หัวข้อในบทที่ 6

- อะไรคือพอยน์เตอร์
- พอยน์เตอร์ศูนย์ (Null Pointer)
- การใช้พอยน์เตอร์แบบ void
- เลขคณิตตัวชี้
- ขนาดหน่วยความจำของพอยน์เตอร์
- การใช้ const กำกับพอยน์เตอร์
- วิธีการแจ้งใช้พอยน์เตอร์ที่ชี้ไปยังฟังก์ชัน
- การสร้างฟังก์ชันที่ให้ที่อยู่ของฟังก์ชันกลับคืน
- การใช้ typedef
- สร้างแบบของพอยน์เตอร์

6

พอยน์เตอร์

สำหรับตัวแปรในภาษาซีนั้นจะมีคุณสมบัติหรือข้อมูลที่เกี่ยวข้องกับตัวแปรคือ ค่าที่ตัวแปรเก็บไว้ (Value) และที่อยู่ของตัวแปรในหน่วยความจำ (Address) ตามปรกติแล้วเราคุ้นเคยกับการอ่านหรือกำหนดค่าของตัวแปรโดยตรง แต่ตอนนี้เราลองมาทำความรู้จักกับโอเปอร์เรเตอร์อีกตัวหนึ่งที่เราใช้ในการหาที่อยู่ของตัวแปรใดๆในหน่วยความจำหลักของคอมพิวเตอร์ เรียกว่า *แอดเดรสออฟ* (Address-Of) โดยใช้สัญลักษณ์ & (Ampersand) วางไว้ข้างหน้าตัวแปรใดๆที่เราต้องการหาที่อยู่ในหน่วยความจำ (โปรดอย่าสับสนกับโอเปอร์เรเตอร์ที่ใช้เปรียบเทียบระหว่างบิตแบบ 'และ' ที่เราได้ทำความรู้จักไปในบทที่สอง) เพราะตัวแปรทุกตัวที่ใช้หน่วยความจำหลักของคอมพิวเตอร์จะต้องมีที่อยู่ไว้เก็บข้อมูล

ที่อยู่ของตัวแปรคือหมายเลขของหน่วยความจำขนาดหนึ่งไบต์ ถ้าตัวแปรใดๆต้องการหน่วยความจำมากกว่าหนึ่งไบต์ ขึ้นไป ที่อยู่ของตัวแปรก็คือหมายเลขของไบต์เริ่มต้นที่ตัวแปรใช้ และเมื่อเราพูดถึงที่อยู่ของตัวแปรใดๆ ก็จะหมายถึงไบต์เริ่มต้นนั้นเอง ตัวอย่างการแรกสำหรับการใช้งานโอเปอร์เตอร์ & มีดังนี้

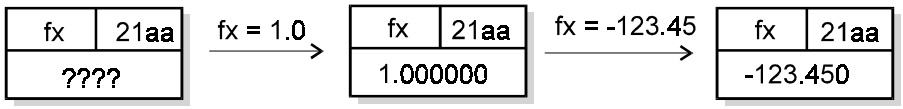
```
#include <stdio.h>

int main()
{
     float  fx;
```

```
        fx = 1.0;
        printf ("Address = %p, Value = %f\n", &fx, fx);
        fx = -123.45;
        printf ("Address = %p, Value = %8.3f\n", &fx, fx);
        return 0;
}
```

ผลการทำงานของโปรแกรมที่แสดงออกทางจอภาพ เช่น

```
Address = 21AA, Value = 1.000000
Address = 21AA, Value = -123.450
```

จากตัวอย่างนี้จะเห็นได้ว่า ตัวแปร `fx` มีที่อยู่ของหน่วยความจำในเลขที่ `21aa` (เลขฐานสิบหก) ซึ่งใช้เก็บค่าของข้อมูลแบบ `float` โปรดสังเกตว่า ที่อยู่ของตัวแปรใดๆที่ได้จากโอเปอร์เรเตอร์ `&` นั้นจะเป็นข้อมูลแบบ `(unsigned) int` ในกรณีนี้ `fx` มีขนาด 4 ไบต์ ดังนั้นหน่วยความจำขนาดหนึ่งไบต์ตั้งแต่หมายเลข `21aa` จนถึง `21ad` จะใช้สำหรับการเก็บข้อมูลของตัวแปรแบบ `float` นี้



เมื่อเราได้แจ้งใช้ตัวแปร `fx` แต่ยังมิได้ติดตั้งค่าใดๆให้ตัวแปรนี้ ค่าของตัวแปรในหน่วยความจำหมายเลข `21aa` จึงเป็นเท่าไหรก็ได้ แต่เราตั้งเงื่อนไขไว้ว่าเราจะอ่านข้อมูลที่เก็บไว้ในตัวแปรนี้และนำไปใช้ก็ต่อเมื่อเราได้กำหนดค่าให้ตัวแปรแล้วอย่างน้อยครั้งหนึ่ง เมื่อได้กำหนดให้ตัวแปรมีค่า `1.0` เป็นครั้งแรกแล้วค่าของตัวแปรจึงเป็นค่าใหม่นี้จนกว่าจะได้มีการกำหนดค่าใหม่อีกค่าให้ตัวแปร การอ่านและกำหนดค่าของตัวแปรจึงเป็นไปโดยตรงตามที่เราคุ้นเคย และเราไม่จำเป็นต้องทราบที่อยู่ของตัวแปรนี้เลย แต่อย่างไรก็ตาม ถ้าเราทราบที่อยู่ของตัวแปรใดๆเราก็สามารถเข้าถึงข้อมูลตามที่อยู่นั้นได้โดยทางอ้อมหรือแม้กระทั่งเปลี่ยนแปลงค่าของตัวแปรนั้นก็ย่อมทำได้และไม่จำเป็นต้องผ่านตัวแปรซึ่งเป็นเจ้าของที่อยู่ วิธีการเข้าถึงตัวแปรและข้อมูลที่เก็บไว้ในตัวแปรโดยทางอ้อมจะอาศัยหลักการทำงานขององค์ประกอบพื้นฐานสำคัญของภาษาซีที่เรียกว่า *เลขคณิตตัวชี้* (Pointer Arithmetic)

