# 7

# อาร์เรย์และสายอักขระ

## 7.1 อาร์เรย์

ก่อนที่เราจะเริ่มเรียนรู้วิธีการใช้แบบข้อมูลที่เรียกว่า อาร์เรย์ (Array) เราลองมาพิจารณาตัวอย่างต่อไปนี้ก่อนโดยกำหนดให้ตัวแปร `a0`, `a1`, `a2`, `a3`, `a4` และ `a5` เป็นตัวแปรสถิตแบบ `int` และมีค่าเริ่มต้นเป็นศูนย์เพราะถ้าเราได้แจ้งใช้ตัวแปรใดๆแบบ `static` แต่มิได้กำหนดค่าเริ่มต้นแล้ว ค่าของตัวแปรสถิตนี้จะถูกกำหนดให้มีค่าเริ่มต้นเป็นศูนย์โดยอัตโนมัติ

```
static int a0, a1, a2, a3, a4, a5;
```

เหตุที่เลือกตัวแปรสถิตและไม่ได้ใช้ตัวแปรอัตโนมัติธรรมดาก็คือว่าเราต้องการให้ตัวแปรเหล่านี้มีหมายเลขของหน่วยความจำมากขึ้นตามลำดับที่เราได้แจ้งใช้ ตัวอย่างเช่นหมายเลขที่อยู่ของ a0 จะต้องน้อยกว่าของ a1 และที่อยู่ของ a1 จะต้องน้อยกว่า a2 ไปเรื่อยๆจนถึง a5 แต่ถ้าเราใช้ตัวแปรอัตโนมัติธรรมดา ก็จะได้ลำดับหมายเลขที่อยู่ของตัวแปรที่เรียงกลับกันจากกรณีแรก

และเราแจ้งใช้ตัวแปร a ที่ทำหน้าที่เป็นพอยน์เตอร์แบบ int และกำหนดให้ชี้ไปยังที่อยู่ของตัวแปร a0

```
int * const a = &a0;
```

และพอยน์เตอร์ตัวนี้จะต้องชี้ไปยังที่อยู่ของตัวแปร a0 เท่านั้น

ต่อไปเราจะนิยามแมโครขึ้นมาใช้ ที่มีรูปแบบดังนี้คือ

```
#define data(array, i)  *(array + (i))
```

สำหรับแมโครนี้อาร์กิวเมนท์ตัวแรกจะต้องเป็นพอยน์เตอร์และตัวที่สองจะต้องเป็นค่าคงที่หรือตัวแปรที่ให้ค่าแบบ int

นอกจากนี้เราได้สร้างฟังก์ชันชื่อ print_out ในการพิมพ์ค่าของข้อมูลจำนวน n ตัว นับตั้งแต่ที่อยู่เริ่มต้นที่อ้างถึงโดยพอยน์เตอร์ p แบบ int

```
void print_out (int *p, int n);
```

และเราลองมาดูรายละเอียดของโปรแกรมข้างล่างนี้

---

```
#include <stdio.h>

#define data(array, i)  *(array + (i))

void print_out (int *p, int n)
{
     int i;
     for(i=0; i<n ; i++) {
          printf("%4d ", data(p,i));
     }
     printf("\n");
}
```

```
int main()
{
     static int a0,a1,a2,a3,a4,a5;
     int * const a = &a0;
     char *format = "%p %p %p %p %p %p\n";

     printf(format, &a0, &a1, &a2, &a3, &a4, &a5);
     print_out(a, 6);

     data(a,0) =  10;
     data(a,1) =  -1;
     data(a,2) = 340;
     data(a,5) =   2;
     print_out(a, 6);

     data(a,3) += data(a,2);
     data(a,4)  = data(a,0) - data(a,5);
     print_out(a, 6);

     return 0;
}
```

ผลของโปรแกรมก็จะเป็นดังนี้

```
     0952 0954 0956 0958 095A 095C
        0    0    0    0    0    0
       10   -1  340    0    0    2
       10   -1  340  340    8    2
```

ในบรรทัดแรกเป็นหมายเลขที่อยู่ของตัวแปร a0 ... a5 ตามลำดับ เราเห็นได้ว่าค่าของตัวเลขจะเรียงจากน้อยไปมากและมีระยะห่างกันเท่ากับ 2 ซึ่งเท่ากับขนาดของข้อมูลแบบ int แต่ละตัวนั้นเอง (เนื่องจากได้ทดลองรันโปรแกรมกับเครื่องพีซีธรรมดาดังนั้นขนาดของ int จึงเป็นสองไบต์) เมื่อเราใช้ฟังก์ชัน print_out() เป็นครั้งแรก ก็จะเป็นการพิมพ์ค่าเริ่มต้นของตัวแปร a0 ... a5 แต่ละตัวซึ่งมีค่าเท่ากับศูนย์

การเปลี่ยนแปลงและอ่านค่าของตัวแปรตัวใดตัวหนึ่งจาก a0 ... a5 เราจะใช้แมโครในการเข้าถึงที่อยู่ของตัวแปรแต่ละตัว เช่น data(a,0) หมายถึงการเข้าถึงตัวแปร a0 หรือ data(a,5) หมายถึงการเข้าถึงตัวแปร a5 เป็นต้น เราจะเห็นได้ว่าถ้าเราต้องการเข้าถึงตัวแปรตัวใดจากตัวแปรหกตัวที่มีหน่วยความจำเรียงต่อกันนั้น เราก็กำหนดใช้แค่ตัวแปรพอยน์เตอร์ที่เก็บที่อยู่เริ่มต้นของตัวแปร a0 และกำหนดระยะห่างจากที่อยู่เริ่มต้นนี้ไปข้างหน้าโดยนับตามจำนวนข้อมูลและมิใช่นับในหน่วยไบต์ เราก็จะเข้าถึงตัวแปรที่ต้องการได้อย่างง่าย ไม่ว่าจะเป็นการอ่านค่าหรือกำหนดค่าของตัวแปรเหล่านั้นก็ตาม

ในกรณีตัวอย่างนี้ เราจะเห็นได้ว่า เราได้แจ้งใช้ตัวแปรหลายๆตัวที่มีแบบข้อมูลเหมือนกัน และมีหน่วยความจำเรียงต่อกันจากน้อยไปมากตามลำดับที่ได้แจ้งใช้ การเข้าถึงข้อมูลของตัวแปรแต่ละตัวก็อาศัยพอยน์เตอร์เข้าช่วยและการกำหนดตำแหน่งของตัวแปร ถ้าเราต้องการแจ้งใช้ตัวแปรทั้งหมด 100 ตัวหรือมากกว่านี้ ปัญหาก็คือเราก็ต้องมาเขียนชื่อตัวแปรแต่ละตัวซึ่งไม่เหมาะสม ในภาษาซีได้มีการกำหนดแบบข้อมูลอีกชนิดหนึ่งที่เปิดโอกาสให้เราแจ้งใช้ข้อมูลแบบเดียวกันหลายๆตัวพร้อมกันและจัดอยู่ในบล็อกของหน่วยความจำเดียวกันและใช้ตัวระบุชื่อร่วมกันโดยที่ภายในบล็อกนี้ข้อมูลแต่ละตัวจะเรียงต่อกันไป เราเรียกแบบข้อมูลนี้ว่า *อาร์เรย์* หรือ *แถวลำดับข้อมูล* แม้ว่าการทำงานของอาร์เรย์จะไม่เหมือนกับตัวอย่างข้างบนโดยตรงแต่ก็กล่าวได้ว่าใช้หลักการเดียวกัน

อาร์เรย์หรือแถวลำดับจัดเป็นแบบข้อมูลรวม (Aggregate Type) ชนิดหนึ่งที่ใช้ในภาษาซี และลักษณะสำคัญของอาร์เรย์คือการรวบรวมข้อมูลแบบเดียวกันมากกว่าหนึ่งตัวขึ้นไปให้อยู่ในชุดหรือแถวเดียวกัน โดยข้อมูลแต่ละตัวในอาร์เรย์ซึ่งมักจะเรียกว่า element หรือส่วนประกอบพื้นฐานจะมีหมายเลขของตนเองที่ไม่ซ้ำกันโดยอัตโนมัติ เริ่มตั้งแต่ 0 และนับต่อไปเรื่อยๆ ดังนั้นอาร์เรย์ก็ไม่แตกต่างอะไรจากแถวของหน่วยความจำที่สามารถเก็บข้อมูลแบบเดียวกันหลายๆตัวไว้ในตำแหน่งต่างๆของหน่วยความจำบริเวณเดียวกัน และเราก็สามารถเข้าถึงข้อมูลพื้นฐานแต่ละตัวได้โดยอาศัยเลขดัชนี (Index) เป็นตัวชี้บอกตำแหน่งของข้อมูลที่ต้องการ เราลองนึกถึงเวกเตอร์ (Vector) ในวิชาคณิตศาสตร์ เช่น

$$a = \begin{bmatrix} 3 \\ 2 \end{bmatrix} \qquad b = \begin{bmatrix} 2.0 \\ -1.0 \\ 3.5 \end{bmatrix}$$

a และ b ต่างก็เป็นเวกเตอร์ซึ่งมีจำนวนของแถวแตกต่างกันไป ถ้าเราต้องการกำหนด ให้ a หรือ b เป็นตัวแปรสองตัวใดๆที่สามารถใช้ในภาษาซีได้ เราก็ต้องกำหนดให้ a และ b เป็นตัวแปรที่มีแบบข้อมูลรวมเป็นอาร์เรย์และอาจจะกำหนดให้อาร์เรย์ a เก็บข้อมูลแบบ `int` และ b เก็บข้อมูลแบบ `double` และสามารถเขียนได้ดังนี้

```
int     a[2];
double  b[3];
```

ซึ่งเป็นรูปแบบการแจ้งใช้ตัวแปรที่เก็บแถวข้อมูลแบบอาร์เรย์อย่างง่าย มิติเดียว ในกรณีตัวอย่างนี้ a เป็นตัวแปรที่มีแบบข้อมูลรวมเป็นอาร์เรย์และมีขนาดหรือความจุ เท่ากับ 2 ในขณะที่ b เป็นตัวแปรที่มีแบบข้อมูลรวมเป็นอาร์เรย์ขนาดเท่ากับ 3 สรุปได้ว่าสำหรับการแจ้งใช้อาร์เรย์เราก็ต้องกำหนดขนาดของอาร์เรย์ไว้ตั้งแต่ต้นเพื่อที่จะให้คอมพิวเตอร์จัดสรรจำนวนของหน่วยความจำแต่ละไบต์ให้แก่อาร์เรย์ได้อย่างถูกต้อง

ตัวอย่างการแจ้งใช้อาร์เรย์ที่ผิด เช่น

```
int    i_array [0];
char  ch_array [];
short sh_array [32768];
```

ขนาดของอาร์เรย์จะต้องมากกว่าศูนย์ และมีขนาดไม่ใหญ่เกินไป (ขึ้นอยู่กับคอมไพล์เลอร์และคอมพิวเตอร์ที่ใช้)

ตัวอย่างอีกตัวอย่างหนึ่งในชีวิตประจำที่เกี่ยวข้องกับอาร์เรย์คือตารางข้อมูล ยกตัวอย่างเช่น ตารางแสดงคะแนนสอบของนักเรียน

| เลขที่สอบของนักเรียน | เลขดัชนีสำหรับอาร์เรย์ | คะแนนที่ได้ |
|---|---|---|
| 001 | 0 | 70.5 |
| 002 | 1 | 89.0 |
| 003 | 2 | 87.5 |
| .... | .... | .... |
| 050 | 49 | 64.5 |

จากตารางข้างบน เรามีจำนวนข้อมูลทั้งหมด 50 ข้อมูล เรียงตามลำดับโดยมีหมายเลขกำกับระหว่าง 001 ถึง 050 ถ้าจะแจ้งใช้ตัวแปรแบบอาร์เรย์ เช่น ให้ชื่อว่า `score` ที่มีขนาดเท่ากับ 50 หน่วยโดยใช้ `float` เป็นแบบข้อมูลสำหรับคะแนนของนักเรียนก็สามารถทำได้ดังนี้

```
float score[50];
```

ในบางครั้งเราก็อาจจะแจ้งใช้อาร์เรย์ที่มีขนาดมากกว่าจำนวนของข้อมูลที่เรามีอยู่จริง เช่น จากตัวอย่างของตารางคะแนนที่มีข้อมูลทั้งหมด 50 หน่วยเราก็อาจจะแจ้งใช้

```
float score[60];
```

ในกรณีนี้เราก็มีที่ว่างสำหรับข้อมูลอีก 10 หน่วยภายในอาร์เรย์ในขณะที่เรามีข้อมูลอยู่เพียง 50 หน่วยเท่านั้น แต่ที่สำคัญขนาดของอาร์เรย์จะต้องมากกว่าจำนวนของข้อมูลทั้งหมดที่เราต้องการจะเก็บไว้ในอาร์เรย์

### 7.1.1 การหาขนาดของอาร์เรย์โดยใช้ sizeof

เมื่อเราได้แจ้งใช้ตัวแปรใดๆที่เป็นอาร์เรย์แล้ว เราก็สามารถทราบขนาดของอาร์เรย์ดังกล่าวได้ทุกเวลาโดยการใช้โอเปอร์เรเตอร์ sizeof ซึ่งขนาดที่ได้นี้เป็นค่าในหน่วยของไบต์ มิได้หมายถึงความจุทั้งหมดของอาร์เรย์ที่เป็นไปได้โดยนับตามจำนวนของข้อมูล แต่ถ้าเราต้องการทราบความจุของอาร์เรย์ก็มีวิธีการง่ายๆดังตัวอย่างต่อไปนี้

```
#include <stdio.h>

int main()
{
     char   a[10];
     int    b[10];
     double c[10];

     printf("size of array a : %d (bytes)\n", sizeof(a));
     printf("number of elements : %d\n\n",
               sizeof(a)/sizeof(char));

     printf("size of array b : %d (bytes)\n", sizeof(b));
     printf("number of elements : %d\n\n",
               sizeof(b)/sizeof(int));

     printf("size of array c : %d (bytes)\n", sizeof(c));
     printf("number of elements : %d\n\n",
               sizeof(c)/sizeof(double));
     return 0;
 }
```

ผลของโปรแกรมที่แสดงออกทางจอภาพคือ

```
     size of array a : 10 (bytes)
     number of elements : 10

     size of array b : 20 (bytes)
     number of elements : 10

     size of array c : 80 (bytes)
     number of elements : 10
```

โดยมีหลักการคำนวณดังนี้ จำนวนข้อมูลแบบ *type* ทั้งหมดที่อาร์เรย์ *a* สามารถเก็บได้คือ

```
sizeof(a)/sizeof(type)
```

หรือเราอาจจะเขียนเป็นแมโครก็ได้ เช่น

```
ARRAY_SIZE(a,type)      (sizeof(a)/sizeof(type))
```

หรือถ้าเขียนให้ง่ายและสั้นลงโดยไม่จำเป็นต้องทราบแบบข้อมูล ก็จะเป็นดังนี้

```
ARRAY_SIZE(a)      (sizeof(a)/sizeof(a[0]))
```

เพราะอาร์เรย์จะต้องมีข้อมูลอย่างน้อยหนึ่งตัวซึ่งจะเป็นข้อมูลตัวแรกของอาร์เรย์ มีดัชนีเท่ากับศูนย์ วิธีการหาขนาดความจุของอาร์เรย์แบบนี้ เราสามารถนำไปใช้งานได้ในโปรแกรมทั่วๆไป

## 7.1.2 รูปแบบการแจ้งใช้ตัวแปรที่เป็นอาร์เรย์มิติเดียว

ขนาดของอาร์เรย์จะต้องกำหนดโดยใช้นิพจน์ค่าคงที่และเป็นเลขจำนวนเต็ม (Integral Constant Expression) เท่านั้นและค่าของนิพจน์นี้จะต้องคำนวณได้เมื่อทำการคอมไพล์ (Compile-time Computable Expression) หรือกล่าวง่ายๆได้ว่านิพจน์ที่เราใช้กำหนดขนาดของอาร์เรย์จะต้องเป็นตัวเลขหรือมีโอเปอร์เรเตอร์ที่ใช้กับเลขจำนวนเต็มเท่านั้น นิพจน์เหล่านี้ไม่สามารถมีตัวแปรใดๆได้ (ยกเว้นตัวแปรที่ใช้ร่วมกับโอเปอร์เรเตอร์ `sizeof` ) ตัวอย่างที่ถูกต้อง เช่น

```
#define SIZE  255

char ch;
char array1 [10*10];
int  array2 [SIZE+1];
char array3 [SIZE*sizeof(ch)+1];
```

เนื่องจาก `ch` เป็นตัวแปรแบบ `char` แต่เราใช้โอเปอร์เรเตอร์ `sizeof` กับตัวแปรนี้ดังนั้นค่าที่ได้จึงเป็นค่าคงที่และสามารถคำนวณได้ในเวลาทำการคอมไพล์โปรแกรมโค้ด

ตัวอย่างที่ผิด เช่น

```
const int Size=100;
int array[Size + 1];              /* ILLEGAL */
```

แม้ว่าเราจะกำหนดให้ `Size` เป็นตัวแปรแบบคงที่ก็ตามแต่เราไม่สามารถใช้ในการกำหนดขนาดของอาร์เรย์ได้

```
int * func (const int size)
```

```
{
     int i;
     static int array[size];

     for(i=0; i < size; i++)
          array[i] = 0;
     return array;
}
```

ตัวอย่างการสร้างฟังก์ชันที่มีการแจ้งใช้อาร์เรย์ภายในฟังก์ชันลักษณะนี้ถือว่าไม่ถูกต้อง เราไม่สามารถใช้พารามิเตอร์หรือตัวแปรใดๆในการกำหนดขนาดของอาร์เรย์เมื่อยามแจ้งใช้ได้เพราะตัวแปรในลักษณะนี้ไม่จัดว่าเป็นนิพจน์ที่คำนวณค่าได้เมื่อเวลาทำการคอมไพล์

## 7.1.3 การเข้าถึงข้อมูลภายในอาร์เรย์

ถ้าอาร์เรย์ a มีขนาดเท่ากับ size ซึ่งเป็นจำนวนเต็มที่มากกว่า 0 แล้ว ข้อมูลแต่ละตัวของอาร์เรย์จะเป็นดังนี้

```
a[0], a[1], a[2],...., a[size-1]
```

เครื่องหมาย [] จัดเป็นโอเปอร์เรเตอร์อย่างหนึ่ง ซึ่งเราจะวางไว้หลังชื่อของตัวแปรที่เป็นอาร์เรย์ (หรือพอยน์เตอร์ก็ได้) ซึ่งใช้ในการเข้าถึงข้อมูลแต่ละตัวของอาร์เรย์ การใช้โอเปอร์เรเตอร์ [] ร่วมกับพอยน์เตอร์ก็จะใช้ในการเข้าถึงข้อมูลเช่นกัน ตัวอย่างเช่น กำหนดให้ p เป็นพอยน์เตอร์ที่ชี้ไปยังที่อยู่ของตัวแปร a0 และเราได้แจ้งใช้ตัวแปรตัวอื่นๆด้วย

```
static int a0,a1,a2,a3,a4,a5;
int * const p = &a0;
```

ดังนั้น p[0] จึงหมายถึง *(p+0) หรือเขียนให้อยู่ในรูปแบบทั่วไปได้คือ p[i] หมายถึง *(p+i) โดยที่ i เป็นเลขจำนวนเต็มและใช้เป็นดัชนีในการเข้าถึงข้อมูลในหน่วยความจำโดยนับจากที่อยู่เริ่มต้นของ a0 ในกรณีนี้นิพจน์ p[0] จึงหมายถึงการเข้าถึงตัวแปร a0 นิพจน์ p[1]หมายถึง a1 ไปเรื่อยๆจนถึง p[5]

โปรดสังเกตและจำไว้ว่าเลขดัชนีที่เราใช้จะต้องเริ่มต้นที่ศูนย์มิใช่หนึ่งซึ่งหมายถึงข้อมูลตัวแรกสุดของอาร์เรย์และข้อมูลตัวท้ายสุดจะมีเลขดัชนีเท่ากับขนาดความจุของอาร์เรย์ลบออกด้วยหนึ่ง เราสามารถกล่าวได้ว่า ถ้า a เป็นอาร์เรย์แล้ว

$$a[i],\ \ 0 \le i \le \frac{\text{sizeof (a)}}{\text{sizeof (a[0])}} - 1$$

ก็คือข้อมูลแต่ละตัวของอาร์เรย์และเราสามารถใช้ a[i] ได้เหมือนกับตัวแปรแบบทั่วๆไป นิพจน์ a[i] นี้เราสามารถเขียนให้อยู่ในอีกรูปแบบหนึ่งได้(เขียนในเชิงของการใช้งานพอยน์เตอร์)

```
*(a+i)
```

| รูปแบบของนิพจน์ที่ให้ผลเหมือนกัน<br>กำหนดให้ a เป็นอาร์เรย์มิติเดียว และ i เป็นเลขจำนวนเต็มแบบ int | |
|---|---|
| a[i] | *(a+i) |
| &a[i] | (a+i) |
| &(a[i]) | (a+i) |
| (&a[0]+i) | (a+i) |

ผู้อ่านบางท่านอาจจะมีความสงสัยในเรื่องของการจัดการหน่วยความจำสำหรับอาร์เรย์และวิธีการเก็บข้อมูลแต่ละตัวภายในอาร์เรย์ ดังนั้นเราจะกล่าวถึงโครงสร้างของอาร์เรย์ในหน่วยความจำก่อน สมมุติว่าเราได้แจ้งใช้ตัวแปร a และกำหนดให้เป็นอาร์เรย์สำหรับข้อมูลแบบ int ที่มีขนาดเท่ากับ 6 หน่วยและเราจะอาศัยโปรแกรมตัวอย่างข้างล่างนี้ในการอธิบายโครงสร้างของหน่วยความจำสำหรับตัวแปร a

```
#include <stdio.h>

int main()
{

     int a[6];

     a[0] = 12;
     a[2] = 6;

     printf("size of int = %d\n", sizeof(int));
     printf("address of a[i], i=0,1,..,5\n");

     for (i=0; i<6; i++)
     {
       printf(" &a[%d] = %p\n", i, &a[i]);
     }

     for (i=0; i<6; i++)
     {
       printf(" (a+%d) = %p\n", i, (a+i));
     }
```

```
    printf("-----------------\n");
    printf("   a  = %p\n",     a);
    printf("  &a  = %p\n",    &a);
    printf("  *a  = %d\n",    *a);
    printf("&(&a) = %p\n", &(&a));
    printf("&(*a) = %p\n", &(*a));

    return 0;
}
```

ผลจากโปรแกรมแสดงอยู่ในรูปของตารางต่อไปนี้

| i | &a[i] | &(a+i) | a[i] | *(a+i) |
|---|---|---|---|---|
| 0 | 235E | 235E | 12 | 12 |
| 1 | 2360 | 2360 | ??? | ??? |
| 2 | 2362 | 2362 | 6 | 6 |
| 3 | 2364 | 2364 | ??? | ??? |
| 4 | 2366 | 2366 | ??? | ??? |
| 5 | 2368 | 2368 | ??? | ??? |

??? หมายถึง ค่าใดๆก็ได้ในหน่วยความจำ เพราะว่าเรายังมิได้ติดตั้งหรือกำหนดค่าเริ่มต้นให้ข้อมูล

ข้อมูลทุกตัวของอาร์เรย์จะถูกเก็บไว้ในหน่วยความจำโดยเรียงต่อกันไปตามลำดับและต่อเนื่องกัน ในกรณีตัวอย่างนี้ ที่อยู่ของข้อมูลแต่ละตัวภายในอาร์เรย์จะมีขนาดเท่ากับ 2 ไบต์ซึ่งเป็นขนาดของ พอยน์เตอร์ เรากล่าวได้ว่าที่อยู่ของข้อมูลแต่ละตัวในอาร์เรย์จะมีระยะห่างกันเท่ากับขนาดของข้อมูลเหล่านั้น เช่น ถ้าเป็นอาร์เรย์ของข้อมูลแบบ `double` ระยะห่างก็จะเป็น 8 ไบต์ ถ้าเป็นอาร์เรย์ของข้อมูลแบบ `char` ระยะห่างก็จะเป็น 1 ไบต์ เป็นต้น สำหรับอาร์เรย์ `a` ระยะห่างระหว่างที่อยู่ของข้อมูลแต่ละตัวจะเป็น 2 ไบต์

ในกรณีตัวอย่างนี้ หมายเลขที่อยู่เริ่มต้นของอาร์เรย์ `a` มีค่าเท่ากับ `235E` ซึ่งจะขึ้นอยู่กับว่าคอมพิวเตอร์จะจัดสรรพื้นที่ของหน่วยความจำที่ว่างให้แก่อาร์เรย์นี้ตรงไหน

สำหรับนิพจน์ `&a[i]` เราจะเห็นได้ว่า ลำดับการทำงานของโอเปอร์เรเตอร์ `[]` จะมาก่อนโอเปอร์เรเตอร์ `&` ดังนั้น นิพจน์นี้ จึงหมายถึงการหาค่าที่อยู่ของข้อมูล `a[i]` ในอาร์เรย์ `a`

| นิพจน์ | ค่าของนิพจน์ |
|---|---|
| a | 235E |
| &a | 235E |
| *a | 12 |
| &(&a) | 235E |
| &(*a) | 235E |

จากตารางเราจะเห็นได้ว่า ค่าของ a, &a, &a[0], และ &(&a) มีค่าเท่ากันคือ 235E ซึ่งเป็นเลขที่อยู่ในฐานสิบหก ถ้าเราคิดว่า a เป็นตัวแปรพอยน์เตอร์แล้ว a ก็จะเก็บค่าที่เป็นที่อยู่ของตนเองในหน่วยความจำ ซึ่งหมายความว่า a เป็นพอยน์เตอร์ที่ชี้ไปยังที่อยู่ของตนเอง แต่ถ้า a เป็นพอยน์เตอร์ที่มีลักษณะเช่นนั้นจริง ทำไม *a จึงให้ค่าเท่ากับ 12 ซึ่งเป็นค่าของ a[0] และไม่ใช่ค่าที่เท่ากับที่อยู่ของ a นอกจากนี้เราก็ไม่สามารถเขียนประโยคคำสั่งในลักษณะต่อไปนี้ได้

```
a  = &a[0];
a++;
a += 1;
```

