บทที่ 10
ข้อผิดพลาดที่มักปรากฏเสมอและวิธีตรวจแก้

10.1 ข้อผิดพลาดที่มักปรากฏ

1) ลืมใส่เครื่องหมายมหัพภาค(;) หรือใส่ในที่ ๆ ไม่ต้องการเครื่องหมายนั้น

โดยปกติในท้ายคำสั่งทุกคำสั่งในภาษา C จะต้องปิดด้วยเครื่องหมายมหัพภาพเพื่อแสดงให้คอมไพเลอร์ทราบว่าได้จบคำสั่งนั้น ๆ แล้ว หากเราไม่ใส่เครื่องหม่าย ; ท้ายคำสั่งคอมไพเลอร์อาจดำเนินการ 2 อย่างอย่างใดอย่างหนึ่งแล้วแต่รุ่นของคอมไพเลอร์ เช่น อาจอ่านคำสั่งบรรทัดต่อไปจนพบเครื่องหมาย; แล้วถือว่าทุกคำสั่งก่อนหน้านี้ เป็นคำสั่ง

เดียวกัน บางรุ่นอาจแจ้ง error message แล้วคอยจนกว่าเราจะแก้ไขให้ถูก โดยเติม ; ท้ายคำสั่งจนถูกต้องแน่นอนแล้วจึงยอมอ่านคำสั่งต่อไป

อนึ่ง คำสั่งหลายคำสั่งในภาษา C จะไม่ต้องการเครื่องหมาย ; เช่น # include # define loop ต่าง ๆ if ชื่อฟังก์ชั่น เช่น

```
# define MAXSIZ 50
```

หรือ

```
# include "stdio.h"
```

หรือ

```
askint ( )
```

หรือ

```
for (num = 0; s[i] > = '0' && s[i] < = '9'; i++)
```

หรือ

```
while (s[i]! = '\0' )
```

ซึ่งเป็นรูปที่ไม่ต้องการเครื่องหมายมหัพภาค หากเราใส่เครื่องหมายมหัพภาคลงไปจะถือว่าผิดไวยากรณ์

2) ลืมใส่วงเล็บปีกกา

วงเล็บปีกกานั้นเราใส่ไว้เพื่อรวมเอาคำสั่งต่าง ๆ หลายคำสั่งไว้ด้วยกันเรียกว่าบล็อค (ยกเว้นรูปไวยากรณ์ของโครงสร้างและยูเนียน) หากเราลืมใส่วงเล็บปีกกาเปิดคอมไพเลอร์จะรับ และทำคำสั่งแรกของบล็อคนั้นเท่านั้น โดยมากบล็อคเป็นสิ่งที่เราใช้กับ loop เช่น for-loop while-loop do-while-loop และ if

เช่นใน while-loop ต่อไปนี้คือ

```
while (i < MAX) {
   x[i] = func1 ( ) ;
   ++i ;
  }
```

จึงจะทำครบทุกรอบตามเงื่อนไข

สิ่งที่มักปรากฏเสมอก็คือ การหลงลืมปิดวงเล็บ (คือ }) ซึ่งนับว่าเป็นปัญหามาก เพราะคอมไพเลอร์จะไม่แจ้ง error message มาให้ทราบ หากเราลืมปิดวงเล็บ คอมไพเลอร์ จะถือว่าคำสั่งทั้งปวงก่อนที่จะตรวจพบวงเล็บปีกกาที่มีอยู่เป็นคำสั่ง ในบล็อคเดียวกัน