6.1 อะไรคือพอยน์เตอร์

พอยน์เตอร์ (Pointer) หรือเรียกเป็นภาษาไทยได้ว่า *ตัวชี้* คือ ตัวแปรที่ทำหน้าที่เก็บ "ที่อยู่" ของตัวแปรอื่นๆในหน่วยความจำหรือหมายเลขของหน่วยความจำในคอมพิวเตอร์ ดังนั้นเราจะเรียกตัวแปรที่ทำ

หน้าที่ชี้ไปยังที่อยู่ของตัวแปรอื่นๆ ว่า *ตัวแปรพอยน์เตอร์* (Pointer Variable) และมีคุณสมบัติที่สำคัญคือ สามารถใช้หาค่าและที่อยู่ของตัวแปรตามที่อยู่ที่พอยน์เตอร์กำลังชี้ไปได้ การแจ้งใช้ตัวแปรพอยน์เตอร์นั้นจะแตกต่างจากการแจ้งใช้ตัวแปรแบบธรรมดาดังนี้

```
data_type  * pointer_name;
```

ซึ่งหมายความว่าตัวแปรพอยน์เตอร์นี้ทำหน้าที่ชี้ไปยังตัวแปรตัวอื่นๆที่มีแบบข้อมูลตามที่กำหนดไว้เท่านั้น เช่น ตัวแปรพอยน์เตอร์ที่ชี้ไปยังแหล่งข้อมูลแบบ `int` เท่านั้น หรืออาจจะกำหนดให้เป็นแบบข้อมูลใดๆก็ได้ ตัวอย่างการแจ้งใช้ ตัวแปรพอยน์เตอร์

```
int    *iptr;
char   *cptr;
double *dptr;
void   *ptr;
```

เวลาเราแจ้งใช้ตัวแปรที่ทำหน้าที่เป็นพอยน์เตอร์และตัวแปรธรรมดาภายในบรรทัดเดียวกัน เราก็ควรจะระมัดระวังด้วย ตัวอย่างข้างล่างนี้แสดงให้เห็นความแตกต่างระหว่าง

```
double *x, y, z;
```

และ

```
double *x, *y, *z;
```

สำหรับแบบแรก เราแจ้งใช้ตัวแปรสามตัวแบบ `double` แต่มีตัวแปร `x` เท่านั้นที่ทำหน้าที่เป็นพอยน์เตอร์ในขณะที่แบบที่สองเป็นการแจ้งใช้ตัวแปรสามตัวเช่นกัน แต่ทั้งสามตัวแปรทำหน้าที่เป็นพอยน์เตอร์

ตัวอย่างที่แสดงให้เห็นการทำงานของพอยน์เตอร์และการใช้พอยน์เตอร์สำหรับการเข้าถึงค่าของข้อมูลที่เก็บไว้ในตัวแปรแบบทางอ้อม (Indirect Access)

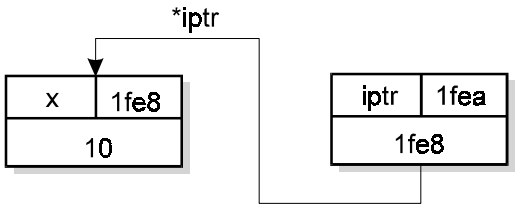
```
#include <stdio.h>

int main()
{
     int *iptr;
     int x = 10;
     iptr = &x;

     printf("x    : Address = %p, Value = %d\n",
            &x, x);
     printf("iptr : Address = %p, Value = %p\n",
            &iptr, iptr);
     printf("---------------------------------\n");
```

```
        printf("iptr points to the address = %p\n", iptr);
        printf("Value at this address = %d\n", *iptr);
        return 0;
}
```

ผลของโปรแกรมคือ

```
x    : Address = 1fe8, Value = 10
iptr : Address = 1fea, Value = 1fe8
---------------------------------
iptr points to the address = 1fe8
Value at this address = 10
```

เราจะเห็นได้ว่า ตัวแปร x เป็นตัวแปรแบบ int ธรรมดา ตัวแปร x นี้มีหน่วยความจำตามที่อยู่ 1fe8 และที่อยู่นี้เก็บข้อมูลแบบ int ที่มีค่าเท่ากับ 10 ส่วน iptr เป็นตัวแปรพอยน์เตอร์ เนื่องจากว่า iptr เป็นตัวแปรชนิดหนึ่งดังนั้นเราก็สามารถหาที่อยู่ของ iptr ในหน่วยความจำได้โดยใช้โอเปอร์เรเตอร์ & และเราจะเห็นได้ว่า iptr มีที่อยู่ที่ตำแหน่ง 1fea ในหน่วยความจำและในที่อยู่นั้นเก็บค่าของที่อยู่ของตัวแปร x ซึ่งมีค่าเท่ากับ 1fe8



ดังนั้นเราจึงกล่าวได้ว่า ตัวแปร iptr ทำหน้าที่ชี้ไปยังที่อยู่ของตัวแปร x เราสามารถอ่านค่าของตัวแปร x ทางอ้อมได้โดยอาศัยพอยน์เตอร์ iptr และใช้เครื่องหมายโอเปอร์เรเตอร์ * ที่วางไว้หน้าตัวแปรพอยน์เตอร์ ซึ่งโอเปอร์เรเตอร์นี้มีชื่อว่า *Indirection Operator* และเราเรียกขั้นตอนนี้ว่า *Pointer Dereferencing* ดังนั้น *iptr จึงหมายถึงการอ่านค่าของข้อมูลที่เก็บอยู่ในที่อยู่ที่พอยน์เตอร์นี้ชี้ไป (ซึ่งก็คือค่าของตัวแปร x นั้นเอง)

โปรดสังเกตว่า เรานิยมใช้ %p ร่วมกับฟังก์ชัน printf() ในการจองที่ไว้สำหรับค่าที่เป็นที่อยู่ของตัวแปรที่เป็นพารามิเตอร์ของฟังก์ชัน หรือเราอาจจะใช้ %u หรือ %x ก็ได้

นิพจน์	*คำอธิบาย*
`iptr`	(ชื่อ)ตัวแปรที่ทำหน้าที่เป็นพอยน์เตอร์ นิพจน์นี้ให้ค่าแบบ `int` หรือ `unsigned int` ซึ่งเป็นที่อยู่ที่พอยน์เตอร์นี้ชี้ไป
`&iptr`	นิพจน์นี้บอกที่อยู่ของตัวแปรพอยน์เตอร์ `iptr` ในหน่วยความจำ (ที่อยู่ที่ได้จะเป็นข้อมูลแบบ `int` หรือ `unsigned int`)
`*iptr`	นิพจน์นี้ให้ค่าของข้อมูลที่เก็บอยู่ในที่อยู่ที่ตัวแปรพอยน์เตอร์ `iptr` กำลังชี้ไป

