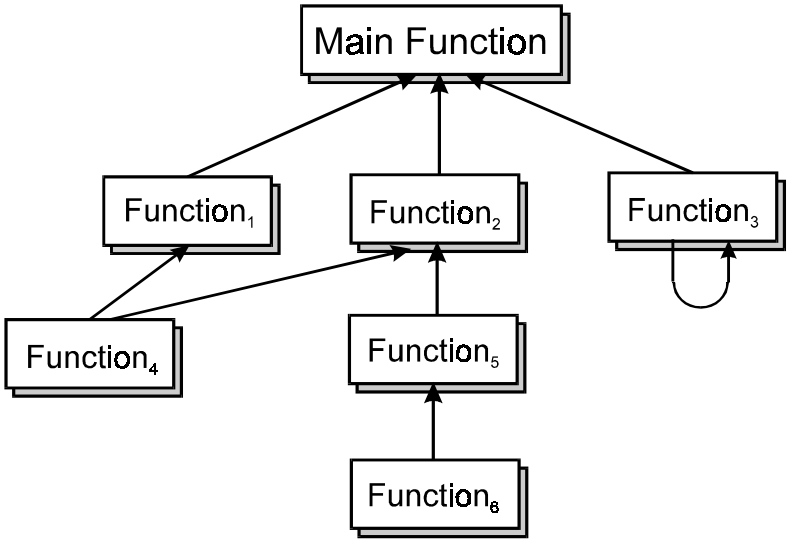
4

ฟังก์ชัน

ในหลายๆบทที่ผ่านมาเราได้เรียนรู้เกี่ยวกับฟังก์ชันในภาษาซีรวมทั้งได้ลองใช้ฟังก์ชันมาตราฐานไปบ้างแล้วพอสมควร ในบทนี้เราจะมาทำความเข้าใจในเรื่องการทำงานของฟังก์ชัน รวมถึงรายละเอียดปลีกย่อยต่างๆ เช่น การสร้างและเรียกใช้ รวมถึงองค์ประกอบต่างๆของฟังก์ชัน เพื่อที่จะสามารถเขียนฟังก์ชันขึ้นใช้เพื่อจุดประสงค์ใดๆได้ด้วยตนเองอย่างถูกต้องและมีประสิทธิภาพ

โปรแกรมในภาษาซีจะประกอบด้วยฟังก์ชันหลักและฟังก์ชันอื่นๆโดยแบ่งออกเป็นหลายระดับตามการเรียกใช้ฟังก์ชัน โปรดพิจารณาภาพประกอบที่ 4.1 จุดเริ่มต้นของโปรแกรมคือ *ฟังก์ชันหลัก*หรือ `main()` ภายในฟังก์ชันหลักก็อาจจะมีการเรียกใช้ฟังก์ชันอื่นๆอีกหลายฟังก์ชัน และภายในส่วนตัวของฟังก์ชันแต่ละฟังก์ชันก็อาจจะมีการเรียกใช้ฟังก์ชันตัวอื่นๆหรือที่เรียกว่า *ฟังก์ชันย่อย* (Subroutine) อีกตามลำดับชั้นลงไป หรือเป็นฟังก์ชันที่เรียกใช้ตัวเองก็ได้ ซึ่งกรณีหลังนี้เรียกว่า *ฟังก์ชันเรียกใช้ตัวเอง* (เราเรียกการทำงานในลักษณะนี้ว่า Recursion)

ฟังก์ชันแต่ละฟังก์ชันติดต่อสื่อสารกันโดยการเรียกใช้ (Function Call) ซึ่งแบ่งออกเป็นสองฝ่ายคือ ผู้เรียกใช้และผู้ถูกเรียกใช้ และอาศัยการผ่านข้อมูลต่างๆระหว่างฟังก์ชัน โดยมีการกำหนดรูปแบบและชนิดของข้อมูลที่ส่งผ่านไปมาอย่างชัดเจน บางฟังก์ชันไม่จำเป็นต้องได้รับข้อมูลใดๆจากผู้เรียกใช้ (Caller) ในขณะที่บางฟังก์ชันก็ต้องการข้อมูลจากผู้เรียกใช้ ซึ่งมีชนิดและจำนวนของพารามิเตอร์แตกต่างกันออกไปขึ้นอยู่กับรูปแบบของฟังก์ชันที่เรากำหนด



ภาพประกอบที่ 4.1 ตัวอย่างโปรแกรมที่มีการเรียกใช้ฟังก์ชัน

การเขียนแผนการทำงานของโปรแกรมในหน่วยของฟังก์ชันตามภาพประกอบที่ 4.1 เป็นการออกแบบการทำงานของโปรแกรมจากบนลงล่าง หรือที่เรามักจะเรียกว่า Top-down Modular Design การออกแบบโปรแกรมและฟังก์ชันในลักษณะนี้ต้องมีการกำหนดว่า ฟังก์ชันใดมีหน้าที่ใด แต่ละฟังก์ชันเราอาจจะมองว่าเป็นเสมือนกับกล่องดำ (Black Box) โดยเราไม่สนใจว่าภายในหรือ *ส่วนตัวของฟังก์ชัน* (Function Body) จะเป็นอย่างไร เราสนใจในตอนแรกเพียงแต่ว่ากล่องดำหรือฟังก์ชันนี้มีหน้าที่ใด ต้องการข้อมูล แบบใดที่ใช้เป็นอินพุต (Input) และเอาท์พุต (Output) รวมเรียกว่า *รายละเอียดของฟังก์ชัน* (Specification) ดังนั้นเราจำเป็นต้องมีรูปแบบมาตราฐานเพื่อเป็นการกำหนดรายละเอียดของฟังก์ชัน และเราเรียกส่วนนี้ว่า *ส่วนหัวของฟังก์ชัน*

4.1 องค์ประกอบของฟังก์ชัน

องค์ประกอบสำคัญในโครงสร้างของฟังก์ชันเราสามารถจำแนกออกอย่างคร่าวๆตามลักษณะการทำงานของฟังก์ชันได้ดังนี้คือ ฟังก์ชันที่ให้ค่ากลับคืนและฟังก์ชันที่ไม่ให้ค่าใดๆกลับคืน ในภาษาปาสคาล ฟังก์ชันที่ไม่ให้ค่าใดๆกลับคืนจะเรียกว่า โพรซีเจอร์ (Procedure) ดังนั้น ถ้าเรานำคำนี้มาใช้ในภาษาซีก็คงจะไม่ผิดอะไร

รูปแบบของฟังก์ชันที่ให้ค่า(ข้อมูล)ใดๆกลับคืน

```
return_type  function_name ( parameter_list )
{
     local_variable_declarations

     function_statements

     return return_value;
}
```

รูปแบบของฟังก์ชันที่ไม่ให้ค่า(ข้อมูล)ใดกลับคืน (Void Function)

```
void function_name ( parameter_list )
{
     local_variable_declarations

     function_statements
}
```

คำว่า *return_type* หมายถึง แบบข้อมูลกลับคืนของฟังก์ชัน ซึ่งเป็นแบบข้อมูลใดๆ อาจจะเป็นแบบข้อมูลพื้นฐานหรือซับซ้อนก็ได้ ดังนั้นนิพจน์ *return_value* จะต้องให้ค่าตามแบบข้อมูลกลับคืนที่กำหนดไว้ ในกรณีที่ฟังก์ชันไม่ให้ค่าใดกลับคืนเราก็กำหนดให้แบบข้อมูลกลับคืนเป็น void คำว่า *function_name* หมายถึงชื่อของฟังก์ชันซึ่งเป็นตัวระบุที่ถูกต้องตามหลักไวยกรณ์ของภาษาซี ตามด้วยเครื่องหมายวงเล็บเปิดและปิดเป็นคู่ และระหว่างเครื่องหมายวงเล็บทั้งสองนี้ก็คือ *parameter_list* ซึ่งเป็นรายการของพารามิเตอร์ที่ฟังก์ชัน ต้องการ มีรูปแบบดังต่อไปนี้

$$(data\_type_1\ variable_1, \ldots, data\_type_N\ variable_N)$$

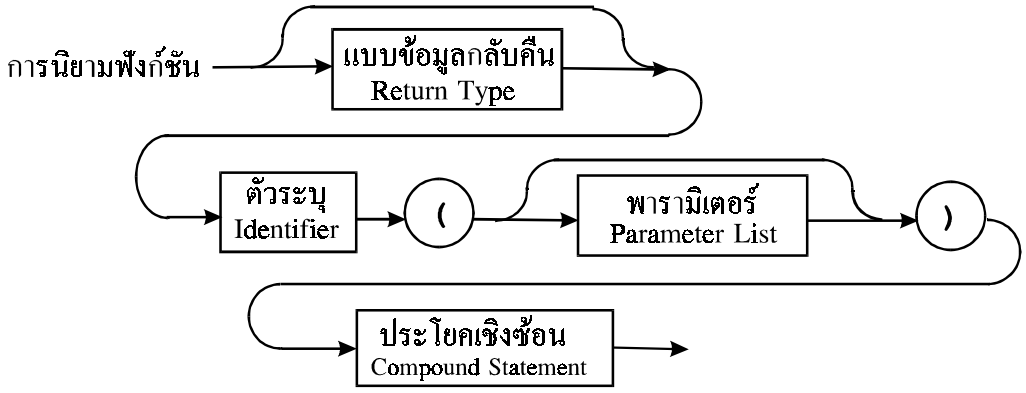
เริ่มต้นด้วยแบบข้อมูลและตามด้วยชื่อของตัวแปรที่ใช้เป็นพารามิเตอร์ ถ้ามีพารามิเตอร์มากกว่าหนึ่งตัวเราก็เขียนแยกกันโดยใช้เครื่องหมายจุดลูกน้ำ (,) ถ้าฟังก์ชันไม่ต้องการพารามิเตอร์ใดๆ เราก็เว้นว่างไว้ หรือ อาจจะใช้คำว่า void เติมไว้ระหว่างเครื่องหมายวงเล็บก็ได้เป็นการเจาะจงลงไปว่า ฟังก์ชันนี้ไม่มีและบังคับไม่ให้มีพารามิเตอร์ใดๆ

1) ส่วนหัวของฟังก์ชัน

ส่วนนี้เป็นการกำหนดรายละเอียดของฟังก์ชันประกอบด้วยชื่อของฟังก์ชันหรือตัวระบุพารามิเตอร์ต่างๆ และแบบข้อมูลกลับคืนสำหรับกรณีที่ฟังก์ชันให้ค่าใดๆกลับคืนไปยังฟังก์ชันอื่นที่มีคำสั่งเรียกใช้ฟังก์ชันตัวนี้ ส่วนหัวของฟังก์ชันนี้ทำหน้าที่เป็นตัวเชื่อมต่อ (Interface) ระหว่างภายในฟังก์ชันและ“โลกภายนอก”

2) ส่วนตัวของฟังก์ชัน

ส่วนที่สองนี้ เป็นชุดของคำสั่งที่ทำงานตามหน้าที่ของฟังก์ชัน จัดเป็นประโยคเชิงซ้อน โดยเริ่มต้นด้วย { และจบด้วย } ซึ่งแบ่งออกได้เป็นสองส่วน ส่วนแรก(ถ้ามี)คือการแจ้งใช้ตัวแปรภายในหรือตัวแปรเฉพาะที่สำหรับใช้ภายในฟังก์ชัน ตัวแปรชนิดนี้จะมีตัวตนก็ต่อเมื่อฟังก์ชันนี้ทำงานเท่านั้น ซึ่งก็คือช่วงเวลาระหว่างที่ผู้เรียกใช้ได้เรียกใช้ฟังก์ชันและฟังก์ชันจบการทำงาน เรากล่าวสรุปได้ว่า ตัวแปรเฉพาะที่จะถูกสร้างขึ้นและใช้งานได้ก็ต่อเมื่อฟังก์ชันยังคงแอคทีฟอยู่จนกว่าฟังก์ชันจะจบการทำงาน ส่วนที่สองคือชุดคำสั่งต่างๆซึ่งเป็นลำดับของประโยคคำสั่งที่เราต้องการใช้ในการทำงานของฟังก์ชัน



ภาพประกอบที่ 4.2 โครงสร้างของฟังก์ชัน

ตัวอย่างการนิยามฟังก์ชัน เช่น

```
double function_one (double x, double y)
{
      int result;

      result = (x + y)*(x - y);
      return result;
}
```

ในส่วนหัวของฟังก์ชัน เราอาจจะเขียนเฉพาะชื่อตัวแปรและไม่ต้องมีแบบข้อมูลนำหน้าชื่อของตัวแปรที่เป็นพารามิเตอร์เหล่านี้ก็ได้ แต่เราจะต้องเขียนพารามิเตอร์และแบบข้อมูลอีกครั้งในบรรทัดถัดไปต่อท้ายเครื่องหมายวงเล็บปิด ตามตัวอย่างต่อไปนี้

```
double function_one (x, y)
double x;
double y;
```

```
{
      int result;

      result = (x + y)*(x - y);
      return result;
}
```

ในกรณีตัวอย่างนี้เราใช้ตัวแปรสองตัวที่เป็นข้อมูลแบบเดียวกัน ดังนั้นเราสามารถเขียนไว้ในประโยคเดียวกันได้

```
double function_one (x, y)
double x, y;
{
      int result = (x + y)*(x - y);
      return result;
}
```

รูปแบบการแจ้งใช้พารามิเตอร์ของฟังก์ชันในลักษณะนี้เป็นรูปแบบเก่าของการนิยามฟังก์ชันในภาษาซี ตามปรกติแล้วคอมไพล์เลอร์สมัยใหม่จะสนับสนุนแต่แบบแรกเท่านั้น แต่อย่างไรก็ตามบางคอมไพล์เลอร์ก็อนุญาตให้ใช้ได้ทั้งสองแบบ