ต่อไปนี้เป็นโปรแกรมสำหรับตรวจสอบว่ามีวงเล็บปีกกาครบคู่หรือไม่ ซึ่งแม้จะช่วยได้ไม่มากเพราะมิได้บอกว่าได้เขียนตกที่ใดก็ยังช่วยเตือนเราได้

```
  /* program to check openning and closing braces */
  /* assumes that filename to he checked*/
  /* is a command line argerment */
#include? "stdio.h"
# define CLEAR 12          /* control codes for ADDS */
# define CURSOR '\033y'    /* viewpoint terminal */
main (argc, argv)
int argc ;
char ** argv ;
{
char last-char ;
int O-count, c-count, c ;
FILE * f1 * fopen ( ) ;
putchar (CLEAR) ;
if((f1 = fopen (argv[1], "r")) = = NULL) {
     printf ("can not open %s n", argv[1]);
     exit (1) ;
 }
```

```
last-char = ' ' ;
o-count = c-count = 0 ;
puts ("open braces close brace\n") ;
while ((c = getc(f1))! = EOF {
      if (last-char = = '/' && c = = '*') {
          while (c! = '/' && last-char! = '*') {
                last-char = c ;
                c = getc (f1) ;
            }
       }
      if(c = = '\') {
         last-char = c :
         c = getc(f1);
         if(c = = '\\') {
                while (c! = '\')
                    c = getc (f1) ;
         }
         if(c = = '{' || c = = '}')
                c = getc (f1 ) ;
       }
         if(c = = " " && last-char! = '\") {
              c = getc(f1) ;
              while (c! = " ")
                  c = getc (f1) ;
          }
         if(c = = '{') {
                o-count + = 1 ;
                set-cur (2, 7, O-count) ;
```

```
                if(c = = '}') {
                     c = count + = 1 ;
                     set-cur (2, 30,  c-count)  :
                }
                last-char = c  ;
          }
          fclose (f1) ;
          if((o-count - c-count) = = 0)
               printf (" n n brace count is okey! ! !\n") ;
          else
               printf ("\n\n brace count is incorrect\n") ;
     }
     set-cur (row, col, num)
     int row, col, num ;
     {
          printf("%s%c%c%d, CURSOR, row+31, col+31, num) ;

     }
```

ตัวแปร last-char ทำหน้าที่เช็คตัวหมายเหตุ สตริงที่อยู่ในเครื่องหมายคำพูด และอักขระเดี่ยวที่ปรากฏอยู่ในวงเล็บส่วนท้ายของโปรแกรมทำหน้าที่นับจำนวนวงเล็บปีกกา เปิด-ปิด

ฟังก์ชัน exit ( ) ทำหน้าที่ที่ปิดไฟล์ที่เปิดไว้แล้วส่งค่าไปให้ OS (โปรแกรมควบคุมระบบ) เพื่อแจ้งให้ทำงานต่อหรือเลิก โดยปกติ main ( ) จะทำหน้าที่เรียกใช้ exit ( ) สำหรับ char ** argv แสดงว่า argv เป็น pointer ที่ชี้ไปที่ char

3) ความสับสนระหว่างตัวดำเนินการกำหนดค่า (=) และตัวดำเนินการเปรียบเทียบ (= =)

ตัวดำเนินการเปรียบเทียบคือตัวดำเนินการที่ใช้เปรียบเทียบ โดยปกติเรามีใช้ 5 แบบ คือ >, <, = =, >= และ <= สิ่งที่มักใช้ผิดคือเท่ากับ (คือ = =) ซึ่งเรามักจะเขียนเป็น = เครื่องหมายเท่ากับเดียว (=) หมายถึงการกำหนดค่ามีความหมายเหมือน ← เช่น x y ก็คือใส่ y ลงใน x มีความหมายเหมือน x←y เมื่อเราต้องการใช้ = = แต่เขียนเป็น = การสื่อความหมายและการแปลคำสั่งจะผิดไปทั้งคอมไพเลอร์ก็รับคำสั่งนั้นได้โดยไม่แจ้ง error message

เช่น `flag = array [i] = = 'x' ;`

แปลว่า `flag = (array [i] = = 'x') ? 1 :0;`

ขณะที่ `flag = array [i] = 'x';`

เป็น multiple assignment แปลว่าแทน array [i] ด้วย 'x' และแทน flag ด้วย ค่าใน array [i] คือ 'x'

สำหรับใน loop ต่าง ๆ นั้น ปลาย loop จะต้องเป็นตัวดำเนินการเปรียบเทียบเสมอ หากใช้เป็นตัวดำเนินการกำหนดค่าจะผิดรูปไวยากรณ์เช่น

`for (i=0; i=MAX; i++)` เป็นแบบที่ผิด

`for (i=0; i= =MAX; i++)` เป็นแบบที่ถูก

4) การลืมใช้เครื่องหมาย */ ปิดท้ายข้อความที่เป็นหมายเหตุ

ถ้าเราต้องการหมายเหตุ (comment หรือ remark) เราจะเขียนข้อความที่ต้องการชี้แจงเป็นพิเศษหรือขยายความไว้ในเครื่องหมาย /* และ */ หากเราลืมปิด