ถ้า a เป็นพอยน์เตอร์จริงประโยคคำสั่งเหล่านี้จะต้องถูกต้องตามหลักไวยากรณ์ แต่คอมไพล์เลอร์จะแจ้งว่าเราไม่สามารถเปลี่ยนแปลงค่าของ a ได้ เพราะนิพจน์ a นี้หมายถึง การอ่านที่อยู่ของอาร์เรย์ a โดยอัตโนมัติ ซึ่งจะได้เป็นนิพจน์แบบค่าคงที่ ดังนั้นเราจึงไม่สามารถเปลี่ยนแปลงค่าของ a ด้วยการใช้โอเปอร์เรเตอร์ = หรือ ++ เป็นต้น เราลองเปรียบเทียบกับตัวแปรพอยน์เตอร์ที่ชี้ไปยังตัวเอง เช่น

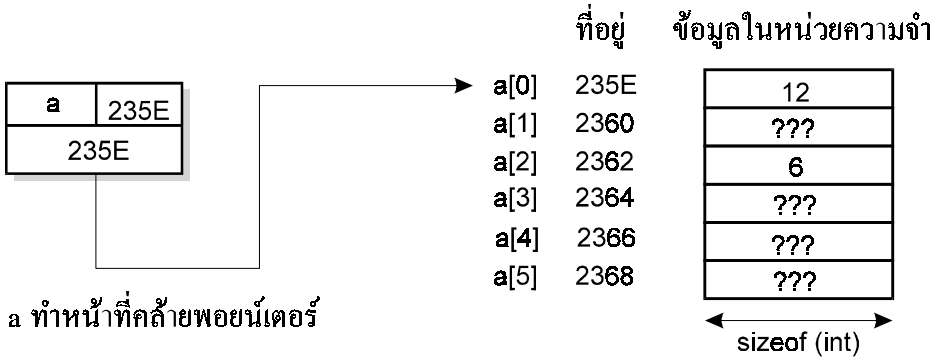
```
int i = 0;
int *ptr;

/* ptr points to itself */
ptr = (int)&ptr;
printf("   ptr = %p\n",  ptr);
printf("  *ptr = %p\n", *ptr);
printf("  &ptr = %p\n", &ptr);

/* ptr points to i */
ptr = &i;
```

ในกรณีนี้เราสามารถกำหนดให้ ptr ชี้ไปยังตัวมันเอง หรือไปยังที่อยู่ของข้อมูลแบบ int ใดๆก็ได้

เราอาจจะกล่าวได้ว่า ตัวแปรใดๆที่เป็นอาร์เรย์จะมีหน้าที่คล้ายกับการทำงานของพอยน์เตอร์ แต่จะมีลักษณะที่แตกต่างจากพอยน์เตอร์ทั่วไป กล่าวคือ ตราบใดที่เราพยายามจะหาที่อยู่ในหน่วยความจำหรือค่าของตัวแปรที่เป็นอาร์เรย์นี้เราก็จะได้ที่อยู่เริ่มต้นของอาร์เรย์ในหน่วยความจำเสมอ แต่ทันทีที่เราอ้างถึงแหล่งข้อมูลภายในอาร์เรย์ เราก็จะได้ข้อมูลตัวแรกของอาร์เรย์ เช่น *a จะให้ค่าเท่ากับ a[0] นอกจากนี้เราจะเห็นได้ว่านิพจน์ a[i], *(a+i) และ *(&a[0]+i) ก็ให้ค่าที่เท่ากัน




ความแตกต่างระหว่างตัวแปรที่เป็นอาร์เรย์และตัวแปรที่เป็นพอยน์เตอร์ คือการใช้โอเปอร์เรเตอร์ sizeof ถ้าเราใช้โอเปอร์เรเตอร์นี้กับอาร์เรย์ เราก็จะได้ขนาดของอาร์เรย์ในหน่วยของไบต์ ซึ่งจะแตกต่างกันไปขึ้นอยู่กับขนาดของอาร์เรย์ที่ได้ถูกกำหนดไว้ตั้งแต่ตอนแจ้งใช้ ถ้าเราใช้โอเปอร์เรเตอร์นี้กับพอยน์เตอร์เราก็ได้ขนาดของพอยน์เตอร์ที่เท่ากันเสมอ ไม่ว่าพอยน์เตอร์นี้จะเป็นแบบใดก็ตาม

สำหรับการกำหนดค่าให้แก่ข้อมูลแต่ละตัวของอาร์เรย์ เราสามารถทำได้ดังนี้

```
int     a[2];
double  b[3];

a[0] =  3;
a[1] =  2;

b[0] =  2.0;
b[1] = -1.0;
b[2] =  3.5;
```

หรือเขียนให้อยู่ในรูปแบบที่คล้ายกับการใช้พอยน์เตอร์ เช่น

```
int     a[2];
double  b[3];

*(a+0) =  3;
*(a+1) =  2;

*(b+0) =  2.0;
*(b+1) = -1.0;
*(b+2) =  3.5;
```

## 7.1.4 การเข้าถึงข้อมูลของอาร์เรย์โดยใช้พอยน์เตอร์

ตัวอย่างต่อไป เราจะลองมาทำความเข้าใจในการเขียนโปรแกรมง่ายๆ ที่อ่านข้อมูลที่ป้อนโดยผู้ใช้ผ่านทางแป้นพิมพ์ซึ่งข้อมูลเหล่านี้เป็นข้อมูลตัวเลขใดๆก็ได้ทั้งจำนวนเต็มหรือเลขทศนิยม และเราจะให้โปรแกรมหาค่าเฉลี่ยของข้อมูลด้วย แต่ก่อนอื่นเราจะต้องกำหนดความจุที่อาร์เรย์สามารถเก็บได้ เช่น กำหนดให้เป็น 100 ดังนั้น เราสามารถเก็บข้อมูลไว้ในอาร์เรย์ได้มากที่สุด เท่ากับ 100 จำนวนเท่านั้น

```
#include <stdio.h>

#define MAX_NUM 100

int main()
{
     int i=0,num;
     int ret_val;
     double sum=0.0;
     double array[MAX_NUM];

     printf("Please enter your data.\n");
     printf("Type 'q' to stop...\n");
     do {
          printf("[%03d] : ", i+1);
          ret_val = scanf("%lf", &array[i]);
          if (ret_val) {
               printf("%lf o.k.\n", array[i]);
               sum += array[i];
               i++;
          }
          else {
               printf("Stop...\n");
          }
     } while (ret_val && (i < MAX_NUM));

     num = i;
     printf("number of inputs = %d\n", num);
     printf("sum = %lf\n", sum);
     printf("average = %lf\n", sum/num);

     return 0;
 }
```

## 7.1.5 การติดตั้งค่าเริ่มต้นให้แก่อาร์เรย์มิติเดียว

ในบางครั้งเมื่อเราแจ้งใช้ตัวแปรบางตัวที่เป็นอาร์เรย์ของข้อมูล เราก็ต้องการติดตั้งค่าเริ่มต้นให้แก่ข้อมูลภายในอาร์เรย์ไปพร้อมกันเลย ซึ่งเราก็สามารถทำได้โดยกำหนดค่าเริ่มต้นให้แก่ข้อมูลทุกตัวหรือเพียงส่วนหนึ่งของข้อมูลเท่านั้น ซึ่งในกรณีหลังนี้ จำนวนของค่าเริ่มต้นจะมีน้อยกว่าจำนวนของข้อมูลในอาร์เรย์ เช่น สมมุติว่าอาร์เรย์มีขนาดเท่ากับสิบหน่วยและเราได้ติดตั้งค่าเริ่มต้นแค่สองหน่วย ซึ่งหมายความว่าข้อมูลสองตัวแรกในอาร์เรย์จะมีค่าเริ่มต้นที่กำหนดไว้ตามลำดับ ส่วนข้อมูลตัวที่เหลือจะมีค่าเป็นศูนย์ ถ้าเราไม่ได้กำหนดค่าเริ่มต้นใดๆให้อาร์เรย์ ค่าของข้อมูลในอาร์เรย์จะเป็นอะไรก็ได้ซึ่งหมายถึงยังไม่ได้นิยาม(ยกเว้นกรณีที่แจ้งใช้ตัวแปรอาร์เรย์ให้เป็น `extern` หรือ `static` ซึ่งในกรณีนี้ข้อมูลทุกตัวในอาร์เรย์จะมีค่าเป็นศูนย์โดยอัตโนมัติ) ตัวอย่างการติดตั้งค่าเริ่มต้นให้แก่อาร์เรย์ เช่น

```
int x_vect[3] = {1,0,0};
int y_vect[3] = {0,1,0};
int z_vect[3] = {0,0,1};
float data[5] = {1.,-5.1,2.0,0,0};
char string[100] = {'H','e','l','l','o','\0'};
double table[20] = {0};
```

ถ้าเรากำหนดค่าเริ่มต้นที่มีแบบข้อมูลไม่ตรงกับแบบข้อมูลของอาร์เรย์ ค่าเริ่มต้นที่มีแบบไม่ตรงกับที่กำหนดไว้ก็จะถูกแปลงเป็นแบบข้อมูลที่ถูกต้องโดยอัตโนมัติ เช่น ในกรณีการติดตั้งค่าเริ่มต้นขอ งอาร์เรย์ `data[5]` ของข้อมูลแบบ `float` เนื่องจาก `0` เป็นจำนวนเต็มแบบ `int` ดังนั้นก็จะถูกแปลงเป็น `0.0` ก่อนโดยอัตโนมัติ

การติดตั้งค่าเริ่มต้นของอาร์เรย์ ในบางครั้งเราไม่จำเป็นต้องกำหนดขนาดของอาร์เรย์ โดยเว้นว่างไว้ ตัวอย่างเช่น

```
char hello[] = {
     'H','e','l','l','o','\0'
};
```

ในกรณีตัวอย่างนี้เราไม่ได้กำหนดขนาดของอาร์เรย์อย่างเจาะจง แต่อย่างไรก็ตาม ขนาดของอาร์เรย์นี้จะถูกกำหนดโดยจำนวนของค่าเริ่มต้นที่เราติดตั้งให้แก่อาร์เรย์นี้ ถ้าเราไม่อยากจะนับด้วยตนเองว่าอาร์เรย์มีขนาดเท่าไหร เราก็สามารถเขียนคำสั่งสั้นๆได้ดังนี้

```
sizeof (hello) / sizeof(hello[0])
```

เราก็จะทราบขนาดของอาร์เรย์ของข้อมูลแบบ `char` ชื่อ `hello` นี้ได้

## 7.1.6 การกำหนดค่าและเปรียบเทียบข้อมูลในอาร์เรย์

ถ้าเราต้องการเปรียบเทียบดูว่าตัวแปรสองตัว เช่น a และ bที่เป็นอาร์เรย์เก็บข้อมูลที่เหมือนกันหรือไม่ เราไม่สามารถเขียนง่ายๆได้ว่า

```
(a == b)
```

ถ้า a และ b เป็นตัวแปรที่มีแบบข้อมูลธรรมดาและมิใช่เป็นอาร์เรย์ นิพจน์นี้ก็จะถูกต้อง แต่ตัวแปรทั้งสองเป็นอาร์เรย์ ดังนั้นจึงเป็นการเปรียบเทียบที่อยู่ของอาร์เรย์ a กับที่อยู่ของอาร์เรย์ b เท่านั้น ซึ่งมิได้หมายถึงการเปรียบเทียบข้อมูลแต่ละตัวในอาร์เรย์ ดังนั้นนิพจน์นี้จึงหมายถึง

```
(&a[0] == &b[0])
```

ถ้าเราต้องการเปรียบเทียบระหว่างสองตัวแปรที่เป็นอาร์เรย์ เราก็ต้องเขียนฟังก์ชันขึ้นใช้ เช่น

```
int cmp_array(void *a, void *b, unsigned size);
```

และต้องกำหนดรายละเอียดของฟังก์ชันนี้ก่อน ฟังก์ชัน cmp_array () จะให้ค่าเท่ากับ 0 ถ้าอาร์เรย์ a มีข้อมูลที่ไม่เหมือนกับข้อมูลในอาร์เรย์ b ถ้าข้อมูลในทั้งสองอาร์เรย์เหมือนกันทุกตัวก็ให้ฟังก์ชันคืนค่าเท่ากับ 1 และ พารามิเตอร์ size จะเป็นขนาดของอาร์เรย์ (หรือค่าใดๆก็ได้ที่น้อยกว่าแต่จะต้องมากกว่าศูนย์) และมีหน่วยเป็นไบต์

สำหรับการกำหนดค่าของอาร์เรย์ โดยที่เราต้องการให้ค่าของข้อมูลทุกตัวในอาร์เรย์หนึ่งเหมือนกับค่าของข้อมูลในอีกอาร์เรย์หนึ่งตามลำดับ เราก็ไม่สามารถเขียนได้ว่า

```
a = b
```

และในทำนองเดียวกันกับการเปรียบเทียบอาร์เรย์ เราก็ต้องสร้างฟังก์ชันขึ้นมาใช้ในการกำหนดค่าของอาร์เรย์ เช่น

```
void assign_array(void *a, void *b, unsigned size);
```

ซึ่งเป็นการกำหนดค่าของข้อมูลใน a ให้มีค่าเท่ากับค่าของข้อมูลแต่ละตัวจาก b

## 7.1.7 การผ่านอาร์เรย์เป็นพารามิเตอร์ของฟังก์ชัน

ในบางครั้งเราก็ต้องการผ่านตัวแปรที่เป็นอาร์เรย์ให้เป็นพารามิเตอร์ตัวหนึ่งของฟังก์ชัน เช่น สมมุติว่า เรามีข้อมูลที่เก็บไว้ในอาร์เรย์และเราต้องการเรียงข้อมูลในอาร์เรย์นี้ใหม่โดยเรียงจากน้อยไปมาก เพื่อจุดประสงค์นี้เราก็สร้างฟังก์ชันขึ้นใช้ที่สามารถรับข้อมูลที่เป็นอาร์เรย์นี้ได้ และจากนั้นก็จัดการเรียงข้อมูลเหล่านี้เสียใหม่ แต่แทนที่เราจะผ่านข้อมูลทั้งหมดของอาร์เรย์ให้แก่ฟังก์ชันเหมือนกับการเรียก ใช้ฟังก์ชันโดยการผ่านค่า เราจะใช้วิธีการผ่านที่อยู่เริ่มต้นของอาร์เรย์ (Base Address ) ซึ่งทำให้โปรแกรมไม่จำเป็นต้องทำสำเนาของข้อมูลทุกตัวในอาร์เรย์ ถ้าสมมุติว่าภาษาซีอนุญาตให้เราผ่านข้อมูลทั้งหมดของอาร์เรย์ให้แก่ฟังก์ชันได้ ก็จะทำให้โปรแกรมทำงานได้ไม่ดีเท่าที่ควรโดยเฉพาะอย่างยิ่งในกรณีที่อาร์เรย์มีขนาดใหญ่มาก ดังนั้นการผ่านอาร์เรย์ให้แก่ฟังก์ชันในภาษาซีจึงเป็นการผ่านที่อยู่เริ่มต้นของอาร์เรย์เท่านั้น และจะผ่านอาร์เรย์ให้อยู่ในรูปของพอยน์เตอร์

การผ่านอาร์เรย์ให้ฟังก์ชัน เราจะทำได้หลายแบบแต่ที่นิยมใช้จะมีอยู่สองแบบ เช่น

```
func (double a[]);
```

หรือ

```
func (double *a);
```

โดยกำหนดให้ a เป็นพอยน์เตอร์ที่ชี้ไปยังอาร์เรย์ที่มีแบบข้อมูลเป็น double สำหรับอาร์เรย์ที่มีข้อมูลเป็นแบบอื่น ก็ใช้หลักการเดียวกัน a ในฟังก์ชันทั้งสองรูปแบบ จึงเป็นพอยน์เตอร์ที่ชี้ไปยังที่อยู่ของอาร์เรย์และไม่ใช่ตัวอาร์เรย์จริง ดังนั้นโอเปอร์เรเตอร์ sizeof จึงใช้ไม่ได้ผลกับตัวแปรที่เป็นพอยน์เตอร์นี้ซึ่งทำหน้าที่ชี้ไปยังที่อยู่ของอาร์เรย์ เช่น

```
#include <stdio.h>

void func1(double *a)
{
     printf("size of a = %d bytes\n", sizeof(a));
}

void func2(double a[])
{
     printf("size of a = %d bytes\n", sizeof(a));
}

int main()
{
     double array[100];
```

```
     func1(array);  /* same as func1(&array[0]) */
     func2(array);  /* same as func2(&array[0]) */
     return 0;
}
```

ผลจากโปรแกรมจะเป็นได้ดังต่อไปนี้

```
     size of a = 2 bytes
     size of a = 2 bytes
```

ซึ่งเป็นขนาดของพอยน์เตอร์ สำหรับเครื่องคอมพิวเตอร์แบบพีซี ขนาดของพอยน์เตอร์มีค่าเท่ากับสองไบต์ ดังนั้น

```
     sizeof(a)
```

จึงเป็นขนาดของ a ที่ทำหน้าที่เป็นพอยน์เตอร์ และมิใช่ขนาดของอาร์เรย์ array ที่แท้จริง เวลาเราใช้ a ภายในฟังก์ชัน ก็ใช้ตามหน้าที่ของพอยน์เตอร์ โดยทั่วไปแล้วเวลาเราผ่านอาร์เรย์ให้แก่ฟังก์ชัน เราก็มักจะกำหนดให้ฟังก์ชันนี้มีพารามิเตอร์เพิ่มขึ้นอีกหนึ่งตัวที่บ่งบอกถึงขนาดของอาร์เรย์นั้น เช่น

```
     void func1(double *a, int array_size);
     void func2(double a[], int array_size);
```

เมื่อได้รู้วิธีการผ่านอาร์เรย์ให้เป็นพารามิเตอร์ของฟังก์ชันแล้ว เราลองมาพิจารณาวิธีการสร้างฟังก์ชัน ตัวอย่าง

```
     int cmp_array( const void *a,
                    const void *b,
                    unsigned size );

     void assign_array(  void *a,
                         const void *b,
                         unsigned size );
```

ตามลำดับ

```
     int cmp_array (
               const void  *a,
               const void  *b,
               unsigned    size
          )
     {
          const char *p=(char *)a, *q=(char *)b;

          assert(p && q);
          while (size--) {
               if(*p++!=*q++) return 0;
```

```
        }
        return 1;
}
```

เนื่องจาก a และ b เป็นอาร์เรย์หรือพอยน์เตอร์ใดๆแบบ void ซึ่งหมายถึงพอยน์เตอร์อเนกประสงค์นั้นเอง และในรายการพารามิเตอร์ เรากำหนดเจาะจงลงไปอีกว่า ห้ามใช้พอยน์เตอร์ทั้งสองนี้ในการเปลี่ยนแปลงค่าในแหล่งข้อมูลที่พอยน์เตอร์ทั้งสองกำลังอ้างถึง คืออ่านได้อย่างเดียวเท่านั้น และการเข้าถึงข้อมูลแต่ละตัวของอาร์เรย์โดยใช้พอยน์เตอร์อเนกประสงค์ เราไม่สามารถทำได้ เช่น ถ้าเขียนนิพจน์ว่า *a หรือ *b ก็จะ ถือว่าผิดหลักไวยากรณ์ และเราจะต้องแจ้งใช้พอยน์เตอร์แบบ char เข้าช่วย ข้อดีของการใช้พอยน์เตอร์แบบ char คือ ขนาดของข้อมูลที่พอยน์เตอร์แบบ char ชี้ไปมีขนาดเท่ากับหนึ่งไบต์

หลังจากที่แจ้งใช้และกำหนดค่าเริ่มต้นแล้ว พอยน์เตอร์ p และ q ชี้ไปยังที่อยู่เริ่มต้นของอาร์เรย์ทั้งสองตามลำดับ และการเข้าถึงข้อมูลแต่ละตัวของอาร์เรย์ทั้งสองเราก็ใช้พอยน์เตอร์ p และ q ภายในวงวนแบบ while จะมีการเปรียบเทียบข้อมูลแต่ละตัวจากอาร์เรย์ทั้งสองเริ่มต้นจากข้อมูลตัวแรกไปจนถึงข้อมูลตัวสุดท้ายที่เรากำหนดไว้โดยค่าของ size ในระหว่างการเปรียบเทียบแต่ละครั้งนั้น ถ้ามีไบต์ตัวใดจากอาร์เรย์ทั้งสองที่มีค่าไม่เท่ากัน คำสั่ง return ก็จะทำงานและหยุดการทำงานของฟังก์ชันโดยคืนค่าเท่ากับ 0 ถ้าการเปรียบเทียบแต่ละครั้งมีค่าเป็นจริงตลอด โปรแกรมก็จะออกจากวงวนของ while เมื่อ size มีค่าลดลงจนเท่ากับศูนย์ (เพราะค่าเริ่มต้นของ size จะเป็นจำนวนเต็มบวกเท่านั้น) และคำสั่ง return ก็จะจบการทำงานของฟังก์ชันโดยคืนค่าเท่ากับ 1 ซึ่งหมายความว่า ข้อมูลทุกตัวจากอาร์เรย์ทั้งสองมีค่าเท่ากัน ในกรณีที่ค่าของ size มีค่าเท่ากับศูนย์ (กรณีพิเศษ) ฟังก์ชันจะให้ค่าเท่ากับ 1 ซึ่งไม่มีการเปรียบเทียบใดๆระหว่างอาร์เรย์ ในอีกกรณีหนึ่งที่เราต้องพิจารณาถึงก็คือ กรณีที่ผู้ใช้ได้เรียกใช้ฟังก์ชันและผ่านพอยน์เตอร์ศูนย์เป็นพารามิเตอร์ ก็จะต้องมีการตรวจสอบกรณีเช่นนี้ด้วย วิธีการก็คือการใช้ฟังก์ชันมาตราฐาน assert() ที่มีการแจ้งใช้ไว้ในไฟล์ <assert.h>

```
assert(p && q);
```

ถ้า p หรือ q มีค่าเท่ากับศูนย์ จะทำให้นิพจน์เงื่อนไข มีค่าเป็นเท็จ และฟังก์ชัน assert() ก็จะหยุดการทำงานของโปรแกรม

เราสามารถเขียนนิพจน์เงื่อนไขสำหรับ assert() ได้ใหม่ ซึ่งเราสามารถทำความเข้าใจได้ง่ายขึ้น คือ

```
assert((p!=NULL) && (q!=NULL));
```

สำหรับอีกฟังก์ชันหนึ่งคือ `assign_array()` ที่ใช้ในการกำหนดค่าของข้อมูลภายในอาร์เรย์หนึ่งให้มีค่าเท่ากับข้อมูลจากอีกอาร์เรย์หนึ่ง ก็จะมีรูปแบบของการทำงานที่คล้ายกัน

```
void assign_array (
          void        *a,
          const void  *b,
          unsigned    size
     )
{
     char *p=(char *)a;
     const char *q=(char *)b;

     assert(p && q);
     while (size--) {
          *p++ = *q++;
     }
}
```

ตัวอย่างการใช้ฟังก์ชันทั้งสองที่เราได้สร้างขึ้น เช่น

```
#define ARR_SIZE(array) (sizeof(array))

/* Function Prototypes */
extern int cmp_array(const void *,const void *,unsigned);
extern void assign_array(void *,const void *,unsigned);

int main()
{
     int  a[5]  = {1,2,3,4,5};
     int  b[10] = {1,2,3,4,5,1,2,3,4,5};
     int  c[10];

     assign_array(&c[0], &a[0], ARR_SIZE(a));
     assign_array(&c[5], &a[0], ARR_SIZE(a));


     printf("b and c are %s\n",
               cmp_array(b,c,ARR_SIZE(b)) ?
               "EQUAL" : "NOT EQUAL" );
     return 0;
}
```

ในโปรแกรมนี้เราแจ้งใช้ตัวแปรสามตัวที่เป็นอาร์เรย์ และพยายามกำหนดค่าของข้อมูลใน `c` ให้มีค่าเท่ากับข้อมูลจาก `b` โดยใช้ฟังก์ชัน `assign_array()` สองครั้ง โดยกำหนดค่าของข้อมูลทีละห้าตัว การเรียกใช้ฟังก์ชันนี้ครั้งแรกจะกำหนดค่า `c[0]` ... `c[4]` ให้มีค่าเท่ากับ `a[0]` ... `a[4]` ตามลำดับ และครั้งที่สอง

จะกำหนดให้c[5] ... c[9] ให้มีค่าเท่ากับ a[0] ... a[4] ตามลำดับ จากนั้นก็เปรียบเทียบระหว่างอาร์เรย์ b และ c ว่าเหมือนกันหรือไม่

เราจะเห็นได้ว่าเราสามารถกำหนดเลือกให้ที่อยู่ของข้อมูลตัวใดในอาร์เรย์เป็นที่อยู่ที่ผ่านเป็นพารามิเตอร์ให้ฟังก์ชันก็ได้ เช่น

```
assign_array(&c[0], &a[0], 5);
```

จะให้ผลเหมือนกับ

```
assign_array(c, a, 5);
```

เพราะเราเริ่มต้นที่ข้อมูลตัวแรกของอาร์เรย์ทั้งสอง ถ้าเราต้องการอ้างถึงข้อมูล 5 ตัวหลังจากอาร์เรย์ c ซึ่งก็คือ c[5] ... c[9] เราก็เขียนว่า

```
assign_array(&c[5], &a[0], 5);
```