ในการนิยามฟังก์ชัน ถ้าเราไม่ได้กำหนดเจาะจงแบบข้อมูลกลับคืน (Return Type) แล้วจะถือว่า ฟังก์ชันนี้ให้ค่าแบบ `int` กลับคืน ตัวอย่างเช่น เวลาเราเขียนฟังก์ชันหลักเราก็ไม่จำเป็นต้องเขียนแบบข้อมูล `int` ไว้ข้างหน้า

```
main()
{
      printf("Main Function : Hello world!\n");
      return 0;
}
```

4.1.1 ชื่อของฟังก์ชัน

ฟังก์ชันจะต้องมีชื่อเพื่อบ่งบอกถึงความแตกต่างระหว่างฟังก์ชัน การตั้งชื่อให้ฟังก์ชันก็มีกฎเกณฑ์เดียวกันกับการตั้งชื่อให้ตัวแปร เราควรจะตั้งชื่อให้มีความหมายซึ่งบอกได้ว่าฟังก์ชันนี้มีหน้าที่อะไรและไม่ควรจะยาวเกินไป ตัวอย่างเช่น เราใช้ตัวระบุ `IsPrimeNum` สำหรับชื่อของฟังก์ชันที่ใช้ในการตรวจสอบว่า ค่าพารามิเตอร์ที่เราผ่านให้ฟังก์ชันนั้นเป็นจำนวนเฉพาะหรือไม่

4.1.2 รายการพารามิเตอร์ของฟังก์ชัน

เราผ่านค่าต่างๆเพื่อใช้ภายในฟังก์ชัน พารามิเตอร์ (หรือเรียกว่า ฟังก์ชันอาร์กิวเมนท์) เหล่านี้ เราเขียนให้อยู่ระหว่างวงเล็บเปิดและปิดต่อจากชื่อของฟังก์ชัน บางฟังก์ชันไม่ต้องการข้อมูลใดๆจากภายนอกหรือจากผู้เรียกใช้ เราก็กำหนดให้พารามิเตอร์ของฟังก์ชันเป็น `void` คือ 'ว่างเปล่า' ถ้าเรานิยามฟังก์ชันใน ลักษณะนี้ เมื่อถึงเวลาเรียกใช้ เราก็ไม่ต้องผ่านข้อมูลใดๆให้เป็นพารามิเตอร์ของฟังก์ชัน แต่ถ้าเราเรียกใช้ฟังก์ชันดังกล่าวพร้อมกับให้พารามิเตอร์ใดๆแก่ฟังก์ชันนี้ ผลก็คือ เมื่อทำการคอมไพล์โปรแกรมโค้ด คอมไพล์เลอร์จะแจ้งความผิดให้เราทราบ หรือในทางกลับกัน ถ้าเรานิยามฟังก์ชันที่มีอาร์กิวเมนท์ แต่เวลาเราเรียกใช้ฟังก์ชัน เราให้ค่าพารามิเตอร์ไม่ครบตามจำนวน เช่น สมมุติว่า ฟังก์ชันมีจำนวนอาร์กิวเมนท์เท่ากับ สอง แต่เวลาเราเรียกใช้ฟังก์ชัน เราใช้พารามิเตอร์ที่มีจำนวนน้อยกว่าหรือมากกว่าที่นิยามไว้ ก็จะถือว่าผิดหลักไวยกรณ์

นอกจากจำนวนของพารามิเตอร์จะต้องเท่ากับจำนวนของพารามิเตอร์ที่แจ้งไว้ในส่วนหัวของฟังก์ชันในครั้งที่เรานิยามฟังก์ชันแล้ว แบบของข้อมูลที่เป็นพารามิเตอร์แต่ละตัวจะต้องตรงกับแบบข้อมูลของพารามิเตอร์ที่นิยามไว้ตามลำดับในส่วนหัวของฟังก์ชันด้วย

การนิยามฟังก์ชันภายในฟังก์ชันอื่นเหมือนกับในภาษาปาสคาลนั้น เราไม่สามารถทำได้ในภาษาซี ตัวอย่างที่ผิด เช่น

```
int printReverseNumber (int number)
{
      int i;

      void printDigit (int n)
      {
            printf("%c", (char)n);
      }

      for (i=0; number; i++)
      {
            printDigit (number % 10);
            number /= 10;
      }
      return i;
}
```

4.1.3 ประโยคคำสั่ง return

สำหรับฟังก์ชันใดๆที่ให้ค่ากลับคืน ซึ่งก็คือฟังก์ชันที่ไม่ได้มีคำว่า void อยู่หน้าชื่อในส่วนหัวของฟังก์ชัน ฟังก์ชันลักษณะนี้จะต้องมีประโยคคำสั่ง return อยู่ในส่วนตัวของฟังก์ชัน

```
return  return_value ;
```

จะวางไว้ในบรรทัดใดภายในฟังก์ชันก็ได้ แต่โปรดจำไว้ว่าเมื่อโปรแกรมทำงานมาถึงคำสั่ง return นี้จะส่งผลให้โปรแกรมจบการทำงานของฟังก์ชันดังกล่าว แล้วผ่านค่าของนิพจน์ return_value ที่อยู่ถัดไปเป็นค่าของฟังก์ชันกลับคืนไปยังผู้เรียกใช้ ตัวอย่างการใช้งาน เช่น

```
int function_one (int choice)
{
     if (choice < 0)
          return -1;
     if (choice > 0)
          return 1;
     return 0;
}
```

ภายในฟังก์ชัน function_one() มีประโยคคำสั่ง return สามประโยค ถ้าตัวแปร choice มีค่าน้อยกว่า 0 ฟังก์ชันก็จะหยุดการทำงานและผ่านค่า -1 กลับคืนอันเป็นผลมาจากประโยคคำสั่ง return -1; ถ้าตัวแปรมีค่ามากกว่า 0 โปรแกรมก็จะกระทำคำสั่ง return 1; และจบการทำงานของฟังก์ชัน ถ้าเงื่อนไขทั้งสองเป็นเท็จเมื่อตัวแปรมีค่าเป็น 0 โปรแกรมก็จบการทำงานของฟังก์ชันด้วยการผ่านค่า 0 เป็นค่าของฟังก์ชัน เราจะเห็นได้ว่า ไม่ว่าตัวแปร choice จะมีค่ามากกว่า น้อยกว่า หรือเท่ากับศูนย์ ทุกกรณีจะมีสายงานที่มีประโยคคำสั่ง return อยู่ ซึ่งถูกต้องตามรูปแบบของฟังก์ชัน เพราะในกรณีนี้ฟังก์ชันจะต้องให้ค่าใดค่าหนึ่งกลับคืนไปยังผู้เรียกใช้

สำหรับฟังก์ชันที่มีแบบข้อมูลกลับคืนเป็น void ซึ่งไม่ให้ค่าใดผ่านไปยังผู้เรียกใช้เมื่อจบการทำงานของฟังก์ชัน เราก็ไม่ต้องใช้คำสั่ง return ในกรณีนี้ฟังก์ชันจะหยุดการทำงานก็ต่อเมื่อฟังก์ชันได้กระทำคำสั่งจนถึงคำสั่งสุดท้ายในส่วนตัวของฟังก์ชันครบแล้ว แต่ถ้าเราต้องการใช้คำสั่ง return เราก็สามารถทำได้ แต่ไม่ต้องเติมนิพจน์ใดๆตามหลัง ตัวอย่างเช่น

```
void PrintNumber (int number)
{
     if (number < 0)
          return ;
     printf ("%d\n", number);
}
```

ตัวอย่างสั้นๆนี้แสดงให้เห็นการทำงานของคำสั่ง `return ;` ในฟังก์ชันแบบ `void` สำหรับฟังก์ชันนี้เราต้องการพิมพ์ตัวเลขที่เป็นพารามิเตอร์ออกทางจอภาพ แต่เฉพาะค่าของตัวเลขที่ไม่ใช่จำนวนเต็มลบเท่านั้น เราจะเห็นได้ว่า ถ้าตัวแปร `number` มีค่าน้อยกว่าศูนย์ โปรแกรมจะกระทำคำสั่ง `return` ซึ่งก็คือ หยุดการทำงานของฟังก์ชันและกลับไปยังผู้เรียกใช้ทันทีโดยไม่ต้องทำคำสั่งอื่นใดในฟังก์ชันต่อไป การใช้ประโยค `return` ในโครงสร้างของประโยค `if` ตามลักษณะนี้จึงเป็นการสร้างกลไลแบบเงื่อนไขในการหยุดการทำงานของฟังก์ชัน

เมื่อเราได้สร้างฟังก์ชันใดๆขึ้นมาใช้ เราก็ควรจะเขียนคำอธิบายที่ให้ข้อมูลเกี่ยวกับฟังก์ชันอย่างเพียงพอ เช่น ฟังก์ชันนี้มีหน้าที่อะไร ใช้พารามิเตอร์ใดบ้างและแบบข้อมูลใด นอกจากนี้ก็ต้องเขียนกำกับไว้ด้วยว่า ฟังก์ชันนี้มีข้อจำกัดหรือเงื่อนไขอย่างไรเมื่อเราต้องการเรียกใช้ฟังก์ชันนี้ในโปรแกรมโค้ด สำหรับบางฟังก์ชันเราเขียนอธิบายการทำงานเพียงสั้นๆถ้าฟังก์ชันไม่ซับซ้อนมากนักหรือในทางตรงกันข้าม ถ้าฟังก์ชันใดมีขั้นตอนการทำงานที่ซับซ้อน คำอธิบายก็อาจจะมีมากเป็นพิเศษ สิ่งเหล่านี้จะช่วยให้การเขียนโปรแกรมง่ายและมีประสิทธิภาพมากขึ้น

4.2 การตั้งชื่อฟังก์ชัน

ดังที่กล่าวไปแล้วเราควรตั้งชื่อฟังก์ชันให้ได้ความหมายและสามารถบ่งบอกได้ว่าหน้าที่ของฟังก์ชันนี้คืออะไร ตัวอย่างเช่น

`sqr()`	ย่อมาจาก `square` หมายถึงการหาค่ายกกำลังสองของตัวเลข
`sqrt()`	ย่อมาจาก `square root` หมายถึงการหาค่ารากที่สองของตัวเลข

หรืออาจจะเป็นอะไรที่ยาวกว่านี้ก็ได้ ซึ่งประกอบด้วยคำหลายๆคำ เช่น

```
getCurrentDir()
getcurrdir()
get_current_dir()
```

โดยเราสามารถเลือกชื่อใดชื่อหนึ่งจากตัวอย่างทั้งสามซึ่งใช้แทนวลี get current current หมายถึงฟังก์ชันนี้ ใช้ในการหาชื่อของไดเรคทรอรี่ปัจจุบันที่โปรแกรมของเรากำลังทำงานอยู่

นอกจากนี้จะต้องระมัดระวังในการเลือกชื่อของฟังก์ชัน โดยเฉพาะอย่างยิ่งไม่ควรเลือกชื่อฟังก์ชันที่ซ้ำกับชื่อของฟังก์ชันมาตราฐานในภาษาซี

ฟังก์ชันสองฟังก์ชันที่มีชื่อเหมือนกันจะต้องเป็นฟังก์ชันเดียวกัน ซึ่งหมายความว่า ถ้าเรานิยามฟังก์ชันสองฟังก์ชันที่มีจำนวนของพารามิเตอร์ แบบข้อมูลของพารามิเตอร์ หรือ แบบข้อมูลกลับคืนที่แตกต่างกันแล้ว เราห้ามใช้ชื่อฟังก์ชันที่เหมือนกัน ตัวอย่างเช่น เราต้องการเขียนฟังก์ชันในการหาค่าสัมบรูณ์ของตัวเลขจำนวนเต็มแบบ `int` เราก็เขียนได้ดังนี้ เช่น

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

ฟังก์ชันนี้สามารถใช้ได้กับข้อมูลแบบ `int` เท่านั้น สมมุติว่าเราต้องการใช้ฟังก์ชันกับข้อมูลแบบ `double` เราก็ต้องเขียนฟังก์ชันเพิ่มเติมขึ้นมาใหม่สำหรับใช้กับข้อมูลแบบ `double` นี้โดยเฉพาะ เช่น

```
double abs (double x)
{
     return (x >=0.0) ? x : -x;
}
```

ปัญหาก็คือว่า ถ้าเราต้องการใช้ฟังก์ชันทั้งสองในโปรแกรมโค้ดเดียวกัน คอมไพล์เลอร์จะถือว่าฟังก์ชันชื่อ `abs()` จะต้องมีเพียงฟังก์ชันเดียวเท่านั้น สมมุติว่า ถ้าเรานิยามฟังก์ชัน `abs()` สำหรับข้อมูลแบบ `int` ขึ้นใช้ก่อน ทำให้ฟังก์ชัน `abs()` สำหรับ `double` ที่เรานิยามขึ้นภายหลังไม่ถูกต้องตามหลักภาษาซี ดังนั้นเราจะต้องใช้ชื่ออื่นที่แตกต่างออกไป เช่น `double_abs()` หรืออาจจะเป็น `dabs()` ก็ได้

4.3 การเรียกใช้ฟังก์ชัน

การเรียกใช้ฟังก์ชันในภาษาซีแบ่งออกได้เป็นสองแบบหลักที่สำคัญคือ เรียกใช้โดยผ่านค่า (Call By Value) และ เรียกใช้โดยผ่านตัวอ้างอิง (Call By Reference)