ท้ายข้อความด้วย */ คอมไพเลอร์จะถือว่าข้อความและคำสั่งต่าง ๆ ที่อยู่หลัง /* เป็นหมายเหตุทั้งหมด และจะถือว่าเป็นเช่นนั้นเรื่อยไปจนกว่าจะพบเครื่องหมาย */ ถ้าหากไม่พบคอมไพเลอร์อาจแจ้งว่า "unexpected end of file"

5) การหลงลืมมิได้กำหนดลักษณะของอาร์กิวเมนต์ในฟังก์ชัน

เมื่อมีการเรียกใช้ฟังก์ชันโดย main ( ) ส่งอาร์กิวเมนต์ไปให้ ฟังก์ชันจะต้องระบุชนิดของตัวแปรรับอาร์กิวเมนต์นั้นให้ถูกต้อง หากไม่ระบุคอมไพเลอร์จะกำหนดให้เป็น int เสมอ (default) การไม่ระบุลักษณะตัวแปรในฟังก์ชันจะมีผลให้เกิดปัญหาการทำงานของฟังก์ชันขึ้นได้ โดยเฉพาะปัญหาเรื่องที่เก็บข้อมูลที่อาจใหญ่เกินไปหรือเล็กเกินไป ถ้าใหญ่เกินไปจะเปลืองที่ ถ้าเล็กเกินไปค่าที่ส่งคืนอาจถูกตัดทิ้งบางส่วน (truncate) หรือถูกปัดเศษ (rounding)

6) การหลงลืมลักษณะของฟังก์ชันใน main ( )

ลักษณะของฟังก์ชันก็คือ ลักษณะข้อมูลที่ส่งออกจากฟังก์ชัน เช่น double func1 ( ) แปลว่า func1 ( ) ส่งค่าเป็น double หากไม่กำหนดชนิดของข้อมูลส่งคืนคอมไพเลอร์จะจัดให้ส่งเป็น int (ตาม default) ปัญหาที่เกิดจะมีลักษณะเดียวกับข้อ 5) เช่น

```
main ( )
{
    double big-num ;
    big-num = func1 (big-num) ;
    printf ("the answer is %s", big-num) ;
       :
}
```

```
    funcl (dbl)
    double dbl ;

    return (dbl) ;
}
```

กรณีนี้ func1( ) จะส่งคืนค่าเป็น int ไปให้ main ( ) โดยใส่ผลลัพธ์ในที่ชื่อ big-num มิใช่ double หากจะให้ fun 1 ( ) ส่งค่าเป็น double เราต้องระบุฟังก์ชั่น func 1 (dbe) เป็น double func 1 (dbl)

อนึ่ง หากเราประสงค์จะให้ฟังก์ชันส่งค่าเป็นอย่างอื่นนอกเหนือจาก int เราจะต้องระบุชนิดของข้อมูลหน้าชื่อฟังก์ชั่นทั้งใน main ( ) และในฟังก์ชัน เช่นใน main( ) ระบุเป็น

```
main ( )
{
   double big-num, func 1 ;
   :
```

และในฟังก์ชั่นระบุเป็น

```
double func 1 (dbl)
```

7) ใช้ pointer โดยไม่กำหนดค่าเริ่มต้น

ค่าเริ่มต้นของ pointer คือ แอดเดรสของตัวแปร ที่เราประสงค์จะชี้ให้-pointer ไปรับเอาค่านั้น ถ้าหากไม่กำหนดค่าเริ่มต้นให้ pointer ก็จะชี้เดาสุ่มคืออาจชี้ไปยังขยะกองใดกองหนึ่ง (garbage) เราจึงต้องจำไว้เป็นกฎเลยว่า "เมื่อจะใช้ pointer จะต้องกำหนดค่าเริ่มต้นคือแอดเดรสให้แก่ pointer ก่อนเสมอ"

สิ่งที่ควรระลึกเกี่ยวกับอะเรย์คือ การส่งชื่ออะเรย์มีผลเสมือนส่งแอดเดรสของสมาชิกแรกของอะเรย์ ดังนั้นอะเรย์จึงเป็น pointer อยู่แล้วในทุกครั้งที่มีการเรียกใช้ฟังก์ชัน แต่ถ้ากำหนดอะเรย์ไว้ในฟังก์ชั่นเราจะไม่อาจส่ง pointer นั้นคือ main ( ) ได้ เพราะทุกสิ่งที่นิยามไว้ในฟังก์ชั่นเป็น auto ซึ่งจะไม่เอื้อเฟื้อให้ถูกเรียกใช้จากภายนอกได้ หากมีการเรียกใช้จากภายนอกค่าที่ส่งออกไปจากฟังก์ชันก็ไม่มีความหมายอะไรเพราะพอค่านั้น (ซึ่งเป็น local) ถูกส่งพ้นฟังก์ชัน (คนละตัวกับค่าส่งคืน ในที่นี้หมายถึงตัวแปรใด ๆ ที่ใช้ทำงานให้เกิดค่าส่งคืน) ค่าก็จะหายไป (lost หรือ die)ไปแล้ว