ตารางแสดงรูปแบบของนิพจน์ที่เกี่ยวข้องกับพอยน์เตอร์ ที่ได้แจ้งใช้โดย **`int * iptr;`**

จากตารางเราจะเห็นได้ว่าตัวแปรพอยน์เตอร์มีนิพจน์ที่เกี่ยวข้องถึงสามรูปแบบ ในขณะที่ตัวแปรธรรมดาไม่สามารถใช้งานร่วมกับโอเปอร์เรเตอร์ `*` ที่วางไว้ข้างหน้าได้

เราลองมาพิจารณาตัวอย่างของฟังก์ชันที่มีพารามิเตอร์เป็นพอยน์เตอร์ ที่ชี้ไปยังตัวแปรแบบ `char`

```
int func(char *s)
{
   static char local_stat = 'A';
   char *p;
   char *t = s;

   p = t;

   s = &local_stat;

   return (int)(*p)*(*p);
}
```

ในการนิยามฟังก์ชัน เรากำหนดให้มีพารามิเตอร์ `s` ทำหน้าที่เป็นพอยน์เตอร์ที่ชี้ไปยังข้อมูลแบบ `char` ภายในฟังก์ชันเราก็ได้แจ้งใช้ตัวแปรหลายตัว เช่น `p` เป็นพอยน์เตอร์ที่ชี้ไปยังข้อมูลแบบ `char` นอกจากนั้นเราก็ได้แจ้งใช้ตัวแปรพอยน์เตอร์ `t` พร้อมกับกำหนดค่าเริ่มต้นของ `t` ให้เป็นค่าเดียวกันกับค่าของ `s` แต่ในขั้นตอนการแจ้งใช้ เรามิได้กำหนดว่า ตัวแปรพอยน์เตอร์ `p` นี้ควรจะชี้ไปที่ใด ซึ่งหมายความว่า เรายังมิได้กำหนดค่าใดๆให้แก่ `p` ในบรรทัดคำสั่งที่ตามมาเราได้กำหนดให้ `p` มีค่าเท่ากับค่าของ `t` (โปรดจำไว้ว่าค่าของพอยน์เตอร์คือหมายเลขที่อยู่ของหน่วยความจำ) ดังนั้นเราสามารถกล่าวได้ว่า เมื่อโปรแกรมทำงานมาถึงขั้นตอนนี้ พอยน์เตอร์ `s`, `t` และ `p` ต่างก็ชี้ไปยังแหล่งข้อมูลเดียวกัน เช่น สมมุติว่าในโปรแกรมมีคำสั่งที่เรียกใช้ฟังก์ชัน `func()` ดังนี้

```
char ch = 'Z';
func(&ch);
```

พอยน์เตอร์ s, t และ p จะชี้ไปยังที่อยู่ของตัวแปร ch ที่กำลังเก็บค่า 'Z' ไว้ในหน่วยความจำของตน

ในบรรทัดถัดไปเรากำหนดให้ s ชี้ไปยังที่อยู่ของตัวแปรสถิตชื่อ local_stat แบบ char ดังนั้นค่าของ s จึงเปลี่ยนไปจากค่าเดิม ซึ่งค่าเดิมของ s คือหมายเลขที่อยู่ของตัวแปร ch (โปรดสังเกตว่าค่าของ s ในตอนแรกที่ได้จากการเรียกใช้ฟังก์ชันจะเป็นสำเนาของหมายเลขที่อยู่ของหน่วยความจำสำหรับตัวแปร ch และเก็บอยู่ในรูปของพอยน์เตอร์) แต่มิได้หมายความว่าค่าหรือหมายเลขที่อยู่ของตัวแปร ch จะเปลี่ยนแปลงไป

ในบรรทัดคำสั่งสุดท้ายของฟังก์ชันเราได้อ่านค่าของนิพจน์ *p และเนื่องจากว่าพอยน์เตอร์ p ยังคงชี้ไปยังที่อยู่ของตัวแปร ch ค่าของนิพจน์ นี้จึงมีค่าเท่ากับค่าของ ch คือ 'Z' มีค่าเท่ากับ 90 ในระบบเลขฐานสิบ และจะถูกแปลงเป็น int ดังนั้นค่ากลับคืนของฟังก์ชัน func() คือ ผลคูณของ *p และ *p ซึ่งในกรณีนี้จะมีค่าเท่ากับ 8100

เราไม่สามารถเปลี่ยนแปลงที่อยู่ของตัวแปรแต่ละตัวในหน่วยความจำได้ ตัวอย่างการใช้โอเปอร์เรเตอร์ & ที่ไม่ถูกต้องเช่น

```
char ch1 = 'A';
char ch2 = 'B';
char *p  = &ch2;
&ch1 = p;
```

ในกรณีนี้ เราพยายามหาที่อยู่ของตัวแปร ch2 และ ch1 ตามลำดับโดยใช้โอเปอร์เรเตอร์ & แต่เนื่องจากว่าโอเปอร์เรเตอร์ให้ค่าคงที่ และเราก็ไม่สามารถเปลี่ยนแปลงค่าคงที่ได้ ซึ่งหมายความว่า นิพจน์ที่อยู่ทางซ้ายของเครื่องหมาย = จะต้องไม่ใช่ค่าคงที่ ดังนั้นประโยคคำสั่งบรรทัดสุดท้ายจึงไม่ถูกต้อง

ปัญหาที่มักพบได้บ่อยสำหรับผู้ที่เริ่มทำความเข้าใจหลักการทำงานของพอยน์เตอร์เราสามารถเห็นได้จากตัวอย่างต่อไปนี้

```
/* 1*/    {
/* 2*/         int x = 0x3f10, y;
/* 3*/         int *iptr = x;
/* 4*/
/* 5*/         &iptr = 100;
/* 6*/         {
/* 7*/              int local_1 = -101;
/* 8*/              iptr = &local_1;
```

```
/* 9*/          }
/*10*/          {
/*11*/               int local_2 = -102;
/*12*/          }
/*13*/          y = *iptr;
/*14*/
/*15*/          {
/*16*/               static int stat_local;
/*17*/               iptr = &stat_local;
/*18*/          }
/*19*/          *iptr = 1;
/*20*/
/*21*/     }
```