การเรียกใช้ฟังก์ชันโดยผ่านค่าเป็นการเรียกใช้ฟังก์ชัน โดยเราผ่านค่าของพารามิเตอร์ไปยังฟังก์ชัน และมิใช่ตัวข้อมูลของพารามิเตอร์ที่แท้จริงแต่เป็นสำเนา เช่น สมมุติว่า เราต้องการเรียกใช้ฟังก์ชัน `abs()` สำหรับหาค่าสัมบรูณ์ของตัวแปร `x` แบบ `int` เช่น `abs(x)` นิพจน์ดังกล่าว

เป็นการเรียกใช้ฟังก์ชัน abs โดยใช้ตัวแปร x เป็นพารามิเตอร์ของฟังก์ชัน (ตัวแปร x มีขอบเขตการทำงานอยู่นอกฟังก์ชัน) ดังนั้นเมื่อ ฟังก์ชันเริ่มทำงานก็จะทำสำเนาของตัวแปร x และใช้สำเนานี้ภายในฟังก์ชัน และมิได้ใช้ตัวแปร x ภายในฟังก์ชัน เมื่อจบการทำงานของฟังก์ชัน สำเนานี้ก็จะถูกทำลายไป (โดยการคืนหน่วยความจำที่จองไว้สำหรับสำเนานี้) ดังนั้นตัวแปร x ที่เป็นพารามิเตอร์จะไม่ถูกเปลี่ยนแปลงแก้ไขโดยขั้นตอนใดๆของฟังก์ชัน เพราะฟังก์ชันที่เราเรียกใช้โดยผ่านค่านี้จะไม่เกี่ยวข้องกับตัวแปรดังกล่าว เพียงแต่นำค่าของตัวแปรที่เก็บไว้ในตัว แปรสำเนาไปใช้งานเท่านั้น

การเรียกใช้ฟังก์ชันอีกลักษณะหนึ่งคือ การเรียกใช้โดยผ่านตัวอ้างอิง ในกรณีนี้แทนที่เราจะผ่านค่าของตัวแปร เราใช้ตัวอ้างอิงที่อยู่ของตัวแปร และทำให้ฟังก์ชันสามารถเข้าถึงตัวแปรที่เป็นพารามิเตอร์ของฟังก์ชันได้โดยตรง

4.4 การทำงานของฟังก์ชัน

เมื่อเราได้รู้จักองค์ประกอบและวิธีการนิยามฟังก์ชันขึ้นมาใช้เองแล้ว ต่อไปเราจะเรียนรู้ขั้นตอนการทำงานของฟังก์ชัน ถ้าเราต้องการเรียกใช้ฟังก์ชันใดๆซึ่งเป็นนิพจน์จากการเรียกใช้ฟังก์ชัน เราก็ต้องเขียนชื่อฟังก์ชันตามด้วยเครื่องหมายวงเล็บเปิดปิด ถ้าฟังก์ชันต้องการพารามิเตอร์ใดๆ เราก็ต้องเติมนิพจน์อาจจะเป็นตัวแปร ค่าคงที่หรือนิพจน์ใดๆ ให้อยู่ระหว่างเครื่องหมายวงเล็บทั้งสอง ถ้ามีมากกว่าหนึ่งพารามิเตอร์ เราก็ต้องเขียนพารามิเตอร์ตามลำดับที่เราได้นิยามไว้ในส่วนหัวของฟังก์ชัน โดยเขียนแบ่งแยกจากกันโดยใช้ เครื่องหมายจุลภาค (,)

เมื่อโปรแกรมทำงานมาถึงประโยคคำสั่งที่เรียกใช้ฟังก์ชันดังกล่าว โปรแกรมก็จะเรียกฟังก์ชันขึ้นมาทำงาน โปรดพิจารณาตัวอย่างต่อไปนี้

```
double FahrToCelsius (double Fahrenheit)
{
      return (Fahrenheit - 32.0) / 1.8;
}
```

ฟังก์ชันตัวอย่างนี้เราใช้ในการเปลี่ยนค่าของอุณหภูมิจากฟาเรนไฮน์เป็นเซลเซียส

```
int main()
{
      double FahrenheitTemp = 90.5, CelsiusTemp;

      CelsiusTemp = FahrToCelsius (FahrenheitTemp);
      return 0;
}
```

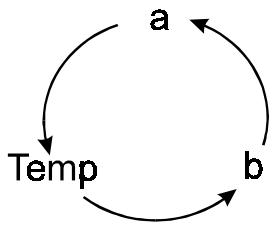
เมื่อโปรแกรมเริ่มต้นทำงานมาถึงขั้นตอนที่ต้องเรียกใช้ฟังก์ชัน `FahrToCelsius()` โปรแกรมก็จะคำนวณค่าของตัวแปร `FahrenheitTemp` แล้วทำสำเนาค่าของพารามิเตอร์ จากนั้นก็เก็บค่านี้ไว้ในตัวแปร `Fahrenheit` และฟังก์ชันก็จะใช้ตัวแปร `Fahrenheit` ซึ่งมีค่าในขณะนั้นเท่ากับ `90.5` ในการคำนวณค่าของอุณหภูมิที่อยู่ในหน่วยเซลเซียส หลังจากนิพจน์ที่อยู่ในประโยค `return` ได้ถูกคำนวณแล้ว คำสั่ง `return` ก็จะผ่านผลลัพธ์จากการคำนวณค่าของนิพจน์กลับคืนไปยังผู้เรียก จากนั้นเราก็จะได้ค่าของนิพจน์ที่เกิดจากการเรียกใช้ฟังก์ชัน และกำหนดค่านี้ให้เก็บไว้ใน `CelsiusTemp` และโปรแกรมก็จะเริ่มกระทำคำสั่งในขั้นต่อไปจนกว่าจะจบโปรแกรม

ในกรณีที่เราเรียกใช้ฟังก์ชันโดยการผ่านค่า (Call By Value) ถ้าฟังก์ชันมีพารามิเตอร์ใดๆ โปรแกรมก็จะผ่านค่าของพารามิเตอร์ไปให้ฟังก์ชัน แต่เป็นสำเนาของพารามิเตอร์เหล่านั้น การเรียกฟังก์ชันโดยผ่านค่าจะไม่ทำให้ค่าของพารามิเตอร์เปลี่ยนแปลงไปเมื่อจบการทำงานของฟังก์ชัน โดยเฉพาะในกรณีที่เราใช้ตัวแปรเป็นอาร์กิวเมนท์ เพราะฟังก์ชันจะใช้สำเนาของพารามิเตอร์(อยู่ในรูปของตัวแปร) ถ้ามีการเปลี่ยนแปลงใดๆในฟังก์ชันก็จะกระทำเฉพาะกับสำเนาเท่านั้น ไม่ใช่กับตัวข้อมูลที่เราผ่านเป็นพารามิเตอร์ ซึ่งแตกต่างจากการเรียกใช้โดยการผ่านตัวอ้างอิง (Call By Reference) เป็นอาร์กิวเมนท์ของฟังก์ชัน

ในกรณีหลังนี้ตัวอ้างอิงจะเป็นตัวชี้ หรือ พอนย์เตอร์ ฟังก์ชันสามารถเปลี่ยนแปลงแก้ไขค่าของพารามิเตอร์ซึ่งเป็นแหล่งข้อมูลจริงได้โดยตรง เพราะไม่มีการสร้างสำเนาขึ้นมา แต่จะใช้ข้อมูลจริง (ตัวแปร) โดยการอ้างแหล่งที่มา (ซึ่งก็คือที่อยู่ของข้อมูลเหล่านั้นในหน่วยความจำ) เมื่อฟังก์ชันทราบที่อยู่ของแหล่งข้อมูลที่เป็นพารามิเตอร์แล้วฟังก์ชันก็สามารถเข้าถึงแหล่งข้อมูลเหล่านั้นได้โดยตรง และสามารถเปลี่ยนแปลงแก้ไขค่าของข้อมูลได้

ข้อดีของการเรียกใช้ฟังก์ชันโดยการอ้างอิง คือ ในกรณีที่เราต้องการให้ฟังก์ชันสร้างผลลัพธ์หรือให้ข้อมูลมากกว่าหนึ่งตัวเป็นค่าของฟังก์ชัน แต่เนื่องจากว่าฟังก์ชันสามารถผ่านค่าได้เพียงค่าเดียวเท่านั้นเมื่อใช้คำสั่ง `return` ดังนั้นเราจึงต้องหาวิธีการที่ใช้ผ่านค่าอื่นๆที่ได้จากการทำงานของฟังก์ชันกลับคืนไปยังผู้เรียกใช้ ตัวอย่างเช่น สมมุติว่าเราต้องการผ่านข้อมูลเป็นพารามิเตอร์ให้ฟังก์ชันตัวหนึ่ง โดยที่ฟังก์ชันนี้ใช้ข้อมูลดังกล่าวในการสร้างผลลัพธ์ซึ่งเป็นข้อมูลสองตัว และเราต้องการนำค่าของข้อมูลทั้งสองไปใช้ต่อหลังจากที่ฟังก์ชันจบการทำงาน ถ้าไม่มีการผ่านผลลัพธ์กลับไปยังผู้เรียก ข้อมูลนี้ก็จะถูกทำลายไปหลังจากจบการทำงานของฟังก์ชัน แต่เนื่องจากว่าเรามีข้อมูลสองตัวที่จะต้องผ่านกลับไปยังผู้เรียกใช้ ถ้าใช้คำสั่ง `return` ก็สามารถผ่านค่าของข้อมูลได้เพียงค่าใดค่าหนึ่งเท่านั้น ปัญหานี้เราสามารถแก้ไขได้โดยการสร้างและเรียกใช้ฟังก์ชันพร้อมกับอ้างที่อยู่ของพารามิเตอร์ซึ่งสามารถทำได้โดยใช้พอยน์เตอร์ แต่จะไม่ขอกล่าวถึงรายละเอียดในบทนี้

ตัวอย่างนี้แสดงความแตกต่างระหว่างการเรียกใช้ฟังก์ชันโดยการผ่านค่าและโดยการอ้างอิง



ภาพประกอบที่ 4.3 การสลับค่าระหว่างตัวแปร a และ b โดยใช้ตัวแปร Temp ช่วย

```
#include <stdio.h>

void Swap (int a, int b)
{
      int temp;

      /* Swap the values of two integer variables */
      temp =  a;
      a    =  b;
      b    =  temp;
      printf ("Swap() : a = %d, b = %d\n", a, b);
}

int main ()
{
      int a = 1, b = 100;

      printf ("main() : a = %d, b = %d\n", a, b);
      Swap (a, b);
      printf ("main() : a = %d, b = %d\n", a, b);
      return 0;
}
```

ผลที่ได้จากโปรแกรมคือ

```
main() : a = 1, b = 100
Swap() : a = 100, b = 1
main() : a = 1, b = 100
```

ในตัวอย่างนี้ค่าของตัวแปร a และ b จะไม่เปลี่ยนแปลงหลังจากที่เราเรียกใช้ฟังก์ชัน Swap() ทั้งๆที่เรา ต้องการให้ฟังก์ชันนี้สลับค่าระหว่าง a และ b ซึ่งควรจะได้ค่า a เท่ากับ 100 และ b เท่ากับ 1 เราเห็นได้ว่า มีการสลับค่าระหว่าง a และ b จริงแต่มีผลเฉพาะในฟังก์ชัน Swap() เท่านั้น เหตุที่เป็นเช่นนี้เพราะว่าเราเรียกใช้ฟังก์ชันแบบผ่านค่า ดังนั้นสิ่งที่เราเปลี่ยนแปลงในฟังก์ชัน Swap() จะกระทำกับสำเนาของพารามิเตอร์ a และ b โปรดสังเกตว่า ตัวแปร a และ b ใน

ฟังก์ชัน Swap() แม้ว่าจะมีชื่อเหมือนกับชื่อของตัวแปรในฟังก์ชันหลัก main() แต่เป็นเพียงสำเนาที่ใช้เฉพาะในฟังก์ชัน Swap() เท่านั้น ถ้าเราต้องการให้ฟังก์ชัน Swap() สลับค่าระหว่าง a และ b ได้จริงหลังจากที่เรียกใช้ฟังก์ชัน เราก็ต้องใช้วิธีผ่านตัวอ้างอิงซึ่งสามารถทำได้โดยใช้พอยน์เตอร์

```
#include <stdio.h>

void Swap (int *a, int *b)
{
      int temp;

      /* Swap the values of two integer variables */
      temp = *a;
      *a  =  *b;
      *b  =  temp;
      printf ("Swap() : a = %d, b = %d\n", a, b);
}

int main ()
{
      int a = 1, b = 100;

      printf ("main() : a = %d, b = %d\n", a, b);
      swap (&a, &b);
      printf ("main() : a = %d, b = %d\n", a, b);
      return 0;
}
```

โปรแกรมจะให้ผลดังนี้

```
Before : a = 1, b = 100
Swap() : a = 100, b = 1
After  : a = 100, b = 1
```

เราได้ค่าของตัวแปรทั้งสองถูกต้องตามที่ต้องการ เพราะฟังก์ชัน Swap() สามารถเปลี่ยนแปลงค่าของตัวแปร a และ b ใน ฟังก์ชันหลักได้โดยตรง