หรือ

```
assign_array((c+5), &a[0], 5);
```

ย่อมให้ผลที่เหมือนกัน

จากการสร้างฟังก์ชันทั้งสองที่ใช้งานกับอาร์เรย์ เราจะเห็นได้ว่าเราได้ใช้พอยน์เตอร์อเนกประสงค์เป็นพารามิเตอร์เพื่อเก็บที่อยู่เริ่มต้นของอาร์เรย์ การใช้พอยน์เตอร์อเนกประสงค์ก็ทำให้เราสามารถผ่านอาร์เรย์แบบข้อมูลพื้นฐานใดๆก็ได้ และเราก็จะจัดการกับหน่วยความจำที่อาร์เรย์ใช้ ให้อยู่ในรูปของแถวไบต์ที่เป็นกลาง คือไม่ขึ้นกับแบบข้อมูลใดๆ ถ้าเราต้องการเข้าถึงข้อมูลขนาดหนึ่งไบต์แต่ละตัว โดยไม่สนใจว่าข้อมูลที่อยู่ในหน่วยความจำของอาร์เรย์ที่จริงแล้วเป็นข้อมูลแบบใด แต่เราจะพิจารณาข้อมูลขนาดหนึ่งไบต์ทีละตัวเท่านั้น เราก็จะใช้พอยน์เตอร์แบบ char เข้าช่วย

## 7.1.8 การนิยามแบบข้อมูลสำหรับอาร์เรย์โดยใช้ typedef

เราสามารถใช้ typedef ในการนิยามแบบข้อมูลรวมที่เป็นอาร์เรย์ได้ เช่น สมมุติว่าเราต้องการแจ้งใช้ตัวแปรที่เป็นอาร์เรย์แบบ int ขนาดความจุเท่ากับ 128 และ 512 ตามลำดับ เราก็สามารถนิยามแบบข้อมูลรวมในรูปแบบของอาร์เรย์ตามลักษณะดังกล่าวได้

```
typedef int SmallArray[128];
typedef int LargeArray[512];
```

ตัวอย่างการแจ้งใช้

```
SmallArray  a;
LargeArray  b;
```

ซึ่งหมายถึง การแจ้งใช้ตัวแปร a ที่เป็นอาร์เรย์แบบ int ขนาด 128 และ ตัวแปร b เป็นอาร์เรย์แบบ int ขนาด 512

## 7.1.9 การจัดสรรหน่วยความจำสำหรับสร้างอาร์เรย์

เวลาที่เราแจ้งใช้ตัวแปรอาร์เรย์ ก็จะมีการจัดสรรหน่วยความจำโดยอัตโนมัติ เป็นจำนวนเท่ากับจำนวนของข้อมูลในอาร์เรย์คูณด้วยขนาดของข้อมูลในหน่วยไบต์ การจัดสรรหน่วยความจำนี้เป็นการใช้หน่วยความจำแบบตายตัว ซึ่งหมายความว่า เราจะต้องกำหนดขนาดของอาร์เรย์ใดๆที่เราจะใช้ในโปรแกรมตั้งแต่ต้น ขนาดของอาร์เรย์จะต้องคำนวณได้เมื่อเวลาคอมไพล์โปรแกรมโค้ด ในบางกรณีถ้าเรากำหนดขนาดของอาร์เรย์ที่มากหรือน้อยเกินไป ก็จะเป็นปัญหาและบ่งบอกถึงคุณสมบัติของอาร์เรย์ในแง่ลบ

เราสามารถใช้พอยน์เตอร์ที่ทำหน้าที่คล้ายอาร์เรย์ได้ แต่เราจะต้องจัดสรรบล็อกหน่วยความจำให้พอยน์เตอร์นี้เอง สำหรับการจัดสรรหน่วยความจำนั้น เราจะใช้ฟังก์ชันมาตราฐาน คือ malloc() ซึ่งมีรูปแบบของฟังก์ชันที่ได้นิยามไว้ในแฟ้ม <stdlib.h>

```
void *malloc (size_t size);
```

เมื่อเวลาเรียกใช้ฟังก์ชัน malloc() เราจะต้องกำหนดขนาดของหน่วยความจำที่เราต้องการจองไว้ใช้ ในหน่วยของไบต์ ขนาดของหน่วยความจำจะถูกกำหนดโดยใช้พารามิเตอร์ size ที่มีแบบข้อมูลเป็น size_t ซึ่งนิยามไว้ในแฟ้ม <stdlib.h>โดยคำสั่ง typedef (หรือ unsigned นั้นเอง) ถ้าคอมพิวเตอร์สามารถจัดสรรหน่วยความจำได้ตามที่เราต้องการ ฟังก์ชัน malloc() ก็จะให้พอยน์เตอร์อเนกประสงค์ที่ชี้ไปยังที่อยู่เริ่มต้นของบล็อกหน่วยความจำดังกล่าว ถ้าคอมพิวเตอร์ไม่สามารถจัดสรรหน่วยความจำตามที่เราต้องการได้ ฟังก์ชันก็จะให้พอยน์เตอร์ที่มีค่าเป็นศูนย์กลับคืน

ตัวอย่างการเรียกใช้ฟังก์ชัน malloc() เช่น สมมุติว่า เราต้องการจัดสรรหน่วยความจำเพื่อใช้ในงานที่คล้ายกับการใช้อาร์เรย์แบบ double ก็จะมีวิธีการดังนี้

```
double *p;
int size = 10;

p = (double *)malloc(size * sizeof(double));
if (p==NULL) {
     printf("Cannot allocate memory!!\n");
     exit(1);
}
```

ในกรณีนี้ เราได้พยายามจัดสรรหน่วยความจำขนาดเท่ากับ 80 ไบต์ โดยใช้ฟังก์ชัน malloc() และกำหนดให้ฟังก์ชันผ่านค่ากลับคืนไปยังพอยน์เตอร์ p แบบ double ดังนั้น ถ้า p มีค่าไม่เท่ากับศูนย์ ก็หมายความว่า เราสามารถจัดสรรหน่วยความจำตามที่ต้องการได้สำเร็จ แต่ถ้า p มีค่าเท่ากับศูนย์ เราก็จำเป็นต้องสร้างสายงานในการตรวจสอบค่าของ p และแจ้งให้ทราบ ถ้าคอมพิวเตอร์ไม่สามารถจัดสรรหน่วยความจำตามที่เราต้องการได้ หรือหยุดการทำงานของโปรแกรมโดยใช้ประโยคคำสั่ง exit(1)

เมื่อจัดสรรหน่วยความจำได้แล้วพอยน์เตอร์ p ก็จะเป็นเสมือนกุญแจสำคัญในการเข้าถึงที่อยู่ของหน่วยความจำที่ได้จองไว้ ถ้าเราเปลี่ยนแปลงค่าของ p เราก็ไม่สามารถเข้าถึงแหล่งข้อมูลดังกล่าวได้อีก เรากล่าวได้ว่าพอยน์เตอร์ p เป็นเจ้าของพื้นที่หน่วยความจำที่ได้ถูกจัดสรรไว้

การใช้พอยน์เตอร์ p เราก็สามารถใช้ได้เหมือนกับตัวแปรที่เป็นอาร์เรย์ทั่วไป ถ้าเราต้องการเข้าถึงข้อมูลขนาด 8 ไบต์ แบบ double เราก็ใช้โอเปอร์เรเตอร์ [] ตัวอย่างเช่น

```
p[0] = 1.0;
p[1] = 2.0;
p[2] *= p[1];
```

โปรดสังเกตว่า เราสามารถใช้โอเปอร์เรเตอร์ [] กับพอยน์เตอร์ได้ ซึ่งใช้ในการเข้าถึงข้อมูลแต่ละตัว

```
double *q = p;
(q++)[0] = 1.0;
```

หมายถึง การผ่านค่าของ p ให้พอยน์เตอร์ q จากนั้นเราก็ใช้พอยน์เตอร์นี้ ในการเข้าถึงข้อมูลแบบ double ตัวแรกในที่อยู่ที่พอยน์เตอร์ q กำลังชี้ไป เนื่องจากว่า เราได้ใช้โอเปอร์เรเตอร์ ++ กับตัวแปรพอยน์เตอร์ q ดังนั้นนิพจน์ดังกล่าวจะหมายถึงการกำหนดค่าข้อมูลตัวแรกในหน่วยความจำมีค่าเท่ากับ 1.0 และพอยน์เตอร์ q ก็จะมีค่าเพิ่มขึ้นอีกแปดไบต์ ซึ่งหมายถึงการชี้ข้อมูลแบบ double ตัวถัดไปในหน่วยความจำ

ถ้าเราต้องการเขียนนิพจน์ที่มีความหมายเหมือนกับ

```
(q++)[0] = 1.0;
```

ก็จะเขียนได้ใหม่ดังนี้

```
*(q++) = 1.0;
```

หรือ

```
*q++ = 1.0;
```

ซึ่งดูแล้วเข้าใจได้ง่ายกว่า และ ย่อมให้ผลเหมือนกัน

หน่วยความจำที่เราได้จัดสรรเอาไว้ เราสามารถให้กลับคืนเป็นหน่วยความจำว่างก็ได้ เพื่อใช้ในจุดประสงค์อื่นต่อไป เราจะเรียกใช้ฟังก์ชันมาตราฐาน ชื่อ free()

```
void free(void *p);
```

ซึ่งต้องการเพียงแค่พอยน์เตอร์เป็นพารามิเตอร์

```
free(p);
```

เมื่อฟังก์ชันทำงานจบแล้วหน่วยความจำที่ได้จองไว้โดยพอยน์เตอร์ p ก็จะถูกปล่อยว่าง ดังนั้นเราไม่สามารถใช้โอเปอร์เรเตอร์ * กับพอยน์เตอร์ p ได้อีก

มีข้อสังเกตอยู่ว่าหน่วยความจำที่ได้จากการใช้ฟังก์ชัน malloc() จะยังมิได้ถูกติดตั้งค่าเริ่มต้น ดังนั้นค่าของข้อมูลจึงเป็นอะไรก็ได้ซึ่งหมายถึงการไม่ทราบค่าที่แน่นอน ถ้าเราต้องการจัดสรรหน่วยความจำและกำหนดให้ทุกๆไบต์ของหน่วยความจำนี้มีค่าเป็นศูนย์ เราก็จะใช้ฟังก์ชันมาตราฐาน calloc() ซึ่งนิยามไว้ใน <stdlib.h>

```
void *calloc(size_t n, size_t element_size);
```

พารามิเตอร์ของฟังก์ชัน calloc() จะแตกต่างจากพารามิเตอร์ของ malloc() คือ เราจะต้องกำหนดจำนวนของข้อมูลและขนาดของข้อมูลดังกล่าว ดังนั้น n จึงหมายถึงจำนวนข้อมูล และ element_size หมายถึงขนาดของข้อมูลแต่ละตัวในหน่วยของไบต์ (แน่นอน ข้อมูลแต่ละตัวต้องมีขนาดเท่ากัน) ตัวอย่างการใช้ฟังก์ชัน calloc()

```
double *p;
int size = 10;

p = (double *)calloc(size, sizeof(double));
if (p==NULL) {
     printf("Cannot allocate memory!!\n");
```

```
        exit(1);
    }
```

ถ้าเรามีหน่วยความจำที่ถูกจัดสรรโดย malloc() หรือ calloc() และต้องการย่อหรือขยายบล็อกหน่วยความจำที่มีอยู่ ให้เล็กลงหรือใหญ่ขึ้น เราจะใช้ฟังก์ชันมาตราฐาน realloc()

```
void *realloc(void *p, size_t size);
```

โดยที่พารามิเตอร์ size จะเป็นขนาดของบล็อกหน่วยความจำใหม่ที่เราต้องการ

ในกรณีที่เราต้องการขยายบล็อกของหน่วยความจำที่มีที่อยู่เริ่มต้นที่พอยน์เตอร์ p กำลังชี้ไป ฟังก์ชันก็จะพยายามขยายขนาดของบล็อกหน่วยความจำโดยพยายามที่จะไม่เปลี่ยนที่อยู่เริ่มต้น ถ้าเป็นไปไม่ได้ฟังก์ชันก็จะหาที่อยู่ใหม่ที่มีขนาดของหน่วยความจำกว้างพอ และจะทำสำเนาค่าของข้อมูลจากที่อยู่เดิมไปยังที่อยู่ใหม่ ส่วนหน่วยความจำตามที่อยู่เดิม ก็จะถูกปล่อยว่างไป หน่วยความจำที่เพิ่มเติมขึ้นมาจะไม่มีการติดตั้งค่าเริ่มต้นใดๆเมื่อฟังก์ชันได้จัดสรรหน่วยความจำแล้วก็จะให้พอยน์เตอร์ที่ชี้ไปยังที่อยู่เริ่มต้นของบล็อกหน่วยความจำที่ถูกต้องหรือให้พอยน์เตอร์ศูนย์กลับคืน ในกรณีที่ไม่สามารถจัดสรรตามที่ต้องการได้ ตัวอย่างการใช้ realloc()

```
double *p;
p = (double *)malloc( 10*sizeof(double) );
p = (double *)realloc(p, 100*sizeof(double) );
```

สำหรับพอยน์เตอร์ p ที่ได้ใช้กับฟังก์ชัน free() แล้ว ถ้าเราต้องการใช้พอยน์เตอร์นี้กับ realloc() เราควรจะกำหนดค่าของพอยน์เตอร์ p เป็นศูนย์ (NULL) ก่อน เพื่อเป็นการเจาะจงว่าพอยน์เตอร์ตัวนี้ไม่ได้ใช้จองบล็อกหน่วยความจำใดๆ

```
double *p;
p = (double *)malloc( 10*sizeof(double) );

free(p);
p = NULL;
p = (double *)realloc( 100*sizeof(double) );
```

โปรดสังเกตว่าถ้าเราผ่านพอยน์เตอร์ที่เป็นพอยน์เตอร์ศูนย์ให้ฟังก์ชัน realloc() แล้วการทำงานของฟังก์ชันนี้ก็จะให้ผลเหมือนกับฟังก์ชัน malloc()

ห้ามใช้ฟังก์ชัน free() และ realloc() กับตัวแปรที่เป็นอาร์เรย์โดยเด็ดขาด ตัวอย่างที่ผิด เช่น

```
int IntArray[10];
```

```
free(IntArray);
```

โปรดสังเกตว่า ถ้าเราได้จองหน่วยความจำไว้ใช้ โดยเรียกใช้ฟังก์ชัน `malloc()` หรือ `calloc()` ก็ตามเมื่อเราไม่ได้ใช้หน่วยความจำนี้อีกต่อไป เราก็ควรใช้คำสั่ง `free()` ในการปล่อยให้หน่วยความจำเหล่านี้กลับคืนไปยังส่วนกลางเพื่อใช้ได้ในจุดประสงค์และเวลาอื่นต่อไป จุดนี้ถือว่าสำคัญมากเพราะถ้าไม่ทำเช่นนี้หน่วยความจำที่ว่างก็จะลดลงเรื่อยๆทั้งๆที่ในความเป็นจริงแล้ว เราสามารถกำหนดให้พื้นที่หน่วยความจำว่างได้ บางครั้งอาจจะเป็นเพราะการเขียนโปรแกรมที่ไม่ระมัดระวัง ตัวอย่างเช่น

```
double *p;
p = (double *)malloc( 10*sizeof(double) );
p = (double *)malloc( 10*sizeof(double) );
```

ตัวอย่างนี้เป็นการจัดสรรหน่วยความจำสองครั้งทั้งๆที่เราใช้แค่บล็อกเดียว ดังนั้นถ้าเราได้กำหนดให้พอยน์เตอร์ `p` เป็นเจ้าของบล็อกหน่วยความจำครั้งหนึ่งแล้ว ถ้าต้องการจะใช้คำสั่ง `malloc()` อีกครั้ง จะต้องใช้คำสั่ง `free()` กับพอยน์เตอร์ `p` ก่อนหรือใช้พอยน์เตอร์ตัวอื่นแทน

ข้อเสียอีกข้อหนึ่งสำหรับอาร์เรย์และการใช้บล็อกหน่วยความจำในเชิงอาร์เรย์โดยอาศัยพอยน์เตอร์คือ จะไม่มีการตรวจสอบขอบเขตของการใช้เลขดัชนี ยกตัวอย่างเช่น ถ้าเราแจ้งใช้อาร์เรย์เป็นจำนวน 10 หน่วย แต่ถ้าเราใช้ดัชนีที่มีค่าตั้งแต่ 10 ขึ้นไปกับโอเปอร์เรเตอร์ `[]` ผลก็คือเราจะเข้าถึงข้อมูลที่อยู่นอกบล็อกหน่วยความจำสำหรับอาร์เรย์ (out of bounds) ดังนั้นถ้าอ่านข้อมูลที่อยู่นอกขอบเขต ก็จะได้ข้อมูลที่ถือว่าไม่ถูกต้อง ถ้าพยายามเขียนข้อมูลทับ ก็อาจจะทำให้เกิดปัญหาในการทำงานของโปรแกรมได้

## 7.1.10 การสร้างอาร์เรย์ที่มีข้อมูลโดยสุ่ม (Random Numbers)

อีกตัวอย่างหนึ่งที่เกี่ยวกับการใช้งานอาร์เรย์คือ การสร้างข้อมูลตัวเลขโดยสุ่ม และเก็บข้อมูลที่ได้นี้ไว้ในอาร์เรย์ การสร้างตัวเลขแบบสุ่มเราจะใช้ฟังก์ชันมาตราฐานชื่อ `srand()` และ `rand()` นิยามไว้ในไฟล์ `<stdlib.h>`

```
void srand( unsigned int seed);
int rand(void);
```

ฟังก์ชัน `srand()` ใช้ในการติดตั้งตัวสร้างเลขโดยสุ่ม (Random Number Generator) เป็นครั้งแรกก่อนที่จะมีการเรียกใช้ฟังก์ชัน `rand()` ฟังก์ชัน `srand()` พยายามสร้างลำดับของตัวเลข โดยอาศัยค่าจากพารามิเตอร์ `seed` ในการเรียกใช้ฟังก์ชัน `rand()` แต่ละครั้ง เราจะได้ค่ากลับคืนจากฟังก์ชัน และค่านี้เป็นค่าที่ได้จากการสุ่มเลือกจากค่าต่างๆภายในลำดับของตัวเลขที่มีอยู่ ซึ่งเป็นตัวเลขแบบ `int` และจะมีค่าอยู่ระ-หว่าง 0 และ `RAND_MAX` โดยที่ `RAND_MAX` เป็นสัญลักษณ์ของค่าคงที่แบบ `int` ถ้า `int` มีขนาดเท่ากับ 16 บิตแล้ว `RAND_MAX` จะมีค่าเท่ากับ `32767` ตัวอย่างการใช้ฟังก์ชันทั้งสอง เช่น

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

int main()
{
     int i;
     int array[5];
     int seed = (int)time(NULL);

     srand(seed);
     for(i=0; i<5; i++)
          printf("%d\n",(array[i]=rand()%100));

     return 0;
}
```

ในโปรแกรมตัวอย่างนี้ เราใช้ฟังก์ชันมาตราฐานอีกตัวหนึ่งชื่อ `time()` นิยามไว้ใน `<time.h>`

```
time_t time(time_t *timer);
```

ฟังก์ชัน `time()` จะให้จำนวนของวินาทีที่ผ่านไปแล้ว โดยนับตั้งแต่วันที่ 1 มกราคม ค.ศ. 1970 จนถึงเวลาที่ได้เรียกใช้ฟังก์ชัน และเราจะเก็บค่าจากฟังก์ชันนี้ไว้ในตัวแปร `seed` ซึ่งหมายความว่า ถ้าเราเรียก

ใช้โปรแกรมแต่ละครั้ง เราก็จะได้ค่าของ seed ที่แตกต่างกันและเมื่อเราผ่านค่าของตัวแปรนี้ไปให้ฟังก์ชัน srand() ความน่าจะเป็นที่เราจะได้ลำดับตัวเลขที่แตกต่างกันออกไปในการเรียกใช้แต่ละครั้งจึงมีมาก

สำหรับนิพจน์

```
(array[i]=rand()%100)
```

หมายถึง การอ่านค่าจากฟังก์ชัน rand() และค่าที่ได้จะเป็นเลขที่ถูกสร้างขึ้นโดยการสุ่มเลือก แต่ถ้าเราต้องการให้ตัวเลขที่ได้มีค่าอยู่ระหว่าง 0 ถึง 99 เท่านั้น ตามตัวอย่างเราก็ใช้โอเปอร์เรเตอร์ % เข้าช่วย และเก็บค่าผลลัพธ์ไว้ในข้อมูลภายในอาร์เรย์

## 7.1.11 ตัวอย่างการประมวลผลข้อมูลในอาร์เรย์

ตัวอย่างโปรแกรมต่อไปนี้ เป็นการใช้อาร์เรย์ในการเก็บข้อมูลตัวเลขแบบ double ซึ่งผู้ใช้โปรแกรมจะต้องป้อนข้อมูลให้โปรแกรมผ่านทางแป้นพิมพ์ โดยเราจะใช้ฟังก์ชัน scanf() ในการอ่านข้อมูลแต่ละตัว เมื่อโปรแกรมเริ่มทำงาน ก็จะให้ผู้ใช้กำหนดจำนวนของข้อมูลที่ต้องการจะป้อนให้โปรแกรม เนื่องจากว่าเราใช้อาร์เรย์ที่มีขนาดตายตัวในการเก็บข้อมูล ดังนั้นจำนวนข้อมูลทั้งหมดที่เป็นไปได้จะต้องไม่เกินขนาดความจุของอาร์เรย์ ในกรณีเรากำหนดให้มีค่าเท่ากับ 100 โดยใช้สัญลักษณ์ชื่อ MAX_NUM แทนที่ค่าคงที่ตัวนี้ เมื่อผู้ใช้ได้กำหนดจำนวนของข้อมูลทั้งหมดแล้วโปรแกรมก็จะเริ่มอ่านข้อมูลทีละตัวจากผู้ใช้จนครบโดยเก็บข้อมูลไว้ในหน่วยความจำของอาร์เรย์ตามลำดับ จากนั้นก็เริ่มประ-มวลผลข้อมูลที่อยู่ในอาร์เรย์ โดยการผ่านพารามิเตอร์ที่ต้องการไปยังฟังก์ชัน stats() ซึ่งใช้ในการหาค่าต่างๆ เช่น ค่าที่มากและน้อยที่สุดของข้อมูลที่มีอยู่ หรือหาผลรวมและค่าเฉลี่ย

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

#define  MAX_NUM  100

/* Function Prototype */
void stats (   const double *, int, double *,
               double *, double *, double *  );

int main()
{
     int       i, n=0;
     double    array [MAX_NUM];
     double    average, max, min, sum;
```

```
     printf("Please enter the number of values(max.%d): ",
               MAX_NUM);
     scanf("%d", &n);

     if((n <= 0)||(n > MAX_NUM))
     {
          printf("The number of values is invalid!");
          return 1;
     }

     printf("Please enter your data now: \n");
     for (i=0; i < n; i++)
          scanf("%lf", &array[i]);

     stats(array,n,&min,&max,&sum,&average);

     printf("%s%4d %s%7.2lf %s%7.2lf %s%7.2lf %s%7.2lf\n",
          "\nNumber of values: ", n,
          "\nMinimum : ", min,
          "\nMaximum : ", max,
          "\nSum     : ", sum,
          "\nAverage : ", average);
     return 0;
}

void stats(const double *a,
                int     n,
                double *min,
                double *max,
                double *sum,
                double *average)
{
     int i;

     assert(a && min && max && sum && average && (n>0));
     *sum = 0.0;
     *min = *max = a[0];
     for (i=0;i<n;i++)
     {
          if(a[i] < *min) *min = a[i];
          if(a[i] > *max) *max = a[i];
          *sum += a[i];
     }
     *average = *sum /(double)n;
}
```

ถ้าจะดัดแปลงให้อาร์เรย์มีขนาดที่ขึ้นอยู่กับจำนวนข้อมูลที่ป้อนให้กับโปรแกรม เราก็สามารถทำได้โดยใช้พอยน์เตอร์และจัดสรรพื้นที่หน่วยความจำด้วยตนเอง

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

int main()
{
     int       i, n=0;
     double    *array;
     double    average, max, min, sum;

     printf("Please enter the number of values : ");
     scanf("%d", &n);

     if(n <= 0)
     {
          printf("The number of values is invalid!");
          return 1;
     }

     array = (double *)malloc(n * sizeof(double));

     if (array==NULL)
     {
          printf("Sorry, cannot allocate memory!");
          return 2;
     }

     printf("Please enter your data now: \n");
     for (i=0; i < n; i++)
          scanf("%lf", &array[i]);

     stats(array,n,&min,&max,&sum,&average);

     printf("%s%4d %s%7.2lf %s%7.2lf %s%7.2lf %s%7.2lf\n",
          "\nNumber of values: ", n,
          "\nMinimum : ", min,
          "\nMaximum : ", max,
          "\nSum     : ", sum,
          "\nAverage : ", average);
     free(array);
     return 0;
}
```

## 7.1.12 การเรียงข้อมูลในอาร์เรย์ตามลำดับ

สมมุติว่าเรามีข้อมูลอยู่หลายจำนวนภายในอาร์เรย์ อาจจะเป็นข้อมูลที่เป็นเลขจำนวนเต็มหรือเลขทศนิยมก็ได้ ในบางครั้งเราก็ต้องการเรียงข้อมูลเหล่านี้เสียใหม่ เช่น เรียงจากน้อยไปมากตามค่าของข้อมูลแต่ละตัว การเรียงข้อมูลถือว่าเป็นปัญหาพื้นฐานอย่างหนึ่งในวิชาการทางคอมพิวเตอร์ สำหรับการเรียงข้อมูลก็มีอัลกอริทึมหรือวิธีการแก้ไขปัญหาที่ได้ถูกคิดค้นขึ้นอยู่หลายๆทาง แต่ที่เราจะทำความเข้าใจในตอนนี้เรียกว่า Bubble Sort ทำไมจึงเขาจึงใช้ ชื่อนี้ เราก็จะหาคำตอบได้โดยสังเกตจากวิธีการเรียงข้อมูล

วิธีการเรียงข้อมูลแบบ Bubble Sort

1. กำหนดให้ a เป็นอาร์เรย์ที่มีข้อมูลทั้งหมด n จำนวน
2. เปรียบเทียบข้อมูลระหว่างข้อมูลสองตัวที่อยู่ในตำแหน่งติดกันเป็นคู่ๆไป

ถ้าเรียงจากค่าน้อยไปมาก

สำหรับข้อมูลคู่ใด ที่มีข้อมูลที่อยู่ทางซ้ายมากกว่าข้อมูลที่อยู่ทางขวาก็ให้สลับตำแหน่งกัน

ถ้าเรียงจากค่ามากไปน้อย

สำหรับข้อมูลคู่ใด ที่มีข้อมูลที่อยู่ทางซ้ายน้อยกว่าข้อมูลที่อยู่ทางขวาก็ ให้สลับตำแหน่งกัน

3. กลับไปทำขั้นตอนที่ 2 อีกครั้ง จนกว่าจะครบทั้งหมด n ครั้ง

ตัวอย่างการใช้เรียงข้อมูลแบบ Bubble Sort จะเป็นดังนี้

