8

โครงสร้าง และยูเนียน

8.1 โครงสร้าง (Structure)

ตามที่เราได้เรียนรู้ไปในบทที่แล้ว อาร์เรย์คือแถวหรือลำดับของข้อมูลหลายๆตัวที่มีแบบข้อมูลชนิดเดียวกัน เช่น ถ้าเป็นอาร์เรย์ของข้อมูลแบบ `int` ข้อมูลทุกตัวในอาร์เรย์ก็จะต้องเป็น `int` เหมือนกันทั้งหมดหรือถ้าเป็นพอยน์เตอร์ ข้อมูลทุกตัวของอาร์เรย์ก็ต้องเป็นพอยน์เตอร์แบบเดียวกันทั้งหมด ข้อจำกัดในการใช้งานอาร์เรย์ ในลักษณะนี้จะไม่มี ถ้าเราหันมาใช้ ข้อมูลแบบโครงสร้าง (Structure) ซึ่งถือว่าเป็นแบบข้อมูลรวม (Aggregate Data Type) ชนิดหนึ่ง

การใช้อาร์เรย์มีข้อดีคือ เราสามารถเข้าถึงข้อมูลแต่ละตัวภายในอาร์เรย์โดยใช้เลขดัชนี ซึ่งทำให้สะดวกและรวดเร็วในการจัดการกับข้อมูลที่มีอยู่ ดังนั้นจึงเหมาะสมกับการเก็บข้อมูลจำนวนมากที่เรา

สามารถเข้าถึงได้โดยการกำหนดตำแหน่งหรือหมายเลข ในขณะที่โครงสร้าง คือ แบบข้อมูลรวมที่ประกอบด้วยข้อมูลต่างๆที่มีแบบข้อมูลเหมือนกันหรือต่างกันก็ได้ ดังนั้นจึงจำเป็นต้องใช้วิธีการอื่นในการเข้าถึงข้อมูลแต่ละตัวของโครงสร้างซึ่งแทนที่จะกำหนดเลขดัชนีก็จะใช้วิธีการการกำหนดชื่อให้ข้อมูลแต่ละตัวของโครงสร้าง ถ้าใครรู้จักภาษาปาสคาลก็คงรู้จักความหมายของคำว่า `Record` ซึ่งก็คือโครงสร้างของข้อมูลนั้นเอง ในภาษาซีจะใช้ คำว่า `struct` เป็นการบ่งบอกถึงแบบข้อมูลที่เป็นโครงสร้าง

ถ้าเราต้องการนิยามข้อมูลแบบโครงสร้าง เราก็ต้องกำหนดก่อนว่า เราจะใช้ ข้อมูลใดและแบบใดบ้าง ที่เราต้องการเก็บไว้ในโครงสร้าง ข้อมูลเหล่านี้ เราจะเรียกว่า *ข้อมูลสมาชิก* เพราะเป็นองค์ประกอบของโครงสร้าง จุดประสงค์ของการใช้งานข้อมูลแบบโครงสร้าง คือ การนำข้อมูลอย่างน้อยหนึ่งตัว หรือหลายๆรูปแบบมาเก็บไว้รวมกัน ข้อมูลสมาชิกเหล่านี้จะต้องมีชื่อที่ไม่ซ้ำกันและจะถูกจัดอยู่รวมกันไว้เป็นหนึ่งหน่วยหรือเป็นหนึ่งกลุ่ม ภายใต้ ชื่อแบบข้อมูลเดียวกัน

สำหรับตัวแปรใดๆที่มีแบบข้อมูลเป็นโครงสร้าง ถ้าเราต้องการจะเข้าถึงข้อมูลสมาชิกก็จะต้องอาศัยชื่อของข้อมูลตัวนั้น ตัวอย่างการนิยามแบบข้อมูลที่เป็นโครงสร้าง เช่น ในทางคณิตศาสตร์ เลขเศษส่วน เราจะแบ่งออกเป็นตัวเศษ (numerator) และตัวส่วน (denominator) ถ้าเราต้องการนิยามแบบข้อมูลสำหรับใช้เก็บเลขเศษส่วน (คือเก็บทั้งตัวเศษและตัวส่วนไว้พร้อมกัน) เช่น ใช้ชื่อว่า `ratio` เราก็สามารถเขียนได้ดังนี้

```
struct ratio {
     long  num;
     long  den;
};
```

จากตัวอย่าง เราได้นิยามแบบโครงสร้างชื่อ `ratio` ที่มีสมาชิกสองตัวแบบ `long` คือ `num` และ `den` ซึ่งใช้สำหรับเก็บค่าของตัวเศษและตัวส่วนตามลำดับ ชื่อของแบบข้อมูลจึงเป็น `struct ratio` โปรดสังเกตว่า เราต้องเขียนคำว่า `struct` ไว้ด้วย เพื่อแจ้งให้ทราบว่า แบบข้อมูลที่ชื่อ `ratio` นี้เป็นโครงสร้าง

ถ้าเราต้องการแจ้งใช้ ตัวแปรใดๆที่เป็นข้อมูลแบบ `struct ratio` เราก็ทำได้เหมือนกับการแจ้งใช้ตัวแปรพื้นฐานทั่วไป แต่อย่าลืมคำว่า `struct` ตัวอย่างการแจ้งใช้ เช่น

```
struct ratio  x, y, z, *p = &x;
```

ตัวแปร `x` `y` และ `z` เป็นตัวแปรแบบโครงสร้าง และ `p` เป็นพอยน์เตอร์ที่ชี้ไปยังแหล่งข้อมูลแบบโครงสร้างที่ชื่อ `ratio` ซึ่งในกรณีนี้ พอยน์เตอร์ `p` จะ ชี้ไปยังที่อยู่ของตัวแปร `x`

เราอาจจะนิยามโครงสร้างและแจ้งใช้ ตัวแปรแบบโครงสร้างนี้ไปในคราวเดียวกันก็ได้ เช่น

```
struct ratio {
     long  num;
     long  den;
} x, y, z,*p = &x;
```

ในการเข้าถึงข้อมูลสมาชิกแต่ละตัว เราจะใช้โอเปอร์เรเตอร์ . (จุด) เขียนต่อท้ายชื่อของตัวแปรและตามด้วยชื่อของสมาชิกที่ต้องการโดยไม่เว้นที่ว่าง เช่น ถ้าต้องการกำหนดค่าต่างๆให้แก่ข้อมูลสมาชิก ก็ทำได้ตามตัวอย่างต่อไปนี้

```
x.num =  1;
x.den =  3;
y.den = -2;
y.den =  5;
```

ดังนั้นจากประโยคคำสั่งเหล่านี้ เราจะได้ค่าของเลขเศษส่วนที่แทนสัญลักษณ์ด้วย x มีค่าเท่ากับ ‘หนึ่งส่วนสาม’ และ y เท่ากับ ‘ลบสองส่วนห้า’ แม้ว่าข้อมูลสมาชิกของตัวแปร x และ y จะมีรูปแบบเหมือนกัน แต่ตัวแปรทั้งสองมีบริเวณหน่วยความจำของตนเองที่ใช้เก็บข้อมูลสมาชิกแต่ละตัว ดังนั้นจึงเก็บค่าที่แตกต่างกันเพราะทันทีที่เราแจ้งใช้ตัวแปรแบบโครงสร้าง คอมพิวเตอร์ก็จะจัดสรรหน่วยความจำให้ตัวแปรแบบโครงสร้างนี้ทันที เช่นเดียวกับการแจ้งใช้ตัวแปรอัตโนมัติทั่วๆไป

ถ้าเราต้องการเข้าถึงข้อมูลสมาชิกของโครงสร้างโดยใช้พอยน์เตอร์ p เราจะเขียนโอเปอร์เรเตอร์ -> (เครื่องหมายลบและเครื่องหมายมากกว่าตามลำดับ รวมเรียกว่า ลูกศร) ต่อท้ายชื่อของพอยน์เตอร์ และตามด้วยชื่อของข้อมูลสมาชิกที่ต้องการ เช่น ถ้าต้องการอ่านและแสดงค่าของข้อมูลสมาชิก

```
printf ("Numerator = %ld\n", p->num);
printf ("Denominator = %ld\n", p->den);
```

ตัวอย่างการเข้าถึงข้อมูลสมาชิกของตัวแปรแบบโครงสร้าง เช่น ถ้าเราต้องการบวกค่าของเลขเศษส่วนจาก x และ y แล้วเก็บผลบวกไว้ในตัวแปร z ซึ่งก็เป็นโครงสร้างแบบ ratio เราก็ทำได้ดังนี้

```
z.num =  x.num * y.den + x.den * y.num;
z.den =  x.den * y.den;
```

ซึ่งเป็นการบวกเลขระหว่างเลขเศษส่วนสองตัว ตามวิธีการที่เราคุ้นเคย เช่น

$$\frac{a}{b}+\frac{c}{d} = \frac{ad+bc}{bd}$$

หรืออีกตัวอย่างหนึ่งสำหรับการนิยามแบบโครงสร้างที่คล้ายกันคือ การนิยามแบบข้อมูลโครงสร้างสำหรับเก็บเลขจำนวนเชิงซ้อน (Complex Number) ถ้า z เป็นเลขจำนวนเชิงซ้อนใดๆ เราสามารถแบ่งออกได้เป็นสองส่วนตามคำนิยาม คือ

$$z = \mathrm{Re}\{z\} + j\,\mathrm{Im}\{z\}$$

โดยกำหนดให้ส่วนแรกมีชื่อว่า Re และ ส่วนที่สองมีชื่อว่า Im ตามลำดับและใช้ ชื่อของแบบข้อมูลโครงสร้างเป็น complex สำหรับใช้ในโปรแกรมภาษาซี ซึ่งมีรูปแบบดังนี้

```
struct complex {
     double  Re;
     double  Im;
};
```

จากตัวอย่างของแบบข้อมูลโครงสร้างทั้งสองที่ผ่านไป เราใช้ข้อมูลสมาชิกที่มีแบบข้อมูลพื้นฐานเดียวกันเท่านั้น แต่อย่างไรก็ตาม ก็ไม่ได้จำกัดอยู่แค่ว่าจะต้องใช้ข้อมูลสมาชิกที่มีแบบข้อมูลเหมือนกันเท่านั้น แต่เรายังสามารถใช้ข้อมูลสมาชิกที่มีแบบข้อมูลแตกต่างกันได้ ตัวอย่างเช่น ถ้าต้องการจัดเก็บข้อมูลเกี่ยวกับที่อยู่ของบุคคลต่างๆ ซึ่งใช้ในการติดต่อผ่านทางไปรษณีย์ หรืออาจจะมีหมายเลขโทรศัพท์ จะมือถือหรือไม่ก็ตาม หรือผ่านอินเตอร์เน็ตโดยใช้ไปรษณีย์อิเล็คทรอนิค ข้อมูลเกี่ยวกับที่อยู่ของบุคคลเหล่านี้ก็จะประกอบด้วยชื่อของผู้รับและที่อยู่ที่ติดต่อได้ เบอร์โทรศัพท์ และที่อยู่ของผู้รับสำหรับไปรษณีย์อิเล็คทรอนิคหรืออีเมล์

```
struct  person {
     char name[40];
     char address[255];
     char phone[16];
     char email[30];
};
```

โครงสร้างชื่อ person มีข้อมูลสมาชิกทั้งหมดสี่ตัวเป็นอาร์เรย์แบบ char มีขนาดแตกต่างกันไป และใช้เก็บข้อความที่เกี่ยวข้องกับบุคคลที่เราต้องการจะติดต่อ เช่น ชื่อนามสกุล เก็บไว้ในรูปของสายอักขระในอาร์เรย์ name ซึ่งจะมีขนาดที่ไม่เกิน 40 ไบต์ address สำหรับที่อยู่ทางไปรษณีย์ phone สำหรับเบอร์โทรศัพท์ และ email สำหรับไปรษณีย์อิเล็คทรอนิค ซึ่งมีขนาดไม่เกิน 255 ไบต์ 16 ไบต์และ 30 ไบต์ตามลำดับ

ตัวอย่างต่อไป เป็นโครงสร้างที่มีข้อมูลสมาชิกเป็นพอยน์เตอร์ที่ชี้ไปยังโครงสร้างของตนเอง

```
struct list {
     int           key;
     void         *data;
     struct list  *next;
};
```

โครงสร้าง list ประกอบด้วยสมาชิกสามตัว ข้อมูลสมาชิกตัวแรกคือ key แบบ int ตัวที่สองคือ data เป็นพอยน์เตอร์อเนกประสงค์ และ ตัวที่สามคือ next ซึ่งเป็นพอยน์เตอร์ที่ชี้ไปยังข้อมูลโครงสร้างแบบ struct list โครงสร้างที่มีสมาชิกอย่างน้อยหนึ่งตัว เป็นพอยน์เตอร์แบบโครงสร้างของตนเอง (พอยน์เตอร์ที่ชี้ไปยังข้อมูลแบบโครงสร้าง) ในลักษณะนี้ จะใช้ สำหรับสร้างโครงสร้างของข้อมูลที่เรียกว่า รายการเชื่อมโยง (Linked List)

สมาชิกของโครงสร้างสามารถเป็นโครงสร้างแบบอื่นๆได้ที่นอกเหนือจากแบบข้อมูลพื้นฐาน เช่น

```
struct person {
     char name[40];
     char address[255];
     char phone[16];
     char email[30];
};

struct person_list {
     int                 key;
     struct person       data;
     struct person_list  *next;
};
```

ในกรณีตัวอย่างนี้ โครงสร้างชื่อ person_list มีสมาชิกหนึ่งตัวเป็นข้อมูลโครงสร้างแบบ struct person โปรดสังเกตว่า เราจะต้องนิยามโครงสร้างขึ้นก่อนที่จะนำไปใช้เป็นแบบข้อมูลสำหรับสมาชิกในโครงสร้างตัวอื่น ดังนั้น จึงต้องนิยาม struct person ก่อน struct person_list

8.1.1 รูปแบบการนิยามและแจ้งใช้ตัวแปรที่เป็นโครงสร้าง

เราสามารถเขียนรูปแบบทั่วๆไป สำหรับการนิยามโครงสร้างและแจ้งใช้ ตัวแปรที่เป็นโครงสร้างได้ดังนี้

```
struct structure_name {
     data_type1     member_name1;
     data_type2     member_name2;
     data_type3     member_name3;

     ...            ...

     data_typeN     member_nameN;
} variable_list;
```

สำหรับ structure_name จะหมายถึงชื่อของโครงสร้าง เราอาจจะไม่ต้องใส่ไว้ก็ได้ ซึ่งหมายความว่าโครงสร้างนี้ไม่มีชื่อ และจะมีข้อจำกัดในการใช้คือ ถ้าโครงสร้างใดๆที่ถูกนิยามขึ้นโดยปราศจากการกำหนดชื่อแล้ว เราจะไม่สามารถแจ้งใช้ตัวแปรที่มีแบบข้อมูลเป็นโครงสร้างนี้ได้อีกในภายหลัง ดังนั้นในกรณีเช่นนี้ เราจะต้องแจ้งใช้ตัวแปรไปพร้อมกับการนิยามโครงสร้างนี้ ตัวอย่างเช่น

```
struct {
     char *name;
     char **args;
} progCmdLine;
```

ในการนิยามโครงสร้าง เราอาจจะแจ้งใช้ตัวแปรไปพร้อมกันก็ได้ แต่ตามปรกติแล้วจะไม่นิยมแจ้งใช้ตัวแปรต่อท้ายเมื่อนิยามโครงสร้าง (ยกเว้นกรณีที่ได้นิยามโครงสร้างโดยปราศจากการกำหนดตัวระบุชื่อ) แต่อย่างไรก็ตามโปรดอย่าลืมเขียนเครื่องหมายเซมิโคลอน (;) จบท้าย

