย์ จงเขียนโค้ดภาษาซีสำหรับการหาผลบวกของเลขจำนวนเต็มที่มีค่าอยู่ในช่วงปิด 1 ถึง 5 (ช่วงปิดจะรวมเลข 1 และ 5 ด้วย) จากนั้นพิมพ์ผลลัพธ์ออกมาทางจอภาพ ให้เขียนด้วยการใช้ while loop และ for loop

วิเคราะห์

1. ไม่มีการรับข้อมูลเข้าจากผู้ใช้ แต่จะต้องสร้างตัวเลขขึ้นมาเอง
2. <u>งานที่ต้อง**ทำซ้ำ**แน่ ๆ คือการบวกเลข</u>
3. ต้องมีการนับเลขที่จะบวกเพิ่มขึ้นเรื่อย ๆ เพื่อให้เปลี่ยนตัวบวกจาก 1 ไปเป็น 2, 3, 4 และ 5 ได้
4. เงื่อนไขที่ควรใช้ในการทำงานคือ ‘ตัวบวกต้องอยู่ในช่วง 1 ถึง 5’

โค้ดที่ใช้ While Loop

```
void main() {
    int sum = 0;
    int adder = 1;

    while (adder <= 5) {
        sum = sum + adder;
        adder = adder + 1;
    }

    printf("%d", sum);
}
```

การกำหนดค่าตัวแปรเริ่มต้นก่อนเข้าลูป

เงื่อนไขที่จะให้ทำลูป

งานที่ต้องการทำซ้ำ

การปรับค่าตัวแปรเงื่อนไขลูป

งานหลังจบลูป

โค้ดที่ใช้ for Loop แบบที่ 1

```
void main() {
    int sum = 0;

    for(int adder = 1; adder <= 5; ++adder) {
        sum = sum + adder;
    }

    printf("%d", sum);
}
```

การกำหนดค่าตัวแปรเริ่มต้นก่อนเข้าลูป

เงื่อนไขที่จะให้ทำลูป

การปรับค่าตัวแปรเงื่อนไขลูป

งานที่ต้องการทำซ้ำ

งานหลังจบลูป

การใช้ for loop จะทำให้โค้ดดูสั้นลง เพราะงานสามอย่างจะกระจุกอยู่ที่ตอนต้นของลูป คนที่เริ่มเรียนอาจจะงงได้ว่าคำสั่งแต่ละอันถูกทำตอนไหน

ลำดับการทำงานของคำสั่งใน for loop

```
int sum = 0;
int adder;
for(adder = 1; adder <= 5; ++adder) {
    sum = sum + adder;
}
...
```

(1) กำหนดค่าตัวแปรเริ่มต้นก่อนเข้าลูป

(2) เงื่อนไขที่จะให้ทำลูป

(4) การปรับค่าตัวแปรเงื่อนไขลูป

(3) งานที่ต้องการทำซ้ำ

ลำดับการเขียนส่วนต่าง ๆ จะเปลี่ยนไป แต่ลำดับการทำงานจะคงเดิม นั่นคือ

(1) → (2) → (3) → (4) → (2) → (3) → (4) → ..... (2) → (3) → (4) → (2) → จบลูป

สังเกตให้ดีว่าส่วนที่ (1) จะถูกทำแค่ครั้งเดียว และส่วนที่ (2) เป็นส่วนสุดท้าย

<u>แต่ส่วนที่ (4) ถูกยกไปอยู่ตรงต้นขั้วลูป</u>

โค้ดที่ใช้ for Loop แบบที่ 2

- หลายคนอาจเห็นว่า ในตัวอย่างที่ยกมา มีตัวแปรที่เกี่ยวกับลูปมากกว่า 1 ตัว เราจะทำการกำหนดค่าตัวแปรเริ่มต้นหลาย ๆ ตัวใน for loop ได้อย่างไร
- เรื่องนี้สบายมาก ให้เราคั่นคำสั่งกำหนดค่าเริ่มต้นในกลุ่ม (1) ด้วยคอมมา แทนที่จะเป็นเซมิโคลอนก็เป็นอันเสร็จพิธี

```
void main() {
    int sum, adder;
    for(sum = 0, adder = 1; adder <= 5; ++adder) {
        sum = sum + adder;
    }
    printf("%d", sum);
}
```