```
#include <stdio.h>

#define  N  10

int main()
{
     int a[N] = {5,3,1,4,10,2,8,6,7,7};

     int i, j, n = N, tmp;

     int n_cmp =0;  /* number of comparisons */
     int n_exch=0;  /* number of exchanges   */

     printf("Unsorted : ");
     for (i=0; i < n; i++)
          printf("%d ", a[i]);
     printf("\n");
```

```
    for (i=0; i < n; i++)
    {
         printf("%2d. pass : ", i+1);
         for (j=1; j < n ; j++, n_cmp++)
         {
            if (a[j-1] > a[j])
            {
               tmp    =  a[j-1];
               a[j-1] =  a[j];
               a[j]   =  tmp;
               n_exch++;
            }
            printf("%d ", a[j-1]);
         }
         printf("%d\n", a[n-1]);
    }

    printf("number of comparisons : %d\n", n_cmp);
    printf("number of exchanges   : %d\n", n_exch);

  return 0;
}
```

ทุกครั้งที่มีการสลับที่ระหว่างข้อมูล ค่าของตัวแปร n_exch จะเพิ่มขึ้นทีละหนึ่ง และทุกครั้งที่มีการเปรียบเทียบกันระหว่างข้อมูลสองตัว ค่าของตัวแปร n_cmp ก็จะเพิ่มขึ้นทีละหนึ่ง

ผลจากโปรแกรมจะเป็นดังนี้

```
    Unsorted : 5 3 1 4 10 2 8 6 7 7
     1. pass : 3 1 4 5 2 8 6 7 7 10
     2. pass : 1 3 4 2 5 6 7 7 8 10
     3. pass : 1 3 2 4 5 6 7 7 8 10
     4. pass : 1 2 3 4 5 6 7 7 8 10
     5. pass : 1 2 3 4 5 6 7 7 8 10
     6. pass : 1 2 3 4 5 6 7 7 8 10
     7. pass : 1 2 3 4 5 6 7 7 8 10
     8. pass : 1 2 3 4 5 6 7 7 8 10
     9. pass : 1 2 3 4 5 6 7 7 8 10
    10. pass : 1 2 3 4 5 6 7 7 8 10
    number of comparisons : 90
    number of exchanges   : 15
```

เพื่อให้เห็นวิธีการทำงานในแต่ละรอบ เราลองพิจารณาการทำงานในรอบที่ 1

```
    Unsorted : 5 3 1 4 10 2 8 6 7 7
```

| | |
|---|---|
| 5 กับ 3 ได้สลับที่กัน | 3 **5** **1** 4 10 2 8 6 7 7 |
| 5 กับ 1 ได้สลับที่กัน | 3 1 **5** **4** 10 2 8 6 7 7 |
| 5 กับ 4 ได้สลับที่กัน | 3 1 4 **5** **10** 2 8 6 7 7 |
| ไม่มีการสลับที่ | 3 1 4 5 **10** **2** 8 6 7 7 |
| 10 กับ 2 ได้สลับที่กัน | 3 1 4 5 2 **10** **8** 6 7 7 |
| 10 กับ 8 ได้สลับที่กัน | 3 1 4 5 2 8 **10** **6** 7 7 |
| 10 กับ 6 ได้สลับที่กัน | 3 1 4 5 2 8 6 **10** **7** 7 |
| 10 กับ 7 ได้สลับที่กัน | 3 1 4 5 2 8 6 7 **10** **7** |
| 10 กับ 7 ได้สลับที่กัน | 3 1 4 5 2 8 6 7 7 10 |

```
1. pass : 3 1 4 5 2 8 6 7 7 10
```

เราจะเห็นได้ว่า การเรียงข้อมูลแบบ Bubble Sort จะมีการเปรียบเทียบข้อมูล เท่ากับ `n*(n-1)` ครั้ง เช่น `n` มีค่าเท่ากับ 10 ก็จะมีการเปรียบเทียบข้อมูลเท่ากับ 90 ครั้ง ข้อเสียของ Bubble Sort คือ เมื่อข้อมูลได้ถูกเรียงลำดับถูกต้องแล้ว การเปรียบเทียบระหว่างข้อมูลยังคงมีอยู่ต่อไป จนกว่าจะครบรอบที่ `n` จากตัวอย่างเราจะเห็นได้ว่าหลังจากการวนรอบที่ 4 ข้อมูลได้เรียงอย่างถูกต้องแล้ว ถ้าเราต้องการจะดัดแปลงให้ Bubble Sort ทำงานได้มีประสิทธิภาพมากขึ้น เราจะใช้วิธีตรวจสอบว่าข้อมูลได้เรียงอยู่ตามลำดับที่ถูกต้องแล้วหรือยัง ซึ่งการตรวจสอบนี้จะมีขึ้นหลังจากการวนแต่ละรอบ โดยอาศัยตัวแปร `n_exch` ถ้า `n_exch` มีค่าเท่ากับศูนย์ก็หมายความว่า ในรอบที่ผ่านมาไม่มีการสลับที่ข้อมูลใดทั้งสิ้น เพราะเนื่องจากว่า ข้อมูลได้เรียงอยู่อย่างถูกต้องแล้ว เมื่อเป็นเช่นนี้เราก็สั่งให้วงวนหยุดการทำงานได้

```
#include <stdio.h>

#define  N  10

int main()
{
     int a[N] = {5,3,1,4,10,2,8,6,7,7};
     int i, j, n = N, tmp;

     int n_cmp = 0;     /* number of comparisons */
     int n_exch= 0;     /* number of exchanges   */

     printf("Unsorted : ");
     for (i=0; i < n; i++)
          printf("%d ", a[i]);
     printf("\n");

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

```
    {
        printf("%2d. pass : ", i+1);
        for (j=1; j < n ; j++, n_cmp++)
        {
           if (a[j-1] > a[j])
           {
             tmp    =  a[j-1];
             a[j-1] =  a[j];
             a[j]   =  tmp;

             /* increase number of exchanges */
             n_exch++;
           }
           printf("%d ", a[j-1]);
        }
        printf("%d\n", a[n-1]);

        /* detect if no exchanges are made */
        if (n_exch==0)
              break;
        else
              n_exch = 0;
    }

    printf("number of comparisons : %d\n", n_cmp);
    printf("number of exchanges   : %d\n", n_exch);

    return 0;
}
```

ผลของโปรแกรมคือ

```
    Unsorted : 5 3 1 4 10 2 8 6 7 7
     1. pass : 3 1 4 5 2 8 6 7 7 10
     2. pass : 1 3 4 2 5 6 7 7 8 10
     3. pass : 1 3 2 4 5 6 7 7 8 10
     4. pass : 1 2 3 4 5 6 7 7 8 10
     5. pass : 1 2 3 4 5 6 7 7 8 10
    number of comparisons : 45
    number of exchanges   : 15
```

ซึ่งจำนวนของการเปรียบเทียบข้อมูลได้ลดลงอย่างมากซึ่งหมายถึงประสิทธิภาพในการทำงานของโปรแกรมนั้นเอง

จากตัวอย่างที่แล้วเราได้ลองเขียนโปรแกรมที่จัดเรียงข้อมูลแบบ `int` ในอาร์เรย์ ถ้าเราต้องการเขียนฟังก์ชันที่จัดเรียงข้อมูลใดๆในอาเรย์ โดยใช้วิธีการตามแบบ Bubble Sort เราสามารถทำได้ เช่น เราใช้ชื่อว่า `bubble_sort` แต่จะต้องมีขั้นตอนที่ใช้จัดการกับรูปแบบของข้อมูล เช่น สมมุติว่าเราต้องการผ่านอาร์เรย์ที่

มีข้อมูลแบบ int หรือ double หรือข้อมูลรูปแบบอื่น เราก็ต้องอาศัยพอยน์เตอร์อเนกประสงค์แบบ void เข้าช่วยเพื่อที่จะชี้ไปยังข้อมูลแบบใดๆก็ได้ นอกจากนั้นเราก็ต้องกำหนดขนาดของข้อมูลแต่ละตัว และจำนวนข้อมูลที่มีอยู่ในอาร์เรย์หรือจำนวนของข้อมูลที่เราต้องการให้ฟังก์ชันเรียงลำดับใหม่ ปัญหาอยู่ที่ว่าเวลาเราผ่านที่อยู่ของอาร์เรย์ให้ฟังก์ชันแล้ว ภายในฟังก์ชันเราจะไม่สามารถรู้ได้ว่าแบบข้อมูลของอาร์เรย์เป็นแบบใด ดังนั้นการเปรียบเทียบระหว่างข้อมูลสองตัวจึงเป็นไปไม่ได้ ดังนั้นเพื่อที่จะแก้ปัญหานี้เราจะใช้พารามิเตอร์อีกตัวที่เป็นพอยน์เตอร์ที่ชี้ไปยังฟังก์ชันตัวใดๆก็ได้ที่เราต้องการใช้ในการเปรียบเทียบข้อมูล พอยน์เตอร์ is_greater จะต้องสร้างขึ้นอย่างเหมาะสม เพื่อที่จะสามารถใช้เปรียบเทียบข้อมูลสองตัวได้

```
int (*is_greater)(const void *, const void *);
```

เช่น พอยน์เตอร์ตัวนี้เราสามารถกำหนดให้ชี้ไปยังฟังก์ชันในรูปแบบต่อไปนี้ได้

```
int int_cmp(const void *p1, const void *p2)
{
  return ((*(int *)p1 > *(int *)p2) ? 1 : 0);
}
```

และฟังก์ชันตัวนี้ก็จะเปรียบเทียบข้อมูลในรูปแบบของข้อมูลที่เป็น int หรือถ้าเราต้องการเปรียบเทียบข้อมูลในรูปแบบของข้อมูลที่เป็น double ก็ทำได้ดังนี้

```
int dbl_cmp(const void *p1, const void *p2)
{
  return ((*(double *)p1 > *(double *)p2) ? 1 : 0);
}
```

ดังนั้น ถ้าเราต้องการเรียงข้อมูลจากอาร์เรย์แบบ int เราก็กำหนดให้พอยน์เตอร์ is_greater ชี้ไปยังฟังก์ชัน int_cmp() ถ้าต้องการเรียงข้อมูลแบบ double ก็กำหนดให้ is_greater ชี้ไปยังฟังก์ชัน dbl_cmp() หรือ ถ้าเรามีข้อมูลแบบอื่นๆ เราก็สร้างฟังก์ชันที่ใช้ในการเปรียบเทียบข้อมูลขึ้นเพิ่มเติมได้

โปรดสังเกตว่า พอยน์เตอร์ p1 และ p2 ที่เป็นพารามิเตอร์ จะมีแบบข้อมูลเป็น void ดังนั้น เราต้องแปลงเป็นแบบข้อมูลอื่นก่อน เช่น int หรือ double เป็นต้น โดยการแปลงแบบพอยน์เตอร์แบบเจาะจง เช่น

```
(double *)p1
```

นิพจน์นี้จะให้พอยน์เตอร์ที่ชี้ไปยังข้อมูลแบบ double ดังนั้น

```
*(double *)p1
```

จึงหมายถึงการเข้าถึงข้อมูลที่เป็น double โดยอาศัยพอยน์เตอร์

ส่วนรูปแบบการสร้างฟังก์ชัน bubble_sort() ก็จะมีรูปร่างหน้าตาดังนี้

```
/* function that sorts an array of n objects in ascending
 * order.
 *
 * keysize          specifies the length of each object.
 * is_greater       the array will be sorted according to
 *                  function pointed to by f_greater
 *                  (*is_greater)(p1, p2) returns 1 if p1
 *                  is greater than p2 else returns 0.
 */
void bubble_sort
 (   void *  array,
     size_t  n,
     size_t  keysize,
     int (*is_greater)(const void *, const void *) )
{

     size_t   i, j;
     size_t   n_exch = 0;

     char *a   = (char *)array;
     void *tmp = malloc(keysize+1);
     void *p1, *p2;

     for (i=0; i < n; i++) {
          for (j=1; j < n ; j++)
          {
            p1 = (void *)(a + keysize*(j-1));
            p2 = (void *)(a + keysize*(j));
            if ( is_greater(p1, p2) )
            {
               memcpy(tmp, p1, keysize);
               memcpy( p1, p2, keysize);
               memcpy( p2,tmp, keysize);
               n_exch++;
            }
          }
          if (n_exch==0) break;
          else n_exch=0;
     }
     free(tmp);
}
```

ตัวอย่างการใช้งานฟังก์ชัน bubble_sort()

```
     /* function prototypes */
     extern int int_cmp(const void *, const void *);
     extern int dbl_cmp(const void *, const void *);
     extern void bubble_sort(void *,size_t,size_t,
          int (*is_greater)(const void *, const void *));
```

```
/* array declarations and initializations */
int    i_array[10] = {6,3,2,8,4,5,1,9,10,7};
double d_array[10] = {6,3,2,8,4,5,1,9,10,7};

/* function calls */
bubble_sort(i_array, 10, sizeof(int),    int_cmp);
bubble_sort(d_array, 10, sizeof(double), dbl_cmp);
```

## 7.1.13 อาร์เรย์หลายมิติ

เท่าที่ผ่านมาเราได้เกี่ยวข้องกับการใช้อาร์เรย์มิติเดียวเท่านั้น นอกเหนือจากอาร์เรย์มิติเดียวแล้ว เราก็สามารถแจ้งใช้อาร์เรย์มีที่มิติมากกว่าหนึ่งก็ได้ เช่น เราลองนึกถึงเมตริกซ์ ในทางคณิตศาสตร์ซึ่งจัดได้เป็นอาร์เรย์สองมิติ

$$A = \begin{bmatrix} 1 & 0 & 3 \\ 2 & 1 & -1 \\ 3 & 1 & 2 \end{bmatrix}_{3\times 3} \qquad B = \begin{bmatrix} 1.0 & -2.5 & 0.0 \\ 2.5 & 0.0 & 3.0 \end{bmatrix}_{2\times 3}$$

A เป็นอาร์เรย์แบบ 3 คูณ 3 และ B เป็นอาร์เรย์แบบ 2 คูณ 3 (จำนวนแถวคูณจำนวนคอลัมน์) ถ้าต้องการแจ้งใช้ตัวแปร A และ B ที่เป็นอาร์เรย์ดังข้างต้น เราก็ทำได้ดังนี้

```
int    A[3][3];
float  B[2][3];
```

หรือ แจ้งใช้พร้อมติดตั้งค่าเริ่มต้น

```
int   A[3][3] = { {1, 0, 3},
                  {2, 1,-1},
                  {3, 1, 2} };

float B[2][3] = { { 1.0,-2.5, 0.0},
                  { 2.5, 0.0, 3.0} };
```

ซึ่งแสดงให้เห็นวิธีการแจ้งใช้พร้อมกับการติดตั้งค่าเริ่มต้นให้แก่อาร์เรย์สองมิติ

จากตัวอย่างข้างบน `B` เป็นอาร์เรย์สองมิติ ขนาด 2 คูณ 3 ซึ่งมีจำนวนแถวเท่ากับ 2 และจำนวนคอลัมน์เท่ากับ 3 การเข้าถึงข้อมูลแต่ละตัวภายในอาร์เรย์ ก็ต้องใช้ดัชนีสองตัว เช่น กำหนดโดยใช้ตัวแปร `i` และ `j` ตามลำดับ ซึ่งถ้าเราใช้หลักการเดียวกับเมตริกซ์ในทางคณิตศาสตร์แล้ว `i` ก็คือหมายเลขแถว และ `j` คือหมายเลขของคอลัมน์ เพียงแต่เราจะต้องเริ่มนับตั้งแต่ศูนย์เท่านั้นเอง

```
B[i][j]
```

โดยที่ i มีค่าตั้งแต่ 0 ถึง 1 และ j มีค่าตั้งแต่ 0 ถึง 2 หรือเราสามารถเขียนนิพจน์รูปแบบอื่นที่ให้ผลเหมือนกันได้ เช่น

```
*(B[i] + j)
(*(B+i))[j]
*(*(B+i) + j)
*(&B[0][0] + 3*i + j)
*(*B + 3*i + j)
*(B[0] + 3*i + j)
```

ซึ่งก็เป็นเรื่องของการใช้พอยน์เตอร์นั้นเอง เราจะสังเกตไว้ว่า B เป็นอาร์เรย์สองมิติและมีลักษณะคล้ายพอยน์เตอร์ที่ชี้ไปยังพอยน์เตอร์

โปรแกรมตัวอย่างต่อไปนี้ จะให้ข้อมูลเกี่ยวกับ อาร์เรย์สองมิติขนาด 3 คูณ 4 ซึ่งเราจะใช้ประกอบการพิจารณาโครงสร้างและการทำงานของอาร์เรย์สองมิตินี้

```
#include <stdio.h>

int main()
{
     int a[3][4]= {{1,2,3,4},{5,6,7,8},{9,10}};
     int i, *p = (int *)a;

     printf(" &a = %p\n", &a);
     printf("  a = %p\n",  a);
     printf(" *a = %p\n", *a);
     printf("**a = %d\n", **a);
     printf("   a[0] = %p,   a[1] = %p,   a[2] = %p\n",
               a[0], a[1], a[2]);
     printf("  &a[0] = %p,  &a[1] = %p,  &a[2] = %p\n",
               &a[0], &a[1], &a[2]);
     printf("  (a+0) = %p,  (a+1) = %p,  (a+2) = %p\n",
               (a+0), (a+1), (a+2));
     printf(" *(a+0) = %p, *(a+1) = %p, *(a+2) = %p\n",
               *(a+0), *(a+1), *(a+2));
     printf("  *a[0] = %d,  *a[1] = %d,  *a[2] = %d\n",
               *a[0], *a[1], *a[2]);

     for(i=0; i < 3*4; i++)
          printf("%d ", p[i]);
     printf("\n");
     return 0;
}
```

การแจ้งใช้อาร์เรย์

```
int a[3][4]= {{1,2,3,4},{5,6,7,8},{9,10}};
```

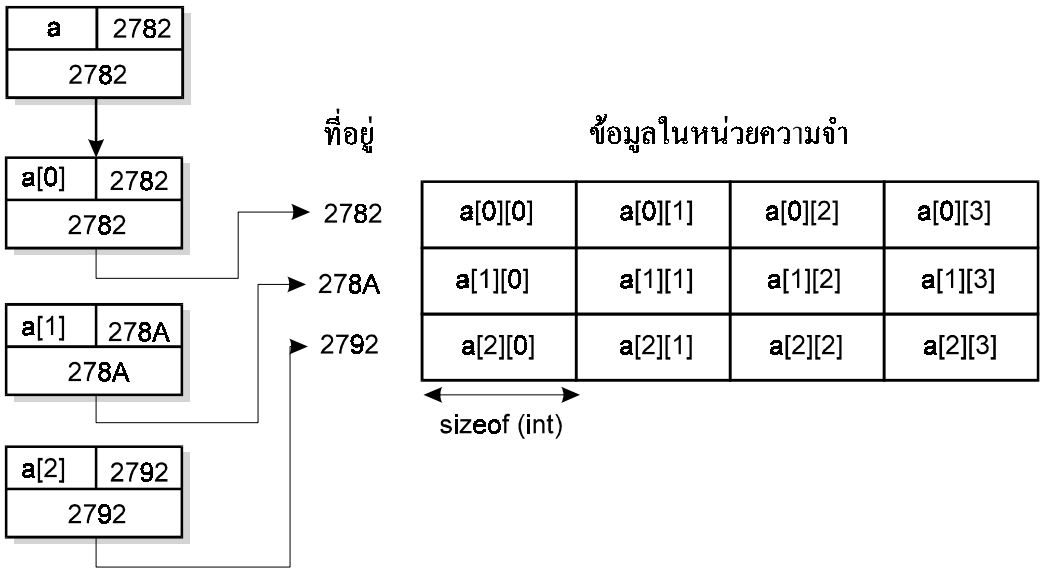
แม้ว่าเราจะกำหนดค่าเริ่มต้นให้แก่ข้อมูลแต่ละตัวของอาร์เรย์สองมิตินี้ แต่มีตัวเลขแค่ 10 จำนวนเท่านั้น ซึ่งน้อยกว่าจำนวนของข้อมูลในอาร์เรย์ แต่อย่างไรก็ตาม คอมไพล์เลอร์จะเข้าใจว่าข้อมูลสองตัวสุดท้ายจะมีค่าเท่ากับศูนย์โดยอัตโนมัติ ดังนั้นการแจ้งใช้อาร์เรย์ข้างบนจึงให้ผลเหมือนกับ

```
int a[3][4] = {{1,2,3,4},{5,6,7,8},{9,10,0,0}};
```

หรือ

```
int a[][4] = {{1,2,3,4},{5,6,7,8},{9,10,0,0}};
```

จากตัวอย่างข้างบนเราใช้พอยน์เตอร์ `p` แบบ `int` ให้ชี้ไปยังที่อยู่เริ่มต้นของ `a` และ ใช้พอยน์เตอร์นี้เข้าถึงข้อมูลแต่ละตัวของอาร์เรย์ได้โดยใช้นิพจน์ `p[i]` ซึ่งในกรณีนี้ `i` มีค่าตั้งแต่ 0 ถึง 11 เนื่องจากขั้นตอนการทำงานของวงวน `for` แสดงผลที่เป็นค่าถูกต้องของอาร์เรย์ `a` ดังนั้นเราก็สามารถสรุปได้ว่าข้อมูลแต่ละตัวของอาร์เรย์สองมิตินั้นจะเรียงติดต่อกันไปตามลำดับโดยไม่เว้นที่ว่างไว้ ซึ่งไม่แตกต่างจากการเก็บข้อมูลไว้ในอาร์เรย์มิติเดียว




แผนภาพข้างบนเป็นตัวอย่างของผลการทำงานของโปรแกรม ซึ่งแสดงความสัมพันธ์ระหว่างพอยน์เตอร์และข้อมูลภายในอาร์เรย์ เนื่องจากว่า `a` เป็นตัวแปรที่เป็นอาร์เรย์สองมิติ เมื่อเราใช้โอเปอร์เรเตอร์ `[]` กับอาร์เรย์ เราจะได้ `a[i]` โดยที่ค่าของ `i` จะมีอยู่ระหว่าง 0 ถึง 2 และตำแหน่ง `a[i]` จะทำหน้าที่เป็น

พอยน์เตอร์ที่ค่อยเก็บที่อยู่ที่เกี่ยวข้องกับข้อมูลในอาร์เรย์ และถ้าสังเกตให้ดี a[i] จะทำหน้าที่เหมือนตัวแปรที่เป็นอาร์เรย์มิติเดียว (หรือพอยน์เตอร์ที่ชี้ไปยังที่อยู่ของข้อมูลตัวแรกในอาร์เรย์มิติเดียว หรือเรียกง่ายๆ ได้ว่า พอยน์เตอร์ที่ชี้ไปยังอาร์เรย์มิติเดียว ) ดังนั้นเราสามารถกล่าวได้ว่า อาร์เรย์สองมิติคือ อาร์เรย์ของอาร์เรย์

สำหรับการเข้าถึงข้อมูลทุกตัวของอาร์เรย์สองมิติ เราใช้ดัชนีสองตัว (หมายเลขของแถวและคอลัมน์ตามลำดับ) ซึ่งหมายความว่า เราต้องใช้วงวน for ซ้อนกันสองครั้ง ตัวอย่างเช่น เราต้องการกำหนดให้ข้อมูลทุกตัวของอาร์เรย์ a เป็นศูนย์ เราก็ทำได้ดังนี้

```
for (i=0; i < 3; i++)
    for (j=0; j < 4; j++)
     a[i][j] = 0;
```

ถ้าเราต้องการใช้อาร์เรย์มากกว่าสองมิติ เราก็สามารถทำได้โดยใช้หลักการเดียวกันกับการแจ้งใช้อาร์เรย์สองมิติ เช่น การแจ้งใช้อาร์เรย์สามมิติ

```
int  a3d[2][3][4] = {
          { {1,2,3,4},{5,6,7,8},{9,10,11,12} },
          { {0,0,0,0},{0,0,0,0},{0,0,0,0}    }
     };
```

หรือ

```
int  a3d[2][3][4] = {
          { {1,2,3,4},{5,6,7,8},{9,10,11,12} },
          {    {0}   ,   {0}   ,     {0}     }
     };
```

อาร์เรย์ที่มากกว่าสามมิติเราจะไม่ค่อยได้พบเห็นบ่อยนักในโปรแกรมภาษาซี

ตัวอย่างการผ่านตัวแปรที่เป็นอาร์เรย์สองมิติให้ฟังก์ชัน เช่น

```
func (int a[3][4], int n_rows);
func (int a[][4],  int n_rows);
func (int (*a)[4], int n_rows);
```