ด้วยเหตุผลดังกล่าวการส่งค่าคืนสู่ main ( ) จึงเป็นเรื่องที่ต้องระมัดระวังอย่าใช้ชื่อตัวแปรเดียวกันกับตัวแปรที่ใช้ทำงานให้เกิดค่าส่งคืน (return value) หากจำเป็นต้องใช้ให้ใช้เป็น pointer

10.2 การตรวจสอบและแก้ไขโปรแกรม

ข้อผิดพลาดในโปรแกรมนั้น โดยทั่วไปจะมีอยู่ 3 แบบ คือ ผิดไวยากรณ์-(syntax error) เขียนโปรแกรมผิด (progamming error) และความผิดที่แอบแฝง (latent error) การผิดไวยากรณ์คือเราใช้รูปไวยากรณ์สำหรับคำสั่งต่าง ๆ ผิดจากแบบที่คอมไพเลอร์จะพึงรับรู้ได้ (ดูบทที่ 1-4) กรณีนี้คอมไพเลอร์จะแจ้งให้ทราบเอง ถ้าเกิดความผิดพลาดขึ้น ซึ่งเป็นความผิดพลาดที่แก้ไขได้ง่ายที่สุด ความผิดพลาดประการที่ 2 คือ การเขียนโปรแกรมผิด คือรูปไวยากรณ์ถูกแต่เราเองเขียนโปรแกรมไม่ถูก กรณีนี้คอมไพเลอร์จะไม่แจ้งอะไรให้ทราบและทำคำสั่งต่าง ๆ เสมือนไม่มีอะไรผิดปกติ แต่ให้ผลลัพธ์ที่ได้ไม่ถูกต้อง กรณีนี้โปรแกรมเมอร์จะต้องตรวจโปรแกรมเองตลอดทั้งหมดเมื่อพบที่ผิดก็แก้ให้ถูกต้อง ความผิดพลาดประการที่สามคือ ความผิดพลาดที่แฝงเร้นอยู่ เรียกว่า

latent bug คือความผิดที่แอบแฝงอยู่เนื่องจากข้อมูลของเราผิด ถ้ารู้ว่าข้อมูลส่วนใดผิดแล้วแก้ให้ถูกต้องก็จะแก้ปัญหาได้

วิธีที่ป้องกันมิให้เกิดข้อผิดพลาดหรือหากมีข้อผิดพลาดแล้วสามารถแก้ไขได้โดยง่ายก็คือให้พยายามใช้ main ( ) เรียกฟังก์ชันในลักษณะของ modular approach อย่าพยายามใส่ทุกสิ่งทุกอย่างลงใน main ( ) โดยไม่ยอมเรียกใช้ฟังก์ชันซึ่งเกิดขึ้นเสมอ สำหรับผู้เริ่มเขียนโปรแกรม เพราะไม่สันทัดในเรื่องการเรียกฟังก์ชัน และการส่งอาร์กิวเมนต์ การเรียกใช้ฟังก์ชันโดยผลักภาระการทำงานที่สับสนออกไปจาก main ( ) ให้แก่ฟังก์ชันจะช่วยให้ตัวแปรต่าง ๆ มีโอกาสสับสนหรือเกิดปฏิสัมพันธ์กันน้อยลง เราจึงควรใช้วิธีแบ่งงานทั้งปวงเป็นส่วนย่อย ๆ ให้ฟังก์ชันรับไปทำแล้ว main ( ) ทำหน้าที่สั่งทำงานสั่งให้ส่งงานเรียกว่า "แบ่งแยกแล้วปกครอง" ซึ่งเป็นวิธีที่เป็นหลักการของภาษา c ทำให้ตรวจแก้โปรแกรมได้ง่ายเพราะเราสามารถแก้เป็นส่วน ๆ ตรวจเป็นส่วน ๆ ภาระการตรวจแก้จึงน้อยกว่าการใส่ทุกสิ่งทุกอย่างไว้ใน main ( )