(1) กำหนดค่าตัวแปรเริ่มต้นก่อนเข้าลูป

(2) เงื่อนไขที่จะให้ทำลูป

(4) การปรับค่าตัวแปรเงื่อนไขลูป

(3) งานที่ต้องการทำซ้ำ

คำสั่ง for

```
void main()
{       int digit;
        for(digit=0; digit<10; digit++)
        {
            printf("%d", digit);
            if(digit==4)
              printf("#");
        }
 }
```

ผลลัพธ์ที่ได้คือ ...

คำสั่งหยุดลูป (break)

บางครั้งเราต้องการออกจากลูปจากช่วงตรงกลาง แทนที่จะการออกจากลูปจากการตรวจเงื่อนไขตอนต้นลูป เช่น หากเราต้องการให้ผู้ใช้ใส่เลขจำนวนเต็มบวกมา 10 ค่าเพื่อให้โปรแกรมหาผลบวกของค่าทั้ง 10 แต่ถ้าผู้ใช้เผลอใส่เลขศูนย์หรือติดลบมา ถือว่าผิดพลาด และโปรแกรมจะหยุดรับค่าทันที

- จากตัวอย่างข้างต้นแสดงว่าเงื่อนไขที่จะหยุดลูปมีสองอย่างคือ
  (1) ผู้ใช้ใส่ค่าครบ 10 จำนวน และ (2) ผู้ใช้ใส่เลขศูนย์หรือค่าติดลบมา
- เงื่อนไขทั้งสองเป็นอิสระจากกัน ไม่ควรเอามาคิดรวมกันตรงต้นลูปพร้อมกัน
- เงื่อนไขที่เป็นอิสระแบบนี้ถ้าจะนำมารวมกันมันจะซับซ้อนมาก
  → การใช้คำสั่ง break; เพื่อหยุดลูปจะทำให้ตรรกะในการคำนวณง่ายขึ้น
- คำสั่ง break; จะทำให้ลูปที่มันอยู่ข้างในจบการทำงานทันที

คำสั่ง break

ถ้าใช้คำสั่ง break ใน loop ใดๆ แล้ว จะทำให้ออกจาก loop ทันที

```
for(i=0;i<6;i++)
{     if(i==3)
            break;
      printf("%d ", i);
 }
```

ผลลัพธ์ที่ได้คือ

การใช้คำสั่ง break;

- คำสั่ง break; โดยตัวของมันเองเป็นการออกจากลูปอย่างไม่มีเงื่อนไข
- ถ้าเราใช้มันตรง ๆ มันก็จะหยุดลูปทุกครั้งไป ลูปจะไม่มีการวนซ้ำเพราะโดนคำสั่ง break; สั่งหยุดทำงานทุกครั้ง เช่น

```
while (i < 10)  {
    scanf("%d", &x);
    break;
    sum = sum + x;
    i++;
}
```

ถ้าทำแบบนี้พอเจอคำสั่ง break ลูปจะหยุดทันที คำสั่งหลัง break จะไม่มีโอกาสได้ทำงานเลย

จะเห็นได้ว่าคำสั่ง break; อยู่ในลูปโดยไม่อยู่ใต้เงื่อนไขของ if ดังนั้นโปรแกรมนี้หลังจากรับค่าจากผู้ใช้มาเก็บไว้ ก็จะออกจากลูปทันที ไม่มีการวนทำซ้ำ

การทำให้คำสั่ง break; มีประโยชน์

ดังนั้นเราจึงใช้มันคู่กับเงื่อนไข if เช่นจากตัวอย่างที่ป้องกันไม่ให้ผู้ใช้ใส่เลขศูนย์และค่าติดลบ

- ถ้าข้อมูลเข้าจากผู้ใช้คือ x เราก็จะได้ตรรกะของ if ที่ควรจะเป็นคือ ‘ถ้า x <= 0 ให้โปรแกรมออกจากลูป’ และได้ลูปเป็น

```
while (i < 10)  {
  scanf("%d", &x);
  if (x <= 0) {
    break;
  }
  sum = sum + x;
  i++;
}
```