เราแจ้งใช้พอยน์เตอร์ iptr ที่ใช้เก็บที่อยู่ของตัวแปรใดๆแบบ int แต่ตามประโยคคำสั่งในบรรทัดที่ 3 เราไม่ได้ผ่านที่อยู่ของ x แต่ผ่านค่าของ x เป็นค่าเริ่มต้นของพอยน์เตอร์ iptr แม้ว่าจะไม่ผิดหลักไวยากรณ์สำหรับบางคอมไพล์เลอร์ แต่จะทำให้เกิดปัญหาขึ้นได้เพราะพอยน์เตอร์นี้ มิได้ชี้ไปยังที่อยู่ของ x แต่อย่างใด แต่กลับชี้ไปยังที่อยู่หมายเลข 3f10 ซึ่งเป็นค่าของตัวแปร x นั้นเอง ถ้าเราต้องการจะกำหนดให้ iptr ชี้ไปยังที่อยู่ของ x เราต้องเขียนว่า

```
/* 3*/          int *iptr = &x;
```

โปรดสังเกตว่า เครื่องหมาย * ที่วางไว้หน้าชื่อของตัวแปรในบรรทัดที่เราแจ้งใช้ตัวแปรนั้นจะมีความหมายแตกต่างจากโอเปอร์เรเตอร์ * ที่วางไว้หน้าชื่อของตัวแปรในบรรทัดคำสั่งทั่วๆไป ในบรรทัดที่เราแจ้งใช้ตัวแปร เครื่องหมาย * จะแจ้งให้คอมไพล์เลอร์ทราบว่า เราต้องการแจ้งใช้ตัวแปรที่ทำหน้าที่เป็นพอยน์เตอร์ ส่วนโอเปอร์เรเตอร์ * ที่วางไว้หน้าชื่อของตัวแปรในบรรทัดคำสั่งทั่วไปจะหมายถึงการเข้าถึงแหล่งข้อมูลที่พอยน์เตอร์กำลังอ้างอิงถึง

ในบรรทัดที่ 5 เป็นคำสั่งที่ไม่ถูกต้อง เพราะนิพจน์ที่อยู่ทางซ้ายของเครื่องหมายเท่ากับเป็นนิพจน์ที่ให้ค่าคงที่ ซึ่งก็คือที่อยู่ของพอยน์เตอร์ iptr ดังนั้นเราไม่สามารถกำหนดหรือเขียนทับค่าของนิพจน์นี้ได้ ในบรรทัดที่ 8 เราได้กำหนดให้พอยน์เตอร์ iptr ชี้ไปยังที่อยู่ของตัวแปรชั่วคราว local_1 ซึ่งอยู่ภายในบล็อกย่อย ตามที่เราได้เรียนรู้ไปแล้วตัวแปรภายในบล็อกจะถูกทำลายไปเมื่อโปรแกรมจบการทำงานของบล็อก ดังนั้นในบรรทัดที่ 13 จะก่อให้เกิดปัญหาขึ้นได้ เพราะเราใช้ iptr ซึ่งเก็บที่อยู่ของตัวแปร local_1 ในการเข้าถึงค่าของตัวแปร local_1 โดยทางอ้อม แต่ตัวแปรภายในนี้จะมีตัวตนก็ต่อเมื่อบล็อก ของตัวแปรนี้กำลังทำงาน แต่เนื่องจากว่าคำสั่งในบรรทัดที่ 13 อยู่นอกบล็อกของตัวแปร local_1 ผลก็คือว่า ค่าของ y ที่ได้จะมิใช่ค่าของตัวแปร local_1 แต่อย่างไร เพราะหน่วยความจำตามที่อยู่ดังกล่าวใน ช่วงเวลานั้นไม่ใช่ของ local_1 อีกต่อไปและคอมพิวเตอร์ก็อาจจะจัดสรรที่อยู่นั้นให้กับตัวแปรชั่วคราวตัวอื่นๆก็ได้ เช่น local_2 หลังจากที่จบการทำงานของบล็อกที่มีตัวแปร local_1

ในบรรทัดที่ 16 และ 17 มีการแจ้งใช้ตัวแปรภายในแบบสถิต stat_local และกำหนดให้พอยน์เตอร์ iptr ชี้ไปยังที่อยู่ของตัวแปรสถิตนี้ เมื่ออยู่นอกบล็อกการทำงานย่อยตัวแปรสถิตก็คงมีตัวตนอยู่ ในบรรทัดที่ 19 เราได้กำหนดค่าของตัวแปรสถิตที่อยู่ภายในบล็อกโดยทางอ้อมโดยอาศัยพอยน์เตอร์ ซึ่งถ้าเราไม่ใช้พอยน์เตอร์เข้าช่วย เราก็ไม่สามารถเข้าถึงตัวแปรสถิตย์นี้ได้ถ้าอยู่ภายนอกบล็อกการทำงานของตัวแปรตัวนี้

ตัวอย่างต่อไปที่แสดงให้เห็นการใช้พอยน์เตอร์ที่ผิด คือ

```
#include <stdio.h>

int main()
{
     int x, y;
     int *iptr;

     printf("Please enter an integer : ");
     scanf("%d", &x);
     printf("Number = %d", x);

     printf("Please enter an integer : ");
     scanf("%d", iptr);
     printf("Number = %d", *iptr);

     return 0;
}
```