ตัวอย่างการเรียกใช้ฟังก์ชันแบบอ้างอิงที่สำคัญอีกตัวอย่างหนึ่งคือการใช้ฟังก์ชันมาตราฐาน scanf() ซึ่งได้ถูกนิยามไว้ใน <stdio.h> เราจะใช้ฟังก์ชันนี้สำหรับการอ่านข้อมูลต่างๆจากแป้นพิมพ์ โดยการผ่านที่อยู่ของตัวแปรเพื่อที่ให้ฟังก์ชันสามารถเก็บค่าที่อ่านได้ไว้ในหน่วยความจำของตัวแปร และสามารถนำไปใช้ต่อไปภายในโปรแกรมของเราได้ รูปแบบการใช้ฟังก์ชัน scanf() นั้นจะคล้ายกับการเรียกใช้ฟังก์ชันมาตราฐาน printf() ที่เราได้ทำความรู้จักไปแล้ว

```
#include <stdio.h>

int main()
{
      int    i;
      double j;

      printf("Please enter any integer number (i) : ");
      scanf("%d", &i);
      printf("Please enter any real number (j) : ");
      scanf("%lf", &j);

      printf("i = %d, j = %lf\n", i, j);
      return 0;
}
```

ตัวอย่างการทำงานของโปรแกรมมีลักษณะดังนี้

```
Please enter any integer number (i) : -1231
Please enter any real number (j) : 2.301e-1
i = -1231, j = 0.230100
```

ตัวเลขที่เป็นตัวเอนและพิมพ์เข้ม ใช้แสดงถึงข้อมูลตัวเลขที่เราป้อนให้โปรแกรมทางแป้นพิมพ์ โปรดสังเกตว่า ฟังก์ชัน `scanf()` จะอ่านข้อมูลที่เป็นลำดับของตัวอักขระ (ตัวอักษรหรือเครื่องหมายต่างๆ รวมทั้งตัวเลข) แต่ในกรณีนี้เราต้องการอ่านข้อมูลเฉพาะตัวเลขเท่านั้น

เราอาจจะกล่าวได้ว่า สำหรับการเรียกใช้ฟังก์ชันโดยผ่านค่าของตัวแปรนั้น เป็นการใช้ค่าของตัวแปรเท่านั้นภายในฟังก์ชัน หรืออ่านได้อย่างเดียว ในขณะที่การเรียกใช้ฟังก์ชันโดยผ่านตัวอ้างอิงที่เก็บที่อยู่ของตัวแปรเปิดโอกาสให้เราอ่านหรือ/และเขียนค่าของตัวแปรที่เป็นพารามิเตอร์ได้ภายในฟังก์ชันได้

4.5 การใช้ต้นแบบของฟังก์ชัน

เท่าที่ผ่านมา เราได้นิยามฟังก์ชันขึ้น โดยเขียนส่วนหัวพร้อมกับสร้างส่วนตัวของฟังก์ชันในคราวเดียวกันแล้วจึงเรียกใช้ฟังก์ชันในภายหลัง บางครั้งเราต้องการเรียกใช้ฟังก์ชันก่อนที่จะสร้างฟังก์ชันอย่างสมบรูณ์ ถ้าเรายังมิได้นิยามฟังก์ชันก่อนที่จะเรียกใช้ ปัญหาที่เกิดขึ้นตามมาก็จะเป็นดังตัวอย่างนี้

```
#include <stdio.h>

int main()
{
     int a = 1453, b = -2313;
     printf ("%d + %d = %d\n", a, b, add(a,b));
     return 0;
}

int add (int a, int b)
{
     return a+b;
}
```

ในตัวอย่างนี้เราเรียกใช้ฟังก์ชัน add() ทั้งๆที่ยังมิได้นิยามฟังก์ชันนี้เลย แต่ได้นิยามและสร้างฟังก์ชันโดยวางขั้นตอนไว้ท้ายฟังก์ชันหลัก สำหรับการเรียกใช้ฟังก์ชันก่อนที่จะมีการนิยามฟังก์ชัน โอกาสที่จะเกิดความผิดพลาดในโปรแกรมมีสูงมาก เพราะคอมไพล์เลอร์ไม่ทราบรายละเอียดของฟังก์ชันเลย การตรวจสอบความถูกต้องของแบบข้อมูลของพารามิเตอร์จึงเป็นไปได้ยาก ดังนั้นจึงควรหลีกเลี่ยงเป็นอย่างยิ่ง

วิธีแก้ไขและนิยมใช้ก็คือ การแจ้งใช้เฉพาะส่วนหัวของฟังก์ชันก่อน โดยวางไว้ในตอนต้นของโปรแกรมโค้ด เราเรียกส่วนนี้ของฟังก์ชันว่า Function Prototype หรือ *ต้นแบบของฟังก์ชัน* เมื่อได้แจ้งต้นแบบของฟังก์ชันแล้ว เราสามารถสร้างฟังก์ชันอย่างสมบรูณ์แบบได้ในภายหลัง และสามารถเรียกใช้ฟังก์ชัน ก่อนหรือหลังการสร้างฟังก์ชันก็ได้ โปรดดูโปรแกรมตัวอย่างข้างล่างนี้

```
#include <stdio.h>

int add (int a, int b);  /* function prototype */

int main()
{
     int a = 1453, b = -2313;
     printf ("%d + %d = %d\n", a, b, add(a,b));
     return 0;
}

int add (int a, int b)
{
     return a+b;
}
```

ในบางครั้งอาจจะพบว่ามีการเขียนฟังก์ชันต้นแบบแต่ไม่มีชื่อของตัวแปรต่างๆที่เป็นพารามิเตอร์มีแต่เพียงแบบข้อมูลเท่านั้น เช่น

```
int add (int, int);  /* function prototype */
```

การแจ้งต้นแบบของฟังก์ชันในลักษณะนี้เราสามารถทำได้ แต่ขอแนะนำว่าควรจะหลีกเลี่ยง เพราะการเขียนชื่อของตัวแปรพร้อมแบบข้อมูลจะทำให้เราทราบได้ง่ายและชัดเจนว่า เราจะใช้ตัวแปรหรือพารามิเตอร์แต่ละตัวอย่างไรเมื่อเวลาเราเรียกใช้ฟังก์ชันในแต่ละครั้ง เราถือว่าต้นแบบของฟังก์ชันควรจะให้รายละเอียด เกี่ยวกับฟังก์ชันมากที่สุดเท่าที่จะเป็นไปได้

4.6 ชนิดของตัวแปร

4.6.1 ตัวแปรอัตโนมัติ (Automatic Variable)

ตัวแปรอัตโนมัติเป็นตัวแปรที่เราแจ้งใช้ภายในโปรแกรมซึ่งก็คือตัวแปรทั่วๆไปที่เรานิยมใช้ เช่น ถ้า เราแจ้งใช้ตัวแปรตามรูปแบบต่อไปนี้

```
int x, y;
double f = 132.201;
```

ก็จะหมายถึงตัวแปรแบบอัตโนมัติที่เราเขียนกำกับไว้ข้างหน้าด้วยคำว่า auto

```
auto int x, y;
auto double f = 132.201;
```

ตามปรกติแล้ว เราไม่นิยมเขียนคำว่า auto ไว้ข้างหน้าเมื่อเวลาเราแจ้งใช้ตัวแปรแบบอัตโนมัติ เพราะถือว่า ไม่จำเป็น

ถ้าตัวแปรแบบอัตโนมัติเหล่านี้ถูกแจ้งใช้ภายในบล็อกหรือฟังก์ชัน เมื่อโปรแกรมเริ่มทำงานภายในบล็อกดังกล่าวโปรแกรมก็จะสร้างตัวแปรนี้ขึ้นและใช้เฉพาะในช่วงเวลาที่โปรแกรมกำลังทำงานอยู่ในบล็อกนี้เท่านั้น เมื่อโปรแกรมจบการทำงานของบล็อก ตัวแปรของบล็อกก็จะถูกทำลายไปโดยอัตโนมัติ ดังนั้นเราจึงไม่สามารถใช้หรือเข้าถึงข้อมูลต่างๆที่เก็บไว้ในตัวแปรอัตโนมัติได้อีกเมื่ออยู่ภายนอกบล็อกของตัวแปรนี้ เนื่องจากว่าตัวแปรแบบอัตโนมัติจะถูกสร้างขึ้นและทำลายไปขึ้นอยู่กับระยะเวลาการทำงานของบล็อก ซึ่งแตกต่างจากการทำงานของตัวแปรสถิต

4.6.2 ตัวแปรสถิต (Static Variable)

ตัวแปรชนิดนี้จะแตกต่างจากตัวแปรอัตโนมัติตรงที่ว่า เวลาโปรแกรมจบการทำงานของบล็อกหรือฟังก์ชัน ตัวแปรสถิตที่อยู่ภายในบล็อกหรือฟังก์ชันนั้นจะไม่ถูกทำลายไป ดังนั้นเมื่อเข้าสู่ภายในบล็อกดังกล่าวอีกครั้ง(ในกรณีที่อยู่ในฟังก์ชัน การเข้าสู่บล็อกของฟังก์ชันอีกครั้งก็คือการเรียกใช้ฟังก์ชันอีกครั้งหนึ่งนั้นเอง) เราสามารถใช้ค่าของตัวแปรสถิตได้ ซึ่งเป็นค่าของตัวแปรจากการทำงานของบล็อกในครั้งก่อน ตัวอย่างต่อไปนี้แสดงให้เห็นความแตกต่างระหว่างตัวแปรอัตโนมัติและตัวแปรสถิต

ตัวอย่างที่หนึ่ง

```
#include <stdio.h>

void function_one()
{
     auto   int i = 100;
     static int j = 100;

     i++;
     j++;

     printf ("i = %d, j = %d\n", i, j);
}

int main()
{
     int i;

     for (i = 0; i < 5; i++)
          function_one();
     return 0;
}
```

ผลที่ได้จากโปรแกรมคือ

```
i = 101, j = 101
i = 101, j = 102
i = 101, j = 103
i = 101, j = 104
i = 101, j = 105
```

จากตัวอย่างนี้เราจะเห็นได้ว่า ตัวแปร i เป็นตัวแปรแบบอัตโนมัติ ในขณะที่ตัวแปร j เป็นตัวแปรแบบสถิต ทุกครั้งที่มีการเรียกใช้ฟังก์ชัน function_one() ค่าของตัวแปร i เมื่อจบการทำงานของฟังก์ชันแต่ละครั้งจะมีค่าเป็น 101 เสมอ ในขณะที่ค่าของตัวแปร j มีค่าเพิ่มขึ้นเรื่อยๆทุกครั้งที่มีการเรียกใช้ฟังก์ชัน

เนื่องจากว่าตัวแปร i เป็นตัวแปรอัตโนมัติ ดังนั้นเมื่อฟังก์ชันถูกเรียกใช้ให้ทำงาน ตัวแปร i ก็จะถูกสร้างขึ้นใหม่ทุกครั้ง นอกจากนั้นเรายังได้แจ้งใช้ตัวแปรพร้อมกับติดตั้งค่าเริ่มต้นให้ตัวแปรเท่ากับ 100 ทุกครั้งที่ฟังก์ชันเริ่มทำงานค่าของตัวแปร i จะมีค่าเริ่มต้นเท่ากับ 100 เสมอ

สำหรับตัวแปร j ในกรณีนี้เราใช้ตัวแปรแบบสถิต เมื่อโปรแกรมเรียกใช้ฟังก์ชันเป็นครั้งแรก ตัว แปร j ก็จะถูกสร้างขึ้นและมีค่าเริ่มต้นเป็น 100 เช่นเดียวกับตัวแปร i แต่จะแตกต่างกันเมื่อมีการเรียก ใช้ฟังก์ชันในครั้งต่อๆไป เมื่อฟังก์ชันเริ่มทำงานอีกครั้ง (ครั้งที่สอง สาม ...) ตัวแปร i จะถูกสร้างขึ้นใหม่ และถูกติดตั้งค่าเริ่มต้นเป็น 100 แต่ตัวแปร j จะไม่ถูกสร้างขึ้นใหม่อีกเพราะเหตุที่ว่าตัวแปรสถิตจะไม่ถูกทำลายไปเมื่อจบบล็อกการทำงานของฟังก์ชัน ดังนั้นเมื่อฟังก์ชันถูกเรียกให้ทำงานในครั้งต่อๆไปก็สามารถใช้ค่าของตัวแปรสถิตต่อไปได้ ในขณะที่ค่าของตัวแปร i จะถูกลบทิ้งไปทุกเมื่อฟังก์ชันจบการทำงานในแต่ละครั้ง

ตัวอย่างที่สอง

```
#include <stdio.h>

void saveValue (int new_value)
{
      static int value = 0;

      if (new_value == 0)
      {
            printf ("Set the value to zero.\n");
            value = 0;
      }
      else if (value == 0)
      {
            printf ("Set the value to %d.\n", new_value);
            value = new_value;
      }
      else
       printf ("Cannot change value! Value = %d\n",
                  value);
}

int main()
{
      printValue(5);
      printValue(1);
```

```
        printValue(0);
        printValue(-1);
        return 0;
}
```

ผลของโปรแกรมคือ

```
Set the value to 5.
Cannot change value! Value = 5
Set the value to zero.
Set the value to -1.
```

ในโปรแกรมตัวอย่างนี้ เรานิยามฟังก์ชัน `saveValue()` เพื่อใช้บันทึกข้อมูลซึ่งเป็นค่าของพารามิเตอร์แบบ `int` โดยเก็บไว้ในตัวแปร `value` ซึ่งเป็นตัวแปรแบบสถิต การทำงานของฟังก์ชันสรุปได้ดังนี้ ถ้าค่าของพารามิเตอร์ `new_value` มีค่าเท่ากับ 0 ก็กำหนดให้ค่าของ `value` มีค่าเป็น 0 ในกรณีที่ค่าของพารามิเตอร์ไม่เท่ากับ 0 และต้องการบันทึกค่านี้ไว้ในตัวแปร `value` จะทำได้ก็ต่อเมื่อค่าของ `value` ในขณะนั้นมีค่าเป็น 0 เท่านั้น ถ้าค่าของ `value` ไม่เท่ากับศูนย์ เราก็ไม่สามารถบันทึกทับค่าของตัวแปรในขณะนั้นได้ ยกเว้นสำหรับค่าของพารามิเตอร์ที่เป็นศูนย์