ตัวอย่างการใช้ break;

โจทย์ จงเขียนโปรแกรมที่รับจำนวนเต็มบวกจากผู้ใช้ได้มากถึง 10 จำนวน และหาผลบวกของเลข 10 จำนวนดังกล่าว แต่หากผู้ใช้ใส่เลขศูนย์หรือติดลบเข้ามาโปรแกรมจะไม่นำค่าดังกล่าวไปบวกกับตัวเลขอื่น ๆ ก่อนหน้า นอกจากนี้โปรแกรมจะหยุดรับค่าจากผู้ใช้ และก่อนจบโปรแกรมจะพิมพ์ข้อความว่า Error แต่หากผู้ใช้ใส่จำนวนเต็มบวกมาทั้ง 10 จำนวน โปรแกรมจะพิมพ์ผลบวกของเลขทั้ง 10 ออกมา

(เช่นเดิม โจทย์ข้อนี้ดูเหมือนไม่มีอะไร แต่มือใหม่ต้องใช้เวลาคิดนานพอสมควร)

วิเคราะห์ปัญหาการหยุดลูป

- จุดยากของปัญหาอยู่ที่ว่าทำอย่างไรตอนที่โปรแกรมออกจากลูปแล้วจะแยกได้ว่าจะพิมพ์ผลบวกหรือคำว่า Error ดี
- วิธีที่ได้ผลดีในข้อนี้คือให้ตรวจว่าค่าตัวเลขที่ได้จากผู้ใช้อันล่าสุดคือค่าบวกหรือว่าเป็นอย่างอื่น
- วิธีอีกอันหนึ่งที่ได้ผลดีก็คือการตรวจว่าลูปวนไปจนครบสมบูรณ์กี่รอบ ถ้าครบสมบูรณ์ดีทั้ง 10 รอบก็แสดงว่าเราควรแสดงผลบวกออกมา

วิธีตรวจความเป็นค่าบวกของเลขสุดท้าย

```
int x;
int sum = 0;
int i = 0;
while(i < 10) {
    scanf("%d", &x);
    if(x <= 0)
        break;
    sum = sum + x;
    i++;
}
if(x <= 0) {
    printf("Error");
} else {
    printf("%d", sum);
}
```

ถ้าถูก break ตรงนี้แสดงว่าลูปจะจบลง โดยที่ค่า x ไม่เป็นบวก

ดังนั้นถ้าเราตรวจตรงนี้ได้ว่า x เป็นศูนย์ หรือติดลบก็สรุปได้เลยว่าผู้ใช้ใส่ค่าผิด

วิธีตรวจจำนวนรอบที่สมบูรณ์

```
int x;
int sum = 0;
int i = 0;
while(i < 10) {
    scanf("%d", &x);
    if(x <= 0)
        break;
    sum = sum + x;
    i++;
}
if(i < 10) {
    printf("Error");
} else {
    printf("%d", sum);
}
```

ถ้าถูก break ตรงนี้แสดงว่าลูปจะจบลงก่อนได้ทำ i++ ทางด้านใต้ ส่งผลให้ i < 10 ในขณะที่การจบลูปแบบปกติจะได้ค่า i == 10

ดังนั้นถ้าเราตรวจตรงนี้ได้ว่า x น้อยกว่า 10 ก็สรุปได้เลยว่าผู้ใช้ใส่ค่าผิด

คำสั่งวกกลับไปต้นลูป (continue)

- บางครั้งจุดที่เราอยากให้โปรแกรมวกกลับไปต้นลูปอาจจะไม่ใช่แค่ตรงด้านท้ายของลูปเท่านั้น เราอาจจะอยากให้มีการวกกลับที่จุดอื่น ๆ ด้วย
- แม้จะเป็นไปได้ที่เราจะแก้ปัญหานี้ผ่านการใช้ if-else ที่ซับซ้อนขึ้น แต่การใช้คำสั่ง continue; เพื่อสั่งให้โปรแกรมวกกลับไปด้านบนของลูปจะทำให้ตรรกะในการคิดดูง่ายขึ้น
- คำสั่ง continue; ไม่ใช่สิ่งที่จำเป็นอย่างยิ่ง แต่มันทำให้เรามีอิสระในการวางแผนการคิดในการเขียนโปรแกรมมากขึ้น จึงควรเรียนรู้ไว้
- เช่นเดียวกับ break; การใช้ continue; ที่มีประโยชน์ ต้องใช้คู่กับเงื่อนไขของ if ไม่เช่นนั้นลูปจะวนกลับไปด้านบนทุกครั้ง ไม่มีทางไปถึงคำสั่งที่อยู่หลังจากมัน