โปรแกรมตัวอย่างนี้มีที่ผิดคือ เราได้ใช้พอยน์เตอร์ iptr ในการอ่านข้อมูลที่ได้จากการทำงานของฟังก์ชัน scanf() ซึ่งมีหน้าที่อ่านข้อมูลที่ถูกป้อนให้คอมพิวเตอร์โดยผู้ใช้ผ่านทางแป้นพิมพ์ เมื่อฟังก์ชันอ่านข้อมูลได้แล้ว ในกรณีนี้คือตัวเลขจำนวนเต็มแบบ int ก็จะเก็บข้อมูลนี้ไว้ในหน่วยความจำตามที่อยู่ที่ตัวแปรพอยน์เตอร์ iptr กำลังชี้ไป แต่เนื่องจากว่าเรามิได้ติดตั้งค่า (ที่อยู่) เริ่มต้นให้แก่ iptr ดังนั้นเราจึงไม่รู้แน่ชัดว่าพอยน์เตอร์นี้ชี้ไปที่ใด และก็อาจจะเป็นไปได้ว่าพอยน์เตอร์นี้ชี้ไปยังที่อยู่ที่เราไม่ได้จองหน่วยความจำไว้ ปัญหาก็จะเกิดขึ้นตามมา ดังนั้นเราจะต้องกำหนดค่าของตัวแปรพอยน์เตอร์ก่อน และจะต้องเป็นที่อยู่ของหน่วยความจำที่ถูกต้องที่เราต้องการใช้จริง เช่น เราอาจจะให้ iptr เก็บที่อยู่ของตัวแปร y ก่อนที่จะเรียกใช้ฟังก์ชัน scanf()

```
     iptr = &y;
     scanf("%d", iptr);
```

และค่าที่ได้ก็จะถูกเก็บไว้ในที่อยู่ของตัวแปร y

สำหรับการแจ้งใช้และติดตั้งค่าของพอยน์เตอร์ในบรรทัดคำสั่งเดียวกันนั้น เราก็ต้องควรระวังเพราะต้องแน่ใจว่า เราต้องการให้พอยน์เตอร์ชี้ไปยังที่อยู่ของตัวแปรใดๆที่ได้ถูกสร้างขึ้นแล้ว ตัวอย่างที่ผิดเช่น

```
double *dptr = &x, x =10.00;
```

ในกรณีนี้เราแจ้งใช้พอยน์เตอร์ dptr และกำหนดให้พอยน์เตอร์ชี้ไปยังที่อยู่ของตัวแปร x แต่เนื่องจากเราได้แจ้งใช้ตัวแปร x หลังจากการติดตั้งค่าของพอยน์เตอร์ ดังนั้นจึงผิด และที่ถูกต้องจะต้องเขียนใหม่เป็น

```
double  x =10.00, *dptr = &x;
```

6.2 พอยน์เตอร์ศูนย์ (Null Pointer)

ถ้าเราต้องการกำหนดให้พอยน์เตอร์ชี้ไปยังที่ใดที่หนึ่งที่เราไม่สามารถใช้โอเปอร์เรเตอร์ * และเข้าถึงแหล่งข้อมูลในที่อยู่นั้นได้ เราก็จะกำหนดให้พอยน์เตอร์มีค่าเท่ากับ 0 และเราเรียกพอยน์เตอร์ใดๆที่มีค่าเท่ากับศูนย์ว่า *พอยน์เตอร์ศูนย์* (Null Pointer) การกำหนดให้พอยน์เตอร์เป็นพอยน์เตอร์ศูนย์ก็เพื่อบ่งบอกว่าเราไม่ต้องการให้พอยน์เตอร์นี้ชี้ไปยังที่อยู่ใดๆในหน่วยความจำ เพราะถ้าค่าของพอยน์เตอร์ไม่เท่ากับศูนย์ก็หมายความว่าพอยน์เตอร์สามารถชี้ไปยังหน่วยความจำที่เราเข้าถึงได้ แต่ถ้าเราไม่ต้องการให้พอยน์เตอร์ชี้ไปยังที่ใดๆเราก็กำหนดค่าของพอยน์เตอร์ให้เป็นศูนย์หรือกล่าวได้ว่าพอยน์เตอร์ชี้ไปยังหน่วยความจำหมายเลขศูนย์ ตัวอย่างเช่น

```
char ch ='A', *ptr;
int x;

ptr = &ch;
x = (int)*ptr;    /* O.K. */

ptr = 0;
x = (int)*ptr;    /* ILLEGAL */
```

ในภาษาซีได้มีการนิยามสัญลักษณ์ค่าคงที่ NULL ที่นิยามไว้ในแฟ้ม <stdio.h> ซึ่งหมายถึง 0 นั้นเอง และจะใช้กับพอยน์เตอร์ศูนย์ เช่น

```
(ptr == NULL)
```

นิพจน์นี้จะมีค่าเป็นจริง ถ้า ptr เป็นพอยน์เตอร์ศูนย์ (พอยน์เตอร์มีค่าเท่ากับศูนย์)

สำหรับพอยน์เตอร์ใดๆก็ตามเราสามารถกำหนดค่าให้เป็นศูนย์หรือ NULL ได้ โดยไม่ต้องคำนึงถึงแบบของพอยน์เตอร์

6.3 การใช้พอยน์เตอร์แบบ void

การใช้พอยน์เตอร์ที่ชี้ไปยังข้อมูลแบบ void หรือที่เราเรียกว่า Void Pointer ตามปรกติแล้วเราจะไม่ถือว่า void เป็นแบบข้อมูลพื้นฐานตามที่เราเข้าใจ เช่น ถ้าเราแจ้งใช้ตัวแปรในลักษณะนี้

```
void x;
```

จะถือว่าผิดหลักไวยากรณ์ ในอีกกรณีหนึ่ง ถ้าเราใช้ void ในการนิยามฟังก์ชัน เช่น

```
void func (void);
```

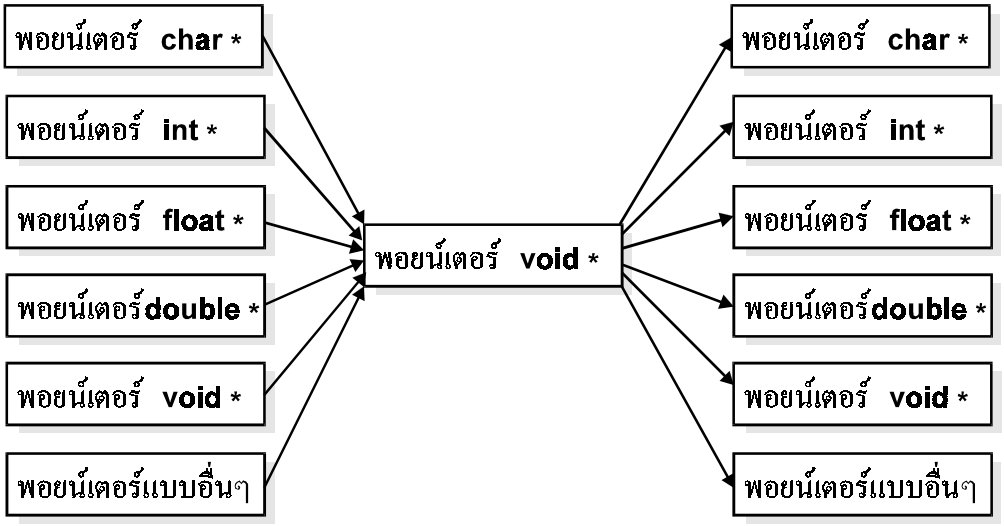
ก็หมายความว่า ฟังก์ชัน func() ไม่ต้องการพารามิเตอร์ใดๆและไม่ให้ค่าใดๆกลับคืนเมื่อยามเรียกใช้การ ใช้คำว่า void ร่วมกับการแจ้งใช้พอยน์เตอร์ โดยมองว่า void เป็นแบบข้อมูลพื้นฐานชนิดหนึ่งก็จะมีความหมายที่แตกต่างออกไป เช่น สมมุติว่าเราแจ้งใช้พอยน์เตอร์ที่มีชื่อว่า vptr

```
void *vptr;
```