แบบใดแบบหนึ่ง ภายในฟังก์ชันเราจะไม่ทราบจำนวนของแถวทั้งหมด ดังนั้นเราจึงต้องผ่านพารามิเตอร์ที่ให้ข้อมูลเกี่ยวกับจำนวนแถวของอาร์เรย์สองมิติด้วย สำหรับจำนวนของคอลัมน์เราสามารถใช้โอเปอร์เรเตอร์ sizeof กับ a[0] ซึ่งได้ค่าในหน่วยไบต์ ดังนั้นนิพจน์

```
sizeof(a[0]) / sizeof (a[0][0])
```

จึงให้ค่าที่เท่ากับจำนวนของคอลัมน์ในแต่ละแถว (ทุกแถวมีจำนวนคอลัมน์เท่ากันหมด) อันที่จริงแล้วจำนวนของคอลัมน์จะถูกกำหนดไว้อย่างแน่นอนแล้ว จากตัวอย่างเราได้กำหนดจำนวนของคอลัมน์ไว้เท่ากับ 4 ซึ่งเป็นสิ่งสำคัญและจำเป็น เพราะเมื่อเราใช้นิพจน์ `a[i][j]` โปรแกรมจำเป็นต้องทราบจำนวนของคอลัมน์ก่อนเพื่อที่จะใช้ในการกำหนดตำแหน่งของข้อมูลที่ต้องการในหน่วยความจำได้อย่างถูกต้อง

ถ้าเราสร้างฟังก์ชันที่มีรูปแบบต่อไปนี้

```
func (int a[3][], int n_rows);
func (int  a[][], int n_rows);
func (int (*a)[], int n_rows);
```

แบบใดแบบหนึ่ง จะถือว่าผิด เพราะโปรแกรมไม่สามารถหาจำนวนของคอลัมน์ในแต่ละแถวได้

## 7.1.14 การจัดสรรหน่วยความจำสำหรับอาร์เรย์สองมิติ

เช่นเดียวกับการจัดสรรหน่วยความจำสำหรับสร้างอาร์เรย์มิติเดียว เราก็สามารถจองบริเวณหน่วยความจำสำหรับเก็บข้อมูลในรูปแบบของอาร์เรย์สองมิติได้ ข้อดีก็คือเราสามารถกำหนดจำนวนแถวและจำนวนของคอลัมน์ให้เหมาะกับจำนวนของข้อมูลที่เรามีอยู่ ซึ่งในเวลาที่โปรแกรมทำงานเราอาจจะต้องการเก็บข้อมูลต่างๆไว้ในอาร์เรย์ที่มีขนาดที่แตกต่างกันออกไป เราลองมาพิจารณาโปรแกรมตัวอย่างข้างล่างนี้

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

int main()
{
     int **b;
     int i, j, m = 3, n = 4;

     b = (int **)malloc(m * sizeof(int *));
     for (i=0; i < m ;i++)
          b[i] = (int *)malloc(n * sizeof(int));

     for(i=0; i < m; i++)
        for(j=0;j < n; j++) {
            b[i][j] = i*n + j;
            printf(" %d ", b[i][j]);
        }
     printf("\n");

     for(i=0; i < m; i++)
```

```
      free(b[i]);
    free(b);

    return 0;
}
```

ขนาดของอาร์เรย์จะกำหนดโดย ตัวแปร m และ n ซึ่งเป็นจำนวนแถวและจำนวนคอลัมน์ตามลำดับ

เรากำหนดให้ b เป็นพอยน์เตอร์ที่ชี้ไปยังพอยน์เตอร์ (คล้ายกับตัวแปรที่เป็นอาร์เรย์สองมิติ) ก่อนอื่นเราจะต้องจัดหาหน่วยความจำสำหรับสร้างอาร์เรย์มิติเดียวที่เก็บพอยน์เตอร์ หลายๆตัวได้

```
b = (int **)malloc(m * sizeof(int *));
```

ในกรณีนี้ เราจะสร้างอาร์เรย์มิติเดียวที่เก็บพอยน์เตอร์แบบ int ได้ทั้งหมด m ตัว และนิพจน์ b[i] ก็จะทำหน้าที่เป็นพอยน์เตอร์ที่ชี้ไปยังข้อมูลแบบ int หรือใช้ให้ทำหน้าที่ชี้ไปยังที่อยู่ของอาร์เรย์มิติเดียวที่เก็บข้อมูลแบบ int ก็ได้

ขั้นต่อไปคือการจัดหาหน่วยความจำสำหรับอาร์เรย์มิติเดียวที่มีความจุเท่ากับ n ซึ่งใช้เก็บข้อมูลแบบ int

```
for (i=0; i < m ;i++)
     b[i] = (int *)malloc(n * sizeof(int));
```

ถ้าเราคิดว่า แต่ละแถวของอาร์เรย์สองมิติคืออาร์เรย์มิติเดียว เราก็ต้องจัดหาหน่วยความจำทั้งหมด m แถว และ ให้ b[i] เก็บที่อยู่เริ่มต้นของอาร์เรย์มิติเดียวที่ถูกสร้างขึ้นสำหรับแต่ละแถว ดังนั้นเราก็ได้จัดหาหน่วยความจำสำหรับข้อมูลแบบ int จำนวน m*n ตัว ถ้าเราต้องการจะเข้าถึงข้อมูลแต่ละตัวเราก็สามารถใช้นิพจน์ b[i][j] ได้คล้ายกับว่า b เป็นอาร์เรย์สองมิติ ( โปรดสังเกตว่า bไม่ใช่อาร์เรย์สองมิติ แต่เป็นพอยน์เตอร์ที่ชี้ไปยังพอยน์เตอร์แบบ int)

ถ้าเราไม่ต้องการใช้หน่วยความจำที่ทำหน้าเก็บข้อมูลในรูปแบบของอาร์เรย์สองมิตินี้แล้ว เราก็ใช้คำสั่ง free() โดยขั้นแรกจะต้องคืนหน่วยความจำของอาร์เรย์สำหรับ m แถว ก่อน และจึงคืนหน่วยความจำสำหรับพอยน์เตอร์ b ตามลำดับ

```
for(i=0; i < m; i++)
  free(b[i]);
free(b);
```

ในการผ่านพอยน์เตอร์ที่ชี้ไปยังบริเวณหน่วยความจำสำหรับทำหน้าที่เก็บข้อมูลในลักษณะที่เป็นอาร์เรย์สองมิตินั้น เราจะต้องกำหนดรูปแบบดังนี้

```
func (int **b,  int n_rows, int n_columns);
func (int *b[], int n_rows, int n_columns);
```

แบบใดแบบหนึ่ง ซึ่งจะแตกต่างจากการผ่านตัวแปรที่เป็นอาร์เรย์สองมิติให้แก่ฟังก์ชัน ภายในฟังก์ชัน เราสามารถใช้นิพจน์ b[i][j] ได้ในการเข้าถึงข้อมูลแต่ละตัวตามดัชนี i ของแถวและ j ของคอลัมน์ เนื่องจากเราไม่สามารถคำนวณหาจำนวนของแถวและคอลัมน์ได้เมื่ออยู่ภายในฟังก์ชัน เราจำเป็นต้องผ่านพารา-มิเตอร์อีกสองตัวเพื่อจุดประสงค์นี้ คือ n_rows และ n_columns จะใช้ในการกำหนดจำนวนแถวและคอลัมน์ของ b ตามลำดับ เช่น ถ้าเราต้องการเข้าถึงและอ่านค่าข้อมูลแต่ละตัวของ b เราก็ทำได้ดังนี้

```
func (int **b, int n_rows, int n_columns)
{
     int i,j;

     for(i=0; i < n_rows; i++)
     {
         for(j=0; j < n_columns; j++)
             printf("%5d", b[i][j]);

         printf("\n");
     }
     return 0;
}
```

## 7.1.15 ตัวอย่างการสร้างฟังก์ชันที่ใช้สำหรับเมตริกซ์

ในการเขียนโปรแกรมที่ใช้ในงานทางด้านคณิตศาสตร์ การจัดเก็บตัวเลขในรูปของเมตริกซ์หรืออาร์เรย์สองมิติถือว่ามีบทบาทสำคัญ ดังนั้นเพื่อให้มองเห็นวิธีการเขียนโปรแกรมสำหรับเมตริกซ์ เราจะลองพิจารณาตัวอย่างของฟังก์ชันที่ใช้ในสร้างและจัดเก็บข้อมูลในรูปแบบของอาร์เรย์สองมิติดังนี้

```
typedef double ** Matrix;
```

เราใช้แบบข้อมูล double ในการเก็บข้อมูลแต่ละตัวของเมตริกซ์ ที่มีขนาด m แถว และ n คอลัมน์ เนื่องจากว่าเราจะใช้พอยน์เตอร์ที่ชี้ไปยังพอยน์เตอร์แบบ double ในการจองหน่วยความจำสำหรับอาร์เรย์สองมิติเพื่อความสะดวกเราจะนิยามแบบของพอยน์เตอร์ขึ้นใหม่ชื่อว่า Matrix

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

```
typedef double ** Matrix;


Matrix createMatrix (int rows, int columns)
{
     Matrix mat;
     size_t i, nbytes, m = rows, n = columns;

     mat = (Matrix)malloc( m * sizeof(*mat) );
     if (!mat){
          printf("Cannot allocate memory!\n");
          return NULL;
     }

     nbytes = n * sizeof(**mat);
     for (i=0; i < m ;i++)
     {
          mat[i] = malloc( nbytes );
          if (!mat[i])
          {
               size_t j;

               for (j=0; j < i ; j++)
                    free(mat[j]);
               printf("Cannot allocate memory!\n");
               return NULL;
          }

          memset(mat[i], 0, nbytes);
     }

     return mat;
}
```

ฟังก์ชัน `createMatrix()` มีหน้าที่จัดสรรหน่วยความจำในการสร้างเมตริกซ์ ถ้าสามารถจัดหาหน่วยความจำตามขนาดที่ต้องการได้ ฟังก์ชันก็จะให้ที่อยู่เริ่มต้นของบล็อกหน่วยความจำดังกล่าวกลับคืนและฟังก์ชันก็จะกำหนดให้ข้อมูลทุกตัวมีค่าเป็นศูนย์ โดยใช้คำสั่ง `memset()` ซึ่งนิยามไว้ใน `<string.h>` ถ้าหน่วยความจำไม่พอ ฟังก์ชันก็จะให้พอยน์เตอร์ศูนย์กลับคืน เช่น `m` และ `n` มีค่าเท่ากับ 3 เมตริกซ์ที่ได้คือ

$$\begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{bmatrix}_{3\times 3}$$

ฟังก์ชันต่อไปคือ `deleteMatrix()` ใช้ในการทำลายตัวแปรแบบ `Matrix` ซึ่งหมายถึงการคืนหน่วยความจำที่ได้จองไว้ให้ว่างสำหรับใช้ในคราวต่อไปหรือเพื่อจุดประสงค์อื่น โดยใช้คำสั่ง `free()`

```
void deleteMatrix(Matrix mat, int rows, int columns)
{
     size_t i, m = rows;

     if (!mat) return ;

     for (i=0; i < m ;i++)
          free(mat[i]);
     free(mat);
}
```

และเพื่อที่จะแสดงค่าของข้อมูลแต่ละตัวในเมตริกซ์ เราก็เขียนฟังก์ชัน `printMatrix()` ขึ้น

```
void printMatrix(Matrix mat, int rows, int columns)
{
     size_t i, j, m = rows, n = columns;

     if (!mat) return ;

     for (i=0; i < m ;i++)
     {
           for (j=0; j < n ;j++)
                printf("%.3e ", mat[i][j]);
           printf("\n");
     }
     printf("-----------------------------\n");
}
```

ซึ่งจะพิมพ์ข้อมูลออกทางจอภาพ และฟังก์ชันต่อไปเราจะใช้ในการกำหนดค่าของเมตริกซ์ให้อยู่ในรูปแบบของเมตริกซ์ที่มีค่าในแนวทแยงเท่ากับค่าที่กำหนดโดยพารามิเตอร์ `d` และนอกจากนั้นจะมีค่าเป็นศูนย์

```
void diagMatrix
     (Matrix mat, int rows, int columns, double d)
{
     size_t i, nbytes, m = rows, n = columns;

     if (!mat) return ;

     assert(rows==columns);

     nbytes = n * sizeof(**mat);
     for (i=0; i < m; i++)
     {
           memset(mat[i], 0, nbytes);
           mat[i][i] = d;
     }
}
```

เช่น ถ้า d มีค่าเป็น 1.0 เราก็จะได้เมตริกซ์ ในรูปแบบดังต่อไปนี้

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix}_{3\times3}$$

หรือเราจะเรียกใช้ฟังก์ชัน unitMatrix() ก็ได้

```
void unitMatrix(Matrix mat, int rows, int columns)
{
    diagMatrix(mat, rows, columns, 1.0);
}
```

ฟังก์ชันเหล่านี้เป็นเพียงตัวอย่างพื้นฐานเท่านั้นเอง ถ้าเราต้องการบวกลบระหว่างเมตริกซ์ หรือคำนวณค่าอื่นๆเกี่ยวกับเมตริกซ์ เราก็สามารถเขียนฟังก์ชันขึ้นใช้เพิ่มเติมได้ ซึ่งต้องใช้ความรู้พื้นฐานทางคณิตศาสตร์ด้วย

## 7.1.16 อาร์เรย์ของพอยน์เตอร์

นอกจากที่อาร์เรย์จะเก็บข้อมูลแบบพื้นฐานธรรมดาแล้ว เราก็สามารถกำหนดให้อาร์เรย์เก็บข้อมูลที่เป็นที่อยู่ซึ่งหมายถึงค่าของพอยน์เตอร์แบบเดียวกันหลายๆตัว ซึ่งชี้ไปยังที่อยู่ต่างๆกัน ตัวอย่างเช่น

```
char  *ap[5] = {NULL};
```

หมายถึง การแจ้งใช้อาร์เรย์ ap ที่สามารถเก็บพอยน์เตอร์แบบ char ได้ทั้งหมด 5 ตัว

```
char *ap[5] = {NULL};

ap[0] = "Gauss";
ap[1] = "Cauchy";
ap[2] = "Euler";
ap[3] = "Fourier";
ap[4] = "Laplace";
```

ในกรณีนี้ เราใช้อาร์เรย์ ap ในการเก็บข้อความที่แตกต่างกันออกไปจำนวนห้าข้อความ

หรือถ้าเราต้องการจะแจ้งใช้อาร์เรย์ที่เก็บพอยน์เตอร์ที่ชี้ไปยังฟังก์ชัน เราก็สามารถทำได้ เช่น

```
void (*methods[3])(int);
```

ตัวแปร methods เป็นอาร์เรย์ที่สามารถเก็บพอยน์เตอร์จำนวน 3 ตัวที่ชี้ไปยังฟังก์ชันที่มีพารามิเตอร์แบบ int และไม่ให้ค่าใดๆกลับคืน ตัวอย่างการใช้งาน เช่น

```
extern  void f1(int data);
extern  void f2(int data);
extern  void f3(int data);
void (*methods[3])(int) = {NULL};
int x = 10;

methods[0] = f1;
methods[1] = f2;
methods[2] = f3;
```

เมื่อเราต้องการจะเรียกฟังก์ชันใดใช้งาน ก็ทำได้ดังนี้ เช่น

```
methods[0](x);
methods[1](x);
methods[2](x);
```

ซึ่งจะให้ผลเหมือนกับการเรียกใช้ฟังก์ชัน f1() f2() และ f3()โดยตรง

```
f1(x);
f2(x);
f3(x);
```

## 7.1.17 พอยน์เตอร์ที่ชี้ไปยังอาร์เรย์

พอยน์เตอร์ที่ชี้ไปยังอาร์เรย์หรือเรียกสั้นๆว่าพอยน์เตอร์แบบอาร์เรย์นั้นจะแตกต่างจากอาร์เรย์ที่เก็บพอยน์เตอร์หลายๆตัว พอยน์เตอร์แบบอาร์เรย์ คือพอยน์เตอร์ที่เก็บที่อยู่ของอาร์เรย์ ซึ่งจะต้องมีการกำหนดขนาดของอาร์เรย์ไว้อย่างแน่นอน ตัวอย่างการแจ้งใช้พอยน์เตอร์ที่ชี้ไปยังอาร์เรย์ เช่น

```
int  (*pa)[10];
```

พอยน์เตอร์ pa จะใช้ในการชี้ไปยังอาร์เรย์แบบ int ที่มีความจุเท่ากับ 10 หน่วย

```
int  a[10] = {0,1,2,3,4,5,6,7,8,9};
int  (*pa)[10] = &a;
```

ในตัวอย่างนี้ pa จะชี้ไปยังอาร์เรย์ a แบบ int ขนาด 10 หน่วย ระยะห่างระหว่างค่าของ pa และ (pa+1) จะมีค่าเท่ากับ 10 คูณด้วยขนาดของข้อมูลแบบ int ซึ่งจะมีค่าเท่ากับ 20 (ไบต์)

ตัวอย่างการใช้ พอยน์เตอร์แบบอาร์เรย์

```
#include <stdio.h>

int main()
{
     int a2d[3][3] = {{1,2,3},{4,5,6},{7,8,9}};
     int (*pa)[3];
     int i,j;

     pa = &a2d[0];
     for (i=0; i < 3; i++)
     {
          for(j=0; j < 3; j++)
               printf("%d ", (*pa)[j]);
          printf("\n");
          pa++;
     }
     return 0;
}
```

เราจะเห็นได้ว่า นิพจน์ `(*pa)` จะมีลักษณะเป็นอาร์เรย์มิติเดียว และการเข้าถึงข้อมูลแต่ละตัวของอาร์เรย์นี้ก็อาศัยนิพจน์ `(*pa)[i]` โดยมี `i` เป็นดัชนีกำหนดหมายเลขของข้อมูลแต่ละตัว

## 7.1.18 การใช้ const กับอาร์เรย์

ถ้าเราต้องการแจ้งใช้อาร์เรย์พร้อมกำหนดค่าเริ่มต้นให้แก่ข้อมูลแต่ละตัว และนอกจากนั้นเราก็ไม่ต้องการให้ขั้นตอนใดแก้ไขหรือเปลี่ยนแปลงข้อมูลที่เก็บไว้ในอาร์เรย์เราก็สามารถใช้ `const` ในการกำหนดคุณสมบัติข้อนี้ได้ เช่น

```
const char vowelTable[5] = {'a','e','i','o','u'};
const char days[][4] = { "Mon","Tue","Wed","Thu",
                         "Fri","Sat","Sun" };
```

และหลังจากการแจ้งใช้แล้ว เราไม่สามารถเปลี่ยนแปลงค่าของข้อมูลในอาร์เรย์ได้อีก

## 7.2 สายอักขระ (String)

สายอักขระจัดเป็นโครงสร้างของข้อมูลแบบอาร์เรย์มิติเดียว และเป็นอาร์เรย์ของข้อมูลแบบ char หรือ unsigned char เท่านั้น และที่สำคัญ สายอักขระที่ถูกต้องจะต้องมีตัวอักขระ '\0' เป็นตัวกำหนดขนาดของอาร์เรย์ สายอักขระประกอบด้วยข้อมูลขนาดหนึ่งไบต์เรียงต่อกันไปในอาร์เรย์ เมื่อใดที่พบว่ามีตัวอักขระ '\0' อยู่ในอาร์เรย์ ก็เป็นอันว่าสิ้นสุดสายอักขระที่เก็บอยู่ภายในอาร์เรย์นี้

ดังนั้นสำหรับความยาวของสายอักขระจะเริ่มนับตั้งแต่ข้อมูลตัวแรกไปเรื่อยๆ จนถึงข้อมูล ที่มีค่าเท่ากับศูนย์ เราจะไม่ถือว่าข้อมูลที่มีค่าเท่ากับศูนย์นี้เป็นส่วนหนึ่งของสายอักขระ เพียงแต่มีเงื่อนไขว่าสายอักขระใดๆจะต้องมีข้อมูลอย่างน้อยหนึ่งตัวที่มีค่าเท่ากับศูนย์จบท้าย (Null-terminated) และที่สำคัญความยาวของสายอักขระจะต้องมีค่าน้อยกว่าขนาดของอาร์เรย์ที่สายอักขระกำลังใช้หน่วยความจำ ของอาร์เรย์ เช่น ถ้าเราแจ้งใช้อาร์เรย์แบบ char ขนาด 10 ตัวอักขระ

```
char string[10];
```

ดังนั้นเราสามารถใช้ตัวแปร string ที่เป็นอาร์เรย์นี้ในการเก็บสายอักขระที่มีความยาวไม่เกิน 9 ตัวอักขระ (อย่างน้อยหนึ่งที่ในอาร์เรย์จะต้องใช้เก็บ '\0') เช่น ถ้าเราต้องการเก็บข้อความสั้นๆ ไว้ในอาร์เรย์ เราก็ทำได้ดังนี้

```
char string[10];

string[0] = 'H';
string[1] = 'e';
string[2] = 'l';
string[3] = 'l';
string[4] = 'o';
string[5] = '\0';
```

จากตัวอย่างเราได้กำหนดค่าของข้อมูลแต่ละตัวขนาดหนึ่งไบต์ในอาร์เรย์ให้มีค่าต่างๆ ซึ่งก็คือ ตัวอักขระในข้อความว่า "Hello" และใช้เนื้อที่ของอาร์เรย์เพียง 6 ที่เท่านั้น (แต่สายอักขระนี้มีความยาวเท่ากับ 5 ตัวอักขระ) จากความจุทั้งหมด 10 ที่

สำหรับความสำคัญของการจบท้ายสายอักขระด้วยศูนย์ หรือ '\0' เราจะเห็นได้จากตัวอย่างต่อไปนี้

```
#include <stdio.h>

int main()
{
```

```
    char string[10] = {
          'X','X','X','X','X',
          'X','X','X','X','\0'
         };

    string[0] = 'H';
    string[1] = 'e';
    string[2] = 'l';
    string[3] = 'l';
    string[4] = 'o';
    printf ("%s\n", string);

    string[5] = '\0';
    printf ("After setting string[5] to zero:");
    printf (" %s\n", string);

    string[1] = '\0';
    printf ("After setting string[1] to zero:");
    printf (" %s\n", string);

    return 0;
}
```

ผลการทำงานของโปรแกรมคือ

```
    HelloXXXX
    After setting string[5] to zero: Hello
    After setting string[1] to zero: H
```

จะเห็นได้ว่า ฟังก์ชัน `printf()` จะพิมพ์ตัวอักขระแต่ละตัวของข้อความ และเมื่อพบว่า ตัวใดมีค่าเท่ากับศูนย์ก็จะหยุดพิมพ์ข้อความนั้น เพราะตัวขระ `'\0'`ทำหน้าที่เป็นตัวกำหนดขอบเขตของสายอักขระในหน่วยความจำของอาร์เรย์

เราได้เรียนรู้ไปในบทแรกๆของหนังสือแล้วว่า ข้อความในภาษาซีจะเริ่มต้นและจบลงด้วยสัญลักษณ์ " และจัดเป็นนิพจน์ค่าคงที่ แม้ว่าข้อความในภาษาซีจะเป็นนิพจน์ค่าคงที่ก็ตาม แต่มีข้อสังเกตในเรื่องของหน่วยความจำของข้อความที่ใช้ เช่น ถ้าเราเขียนว่า

```
    "Hello World!"
```

ก็เป็นนิพจน์ที่เป็นข้อความหรือสายอักขระ ซึ่งถ้าเราใช้ในโปรแกรมข้อความนี้ก็จะต้องใช้พื้นที่ส่วนหนึ่งของหน่วยความจำหลัก ดังนั้นเราสามารถหาที่อยู่ของข้อความในหน่วยความจำได้ โดยใช้โอเปอร์เรเตอร์ &

```
    &"Hello World!"
```

แต่โปรดจำไว้ว่า ข้อความหลายๆข้อความในภาษาซีที่เหมือนกัน ใช่ว่าจะมีที่อยู่ในหน่วยความจำเดียวกัน และเราสามารถสังเกตได้จากตัวอย่างต่อไปนี้

```
#include <stdio.h>

int main()
{
     char *s1 = "Hello World!";
     char *s2 = "Hello World!";

     printf ("s1 points to the address : %p\n", s1);
     printf ("s2 points to the address : %p\n", s2);
     printf ("address of \"Hello World!\" : %p\n",
               &"Hello World!" );
     return 0;
}
```

ผลของโปรแกรมคือ

```
     s1 points to the address : 007E
     s1 points to the address : 008B
     address of "Hello World!" : 00F6
```

ทั้งๆที่เราได้กำหนดให้ s1 และ s2 เป็นพอยน์เตอร์ที่ชี้ไปยังข้อความที่เหมือนกัน คือ

```
     "Hello World!"
```

ถ้าพอยน์เตอร์ทั้งสองชี้ไปยังแหล่งข้อมูลเดียวกัน แล้วค่าของ s1 และ s2 จะต้องเท่ากัน แต่ผลจากโปรแกรมแสดงให้เห็นว่า s1 และ s2 ชี้ไปยังแหล่งข้อมูลที่อยู่ในที่แตกต่างกัน เพียงแต่แหล่งข้อมูลทั้งสองเก็บข้อความที่เหมือนกัน นอกจากนี้เราจะเห็นได้ว่าเรามีนิพจน์สามนิพจน์เป็นข้อความที่เหมือนกันคือ "Hello World!" แต่เก็บไว้ในที่อยู่ที่แตกต่างกันสามที่

โปรดสังเกตว่า เราไม่สามารถใช้โอเปอร์เรเตอร์ & ในการหาที่อยู่ของนิพจน์ที่เป็นค่าคงที่ของข้อมูลแบบพื้นฐานอื่นๆได้

## 7.2.1 การแจ้งใช้ตัวแปรสายอักขระ

เนื่องจากสายอักขระก็เป็นอาร์เรย์ชนิดขึ้น ถ้าเราต้องการแจ้งใช้ตัวแปรที่ใช้เก็บข้อความใดๆ เราก็ทำได้เหมือนกับการแจ้งใช้อาร์เรย์มิติเดียว ตัวอย่างการแจ้งใช้ ตัวแปรสายอักขระ เช่น