ตัวอย่างการวนรับค่าไม่จำกัด

โจทย์ จงเขียนโปรแกรมที่รับค่าตัวเลขจำนวนเต็มจากผู้ใช้เข้ามาเรื่อย ๆ โปรแกรมจะทำการนับและบวกเลขที่เป็นบวก แต่หากผู้ใช้ใส่เลขที่เป็นลบหรือศูนย์เข้ามา โปรแกรมจะหยุดรับค่าจากผู้ใช้ แล้วพิมพ์จำนวนตัวเลขค่าบวกที่รับมาทั้งหมด รวมทั้งผลรวมของเลขบวกเหล่านี้

วิเคราะห์ ที่ผ่านมาเรามักจะหยุดลูปเมื่อผู้ใช้ใส่ตัวเลขเข้ามาถึงจำนวนหนึ่ง แต่ในปัญหานี้ ผู้ใช้สามารถใส่ตัวเลขเข้ามาได้ไม่จำกัด ดังนั้นการตั้งเงื่อนไขลูปโดยการจำกัดจำนวนครั้งไว้จึงเป็นเรื่องที่ผิด เพราะแท้จริงผู้ใช้จะใส่เลขเข้ามากี่ตัวก็ได้

1. เงื่อนไขที่จะใช้หยุดลูปจึงควรผูกอยู่กับค่าที่ผู้ใช้ใส่เข้ามา
2. ต้องป้องกันการนับและบวกค่าที่ไม่เป็นบวก แต่ให้โปรแกรมหยุดลูปแทน

โปรแกรมหาผลรวมตัวเลขบวก แบบไม่ใช้ break;

```
void main() {
  int count = 0;
  int sum = 0;
  int x = 1;

  while(x > 0) {
    scanf("%d", &x);
    if(x > 0) {
      count++;
      sum += x;
    }
  }
  printf("%d %d", count, sum);
}
```

แบบนี้ไม่ใช้คำสั่ง break; แต่ก็คงพอจะเห็นได้ว่าเราต้องมีการทำอะไรแปลก ๆ เช่นเริ่มมาก็ให้ x = 1 ไปก่อนเลย ทั้งนี้ก็เพื่อรับประกันว่าเงื่อนไขที่กำหนดไว้จะเป็นจริงในรอบแรกของการทำงานแน่ ๆ

ถ้าผู้ใช้ใส่เลขศูนย์หรือค่าติดลบมา การนับและบวกค่าจะไม่เกิดขึ้น จากนั้นโปรแกรมจะวนกลับขึ้นไปทางด้านบน และลูปจะจบการทำงานเพราะเงื่อนไขลูปไม่เป็นจริง

โปรแกรมหาผลรวมตัวเลขบวก แบบใช้ break;

```
void main() {
  int count = 0;
  int sum = 0;
  int x;

  while(1) {
    scanf("%d", &x);
    if(x <= 0) {
      break;
    }
    count++;
    sum += x;
  }
  printf("%d %d", count, sum);
}
```

แบบนี้ใช้คำสั่ง break; เงื่อนไขลูปคือ 0 < 1 ซึ่งแปลว่า ‘จริง’ ดังนั้นลูปจะวนไปเรื่อย ๆ จนกว่าจะโดนคำสั่ง break; ที่อยู่ด้านใน รูปแบบเงื่อนไขที่นิยมกว่าสำหรับวิธีนี้คือ while (1) { … }

ถ้าผู้ใช้ใส่เลขศูนย์หรือค่าติดลบมา จะเข้าเงื่อนไขนี้ และจะเกิดการหยุดลูปขึ้น คำสั่งนับและบวกค่าด้านใต้ก็จะถูกข้ามไปด้วย