จะหมายความว่า พอยน์เตอร์ vptr สามารถชี้ไปยังแหล่งข้อมูลแบบใดก็ได้หรือเราอาจจะกล่าวไว้ว่าพอยน์เตอร์ตามรูปแบบนี้จะใช้งานเป็น *พอยน์เตอร์อเนกประสงค์* โดยไม่ขึ้นอยู่กับแบบของข้อมูลตามที่อยู่ที่พอยน์เตอร์นี้กำลังอ้างถึง แต่อย่างไรก็ตามเราไม่สามารถใช้โอเปอร์เรเตอร์ * กับพอยน์เตอร์แบบ void ได้เพราะข้อมูลแบบ void ไม่ถือว่าเป็นแบบข้อมูลชนิดหนึ่งของภาษาซี ตัวอย่างที่ผิด เช่น

```
*vptr = 1;
```

ดังที่กล่าวไป เราสามารถผ่านค่าของพอยน์เตอร์ชนิดอื่นๆให้พอยน์เตอร์แบบ void ได้ และในทางกลับกันเราก็สามารถผ่านค่าของพอยน์เตอร์แบบ void ให้พอยน์เตอร์ชนิดอื่นๆได้ แต่ควรจะใช้วิธีการแปลงแบบของพอยน์เตอร์โดยเจาะจง (Type Casting)



ตัวอย่างการใช้พอยน์เตอร์แบบ void ร่วมกับพอยน์เตอร์แบบอื่น

```
/* 1 */ #include <stdio.h>
/* 2 */
/* 3 */ int main()
/* 4 */ {
/* 5 */   void *ptr;   /* Generic Pointer */
/* 6 */   short i = 1, j, *sp = &i;
/* 7 */   long  x = 0x10ff33efL, *lp;
/* 8 */
/* 9 */   ptr = sp;
/*10 */   j = *((short *)ptr);
/*11 */   printf("i = %d, j = %d\n", i, j);
/*12 */
/*13 */   ptr = &x;
/*14 */   lp = (long *)ptr;
/*15 */   printf("*lp = %lx\n", *lp);
/*16 */
/*17 */   sp = (short *)ptr;
/*18 */   printf("*sp = %x\n", *sp);
/*19 */
/*20 */   return 0;
/*21 */ }
```

ผลของโปรแกรมคือ

```
i = 1, j = 1
*lp = 0x10ff33ef
*sp = 0x33ef
```

ในตัวอย่างนี้ ptr เป็นพอยน์เตอร์แบบ void ในขณะที่ sp และ lp เป็นพอยน์เตอร์แบบ short และ long ตามลำดับ ในบรรทัดที่ 9 เรากำหนดให้ ptr มีค่าเท่ากับค่าของ sp ซึ่งหมายความว่า ทั้งสองชี้ไปยังแหล่งข้อมูลเดียวกัน ซึ่งก็คือ ตัวแปร i นั้นเอง เพราะ sp มีค่าเริ่มต้นเป็นที่อยู่ของตัวแปร i การทำเช่นนี้เราสามารถทำได้เพราะว่า ptr เป็นพอยน์เตอร์อเนกประสงค์ ถ้าเราต้องการอ่านค่าจากแหล่งข้อมูลที่พอยน์เตอร์ ptr นี้กำลังชี้ไป (ค่าของตัวแปร i) เราก็ต้องแปลงแบบพอยน์เตอร์ก่อน โดยแปลงจาก void เป็น short เพราะแหล่งข้อมูลที่เราต้องการอ่านนั้นที่จริงแล้วเก็บข้อมูลแบบ short โปรดสังเกตว่า การแปลงแบบพอยน์เตอร์จะต้องมีเครื่องหมายดอกจันทร์อยู่ด้วยเพื่อบอกถึงลักษณะของพอยน์เตอร์ เช่น ถ้าจะแปลงให้เป็นแบบของพอยน์เตอร์ที่ชี้ไปยังข้อมูลแบบ short เราก็ต้องเขียนว่า (short *) ไว้ข้างหน้า เมื่อแปลงแบบพอยน์เตอร์แล้วจากนั้นเราจึงอ่านค่าของแหล่งข้อมูลโดยใช้โอเปอร์เรเตอร์ * วางไว้ข้างหน้า ดังนั้นค่าของ j จะเท่ากับค่าของตัวแปร i นั้นเอง

ในบรรทัดที่ 13 เรากำหนดให้พอยน์เตอร์ ptr ชี้ไปยังตัวแปร x ซึ่งเก็บข้อมูลแบบ long ในบรรทัดที่ 14 เราผ่านค่าของพอยน์เตอร์ ptr ที่เป็นที่อยู่ของ x ให้แก่พอยน์เตอร์ lp ขั้นตอนนี้เป็นการกระทำที่ถูกต้อง แม้ว่าเราจะเจาะจงแปลงแบบพอยน์เตอร์หรือไม่ก็ตาม ในบรรทัดที่ 17 เราพยายามกำหนดให้พอยน์เตอร์ sp แบบ short ชี้ไปยังที่อยู่ของตัวแปรแบบ long เพราะว่าค่าของ ptr ในขณะนั้นเป็นหมายเลขที่อยู่ของตัวแปร x