วิธีที่นิยมใช้ในการตรวจแก้โปรแกรมก็คือให้เราใช้ข้อมูลที่เรารู้จักคือ รู้ว่าผลลัพธ์จะต้องเป็นเท่าไรแน่นอนมาทดสอบโปรแกรม แล้วสั่งพิมพ์ผลลัพธ์ออกมาดูเป็นระยะ ๆ ด้วยฟังก์ชัน printf ( ) หากไม่ชอบใช้วิธีสั่งพิมพ์ผลลัพธ์เป็นระยะด้วย printf ( ) เราอาจสร้างฟังก์ชันสั่งตรวจแก้ขึ้นใช้เองได้ดังตัวอย่างฟังก์ชั่น debug ( ) ต่อไปนี้ ซึ่งใช้พิมพ์ค่าต่าง ๆ ที่ส่งจาก main ( )ออกมาดูผลลัพธ์

```
/* function to print int, char, and numeric character array */
debug (let, c-array, n-array, asize, num opt)
int num, narray [ 1, opt, asize ;
char let, c-array [ 1 ;
{
```

```
int i ;
switch (opt) {
   case 1 :
          printf("\n the value is %d", num) ;
          break ;
   case 2 :
          printf ("\n the letter is %c", let) ;
          break ;
   case 3 :
          puts ("\n the numeric array contains\n") ;
          for (i=0; i<asize; ++i)
               printf ("%d", naray [i]) ;
          break ;
   case 4 :
          puts("\n the character array contains\n") ;
          for (i=0; i<asize; ++i)
               printf("%c", c-array [i]) ;
          break ;
   default :
          puts ("\n invalid option selected") ;
          break ;
   }
   puts("\n\t press any key to continue ") ;
   getchar ( ) ;
}
```

ถ้าเราผนวกฟังก์ชั่นนี้เข้ากับ main ( ) โดยเรียกใช้ตามโปรแกรมต่อไปนี้คือ

```
#define  CLEARS12
# include "debug .c"
main ( )
{
   int i ,  x,val [5] ;
   c h a r  alpha,let [5] ;
    /* input some sample data */
    for(i = 65, x = 0; i < 70;++i   ++x) {
        val [x] = x ;
        let [xl = i ;
    }
    putchar (CLEARS) ;
    alpha = 'z' ;
    debug (0, 0, 0, 0, i, 1);  /* display the valve */
    debug (alpha, 0,0,0,0,2) ; /* display letter */
    debug(0,0,val,5, 0 , 3) ;   /* display numeric array */
    debug(0,let,0,5,0,4) ;      /* display char array */
    debug (0,0,0,0,0,7),        /* error */
}
```

การเรียกฟังก์ชัน debug ( ) มาใช้นั้นเราจะต้องส่งอาร์กิวเมนต์ไปในทั้งสิ้น 6 รายการ ดังนี้คือ

debug (let, c-array, narray, asize, num , opt)

โดยที่ c-array เป็น char array ขณะที่ narrayเป็น int array การเรียกในโปรแกรมข้างบนกระทำเป็น 5 รอบ

รอบที่ 1 debug ( ) ได้รับสำเนา (copy) ค่าของตัวแปร i ส่งมาให้ num และ opt ได้รับค่าเท่ากับ 1 ผลก็คือ case 1: ถูกเรียกทำงานคือ printf

("\n the value is % d" , num) ; คือพิมพ์ค่าของ i

รอบที่ 2 debug ( ) ถูกเรียกทำงานโดยรับอาร์กิวเมนต์ชื่อ alpha ซึ่งเป็น char และ opt(คือ option) หมายเลข 2 ผลคือ case 2 : ถูกเรียกทำงาน

รอบที่ 3 รอบที่ 4 และรอบที่ 5 ก็ทำงานลักษณะเดียวกัน หากเราสนใจจะสั่งพิมพ์สิ่งอื่น ๆ มาดูก็เพิ่มค่าของอาร์กิวเมนต์ให้มากกว่าที่แสดงไว้ในตัวอย่าง

หนังสืออ้างอิง

Purdum, Jack, C programming guide, Que Corporation, Indiana, 1983, 250 Pp.

Pugh, Kenneth, C language for programmers Scott, Foresman and Company, Illinois, 1985, 198 pp

Wrotman, Leon A. and Sidebottom, Thomas O., " The C programming tutor, Prentice, Hall international, Englewood Cliffs, N.J., 1984, 274 pp.