4.6.3 ตัวแปรรีจิสเตอร์ (Register Variable)

ตามปรกติแล้วหน่วยความจำของตัวแปรจะอยู่ในหน่วยความจำหลักของคอมพิวเตอร์ แต่ในภาษาซีเราสามารถใช้รีจิสเตอร์ของคอมพิวเตอร์ (Machine Register) ซึ่งมีขนาดหน่วยความจำขนาดเท่ากับหนึ่งเวิร์ดในการเก็บข้อมูลของตัวแปรได้ ข้อดีก็คือ การอ่านหรือเขียนข้อมูลลงในหน่วยความจำของรีจิสเตอร์จะทำได้เร็วกว่าการอ่านและเขียนข้อมูลลงในหน่วยความจำหลัก เพราะรีจิสเตอร์เป็นส่วนหนึ่งของหน่วยประมวล ผลกลางหรือ ซีพียู (Central Processing Unit)

แต่อย่างไรก็ตาม รีจิสเตอร์ที่เราสามารถใช้ได้ในโปรแกรมของเราจะมีจำนวนจำกัดซึ่งแตกต่างกันไปในแต่ละคอมพิวเตอร์ ดังนั้นถ้าเราได้แจ้งใช้ตัวแปรรีจิสเตอร์ในโปรแกรมโค้ด แต่เมื่อโปรแกรมทำงาน พบว่าไม่มีหน่วยความจำของรีจิสเตอร์ใดๆว่าง ตัวแปรนี้ก็จะใช้หน่วยความจำหลักแทนโดยอัตโนมัติ

ตัวแปรรีจิสเตอร์นี้เราสามารถสังเกตได้จากคำว่า `register` ที่เขียนไว้ข้างหน้าแบบข้อมูลของตัวแปรเมื่อแจ้งใช้ตัวอย่างการใช้ตัวแปรรีจิสเตอร์ เช่น

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

```
int main()
{
   int result;

   /* useful part of program goes here.... */

   {  /* we use registers as temporary variables
         within a block. */

      register unsigned int i;
      register int j = 1;

      for (i=0; i < 65535; i++)
      {
          j = (j * 7) % 31;
      }
      result = j;
   } /* end of block */

   printf ("result = %d\n", result);

   /* another useful part of program goes here.... */

   return 0;
}
```

เหตุที่เรากำหนดให้ตัวแปร `i` และ `j` เป็นตัวแปรรีจิสเตอร์ก็เพราะว่า ในวงวน `for` มีการอ่านข้อมูลจากตัวแปรและเขียนข้อมูลลงในหน่วยความจำของตัวแปรบ่อยครั้ง ดังนั้นการใช้หน่วยความจำของรีจิสเตอร์จะทำให้ความเร็วในการทำงานของโปรแกรมโดยรวมเพิ่มขึ้น (แม้ว่าเราอาจจะไม่รู้สึกว่าโปรแกรมทำงานได้เร็วขึ้นกว่าในกรณีที่ไม่ได้ใช้รีจิสเตอร์ก็ตาม)

ตามปรกติแล้วคอมไพล์เลอร์จะพยายามใช้รีจิสเตอร์ที่ว่างอยู่ ในการคำนวณค่าของนิพจน์ที่ซับซ้อนเพื่อเพิ่มความเร็วในการคำนวณ ดังนั้นถ้าเราแจ้งตัวแปรรีจิสเตอร์หลายๆตัวภายในโปรแกรมและตัวแปรรีจิสเตอร์เหล่านี้มีอายุการใช้งานในโปรแกรมนาน ผลก็คือว่า รีจีสเตอร์จะถูกจองไว้โดยตัวแปร ทำให้จำนวนของรีจิสเตอร์มีน้อยลงทั้งๆที่ในบางช่วงเวลาเราไม่ได้อ่านหรือเขียนค่าของตัวแปรเหล่านี้เลย ในขณะเดียวกันคอมไพล์เลอร์ก็ต้องการใช้รีจิสเตอร์ให้มากที่สุดเพื่อใช้คำนวณค่าของนิพจน์ที่ซับซ้อน ถ้ารีจิสเตอร์ที่ว่างอยู่มีน้อยเกินไปคอมไพล์เลอร์ก็ต้องหันไปใช้หน่วยความจำหลักแทน ในกรณีแย่ที่สุดก็กลับกลายเป็นว่า แทนที่โปรแกรมจะทำงานได้เร็วขึ้น ก็ช้าลง เพราะเหตุที่ว่า เราไปกำหนดจองรีจิสเตอร์ไว้แล้วไม่ได้ใช้อย่างถูกต้อง ดังนั้นควรจะใช้ตัวแปรรีจิสเตอร์เมื่อจำเป็นเท่านั้น และใช้ภายในบล็อกที่มีอายุของการทำงานสั้นๆเท่านั้น ถ้าไม่แน่ใจก็ไม่จำเป็นต้องใช้ และหันไปใช้ตัวแปรอัตโนมัติตามปรกติ

ตามเหตุผลที่ได้กล่าวไปแล้ว เราไม่สามารถใช้คำว่า `register` ร่วมกับ `static` ได้ เพราะถ้าอนุญาตให้ทำเช่นนี้ได้ รีจิสเตอร์ก็จะถูกยึดครองโดยตัวแปรสถิต ซึ่งจะคืนหน่วยความจำในรีจิสเตอร์ก็ต่อเมื่อถูกทำลายไปในตอนจบการทำงานของโปรแกรมเท่านั้น ดังนั้นจึงห้ามกระทำเช่นนี้ และถ้าเราพยายามเขียนคำสั่ง เช่น

```
static register  int    i;
register static  short  j;
```

ก็จะเป็นประโยคคำสั่งที่ผิดหลักไวยกรณ์ในภาษาซี

ข้อควรระวังเกี่ยวกับการใช้ตัวแปรรีจิสเตอร์อีกข้อหนึ่งคือ เราไม่สามารถหาที่อยู่ของตัวแปรรีจิสเตอร์ได้ เพราะตัวแปรรีจิสเตอร์ไม่ได้อยู่ในหน่วยความจำหลักของคอมพิวเตอร์ ดังนั้นจึงไม่ใช้โอเปอร์เรเตอร์ & วางไว้ข้างหน้าตัวแปรรีจิสเตอร์สำหรับหาที่อยู่ของตัวแปรนี้

4.6.4 ตัวแปรภายนอก (GlobalVariable)

การเขียนโปรแกรมโค้ดในภาษาซี เราสามารถแบ่งออกเป็นหลายๆส่วนและเก็บไว้ในไฟล์ต่างกัน โดยเฉพาะในกรณีที่มีนักเขียนโปรแกรมหลายคนทำงานในส่วนต่างๆพร้อมกัน ถ้าในโปรแกรมนี้จำเป็นต้องใช้ตัวแปรภายนอก เมื่อได้ตกลงกันระหว่างนักเขียนโปรแกรมแล้วว่าจะใช้ตัวแปรภายนอกสำหรับจุดประสงค์ใดบ้างและใช้ชื่อใด จากนั้นต่างคนก็เริ่มงานในส่วนของตน ปัญหาก็อยู่ที่ว่าจะแจ้งใช้ตัวแปรภายนอกเหล่านี้ในไฟล์ใด ถ้าต่างคนต่างแจ้งใช้ตัวแปรภายนอกเหล่านี้ในส่วนของตน เมื่อทำการคอมไพล์โปรแกรมโค้ดจากไฟล์ย่อยทั้งหมด ก็จะเป็นไปได้ว่าตัวแปรภายนอกตัวเดียวกันนี้ได้ถูกแจ้งใช้มากกว่าหนึ่งครั้ง ซึ่งผิดหลักไวยกรณ์ของภาษาซี วิธีแก้ไขก็คือการใช้คำว่า `extern` เช่น สมมุติว่า ตัวแปรภาย นอกของเรามีชื่อว่า `ErrorNumber` เมื่อเราแจ้งใช้ตัวแปรนี้ เราก็ทำได้ดังนี้

```
extern int ErrorNumber;
```

คำว่า `extern` แจ้งให้คอมไพล์เลอร์ทราบว่า เราได้แจ้งใช้ตัวแปรชื่อนี้สำหรับแบบข้อมูลลักษณะนี้ ถ้าหาก ว่ามีการแจ้งใช้ตัวแปรลักษณะนี้แล้ว ก็ไม่ต้องแจ้งใช้ตัวแปรนี้อีกและใช้ตัวแปรดังกล่าวได้เลย แต่ถ้าทำการคอมไพล์โปรแกรมโค้ดจากไฟล์ทั้งหมดแล้วพบว่า ยังไม่ได้มีการแจ้งใช้ตัวแปรภายนอกนี้จริงๆ คอมไพล์เลอร์ก็จะเตือนให้เราทราบ ดังนั้นการแจ้งใช้ตัวแปรภายนอกที่ซ้ำซ้อนกันจึงไม่เกิดขึ้น ไม่ว่าจะอยู่ในไฟล์ เดียวกันหรือต่างไพล์กันก็ตาม

ตัวอย่างง่ายๆ ที่แสดงให้เห็นผลของการใช้ extern คือ

```
int ErrorNumber=5; /* global definition of ErrorNumber */

extern int ErrorNumber;

int main ()
{
     extern int ErrorNumber;

     return 0;
}
```

สมมุติว่า เราเขียนประโยคคำสั่งแจ้งใช้ตัวแปร ErrorNumber ทั้งสามประโยคไว้ในไฟล์ของโปรแกรมโค้ดเดียวกัน ในกรณีนี้เราใช้ตัวแปรเพียงตัวเดียวและเป็นตัวแปรภายนอก ถ้าเราลบคำว่า extern ในประโยคที่อยู่ในฟังก์ชันหลักออก ก็จะเป็นการแจ้งใช้ตัวแปรที่ชื่อ ErrorNumber อีกตัวซึ่งทำหน้าที่เป็นตัวแปร ภายในของฟังก์ชันหลัก

คำว่า auto, static, register และ extern รวมเรียกว่า Storage Class Specifier หรือตัวกำหนดชนิดของการเก็บข้อมูลซึ่งเป็นตัวแจ้งให้คอมไพล์เลอร์ทราบว่า จะสร้างและเก็บข้อมูลของตัวแปรอย่างไร เก็บไว้ที่ใดหรือมีระยะเวลานานเท่าใด

นอกจากนี้ภาษาซียังได้นิยามตัวดัดแปลงสำหรับควบคุมการเข้าถึงตัวแปร (Access Modifier) สองตัว คือ const และ volatile ซึ่งใช้กำหนดว่า ตัวแปรสามารถเปลี่ยนแปลงและเข้าถึงได้อย่างไร

4.6.5 ตัวแปรคงที่ (Constant Variable)

ตัวควบคุม const สำหรับตัวแปร เป็นการกำหนดว่า ตัวแปรนี้สามารถอ่านได้อย่างเดียวเท่านั้น ไม่สามารถเปลี่ยนแปลงแก้ไขได้หลังจากที่มีการแจ้งใช้ตัวแปรแบบค่าที่นี้แล้ว ดังนั้นเมื่อเราแจ้งใช้ตัวแปรแบบคงที่จะต้องให้ค่าเริ่มต้นสำหรับตัวแปรด้วย ตัวอย่างการแจ้งใช้ตัวแปรคงที่

```
const int    ConstantNumber = 1000;
const char   ENTER  = 10;
const double SIN_45 = 0.707106781;
const double LN_2   = 0.69314718;
const char*  EMAIL_ADDR = "webmaster@localhost";
```

เราจะเห็นได้ว่า เราสามารถใช้ `const` ได้กับการแจ้งตัวแปรสำหรับข้อมูลพื้นฐานต่างๆ ถ้า `const` ใช้กับตัวแปรที่ทำหน้าที่เป็นพอยน์เตอร์ ก็จะหมายความว่า เราไม่สามารถเปลี่ยนแปลงค่าของข้อมูลตามที่อยู่ที่ พอยน์เตอร์นี้อ้างอิงได้

4.6.6 ตัวแปรที่อาจจะลบเลือนได้ (Volatile Variable)