```
char  str1[]   = {'A','B','C','D',0};
char  str2[10] = {'A','B','C','D',0};
char  str3[]   = "ABCD";
char *str4     = "ABCD";
char *str5     = "ABCD\0";
```

และเราจะได้ตัวแปร `str1` ที่เป็นอาร์เรย์ ขนาด 5 หน่วย `str2` เป็นอาร์เรย์ขนาด 10 หน่วยแบบ `char` ตัวแปร `str3` ก็เป็นอาร์เรย์เช่นกัน มีขนาดเท่ากับ 5 ไบต์ และเราก็กำหนดข้อมูลเริ่มต้นที่เก็บไว้ในอาร์เรย์ด้วย (เราจะต้องนับ `'\0'`ด้วย) ดังนั้น `str1` และ `str3` จึงเหมือนกันทั้งขนาดและข้อความที่เก็บอยู่ในอาร์เรย์ ในขณะที่ `str4` และ `str5` เป็นพอยน์เตอร์แบบ `char` ที่ชี้ไปยังข้อความที่เป็นนิพจน์ค่าคงที่ และนิพจน์ทั้งสองมีขนาดของหน่วยความจำเท่ากับ 5 และ 6 ไบต์ตามลำดับ

ตัวอย่างการแจ้งใช้ตัวแปรที่ผิด เช่น

```
char  str6[2] = {'A','B','C','D',0};
char  str7[4] = "ABCD";
char  str8[]  = "";
```

เพราะอาร์เรย์ที่เราใช้จะต้องสามารถเก็บข้อความที่เป็นค่าเริ่มต้นได้ครบทุกตัวอักษรและอย่างถูกต้อง แต่ตัวอย่างการแจ้งใช้ทั้งสองไม่เป็นไปตามเงื่อนไขนี้ สำหรับการแจ้งใช้ตัวแปร `str8` แม้ว่าจะถูกหลักไวยากรณ์แต่ไม่มีประโยชน์เพราะตัวแปรนี้เป็นอาร์เรย์ขนาดหนึ่งไบต์เท่านั้น และใช้เก็บข้อความว่างเปล่าซึ่งหมายความว่า ตัวอักขระตัวแรก(และตัวเดียวเท่านั้น)ในอาร์เรย์มีค่าเป็นศูนย์ และเราไม่สามารถใช้เก็บค่าอื่นๆได้ถ้าเราจะใช้อาร์เรย์นี้ในการเก็บข้อความ เพราะมีกฎการใช้สายอักขระอยู่ว่า สายอักขระใดๆจะต้องมีตัวอักขระอย่างน้อยหนึ่งตัวที่มีค่าเท่ากับศูนย์ ในกรณีอาร์เรย์มีหน่วยความจำเพียงหนึ่งไบต์เท่านั้น ถ้าเราต้องการเก็บสายอักขระก็จะเป็นสายอักขระว่างเปล่าเท่านั้น

แม้ว่า เราจะใช้พอยน์เตอร์หรืออาร์เรย์ได้ในการเก็บข้อความใดๆก็ตาม แต่ก็มีความแตกต่างระหว่างทั้งสองสิ่ง เพราะถ้าเราใช้ ตัวแปรอาร์เรย์ ซึ่งทำหน้าที่คล้ายพอยน์เตอร์ เราไม่สามารถกำหนดให้ชี้ไปยังที่อยู่ของข้อความอื่นได้ ถ้าเราใช้พอยน์เตอร์เราสามารถกำหนดให้ชี้ไปยังที่อยู่ของข้อความใดๆก็ได้ที่เราต้องการแต่อย่างไรก็ตามเราจะต้องคำนึงถึงที่อยู่และหน่วยความจำที่ใช้เก็บข้อความด้วย ตัวอย่างเช่น

```
#include <stdio.h>

int main()
{
     char storage[255] = {0};
     char *cpointer = "XXXXX";

     printf("Please enter any message : ");
     scanf("%s", cpointer);
     printf("Input string = %s\n", cpointer);

     printf("Please enter any message : ");
     scanf("%s", storage);
     printf("Input string = %s\n", storage);

     return 0;
}
```

เราได้แจ้งใช้ ตัวแปรพอยน์เตอร์ cpointer และกำหนดให้ชี้ไปยังบริเวณหน่วยความจำที่เก็บข้อความ "XXXXX" และเมื่อเราผ่านพอยน์เตอร์นี้ ให้เป็นพารามิเตอร์ของฟังก์ชัน scanf() เพื่อใช้เก็บข้อความที่อ่านได้จากแป้นพิมพ์ แต่เนื่องจากว่า มีหน่วยความจำเพียง 5 ไบต์เท่านั้น ถ้าเราป้อนข้อความที่ยาวเกินไปก็จะมีปัญหาได้ เมื่อเปรียบเทียบกับตัวแปร storage ที่เป็นอาร์เรย์ขนาด 255 ตัวอักขระ ถ้าเราผ่านตัวแปรนี้ให้ฟังก์ชัน scanf() เราก็สามารถป้อนข้อความที่มีความยาวหลายตัวอักขระได้ ถ้าเราต้องการป้องกันไม่ให้ฟังก์ชัน scanf()อ่านข้อมูลจากแป้นพิมพ์และเก็บข้อความที่ยาวเกินไป เราก็สามารถกำหนดจำนวนของตัวอักขระที่เราต้องการเก็บได้ เช่น

```
     printf("Please enter any message(max. 5 chars): ");
     scanf("%5s", cpointer);
     printf("Input string = %s\n", cpointer);
```

ในกรณีนี้ฟังก์ชันจะเก็บข้อมูลที่เป็นตัวอักขระลงในหน่วยความจำที่พอยน์เตอร์ cpointer กำลังชี้ไป เป็นจำนวนสูงสุด 5 ตัวเท่านั้น

## 7.2.2 การกำหนดค่าของตัวแปรสายอักขระ

การกำหนดค่าของอาร์เรย์ให้เป็นข้อความนั้น เราไม่สามารถใช้โอเปอร์เรเตอร์ = ได้โดยตรง เช่น ถ้า a เป็นอาร์เรย์ที่ใช้เก็บข้อความ และข้อมูลแต่ละตัวมีค่าเริ่มต้นเป็นศูนย์ และเราต้องการกำหนดให้เก็บข้อความ เช่น "Hello World!" เราก็ไม่สามารถเขียนคำสั่งต่อไปนี้ได้

```
a = "Hello World!";
```

เพราะเราไม่สามารถเปลี่ยนแปลงที่อยู่ของตัวแปรที่เป็นอาร์เรย์ได้ (อาร์เรย์ ไม่ใช่พอยน์เตอร์) ถ้าเราต้องการจะเก็บข้อความใดๆลงในอาร์เรย์ เราก็ต้องใช้วิธีการเรียกใช้ฟังก์ชัน เช่น ฟังก์ชันมาตราฐาน `strcpy()` ที่นิยามไว้ใน `<string.h>` ตัวอย่างการใช้งาน เช่น

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

int main()
{
     char a[20] = "XXXXXXXXXX";

     printf("%s\n", a);

     strcpy(a, "ABCD");
     printf("%s\n", a);

     strcpy(a, "0123456789");
     printf("%s\n", a);

     return 0;
}
```

ฟังก์ชัน `strcpy()` จะทำหน้าที่คัดลอกข้อความต้นฉบับใดๆที่เราต้องการไปยังอาร์เรย์ `a` โดยข้อความที่มีอยู่เดิมจะถูกเขียนทับโดยข้อความใหม่ และขนาดของข้อความใน `a` จะเป็นขนาดของข้อความใหม่ (โปรดสังเกตว่า ขนาดของอาร์เรย์ `a` ยังคงเท่าเดิม)

ความแตกต่างระหว่างตัวอักขระกับสายอักขระที่มีเพียงตัวอักขระตัวเดียว จะสังเกตได้จากตัวอย่างต่อไปนี้

```
'A'
"A"
```

ในบรรทัดแรก เป็นนิพจน์ที่เป็นตัวอักขระ ในขณะที่ในบรรทัดที่สองเป็นสายอักขระขนาดหนึ่งตัวอักษร ถ้าสมมุติว่า `a` เป็นอาร์เรย์แบบ `char` ขนาด 20ไบต์ และถ้าเราเขียนว่า

```
a[0] = 'A';
a[0] = "A";
```

คำสั่งในบรรทัดแรกถือว่าถูกต้อง แต่ในบรรทัดที่สองจะผิดหลักไวยากรณ์ เพราะ ถ้าเราอ่านค่านิพจน์ที่เป็นสายอักขระ เราก็จะได้หมายเลขที่อยู่ของสายอักขระนี้ในหน่วยความจำ ดังนั้นคำสั่งในบรรทัดที่สองจึงไม่ถูกต้อง

## 7.2.3 การหาความยาวของสายอักขระ

การหาความยาวของสายอักขระใดๆ เราจะใช้ฟังก์ชันมาตราฐาน `strlen()` ที่นิยามไว้ในแฟ้มส่วนหัว `<string.h>` การใช้งานฟังก์ชันนี้ ก็เพียงผ่านที่อยู่ของสายอักขระ (ในรูปของพอยน์เตอร์) เป็นพารามิ-เตอร์ของฟังก์ชัน และเมื่อฟังก์ชันทำงานจบก็จะได้ขนาดของสายอักขระซึ่งเป็นจำนวนของตัวอักขระที่มีอยู่ ทั้งหมดก่อนตัวอักขระที่มีค่าเท่ากับศูนย์ ลองพิจารณาตัวอย่างต่อไปนี้

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

int main()
{

     char string[10] = {
           'X','X','X','X','X',
           'X','X','X','X','\0'
          };

     string[0] = 'H';
     string[1] = 'e';
     string[2] = 'l';
     string[3] = 'l';
     string[4] = 'o';
     printf ("%s\n", string);
     printf ("size of string = %d\n",
            strlen(string));

     string[5] = '\0';
     printf ("After setting string[5] to zero:");
     printf (" %s\n", string);
     printf ("size of string = %d\n",
            strlen(string));

     string[1] = '\0';
     printf ("After setting string[1] to zero:");
     printf (" %s\n", string);
     printf ("size of string = %d\n",
            strlen(string));
```

```
    return 0;
}
```

ผลการทำงานของโปรแกรมจะเป็นดังนี้

```
HelloXXXX
size of string = 9
After setting string[5] to zero: Hello
size of string = 5
After setting string[1] to zero: H
size of string = 1
```

การใช้งานตัวแปรที่เป็นสายอักขระก็จะเกี่ยวข้องกับการประมวลผลตัวอักขระแต่ละตัวหรือหลายๆตัว ในการใช้งานสายอักขระนั้น เราจะต้องคำนึงถึงเสมอว่าเราจะไม่ใช้สายอักขระเก็บข้อความที่มีความยาวเกินเนื้อที่ของหน่วยความจำที่ได้จัดสรรไว้ เช่น ถ้าเราใช้อาร์เรย์ขนาด 10 หน่วย ในการเก็บข้อความ เราก็จะเก็บได้ไม่เกิน 9 ตัวอักขระ ถ้าเราพยายามเก็บข้อความที่ยาวเกินไป ปัญหาก็จะเกิดขึ้นได้เมื่อโปรแกรมทำงาน

## 7.2.4 อาร์เรย์ของพอยน์เตอร์

นอกจากข้อมูลแบบพื้นฐานต่างๆแล้ว อาร์เรย์ยังสามารถเก็บพอยน์เตอร์ไว้ในตัวได้ พอยน์เตอร์ทั้งหลายเหล่านี้ อาจจะเป็นพอยน์เตอร์ที่ชี้ไปยังข้อมูลพื้นฐานอื่นๆแบบใดแบบหนึ่ง หรือชี้ไปยังอาร์เรย์ของข้อมูลชนิดใดชนิดหนึ่ง หรืออาจจะเป็นข้อมูลแบบโครงสร้างชนิดอื่น

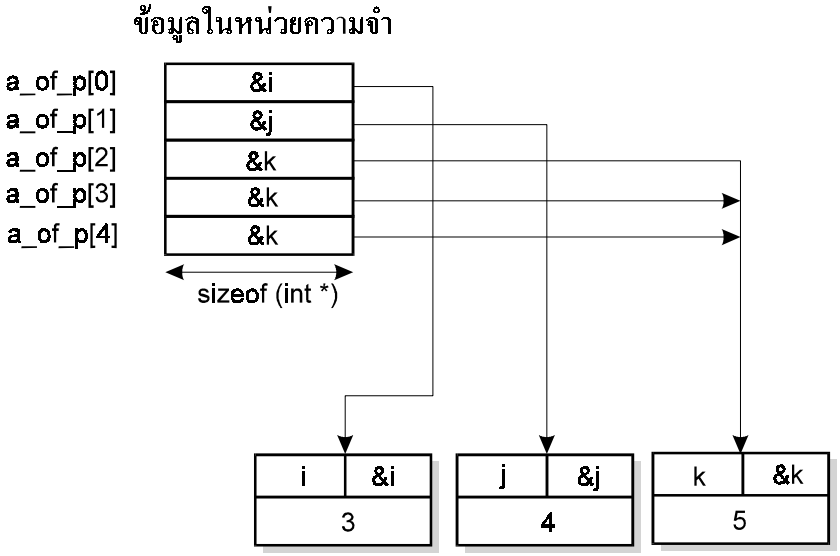
ตัวอย่าง เช่น เราต้องการแจ้งใช้ อาร์เรย์ที่เก็บพอยน์เตอร์หลายๆตัว ที่ใช้ชี้ไปยังข้อมูลแบบ `int` ถ้าเรากำหนดไว้ว่า อาร์เรย์ มีขนาดเท่ากับ 5 ดังนั้นอาร์เรย์นี้ก็สามารถเก็บพอยน์เตอร์ได้ 5 ตัว

```
int *a_of_p[5] = {0,0,0,0,0};
int i=3, j=4, k=5;
```

ในขั้นต้นนี้เราติดตั้งให้พอยน์เตอร์ในอาร์เรย์มีค่าเป็นศูนย์ทุกตัว ซึ่งหมายความว่า พอยน์เตอร์ทั้งห้าตัวเป็นพอยน์เตอร์ศูนย์ (Null Pointer)

```
a_of_p[0] = &i;
a_of_p[1] = &j;
a_of_p[2] = &k;
a_of_p[3] = a_of_p[2];
a_of_p[4] = a_of_p[3];
```

จากนั้นเราก็กำหนดให้พอยน์เตอร์แต่ละตัวชี้ไปยังข้อมูลแบบ int เช่น ชี้ไปยังที่อยู่ของตัวแปร i, j และ k เป็นต้น




จากตัวอย่าง มีพอยน์เตอร์จำนวนสามตัวในอาร์เรย์ ที่อ้างถึงแหล่งข้อมูลเดียวกัน คือชี้ไปยังแหล่งข้อมูลของตัวแปร k ในกรณีนี้ เราสามารถใช้พอยน์เตอร์ตัวใดก็ได้ในสามตัวเข้าถึงตัวแปร k โดยทางอ้อมได้และโปรดสังเกตว่า a_of_p[index] ให้ค่าที่เป็นที่อยู่ ซึ่ง index เป็นตัวแปรแบบ int มีค่าอยู่ระหว่าง 0 และ 4 ดังนั้นจึงหมายความว่า a_of_p[index] ทำหน้าที่เหมือนพอยน์เตอร์ที่ชี้ไปยังข้อมูลแบบ int ถ้าเราต้องการเข้าถึงข้อมูลที่ถูกเก็บไว้ในแหล่งข้อมูลที่พอยน์เตอร์นี้กำลังชี้ไป เราก็ต้องเขียนว่า *a_of_p[index] ซึ่งมีเครื่องหมายดอกจันทร์อยู่ข้างหน้า

```
#include <stdio.h>

int main()
{
     int *a_of_p[5] = {0,0,0,0,0};
     int i=3, j=4, k=5;
     int index;

     a_of_p[0] = &i;
     a_of_p[1] = &j;
     a_of_p[2] = &k;
     a_of_p[3] = a_of_p[2];
     a_of_p[4] = a_of_p[3];

     for(index=0; index < 5; index++)
          printf("%4d", *a_of_p[index]);
     printf("\n");

     *a_of_p[2] += 100;
```

```
     for(index=0; index < 5; index++)
          printf("%4d", *a_of_p[index]);
     printf("\n");

     return 0;
}
```

ผลการทำงานของโปรแกรมคือ

```
     3   4   5   5   5
     3   4 105 105 105
```

จากผลการทำงานของโปรแกรม เราจะเห็นได้ว่า ถ้าเราเปลี่ยนแปลงค่าของ

```
     *a_of_p[2]
     *a_of_p[3]
     *a_of_p[4]
```

นิพจน์ใดนิพจน์หนึ่ง ค่าของแหล่งข้อมูลที่พอยน์เตอร์กำลังชี้ไป ก็ไปเปลี่ยนไป เพราะพอยน์เตอร์ทั้งสามชี้ไปยังแหล่งข้อมูลเดียวกัน คือ ตัวแปร k ตามปรกติแล้วพอยน์เตอร์แต่ละตัวของอาร์เรย์จะถูกกำหนดให้ชี้ไปยังแหล่งข้อมูลที่แตกต่างกัน ก็เพราะเหตุผลที่ว่า ถ้าเราใช้พอยน์เตอร์มากกว่าหนึ่งตัวในการอ้างอิงแหล่งข้อมูลเดียวกัน ก็จะอาจจะทำให้เกิดปัญหาได้ เพราะจะทำให้ตรวจสอบได้ยากว่าพอยน์เตอร์ตัวใดและในช่วงเวลาใดเราได้ใช้ในการเปลี่ยนแปลงค่าของข้อมูล

ตัวอย่างต่อไปที่แสดงให้เห็นการใช้งานของอาร์เรย์ที่มีพอยน์เตอร์คือ การสร้างอาร์เรย์ที่ เก็บพอยน์เตอร์และชี้ไปยังข้อความหรือสายอักขระ เช่น

```
     char *days[]= {
          "Sunday", "Monday", "Tuesday", "Wednesday",
          "Thursday", "Friday", "Saturday"
     };
```

days เป็นอาร์เรย์ที่เก็บพอยน์เตอร์ทั้งหมด 7 ตัว โดยที่พอยน์เตอร์แต่ละตัวชี้ไปยังสายอักขระซึ่งเป็นชื่อของวันในภาษาอังกฤษ ดังนั้นนิพจน์ต่อไปนี้

```
     days[0]   days[1]   days[2]   days[3]
     days[4]   days[5]   days[6]
```

จึงทำหน้าที่เป็นพอยน์เตอร์ที่เก็บหมายเลขที่อยู่เริ่มต้นของข้อความ ต่างๆในหน่วยความจำ ซึ่งหมายถึงสายอักขระนั้นเอง ดังนั้นเราสามารถกล่าวได้ว่าตัวแปร days คืออาร์เรย์ของสายอักขระ

## 7.2.5 ตัวอย่างการเรียงสายอักขระในอาร์เรย์

เพื่อให้เห็นตัวอย่างของการใช้อาร์เรย์ที่เก็บที่อยู่ของสายอักขระต่างๆ (อาร์เรย์ของสายอักขระ) เรามาลองเรียนรู้ วิธีการง่ายๆในการเรียงข้อความตามลำดับตัวอักษร สำหรับการเรียงข้อมูล เราจะต้องมีการเปรียบเทียบข้อมูลเป็นคู่ๆ ว่า ข้อมูลใดมีค่ามากกว่าหรือน้อยกว่ากัน ซึ่งสำหรับการเปรียบเทียบนี้ เราก็ต้องกำหนดกฎเกณฑ์ขึ้น เช่น ถ้าข้อมูลในที่นี้หมายถึง ข้อความหรือสายอักขระ แล้วเราจะเปรียบเทียบกันอย่างไร จึงจะรู้ว่าข้อความไหนจะมาก่อนหรือหลัง สำหรับการเปรียบเทียบข้อความเราจะใช้ฟังก์ชัน `strcmp()` ซึ่งเป็นฟังก์ชันมาตราฐาน ในการเปรียบเทียบระหว่างข้อความ

```
int strcmp(const char *s1, const char *s2);
```

ถ้า `s1` มีค่ามากกว่า `s2` ฟังก์ชันก็จะให้ค่าที่มากกว่าศูนย์กลับคืน ถ้า `s1` และ `s2` มีค่าเท่ากันก็ให้ค่าที่จะเท่ากับศูนย์ ถ้าสายอักขระ `s1` มีค่าน้อยกว่า `s2` ก็จะได้ค่าที่น้อยกว่าศูนย์ เช่น

```
strcmp("aaa","aaaa")
```

จะได้ค่าที่น้อยกว่า ศูนย์ เพราะตัวอักขระสามตัวแรกของทั้งสองสายอักขระมีค่าเท่ากัน แต่เนื่องจากพารามิเตอร์ตัวแรกมีความยาวน้อยกว่า ดังนั้น พารามิเตอร์ตัวแรกจึงมีค่าน้อยกว่าพารามิเตอร์ตัวที่สอง

```
strcmp("aaa","AAA")
```

นิพจน์นี้ จะให้ค่าที่มากกว่าศูนย์ เนื่องจากฟังก์ชันจะเปรียบเทียบตัวอักขระแต่ละตัวจากสายอักขระทั้งสองไปทีละตัว และค่าของตัวอักษร `'a'` มีค่ามากกว่าค่าของตัวอักษร `'A'` ดังนั้นฟังก์ชันจึงให้ค่าที่มากกว่าศูนย์ โปรดสังเกตว่า ฟังก์ชัน `strcmp()` จะเปรียบเทียบตัวอักขระโดยเน้นความแตกต่างระหว่างตัวพิมพ์ใหญ่และเล็กด้วย

สำหรับการเรียงข้อความตามลำดับ เราจะใช้วิธีการเรียงแบบ Bubble Sort เพราะง่ายต่อการใช้งานและเราก็ได้ทำความรู้จักไปแล้ว แต่ก่อนที่เราจะใช้ฟังก์ชัน `bubble_sort()` เราจะต้องสร้างฟังก์ชันที่ใช้สำหรับการเปรียบเทียบสายอักขระขึ้นก่อน เนื่องจากว่า ฟังก์ชัน `bubble_sort()` ต้องการพอยน์เตอร์ที่ชี้ไปยังฟังก์ชันที่ใช้เปรียบเทียบพารามิเตอร์สองตัวว่าพารามิเตอร์ตัวแรกมีค่ามากกว่า พารามิเตอร์ตัวที่สองหรือไม่

```
int IsGreater(const void *s1, const void *s2)
{
   return (strcmp(*(const char **)s1,
                  *(const char **)s2) > 0);
}
```

s1 และ s2 เป็นพอยน์เตอร์อเนกประสงค์ที่ชี้ไปยังข้อมูลแต่ละตัวของอาร์เรย์ แต่เนื่องจากว่าข้อมูลของอาร์-เรย์แต่ละตัวเป็นพอยน์เตอร์ที่ชี้ไปยังสายอักขระ ดังนั้นเมื่อเราแปลงแบบของพอยน์เตอร์อเนกประสงค์เพื่อนำไปใช้งานต่อ เราจะต้องแปลงให้เป็นแบบของพอยน์เตอร์ที่ชี้ไปยังพอยน์เตอร์ จากนั้นจึงใช้โอเปอร์เรเตอร์ * วางไว้ข้างหน้าสำหรับอ่านค่า และค่าที่ได้จากนิพจน์ทั้งสองตามลำดับ คือ ที่อยู่ของสายอักขระสองสายที่เราต้องการนำเปรียบเทียบค่ากันโดยใช้ฟังก์ชัน strcmp()

โปรแกรมสมบูรณ์แบบที่แสดงให้เห็นการเรียงข้อความหลายๆข้อความที่มีอาร์เรย์เก็บที่อยู่ของข้อความเหล่านี้ไว้ ตัวอย่างข้างล่างนี้ ตัวแปร a ทำหน้าที่เป็นอาร์เรย์ที่เก็บพอยน์เตอร์ และพอยน์เตอร์แต่ละตัวก็จะชี้ไปยังข้อความ(สายอักขระ) ต่างๆ

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


void bubble_sort
 (   void *  array,
     size_t  n,
     size_t  keysize,
     int (*is_greater)(const void *, const void *) )
{
     size_t   i, j;
     size_t   n_exch = 0;

     char *a   = (char *)array;
     void *tmp = malloc(keysize+1);
     void *p1, *p2;

     for (i=0; i < n; i++)
     {
          for (j=1; j < n ; j++)
          {
            p1 = (void *)(a + keysize*(j-1));
            p2 = (void *)(a + keysize*(j));
            if ( is_greater(p1, p2) )
            {
               memcpy(tmp, p1, keysize);
               memcpy( p1, p2, keysize);
               memcpy( p2,tmp, keysize);
               n_exch++;
            }
          }
          if (n_exch==0) break;
          else n_exch=0;
     }
```

```
     free(tmp);
}

int IsGreater(const void *s1, const void *s2)
{
     return (strcmp(*(const char **)s1,
                    *(const char **)s2) > 0);
}

int main()
{
     char *a[] = {
          "strcpy"    , "strlen"  ,
          "strncmp"   , "strchr"  ,
          "strstr"    , "strcat"  ,
          "strncpy"   , "strncat" ,
          "strtok"    , "strdup"  ,
          "strstr"    , "strpbrk"
     };

     int  i, size, keysize;

     keysize = sizeof(char *);
     size = sizeof(a) / keysize;

     bubble_sort(a, size, keysize, IsGreater);

     for(i=0; i < size; i++)
          printf("%s\n", a[i]);

     return 0;
}
```