ตัวอย่างการใช้ continue;

โจทย์ จงเขียนโปรแกรมที่รับค่าจำนวนเต็มจากผู้ใช้มา 10 จำนวน หากจำนวนเต็มนั้นหารด้วย 5 ลงตัว โปรแกรมจะไม่พิมพ์ข้อความใด ๆ ออกมา และวนกลับไปเตรียมรับตัวเลขตัวต่อไปจากผู้ใช้ แต่หากไม่เป็นเช่นนั้น โปรแกรมจะพิมพ์คำว่า Accept และนับจำนวนตัวเลขแบบนี้ว่ามีกี่ตัว สุดท้ายเมื่อผู้ใช้ใส่เลขครบสิบตัว โปรแกรมจะพิมพ์จำนวนครั้งที่โปรแกรมแสดงคำว่า Accept ออกมา และจบการทำงาน

วิเคราะห์ โปรแกรมมีการวนกลับกลางทาง เราสามารถใช้คำสั่ง continue; เพื่อให้โปรแกรมวนกลับไปตรวจเงื่อนไขของลูปด้วย

โปรแกรมนับตัวเลขที่หาร 5 ไม่ลงตัว

```
int count = 0;
int i = 0;
int x;
while(i < 10) {
    scanf("%d", &x);
    if(x % 5 == 0) {
        i++;
        continue;
    }
    printf("Accept\n");
    count++;
    i++;
}
printf("%d", count);
```

เมื่อโปรแกรมทำงานมาถึงจุดนี้โปรแกรมจะตีกลับไปที่ด้านบนของลูป และตรวจเงื่อนไขของลูปอีกครั้ง

สังเกตดูให้ดีว่า เป็นไปได้ว่าโปรแกรมจะมีการปรับค่าตัวแปรเงื่อนไขสองที่ในลูปก็ได้

คำสั่ง continue

จะทำให้คำสั่งทั้งหมดที่อยู่หลังจากคำสั่ง continue ภายใน loop นั้นๆ ไม่ถูกประมวลผล โดยจะข้ามการทำงานไปตรวจสอบเงื่อนไขของลูปใหม่

```
for(i=0;i<6;i++)
{       if(i==3)
                continue;
        printf("%d ", i);
 }
```

ผลลัพธ์ที่ได้คือ

การซ้อนลูป

(Nested Loop)

การซ้อนลูปคืออะไร

การซ้อนลูปคือการกำหนดขั้นตอนการทำงานที่มีการวนซ้ำมากกว่าหนึ่งระดับ

- ถ้ามีสองระดับเรามักเรียกว่าลูปสองชั้น

```
for(int j = 0; j < M; ++j) {

    for(int i = 0; i < N; ++i) {
    printf("%d %d", j, i);
   }

}
```

ลูปชั้นนอก

ลูปชั้นใน

- ถ้ามีสามระดับเรามักเรียกว่าลูปสามชั้น
- โดยปรกติจะไม่ค่อยเจอลูปที่มากกว่าสามชั้นโดยตรง เพราะผู้เขียนโปรแกรมจะเลี่ยงไปใช้ฟังก์ชันประกอบเพื่อให้โค้ดในโปรแกรมเข้าใจง่ายขึ้น

ลองใช้ลูปสองชั้นกับข้อมูลในรูปแบบตาราง

สมมติว่าตารางของเรามีจำนวน 5 แถว และ 6 คอลัมน์

เราต้องการพิมพ์หมายเลขแถวและคอลัมน์ของตารางลงไปในแบบข้างล่าง

```
(1, 1) (1, 2) (1, 3) (1, 4) (1, 5) (1, 6)
(2, 1) (2, 2) (2, 3) (2, 4) (2, 5) (2, 6)
(3, 1) (3, 2) (3, 3) (3, 4) (3, 5) (3, 6)
(4, 1) (4, 2) (4, 3) (4, 4) (4, 5) (4, 6)
(5, 1) (5, 2) (5, 3) (5, 4) (5, 5) (5, 6)
```