คำว่า `volatile` แจ้งให้คอมไพล์เลอร์ทราบว่า ค่าของตัวแปรชนิดนี้สามารถเปลี่ยนแปลงได้ ซึ่ง ไม่จำเป็นต้องเกิดขึ้นโดยการกระทำใดๆของคำสั่งในโปรแกรม เช่น อาจจะเกิดขึ้นจากการทำงานของระบบปฏิบัติการของคอมพิวเตอร์ก็ได้ ลองยกตัวอย่างง่ายๆ สมมุติว่าเราต้องการผ่านที่อยู่ของตัวแปรให้แก่ระบบปฏิบัติการเพื่อหาเวลาของนาฬิกาในคอมพิวเตอร์ขณะนั้น ในกรณีนี้ค่าของตัวแปรจะถูกเปลี่ยนแปลง โดยขั้นตอนการทำงานของระบบปฏิบัติการและไม่ได้เกี่ยวข้องกับขั้นตอนใดๆของโปรแกรม คอมไพล์เลอร์บางตัวเมื่อพบว่าไม่มีคำสั่งของโปรแกรมในช่วงระยะเวลาดังกล่าวที่เปลี่ยนแปลงค่าของตัวแปรก็อาจจะพยายามเปลี่ยนแปลงลำดับการทำงานของคำสั่งเพื่อที่จะทำให้โปรแกรมทำงานได้เร็วขึ้นโดยมิได้ทราบว่า ถ้าเปลี่ยนลำดับการทำงานของคำสั่งเหล่านั้นแล้วจะทำให้เกิดผลข้างเคียงที่ตามมา เช่น อ่านเวลาของนาฬิกาไม่ถูกต้องตามช่วงเวลาที่ต้องการ ดังนั้นเพื่อเป็นการเตือนให้คอมไพล์เลอร์คำนึงถึงลำดับของขั้นตอน หรือคำสั่งตามที่เขียนไว้ในโปรแกรมโค้ด เราก็จะใช้คำว่า `volatile` บอกกำกับ

4.7 ขอบเขตของการใช้ตัวแปร

เราสามารถจำแนกตัวแปรที่ใช้ในโปรแกรมออกได้เป็นสองจำพวก โดยแบ่งตามตำแหน่งของการแจ้งใช้ตัวแปรได้แก่ ตัวแปรที่แจ้งใช้ภายในฟังก์ชันหรือบล็อก และตัวแปรที่แจ้งใช้ภายนอกฟังก์ชัน

ตัวแปรที่แจ้งใช้ภายในฟังก์ชัน เรามักจะเรียกว่า *ตัวแปรเฉพาะที่* (Local Variable) ตัวแปรชนิดนี้มีตัวตนหรือสามารถทำงานได้เฉพาะภายในฟังก์ชันเท่านั้น เนื่องจากว่า ฟังก์ชันมีโครงสร้างเป็นแบบบล็อก ดังนั้นเราสามารถกล่าวได้ว่าตัวแปรใดๆภายในบล็อกก็จัดได้ว่าเป็นตัวแปรเฉพาะที่ ตัวแปรเฉพาะที่นั้นจะ“ถูกสร้างขึ้น”ในเวลาที่โปรแกรมทำงานและเริ่มเข้าไปในส่วนของบล็อกดังกล่าวเท่านั้นและจะ“ถูกทำลายไป”เมื่อโปรแกรมจบการทำงานของแต่ละบล็อก เช่น เวลาเราเรียกใช้ฟังก์ชันใดๆเมื่อฟังก์ชันเริ่มทำงานถ้ามีการแจ้งใช้ตัวแปรเฉพาะที่ โปรแกรมก็จะสร้างตัวแปรเหล่านี้ขึ้นมาโดยมีการกำหนดหน่วยความจำของคอมพิวเตอร์ไว้สำหรับเก็บค่าของตัว

แปรซึ่งมีขนาดแตกต่างกันไปขึ้นอยู่กับแบบข้อมูลของตัวแปร และหลังจากที่ตัวแปรเฉพาะที่ถูกสร้างขึ้นแล้วโปรแกรมสามารถใช้ตัวแปรเหล่านี้ได้ เมื่อการทำงานของโปรแกรมยังคงอยู่ภายในบล็อกของฟังก์ชัน ถ้าภายในบล็อกของฟังก์ชันนี้มีการแบ่งออกเป็นบล็อกย่อยๆลงไปอีกและมีการแจ้งใช้ตัวแปรเฉพาะที่เพิ่มเติมสำหรับบล็อกย่อยแต่บล็อก โปรแกรมก็จะสร้างตัวแปรขึ้น เมื่อใดที่โปรแกรมจบการทำงานของบล็อกหรือออกจากบล็อกใดๆ ตัวแปรเฉพาะที่สำหรับบล็อกนั้นก็จะ“ถูกทำลาย”ไป (ยกเว้นสำหรับกรณีที่ ตัวแปรเป็นตัวแปรสถิตหรือ Static Variable) ดังนั้นเมื่ออยู่นอกขอบเขตของบล็อกใด เราจึงไม่สามารถใช้ตัวแปรเฉพาะที่ภายในบล็อกนั้นได้ เพราะเราจะ“ไม่สามารถมองเห็น”ตัวแปรเหล่านั้นได้ เปรียบเสมือนว่าตัวแปรนั้น“ถูกปิดบังด้วยกำแพงของบล็อก”ดังกล่าว ตัวอย่างเช่น

```
#include <stdio.h>

int main()
{
      int x;

      x = 5;
      {
            double x;

            x = 2.5;
            printf("x = %lf\n", x);
      }
      printf("x = %d\n", x);
}
```

ผลของโปรแกรมคือ

```
x = 2.500000
x = 5
```

เราจะเห็นได้ว่า ในฟังก์ชัน `main()` มีการแจ้งใช้ตัวแปรเฉพาะที่ `x` สองครั้ง แบ่งออกเป็นสองระดับคือ ตัวแปร `x` แบบ `int` ที่ใช้ภายใน (บล็อกของ) ฟังก์ชัน แต่อยู่นอกบล็อกย่อย ส่วนตัวแปร `x` แบบ `double` นั้นใช้ภายในฟังก์ชันและภายในบล็อกย่อยเท่านั้น เราจะเห็นได้ว่าเราใช้ตัวแปรเฉพาะที่สองตัวที่มีชื่อเหมือนกันแต่ใช้อยู่ภายใน “ขอบเขตของการทำงาน”ที่แตกต่างกันหรือ *ต่างระดับกัน*

ตามปรกติแล้ว ถ้าตัวแปรสองตัวอยู่ในขอบเขตของการทำงานเดียวกัน(บล็อกเดียวกันและระดับเดียวกัน) ตัวแปรทั้งสองห้ามมีชื่อเหมือนกัน เราจะเห็นได้ว่าการใช้เครื่องหมายวงเล็บปีกกาคู่

{} นี้เป็นการสร้างบล็อกและเป็นการแบ่งขอบเขตของการทำงานให้อยู่ในระดับลึกลงไปอีก สรุปคือว่า เราสามารถใช้ตัวแปรหลายตัวที่มีชื่อเหมือนกันและต่างแบบข้อมูลได้ แต่ตัวแปรที่มีชื่อเหมือนกันนี้จะต้องอยู่จะต่างระดับกัน

ตัวอย่างอีกตัวอย่างหนึ่งที่แสดงให้เห็นขอบเขตของการทำงานของตัวแปรแต่ละตัวที่อยู่ต่างระดับกันและมีชื่อแตกต่างกันคือ

```
void function_one()
{
     int x;

     x = 5;
     {
          double y;

          y = x + 2.5;          /* O.K. */
          printf("x = %lf\n", y);
     }
     y = x - 2.5;               /* Illegal */
     printf("x = %d\n", x);
}
```

ในตัวอย่างนี้ตัวแปร y อยู่ในระดับที่ต่ำกว่าระดับของ x เราสามารถใช้ตัวแปร x ในระดับของตัวแปร y แต่ในทางกลับกัน เราไม่สามารถใช้ตัวแปร y ในระดับเดียวกันกับ x ได้ เพราะอยู่นอกขอบเขตของบล็อก ที่ตัวแปร y ได้ถูกแจ้งใช้ เมื่อโปรแกรมจบการทำงานบล็อกย่อย ตัวแปร y จะถูกทำลายไปโดยอัตโนมัติ ดังนั้นเราจึงไม่สามารถใช้ตัวแปร y ในประโยคคำสั่ง

```
y = x - 2.5;
```

นี้ได้

ตัวแปรอีกชนิดหนึ่งคือ ตัวแปรที่แจ้งใช้ภายนอกฟังก์ชันหรือเรียกสั้นๆว่า ตัวแปรภายนอก (Global Variable) ตัวแปรชนิดนี้ใช้ได้ตลอดทั่วทั้งโปรแกรม ไม่ว่าจะในฟังก์ชันหลักหรือในฟังก์ชันอื่นๆ ข้อดีสำหรับการใช้ตัวแปรภายนอกคือ เราสามารถแจ้งใช้ตัวเพียงครั้งเดียว เช่น แจ้งใช้ตัวแปรไว้ในตอนต้นของโปรแกรมโค้ด และสามารถใช้ตัวแปรนี้ได้ในทุกๆส่วนของโปรแกรมโค้ดนับตั้งแต่บรรทัดที่เราแจ้งใช้ตัวแปรนี้เป็นต้นไป แต่การใช้ตัวแปรภายนอกนั้นเปรียบเสมือนกับดาบสองคม ซึ่งมีข้อเสียคือ ถ้าเราใช้ตัวแปรภายนอกฟังก์ชันใดๆในโปรแกรมสามารถเข้าถึงตัวแปรนี้ได้ ดังนั้นจึงเป็นการยากที่จะตรวจสอบได้ว่าค่าของตัวแปรเป็นเท่าไหรและถูกเปลี่ยนแปลงโดยคำสั่งใดในส่วนใดของโปรแกรมโค้ดและในเวลาใดบ้างโดยเฉพาะอย่างยิ่งเมื่อโปรแกรมโค้ดมีความยาวและซับซ้อนมาก ดังนั้นถ้าไม่จำเป็นเราก็ควรจะหลีกเลี่ยงการใช้ ตัวแปรภายนอก

4.8 พารามิเตอร์สำหรับฟังก์ชันหลัก

เนื่องจากว่าฟังก์ชันหลักเป็นฟังก์ชันหนึ่งในภาษาซีและเป็นจุดเริ่มต้นของโปรแกรม บางครั้งเราต้องการผ่านข้อมูลไปยังโปรแกรมเมื่อเรารันโปรแกรมนี้ เราสามารถทำได้โดยใช้อาร์กิวเมนท์ในบรรทัด คำสั่งงาน (Command Line Argument) ลองนึกถึงคำสั่ง FORMAT ของดอส (DOS) เมื่อเราต้องการให้คอมพิวเตอร์จัดรูปแบบของหน่วยความจำบนแผ่นดิสก์ใหม่ เราก็เขียนคำสั่งว่า

```
C:\>FORMAT \Q A:
```

ในตัวอย่างนี้ \Q และ A: เป็นอาร์กิวเมนต์สองตัวของคำสั่งงานที่เราป้อนให้คอมพิวเตอร์

สำหรับโปรแกรมใดๆที่เขียนในภาษาซี เราสามารถอ่านอาร์กิวเมนท์ของคำสั่งงานเหล่านี้ได้จากระบบปฏิบัติการของคอมพิวเตอร์เมื่อโปรแกรมเริ่มทำงาน โปรแกรมตัวอย่างต่อไปนี้จะพิมพ์อาร์กิวเมนท์ ต่างๆของโปรแกรมออกจากจอภาพ

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

int main (int argc, char *argv[])
{
      int i;

      for (i=0; i < argc; i++)
      {
            printf("(%02d) %s\n", i+1, argv[i]);
      }
      return 0;
}
```

ตัวแปรสองตัวชื่อ argc และ argv ทำหน้าที่เป็นอาร์กิวเมนท์ของฟังก์ชันหลัก ตัวแปร argc (ย่อมาจาก Argument Count) ใช้เก็บค่าที่เท่ากับจำนวนของอาร์กิวเมนท์ของคำสั่งงาน ซึ่งเป็นข้อมูลแบบ int และมีค่า มากกว่าหรือเท่ากับหนึ่ง ถ้าไม่มีอาร์กิวเมนท์ใดๆที่ผ่านให้โปรแกรม ค่าของ argc จะเท่ากับหนึ่งเพราะชื่อของโปรแกรมจะถือว่าเป็นอาร์กิวเมนท์ตัวแรก ตัวแปร argv เป็นพอนย์เตอร์ทำหน้าที่ชี้ไปยังหน่วยความจำที่เก็บอาร์กิวเมนต์ของโปรแกรมซึ่งแหล่งข้อมูลนี้จะสร้างไว้โดยระบบปฏิบัติการ

4.9 การสร้างฟังก์ชันที่มีจำนวนพารามิเตอร์ไม่แน่นอน

บางครั้งเราต้องการสร้างฟังก์ชันใดๆที่สามารถมีจำนวนพารามิเตอร์ไม่แน่นอน เช่น สมมุติว่าเรา ต้องการเขียนฟังก์ชันในการหาผลรวมของพารามิเตอร์ที่เป็นเลขจำนวนเต็มแบบ int ตั้งแต่สองตัวขึ้นไป

ตามปรกติแล้วฟังก์ชันทั่วๆไปในภาษาซีจะต้องมีจำนวนของพารามิเตอร์ที่คงที่ แต่อย่างไรก็ตาม ยังมีวิธีการที่เราสามารถใช้นิยามฟังก์ชันที่มีจำนวนของพารามิเตอร์ที่แตกต่างกันไปตามเวลาที่เรียกใช้ และสามารถทำได้โดยอาศัยแมโคร(Macro) ที่นิยามไว้ใน <stdarg.h> คือ va_start() va_arg() และ va_end()

```
void  va_start( va_list argument_list, last_parameter);
data_type va_arg( va_list argument_list, data_type);
void  va_end( va_list argument_list);
```

ซึ่ง va บ่งบอกถึง Varying-argument List เพื่อที่จะทำให้เข้าใจได้ง่ายขึ้นเราจะพิจารณตัวอย่างต่อไปนี้

เราต้องการสร้างฟังก์ชันที่หาผลรวมของตัวเลขที่เป็นพารามิเตอร์แบบ int และกำหนดไว้ว่า เวลาเรียกใช้จะต้องมีพารามิเตอร์อย่างน้อยหนึ่งตัวที่บ่งบอกว่ามีพารามิเตอร์ที่อยู่ถัดไปอีกทั้งหมดกี่ตัว รูปแบบของฟังก์ชันจะเป็นดังนี้