ถ้าเราต้องการตรวจสอบการทำงานของฟังก์ชันที่ใช้ในการเรียงข้อความ เราก็น่าจะลองสร้างข้อความหลายๆข้อความและมีขนาดที่แตกต่างกันไป เช่น การสร้างข้อความโดยสุ่ม

```
static const char ch_table[] =
 "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

ฟังก์ชัน `mkstr()` ใช้ในการสร้างสายอักขระที่มีความยาวเท่ากับ `len` และตัวอักขระแต่ละตัวจะได้จากสุ่มโดยใช้ฟังก์ชัน `rand()` และตัวอักขระเหล่านี้จะเป็นสมาชิกตัวใดตัวหนึ่งของอาร์เรย์ `ch_table` เท่านั้น

```
char *mkstr(int len)
{
     register unsigned i;
     int n = strlen(ch_table);
     char *new_str = malloc((len+1)*sizeof(char));
```

```
     assert(new_str);

     for(i=0; i < len; i++) {
          new_str[i] = ch_table[rand() % n];
     }
     new_str[len]= '\0';  /* null-terminated */

     return new_str;
}

void delstr(char *s)
{
     if(s) free(s);
}
```

ฟังก์ชัน `delstr()`จะใช้ในการปล่อยหน่วยความจำที่จองไว้สำหรับสายอักขระให้ว่าง

ตัวอย่าง เช่น

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

/* function prototypes */

char *mkstr (int);
extern void delstr (char *);
void bubble_sort (void *,size_t, size_t,
     int (*is_greater)(const void *, const void *));

#define N_STRING   100
#define LEN         60

int main()
{
     char *a[N_STRING] = {0};
     int  i, size, keysize;

     keysize = sizeof(char *);
     size    = sizeof(a) / keysize;

     for(i=0; i < size; i++)
          a[i] = mkstr(LEN);

     bubble_sort(a, size, keysize, IsGreater);

     for(i=0; i < size; i++) {
          printf("%3d) %s\n", i+1, a[i]);
          delstr(a[i]);
```

```
    }
    return 0;
}
```

ในโปรแกรมตัวอย่างข้างบน เราสร้างข้อความที่มีความยาว 60 ตัวอักขระทั้งหมด 100 ข้อความ แล้วเราก็จะใช้ฟังก์ชัน `bubble_sort()` ในการจัดเรียงข้อความใหม่ตามลำดับ

## 7.2.6 ฟังก์ชันมาตราฐานจาก <string.h>

สำหรับการประมวลผลตัวอักขระและข้อความที่เป็นสายอักขระ เรามักจะเกี่ยวข้องกับการใช้ฟังก์ชันมาตราฐานต่างๆที่นิยามไว้ใน <string.h> ซึ่งใช้ในหน้าที่และจุดประสงค์ที่แตกต่างกันออกไป ส่วนท้ายของบทนี้จึงเป็นการรวบรวมคำอธิบายสำหรับฟังก์ชันมาตราฐานเหล่านี้บางส่วน รวมทั้งตัวอย่างง่ายๆในการใช้งาน นอกจากนั้นยังได้นำเสนอตัวอย่างของการสร้างฟังก์ชันมาตราฐานแต่ละตัวเพื่อช่วยให้ผู้อ่านสามารถมองเห็นวิธีการสร้างฟังก์ชันขึ้นใช้ และเข้าใจการทำงานของฟังก์ชันได้ดียิ่งขึ้น

ฟังก์ชัน *memchr( )* <string.h>

■ **รูปแบบ**

```
void *memchr(const void *s, int c, size_t n);
```

■ **คำอธิบาย**

ฟังก์ชันนี้ใช้ในการค้นหาหน่วยความจำขนาดหนึ่งไบต์ที่มีค่าเท่ากับพารามิเตอร์ c (ค่าของ c จะถูกแปลงเป็น unsigned char ) ในบริเวณหน่วยความจำของอาร์เรย์ที่มีที่อยู่เริ่มต้นเท่ากับค่าของพอยน์เตอร์ s และจะค้นหาในหน่วยความจำทั้งหมดไม่เกิน n ไบต์ (มีค่ามากกว่าหรือเท่ากับศูนย์) นับตั้งแต่ไบต์ตัวแรกในที่อยู่ที่กำหนดโดยพอยน์เตอร์ s ไปต้นไป เมื่อฟังก์ชันพบไบต์ที่ต้องการในบริเวณหน่วยความจำดังกล่าวก็จะให้ที่อยู่ของไบต์นั้นกลับคืนทันที โดยเก็บไว้ในรูปของพอยน์เตอร์แบบ void ถ้าไม่พบไบต์ที่ต้องการก็จะคืนค่าเป็นพอยน์เตอร์ศูนย์

■ **โปรแกรมโค้ดสำหรับฟังก์ชัน**

```
#include <string.h>

void *memchr(const void *s, int c, size_t n)
{
      if (n > 0) {
            const unsigned char *p = (const unsigned char *)s;
            unsigned char uc = (unsigned char)c;
            do {
                  if (*p==uc)
                        return ((void *)p);
                  p++;
            }  while (--n);
      }
      return (NULL);
}
```

■ **ตัวอย่างการใช้งาน**

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

int main()
{
      char str[] = "Hello World!";
      char *cptr;

      cptr = (char *)memchr(str,'W',strlen(str));
      printf ("%s\n", cptr);
      return 0;
}
```

ฟังก์ชัน *memcmp( )* <string.h>

■ **รูปแบบ**

```
int memcmp(const void *s1, const void *s2, size_t n);
```

■ **คำอธิบาย**

ฟังก์ชันนี้ใช้ในการเปรียบเทียบข้อมูลแต่ละไบต์จากบริเวณหน่วยความจำสองแห่งที่กำหนดโดย s1 และ s2 ตามลำดับ พารามิเตอร์ n เป็นตัวกำหนดว่า จะต้องเปรียบเทียบข้อมูลที่มีขนาดหนึ่งไบต์เป็นจำนวนกี่ตัว ถ้าข้อมูลจากบริเวณหน่วยความจำทั้งสองแห่งมีข้อมูลอย่างน้อย n ตัวแรกที่เหมือนกัน ฟังก์ชันจะให้ค่ากลับคืนเท่ากับศูนย์กลับคืน ถ้าไม่เป็นเช่นนั้น ฟังก์ชันก็จะเปรียบเทียบจนกว่าจะพบว่ามีข้อมูลขนาดหนึ่งไบต์จากทั้งสองแห่งที่มีค่าไม่เท่ากัน และก็จะเปรียบเทียบไบต์ทั้งสองในเชิงตัวเลขว่าตัวใดมีค่ามากกว่ากัน ถ้าข้อมูลจาก s1 มีค่ามากกว่า s2 ค่าที่ได้จากฟังก์ชันจะเป็นบวก และถ้ามีค่าน้อยกว่าก็จะให้ค่าที่เป็นลบ

| ค่ากลับคืนของฟังก์ชัน | ความหมาย |
|---|---|
| น้อยกว่าศูนย์ | s1 มีค่าน้อยกว่า s2 |
| เท่ากับศูนย์ | s1 มีค่าเท่ากับ s2 |
| มากกว่าศูนย์ | s1 มีค่ามากกว่า s2 |

ตามปรกติแล้ว ฟังก์ชันนี้จะใช้สำหรับการเปรียบเทียบค่าของข้อมูลจากสองอาร์เรย์หรือสายอักขระ

■ **โปรแกรมโค้ดสำหรับฟังก์ชัน**

```
#include <string.h>

int memcmp(const void *s1, const void *s2, size_t n)
{
      unsigned char *p1, *p2;

      if (n==0) {
            return(0);
      }
      p1 = (unsigned char *)s1;
      p2 = (unsigned char *)s2;
      while (*p1++ == *p2++)
            if (--n == 0)
                  return (0);
      return (*(p1-1) - *(p2-1));
}
```

■ **ตัวอย่างการใช้งาน**

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

int main()
{
      char s1[] = "AAA";
      char s2[] = "BBB";
      int ret;

      ret = memcmp(s1, s2, strlen(s1));
      printf("s1 is %s than s2\n",
             (ret > 0) ? "greater" : "less");

      return 0;
}
```

---

ฟังก์ชัน *memcpy( )* <string.h>

---

■ **รูปแบบ**

**`void *memcpy (void *s1, const void *s2, size_t n);`**

■ **คำอธิบาย**

ฟังก์ชัน `memcpy()` จะนำค่าของข้อมูลจำนวน `n` ไบต์ จาก `s2` ไปใส่ไว้ในบริเวณหน่วยความจำที่กำหนดโดย `s1` ฟังก์ชันนี้ให้ที่อยู่ที่ `s1` กำลังชี้ไปเป็นค่ากลับคืน ข้อมูลจำนวน `n` ไบต์ที่มีอยู่เดิมใน `s1` จะถูกเขียนทับโดยข้อมูลจาก `s2`

ฟังก์ชันนี้จะทำงานไม่ถูกต้อง ถ้า `s1` และ `s2` ใช้บริเวณหน่วยความจำร่วมกัน

■ **โปรแกรมโค้ดสำหรับฟังก์ชัน**

```
#include <string.h>

void *memcpy (void *s1, const void *s2, size_t n)
{
      if (n > 0) {
         unsigned char       *p=(unsigned char *)s1;
         const unsigned char *q=(const unsigned char *)s2;
         do {
             *p++ = *q++;
         } while (--n);
         return s1;
      }
      return (NULL);
}
```

■ **ตัวอย่างการใช้งาน**

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

int main()
{
      char  src[] = "-----";
      char dest[] = "0123456789";
      char *ptr;

      printf("source : %s\n", src);
      printf("destination before memcpy : %s\n",
             dest);
      ptr = (char *)memcpy(dest, src, strlen(src));

      if (ptr)
         printf("destination after memcpy  : %s\n",
                dest);
      else
         printf("memcpy failed\n");
      return 0;
}
```

ฟังก์ชัน *memmove( )* <string.h>

■ **รูปแบบ**

```
void *memmove (void *s1, const void *s2, size_t n);
```

■ **คำอธิบาย**

ฟังก์ชัน `memmove()` ทำงานในลักษณะเดียวกันกับ `memcpy()` แต่สามารถใช้ได้ในกรณีที่ `s1` และ `s2` มีบริเวณหน่วยความจำร่วมกัน

■ **โปรแกรมโค้ดสำหรับฟังก์ชัน**

```
#include <string.h>

void *memmove (void *s1, const void *s2, size_t n)
{
      if (n > 0) {
         size_t sn;
         unsigned char *st,*t,*p = (unsigned char *)s1;
         const unsigned char *q = (const unsigned char *)s2;

         st = malloc( n*sizeof(unsigned char) );
         for (t=st,sn=n; sn--; ) {
              *t++ = *q++;
         }
         for (t=st,sn=n; sn--; ) {
              *p++ = *t++;
         }
         free(st);
         return s1;
      }
      return (NULL);
}
```

■ **ตัวอย่างการใช้งาน**

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

int main()
{
      char text[] = "0123456789";
      char *src = (text+5), *dest = (text+0);
      char *ptr;

      printf("destination before memmove : %s\n",
             dest);
      ptr = (char *)memmove(dest, src, strlen(src));

      if (ptr)
         printf("destination after memmove  : %s\n",
                dest);
      else
         printf("memmove failed\n");
      return 0;
}
```

## ฟังก์ชัน *memset( )* <string.h>

■ **รูปแบบ**

```
void *memset(void *s, int c, size_t n);
```

■ **คำอธิบาย**

ฟังก์ชัน memset() เขียนข้อมูลจำนวน n ไบต์ลงในหน่วยความจำที่กำหนดโดย s ซึ่งค่าของข้อมูล n ตัวแรกในพื้นที่หน่วยความจำนั้นทุกตัวจะมีค่าเท่ากับ c (จะถูกแปลงเป็น unsigned char) ฟังก์ชันจะให้หมายเลขที่อยู่ที่เป็นค่าของ s เป็นค่ากลับคืน

ตามปรกติแล้วเราจะใช้ฟังก์ชันนี้ในการติดตั้งค่าเริ่มต้นของอาร์เรย์ เช่น ติดตั้งค่าเริ่มต้นให้ข้อมูลทุกตัวของอาร์เรย์เป็นศูนย์ โดยใช้ ศูนย์เป็นพารามิเตอร์สำหรับ c

■ **โปรแกรมโค้ดสำหรับฟังก์ชัน**

```
#include <string.h>

void *memset (void *s, int c, size_t n)
{
      if (n > 0) {
            unsigned char *p = (unsigned char *)s;
            unsigned char uc = (unsigned char)c;
            do {
                  *p++ = uc;
            } while (--n);
      }
      return (s);
}
```

■ **ตัวอย่างการใช้งาน**

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

int main()
{
      char text[] = "0123456789";

      printf ("text before memset : %s\n", text);
      memset (text, '-', strlen(text) - 5);
      printf ("text after memset  : %s\n", text);
      return 0;
}
```

---

ฟังก์ชัน *strcasecmp( )* <string.h>

---

■ **รูปแบบ**

**int strcasecmp (const char *s1, const char *s2);**

**■ คำอธิบาย**

ฟังก์ชันนี้จะเปรียบเทียบสายอักขระ s1 กับ s2 โดยไม่คำนึงถึงว่า ตัวอักขระแต่ละตัวที่ใช้เปรียบเทียบกันจะเป็นตัวพิมพ์ใหญ่หรือเล็ก (ตัวอักขระจะถูกแปลงเป็นตัวพิมพ์เล็กก่อนแล้วจึงนำมาเปรียบเทียบกันเป็นคู่ๆไป) ฟังก์ชันจะให้ค่ากลับคืนที่เป็นศูนย์ ถ้า s1 มีค่าเท่ากับ s2 ถ้าสายอักขระทั้งสองไม่เท่ากัน ฟังก์ชันจะให้ค่าเป็นบวก ถ้า s1 มีค่ามากกว่า s2 หรือ ถ้ามีค่าน้อยกว่า ก็จะให้ค่าที่เป็นลบ

สายอักขระทั้งสองจะต้องลงท้ายด้วย '\0' เพราะตัวอักขระที่มีค่าเท่ากับศูนย์จะใช้ในการกำหนดความยาวของสายอักขระ

สำหรับการสร้างฟังก์ชันนี้ เราจะใช้ตารางเข้าช่วย ตารางนี้คืออาร์เรย์ของข้อมูลแบบ unsigned char เก็บข้อมูลเกี่ยวกับการแปลงตัวอักขระจากตัวพิมพ์ใหญ่ เป็นตัวพิมพ์เล็ก (เช่น จาก 'A' เป็น 'a') เพื่อเพิ่มความเร็วในการทำงานของฟังก์ชัน และถ้าเราไม่ใช้อาร์เรย์ในลักษณะนี้เราก็ ต้องใช้ฟังก์ชัน lowercase() แทน

```
static const unsigned char  charmap[] =
{
 '\000', '\001', '\002', '\003', '\004', '\005', '\006', '\007',
 '\010', '\011', '\012', '\013', '\014', '\015', '\016', '\017',
 '\020', '\021', '\022', '\023', '\024', '\025', '\026', '\027',
 '\030', '\031', '\032', '\033', '\034', '\035', '\036', '\037',
 '\040', '\041', '\042', '\043', '\044', '\045', '\046', '\047',
 '\050', '\051', '\052', '\053', '\054', '\055', '\056', '\057',
 '\060', '\061', '\062', '\063', '\064', '\065', '\066', '\067',
 '\070', '\071', '\072', '\073', '\074', '\075', '\076', '\077',
 '\100', '\141', '\142', '\143', '\144', '\145', '\146', '\147',
 '\150', '\151', '\152', '\153', '\154', '\155', '\156', '\157',
 '\160', '\161', '\162', '\163', '\164', '\165', '\166', '\167',
 '\170', '\171', '\172', '\133', '\134', '\135', '\136', '\137',
 '\140', '\141', '\142', '\143', '\144', '\145', '\146', '\147',
 '\150', '\151', '\152', '\153', '\154', '\155', '\156', '\157',
 '\160', '\161', '\162', '\163', '\164', '\165', '\166', '\167',
 '\170', '\171', '\172', '\173', '\174', '\175', '\176', '\177',
 '\200', '\201', '\202', '\203', '\204', '\205', '\206', '\207',
 '\210', '\211', '\212', '\213', '\214', '\215', '\216', '\217',
 '\220', '\221', '\222', '\223', '\224', '\225', '\226', '\227',
 '\230', '\231', '\232', '\233', '\234', '\235', '\236', '\237',
 '\240', '\241', '\242', '\243', '\244', '\245', '\246', '\247',
 '\250', '\251', '\252', '\253', '\254', '\255', '\256', '\257',
 '\260', '\261', '\262', '\263', '\264', '\265', '\266', '\267',
 '\270', '\271', '\272', '\273', '\274', '\275', '\276', '\277',
 '\300', '\301', '\302', '\303', '\304', '\305', '\306', '\307',
 '\310', '\311', '\312', '\313', '\314', '\315', '\316', '\317',
 '\320', '\321', '\322', '\323', '\324', '\325', '\326', '\327',
 '\330', '\331', '\332', '\333', '\334', '\335', '\336', '\337',
 '\340', '\341', '\342', '\343', '\344', '\345', '\346', '\347',
 '\350', '\351', '\352', '\353', '\354', '\355', '\356', '\357',
 '\360', '\361', '\362', '\363', '\364', '\365', '\366', '\367',
 '\370', '\371', '\372', '\373', '\374', '\375', '\376', '\377'
};
```

หมายเหตุ สำหรับดอสหรือวินโดวส์ ฟังก์ชันนี้จะใช้ชื่อว่า strcmpi()

■ **โปรแกรมโค้ดสำหรับฟังก์ชัน**

```
#include <string.h>

int strcasecmp (const char *s1, const char *s2)
{
      const  unsigned char *cm = charmap,
            *p = (const unsigned char *)s1,
            *q = (const unsigned char *)s2;

      while (cm[*p] == cm[*q++]) {
            if (*p++ == '\0')
                  return (0);
      }
      return (cm[*p] - cm[*(q-1)]);
}
```

■ **ตัวอย่างการใช้งาน**

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

#ifndef UNIX
#define strcasecmp strcmpi
#endif

int main()
{
      char s1[] = "AbCdefG";
      char s2[] = "aBCDEFg";

      if (strcasecmp(s1,s2)==0)
          printf("s1 and s2 are equal.\n");
      else
          printf("s1 and s2 are different.\n");
      return 0;
}
```

ฟังก์ชัน *strncasecmp( )* <string.h>

■ **รูปแบบ**

```
int strncasecmp (const char *s1, const char *s2,
                 size_t n);
```

■ **คำอธิบาย**

ฟังก์ชันนี้ใช้ในหน้าที่เดียวกันกับฟังก์ชัน `strcasecmp()` เพียงจำกัดอยู่แค่ `n` ไบต์แรกของข้อมูลที่มีอยู่ (ยกเว้นในกรณีที่สายอักขระ `s1` มีความยาวน้อยกว่าค่าของ `n` ซึ่งในกรณีนี้ จะไม่มีความแตกต่างระหว่างฟังก์ชันทั้งสอง)

หมายเหตุ สำหรับดอสหรือวินโดวส์ ฟังก์ชันนี้จะใช้ชื่อว่า `strncmpi()`

**■ โปรแกรมโค้ดสำหรับฟังก์ชัน**

```
int strncasecmp (const char *s1, const char *s2, size_t n)
{
      if (n > 0) {
            const unsigned char *cm = charmap,
                  *p = (const unsigned char *)s1,
                  *q = (const unsigned char *)s2;
            do {
                  if (cm[*p] != cm[*q++])
                  return (cm[*p] - cm[*(q-1)]);
                  if (*p++ == '\0')
                  break;
            } while (--n);
      }
      return (0);
}
```

**■ ตัวอย่างการใช้งาน**

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

#ifndef UNIX
#define strncasecmp strncmpi
#endif

int main()
{
      char s1[] = "AbCdefG";
      char s2[] = "aBCDEFg";

      if (strncasecmp(s1,s2,strlen(s1))==0)
         printf("s1 and s2 are equal.\n");
      else
         printf("s1 and s2 are different.\n");

      return 0;
}
```

## ฟังก์ชัน *strcat( )* <string.h>

**■ รูปแบบ**

```
char *strcat(char *s1, const char *s2);
```

**■ คำอธิบาย**

ฟังก์ชันนี้ จะนำข้อมูลของสายอักขระ s2 ไปใส่ต่อท้ายสายอักขระ s1 และตัวขระ '\0' ที่อยู่ท้าย s1 จะถูกเขียนทับโดยตัวอักขระตัวแรกของ s2 และเมื่อฟังก์ชันจบการทำงาน s1 ก็จะมีความยาวเท่ากับ strlen(s1) + strlen(s2) และมีตัวขระ '\0' อยู่ข้างท้าย และค่าที่ได้จากฟังก์ชันจะเป็นที่อยู่เริ่มต้นของสายอักษร s1

โปรดสังเกตว่า สายอักขระ s1 จะต้องมีหน่วยความจำเพียงพอสำหรับตัวอักขระแต่ละตัวของ s2 เพราะตัวอักขระจาก s2 จะถูกทำสำเนาและเก็บต่อท้าย s1 ไปตามลำดับ ถ้าหน่วยความจำสำหรับ s1 มีค่าน้อยกว่า strlen(s1)+strlen(s2)+1 ก็จะเกิดปัญหาเวลาใช้งาน นอกจากนี้ฟังก์ชัน strcat() ตั้งเงื่อนไขไว้อีกว่า s1 และ s2 จะต้องเป็นสายอักขระที่ไม่ใช่ตัวเดียวกัน หรือใช้หน่วยความจำร่วมกัน

ตัวอย่างที่ไม่ถูกต้อง เช่น

```
char *s[] = "Hello World!";
strcat (s+4, s);
strcat (s, s);
```

**■ โปรแกรมโค้ดสำหรับฟังก์ชัน**

```
#include <string.h>

char *strcat (char *s1, const char *s2)
{
      char *s = s1;

      while (*s1) ++s1;
      while ((*s1++ = *s2++)!= '\0');
      return(s);
}
```

**■ ตัวอย่างการใช้ฟังก์ชัน**

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

int main()
{
      char s[20] = "Hello ";

      printf("%s", strcat(s, "World!\n") );
      return 0;
}
```

ฟังก์ชัน *strchr( )* <string.h>

■ **รูปแบบ**

```
char *strchr(const char *s, int c);
```

■ **คำอธิบาย**

ฟังก์ชันจะค้นหาตำแหน่งของตัวอักขระที่มีค่าเท่ากับ c (ค่าของ c จะถูกแปลงเป็น char) ฟังก์ชันจะให้ค่ากลับคืนเป็นพอยน์เตอร์ศูนย์ NULL ถ้าไม่พบว่ามีข้อมูลใดๆในบริเวณหน่วยความจำที่อ้างถึง มีค่าเท่ากับ c

■ **โปรแกรมโค้ดสำหรับฟังก์ชัน**

```
#include <string.h>

char *strchr(const char *s, int c)
{
      char ch = (char)c;

      for ( ; ; ++s) {
            if (*s == ch)
                  return ((char *)s);
            if (*s == '\0')
                  return(NULL);
      }
}
```

■ **ตัวอย่างการใช้ฟังก์ชัน**

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

int main()
{
      char s[12];
      char *ptr, ch = 'W';

      strcpy(s, "Hello World!");
      ptr = strchr(s, ch);
      if (ptr)
         printf("The character %c is at position: %d\n",
                  ch, (int)(ptr-s));
      else
         printf("The character was not found\n");
      return 0;
}
```

ฟังก์ชัน *strcmp( )* <string.h>

■ **รูปแบบ**

```
int strcmp(const char *s1, const char *s2);
```

■ **คำอธิบาย**

ฟังก์ชันนี้เปรียบเทียบระหว่างสายอักขระ s1 และ s2 เพื่อดูว่า ค่าของสายอักขระ s1 มีค่ามากกว่า s2 หรือไม่ ถ้า s1 มีค่าเท่ากับ s2 แล้วฟังก์ชันจะให้ค่ากลับคืนเท่ากับศูนย์ ถ้า s1 มีค่าน้อยกว่า s2 ค่าที่ได้จะเป็นลบ ถ้า s1 มีค่ามากกว่า s2 ก็ได้ค่าที่จะเป็นบวก

| ค่ากลับคืนของฟังก์ชัน | ความหมาย |
|---|---|
| น้อยกว่าศูนย์ | s1 มีค่าน้อยกว่า s2 |
| เท่ากับศูนย์ | s1 มีค่าเท่ากับ s2 |
| มากกว่าศูนย์ | s1 มีค่ามากกว่า s2 |

ฟังก์ชัน strcmp() มักจะใช้ในการเปรียบเทียบเพื่อเรียงข้อมูลที่เป็นสายอักขระหลายๆตัว

■ **โปรแกรมโค้ดสำหรับฟังก์ชัน**

```
#include <string.h>

int strcmp(const char *s1, const char *s2)
{
   while (*s1 == *s2++)
      if (*s1++ == '\0')
         return (0);
   return (*(unsigned char*)s1 - *(unsigned char*)(s2-1));
}
```

■ **ตัวอย่างการใช้ฟังก์ชัน**

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

int main()
{
      char s1[] = "abcdef";
      char s2[] = "abcdeF";
      int ret;

      ret = strcmp(s1,s2);
```

```
        if (ret > 0)
              printf("s1 is greater than s2.\n");
        else if (ret < 0)
              printf("s1 is less than s2.\n");
        else
              printf("s1 and s2 are equal.\n");
        return 0;
}
```

## ฟังก์ชัน *strcpy( )* <string.h>

**■ รูปแบบ**

```
char *strcpy(char *s1, const char *s2);
```

**■ คำอธิบาย**

ฟังก์ชันนี้จะคัดลอกข้อมูลแต่ละตัวจากสายอักขระ s2 แล้วเขียนลงในสายอักขระ s1 ตามลำดับ s2 จะต้องเป็นสายอักขระหรืออาร์เรย์ที่มีข้อมูลตัวท้ายเป็น '\0'