ความสำคัญของเซมิโคลอนที่จบท้ายการนิยามโครงสร้าง เราจะเห็นได้จากตัวอย่างต่อไปนี้

```
/* Structure Definition */
struct struct_type {
     int x, y;
}

/* Function Declaration */
func(int, int);
```

ในตัวอย่างแรกนี้ ไม่มีเซมิโคลอนจบท้ายหลังจากที่ได้นิยามโครงสร้าง struct_type ดังนั้นคอมไพล์เลอร์จะเข้าใจว่า เราได้นิยามโครงสร้าง struct_type และแจ้งใช้ ฟังก์ชัน func() ที่ให้ข้อมูลแบบโครงสร้าง struct_type เป็นค่ากลับคืนจากฟังก์ชัน

```
struct struct_type {
     int x, y;
};

/* Function Declaration */
func(int, int);
```

ในขณะที่ตัวอย่างที่สองหมายความว่า เราได้นิยามโครงสร้าง struct_type และแจ้งใช้ ฟังก์ชัน func() ที่ให้ค่าแบบ int กลับคืน

การนิยามแบบของข้อมูลที่เป็นโครงสร้าง เราสามารถทำได้ทั้งภายนอกและภายในบล็อกหรือฟังก์ชัน ตัวอย่างเช่น

```
void func1()
{
  struct complex {
      double Re;
      double Im;
  } x, y;

  x.Re = y.Re = 1.0;
  x.Im = y.Im = 1.0;
}
```

เปรียบเทียบกับ

```
void func2()
{
  struct complex {
      float Re;
      float Im;
  } x, y;

  x.Re = y.Re = 1.0;
  x.Im = y.Im = 1.0;
}
```

จะเห็นได้ว่า ภายในทั้งสองฟังก์ชันที่ปรากฏอยู่ในโปรแกรมเดียวกัน มีการนิยามแบบโครงสร้าง `struct complex` ขึ้นใช้ แม้ว่า จะมีชื่อเหมือนกัน แต่แบบโครงสร้างทั้งสองจะไม่เกี่ยวข้องกัน และมีข้อสังเกตอยู่อีกว่า ถ้าเรานิยามแบบข้อมูลที่เป็นโครงสร้าง ภายในบล็อกหรือฟังก์ชัน แบบโครงสร้างที่ได้นิยามไว้นี้ จะใช้ได้เฉพาะภายในบล็อกหรือบล็อกของฟังก์ชันที่มีการนิยามแบบโครงสร้างนี้เท่านั้น

8.1.2 การใช้ `typedef` กับแบบข้อมูลโครงสร้าง

การใช้คำสั่ง `typedef` ในการนิยามแบบข้อมูลขึ้นใหม่สำหรับโครงสร้างใดๆมีข้อดีคือ ทำให้เราสามารถเลือกชื่อใหม่สำหรับแบบข้อมูลที่สั้นและสะดวกสำหรับการเขียนหลายๆครั้งได้ ตัวอย่างเช่น

```
struct node {
     char         *str;
     struct node  *next;
};

typedef struct node  Node;
```

เวลาเราต้องการแจ้งใช้ ตัวแปร เราก็เขียนได้ง่ายๆ เช่น

```
Node list, *plist;
```

และให้ความหมายว่า ตัวแปร list เป็นโครงสร้างแบบ struct node และ plist เป็นพอยน์เตอร์ที่ชี้ไปยังโครงสร้างแบบ struct node ซึ่งแทนที่เราจะเขียนว่า struct node ทุกครั้งไป เราก็เขียนแค่คำว่า Node แทน

นอกจากนี้เรายังสามารถเขียนคำสั่ง typedef ไว้ก่อนการนิยามโครงสร้างก็ได้ เช่น

```
typedef struct node   Node;
struct node {
     char  *str;
     Node  *next;
};
```

หรือเขียนใหม่ได้เป็น

```
typedef struct node {
     char        *str;
     struct node *next;
} Node;
```

ในกรณีหลังนี้ คำว่า Node ที่อยู่ท้าย มิใช่ ชื่อของตัวแปรแต่อย่างใด แต่เป็นชื่อใหม่ของแบบข้อมูลที่ใช้แทนชื่อของโครงสร้าง struct node แต่อย่างไรก็ตาม เราห้ามเขียนว่า

```
typedef struct node {
     char  *str;
     Node  *next;
} Node;
```

ซึ่งในกรณีนี้ จะถือว่า ผิด เพราะเราได้ใช้ ชื่อใหม่สำหรับแบบข้อมูลคือ Node ภายในโครงสร้าง ก่อนที่จะจบคำสั่ง typedef ดังนั้น คำว่า Node ที่อยู่ภายในโครงสร้าง จึงยังมิได้นิยาม

นอกจากนี้ ก็อาจจะใช้ ชื่อของแบบข้อมูลใหม่ซ้ำกับชื่อของโครงสร้างเดิมได้ เช่น

```
typedef struct node {
     char         *str;
     struct node  *next;
} node;
```

ผลจากการกระทำเช่นนี้ ทำให้เราใช้ได้ทั้งสองชื่อสำหรับโครงสร้างคือ struct node และ node และมีความหมายเหมือนกัน

8.1.3 การเข้าถึงข้อมูลสมาชิกในโครงสร้าง

การเข้าถึงข้อมูลสมาชิกสามารถเขียนสรุปได้ดังนี้ ถ้า `structure_variable` แทนชื่อของตัวแปรแบบโครงสร้างใดๆและ `member_name` แทนชื่อของข้อมูลสมาชิกใดๆในโครงสร้างแล้ว นิพจน์ที่เขียนอยู่ในรูปแบบต่อไปนี้

structure_variable.*member_name*

จะหมายถึง ข้อมูลสมาชิกที่เราต้องการเข้าถึงและใช้งานได้เหมือนตัวแปรทั่วๆไป สำหรับพอนย์เตอร์ที่ชี้ไปยังโครงสร้างเราจะใช้โอเปอร์เรเตอร์ -> แทนจุด และสำหรับนิพจน์ตามรูปแบบต่อไปนี้

structure_pointer->*member_name*

เราก็สามารถใช้ได้เหมือนตัวแปรพอยน์เตอร์ ทั่วๆไป หรือ ถ้าจะเขียนนิพจน์ที่มีความหมายเหมือนกัน ก็จะเป็น

(**structure_pointer*).*member_name*

ถ้ากำหนดไว้ว่า `structure_pointer` ใช้แทนชื่อของพอยน์เตอร์ใดๆที่ชี้ไปยังข้อมูลแบบโครงสร้าง

สำหรับตัวอย่างโครงสร้าง

```
struct person {
     char name[40];
     char address[255];
     char phone[16];
     char email[30];
};

struct person_list {
     int                 key;
     struct person       data;
     struct person_list  *next;
} item;
```

ในกรณีนี้ `item` เป็นตัวแปรแบบ `struct person_list` และเราสามารถเขียนนิพจน์สำหรับสมาชิกแต่ละตัวจำแนกได้ดังนี้

```
item.key
item.data
item.next
item.data.name
```

```
item.data.address
item.data.phone
item.data.email
```

คุณสมบัติของนิพจน์แต่ละตัว จะถูกกำหนดโดยตัวระบุชื่อที่อยู่ท้ายสุด เช่น

```
item.data.address
```

เป็นนิพจน์ที่มีหน้าที่เหมือนตัวแปรที่เป็นอาร์เรย์ขนาด 255 ไบต์ ซึ่งหมายถึงข้อมูลสมาชิก `address` ในโครงสร้างนั้นเอง แต่เป็นส่วนหนึ่งของตัวแปรที่ ชื่อ `item`
ถ้าเรากำหนดให้ ข้อมูลสมาชิก `item.next` ชี้ไปยังโครงสร้างของตนเอง (อ้างอิงตัวเอง) ก็ทำได้โดยการผ่านที่อยู่ของตัวแปร `item` ไปยังตัวแปร `item.next` ที่เป็นพอยน์เตอร์

```
item.next = &item;
```

การกระทำเช่นนี้ จึงเป็นการอ้างถึงตัวเอง ดังนั้นทั้งสองนิพจน์นี้

```
item
*(item.next)
```

จึงหมายถึงข้อมูลแบบโครงสร้างตัวเดียวกัน และผู้อ่านสามารถตรวจสอบได้ด้วยตนเองว่า สำหรับกรณีตัวอย่าง นิพจน์ทั้งหลายต่อไปนี้

```
item.data.name
(*(item.next)).data.name
item.next->data.name

(*((*(item.next)).next)).data.name
(*(item.next->next)).data.name
```

จะให้ผลเหมือนกัน

โปรดสังเกตว่า กรณีตัวอย่างในลักษณะนี้ จะไม่มีความหมายมากนักในการนำไปใช้ประโยชน์ เพียงแต่ช่วยเน้นถึงความเข้าใจของการใช้พอยน์เตอร์และโครงสร้างเท่านั้นเอง

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

struct person {
     char name[40];
     char address[255];
     char phone[16];
     char email[30];
```

```
};

struct person_list {
     int                  key;
     struct person        data;
     struct person_list  *next;
} item;

int main()
{
     item.next = &item;

     strcpy (item.data.name, "Mr. C");

     printf("%s\n", item.data.name);
     printf("%s\n",(*item.next).data.name);
     printf("%s\n", item.next->data.name);
     printf("%s\n",(*((*item.next).next)).data.name);
     printf("%s\n",(*item.next->next).data.name);
     return 0;
}
```

โปรดสังเกตอีกว่า โอเปอร์เรเตอร์ . มีลำดับการทำงานมาก่อนโอเปอร์เรเตอร์ * และ & ที่วางไว้ข้างหน้า พอยน์เตอร์ (สำหรับรายละเอียดเพิ่มเติมเกี่ยวกับลำดับการทำงานของโอเปอร์เรเตอร์ต่างๆ ผู้อ่านสามารถพบได้ในตารางท้ายเล่ม)

8.1.4 การหาที่อยู่ของข้อมูลโครงสร้างและข้อมูลสมาชิก

การหาที่อยู่ของตัวแปรแบบโครงสร้างใดๆ ในหน่วยความจำ เราจะใช้โอเปอร์เรเตอร์ & เหมือนในกรณีที่ใช้กับตัวแปรพื้นฐานแบบอื่น ๆ

&*structure_variable*
&*structure_variable*.*member_name*

ในบรรทัดแรกเป็นการหาที่อยู่ของข้อมูลรวมชื่อ struct_variable ในบรรทัดที่สองเป็นการหาที่อยู่ของข้อมูลสมาชิกชื่อ member_name และโปรดสังเกตว่า โอเปอร์เรเตอร์ . (จุด) มีลำดับการทำงานมาก่อนโอเปอร์เรเตอร์ & ดังนั้นเราจึงไม่จำเป็นต้องใส่เครื่องหมายวงเล็บไว้ดังรูปแบบข้างล่างนี้

&*structure_variable*.*member_name*
&(*structure_variable*.*member_name*)

ตัวอย่างการใช้งาน เช่น

```
#include <stdio.h>

struct complex {
     double  Re;
     double  Im;
};

int main()
{
     struct complex  c = {1.0,-2.0};

     printf("Address of c = %p\n", &c);
     printf("Address of c.Re = %p\n", &(c.Re));
     printf("Address of c.Im = %p\n", &(c.Im));
     return 0;
}
```

ตัวอย่างของผลการทำงานของโปรแกรม

```
     Address of c = 21BE
     Address of c.Re = 21BE
     Address of c.Im = 21C6
```

จะเห็นได้ว่า หมายเลขที่อยู่ของตัวแปร `c` และสมาชิกตัวแรก `c.Re` มีค่าเท่ากัน ดังนั้น ที่อยู่ของข้อมูลตัวแรกจึงเป็นที่อยู่ของโครงสร้าง และ เนื่องจากขนาดของข้อมูลแบบ `double` มีค่าเท่ากับ 8 ไบต์ ดังนั้นเราจึงสรุปได้ว่า ข้อมูลสมาชิกตัวที่สอง `c.Im` จึงอยู่ติดต่อกันไป ถัดจากข้อมูลสมาชิกตัวแรกนั้นเอง

8.1.5 การติดตั้งค่าเริ่มต้นสำหรับข้อมูลโครงสร้าง

เมื่อเราได้แจ้งใช้ตัวแปรแบบโครงสร้างและต้องการติดตั้งค่าเริ่มต้นให้แก่ข้อมูลสมาชิกแต่ละตัวไปพร้อมกัน เราก็สามารถทำได้ในทำนองเดียวกับการแจ้งใช้และติดตั้งค่าของอาร์เรย์ เช่น ถ้าเราต้องการติดตั้งค่าเริ่มต้นให้แก่ตัวแปร `x` `y` และ `z` แบบ `struct ratio` เราก็ทำได้ดังนี้

```
     struct ratio {
          long  num;
          long  den;
     };

     struct ratio x = { 1, 3},
```

```
                    y = {-2, 5},
                    z = { 0, 1};
     struct ratio *p = &x;
```

หรือเขียนใหม่ได้เป็น

```
     struct ratio {
          long  num;
          long  den;
     } x={1,3}, y={-2,5}, z={0,1}, *p = &x;
```

แต่ขอแนะนำว่า ควรจะเขียนตามแบบแรกจะดีกว่า เพราะจะทำให้อ่านและเข้าใจได้ง่าย

8.1.6 การหาขนาดของข้อมูลโครงสร้าง

การหาขนาดหน่วยความจำที่ข้อมูลโครงสร้างต้องการใช้ สามารถทำได้โดยการเรียกใช้โอเปอร์เรเตอร์ `sizeof` ซึ่งจะให้ขนาดหน่วยความจำของข้อมูลสมาชิกรวมกันทั้งหมดในหน่วยของไบต์ เช่น

```
     sizeof(structure_variable)
```

ดังนั้นเราสามารถทราบขนาดของหน่วยความจำที่ตัวแปรแบบโครงสร้างใช้ได้

ตัวอย่างการใช้งานเช่น

```
#include <stdio.h>

struct person {
     char name[40];
     char address[255];
     char phone[16];
     char email[30];
};


struct person_list {
     int                 key;
     struct person       data;
     struct person_list  *next;
} item;

int main()
{
     struct person_list  List;
```

```
    printf("Size of 'struct person_list'= %3d bytes\n",
           sizeof(List));
    printf("Size of member 'key'= %3d bytes\n",
           sizeof(List.key));
    printf("Size of member 'data'= %3d bytes\n",
           sizeof(List.data));
    printf("Size of member 'next'= %3d bytes\n",
           sizeof(List.next));
    printf("Size of 'struct person'= %d bytes\n",
           sizeof(struct person));
    return 0;
}
```

ขนาดของหน่วยความจำที่โครงสร้างต้องการจะเท่ากับหรือมากกว่าผลรวมของหน่วยความจำที่ข้อมูลสมาชิกแต่ละตัวต้องการใช้

8.1.7 การผ่านโครงสร้างให้ฟังก์ชันและเป็นค่ากลับคืน

คุณสมบัติข้อหนึ่งที่แสดงให้เห็นความแตกต่างระหว่างโครงสร้างและอาร์เรย์ คือเราสามารถผ่านตัวแปรที่เป็นโครงสร้างให้เป็นพารามิเตอร์ของฟังก์ชันใดๆได้โดยไม่ต้องใช้พอยน์เตอร์หรือเป็นค่ากลับคืนจากฟังก์ชันก็ได้ ตัวอย่างเช่น