```
int IntSum (int numOfParams, ...);
```

ฟังก์ชันนี้จะต้องมีพารามิเตอร์อย่างน้อยหนึ่งตัว ซึ่งก็คือ numOfParams

---

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

int IntSum (int numOfParams, ...)
{
     int i,sum=0;
     va_list arg_list;

     va_start(arg_list, numOfParams);
     for (i=0; i < numOfParams; i++)
     {
          int arg = va_arg(arg_list, int);
          printf("%3d. Parameter\t%5d\n", i+1, arg);
          sum += arg;
     }
     va_end(arg_list);

     return sum;
```

```
}

int main()
{
      int a = 10, b = 101;
      int sum1, sum2, sum3;

      sum1 = IntSum(2, a, b);
      printf("---------------------\n");

      sum2 = IntSum(3, 3*sum1-10, 111 ,10-3*sum1);
      printf("---------------------\n");

      sum3 = IntSum(5, 3, 0L, 5, 1, 7);
      printf("---------------------\n");

      printf("Sum1 = %d\n", sum1);
      printf("Sum2 = %d\n", sum2);
      printf("Sum3 = %d\n", sum3);
      return 0;
}
```

va_list เป็นแบบข้อมูลซับซ้อนที่นิยามไว้ใน <stdarg.h> ดังนั้นเราสามารถนิยามตัวแปร arg_list ให้มีแบบข้อมูลเป็น va_list ตามตัวอย่าง หลังจากที่ได้แจ้งใช้ตัวแปร arg_list แล้วขั้น ต่อไปคือการเรียกใช้ va_start() โดยมีพารามิเตอร์เป็น arg_list และ numOfParams ตามลำดับ โปรดสังเกตว่า พารามิเตอร์ตัวที่สองของ va_start() จะเป็นพารามิเตอร์ตัวสุดท้ายของฟังก์ชันก่อนที่เราจะเริ่มนับว่า มีพารามิเตอร์ที่ตามมาอีกกี่ตัว ซึ่งหมายถึงพารามิเตอร์ตัวที่อยู่ก่อนหน้า ... ขั้นตอนต่อไป คือการอ่านพารามิเตอร์ทีละตัวจากตัวแปร va_list โดยเรียกใช้ va_arg() ภายในวงวน for พารามิเตอร์ตัวที่สองของ va_arg() จะต้องเป็นแบบข้อมูลของพารามิเตอร์แต่ละตัว ในกรณีตัวอย่างนี้เรากำ-หนดไว้ ว่าพารามิเตอร์แต่ละตัวเป็นตัวแปรหรือค่าคงที่แบบ int เหมือนกันหมด (ในกรณีพารามิ-เตอร์ต่างแบบกัน จะทำให้เกิดปัญหาในการอ่านพารามิเตอร์แต่ละตัว) ดังนั้นเราจึงใช้คำว่า int และขั้นตอนสุดท้ายคือ การเรียกใช้ va_end() เป็นอันว่าจบขั้นตอนของการจัดการกับพารามิเตอร์ของฟังก์ชัน IntSum()ที่มีจำนวนไม่คงที่ ผลของโปรแกรมที่แสดงออกทางจอภาพจะเป็นดังนี้

```
 1. Parameter       10
 2. Parameter      101
--------------------
 1. Parameter      323
 2. Parameter      111
 3. Parameter     -323
---------------------
 1. Parameter        3
 2. Parameter        0
 3. Parameter        0
 4. Parameter        5
```

```
 5. Parameter        1
---------------------
Sum1 = 111
Sum2 = 111
Sum3 = 9
```

ถ้าสังเกตให้ดี โปรแกรมให้ผลลัพธ์ที่ไม่ถูกต้องสำหรับ `Sum3` เพราะเราได้ใช้พารามิเตอร์ `0L` สำหรับฟังก์ชัน `IntSum()` เนื่องจากว่า `0L` เป็นค่าคงที่แบบ `long int` ซึ่งมีขนาดมากกว่า `int` ส่งผลให้การอ่านค่าพารามิเตอร์โดย `va_arg()` จึงเป็นไปอย่างไม่ถูกต้อง ดังนั้นการผ่านค่าพารามิเตอร์ที่มีแบบข้อมูลแตกต่างจากที่ได้นิยามไว้จะทำให้เกิดปัญหาได้เพราะ `va_arg()` เป็นแมโครมิใช่เป็นฟังก์ชัน ดังนั้นการเปลี่ยนแปลงแบบข้อมูลโดยอัตโนมัติจึงไม่จำเป็นต้องเกิดขึ้น

4.10 รีเคอร์ชันหรือฟังก์ชันเรียกใช้ตัวเอง

ในภาษาซีการสร้างโปรแกรมมักจะเกี่ยวข้องกับการสร้างและเรียกใช้ฟังก์ชัน เราได้เห็นตัวอย่างการเรียกใช้ฟังก์ชันโดยฟังก์ชันแต่เป็นฟังก์ชันสองตัวที่ไม่ใช่ตัวเดียวกัน การเรียกใช้ฟังก์ชันอีกแบบหนึ่ง คือ *การเรียกใช้ตัวเอง* หรือ Recursion บางครั้งก็เรียกว่า การเรียกซ้ำ หรือ เวียนซ้ำ

การเรียกตัวเองซ้ำมีลักษณะที่สำคัญคือ ผลของการเรียกใช้ฟังก์ชันจะขึ้นอยู่กับผลของการเรียกใช้ในครั้งต่อไป เช่น ผลของการเรียกใช้ครั้งแรกจะขึ้นอยู่กับผลของการเรียกใช้ครั้งที่สอง และผลของการเรียกใช้ครั้งสองก็ขึ้นอยู่กับผลของการเรียกใช้ครั้งต่อๆไปด้วย ดังนั้นถ้าต้องการจะหาผลการทำงานของฟังก์ชัน ซึ่งเป็นครั้งแรกที่เรียกใช้ฟังก์ชัน ก็ต้องทราบผลของการทำงานครั้งต่อไปข้างหน้าก่อน

แน่นอนเราไม่ต้องการให้การทำงานของฟังก์ชันมีการเรียกใช้ตัวเองแบบไม่รู้จบ เช่นสมมุติว่าฟังก์ชันจะต้องมีการเรียกใช้ตัวเองเป็นจำนวน `n` ครั้งเท่านั้นโดยที่เป็นจำนวนนับใดๆที่จำกัด เราสามารถเขียนความสัมพันธ์ของการเรียกใช้ฟังก์ชัน ตั้งแต่ครั้งแรกไปจนถึงครั้งที่ `n` ได้ในเชิงคณิตศาสตร์

$$f(i) = g(i,\ f(i-1)) \qquad i = 1,2,\ldots,n$$

`f` และ `g` เป็นฟังก์ชันสองฟังก์ชันใดๆ ในขณะที่ค่าของ `f` ขึ้นอยู่กับตัวแปร `i` เท่านั้น การหาค่าของ `g` ต้องอาศัยทั้งค่าของ `i` และ ค่าของฟังก์ชัน `f(i-1)` ดังนั้นถ้าเราทราบค่าเริ่มต้น `f(0)` เราก็สามารถคำนวณค่าของ `f(i)`ได้จากฟังก์ชัน `f` และ `g`

ตัวอย่างง่ายๆที่เราสามารถเขียนให้อยู่ในรูปของความสัมพันธ์ดังที่กล่าวไปในข้างต้นคือการคำนวณค่าของแฟคทอเรียล (Factorial) ในทางคณิตศาสตร์และมีสัญลักษณ์เป็น n!

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n > 0 \end{cases}$$

หรือถ้าเราเขียนชื่อฟังก์ชันว่า Fac( ) ซึ่งย่อมาจาก Factorial และ ใช้แทนที่สัญลักษณ์ของแฟคทอเรียล ก็จะเขียนได้ดังนี้

$$Fac(n) = \begin{cases} 1 & n = 0 \\ n \cdot Fac(n-1) & n > 0 \end{cases}$$

เช่น ถ้าเราต้องการจะคำนวณค่าของ Fac(6) เราก็จะต้องทราบค่าของ Fac(5) ก่อน และในเวลาเดียวกัน เราก็ต้องทราบว่า Fac(4) มีค่าเท่าไหรเพื่อใช้ประกอบในการคำนวณค่าของ Fac(5) ซึ่งมีเงื่อนไขในทำ-นองเดียวกันสำหรับการคำนวณ Fac(3), Fac(2), Fac(1) จนถึง Fac(0) = 1 ดังนั้นเมื่อเราทราบ ค่าของ Fac(0) แล้ว เราก็สามารถย้อนกลับไปคำนวณค่าของ Fac(1), Fac(2),..., Fac(6) ได้ตามลำดับ ท้ายสุดเราก็จะได้ค่าของ Fac(6) เท่ากับ

$$Fac(6) = 6 \times 5 \times 4 \times 3 \times 2 \times 1 \times 1$$

เราจะเห็นได้ว่าความสัมพันธ์ของฟังก์ชัน Fac(n) นี้ เขียนอยู่ในรูปของรีเคอร์ชันหรือการทำงานแบบเรียกตัวเองอยู่แล้ว ดังนั้นเราสามารถถ่ายทอดลักษณะการทำงานของฟังก์ชันทางคณิตศาสตร์ให้อยู่ในรูปแบบ ของฟังก์ชันภาษาซีได้ดังนี้

```
/* assume n is a non-negative integer */
int Fac(int n)
{
      return (n ? n*Fac(n-1) : 1);
}
```

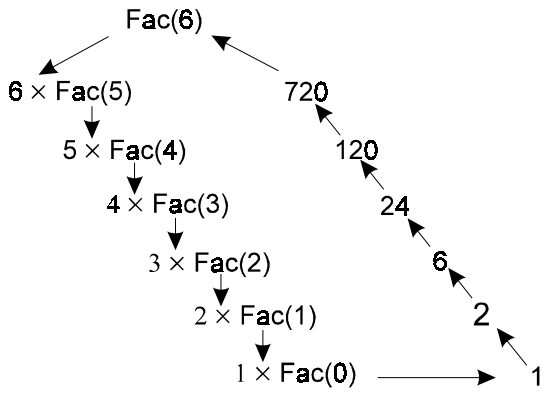
ถ้าผู้อ่านท่านใดคิดว่า ฟังก์ชันนี้เมื่อมองดูแล้วรู้สึกว่าสั้นเกินไป มองดูแล้วเข้าใจยาก ก็ให้ลองเปรียบเทียบกับอีกแบบหนึ่งคือ

```
/* assume n is a non-negative integer */
int Fac (int n)
{
      int answer;

      if (n==0) {
            answer = 1;
      }
```

```
        else {
              answer = n * Fac(n-1);
        }
        return answer;
}
```

ซึ่งให้ผลเหมือนกัน การตรวจสอบเงื่อนไขของ `if` ถือว่าสำคัญเพราะเป็นกลไกควบคุมการหยุดการเรียกใช้ ตัวเอง พารามิเตอร์ n เป็นเลขจำนวนเต็มที่มากกว่าหรือเท่ากับศูนย์ เมื่อ n มีค่ามากกว่าศูนย์ ฟังก์ชันก็จะเรียกใช้ตัวเองต่อไปโดยการเรียกตัวเองแต่ละครั้งจะลดค่าของพารามิเตอร์ลงทีละหนึ่งจนกว่าพารามิเตอร์จะมีค่าเท่ากับศูนย์ เมื่อไม่มีการเรียกใช้ตัวเองอีกต่อไป โปรแกรมก็จะย้อนกลับไปทำงานภายในฟังก์ชันที่เคยเรียกใช้แล้วแต่ยังมิได้จบการทำงานตามลำดับจนถึงฟังก์ชันที่ถูกเรียกใช้ในครั้งแรกสุด

$$
\begin{aligned}
Fac(6) &= (6 * Fac(5)) \\
&= (6 * (5 * Fac(4))) \\
&= (6 * (5 * (4 * Fac(3)))) \\
&= (6 * (5 * (4 * (3 * Fac(2))))) \\
&= (6 * (5 * (4 * (3 * (2 * Fac(1)))))) \\
&= (6 * (5 * (4 * (3 * (2 * (1 * (1))))))) \\
&= (6 * (5 * (4 * (3 * (2 * 1))))) \\
&= (6 * (5 * (4 * (3 * 2)))) \\
&= (6 * (5 * (4 * 6))) \\
&= (6 * (5 * 24)) \\
&= (6 * 120) \\
&= 720
\end{aligned}
$$



รูปภาพที่ 4.4 ภาพแสดงการคำนวณค่าแฟคทอเรียลจากฟังก์ชันเรียกตัวเอง

ผู้อ่านบางคนอาจจะตั้งคำถามว่า เมื่อได้เห็นความสัมพันธ์ของตัวเลขต่างๆแล้ว ทำไมเราไม่คำนวณค่าของ Fac(6)โดยเริ่มจากข้างล่างขึ้นบน(คือเริ่มจากตัวที่เราทราบค่าแล้ว เช่น เริ่มจาก Fac(0) แล้วก็คำนวณค่าของ Fac(1) ไปตามลำดับ) แทนที่จะเริ่มคำนวณจากบนลงล่างแล้วก็ต้องย้อนกลับขึ้นข้างบนอีกตามแผนภาพการคำนวณ เช่น โดยการสร้างวงวนแบบ for

```
int Fac(int n)
{
      int i, fac=1;

      for (i=1; i<=n; i++)
            fac *= i;
      return fac;
}
```