นั่นคือเราต้องการแสดงตำแหน่งออกมาเป็นคู่ลำดับ โดยแสดงตำแหน่งแถวออกมาก่อนตำแหน่งคอลัมน์ เมื่อจบแต่ละแถวเราก็สั่งขึ้นบรรทัดใหม่ แล้วพิมพ์คู่ลำดับออกมาในลักษณะเดิม

โค้ดแสดงคู่ลำดับของหมายเลขแถวและคอลัมน์

```
#include <stdio.h>

void main() {
  for(int row = 1; row <= 5; ++row) {
    for(int col = 1; col <= 6; ++col) {
      printf("(%d, %d) ", row, col);
    }
    printf("\n");
  }
}
```

เนื่องจากผลลัพธ์ถูกจัดการตามแถวก่อน เราจึงเอาแถวออกมาเป็นลูปด้านนอก ส่วนคอลัมน์เป็นลูปด้านใน

ลูปด้านในพิมพ์ข้อความออกมา ขอให้เข้าใจด้วยว่าลูปด้านในจะต้องวนจนครบ 6 รอบก่อน มันถึงจะหลุดออกมา และใน 6 รอบนี้ค่า row จะเหมือนเดิมเพราะลูปด้านนอกไม่ถูกแตะต้อง ค่า row จึงไม่เปลี่ยน

พิมพ์ขึ้นบรรทัดใหม่หลังจากพิมพ์คอลัมน์จนครบ (จัดเป็นส่วนของลูปด้านนอก)

ลำดับการทำงานของลูป 2 ชั้น

- อันที่จริง ลำดับการทำงานของลูป 2 ชั้นนั้นก็เป็นไปตามแนวคิดของลูปชั้นเดียวทุกประการ เพียงแต่ผู้เรียนบางท่านยังไม่คล่องเรื่องลูปชั้นเดียว → ทำให้งงหนักยิ่งกว่าเดิม และเราควรกลับมาดูที่พื้นฐานตรงนี้ก่อน
- ขอใช้ลูปด้านในจากตัวอย่างที่แล้วเป็นกรณีศึกษา โดยกำหนดให้ row = 1 เป็นค่าคงที่ตายตัวไว้ก่อน

```
int row = 1;
for(int col = 1; col <= 6; ++col) {
  printf("(%d, %d) ", row, col);
}
printf("\n");
```

- ตอนนี้เราคงเห็นได้ชัดเจนขึ้นว่าทำไมตอนที่วนลูปด้านในแล้วค่าตัวเลขคอลัมน์จึงเปลี่ยนไปเรื่อย ๆ แต่ตัวเลขแสดงแถวเป็นค่าคงที่ตลอด

ถ้าทำแบบเดิมซ้ำ ๆ ต่อกันไปล่ะ

ถ้าเราเขียนลูปแบบเดิมซ้ำ แต่เปลี่ยนค่า row ไปด้วย เราก็จะได้ผลลัพธ์ออกมาสองแถว โดยมีเลขแถวเปลี่ยนไป ส่วนเลขคอลัมน์จะมีการวนเปลี่ยนแปลงในลักษณะเดิม

```
int row = 1;
for(int col = 1; col <= 6; ++col) {
  printf("(%d, %d) ", row, col);
}
printf("\n");
row = 2;
for(int col = 1; col <= 6; ++col) {
  printf("(%d, %d) ", row, col);
}
printf("\n");
```

ถ้าเราเอาลูปด้านนอกมาครอบ ผลก็คือเวลาที่มันทำลูปด้านในเสร็จ มันจะออกมาที่ printf(“\n”); เสร็จแล้วก็ตีกลับขึ้นไปเปลี่ยนค่า row ใหม่ วนเช่นนี้ไปเรื่อย ๆ จนกว่าเงื่อนไขลูปด้านนอกจะไม่เป็นจริง

```
for(int row = 1; row <= 5; ++row) {


    for(int col = 1; col <= 6; ++col) {
        printf("(%d, %d) ", row, col);
    }
   printf("\n");


}
```

ลองตอบคำถามนี้ดู

- โปรแกรมนี้จะพิมพ์เลขอะไรออกมาบ้าง

```
#include <stdio.h>

void main() {
    for(int i = 0; i < 7; ++i) {
        for(int j = 0; j < 3; ++j) {
            printf("%d ", i + j);
        }
    }
}
```