```
struct record {
     int   key;
     char *str;
};
```

เป็นโครงสร้างที่มีข้อมูลสมาชิกสองตัว และ `func()` เป็นฟังก์ชัน ที่มีพารามิเตอร์หนึ่งตัวเป็นโครงสร้างและให้ค่าที่เป็นโครงสร้างแบบ `struct record` กลับคืน

```
struct record func(const struct record copy)
{
     return copy;
}
```

สำหรับตัวอย่างนี้ การทำงานของฟังก์ชันก็เป็นไปอย่างง่าย คือ ให้ค่าของพารามิเตอร์เป็นค่าของฟังก์ชัน

เราลองมาดูตัวอย่างของโปรแกรมที่เรียกใช้ฟังก์ชันดังกล่าว

```
#include <stdio.h>

struct record {
     int   key;
     char *str;
};

int main()
{
     struct record x = {10,"Hello World!"};

     printf("%s", func(x).str);
     return 0;
}
```

โปรดสังเกตว่า นิพจน์

```
func(x).str
```

หมายถึงการเข้าถึงข้อมูลสมาชิกชื่อ `str` ของค่ากลับคืนที่ได้จากการเรียกใช้ฟังก์ชัน `func(x)` ดังนั้น นิพจน์นี้จึงหมายถึง พอยน์เตอร์ที่ ชี้ไปยังข้อความ ในกรณีนี้คือ `"Hello World!"` หรือถ้าเราจะใช้ ตัวแปรช่วยเหลือ (เช่น y) ก็ทำได้ดังนี้

```
int main()
{
     struct record x = {10,"Hello World!"};
     struct record y;

     y = func(x);
     printf("%s", y.str);
     return 0;
}
```

โดยที่เราผ่านค่าที่ได้จากฟังก์ชัน ไปเก็บไว้ที่ตัวแปร y ก่อน แล้วจึงนำไปใช้ เช่น อ่านค่าของข้อมูลสมาชิกในภายหลัง

8.1.8 โครงสร้างที่มีข้อมูลสมาชิกเป็นอาร์เรย์

ในหัวข้อที่แล้วเราได้นิยามโครงสร้างชื่อ `struct record` โดยผ่านเป็นพารามิเตอร์ของฟังก์ชัน และเป็นแบบข้อมูลสำหรับค่ากลับคืนของฟังก์ชัน เมื่อเรียกใช้ฟังก์ชัน โปรแกรมก็จะทำสำเนาของข้อมูลที่เป็นโครงสร้างจากผู้เรียก แล้วผ่านไปยังภายในฟังก์ชันเพื่อใช้งาน ดังนั้นการเรียกใช้ฟังก์ชันจึงเป็นการเรียก

ใช้โดยการผ่านค่า แต่การผ่านโครงสร้างให้ฟังก์ชันจะมีข้อเสีย คือถ้าโครงสร้างมีขนาดใหญ่ซึ่งหมายความว่า ถ้าโครงสร้างเก็บข้อมูลสมาชิกต่างๆไว้ในตัว และใช้หน่วยความจำโดยรวมเป็นจำนวนมากหลายไบต์แล้ว โปรแกรมก็จะต้องเสียเวลามากขึ้นในการทำสำเนาค่าของโครงสร้างเพื่อผ่านให้กับฟังก์ชันที่ถูกเรียกใช้ ตัวอย่างเช่น ถ้าเรานิยามโครงสร้าง `struct record` ใหม่เป็น

```
typedef struct record {
     int   key;
     char  block_1[512];
     char  block_2[512];
} Record;
```

ในกรณีนี้ขนาดของโครงสร้างจะมีค่าอย่างน้อยหรือเท่ากับ 1026 ไบต์โดยรวม ซึ่งถือว่ามีขนาดค่อนข้างใหญ่ เราลองเปรียบเทียบสองฟังก์ชันต่อไปนี้

```
Record func1(Record s)
{
     s.key++;
     return s;
}

Record *func2(Record *s)
{
     s->key++;
     return s;
}
```

ฟังก์ชันแรกมีพารามิเตอร์และค่ากลับคืนที่อยู่ในรูปของโครงสร้าง ในขณะที่ฟังก์ชันที่สองต้องการพารามิเตอร์และค่ากลับคืนที่เป็นพอยน์เตอร์ที่ชี้ไปยังโครงสร้าง ดังนั้นการเรียกใช้ฟังก์ชันแบบแรกจึงเป็นการเรียกใช้โดยผ่านค่า และการเรียกใช้ฟังก์ชันแบบที่สองจึงเป็นการเรียกใช้โดยผ่านตัวอ้างอิง

การเรียกใช้โดยผ่านตัวอ้างอิงจะมีประสิทธิภาพในการทำงานมากกว่า เพราะโปรแกรมไม่ต้องเสียเวลาทำสำเนาค่าในหน่วยความจำของโครงสร้าง สรุปได้ว่าถ้าเราต้องการผ่านข้อมูลแบบโครงสร้างให้แก่ฟังก์ชัน เราควรจะผ่านโดยการอ้างอิงในรูปของพอยน์เตอร์ที่ชี้ไปยังโครงสร้าง เช่นเดียวกับการผ่านค่ากลับคืนโดยใช้พอยน์เตอร์ แต่อย่างไรก็ตาม ก็ยังขึ้นอยู่กับจุดประสงค์ของการเขียนฟังก์ชันขึ้นใช้งาน โปรดสังเกตว่าการทำงานของทั้งสองฟังก์ชันให้ผลแตกต่างกัน

8.1.9 โครงสร้างที่มีข้อมูลสมาชิกเป็นโครงสร้าง

นอกจากที่โครงสร้างจะมีสมาชิกเป็นข้อมูลแบบพื้นฐานแล้ว เราก็ยังสามารถกำหนดให้ข้อมูลสมาชิกเป็นโครงสร้างได้ โดยที่เรานิยามโครงสร้างขึ้นใหม่ ภายในโครงสร้างใดๆ ตัวอย่างเช่น

```
struct person_list {
     int key;
     struct person_list *next;

     struct person {
          char name[40];
          char address[255];
          char phone[16];
          char email[30];
     } data;

} item;
```

จากตัวอย่างข้างบน โครงสร้าง person_list เป็นโครงสร้างที่มีสมาชิก 3 ตัว คือ สมาชิก key เป็นตัวแปรแบบ int สมาชิก next เป็นพอยน์เตอร์ที่ชี้ไปยังโครงสร้างแบบ struct person_list และสมาชิกตัวที่สาม คือ data เป็นข้อมูลโครงสร้างแบบ struct person และได้แจ้งใช้ ตัวแปร item เป็นโครงสร้างแบบ struct person_list

เนื่องจากว่า เราได้นิยามโครงสร้าง struct person ภายในบล็อกของโครงสร้าง ดังนั้น แบบข้อมูลที่เป็นโครงสร้างนี้ จึงใช้ได้แต่เฉพาะในโครงสร้างของ person_list เท่านั้น ดังนั้นถ้าเราพยายามแจ้งใช้ตัวแปรใดภายนอกที่เป็นโครงสร้างแบบ struct person จึงไม่สามารถทำได้

8.1.10 อาร์เรย์ ที่มีข้อมูลสมาชิกเป็นโครงสร้าง

ถ้าเราต้องการนำข้อดีของการใช้อาร์เรย์และโครงสร้างมาประกอบเข้าด้วยกัน เราก็จะกำหนดใช้อาร์เรย์ที่มีข้อมูลแต่ละตัวเป็นโครงสร้าง ตัวอย่างเช่น

```
#define  MAX_SIZE   100

struct ratio {
     long  num;
     long  den;
};
```

```
int  i;
struct ratio  table[MAX_SIZE];
```

ในตัวอย่างนี้ ตัวแปร table เป็นอาร์เรย์ที่สามารถเก็บข้อมูลที่เป็นโครงสร้างแบบ struct ratio ได้รวมทั้งหมด 100 จำนวน สมมุติว่า เราต้องการเก็บ ลำดับของตัวเลขเศษส่วนต่อไปนี้

$$1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots, \frac{1}{99}, \frac{1}{100}$$

ไว้ในตัวแปร table เราก็สามารถเขียนขั้นตอนการทำงานได้ดังนี้

```
for(i=0; i < MAX_SIZE; i++)
{
     table[i].num = 1;
     table[i].den = (i+1);
}
```

นอกจากที่เราจะใช้ ตัวแปรที่เป็นอาร์เรย์โดยตรงแล้ว เราอาจจะใช้ตัวแปรที่เป็นพอยน์เตอร์และมีการจัดสรรหน่วยความจำด้วยตนเองก็ได้ เช่น ใช้ฟังก์ชัน malloc() หรือ calloc() เป็นต้น ถ้าเราไม่แจ้งตัวแปร table ให้เป็นอาร์เรย์ แต่จะกำหนดให้เป็นพอยน์เตอร์แบบ struct ratio แล้ว ขั้นตอนแรก คือจะต้องจัดหาหน่วยความจำมาเพื่อให้ตัวแปร table สามารถทำหน้าที่คล้ายอาร์เรย์ ได้

```
const int max_size = 100;
struct ratio *table;

table = (struct ratio *)
      malloc(sizeof(struct ratio) * max_size);
```

จากขั้นตอนการทำงานข้างบน เราใช้ฟังก์ชัน malloc() ในการจัดสรรหน่วยความจำสำหรับเก็บข้อมูลที่เป็นโครงสร้างแบบ struct ratio ทั้งหมด 100 จำนวน และกำหนดให้ตัวแปร table เป็นพอยน์เตอร์ชี้ไปยังที่อยู่เริ่มต้นของหน่วยความที่ได้จัดสรรนี้ ถ้าการจัดสรรหน่วยความจำเป็นผลสำเร็จ เราก็สามารถใช้พอยน์เตอร์ table ในเชิงของอาร์เรย์ได้

8.1.11 การเรียงข้อมูลในอาร์เรย์ของโครงสร้าง

เราได้เรียนรู้วิธีการเรียงข้อมูลพื้นฐานในอาร์เรย์ไปบ้าง แล้ว ในตอนนี้ เราลองมาทำความเข้าใจในเรื่องของการเรียงข้อมูลแบบโครงสร้างในอาร์เรย์ ก่อนอื่นเราจะต้องตั้งคำถามก่อนว่าเราจะเปรียบเทียบข้อมูลแบบใด เพราะโครงสร้างหนึ่งๆจะมีข้อมูลสมาชิกได้หลายตัว และเราจะใช้ข้อมูลสมาชิกใดในการ

เปรียบเทียบ ซึ่งจะเป็นตัวกำหนดว่าข้อมูลรวมแบบโครงสร้างในอาร์เรย์จะถูกเรียงตามลำดับอย่างไร ตัวอย่างเช่น ในการสอบแข่งขันของนักเรียนระดับมัธยมศึกษาตอนปลายทั่วประเทศ ซึ่งประกอบด้วยวิชาสอบหลายวิชา อาทิเช่น วิชาคณิตศาสตร์ วิทยาศาสตร์ เป็นต้น ถ้าผู้ดำเนินการสอบแข่งขันต้องการจะประกาศผลของการสอบ ก็สามารถเลือกประกาศผลสอบตามรายชื่อ หมายเลขผู้สอบหรือแบ่งแยกตามคะแนนในแต่วิชาก็ได้ ถ้าเราจะต้องเขียนโปรแกรมเพื่อแก้ปัญหาดังกล่าว เราจะดำเนินการอย่างไร

สมมุติว่า เรามีโครงสร้างที่มีลักษณะดังต่อไปนี้

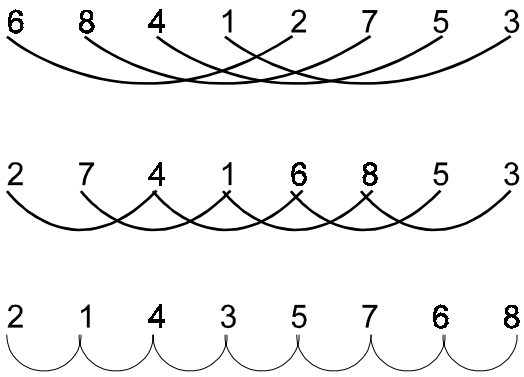
```
struct record {
     int   key;
     char *str;
};
```

และมีอาร์เรย์หนึ่งที่เก็บข้อมูลแบบโครงสร้างหลายๆตัว เราจะเรียงโครงสร้างเหล่านี้ โดยการพิจารณาค่าของข้อมูลสมาชิก key ดังนั้นเราสรุปได้ว่า เราจะเรียงข้อมูลที่เป็นโครงสร้างโดยการเปรียบเทียบค่าของข้อมูลสมาชิก key แต่ละตัว

สำหรับวิธีการเรียงข้อมูลในคราวนี้ เราจะใช้วิธีการที่เรียกว่า Shell Sort ซึ่งมีประสิทธิภาพมากกว่า Bubble Sort ในการทำงานทั่วๆไป ดังนั้นเราลองทำความเข้าใจการทำงานของ Shell Sort ก่อน

Shell Sort เป็นวิธีการเรียงข้อมูลที่คิดค้นโดย Donald Shell ในปี ค.ศ. 1959 ดังนั้น Shell Sort จึงเรียกตามชื่อผู้คิดค้นนั้นเอง

การทำงานของ Shell Sort สังเกตได้ง่ายจากรูปภาพข้างล่างนี้



รูปภาพที่ 8.1 แสดงระยะห่างของตัวเลขที่จะต้องเปรียบเทียบกันเป็นคู่ๆ

ระยะห่างระหว่างข้อมูลสองตัวเรียกว่า gap ในกรณีตัวอย่างนี้ เรามีข้อมูลอยู่ 8 ตัว ค่าเริ่มต้นของ gap จะเท่ากับ 4 และลดลงเรื่อยๆ เป็น 2 และ 1 ตามลำดับ การเรียงข้อมูลจะอาศัยการเปรียบเทียบข้อมูลสองตัวที่มีระยะห่างเท่ากับ gap ที่กำหนด ถ้าเรียงข้อมูลจากน้อยไปมาก ก็จะเปรียบเทียบดูว่า ข้อมูลตัวแรกมีค่ามากกว่าข้อมูลตัวที่สองหรือไม่ ถ้ามากกว่า ก็ให้สลับที่กัน

ตัวเลขที่ขีดเส้นใต้ จะเปรียบเทียบกับตัวเลขถัดไปที่มีระยะห่างเท่ากับ gap ไปทางซ้ายมือ และจะสลับที่กันถ้าตัวเลขที่ขีดเส้นใต้มีค่าน้อยกว่าตัวเลขอีกตัวที่เปรียบเทียบกัน สำหรับการทำงานในแต่ละขั้นตอนเราสามารถสังเกตได้จากลำดับของตัวเลขในตารางต่อไปนี้

6	8	4	1	**2**	7	5	3
2	**8**	4	1	6	**7**	5	3
2	7	**4**	1	6	8	**5**	3
2	7	4	**1**	6	8	5	**3**
2	7	4	1	6	8	5	3
หลังจากจบรอบแรก (gap เท่ากับ 4)							

ตารางที่ 8.1 แสดงตัวอย่างการเรียงแถวตัวเลขโดยใช้ Shell Sort รอบแรก

2	7	**4**	1	6	8	5	3
2	**7**	4	**1**	6	8	5	3
2	1	**4**	7	**6**	8	5	3
2	**1**	4	**7**	6	**8**	5	3
2	1	**4**	7	**6**	8	**5**	3
2	**1**	4	**7**	5	**8**	6	**3**
2	1	4	3	5	7	6	8
หลังจากจบรอบที่สอง (gap เท่ากับ 2)							

ตารางที่ 8.2 แสดงตัวอย่างการเรียงแถวตัวเลขโดยใช้ Shell Sort รอบที่สอง

2	**1**	4	3	5	7	6	8
1	**2**	**4**	3	5	7	6	8
1	**2**	**4**	**3**	5	7	6	8
1	**2**	**3**	**4**	**5**	7	6	8
1	**2**	**3**	**4**	**5**	**7**	6	8
1	**2**	**3**	**4**	**5**	**7**	**6**	8
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	2	3	4	5	6	7	8
หลังจากจบรอบที่สาม (gap เท่ากับ 1)							

ตารางที่ 8.3 แสดงตัวอย่างการเรียงแถวตัวเลขโดยใช้ Shell Sort รอบที่สาม

เพื่อให้เห็นการทำงานในแต่ละขั้นอย่างชัดเจน ลองพิจารณา ขั้นตอนสุดท้ายของรอบที่สองที่มีตัวเลขดังต่อไปนี้

2 **1** 4 **7** 5 **8** 6 **3**

ในกรณีนี้ ค่าของ gap มีค่าเท่ากับ 2 เราเริ่มจากตัวเลขที่ขีดเส้นใต้ คือเลข 3 แล้วเปรียบเทียบกับเลข 8 เนื่องจากว่า 8 มีค่ามากกว่า 3 จึงต้องสลับที่กัน แต่ยังไม่จบเท่านั้น เราจะต้องดูต่อไปว่า 3 กับ 7 ตัวใดมีค่ามากกว่ากัน เพราะ 7 มีค่ามากกว่า 3 จึงต้องสลับที่กัน และท้ายสุด จะต้องเปรียบเทียบกับเลข 1 ก่อน แต่เนื่องจาก 1 มีค่าน้อยกว่า 3 จึงไม่มีการสลับที่กัน และผลที่ได้ในรอบนี้ คือ

2 **1** 4 **3** 5 **7** 6 **8**

เมื่อเห็นการทำงานของ Shell Sort แล้ว เราก็สามารถเขียนฟังก์ชันได้ตามรูปแบบต่อไปนี้