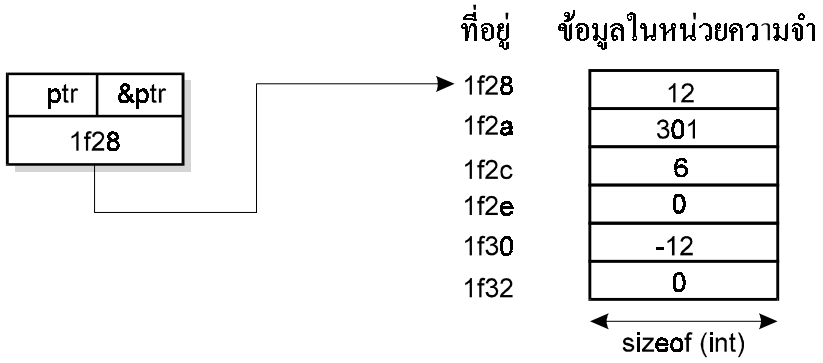
สรุปการทำงานตั้งแต่บรรทัดที่ 9 ก็คือว่า เราเริ่มต้นด้วยพอยน์เตอร์แบบ long ผ่านค่าไปให้พอยน์เตอร์แบบ void จากนั้นก็ผ่านค่าไปให้พอยน์เตอร์แบบ short ปัญหาที่เกิดขึ้นก็คือว่า ถ้าเราต้องการจะอ่านค่าจากแหล่งข้อมูลที่พอยน์เตอร์ sp และ lp กำลังชี้ไป แม้ว่าจะเป็นแหล่งข้อมูลเดียวกันก็ตาม แต่ข้อมูลที่อ่านได้จะแตกต่างกันเพราะมีรูปแบบพอยน์เตอร์ที่แตกต่างกัน lp เป็นพอยน์เตอร์แบบ long ดังนั้นนิพจน์ *lp ก็จะให้ข้อมูลแบบ long ขนาด 4 ไบต์ ในขณะที่ *sp จะให้ข้อมูลขนาด 2 ไบต์ และ *sp ก็จะเป็นสองไบต์แรกของค่าจากตัวแปร x เพราะตัวแปรแบบ short มีขนาดสองไบต์เท่านั้น เช่น ถ้า x มีค่าเท่ากับ 10ff33ef (เลขฐานสิบหก) นิพจน์ *sp ก็จะมีค่าเท่ากับ 33ef

ตัวอย่างที่แล้วแสดงให้เห็นหน้าที่ของพอยน์เตอร์แบบ void และข้อควรระวังเวลาใช้ แม้ว่าพอยน์เตอร์แบบ void จะเป็นพอยน์เตอร์อเนกประสงค์ที่เป็นตัวกลางระหว่างพอยน์เตอร์แบบอื่นๆ แต่ที่สำคัญก็คือเราจะต้องแปลงแบบพอยน์เตอร์และเข้าถึงข้อมูลอย่างถูกต้องมิฉะนั้นจะได้ข้อมูลที่ไม่ถูกต้องเมื่อเราพยายามเข้าถึงแหล่งข้อมูลที่ถูกอ้างอิงโดยใช้พอยน์เตอร์เหล่านั้น

6.4 เลขคณิตตัวชี้

เนื่องจากว่าค่าที่ตัวแปรพอยน์เตอร์เก็บไว้ในตัวเป็นที่อยู่ของหน่วยความจำของตัวแปรใดๆซึ่งหมายเลขที่อยู่นี้เป็นหมายเลขของไบต์แต่ละตัวในหน่วยความจำนั้นเอง สำหรับข้อมูลที่มีขนาดใหญ่กว่าหนึ่งไบต์ก็จะใช้พื้นที่หน่วยความจำที่ประกอบด้วยหน่วยความจำขนาดหนึ่งไบต์หลายๆตัวและหมายเลขที่อยู่ของข้อมูลนี้ก็จะเป็นหมายเลขของไบต์ตัวแรกที่ใช้

สำหรับพอยน์เตอร์ เราสามารถนำค่าของพอยน์เตอร์ใดๆ มาบวกหรือลบกับเลขจำนวนเต็มได้ ยกตัวอย่างเช่น ถ้า `ptr` เป็นพอยน์เตอร์แบบ `int` และกำหนดให้ชี้ไปยังที่อยู่ใดที่อยู่หนึ่งของหน่วยความจำสำหรับข้อมูลแบบ `int` ดังนั้นถ้าเราเพิ่มค่าของพอยน์เตอร์ขึ้นอีกหนึ่งแล้ว `ptr` ก็จะชี้ไปยังที่อยู่ของข้อมูลตัวถัดไป เป็นระยะเท่ากับขนาดของข้อมูลแบบ `int` สมมุติว่า เราได้จัดการเก็บข้อมูลแบบ `int` หลายตัวไว้ในพื้นที่ของหน่วยความจำบริเวณเดียวกัน และเรียงต่อกันไปตามลำดับ เช่น กำหนดให้ข้อมูลที่อยู่ในหน่วยความจำหมายเลข `1f28` ถึง `1f32` เป็นข้อมูลแบบ `int` ดังนั้นถ้าเรากำหนดให้ `ptr` ชี้ไปยังหน่วยความจำหมายเลข `1f28` แล้วเราจะเข้าใจความหมายของนิพจน์ต่างๆได้ดังนี้



นิพจน์	ค่าของนิพจน์
*(ptr+0)	12
*(ptr+1)	301
*(ptr+2)	6
*(ptr+3)	0
*(ptr+4)	-12
*(ptr+5)	0

เราจะเห็นได้ว่า ค่าของข้อมูลที่พอยน์เตอร์ `ptr` ชี้ไปนั้นคือ ข้อมูลที่มีค่าเท่ากับ 12 ถ้าเราอาศัยค่าของ `ptr` เป็นที่อยู่เริ่มต้นและถ้าเราบวกพอยน์เตอร์นี้ด้วยจำนวนเต็มบวกใดๆ เราก็จะได้ที่อยู่ของข้อมูลที่อยู่ในหน่วยความจำถัดไป การกระทำเช่นนี้จัดได้ว่าเป็นวิธีการหนึ่งของเลขคณิตตัวชี้