- คำตอบ

พิมพ์ตัวเลขขั้นบันได

- เพื่อที่จะเรียนรู้แนวคิดลูปสองชั้น เรามาดูตัวอย่างเพิ่มเติม
- สมมติว่าผู้ใช้ใส่เลข 5 เข้ามาแล้วเราต้องการพิมพ์ว่า

```
1
1  2
1  2  3
1  2  3  4
1  2  3  4  5
```

- ถ้าผู้ใช้ใส่เลข 10 เข้ามา โปรแกรมก็ต้องไล่ไปจนถึง 10 ถ้าใส่จำนวนเต็มบวก N เข้ามา ก็ต้องไล่ไปเรื่อย ๆ จนถึง N
- แสดงว่าต้องมี N แถว ส่วนจำนวนคอลัมน์ก็ตรงกับหมายเลขแถวที่โปรแกรมกำลังพิมพ์นั่นเอง

พิจารณาการพิมพ์ในแต่ละแถว

- ลองกำหนดเลขแถวตายตัวไว้ที่ int row = 4; ก่อน

  → แสดงว่าเราจะพิมพ์เลข 1 2 3 4

- โค้ดที่ได้ก็จะมีหน้าตาทำนองนี้

```
int row = 4;
for(int col = 1; col <= row; ++col) {
    printf("%d ", col);
}
printf("\n");
```

- ที่เหลือก็คือเราต้องเอาลูปอีกชั้นมาครอบเพื่อให้มันเปลี่ยนเลขแถวได้อย่างที่ควรจะเป็น

  (ลองคิดดูด้วยตัวเองให้ดีก่อนว่าควรจะเขียนลูปด้านนอกอย่างไร)

เอาลูปด้านนอกมาแปะเพื่อเปลี่ยนค่า row

- จุดหลักคือเปลี่ยนค่า row ให้ได้อย่างที่ควรเป็น ถ้าทำได้ตัวเลขที่ถูกพิมพ์ออกมาในแต่ละแถวก็จะออกมาอย่างที่ควรเป็น (เพราะเลขแถวมันถูกนั่นเอง)

```
int N;
scanf("%d", &N);
for(int row = 1; row <= N; ++row) {
    for(int col = 1; col <= row; ++col) {
        printf("%d ", col);
    }
    printf("\n");
}
```

- สังเกตหรือไม่ว่าจริง ๆ แล้วการคิดมาจากลูปด้านในก่อนเป็นเรื่องธรรมดา
- สำหรับคนที่ยังไม่คล่องควรคิดมาตามแนวทางนี้ แบบที่คิดรวดเดียวจบมันทำได้เฉพาะคนที่เข้าใจและเริ่มชำนาญกับเรื่องลูปสองชั้นแล้ว

สรุปใจความ

- งานที่ใช้ลูปสองชั้นมีลำดับการคิดคล้ายกับลูปชั้นเดียว
  - พอมันวิ่งเข้าไปที่ลูปด้านใน มันก็ต้องทำของข้างในให้เรียบร้อยหมดก่อน จึงค่อยหลุดลงมาต่อด้านท้าย และวนขึ้นไปต่อที่ลูปด้านนอกได้
  - เวลาที่จินตนาการไม่ออกให้มองว่าลูปด้านในเป็นเหมือนงานบรรทัดหนึ่งที่ทำเสร็จแล้วก็ข้ามไปทำบรรทัดถัดไป
  - ด้วยเหตุนี้เราจึงอาจจะลองเริ่มคิดจากเนื้อหาของลูปด้านในให้เสร็จก่อน แล้วค่อยเอาลูปด้านนอกมาครอบอีกชั้น
- ถ้างานมันมีลักษณะแบ่งเป็นหลาย ๆ แบบ ลูปของเราจะมีหลาย ๆ ชุดก็สมเหตุผลดี หรือถ้าจะมีชุดเดียวใหญ่ ๆ ก็อาจจะต้องพึ่งพาการใช้ if จำนวนมากเพื่อแยกประเภทงาน (วิธีหลังนี้อาจจะทำให้ยุ่งกว่าเดิม)