```
void ShellSort (int a[], const int n)
{
     register int i, j;
     int gap, tmp;

     for (gap = n/2; gap > 0; gap /= 2)
        for (i=gap; i < n; i++)
        {
           tmp = a[i];
           for (j=i-gap; j>=0 && a[j] > tmp; j-=gap)
               a[j+gap] = a[j];

           a[j+gap] = tmp;
        }
}
```

ในทางปฏิบัติ เราสามารถดัดแปลงฟังก์ชัน `ShellSort()` ให้มีประสิทธิภาพในการทำงานมากขึ้น โดยการเปลี่ยนแปลงค่าของ gap ในแต่ละรอบ ซึ่งแทนที่จะลดลงทีละครึ่ง (2 เท่า)

```
gap /= 2
```

เป็นการลดลงทีละ 2.2 เท่า ถ้า gap ยังคงมีค่ามากกว่า 2

```
gap = (gap==2) ? 1 : (gap / 2.2)
```

แต่ถ้า gap ลดลงจนมีค่าเท่ากับสองแล้ว ค่าต่อไปสำหรับ gap จะต้องเป็นหนึ่งเสมอ

เมื่อทราบแล้วว่า เราจะเขียนฟังก์ชันที่ใช้ในการเรียงข้อมูลแบบ Shell Sort อย่างไรแล้ว เราก็กลับมายังปัญหาที่ค้างไว้ นั้นคือ การเรียงข้อมูลแบบโครงสร้างในอาร์เรย์ โดยใช้ Shell Sort เพื่อที่จะเขียนฟังก์ชันให้ใช้

ได้กับอาร์เรย์ ที่มีข้อมูลแบบ (โครงสร้าง) ใดๆ เราจะพยายามเขียนฟังก์ชัน `ShellSort()`ให้มีรูปแบบคล้ายกับฟังก์ชัน `BubbleSort()` ที่เราได้เรียนรู้ไป คือ มีการผ่านพอยน์เตอร์ที่ชี้ไปยังฟังก์ชันสำหรับใช้ในการเปรียบเทียบข้อมูล และใช้พอยน์เตอร์อเนกประสงค์สำหรับอาร์เรย์

```
void ShellSort
  (  void *array,
     const unsigned int size,
     const unsigned int n,
     int (*is_greater)(void *, void *) )
{
     register int i, j;
     int gap;
     char *tmp = (char *)malloc(size);
     char *a   = (char *)array;

     for (gap = n/2; gap > 0; gap /= 2)
       for (i = gap; i < n; i++)
       {
           memcpy((void *)tmp,
                  (void *)(a + i*size), size);

           for (j=i-gap; j>=0; j-=gap)
           {
               if(is_greater((void *)(a+j*size), tmp))
                    memcpy((void *)(a + (j+gap)*size),
                           (void *)(a + j*size), size);
               else
                    break;
           }
           memcpy((void *)(a + (j+gap)*size),
                  (void *)tmp, size);
       }
     free(tmp);
}
```

ฟังก์ชันนี้ เราสามารถใช้ในการเรียงข้อมูลของอาร์เรย์ใดๆก็ได้ โดยไม่ขึ้นอยู่กับแบบของข้อมูลสมาชิก แต่ที่สำคัญ คือ เวลาจะเรียกใช้ฟังก์ชันนี้ จะต้องนิยามฟังก์ชันที่ใช้ในการเปรียบเทียบข้อมูลแต่ละตัวก่อน

ดังนั้น ถ้าเราต้องการจะใช้ ฟังก์ชันในการเรียงข้อมูลที่เป็นโครงสร้าง เราก็ต้องสร้างฟังก์ชันที่ใช้ในการเปรียบเทียบระหว่างข้อมูลที่เป็นโครงสร้าง เช่น โครงสร้าง `struct record`

```
int structCompare(void *s1, void *s2)
{
     struct record {
          int   key;
          char *str;
     } *p1, *p2;
```

```
     p1 = (struct record *)s1;
     p2 = (struct record *)s2;
     return (p1->key > p2->key);
}
```

ฟังก์ชัน structCompare() จะทำการเปรียบเทียบข้อมูลสมาชิกชื่อ key ของข้อมูลแบบโครงสร้างทั้งสอง แล้วให้ค่าแบบ int เป็นค่ากลับคืนของฟังก์ชัน โปรดสังเกตว่า พารามิเตอร์ทั้งสองของฟังก์ชัน s1 และ s2 แม้ว่าจะเป็นพอยน์เตอร์อเนกประสงค์ แต่ที่จริงแล้วจะต้องชี้ไปยังแหล่งข้อมูลที่เป็นโครงสร้างแบบ struct record เท่านั้นเพราะภายในฟังก์ชันจะมีการแปลงแบบให้เป็นพอยน์เตอร์แบบ struct record ถ้าแหล่งข้อมูลที่พอยน์เตอร์ทั้งสองชี้ไป ไม่ใช่โครงสร้าง struct record แล้วค่าของข้อมูลสมาชิกที่อ่านได้จะไม่ถูกต้อง

หรือ ถ้าเราต้องการเขียนฟังก์ชันสำหรับการเปรียบเทียบข้อมูลแบบ int เช่น ถ้าอาร์เรย์มีข้อมูลที่เป็นแบบ int และมิใช่โครงสร้าง เราก็เขียนได้ดังนี้

```
int intCompare(void *s1, void *s2)
{
     return (*(int *)s1 > *(int *)s2);
}
```

ตัวอย่างการเรียกใช้ฟังก์ชันทั้งสองและการเรียงข้อมูล

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>

int main()
{
     /** Function Prototypes **/
     int    intCompare (void *, void *);
     int structCompare (void *, void *);

     void ShellSort (
            void *,
            const unsigned int,
            const unsigned int,
            int (*is_greater)(void *, void *)
          );

     /** Variable Declarations and Initializations **/
     struct record {
          int key;
          char *str;
     } a[10] = {
          {5, "Five"},{3, "Three"},{7, "Seven"},
```

```
        {6,  "Six"},{2,   "Two"},{4,  "Four"},
        {1,  "One"},{10,  "Ten"},{8, "Eight"},
        {9, "Nine"}
    };

    int b[10] = {5,3,7,6,2,4,1,10,8,9};
    register int i;


    /** Programm Statements **/
    ShellSort ((void *)a, sizeof(struct record),
               10, structCompare);
    for(i=0; i < 10; i++)
        printf("%s ", a[i].str);
    printf("\n");

    ShellSort ((void *)b, sizeof(int),
               10, intCompare);
    for(i=0; i < 10; i++)
        printf("%d ", b[i]);
    printf("\n");

    return 0;
}
```

ผลการทำงานของโปรแกรม

```
    One Two Three Four Five Six Seven Eight Nine Ten
    1 2 3 4 5 6 7 8 9 10
```

จะเห็นได้ว่า เราสามารถใช้ฟังก์ชัน `ShellSort()` ได้กับอาร์เรย์หลายรูปแบบ ไม่ว่าจะเป็นอาร์เรย์ของข้อมูลแบบ `int` หรืออาร์เรย์ของโครงสร้าง เพียงแต่เราจะต้องสร้างฟังก์ชันที่ใช้ในการเปรียบเทียบอย่างเหมาะสมเท่านั้นเอง

ในโปรแกรมตัวอย่างนี้ เราได้นิยามโครงสร้างและแจ้งใช้ ตัวแปรที่มีแบบข้อมูลตามโครงสร้าง `struct record` และยังได้ติดตั้งค่าเริ่มต้นให้แก่ข้อมูลสมาชิกของโครงสร้างในอาร์เรย์ไปพร้อมกับการแจ้งใช้

```
    struct record {
        int   key;
        char *str;
    } a[10] = {
        {5,"Five"},{3,"Three"},{7,"Seven"},
        {6, "Six"},{2,  "Two"},{4, "Four"},
        {1, "One"},{10, "Ten"},{8,"Eight"},
        {9,"Nine"}
    };
```

ถ้าสังเกตการทำงานของ ฟังก์ชัน ShellSort()ให้ดี เราก็จะเห็นได้ว่า มีการทำสำเนาข้อมูลแต่ละตัวของอาร์เรย์บ่อยครั้ง เมื่อข้อมูลเหล่านั้นต้องสลับที่กัน ซึ่งสังเกตได้จากการเรียกใช้ฟังก์ชัน memcpy() และจุดนี้จะเป็นข้อเสียของการทำงานของฟังก์ชัน ถ้าข้อมูลแต่ละตัวมีขนาดใหญ่มาก เช่น ข้อมูลที่เป็นโครงสร้างตามรูปแบบต่อไปนี้

```
typedef struct record {
     int   key;
     char  str[512];
} Record;
```

วิธีการดัดแปลงให้ฟังก์ชันสามารถทำงานได้มีประสิทธิภาพมากขึ้น คือการใช้อาร์เรย์ของพอยน์เตอร์เข้ามาช่วย และมีการสลับที่ระหว่างข้อมูลแต่ละตัว (ที่มีขนาดใหญ่) น้อยที่สุด โดยเฉพาะอย่างยิ่งในกรณีที่ข้อมูลแต่ละตัวของอาร์เรย์ array มีขนาดใหญ่ เช่น ข้อมูลที่เป็นโครงสร้างขนาดหลายร้อยไบต์

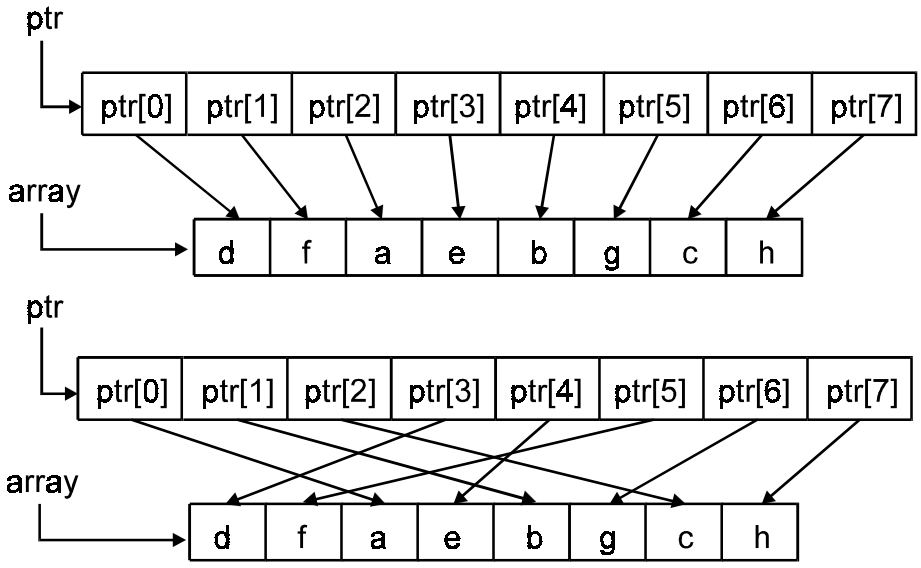
```
void ShellSort
  (  void *array,
     const unsigned int size,
     const unsigned int n,
     int (*is_greater)(void *, void *)
   )
{
     register int i, j;
     int  index, gap;
     char *tmp;
     char *a = (char *)array;

     char **ptr = (char **)malloc(n * sizeof(char *));
     char *copy = (char  *)malloc(size);
     int  *rank = (int   *)malloc(n * sizeof(int));

     for (i = 0; i < n; i++)
          ptr[i] = (char *)(a + i*size);

     for (gap = n/2; gap > 0; gap /= 2)
       for (i = gap; i < n; i++)
       {
          tmp = ptr[i];
          for (j=i-gap; j>=0; j-=gap)
          {
             if(is_greater((void *)ptr[j],(void *)tmp))
               ptr[j+gap] = ptr[j];
             else
               break;
          }
          ptr[j+gap] = tmp;
       }
```

```
     for (i = 0; (i < n); i++) {
          rank[i] = (int)(ptr[i] - a)/ size;
     }

     for (i = 0; i < n; i++) {

       if(rank[i] != i)
       {
            index   = rank[i];
            rank[i] = i;
            memcpy((void *)copy,
                   (void *)ptr[index], size);
            memcpy((void *)ptr[index],
                   (void *)ptr[i], size);
            memcpy((void *)ptr[i],
                   (void *)copy, size);

            for (j=i+1; j < n; j++)
               if (rank[j]== i) break;
            rank[j] = index;
       }
     }

     free(copy);  free(rank);  free(ptr);
}
```

ฟังก์ชันที่เราได้ดัดแปลงใหม่นี้ จะใช้ ptr ซึ่งทำหน้าที่เป็นอาร์เรย์ของพอยน์เตอร์ ที่ชี้ไปยังข้อมูลโครงสร้าง (หรือข้อมูลแบบใดๆ) แต่ละตัวในอาร์เรย์

```
     for (i = 0; i < n; i++) {
          ptr[i] = (char *)(a + i*size);
     }
```