หรืออาจจะเขียนใหม่ได้อีกแบบหนึ่งคือ

```
int Fac(int n)
{
      int fac=1;
      for (;n;fac*=n--);
      return fac;
}
```

ในตัวอย่างนี้การคำนวณค่าของแฟคทอเรียล เราสามารถทำได้โดยใช้วงวน และไม่จำเป็นต้องเขียนฟังก์ชัน ในลักษณะที่เรียกใช้ตัวเองซึ่งการคำนวณโดยใช้วงวนจะคำนวณได้มีประสิทธิภาพมากกว่า แต่อย่างไรก็ตาม สำหรับการแก้ไขปัญหาบางอย่างในบางครั้งก็เป็นการยากที่จะคิดหาหนทางหรือวิธีการที่สามารถเขียนให้อยู่ในรูปของวงวนได้ แต่จะง่ายกว่าเมื่อสร้างขั้นตอนการแก้ปัญหาที่เรียกใช้ตัวเอง

ตัวอย่างต่อไปแสดงให้เห็นว่า เมื่อมีการเรียกใช้ตัวเองแล้วขั้นตอนการทำงานของคำสั่งต่างๆภายในฟังก์ชัน จะถูกดำเนินการก่อนหลังอย่างไร ลองพิจารณาสองฟังก์ชันที่เรียกใช้ตัวเองและมีลักษณะเหมือนกัน เพียงแต่มีข้อแตกต่างอยู่ตรงที่ลำดับการทำงานก่อนหลังของคำสั่ง printf() และคำสั่งเรียกใช้ตัวเอง

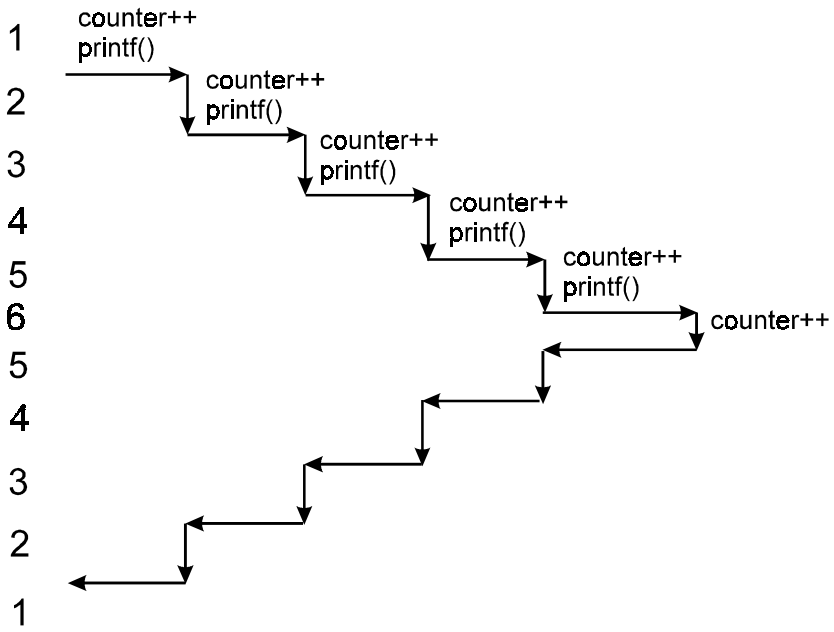
```
#include <stdio.h>

void recursion1()
{
      static int counter=0;
      if(counter++ < 5) {
            printf("\tFunction Call : %d\n", counter);
            recursion1();
```

```
        }
    }

    void recursion2()
    {
        static int counter=0;
        if(counter++ < 5) {
            recursion2();
            printf("\tFunction Call : %d\n", counter);
        }
    }

    int main()
    {
        printf("Recursion1\n");
        recursion1();
        printf("Recursion2\n");
        recursion2();

        return 0;
    }
```

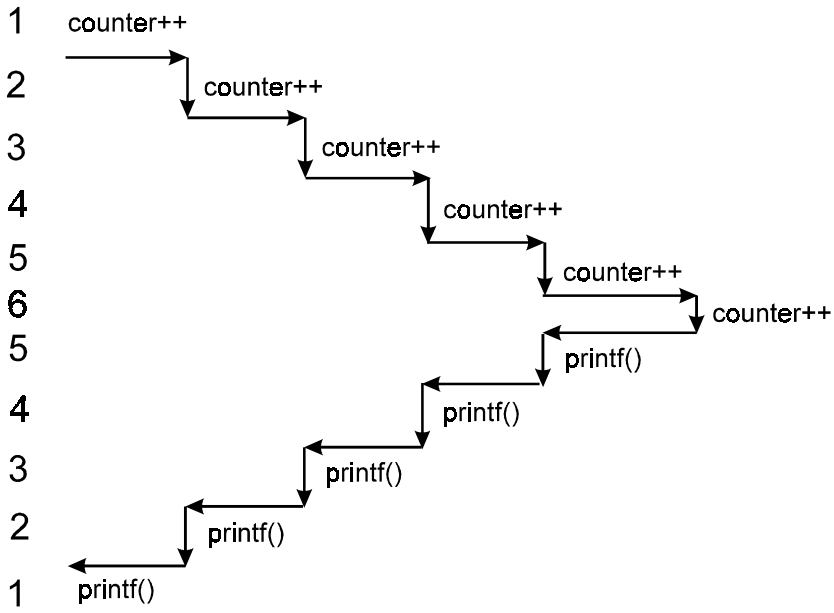
ผลของโปรแกรมคือ

```
Recursion1
        Function Call : 1
        Function Call : 2
        Function Call : 3
        Function Call : 4
        Function Call : 5
Recursion2
        Function Call : 6
        Function Call : 6
        Function Call : 6
        Function Call : 6
        Function Call : 6
```

ภายในฟังก์ชันเราใช้ตัวแปรเฉพาะที่แบบสถิตชื่อ counter ดังนั้นฟังก์ชันเมื่อเรียกตัวเองจะใช้ตัวแปรนี้ ร่วมกัน และในการเรียกตัวเองแต่ละครั้งค่าของตัวแปรจะเพิ่มขึ้นทีละหนึ่ง เมื่อตัวแปรนี้มีค่าเกิน 5 จึงหยุดเรียกซ้ำ บล็อกของฟังก์ชันที่ถูกเรียกใช้ครั้งล่าสุดจะไม่มีการเรียกใช้ฟังก์ชันของตัวเองอีกต่อไป และเมื่อจบการทำงานของบล็อกก็จะย้อนกลับไปกระทำคำสั่งต่อในบล็อกของฟังก์ชันที่ถูกเรียกก่อนหน้านี้ โดยย้อนกลับไปตามลำดับ



รูปภาพที่ 4.5 แผนภาพแสดงการกระทำคำสั่งภายในฟังก์ชัน `recursion1()`



รูปภาพที่ 4.6 แผนภาพแสดงการกระทำคำสั่งภายในฟังก์ชัน `recursion2()`

ภายในฟังก์ชัน `recursion1()` คำสั่ง `printf()` จะถูกดำเนินการก่อนที่จะมีการเรียกตัวเองในครั้งต่อไป ดังนั้นจึงแสดงค่าของตัวแปร `counter` ในขณะนั้นออกทางจอภาพโดยทันที ในขณะที่ฟังก์ชัน `recursion2()` มีคำสั่ง `printf()` อยู่หลังคำสั่งที่เรียกใช้ฟังก์ชัน ผลก็คือว่า ค่าของตัวแปร `counter` จะเพิ่มขึ้นเรื่อยๆเมื่อมีการเรียกตัวเองแต่ละครั้ง และเมื่อ `counter` มีค่าเกิน 5 ซึ่งในที่นี้ `counter` จะมีค่า เท่ากับ 6 ก็จะไม่มีการเรียกใช้ตัวเองอีกต่อไป ดังนั้นโปรแกรมจึงย้อนกลับไปทำคำสั่ง `printf()`ที่คั่งค้างอยู่ในคราวที่ผ่านมา เนื่องจากว่าไม่มีขั้นตอนใดที่เหลือที่

เปลี่ยนแปลงค่าของ counter ที่ได้จากการเรียกใช้ฟังก์ชันครั้งท้ายสุด ดังนั้นฟังก์ชัน printf() จึงพิมพ์ค่าของตัวแปรนี้ในแต่ละครั้งเท่ากับ 6 ออกทางจอภาพ

แบบฝึกหัดท้ายบท

1. จงหาค่าของตัวแปร x ในขั้นตอนการทำงานต่างๆของโปรแกรมต่อไปนี้

```
#include <stdio.h>

int f1(int x)
{
     return x = 10;
}

int main()
{
     const int x = 1;
     int i;

     {
          int x=0;
          x += f1(x);
     }

     for (i=10; i; --i)
     {
          static double x=(double)(i*i);
     }
     return 0;
}
```

ในโปรแกรมโค้ดนี้มีอยู่ที่ผิดอยู่หนึ่งแห่ง ที่มีการแจ้งใช้ตัวแปร x ที่ผิดหลักไวยกรณ์ จงหาคำสั่งที่ผิด และให้เหตุผลว่า ทำไมจึงผิด

2. โปรแกรมโค้ดต่อไปนี้มีที่ผิดซึ่งเกี่ยวข้องกับการใช้คำว่า extern กับการแจ้งใช้ตัวแปร จงหาตำแหน่งที่ ผิดในโปรแกรม

```
#include <stdio.h>

int i = 5;
extern double j = 3.0;

int main()
{
     extern double j = 3.0;
     {
        extern unsigned int k;
     }
     return 0;
}
```

3. จงเขียนฟังก์ชันที่ใช้หาผลคูณแบบสกาลาร์ (Scalar Product) ของเวกเตอร์สองตัว

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \cdot \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = x_1 x_2 + y_1 y_2$$

โดยมีรูปแบบของฟังก์ชันดังนี้

```
double scalarProduct (double x1, double y1,
                      double x2, double y2);
```

4. ลองคิดว่า ถ้าเรียกใช้ฟังก์ชัน `f1();` ที่นิยามไว้ตามรูปแบบข้างล่างนี้ในโปรแกรมแล้วผลที่เกิดขึ้นจะเป็นอย่างไร

```
extern void f1(void);
extern void f2(void);
extern void f3(void);

void f1(void)
{
     static int i=3;
     printf("+f1\n");
     if (--i)
          f2();
     printf("-f1\n");
}

void f2(void)
{
     printf("+f2\n");
     f3();
     printf("-f2\n");
}

void f3(void)
{
     printf("+f3\n");
     f1();
     printf("-f3\n");
}
```

5. จงอธิบายการทำงานของฟังก์ชันแบบเรียกตัวเองสองฟังก์ชันต่อไปนี้ และโปรดสังเกตความแตกต่างของสองฟังก์ชันนี้

```
void  P(int n)
{ /* assume n is a non-negative integer */
     if (n < 10) {
          printf("%d", n);
     }
     else {
          P(n/10);
          printf("%d", n%10);
     }
```

```
        return;
}

void  R(int n)
/* assume n is a non-negative integer */
{
        printf("%d", n % 10);
        if ((n = n/10) != 0)
                R(n);
        return;
}
```

6. ในการคำนวณค่าของสัมประสิทธิ์ไบโนเมียล (Binomial Coefficient)

$$C(n,k) \quad = \quad \binom{n}{k} \quad = \quad \frac{n!}{k!(n-k)!}$$

เราสามารถเขียนฟังก์ชันแบบเรียกตัวเองโดยใช้สูตรที่แสดงความสัมพันธ์ของสัมประสิทธิ์ต่อไปนี้ได้

$$\begin{aligned} C(n,0) &= 1, \qquad n \geq 0 \\ C(n,n) &= 1, \qquad n \geq 0 \\ C(n,k) &= C(n-1,k) + C(n-1,k-1), \quad n > k > 0 \end{aligned}$$

จงเขียนฟังก์ชันภาษาซีที่ใช้คำนวณค่าของ `C(n,k)` เมื่อ `n` และ `k` เป็นจำนวนเต็มที่มากกว่าหรือเท่ากับศูนย์

7. ในการหาค่าของตัวหารร่วมมาก (the greatest common divisor) ของตัวเลขจำนวนเต็มบวกสองจำนวน เราสามารถใช้อัลกอริทึมต่อไปนี้ได้

$$\gcd(m,n) = \begin{cases} u, & v = 0 \\ \gcd(u \bmod v,\ v), & v > 0 \end{cases}$$

$$u \equiv \max\{m,n\} \qquad v \equiv \min\{m,n\}$$

เนื่องจากว่า ฟังก์ชัน `gcd(m,n)` เป็นฟังก์ชันเรียกตัวเอง ดังนั้นจงเขียนฟังก์ชันภาษาซีสำหรับฟังก์ชันนี้ (`mod` หมายถึงโอเปอร์เรเตอร์สำหรับการหาค่าโมดูโล)

8. ฟังก์ชันชนิดหนึ่งที่มีชื่อว่า Ackermann's Function เป็นฟังก์ชันสองตัวแปร A(m,n) และคุณสมบัติ ดังต่อไปนี้

$$
\begin{aligned}
A(0,\ n) &= n+1, && n \geq 0 \\
A(m,\ 0) &= A(m-1,\ 1), && m > 0 \\
A(m,\ n) &= A(m-1,\ A(m,\ n-1)), && m,\ n > 0
\end{aligned}
$$

จงเขียนฟังก์ชันแบบเรียกตัวเองสำหรับใช้ในการหาค่าของ A(m, n) เมื่อ m และ n เป็นจำนวนเต็มบวกหรือศูนย์