**■ โปรแกรมโค้ดสำหรับฟังก์ชัน**

```
#include <string.h>

char *strcpy(char *s1, const char *s2)
{
      char *s = s1;
      for ( ; (*s1 = *s2) != '\0'; ++s1, ++s2);
      return (s);
}
```

**■ ตัวอย่างการใช้ฟังก์ชัน**

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

int main()
{
      char str[20];

      strcpy (str,"Hello ");
      printf ("%s\n", str);

      strcpy (str+6, "World!");
      printf ("%s\n", str);
      return 0;
}
```

ฟังก์ชัน *strcspn( )* <string.h>

■ **รูปแบบ**

```
size_t  strcspn(const char *s1, const char *s2);
```

■ **คำอธิบาย**

ฟังก์ชันนี้จะเปรียบเทียบสายอักขระ s2 กับ s1 โดยเริ่มตรวจสอบอักขระตัวแรกของ s2 ว่ามีตัวอักขระแบบนี้ใน s1 หรือไม่ ถ้ามีก็ให้ค่าที่เท่ากับตำแหน่งของตัวอักขระนี้ใน s1 ถ้าไม่มีก็เปรียบเทียบตัวอักขระตัวที่สองของ s2 ต่อไปตามวิธีการ ไปเรื่อยๆ จนกว่าจะพบหรือได้เปรียบเทียบตัวอักขระใน s2 ครบทุกตัวแล้ว ถ้าสายอักขระทั้งสองไม่มีตัวอักขระใดๆที่เหมือนกัน ค่าที่ได้จะเท่ากับความยาวของสายอักขระ s1

■ **โปรแกรมโค้ดสำหรับฟังก์ชัน**

```
#include <string.h>

size_t  strcspn (const char *s1, const char *s2)
{
      const char *p, *spanp;
      char c, sc;

      for (p = s1; ;) {
            c = *p++;
            spanp = s2;
            do {
                  if ((sc = *spanp++) == c)
                        return (p - 1 - s1);
            } while (sc != 0);
      }
}
```

■ **ตัวอย่างการใช้ฟังก์ชัน**

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

int main()
{
      char s1[] = "!$@C#!&*";
      char s2[] = "ABCD";
      char s3[] = "X";
      int pos;

      pos = strcspn(s1, s2);
      if (pos < strlen(s1))
        printf("Character where s1 and s2 intersect is %c.\n",
              s1[pos]);
      else
```

```
        printf("No character where s1 and s2 intersect.\n");

      pos = strcspn(s1, s3);
      if (pos < strlen(s1))
        printf("Character where s1 and s3 intersect is %c\n",
               s1[pos]);
      else
        printf("No character where s1 and s3 intersect.\n");
      return 0;
}
```

ฟังก์ชัน *strlen( )* <string.h>

■ **รูปแบบ**

```
size_t  strlen(const char *str);
```

■ **คำอธิบาย**

ฟังก์ชันนี้ใช้ในการหาความยาวของสายอักขระ ซึ่งก็คือการนับตัวอักขระที่มีอยู่จากตัวแรกไปเรื่อยๆ เมื่อพบตัวอักขระใดที่มีค่าเท่ากับศูนย์ก็จะหยุดนับและจำนวนของตัวอักขระที่นับได้ (ไม่นับตัวอักขระที่มีค่าเท่ากับศูนย์) จะเป็นความยาวของสายอักขระ `str` ในขณะนั้น

■ **โปรแกรมโค้ดสำหรับฟังก์ชัน**

```
#include <string.h>

size_t  strlen(const char *str)
{
      register const char *s;
      for (s = str; *s; ++s);
      return( (size_t)(s - str) );
}
```

■ **ตัวอย่างการใช้ฟังก์ชัน**

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

int main()
{
      char  s1[]   = "12345";
      char  s2[20] = "12345";
      char  s3[10] = {'1','2','3','4','5'};
      char *s4     = "12345";
      char *s5     = "12345\0";

      printf("strlen(s1) = %2d\n",   strlen(s1));
      printf("strlen(s2) = %2d\n",   strlen(s2));
      printf("strlen(s3) = %2d\n",   strlen(s3));
      printf("strlen(s4) = %2d\n",   strlen(s4));
      printf("strlen(s5) = %2d\n\n", strlen(s5));
```

```
        printf("sizeof(s1) = %2d\n",   sizeof(s1));
        printf("sizeof(s2) = %2d\n",   sizeof(s2));
        printf("sizeof(s3) = %2d\n",   sizeof(s3));
        printf("sizeof(s4) = %2d\n",   sizeof(s4));
        printf("sizeof(s5) = %2d\n\n", sizeof(s5));
        return 0;
}
```

ฟังก์ชัน *strncat( )* <string.h>

■ **รูปแบบ**

**char *strncat(char *s1, const char *s2, size_t n);**

■ **คำอธิบาย**

ฟังก์ชันนี้ทำหน้าที่เหมือนฟังก์ชัน strcat() เพียงแต่เราจะกำหนดไว้ว่า จะเติมตัวอักขระจาก s2 ต่อท้าย s1 อย่างมากที่สุดไม่เกิน n ตัว ถ้า s2 มีความยาวมากกว่า n แล้ว ตัวอักขระ n ตัวแรกเท่านั้นจาก s2 จะถูกเขียนต่อท้าย s1 พารามิเตอร์ n จะต้องมีค่ามากกว่าหรือเท่ากับศูนย์

■ **โปรแกรมโค้ดสำหรับฟังก์ชัน**

```
#include <string.h>

char *strncat(char *s1, const char *s2, size_t n)
{
      if (n != 0) {
            register char *d = s1;
            register const char *s = s2;

            while (*d != 0)
                  d++;
            do {
                  if ((*d = *s++) == 0)
                        break;
                  d++;
            }  while (--n != 0);
            *d = 0;
      }
      return (s1);
}
```

■ **ตัวอย่างการใช้ฟังก์ชัน**

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

int main()
{
      char  s1[30]= {0};
      char *s2  = "C Programming";
```

```
        char *s3  = "Language";

        strncat(s1, s2, strlen(s2));
        printf("[%s]\n", s1);
        strncat(s1, " ", 1);
        printf("[%s]\n", s1);
        strncat(s1, s3, strlen(s3));
        printf("[%s]\n", s1);
        printf("size of string = %d\n", strlen(s1));
        return 0;
}
```

## ฟังก์ชัน *strncmp( )* <string.h>

### ■ รูปแบบ

```
int strncmp(const char *s1, const char *s2, size_t n);
```

### ■ คำอธิบาย

ฟังก์ชันนี้ทำหน้าที่เหมือน `strcmp()` ยกเว้นแต่ ฟังก์ชันจะเปรียบเทียบเฉพาะตัวอักขระจากสายอักขระ `s1` และ `s2` ไม่เกิน `n` ตัว

### ■ โปรแกรมโค้ดสำหรับฟังก์ชัน

```
#include <string.h>

int strncmp (const char *s1, const char *s2, size_t  n)
{
      if (n == 0)
            return (0);
      do {
            if (*s1 != *s2++)
                  return ( *(unsigned char *)s1 -
                           *(unsigned char *)(s2-1) );
            if (*s1++ == 0)
                  break;
      } while (--n != 0);
      return (0);
}
```

### ■ ตัวอย่างการใช้ฟังก์ชัน

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

int main()
{
      char  s1[30] = "aabecdccdedacdbebeaaacddbbebea";
      char *s2 = "bebea";
      int i, found = 0,
          len1 = strlen(s1),
          len2 = strlen(s2),
```

```
            len  = len1 - len2;

      for (i=0; i < len; ++i)
        if (!strncmp(s1+i, s2, len2)) {
           printf("substring found at position %d!\n", i);
           found++;
        }

      if (!found)
           printf("substring not found!\n");
      return 0;
}
```

ฟังก์ชัน *strncpy( )* <string.h>

■ **รูปแบบ**

**char *strncpy(char *s1, const char *s2, size_t n);**

■ **คำอธิบาย**

ฟังก์ชันนี้ ทำหน้าที่เหมือน `strcpy()` ยกเว้นแต่ฟังก์ชันจะทำการคัดลอกเฉพาะตัวอักขระจากสายอักขระ `s2` ไปยัง `s1` ไม่เกิน `n` ตัว

■ **โปรแกรมโค้ดสำหรับฟังก์ชัน**

```
#include <string.h>

char  *strncpy(char *s1, const char *s2, size_t  n)
{
      if (n != 0) {
            register       char *d = s1;
            register const char *s = s2;

            do {
               if ((*d++ = *s++) == 0) {
                  /* NULL pad the remaining n-1 bytes */
                  while (--n != 0)
                     *d++ = 0;
                     break;
                }
            } while (--n != 0);
      }
      return (s1);
}
```

■ **ตัวอย่างการใช้ฟังก์ชัน**

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

int main()
{
```

```
        char  s1[31];
        char *s2 = "*****";
        int i, len = sizeof(s1)-1;

        for (i=0; i < len; i+=6)
              strncpy(s1+i, s2, 6);
        printf("%s\n", s1);

        for (i=0; i < len; i+=5)
              strncpy(s1+i, s2, 6);
        printf("%s\n", s1);
        return 0;
}
```

## ฟังก์ชัน *strpbrk( )* <string.h>

### ■ รูปแบบ

```
char *strpbrk(const char *s1, const char *s2);
```

### ■ คำอธิบาย

ฟังก์ชันนี้จะให้ค่าที่อยู่ของตัวอักขระใน s1 ตัวแรกสุดที่ตรงกับตัวอักขระใดๆใน s2 ถ้าไม่พบว่าทั้งสองสายอักขระมีตัวอักขระใดๆเลยที่เหมือนกันก็จะให้พอยน์เตอร์เท่ากับศูนย์เป็นค่ากลับคืน

### ■ โปรแกรมโค้ดสำหรับฟังก์ชัน

```
#include <string.h>

char *strpbrk(const char *s1, const char *s2)
{
        register const char *scanp;
        register int c, sc;

        while ((c = *s1++) != 0) {
              for (scanp = s2; (sc = *scanp++) != 0;)
                    if (sc == c)
                          return ((char *)(s1 - 1));
        }
        return (NULL);
}
```

### ■ ตัวอย่างการใช้ฟังก์ชัน

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

int main()
{
        char s1[]   = "abcde";
        char s2[]   = "%^:!b$c%&a*()c#@d";
        char *p;
        int i, len = strlen(s1);
```

```
        for(i=0; i < len; i++)
          if ((p=strpbrk(s1+i,s2))!=NULL)
                printf("%s\n", p);
          else
                printf("Not found\n");
        return 0;
}
```

ฟังก์ชัน *strrchr( )* <string.h>

■ **รูปแบบ**

```
char *strrchr(const char *p, char ch);
```

■ **คำอธิบาย**

ฟังก์ชันนี้ใช้ในการค้นหาตัวอักขระของสายอักขระ p ที่มีค่าเท่ากับ ch โดยเริ่มค้นหาจากข้างหลังไปยังข้างหน้าของสายอักขระเมื่อพบตัวอักขระที่มีค่าตามที่ต้องการ ฟังก์ชันก็จะให้พอยน์เตอร์ที่ชี้ไปยังตัวอักขระตัวนั้นเป็นค่ากลับคืน ถ้าไม่พบตัวขระใดๆที่มีค่าเท่ากับค่าของพารามิเตอร์ ch ก็จะได้พอยน์เตอร์ศูนย์เป็นค่ากลับคืน

■ **โปรแกรมโค้ดสำหรับฟังก์ชัน**

```
#include <string.h>

char *strrchr(const char *p, char ch)
{
        register char *s = NULL;

        for (  ;  ; ++p) {
              if (*p == ch)
                    s = (char *)p;
              if (!*p)
                    return(s);
        }
}
```

■ **ตัวอย่างการใช้ฟังก์ชัน**

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

int main()
{
        char s[] = "LrrrRrrRrrr";
        char *p;

        if ((p=strrchr(s,'R')) != NULL)
```

```
            printf("%s\n", p);

      if ((p=strrchr(s,'L')) != NULL)
            printf("%s\n", p);
      return 0;
}
```

ฟังก์ชัน *strspn( )* <string.h>

■ **รูปแบบ**

**size_t  strspn(const char *s1, const char *s2);**

■ **คำอธิบาย**

ฟังก์ชันนี้จะเปรียบเทียบสายอักขระสองสาย เริ่มต้นจากตัวอักขระตัวแรก แล้วเปรียบเทียบไปเรื่อยๆ จนกว่าจะสิ้นสุดสายอักขระ แต่ทันทีที่พบว่าตัวอักขระในตำแหน่งเดียวกันจากทั้งสองสายอักขระมีค่าไม่เท่ากัน ฟังก์ชันก็จะจบการทำงานและให้ค่าที่เท่ากับจำนวนของตัวอักขระที่เหมือนกันตั้งแต่เริ่มต้น ถ้าค่าที่ได้มีเท่ากับศูนย์ หมายความว่าตัวอักขระตัวแรกสุดจากสายอักขระทั้งสองไม่เหมือนกัน

■ **โปรแกรมโค้ดสำหรับฟังก์ชัน**

```
#include <string.h>

size_t  strspn(const char *s1, const char *s2)
{
      register const char *p = s1, *spanp;
      register char c, sc;

cont:
      c = *p++;
      for (spanp = s2; (sc = *spanp++) != 0;)
            if (sc == c)
                  goto cont;
      return (p - 1 - s1);
}
```

■ **ตัวอย่างการใช้ฟังก์ชัน**

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

int main()
{
      char *s1 = "c:\program\win31";
      char *s2 = "c:\program\msdos";
      int pos;

      pos = strspn (s1, s2);
```

```
        printf("%s\n %s\n", s1+pos, s2+pos);
        return 0;
}
```

ฟังก์ชัน *strstr( )* <string.h>

■ **รูปแบบ**

```
char  *strstr(const char *s1, const char *s2);
```

■ **คำอธิบาย**

ฟังก์ชันนี้ใช้ในการค้นหาข้อความของสายอักขระ s2 ว่ามีอยู่ใน s1 หรือไม่ ถ้ามีฟังก์ชันก็จะให้พอยน์เตอร์ที่ชี้ไปยังที่อยู่ของข้อความที่ใน s1 ที่ตรงกับ s2 ถ้าไม่มีข้อความส่วนใดของ s1 ที่เหมือนกับ s2 ฟังก์ชันก็จะให้พอยน์เตอร์ศูนย์กลับคืน

■ **โปรแกรมโค้ดสำหรับฟังก์ชัน**

```
#include <string.h>

char  *strstr(const char *s1, const char *s2)
{
        register char c, sc;
        size_t len;

        if ((c = *find++) != 0) {
                len = strlen(s2);
                do {
                        do {
                                if ((sc = *s1++) == 0)
                                        return (NULL);
                        } while (sc != c);
                } while (strncmp(s1, s2, len) != 0);
                s1--;
        }
        return ((char *)s1);
}
```

■ **ตัวอย่างการใช้ฟังก์ชัน**

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

int main()
{
        char *s1 = "abdcacdcaacddabcbda";
        char *s2 = "aacd";
        char *p;

        p = strstr (s1, s2);
```

```
        p = strncpy (s1, p, 8);
        printf("%s\n", p);
        p[8] = '\0';
        printf("%s\n", p);
        return 0;
}
```

ฟังก์ชัน *strtok( )* <string.h>

■ **รูปแบบ**

**`char *strtok(char *s, const char *delim);`**

■ **คำอธิบาย**

ฟังก์ชันจะค้นหาสายอักขระที่ถูกแบ่งแยกโดยตัวอักขระตัวใดตัวหนึ่งที่กำหนดโดยพารามิเตอร์ `delim` เมื่อค้นพบตัวอักขระซึ่งทำหน้าที่เป็นตัวแบ่งแยก ฟังก์ชันก็จะเปลี่ยนค่าของตัวอักขระตัวนี้ให้เป็นศูนย์ และให้พอยน์เตอร์ที่ชี้ไปยังที่อยู่เริ่มต้นของสายอักขระ ตัวอย่างเช่น พารามิเตอร์ `s` มีค่าเท่ากับ

```
"aaa:bbb:ccc"
```

และ พารามิเตอร์ `delim` มีค่าเท่ากับ `":"` ฟังก์ชันจะเปลี่ยนสายอักขระเป็น

```
"aaa\0bbb:ccc"
```

และให้พอยน์เตอร์ที่ชี้ไปยังตัวอักขระ `'a'` ตัวแรก สำหรับการเรียกฟังก์ชันในครั้งต่อไป ถ้าพารามิเตอร์ตัวแรกเป็นพอยน์เตอร์ศูนย์ ฟังก์ชันก็จะเริ่มหาตัวอักขระแบ่งแยกในตำแหน่งถัดจากที่หาในคราวที่แล้ว คือ `'\0'` และเมื่อฟังก์ชันพบตัวอักขระ `':'` อีกครั้งก็จะเปลี่ยนสายอักขระ `s` ให้เป็น

```
"aaa\0bbb\0ccc"
```

และให้พอยน์เตอร์ที่ชี้ไปยังที่อยู่ของตัวอักขระ `'b'` ตัวแรก จากตัวอย่างถ้าฟังก์ชันไม่พบตัวอักขระแบ่งแยก `':'` ก็จะให้พอยน์เตอร์ที่ชี้ไปยังข้อความส่วนที่เหลือทั้งหมด สำหรับการเรียกใช้ ฟังก์ชันครั้งต่อๆไปที่ยังคงมีพารามิเตอร์ตัวแรกเป็นพอยน์เตอร์ศูนย์และถ้าฟังก์ชันค้นหาตัวอักขระมาถึงท้ายสุดของสายอักขระ `s` ก็จะให้พอยน์เตอร์ศูนย์กลับคืน โปรดสังเกตว่า การเรียกฟังก์ชันในครั้งใดๆที่มีพารามิเตอร์ตัวแรกเป็นพอยน์เตอร์ศูนย์ เราจะต้องแน่ใจว่า หน่วยความจำของสายอักขระ `s` จะต้องมีอยู่และไม่ถูกเปลี่ยนแปลงแก้ไขโดยขั้นตอนการทำงานอื่นๆ

■ **โปรแกรมโค้ดสำหรับฟังก์ชัน**

```
      #include <string.h>

      char *strtok(char *s, const char *delim)
      {
            register char *spanp;
            int c, sc;
            char *tok;
            static char *last;


            if (s == NULL && (s = last) == NULL)
                  return (NULL);
      cont:
            c = *s++;
            for (spanp = (char *)delim; (sc = *spanp++) != 0;) {
                  if (c == sc)
                        goto cont;
            }

            if (c == 0) { /* no non-delimiter characters */
                  last = NULL;
                  return (NULL);
            }
            tok = s - 1;

            for (;;) {
                  c = *s++;
                  spanp = (char *)delim;
                  do {
                        if ((sc = *spanp++) == c) {
                              if (c == 0)
                                    s = NULL;
                              else
                                    s[-1] = 0;
                              last = s;
                              return (tok);
                        }
                  } while (sc != 0);
      }
      /* NOT REACHED */
}
```

■ **ตัวอย่างการใช้ฟังก์ชัน**

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

      int main()
      {
            char *s = "aaa:bbb:ccc:ddd";
            char delimiter[] = ":";
            char *p;

            p = strtok(s, delimiter);
            printf("%s\n", p);

            while (p = strtok(NULL, delimiter))
                printf("%s\n", p);
            return 0;
      }
```

ฟังก์ชัน *strdup( )* <string.h>

■ **รูปแบบ**

```
char *strdup(const char *str);
```

■ **คำอธิบาย**

ฟังก์ชันจะทำสำเนาของสายอักขระที่เป็นพารามิเตอร์ของฟังก์ชันและให้พอยน์เตอร์ชี้ไปยังที่อยู่ของสายอักขระใหม่นี้ ถ้าไม่สำเร็จค่าที่ได้จะเป็นพอยน์เตอร์ศูนย์

■ **โปรแกรมโค้ดสำหรับฟังก์ชัน**

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

char *strdup(const char *str)
{
      size_t len;
      char *copy;

      len = strlen(str) + 1;
      if (!(copy = malloc(len)))
            return((char *)NULL);
      memcpy(copy, str, len);
      return(copy);
}
```

■ **ตัวอย่างการใช้ฟังก์ชัน**

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

int main()
{
      char *src = "Hello World!";
      char *new_str;

      new_str = strdup(src);
      printf("%s\n", new_str);
      free(new_str);
      return 0;
}
```

ฟังก์ชัน *bzero( )* <string.h>

■ **รูปแบบ**

```
void bzero (void *b, size_t length);
```

■ **คำอธิบาย**

ฟังก์ชันนี้ใช้สำหรับการกำหนดค่าของข้อมูลทุกตัวในอาร์เรย์ หรือสายอักขระให้มีค่าเป็นศูนย์ ขึ้นอยู่กับความยาวที่กำหนดโดยพารามิเตอร์ `length`

■ **โปรแกรมโค้ดสำหรับฟังก์ชัน**

```
void bzero (void *b, size_t length)
{
      register char *p;
      for (p = (char *)b; length--; *p++ = '\0') ;
}
```

ฟังก์ชัน *bcmp( )* <string.h>

■ **รูปแบบ**

```
int bcmp(const void *b1,const void *b2,size_t n)
```

■ **คำอธิบาย**

ฟังก์ชันนี้ เปรียบเทียบไบต์แต่ละตัว จำนวน `n` ตัว (มีค่ามากกว่าหรือเท่ากับศูนย์) จากบล็อกหน่วยความจำสองแห่ง ซึ่งที่อยู่ของบล็อกหน่วยความจำแต่ละแห่งจะถูกกำหนดโดยพอยน์เตอร์อเนกประสงค์ `b1` และ `b2` ตามลำดับ ฟังก์ชันจะคืนค่าที่เป็นข้อมูลแบบ `int` ซึ่งจะเท่ากับจำนวนของไบต์จากสองบริเวณที่แตกต่างกันแต่จะมีค่าไม่เกิน `n` ถ้าค่ากลับคืนมีค่าเท่ากับศูนย์ หมายความว่าบริเวณหน่วยความจำทั้งสองแห่งมีข้อมูล `n` ไบต์แรกที่เหมือนกันทั้งหมด

■ **โปรแกรมโค้ดสำหรับฟังก์ชัน**

```
#include <string.h>

int bcmp(const void *b1, const void *b2, size_t n)
{
      char *p1, *p2;
```

```
        if (!n)
              return(0);
        p1 = (char *)b1;
        p2 = (char *)b2;

        do {
              if (*p1++ != *p2++)
                    break;
        } while (--n);
        return(n);
  }
```

# แบบฝึกหัดท้ายบท

1. กำหนดให้ a และ p เป็นอาร์เรย์ และพอยน์เตอร์ตามลำดับ ซึ่งนิยามไว้ดังนี้

```
int a[]={1,-4,4,3,10,25,19,0,2};
int *p = &a[3];
```

จงหาค่าของนิพจน์ต่อไปนี้

```
1) *(p+2)
2) p[-2]
3) *(&a[0] - p)
4) a[*p + 1]
5) *(a+a[0])
```

2. นิพจน์ใดต่อไปนี้หมายถึง `a[i][j]`

1) `*(a[i] + j)`
2) `(*(a+i))[j]`
3) `**(a[i+j])`
4) `**(a+i) + j`
5) `*((*(a+i)) + j)`
6) `*(&a[0][0] + i + j)`

3. จงอธิบายความแตกต่างระหว่าง

```
char s[255];
```

และ

```
char *s;
```

4. ถ้ากำหนดให้อาร์เรย์ a แบบ `double` เก็บค่าสัมประสิทธิ์แต่ละตัวของโพลิโนเมียล `p(x)`

$$p(x) = a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1 x + a_0 = \sum_{i=0}^{n} a[i] \cdot x^i$$
$$= (((\ldots((a_n \cdot x + a_{n-1}) \cdot x + a_{n-2}) \cdot x + \ldots))) \cdot x + a_0$$

จงเขียนฟังก์ชันที่ใช้ในการคำนวณค่าของโพลิโนเมียลในตำแหน่งของ x เมื่อกำหนดให้ x เป็นพารามิเตอร์หนึ่งของฟังก์ชันนี้

```
double PolyEvaluate(double a[], unsigned n, double x);
```

5. จงเขียนฟังก์ชันที่ใช้ในการบวกหรือลบโพลิโนเมียลที่เก็บอยู่ในอาร์เรย์ ซึ่งมีรูปแบบดังต่อไปนี้

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \ldots + a_1 x + a_0$$
$$q(x) = b_n x^n + b_{n-1} x^{n-1} + b_{n-2} x^{n-2} + \ldots + b_1 x + b_0$$

โดยที่ p และ q เป็นโพลิโนเมียลของดีกรีเท่ากับ n ซึ่ง n เป็นจำนวนเต็มบวกหรือศูนย์เท่านั้น สำหรับการเก็บค่าสัมประสิทธิ์แต่ละตัวของโพลิโนเมียลลงในอาร์เรย์ เราจะใช้รูปแบบต่อไปนี้

```
#define  MAX_DEGREE   20
typedef  double Polynomial[MAX_DEGREE];
```

ดังนั้น n จะมีค่าเท่ากับ 20 และฟังก์ชันที่ใช้ในการบวกหรือลบโพลิโนเมียล p และ q ควรจะมีรูปแบบดังนี้

```
void PolyAdd( Polynomial p, Polynomial q,
              Polynomial sum);
void PolySub( Polynomial p, Polynomial q,
              Polynomial diff);
```

6. จงเขียนโปรแกรมที่ใช้ฟังก์ชันต่อไปนี้ในการเรียงข้อมูลในอาร์เรย์แบบ int จำนวน n ตัว

```
void shell_sort (int a[],  unsigned  int n)
{
      unsigned int i, j, x;
      int tmp;

      for (x=n/2; x>0; x = (x==2) ? 1 : x/2.2)
         for (i=x; i < n; i++)
         {
            tmp = a[i];
            for(j = i; j>=x && tmp<a[j-x]; j -=x)
                  a[j] = a[j-x];
            a[j] = tmp;
         }
}
```