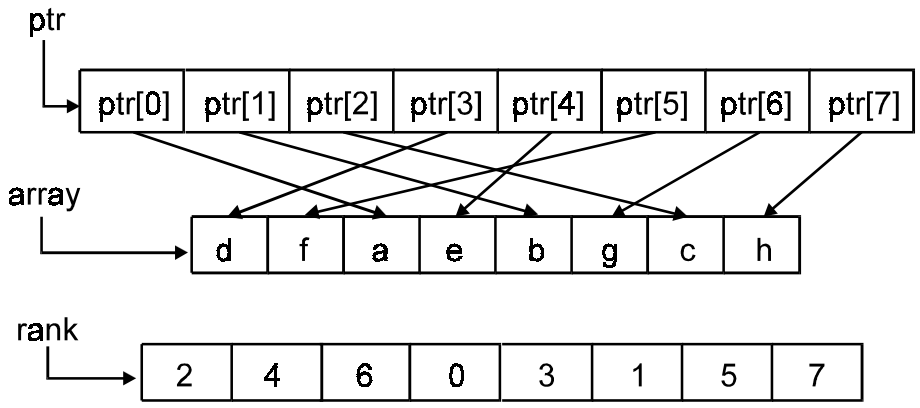
แทนที่จะเข้าถึงข้อมูลแต่ละตัวของอาร์เรย์โดยตรง เราก็ใช้อาร์เรย์ ptr นี้แทนอาร์เรย์ array



รูปภาพที่ 8.2 แสดงตัวอย่างการใช้อาร์เรย์ของพอยน์เตอร์ ในการเรียงข้อมูลจากอาร์เรย์

เนื่องจากข้อมูลแต่ละตัวมีขนาดเท่ากับ `size` ไบต์ ดังนั้น พอยน์เตอร์ `ptr[i]` จึงชี้ไปยังที่อยู่ของหน่วยความจำที่อาร์เรย์ `array` ใช้ และมีระยะห่างจากจุดเริ่มต้นเท่ากับ `i*size` ไบต์ (`i` มีค่าตั้งแต่ `0` ถึง `n-1` โดยที่ `n` เป็นจำนวนของข้อมูลทั้งหมดในอาร์เรย์)

เมื่อได้กำหนดค่าของพอยน์เตอร์ `ptr[i]` แต่ละตัวแล้วเราจึงใช้พอยน์เตอร์เหล่านี้ในการเรียงข้อมูลแบบ Shell Sort เนื่องจากว่าเป็นการใช้พอยน์เตอร์เท่านั้น ดังนั้น จึงไม่มีการทำสำเนาข้อมูลที่มีขนาดใหญ่ และจะเห็นได้ว่า ในขั้นตอนของการเรียงข้อมูลแบบ Shell Sort จะไม่มีการเรียกใช้ฟังก์ชัน `memcpy()` แม้แต่ครั้งเดียว การเรียงข้อมูลในขั้นแรกนี้จึง เป็นกำหนดให้พอยน์เตอร์ `ptr[i]` ชี้ไปยังตำแหน่งต่างๆเท่านั้นเอง



รูปภาพที่ 8.3 การใช้อาร์เรย์ `rank` ในการเก็บตำแหน่งที่ถูกต้องของข้อมูลจากอาร์เรย์

หลังจากการทำงานของ Shell Sort ในขั้นแรกจบลงแล้ว พอยน์เตอร์ `ptr[i]` จะชี้ไปยังข้อมูลแต่ละตัวที่ได้เรียงตามลำดับแล้ว แต่ข้อมูลในอาร์เรย์ `array` ยังมิได้มีการเปลี่ยนแปลงได้ใดๆเลย ซึ่งหมายความว่าเรายังมิได้โยกย้ายตำแหน่งของข้อมูลแต่ละตัว ดังนั้นเราจะต้องมาเรียงข้อมูลในอาร์เรย์อีก และจะเกี่ยวข้องกับการเรียงใช้ฟังก์ชัน `memcpy()` ด้วย แต่เราทราบแล้วว่า ข้อมูลใดควรจะอยู่ในตำแหน่งใด ตามลำดับที่ถูกต้อง เพราะเราได้เรียงข้อมูลเหล่านี้แล้วโดยใช้อาร์เรย์ของพอยน์เตอร์ `ptr` แทน (อาร์เรย์ `ptr` ทำหน้าที่ช่วยเหลือในการเรียงข้อมูลของอาร์เรย์ `array` และจะเป็นตัวบ่งบอกว่า ข้อมูลตัวใดของอาร์เรย์ `array` จะต้องอยู่ในตำแหน่งใด) ดังนั้นการเรียงข้อมูลในอาร์เรย์ `array` จึงทำได้อย่างรวดเร็ว ปัญหาที่เหลืออยู่ จึงเป็น การเรียงข้อมูลในอาร์เรย์ `array` โดยอาศัยค่าของ `ptr[i]` ที่บ่งบอกตำแหน่งที่ถูกต้องนั้นเอง

```
for(i=0; (i < n); i++)
     rank[i] = (int)(ptr[i] - a)/ size;
```

เพื่อความสะดวก จึงได้ใช้อาร์เรย์ชื่อ rank แบบ int และใช้เก็บค่าต่างๆที่บอกตำแหน่งที่ถูกต้องของข้อมูลในอาร์เรย์ array แต่ละตัว

จากรูปภาพประกอบที่ 8.2 และ 8.3 อาร์เรย์ array เก็บข้อมูลที่เราต้องการเรียงลำดับ เช่น ข้อมูลแต่ละตัวเป็นอักษรในภาษาอังกฤษ ในขณะที่อาร์เรย์ ptr เก็บพอยน์เตอร์ที่ชี้ไปยังที่อยู่ของข้อมูลแต่ละตัว หลังจากที่มีการเรียงข้อมูลในช่วงแรกแล้ว พอยน์เตอร์ ptr[i] แต่ละตัวจะชี้ไปยังตำแหน่งที่ถูกต้อง จากนั้นเราก็เก็บไว้ในรูปของอาร์เรย์ rank เช่น rank[0] มีค่าเท่ากับ 2 หมายความว่า array[rank[0]] คือ ตัวอักขระ a และเราจะต้องหาทางย้ายตัวอักขระนี้มาไว้ในตำแหน่ง แรกของอาร์เรย์ array ให้ได้ สำ หรับตัวอักษรอื่นๆ ที่เหลือ เราก็ต้องย้ายไปยังตำแหน่งที่ถูกต้องตามลำดับไป

ตัวอย่างต่อไป เราจะเขียนโปรแกรมตัวอย่าง ที่เราสามารถนำวิธีการไปประยุกต์ใช้ เขียนโปรแกรมสำหรับงานในชีวิตประจำวันได้ เช่น การเรียงคะแนนสอบของนักศึกษาตามวิชาที่สอบ และคะแนนรวม ก่อนอื่นเราจะต้องนิยามโครงสร้างข้อมูลที่เหมาะสม เช่น สมมุติว่า วิชาที่สอบ คือ คณิตศาสตร์ และฟิสิกส์

```
struct  score_record {
     char       studentID[8];
     float      math;
     float      physics;
};
```

ขั้นต่อไปคือ การสร้างฟังก์ชัน ที่ใช้ในการเปรียบเทียบระหว่างข้อมูลโครงสร้าง โดยพิจารณาตามคะแนนสอบในแต่ละวิชา และคะแนนรวม ตามลำดับ

```
int mathCompare(void *s1, void *s2)
{
     struct score_record  *p1, *p2;

     p1 = (struct score_record *)s1;
     p2 = (struct score_record *)s2;
     return (p1->math > p2->math);
}

int physicsCompare(void *s1, void *s2)
{
     struct score_record  *p1, *p2;

     p1 = (struct score_record *)s1;
     p2 = (struct score_record *)s2;
     return (p1->physics > p2->physics);
}

int totalCompare(void *s1, void *s2)
```

```
{
     struct score_record  *p1, *p2;

     p1 = (struct score_record *)s1;
     p2 = (struct score_record *)s2;
     return (p1->math + p1->physics
             > p2->math + p2->physics);
}
```

เพื่อที่จะแสดงให้เห็นวิธีทำงานอย่างคร่าวๆ เราจะพิจารณาตัวอย่างคะแนนสอบของนักศึกษาแค่สิบคน เท่านั้น โดยสมมุติว่า มีค่าอยู่ในอาร์เรย์ `Table[]` ดังนี้

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>

struct score_record Table[] = {
  {"u381023", 32.5, 40.0}, {"u381034", 27.0, 41.0},
  {"u381026", 18.5, 22.5}, {"u381133", 33.0, 31.5},
  {"u381140", 42.5, 29.0}, {"u381009", 26.5, 37.5},
  {"u381085", 26.5, 30.5}, {"u381077", 35.0, 27.5},
  {"u381055", 38.5, 43.0}, {"u381049", 43.0, 45.5}
};

int main()
{
     int i;

     extern void ShellSort( void *,
          const unsigned int, const unsigned int,
          int (*is_greater)(void *, void *));

     ShellSort((void *)Table,
               sizeof(struct score_record), 10,
               mathCompare);
     printf("+---------------+\n");
     printf("|  MATHEMATICS  |\n");
     printf("+---------------+\n");
     for(i=0; i < 10; i++)
          printf("%s %5.1f\n",
                  Table[i].studentID, Table[i].math);

     ShellSort((void *)Table,
               sizeof(struct score_record),10,
               physicsCompare);
     printf("+------------+\n");
     printf("|  PHYSICS   |\n");
     printf("+------------+\n");
     for(i=0; i < 10; i++)
          printf("%s %5.1f\n",
                  Table[i].studentID, Table[i].physics);
```

```
     ShellSort((void *)Table,
               sizeof(struct score_record),
               10, totalCompare);
     printf("+-----------------+\n");
     printf("|  TOTAL SCORE   |\n");
     printf("+-----------------+\n");
     for(i=0; i < 10; i++)
          printf("%s %5.1f\n", Table[i].studentID,
               Table[i].math + Table[i].physics);
     return 0;
}
```

8.1.12 อาร์เรย์ของโครงสร้างที่มีสมาชิกเป็นพอยน์เตอร์ชี้ไปยังฟังก์ชัน

ในหัวข้อนี้ อันที่จริงแล้ว เป็นการยกตัวอย่างการใช้งานของอาร์เรย์ที่มีข้อมูลแบบโครงสร้าง ซึ่งเป็นโครงสร้างที่มีข้อมูลสมาชิกตัวหนึ่งเป็นพอยน์เตอร์สำหรับฟังก์ชัน ตัวอย่างเช่น `FunctionStruct` เป็นชื่อของโครงสร้างที่มีข้อมูลสมาชิกสองตัว คือ `command` และ `Function` ตามลำดับ

```
typedef void * DataPointer;

typedef struct {
     char   *command;
     void  (* Function)(DataPointer);

} FunctionStruct;
```

แนวความคิด ก็คือ เราจะสร้างตารางในรูปของอาร์เรย์ที่มีข้อมูลแต่ละตัวเป็นแบบ `FunctionStruct` โดยทำหน้าที่เป็น Look-Up Table ในทำนองที่ว่า เรามีชุดของคำสั่งที่เป็นฟังก์ชัน อยู่หลายๆตัว โดยมีข้อแม้ในขั้นต้นว่า ฟังก์ชันทุกตัวจะมีรูปแบบของพารามิเตอร์เหมือนกัน เวลาจะเรียกใช้คำสั่งใดๆเราก็กำหนดแค่ชื่อของฟังก์ชัน (ในรูปของสายอักขระ) และข้อมูลสำหรับเป็นพารามิเตอร์ที่จะผ่านให้ฟังก์ชันโดยมีฟังก์ชัน `Execute()`

```
void Execute(char *command, DataPointer data_ptr);
```

เป็นตัวดำเนินการ ฟังก์ชันนี้จะค้นหาคำสั่งในตาราง `Table[]` โดยอาศัยชื่อที่กำหนดโดยพารามิเตอร์ `command` และ เปรียบเทียบกับรายการคำสั่งที่มีอยู่ในตารางทั้งหมด ถ้าค้นพบก็ให้เรียกฟังก์ชันที่เหมาะสมให้ทำงานตามที่ต้องการ ถ้าไม่พบคำสั่งที่ ต้องการก็ให้แจ้ง กรณีดังกล่าวให้ทราบ

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

typedef void * DataPointer;

/** Structure Declaration **/
typedef struct {
     char   *command;
     void  (* Function)(DataPointer);

} FunctionStruct;

/** Function Prototypes **/
void routine_1 (DataPointer);
void routine_2 (DataPointer);
void routine_3 (DataPointer);

/** Variable Declaration **/
static FunctionStruct TableOfCommands[] =
     {
          {"DO_ROUTINE_1", routine_1},
          {"DO_ROUTINE_2", routine_2},
          {"DO_ROUTINE_3", routine_3},
          {NULL, NULL}   /* NULL terminated */
     };

void routine_1(DataPointer pointer)
{
     struct intern_struct {
          int x;
     } *ptr = (struct intern_struct *) pointer;

     if (ptr!=NULL)
          printf("%d\n", ptr->x);
}

void routine_2(DataPointer pointer)
{
     struct intern_struct {
          double x;
     } *ptr = (struct intern_struct *) pointer;

     if (ptr != NULL)
          printf("%lf\n", ptr->x);
}

void routine_3(DataPointer pointer)
{
     struct intern_struct {
          char *s;
     } *ptr = (struct intern_struct *) pointer;

     if (ptr != NULL)
```

```
       printf("%s\n", ptr->s);
}

void Execute(char *command, DataPointer data_ptr)
{
  FunctionStruct *p;

  for(p=TableOfCommands; p->command != NULL; p++)
  {
     if(strcmp(p->command, command)==0)
     {
          p->Function(data_ptr);
          return ;
     }
  }
  printf ("Unrecognized command!\n");
}

int main()
{
     struct int_struct {
          int x;
     } int_data = {501};

     struct double_struct {
          double x;
     } dbl_data = {13.105};

     struct string_struct {
          char *message;
     } str_data = {"Hello World!"};

  Execute("DO_ROUTINE_1", (DataPointer)&int_data);
  Execute("DO_ROUTINE_2", (DataPointer)&dbl_data);
  Execute("DO_ROUTINE_3", (DataPointer)&str_data);

  str_data.message =
     "Array of Structures with Pointers to Functions";
  Execute("DO_ROUTINE_3", (DataPointer)&str_data);

  /** Calling function as follows is illegal !!! **/
  Execute("DO_ROUTINE_1", (DataPointer)&dbl_data);
  Execute("DO_ROUTINE_2", (DataPointer)&int_data);

  return 0;
}
```

ตัวอย่างผลการทำงานของโปรแกรม เช่น

```
     501
     13.105000
     Hello World!
```

```
Array of Structures with Pointers to Functions
10486
0.000000
```

โปรดสังเกตว่า ถ้าเราผ่านค่าพารามิเตอร์ ที่ไม่ถูกต้องตามแบบที่คำสั่งต้องการ ก็จะได้ผลการทำงานที่ไม่ถูกต้อง เช่น

```
Execute("DO_ROUTINE_1", (DataPointer)&dbl_data);
Execute("DO_ROUTINE_2", (DataPointer)&int_data);
```

ถ้ากำหนดเลือก "DO_ROUTINE_1" แต่ผ่านพารามิเตอร์ที่อ้างถึงข้อมูลแบบ double แทนที่จะเป็นแบบ int ก็ถือว่าผิด หรือในทางกลับกัน ถ้ากำหนดเลือก "DO_ROUTINE_2" แต่ผ่านพารามิเตอร์ที่อ้างถึงข้อมูลแบบ int แทนที่จะเป็นแบบ double ก็ถือว่าผิดเช่นกัน ดังนั้นผลการทำงานของฟังก์ชันที่เกิดจากการกระทำคำสั่งทั้งสองจึงไม่ถูกต้อง

การใช้งานอาร์เรย์ที่เก็บข้อมูลแบบโครงสร้างสำหรับเลือกกระทำคำสั่งต่างๆในลักษณะนี้ เราจะได้พบและใช้งานอีกในการสร้าง ตัวแปลคำสั่ง ซึ่งเป็นส่วนหนึ่งของการสร้างเครื่องคิดเลขอย่างง่าย

8.1.13 ตัวอย่างการใช้งานโครงสร้างและการเขียนฟังก์ชันขึ้นใช้

สมมุติว่า เราต้องการเขียนฟังก์ชันสำหรับการคำนวณเกี่ยวกับเลขจำนวนเชิงซ้อนที่เก็บอยู่ในรูปของโครงสร้าง struct Complex ซึ่งมีรูปแบบ เช่น

```
struct Complex {
     double   Re;
     double   Im;
};
```

รวมทั้ง ฟังก์ชันที่เกี่ยวข้องกับการคำนวณเลขเชิงซ้อน เช่น การบวกลบ คูณ หาร ระหว่างเลขจำนวนเชิงซ้อนสองตัว เป็นต้น และรวมถึงการคำนวณค่าคอนจูเกต (Conjugate) ค่าสัมบรูณ์ และค่าอาร์กิวเมนต์ของเลขจำ นวนเชิงซ้อนใดๆ

ถ้าเราต้องการเขียนฟังก์ชันเหล่านี้ และกำหนดให้เป็นฟังก์ชันมาตราฐาน ซึ่งเราสามารถนำไปใช้งานได้ในจุดประสงค์ทั่วไป เราก็จะแบ่งโปรแกรมออกเป็น สามส่วน ส่วนแรกคือ ไฟล์ส่วนหัว Complex.h ส่วนที่สองชื่อ Complex.c และส่วนที่สามเป็นไฟล์ที่เก็บโปรแกรมโค้ดที่มีการเรียกใช้ฟังก์ชันสำหรับจำนวนเชิง

ซ้อน เช่น ในกรณีตัวอย่างนี้ เราจะให้ชื่อว่า testcmpl.c มีจุดประสงค์สำหรับทดลองและตรวจสอบการเรียกใช้ฟังก์ชันที่ได้สร้างขึ้น

1) ไฟล์ส่วนหัว Complex.h

เราจะรวบรวมรายละเอียดต่างๆ เช่น รายการฟังก์ชันต่างๆ ที่เราจะสร้างขึ้นใช้ โดยเขียนไว้ในรูปของฟังก์ชันโปรโตไทพ์ (Function Prototype) รวมทั้ง แบบข้อมูลสำหรับจำนวนเชิงซ้อน `Complex` เป็นต้น ภายในไฟล์ Complex.h จะมีเนื้อหาดังต่อไปนี้

File name : **Complex.h**

```
#ifndef  _COMPLEX_H
#define  _COMPLEX_H

/* incomplete structure declaration */
typedef struct Complex *Complex;

/* Function Prototypes */
Complex cInit (double real_part, double imag_part);
void cAdd (const Complex a, const Complex b, Complex ret);
void cSub (const Complex a, const Complex b, Complex ret);
void cMul (const Complex a, const Complex b, Complex ret);
void cDiv (const Complex a, const Complex b, Complex ret);
void cInv (const Complex a, Complex ret);
void cConj(const Complex a, Complex ret);

double cAbs(const Complex a);
double cArg(const Complex a);
double getReal(const Complex a);
double getImag(const Complex a);

void cPrint (const Complex a, int digits);

#endif  /* _COMPLEX_H */
```

โปรดสังเกตว่า เราได้นิยามแบบข้อมูล Complex ที่ใช้แทนที่แบบของพอยน์เตอร์สำหรับโครงสร้างแบบ `struct Complex` แต่เรายังมิได้นิยามโครงสร้างนี้ การนิยามโครงสร้าง `struct Complex` เราจะกระทำในไฟล์ส่วนตัว Complex.c ซึ่งมีเหตุผลที่สำคัญคือ การซ่อนรายละเอียดเกี่ยวกับโครงสร้างนี้

นอกจากนี้ ยังได้มีการใช้คำสั่งของพรีโปรเซสเซอร์ไดเรคทีฟ ทั้งนี้ ก็เพื่อเป็นการตรวจสอบว่า มีการแทรกไฟล์ส่วนหัว Complex.h แล้วหรือยัง ถ้ามีการนิยามแมโครชื่อ `_COMPLEX_H` แล้ว ก็หมายความว่า ในขณะที่ทำการคอมไพล์ ได้มีการอ่านเนื้อหาของไฟล์นี้แล้ว ดังนั้นเพื่อป้องกันการอ่านและแทรกไฟล์นี้ซ้ำซ้อน

จึงได้กำหนดเงื่อนไขในการคอมไพล์สำหรับไฟล์นี้ไว้ ถ้ายังไม่มีการนิยามแมโครนี้ ก็ให้อ่านเนื้อหาของไฟล์ส่วนหัวที่เหลือทั้งหมด ถ้ามีการนิยามแมโครนี้แล้ว ก็ให้ข้ามขั้นตอนนี้ไป

2) ไฟล์ส่วนตัว Complex.c

ในไฟล์นี้ เราจะนิยามและสร้างฟังก์ชันต่างๆที่ได้แจ้งไว้เป็นต้นแบบในไฟล์ส่วนหัว Complex.h รวมทั้งนิยามโครงสร้าง `struct Complex` ด้วย

File name : **Complex.c**

```
#include <stdio.h>
#include <assert.h>
#include <malloc.h>
#include <math.h>
#include "Complex.h"

struct Complex {
     double   Re;
     double   Im;
};

Complex cInit (double real_part, double imag_part)
{
     Complex new_c = (Complex)malloc(sizeof(struct Complex));
     new_c->Re = real_part;
     new_c->Im = imag_part;
     return new_c;
}

void cPrint (const Complex a, int digits)
{
     int n = (digits < 0 || digits > 10) ? 6 : digits;
     char format[30];

     assert(a);
     if(a->Im < 0) {
          sprintf(format, "%%.%dlf - j%%.%dlf\n", n, n);
          printf(format, a->Re, -(a->Im));
     }
     else {
          sprintf(format, "%%.%dlf + j%%.%dlf\n", n, n);
          printf(format, a->Re, a->Im);
     }
}

void cAdd (const Complex a, const Complex b, Complex ret)
{
     assert(a && b && ret);
```

```
     ret->Re = a->Re + b->Re;
     ret->Im = a->Im + b->Im;
}

void cSub (const Complex a, const Complex b, Complex ret)
{
     assert(a && b && ret);
     ret->Re = a->Re - b->Re;
     ret->Im = a->Im - b->Im;
}

void cMul (const Complex a, const Complex b, Complex ret)
{
     assert(a && b && ret);
     ret->Re = (a->Re * b->Re) - (a->Im * b->Im);
     ret->Im = (a->Re * b->Im) + (a->Im * b->Re);
}

void cInv (const Complex a, Complex ret)
{
     double div;

     assert(a && ret);
     div = (a->Re * a->Re) + (a->Im * a->Im);
     ret->Re =  a->Re / div;
     ret->Im = -(a->Im) / div;
}

void cDiv (const Complex a, const Complex b, Complex ret)
{
     struct Complex tmp;
     Complex p = &tmp;

     assert(a && b && ret);
     cInv(b, p);
     cMul(a, p, ret);
}

void cConj (const Complex a, Complex ret)
{
     assert(a && ret);
     ret->Re =  a->Re;
     ret->Im = -(a->Im);
}

double cAbs(const Complex a)
{
     double re = a->Re, im = a->Im;

     assert(a);
     return sqrt(re*re + im*im);
}
```

```
double cArg(const Complex a)
{
     assert(a);
     return atan2(a->Re, a->Im);
}

double getReal(const Complex a)
{
     assert(a);
     return a->Re;
}

double getImag(const Complex a)
{
     assert(a);
     return a->Im;
}
```

จากการสร้างฟังก์ชันตามวิธีการข้างต้น เราได้ใช้ ฟังก์ชัน `assert()`

```
assert(a && b && ret);
```

ในการตรวจสอบพารามิเตอร์ของฟังก์ชัน ซึ่ง กำหนดไว้ว่า พารามิเตอร์ `a` `b` และ `ret` จะต้องเป็นพอยน์เตอร์แบบ Complex ที่ไม่ใช่พอยน์เตอร์ศูนย์ ประโยคคำสั่งข้างบนนี้ อาจจะเขียนใหม่ได้เป็น

```
assert(a!=NULL && b!=NULL && ret!=NULL);
```

แต่เพื่อที่จะเขียนให้สั้นและกระทัดรัด เราจะเขียนตามแบบแรก

การสร้างฟังก์ชันสำหรับคำนวณค่าต่างๆ ของเลขจำนวนเชิงซ้อน จะอาศัยความรู้พื้นฐานทางคณิตศาสตร์ที่สรุปได้ดังนี้ กำหนดให้ a และ b เป็นเลขจำนวนเชิงซ้อนใดๆ ดังนั้น เราจะเขียนได้ว่า

$$a = \mathrm{Re}\{a\} + j\,\mathrm{Im}\{a\}$$
$$b = \mathrm{Re}\{b\} + j\,\mathrm{Im}\{b\}$$

สำหรับการบวกลบระหว่าง a และ b ก็ทำได้ตามหลักการต่อไปนี้

$$a + \mathrm{b} = (\mathrm{Re}\{a\} + \mathrm{Re}\{b\}) + j(\mathrm{Im}\{a\} + \mathrm{Im}\{b\})$$
$$a - \mathrm{b} = (\mathrm{Re}\{a\} - \mathrm{Re}\{b\}) + j(\mathrm{Im}\{a\} - \mathrm{Im}\{b\})$$

และสำหรับการคูณและหาร

$$a \cdot \mathrm{b} = (\mathrm{Re}\{a\}\cdot\mathrm{Re}\{b\} - \mathrm{Im}\{\mathrm{a}\}\cdot\mathrm{Im}\{\mathrm{b}\}) + j(\mathrm{Re}\{a\}\cdot\mathrm{Im}\{b\} + \mathrm{Im}\{a\}\cdot\mathrm{Re}\{b\})$$

$$a \div \mathrm{b} = (\mathrm{Re}\{a\} + j\,\mathrm{Im}\{a\})\cdot\frac{(\mathrm{Re}\{b\} - j\,\mathrm{Im}\{b\})}{[\mathrm{Re}\{b\}]^2 + [\mathrm{Im}\{b\}]^2}$$

การหาค่าคอนจูเกต ค่าสัมบรูณ์ และค่าอาร์กิวเมนต์ของจำนวนเชิงซ้อน ทำได้โดย

$$\overline{a} = \overline{\mathrm{Re}\{a\} + j\,\mathrm{Im}\{a\}} = \mathrm{Re}\{a\} - j\,\mathrm{Im}\{a\}$$

$$|a| = \sqrt{[\mathrm{Re}\{a\}]^2 + [\mathrm{Im}\{a\}]^2}$$

$$\angle a = \begin{cases} \arctan(\dfrac{\mathrm{Im}\{a\}}{\mathrm{Re}\{a\}}), & \mathrm{Re}\{a\} \ge 0 \\ \pi + \arctan(\dfrac{\mathrm{Im}\{a\}}{\mathrm{Re}\{a\}}), & \mathrm{Re}\{a\} < 0 \end{cases}$$

สำหรับการหาค่าอาร์กิวเมนต์ เพื่อความสะดวกเราจะเรียกใช้ฟังก์ชันมาตราฐานชื่อ `atan2()` ที่นิยามไว้ใน `<math.h>`ในการคำนวณ

3) ไฟล์สำหรับโปรแกรมทดสอบ testcmpl.c

สำหรับการใช้งานฟังก์ชันต่างๆที่สร้างขึ้น เราก็ทำได้ดังนี้ ตัวอย่างเช่น เราเขียนโปรแกรมโค้ดเก็บไว้ในแฟ้มข้อมูล testcmpl.c

File name : **testcmpl.c**

```
#include "Complex.h"

int main()
{
  Complex z1 = cInit(5,2),
          z2 = cInit(-1,1),
          z3 = cInit(0,0),
          z4 = cInit(0,0);

     cAdd(z1, z2, z3); cPrint(z3,3);
     cSub(z1, z2, z3); cPrint(z3,3);
     cMul(z1, z2, z3); cPrint(z3,3);
     cDiv(z1, z2, z3); cPrint(z3,3);

     cInv (z1, z3);
     cMul (z1, z3, z4); cPrint(z4,3);

     z1 = cInit(-4,3);  cPrint(z1,3);
     printf ("Abs(z1) = %.3lf\n", cAbs(z1));
```

```
    printf ("Arg(z1) = %.3lf deg\n",
              180 * cArg(z1) /3.1415926);

    return 0;
}
```

ภายในไฟล์นี้ เราไม่สามารถแจ้งใช้ ตัวแปรแบบ struct Complex ได้เพราะเป็นแบบโครงสร้างที่ไม่สมบรูณ์ ในไฟล์ส่วนหัว Complex.h เราได้อ้างถึงโครงสร้างนี้ โดยใช้ในการนิยามแบบข้อมูลชื่อ Complex

```
typedef struct Complex *Complex;
```

แต่โครงสร้าง struct Complex นี้จะถูกนิยามไว้ในภายหลัง ในไฟล์ Complex.c สรุปได้ว่า โครงสร้าง struct Complex จะใช้ได้เฉพาะภายในไฟล์ Complex.c เท่านั้น และนี้เป็นข้อดีของการซ่อนรายละเอียดที่เกี่ยวกับโครงสร้าง struct Complex เพราะในกรณีนี้ โปรแกรมภายในไฟล์ testcmpl.c เป็นการใช้งานของฟังก์ชันสำหรับจำนวนเชิงซ้อน จุดประสงค์หลักคือ เราต้องการจะแยกระหว่าง “การสร้างฟังก์ชัน” และ “การเรียกใช้ฟังก์ชัน” และเพื่อป้องกันมิให้ผู้ที่จะนำฟังก์ชันเหล่านี้ไปใช้ ไปยุ่งเกี่ยวกับรายละ เอียดต่างๆของโครงสร้าง struct Complex เราจึงได้นิยามโครงสร้าง struct Complex ไว้ในไฟล์ Complex.c

ตามปรกติแล้ว การสร้างฟังก์ชันอเนกประสงค์ เช่น ฟังก์ชันสำหรับคำนวณเกี่ยวกับเลขจำนวนเชิงซ้อนที่เราได้สร้างขึ้นนั้น จะถูกแปลงเป็นออปเจคโค้ด และเก็บไว้ในคลังของฟังก์ชัน ซึ่งหมายความว่า เราจะทำการแปลงเนื้อหาที่อยู่ในไฟล์ Complex.c เป็นออปเจคโค้ดแล้วเก็บไว้ในคลังฟังก์ชัน เมื่อผู้ใช้ ต้องการใช้ฟังก์ชันดังกล่าวก็ให้แทรกไฟล์ส่วนหัว Complex.h เข้าในโปรแกรมโค้ดของตน

```
#include "Complex.h"
```

เพื่อจะได้ทราบว่า ฟังก์ชันเหล่านี้ มีรูปแบบเป็นอย่างไร และก็สามารถเรียกใช้ฟังก์ชันตามจุดประสงค์ที่ต้องการได้ เมื่อเวลาทำการคอมไพล์โปรแกรมโค้ด ผู้ใช้ จะต้องกำหนดด้วยว่า จะให้คอมไพล์เลอร์ค้นหาออปเจคโค้ดสำหรับฟังก์ชันเหล่านั้นในคลังของฟังก์ชันชื่ออะไรและจะพบได้ที่ใด เมื่อคอมไพล์เลอร์ค้นพบออปเจคโค้ดสำหรับฟังก์ชันที่มีการเรียกใช้แล้ว ก็จะนำไปประกอบเข้ากับออปเจคโค้ดของโปรแกรมส่วนอื่นต่อไปเพื่อสร้างโปรแกรมที่ครบสมบรูณ์และสามารถทำงานได้ ดังนั้นสำหรับผู้ใช้แล้ว ไฟล์ Complex.c จึงไม่จำเป็นต้องใช้ ถ้าได้สร้างคลังของฟังก์ชันนี้แล้ว

8.2 ยูเนี่ยน (Union)

แบบข้อมูลรวมอีกชนิดหนึ่ง คือ ยูเนียน (Union) ซึ่งมีลักษณะคล้ายโครงสร้าง ตลอดจนคุณสมบัติต่างๆ ยกเว้นในเรื่องของการจัดเก็บข้อมูลสมาชิกแต่ละตัว

8.2.1 รูปแบบการนิยามและแจ้งใช้ตัวแปรที่เป็นยูเนียน

```
union structure_name {
     data_type1     member_name1;
     data_type2     member_name2;
     data_type3     member_name3;

     ...            ...

     data_typeN     member_nameN;
} variable_list;
```

รูปแบบการนิยามใช้ ตลอดจนการเรียกใช้ และเข้าถึงข้อมูลสมาชิกของยูเนียนจะเหมือนกับของโครงสร้างทุกประการ ตัวอย่าง การนิยามยูเนียน เช่น

```
union alpha {
     int  i;
     char a;
};
```

ซึ่งมีสมาชิกสองตัวคือ ข้อมูลแบบ `int` และ `char` ตามลำดับ

หน่วยความจำสำหรับยูเนียนที่ใช้ จะเท่ากับขนาดของข้อมูลสมาชิกตัวที่ต้องการหน่วยความจำมากที่สุดซึ่งหน่วยความจำนี้จะถูกใช้ร่วมกันระหว่างข้อมูลสมาชิกแต่ละตัว และจะถูกใช้โดยข้อมูลสมาชิกเพียงตัวใดตัวหนึ่งเท่านั้นในช่วงเวลาหนึ่งๆ จากตัวอย่าง ยูเนียน `alpha` มีหน่วยความจำที่ใช้เท่ากับ ขนาดของข้อมูลแบบ `int` ถ้าเรากำหนดให้ `alpha` เป็นโครงสร้างแทนที่จะเป็นยูเนียนแล้ว หน่วยความจำที่ต้องการใช้จะเท่ากับ ขนาดของข้อมูลแบบ `int` บวกด้วยขนาดของข้อมูลแบบ `char` อย่างละหนึ่งตัว

เราสามารถกล่าวสรุปได้ว่า ยูเนียนและโครงสร้างต่างก็มีลักษณะที่คล้ายกัน ยกเว้นแต่ในเรื่องของขนาดของหน่วยความจำที่ต้องการใช้ และสำหรับยูเนียนแล้ว เราสามารถเก็บค่าของข้อมูลสมาชิกได้เพียงตัวใดตัวหนึ่งในช่วงเวลาหนึ่งเท่านั้น และแตกต่างจากวิธีการเก็บข้อมูลในโครงสร้าง ซึ่งถ้าเป็นโครงสร้างแล้ว เรา

สามารถเก็บข้อมูลไว้ในข้อมูลสมาชิกได้ทุกตัวในช่วงเวลาเดียวกัน เพราะต่างก็มีหน่วยความจำของตนเอง ดังนั้นในช่วงเวลาหนึ่งๆ ยูเนียนสามารถเก็บข้อมูลไว้ในสมาชิกตัวใดตัวหนึ่ง และเพียงตัวเดียวเท่านั้น หรือเรากล่าวได้ว่า ยูเนียน คือโครงสร้างที่มีข้อมูลสมาชิกที่ใช้พื้นที่หน่วยความจำร่วมกัน

จากตัวอย่างของยูเนียน alpha ในข้างต้น ถ้าเราต้องการเก็บข้อมูลแบบ char ไว้ใน a พื้นที่หน่วยความจำทั้งหมดก็จะถูกจองไว้สำหรับสมาชิก a แม้ว่าจะใช้จริงเพียงหนึ่งไบต์เท่านั้น แต่ถ้าเราพยายามเก็บข้อมูลแบบ int ไว้ในสมาชิก i ข้อมูลที่เราเคยเก็บไว้ในสมาชิก a จะถูกลบทิ้งไป เพราะจะต้องใช้เก็บข้อมูลของสมาชิก i

การเข้าถึงสมาชิกของยูเนียนแต่ละตัวก็ทำได้เหมือนกับของโครงสร้าง เช่น

```
#include <stdio.h>

union alpha {
     int i;
     char a;
} global;

int main()
{
     printf ("Size of union alpha = %d bytes\n",
               sizeof(union alpha));

     printf (" &global.i = %p\n", &global.i);
     printf (" &global.a = %p\n", &global.a);

     global.i = 100;
     printf ("i = %d, a = %c\n", global.i, global.a);

     global.a = 'a';
     printf ("i = %d, a = %c\n", global.i, global.a);

     return 0;
}
```

อันที่จริงแล้วยูเนียนก็ทำหน้าที่คล้ายพอยน์เตอร์อเนกประสงค์ที่สามารถชี้ไปยังข้อมูลหลายๆแบบได้ แต่ใช้หน่วยความจำร่วมกัน เราลองพิจารณาตัวอย่างต่อไปนี้

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

```
typedef union {
     short  s;
     long   l;
     float  f;
} u_of_data;

int main()
{
     short     *sptr;
     long      *lptr;
     float     *fptr;
     u_of_data  storage;

     /* points to the address of 'storage'. */
     u_of_data *ptr = &storage;

     sptr = (short *)ptr;
     lptr = (long  *)ptr;
     fptr = (float *)ptr;

     ptr->s = 10;
     printf ("ptr->s    = %d \n", ptr->s);
     printf ("storage.s = %d \n", storage.s);
     printf ("*sptr     = %d \n", *sptr);

     ptr->l = 123456789L;
     printf ("ptr->l    = %ld \n", ptr->l);
     printf ("storage.l = %ld \n", storage.l);
     printf ("*lptr     = %ld \n", *lptr);

     ptr->f = 1.11;
     printf ("ptr->f    = %f \n", ptr->f);
     printf ("storage.f = %f \n", storage.f);
     printf ("*fptr     = %f \n", *fptr);

     return 0;
}
```

จากตัวอย่างข้างบน นิพจน์ต่อไปนี้

```
     *(short *)ptr
     *(long  *)ptr
     *(float *)ptr
```

จะให้ผลเหมือนกับการใช้นิพจน์

```
     storage.s
     storage.l
     storage.f
```

ตามลำดับ เพราะเราได้กำหนดให้พอยน์เตอร์ ptr ชี้ไปยังแหล่งข้อมูลของตัวแปร storage แบบยูเนียน

8.2.2 การติดตั้งค่าเริ่มต้นให้ตัวแปรแบบยูเนียน

การติดตั้ง ค่าเริ่มต้นให้แก่ยูเนียนนั้น จะเป็นการติดตั้งค่าเริ่มต้นของสมาชิกตัวแรกในยูเนียน ตัวอย่างเช่น

```
union alpha  {
     int  i;
     char a;
};

alpha A = {65};
```

ดังนั้น จึงเป็นการติดตั้งค่าให้แก่สมาชิก i ของยูเนียนและมีค่าเท่ากับ 65 ถ้าเรานิยามยูเนียนใหม่เป็น

```
union alpha  {
     char a;
     int  i;
};

alpha A = {'A'};
```

โดยที่นิยามสมาชิก a ก่อน i ดังนั้น เวลาเราติดตั้งค่าเริ่มต้นให้แก่ยูเนียน จึงเป็นค่าเริ่มต้นสำหรับ a

ถ้าสมาชิกตัวแรกของยูเนียนเป็นโครงสร้างหรืออาร์เรย์ เวลาเราติดตั้งค่าเริ่มต้นให้แก่ยูเนียน เราจะกำหนดค่าเริ่มต้นให้แก่สมาชิกแต่ละตัวของโครงสร้างหรือข้อมูลแต่ละตัวของอาร์เรย์นี้ ตัวอย่างเช่น

```
union alpha {
     struct {
          char h;
          char l;
     }    ch;
     int  i;
};

union alpha A = {{'\0', 'A'}};
```

ซึ่งหมายความว่า เราได้กำหนดให้ นิพจน์ A.ch.h และ A.ch.l มีค่าเป็น '\0' และ 'A' ตามลำดับ หรือ ถ้าเป็นอาร์เรย์ เช่น

```
union alpha  {
```

```
        char ch[2];
        int  i;
};

union alpha A = {'\0', 'A'};
```

ก็จะหมายความว่า นิพจน์ A.ch[0] และ A.ch[1] มีค่าเป็น '\0' และ 'A' ตามลำดับ

ในการส่งข้อมูลจากคอมพิวเตอร์หนึ่งไปยังเครื่องคอมพิวเตอร์อีกเครื่องหนึ่ง เช่น ผ่านโมเด็มไปตามสายโทรศัพท์ เป็นต้น จะต้องส่งหรือรับข้อมูลทีละไบต์ตามลำดับไป เช่น ถ้าเราต้องการจะส่งข้อมูลแบบ long int ซึ่งมีความยาวเท่ากับ 4 ไบต์ ก็จะต้องแยกส่งทีละไบต์ และเมื่อข้อมูลแต่ละไบต์ไปถึงผู้รับแล้วก็ต้องทำการประกอบและเรียบเรียงไบต์ที่ได้รับเข้าด้วยกันใหม่ เพื่อให้ได้ข้อมูลตามที่เป็นจริง แต่ในทางปฏิบัติแล้วจะมีหลายขั้นตอนกว่าจะส่งและรับข้อมูลแต่ละไบต์ได้ แต่เพื่อที่จะแสดงให้เห็นวิธีการใช้ประโยชน์ของแบบข้อมูลยูเนียน เราจะพิจารณาโมเดลการทำงานของโปรแกรมสำหรับส่งและรับข้อมูลที่เป็นโครงสร้างแบบ PACKET อย่างง่ายๆ

```
typedef unsigned char BYTE;

struct st_data {
        int   packetID;
        char  message[255];
};

typedef union {
        struct st_data  data;
        BYTE array[sizeof(struct st_data)];
} PACKET;
```

ถ้าต้องการจะส่งข้อมูลที่เป็นโครงสร้างแบบ PACKET ก็จะเรียกใช้ฟังก์ชัน set_data() โดยที่ฟังก์ชันนี้จะกำหนดค่าของตัวแปร pk_src ซึ่งคล้ายกับว่าเราต้องการจะส่งข้อมูลที่เป็นโครงสร้างนี้ออกไป และฟังก์ชันในลักษณะดังกล่าวจะทำหน้าที่จัดการส่งข้อมูลทีละไบต์ออก และเมื่อต้องการจะอ่านข้อมูลสำหรับฝ่ายผู้รับที่เป็นจุดหมายปลายทาง ก็จะเรียกใช้ฟังก์ชัน get_data() โดยอ่านข้อมูลเข้าทีละไบต์และเก็บไว้ในตัวแปร pk_dest และเมื่อได้อ่านข้อมูลครบแล้วก็ให้แสดงข้อมูลสมาชิกของโครงสร้างที่ได้รับ

```
PACKET pk_src, pk_dest;

void set_data(struct st_data data)
{
        pk_src.data = data;
}
```

```
BYTE read_byte()
{
     static int i = 0;
     BYTE byte = pk_src.array[i];

     i = (i+1) % sizeof(struct st_data);
     return byte;
}


void get_data()
{
     int i;

     for(i=0; i < sizeof(struct st_data); i++)
          pk_dest.array[i] = read_byte();
}

void print_data()
{
     printf("packetID = %d\n", pk_dest.data.packetID);
     printf("message  = %s\n", pk_dest.data.message);
}
```

เมื่อได้กำหนดแล้วว่าจะส่งข้อมูลใด โดยเรียกใช้ฟังก์ชัน `set_data()` ต้องไปก็จะเกี่ยวข้องกับการรับข้อมูลและสมมุติข้อมูลแต่ละตัวไปถึงจุดหมายปลายทางแล้ว ก็จะอ่านข้อมูลขาเข้าทีละไบต์โดยใช้ฟังก์ชัน `read_byte()` จะเห็นได้ว่า เวลาเราอ่านข้อมูลทีละไบต์ เราจะใช้ข้อมูลสมาชิก `array` ของยูเนียนและเมื่อได้ข้อมูลครบแล้ว เราก็จะใช้สมาชิก `data` ของยูเนียนที่เป็นโครงสร้างแบบ `struct st_data`

```
extern void set_data(struct st_data);
extern void get_data()
extern void print_data();

int main()
{
     int i;
     struct st_data data_to_send =
          {3021,"Hello World!"};

     set_data(data_to_send);
     get_data();
     print_data();
     return 0;
}
```

สรุปได้ว่า จุดประสงค์ของการใช้งานโครงสร้างที่เป็นยูเนียน ก็เพื่อที่จะประหยัดการใช้หน่วยความจำสำหรับเก็บข้อมูลสมาชิก การตัดสินใจว่าจะเลือกใช้โครงสร้างหรือยูเนียน หรือใช้โครงสร้างที่ประกอบด้วยยูเนียนเมื่อใด ก็ขึ้นอยู่กับลักษณะของปัญหาและรูปแบบของข้อมูลที่ต้องการจะเก็บไว้ รวมทั้งประสบการณ์ของนักเขียนโปรแกรมด้วย

8.3 เขตข้อมูลบิต (Bit Field)

แบบข้อมูลพื้นฐานที่เล็กที่สุดในภาษาซี คือ `char` เพราะมีขนาดแค่ 8 บิตหรือ 1 ไบต์เท่านั้น แต่สำหรับแบบข้อมูลรวมแล้วยังมีโครงสร้างของข้อมูลอีกชนิดหนึ่งที่สามารถมี ข้อมูลสมาชิกที่มีขนาดเล็กกว่าหนึ่งไบต์ได้ เราเรียกแบบข้อมูลชนิดนี้ว่า เขตข้อมูลบิตหรือบิตฟิลด์ (Bit Field) ซึ่งตามปรกติแล้วจะใช้ภายในโครงสร้าง ประกอบด้วยแถวบิตที่มีความยาวแตกต่างกันไป ตั้งแต่หนึ่งแถวขึ้นไป แถวบิตที่สั้นที่สุดจะมีขนาดเพียงหนึ่งบิตเท่านั้น และเรายังสามารถกำหนดตัวระบุชื่อให้แก่แถวบิตแต่ละแถวได้ด้วย

เราจะใช้เขตข้อมูลบิตในโครงสร้างที่ประกอบด้วยแบบข้อมูลที่ใช้แทนเลขจำนวนเต็มแบบ `int` ตั้งแต่หนึ่งตัวขึ้นไป เพียงแต่ข้อมูลแบบ `int` แต่ละตัวนี้ เราจะเลือกใช้บิตเพียงบางส่วนเท่านั้น (หรือทั้งหมด) ขึ้นอยู่ว่าเราได้กำหนดใช้จำนวนบิตกี่ตัว

8.3.1 การแจ้งใช้โครงสร้างที่ประกอบด้วยเขตข้อมูลบิต

การแจ้งใช้ ตัวแปรที่มีลักษณะเป็นเขตข้อมูลบิต มีรูปแบบทั่วไปดังนี้

```
struct name {
     type name1 : length1;
     type name2 : length2;
     ....
     type nameN : lengthN;
} variable_list;
```

`type` เป็นแบบข้อมูลพื้นฐานที่เราใช้ สำหรับแถวบิตแต่ละแถว และจะต้องเป็น `int` , `unsigned` หรือ `signed` เท่านั้น (แต่บางคอมไพล์เลอร์ ก็อนุญาตให้เราใช้ `char` , `enum` หรือ `short` ได้) ภายในโครงสร้างจะมีแถวบิตอย่างน้อยหนึ่งชุด แถวบิตแต่ละแถวจะมีชื่อกำกับ และมีขนาดแตกต่างกันไปตามแต่เราจะ

กำหนดแต่ความยาวของแถวบิตแต่ละแถวจะต้องเป็นเลขจำนวนเต็มบวกที่มีค่าไม่เกินขนาดของข้อมูลแบบ `int` เช่น ถ้า `sizeof(int)` มีค่าเท่ากับ 16 ความยาวของแถวบิตแต่ละ ชุด ในบิตฟิลด์ต้องไม่เกิน 16

8.3.2 การเก็บแถวบิตในหน่วยความจำ

ถ้าเราใช้แถวบิตหลายๆแถวในโครงสร้างของเขตข้อมูลบิต คอมไพล์เลอร์จะพยายามจัดเก็บแถวบิตทั้งหมดไว้ในหน่วยความจำขนาดหนึ่งเวิร์ด ถ้าไม่เพียงพอก็ใช้หน่วยความจำเพิ่มขึ้นอีกหนึ่งเวิร์ดไปเรื่อยๆ จนกว่าจะครบแถวบิตทั้งหมด

```
struct bit_fields_1{
     unsigned int x : 10;
     unsigned int y : 4;
     unsigned int z : 1;
} bf1;

struct bit_fields_2 {
     int a : 12;
     int b : 12;
} bf2;
```

การใช้เขตข้อมูลบิตนี้ ส่วนใหญ่จะพบเห็นได้บ่อย เมื่อต้องเกี่ยวข้องกับการเขียนโปรแกรมในการควบคุมอุปกรณ์ด้านฮาร์ดแวร์ (Hardware Device) เช่น การเขียนโปรแกรมควบคุมโมเด็ม (Modem) หรืออุปกรณ์ที่ต่อเข้ากับคอมพิวเตอร์อื่นๆ เป็นต้น

มีข้อจำกัดเกี่ยวกับการใช้เขตข้อมูลบิต คือ เราไม่สามารถใช้โอเปอร์เรเตอร์ `sizeof` ในการหาขนาดหน่วยความจำที่ใช้ และโอเปอร์เรเตอร์ `&` สำหรับหาที่อยู่ของหน่วยความจำของบิตฟิลด์ได้ ตัวอย่างที่ผิด เช่น

```
sizeof (bf1.x)
&bf2.a
```

8.3.3 ตัวอย่างการนิยามโครงสร้างที่มีข้อมูลสมาชิกเป็นบิตฟิลด์

สมมุติว่า เราต้องการนิยามโครงสร้างที่ประกอบด้วยบิตฟิลด์ สำหรับเก็บข้อมูลเกี่ยวกับเวลา และวันเดือนปี เช่น กำหนดให้ `struct st_time` และ `struct st_date` เป็นโครงสร้างที่มีคุณสมบัติต่อไปนี้

จำนวนบิต (หมายเลข)	ข้อมูลที่เก็บ
5 (0-4)	วินาที (0-59)
6 (5-10)	นาที (0-59)
5 (11-15)	ชั่วโมง (0-24)

จำนวนบิต (หมายเลข)	ข้อมูลที่เก็บ
6 (0-5)	วัน (1-31)
3 (6-8)	เดือน (1-12)
7 (9-15)	ปี (0-119) โดย เริ่มนับตั้งแต่ปี ค.ศ. 1980

และเขียนโครงสร้างทั้งสองในภาษาซีได้ดังนี้

```
struct st_time
{
     int sec    : 5;  /* second */
     int min    : 6;  /* minute */
     int hr     : 5;  /* hour   */
};

struct st_date
{
     int day    : 6;
     int month  : 3;
     int year   : 7;
};
```

ต่อไปเราก็จะเขียนฟังก์ชันที่อ่านค่าพารามิเตอร์สำหรับเวลา แยกเป็น ชั่วโมง นาที วินาที และเก็บข้อมูลนี้ไว้ในข้อมูลสมาชิกแต่ละตัวของโครงสร้าง `struct st_time`

```
struct st_time mkTime(int hr, int min, int sec )
{
     struct st_time time;

     assert((0 <= sec && sec < 60) &&
            (0 <= min && min < 60) &&
            (0 <= hr && hr <= 24));
     time.sec = sec;
     time.min = min;
     time.hr  = hr;
     return time;
}
```

และในทำนองเดียวกัน เราก็เขียนฟังก์ชันสำหรับเก็บข้อมูลเกี่ยวกับวันเดือนปี ไว้ในข้อมูลสมาชิกของโครงสร้าง `struct st_date`

```
struct st_date mkDate(int day, int month, int year)
{
     struct st_date date;

     assert((1 <= day && day <= 31) &&
            (1 <= month && month <= 12) &&
            (1980 <= year && year <= 2099));
     date.day   = day;
     date.month = month;
     date.year  = year - 1980;
     return date;
}
```

และท้ายสุด ก็เป็นการทดลองเรียกใช้ฟังก์ชันทั้งสองที่เราได้สร้างขึ้น

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

static char *Month[]= {
     "Jan", "Feb", "Mar", "Apr",
     "May", "Jun", "Jul", "Aug",
     "Sep", "Oct", "Nov", "Dec"
};

int main()
{
     struct st_time t;
     struct st_date d;

     t = mkTime(12, 30, 0);
     d = mkDate(29, 10, 1984);

     printf("%02d:%02d:%02d\n", t.hr, t.min, t.sec);
     printf("%02d %s %d\n",
               d.day, Month[d.month], d.year + 1980);
     return 0;
}
```

จากตัวอย่างข้างบน โปรดสังเกตว่า ในกรณีที่เราผ่านค่าแบบ `int` ให้เป็นค่าของข้อมูลสมาชิกที่เป็นบิตฟิลด์ ซึ่งมีขนาดน้อยกว่า ขนาดของข้อมูลแบบ `int` เช่น

```
     date.day = day;
```

day เป็นบิตฟิลด์ ที่มีความยาวเพียง 6 บิตเท่านั้น ดังนั้น ค่าที่เราจะเก็บไว้ในข้อมูลสมาชิกนี้ จึงเป็นบิตหกตัวแรก ของข้อมูลแบบ int ดังกล่าว

8.4 การใช้ enumeration types

นอกเหนือจากการใช้แบบข้อมูลพื้นฐานต่างๆในภาษาซีแล้ว เรายังสามารถกำหนดแบบข้อมูลขึ้นมาใหม่ที่มีค่าตามที่เราต้องการเท่านั้น และสำหรับค่าต่างๆนี้ เราจะไม่เขียนให้อยู่ในรูปของตัวเลขแต่จะเขียนให้อยู่ในเชิงสัญลักษณ์ ซึ่งหมายถึง การใช้ตัวระบุชื่อนั้นเอง แบบข้อมูลในลักษณะนี้มักจะใช้ สำหรับค่าต่างๆที่แทนที่ด้วยกลุ่มของสัญลักษณ์ หรือ เซ็ตนั้นเอง ยกตัวอย่างเช่น เราต้องการสร้างเซ็ตของข้อมูลสำหรับใช้ในทางตรรกศาสตร์ที่ประกอบด้วยสมาชิก True และ False ซึ่งหมายถึง จริง และ เท็จ ตามลำดับ และกำหนดให้สัญลักษณ์ทั้งสองที่เราได้นิยามขึ้นมาใหม่นี้ เป็นค่าที่เป็นไปได้ของแบบข้อมูลในภาษาซี เช่น มีชื่อว่า boolean เราก็สามารถทำได้โดยอาศัยโครงสร้างแบบข้อมูลแบบ enum ซึ่งย่อมาจากคำว่า enumeration และนิยามได้ดังนี้

```
enum boolean {True, False};
```

ถ้าเราต้องการจะแจ้งใช้ตัวแปรใดๆที่มีแบบข้อมูลดังข้างต้นนี้ เราก็เขียนได้ดังนี้

```
enum boolean  ok, done;
```

โปรดสังเกตว่า เมื่อเวลาแจ้งใช้ ตัวแปรที่มีแบบข้อมูลที่นิยามโดย enum เราก็ต้องเขียนคำว่า enum เริ่มต้นและตามด้วยชื่อของแบบข้อมูล จากนั้นจึงเป็นชื่อของสัญลักษณ์ต่างๆ

ถ้าเราต้องประหยัดคำว่า enum เวลาแจ้งใช้ ตัวแปร เราก็สามารถทำได้ โดยใช้คำสั่ง typedef เข้าช่วย คือเป็นการนิยามชื่อใหม่ให้แก่แบบข้อมูล ดังรูปแบบข้างล่างนี้

```
typedef enum boolean {False, True} Boolean;
typedef enum         {False, True} Boolean;
typedef enum Boolean {False, True} Boolean;
```

แบบใดแบบหนึ่ง ในกรณีแรกเราได้ให้ ชื่อ boolean แก่ enum {False, True} ในกรณีที่สองเราไม่ได้กำหนดชื่อ เพราะเราได้ใช้คำสั่ง typedef ในการกำหนดชื่อ ดังนั้นเราก็ไม่จำเป็นต้องให้ ชื่อ แก่ enum หรือในกรณีที่สาม เรากำหนดชื่อ Boolean ให้ num {False, True} และใช้ ชื่อเดียวกันนี้สำหรับ ก็ได้ โดยไม่จำเป็นต้องหาชื่อใหม่

และเราก็จะได้แบบข้อมูลที่เรานิยามขึ้นใหม่ และ สามารถใช้ Boolean ในลักษณะของแบบข้อมูลทั่วๆไป เวลาแจ้งใช้ตัวแปรก็จะมีลักษณะดังตัวอย่างนี้

```
Boolean  ok, done;
```

และค่าของตัวแปร ok และ done ที่เป็นไปได้คือ True และ False เท่านั้น

ตัวอย่างการสร้างและนิยามแบบบข้อมูลอื่นๆที่ใช้แทนสัญลักษณ์ต่างๆในเซ็ต เช่น

```
enum day {
          Sunday, Monday, Tuesday, Wednesday,
          Thursday, Friday, Saturday
         };

enum rgb {red, green, blue};

typedef enum {
          Sunday, Monday, Tuesday, Wednesday,
          Thursday, Friday, Saturday
        } Days;

typedef enum {red, green, blue} RGBcolors;


Days yesterday, today=Monday, tomorrow;
```

เราจะเห็นได้ว่า สัญลักษณ์หรือตัวระบุชื่อต่างๆที่เป็นสมาชิกของเซ็ตที่เป็นแบบข้อมูล ซึ่งนิยามโดย enum นั้น ที่จริงแล้วคือค่าคงที่แบบ int นั้นเอง โดยมีหลักการดังนี้ สัญลักษณ์ตัวแรกจะมีค่าเท่ากับ 0 โดยอัตโนมัติ และสัญลักษณ์ตัวถัดไปในเซ็ตจะมีค่าเป็น 1 และ ต่อไปเรื่อยๆ ดังนั้น

```
enum boolean {False,True} Boolean;
```

จะมีค่าสำหรับ False เท่ากับ ศูนย์ และ True มีค่าเท่ากับหนึ่ง แต่อย่างไรก็ตาม เราสามารถกำหนดค่าของสัญลักษณ์ภายใน เซ็ตได้เอง ซึ่งคล้ายกับการติดตั้งค่าเริ่มต้นของตัวแปร เช่น

```
enum boolean {False=0,True=1} Boolean;
```

นอกจากนี้ เรายังสามารถกำหนดค่าของสัญลักษณ์เฉพาะบางตัวก็ได้ เช่น

```
typedef enum {
          Sunday=1,Monday,Tuesday, Wednesday,
          Thursday, Friday, Saturday
        } Days;
```

ในตัวอย่างค่าของ Monday จะมีค่าเท่ากับ 2 Tuesday เท่ากับ 3 ไปเรื่อยๆจนถึง Saturday ซึ่งมีค่าเท่ากับ 7 ตามลำดับ

อีกตัวอย่างที่แสดงให้เห็นวิธีการกำหนดค่าของสัญลักษณ์เฉพาะบาง ตัว คือ

```
enum fruit {apple=3, lemon, orange=1, pear};
```

ในกรณีนี้ apple มีค่าเท่ากับ 3 lemon มีค่าเท่ากับ 4 orange มีค่าเท่ากับ 1 และ pear มีค่าเท่ากับ 2 ดังนั้นถ้าเราเขียนว่า

```
enum fruit {orange=1, pear, apple, lemon};
```

ก็ย่อมให้ผลเหมือนกัน

หรือ ในบางครั้ง เราอาจจะกำหนดค่าของสัญลักษณ์ให้มีค่าใดๆที่แตกต่างกันไปก็ได้ เช่น

```
enum mathematician {
     Cauchy=1789,
     Euler=1707,
     Fourier=1768,
     Gauss=1777,
     Hesse=1811,
     Hilbert=1862,
     Kronecker=1823,
     Laplace=1749
};
```

ตามปรกติแล้ว เรามักจะไม่เกี่ยวข้องกับคำถามที่ว่า สัญลักษณ์ต่างๆของ enum ตัวใดมีค่าเท่าใด เพราะจุดประสงค์ของการใช้แบบข้อมูลที่สร้างโดย enum คือ เราต้องการใช้ค่าเชิงสัญลักษณ์มากกว่าค่าเชิงตัวเลข ยกเว้นเสียแต่ว่าเราต้องการจะเปลี่ยนสัญลักษณ์เหล่านี้ให้เป็นค่าตัวเลขแบบ int

แบบฝึกหัดท้ายบท

1. จงอธิบายความแตกต่างและความเหมือนของนิพจน์ต่อไปนี้ และนิพจน์ใดที่ไม่ถูกต้องตามหลักไวยกรณ์ ถ้ากำหนดให้ `struct_name` เป็นแบบข้อมูลที่เป็นโครงสร้าง และ `member_name` เป็นชื่อของสมาชิกตัวหนึ่งของโครงสร้างนี้ ที่ไม่ใช่พอยน์เตอร์

```
struct_name.member_name
&struct_name.member_name
(&struct_name)->member_name
*(&struct_name)->member_name
```

2. จงใช้ฟังก์ชันที่ได้สร้างขึ้นสำหรับคำนวณเลขจำนวนเชิงซ้อน ในการคำนวณหาค่าของตัวแปร Z เมื่อ ω มีค่าเท่ากับ 50000 และ 629995 ตามลำดับ

$$Z = \left| R + \frac{1}{j\omega C} + j\omega L \right|$$

$$R = 5 \cdot 10^{3}$$

$$C = 1 \cdot 10^{-7}$$

$$L = 4 \cdot 10^{-3}$$

3. ถ้าเรานิยามแบบข้อมูลใหม่ชื่อ `Status` โดยใช้คำสั่ง `enum` ซึ่งมี ค่าเป็น `Yes` หรือ `No` ถ้าเราได้นิยามแมโครขึ้นใหม่ตามรูปแบบข้างล่างนี้

```
typedef enum {Yes=0,No=1} Status;

#define Yes   1
```

ผลที่เกิดขึ้นจะเป็นอย่างไร

4. จงอธิบายว่า ทำไมประโยคคำสั่งที่ใช้กำหนดค่าของตัวแปร `color` จึงไม่ถูกต้อง

```
{
    enum {red, green, blue}      color;
    enum {bright, medium, dark}  intensity;

    color = bright;
}
```