โปรดสังเกตว่า การบวกพอยน์เตอร์ด้วยเลขจำนวนเต็มแบบ int (ทั้งค่าบวก ลบ หรือศูนย์) จะได้ผลลัพธ์ที่ยังคงเป็นพอยน์เตอร์ ซึ่งชี้ไปยังที่อยู่ถัดจากค่าของพอยน์เตอร์ไปเป็นระยะเท่ากับขนาดของข้อมูลคูณด้วยเลขจำนวนเต็มที่เราบวกเข้ากับค่าของพอยน์เตอร์นี้ ดังนั้นโอเปอร์เรเตอร์ต่างๆสำหรับการบวกหรือลบเลขจำนวนเต็มจึงสามารถนำมาใช้ได้กับพอยน์เตอร์ แต่จะมีความแตกต่างอยู่ด้วยเหมือนกัน ตัวอย่างเช่น

```
ptr = (int *)0x1f28;
ptr += 1;
```

ดังนั้น ptr จะได้ค่าใหม่เป็น 1f2a เพราะ1f28 บวกด้วย 2 (ขนาดของข้อมูลแบบ int) จะเท่ากับ 1f2a (และมิใช่ 0x1f29) และค่าของนิพจน์ *ptr จะเป็น 301

```
ptr = (int *) 0x1f30;
ptr -= 2;
```

ptr จะมีค่าใหม่เป็น 1f2c คือนับถอยหลังกลับไปสองตำแหน่งหรือ สี่ไบต์ และค่าของนิพจน์ *ptr ที่ได้จะเท่ากับ 6 เราสามารถกล่าวสรุปได้ดังนี้ ถ้าเราเขียนว่า

```
ptr = ptr + i;
```

โดยที่ i เป็นตัวแปรแบบ int และ ptr เป็นตัวแปรพอยน์เตอร์ แบบ *TYPE* ซึ่งอาจจะเป็น int หรือแบบอื่นๆก็ได้ ดังนั้นประโยคคำสั่งข้างบนจะให้ผลเหมือนกับ

```
ptr = (TYPE *)((int)ptr + i * sizeof(TYPE));
```

เราลองพิจารณาความแตกต่างระหว่างนิพจน์ที่เกี่ยวข้องกับการใช้พอยน์เตอร์ดังต่อไปนี้

```
*ptr++
```

นิพจน์นี้เราอาจจะเข้าใจได้สองรูปแบบ คือ *(ptr++) และ (*ptr)++ ซึ่งจะให้ผลและขั้นตอนการทำงานที่แตกต่างกัน แต่ในภาษาซีได้มีการกำหนดลำดับการทำงานของโอเปอร์เรเตอร์ทั้งสองไว้แล้วคือ โอเปอร์เรเตอร์ ++ จะมีลำดับการทำงานมาก่อนโอเปอร์เรเตอร์ * ที่วางไว้ข้างหน้า ดังนั้น *ptr++ จึงหมายถึงการเลื่อนลูกศรของพอยน์เตอร์ ptr ให้ชี้ไปยังหมายเลขที่อยู่ของข้อมูลตัวถัดไปในหน่วยความจำแล้วอ่านค่าจากแหล่งข้อมูลนั้น ในขณะที่ (*ptr)++ เป็นการอ่านข้อมูลจากแหล่งข้อมูลที่ ptr กำลังชี้ไปแล้วเพิ่มค่าของข้อมูลที่อ่านได้นี้อีกหนึ่ง(ในกรณีที่ชี้ไปยังข้อมูลที่เราสามารถใช้โอเปอร์เรเตอร์ ++ นี้ได้) ตัวอย่างเช่น

```
int x = 10;
int *ptr = &x;
```

```
     (*ptr)++;
```

ค่าของตัวแปร x จะมีค่าเท่ากับ 11

สำหรับเลขคณิตตัวชี้ เราสามารถนำค่าของพอยน์เตอร์แบบเดียวกันสองตัวมาลบออกจากกันได้ ตัวอย่างเช่น

```
#include <stdio.h>

int main()
{
     double  x;
     int     y;
     double *dp1 = &x, *dp2 = dp1 + 1;
     int    *ip1 = &y, *ip2 = ip1 + 2;

     printf("dp1 - dp2 = %d \n", dp1 - dp2);
     printf("dp2 - dp1 = %d \n", dp2 - dp1);
     printf("(int)dp1 -(int)dp2 = %d \n",
             (int)dp1 - (int)dp2);
     printf("(int)dp2 - (int)dp1 = %d \n",
             (int)dp2 - (int)dp1);

     printf("ip1 - ip2 = %d \n", ip1 - ip2);
     printf("ip2 - ip1 = %d \n", ip2 - ip1);
     printf("(int)ip1 - (int)ip2 = %d \n",
             (int)ip1 - (int)ip2);
     printf("(int)ip2 - (int)ip1 = %d \n",
             (int)ip2 - (int)ip1);
}
```

ถ้าเราต้องการหาระยะห่างในหน่วยเป็นไบต์ระหว่างพอยน์เตอร์แบบเดียวกันสองตัว เราก็ต้องอ่านค่าของพอยน์เตอร์ และแปลงเป็น int ก่อน แล้วจึงนำค่าที่ได้มาลบออกจากกัน ถ้าไม่มีการแปลงเป็นข้อมูลแบบ int ค่าที่ได้จากการลบระหว่างพอยน์เตอร์สองตัวจะนับตามขนาดของข้อมูลที่พอยน์เตอร์กำลังชี้ไป

ตัวอย่างต่อไปแสดงให้เห็นวิธีการใช้ โอเปอร์เรเตอร์ ++ กับพอยน์เตอร์ที่ชี้ไปยังข้อมูลแบบ int

```
#include <stdio.h>

int main()
{
     int i;
```

```
     static int a=10, b=20, c=30, d=40;
     int *ptr;

     printf ("&a = %p\n", &a);
     printf ("&b = %p\n", &b);
     printf ("&c = %p\n", &c);
     printf ("&d = %p\n", &d);

     ptr = &a;
     for (i=0; i<4; i++)
     {
          int *address = ptr;
          int value = *ptr++;

          printf("ptr = %p, *ptr = %d\n", address, value);
     }
     return 0;
}
```

ผลของโปรแกรมคือ

```
     &a = 007F
     &b = 0081
     &c = 0083
     &d = 0085
     ptr = 007F, *ptr = 10
     ptr = 0081, *ptr = 20
     ptr = 0083, *ptr = 30
     ptr = 0085, *ptr = 40
```

เหตุที่ได้เขียนขั้นตอนให้โปรแกรมพิมพ์ที่อยู่ของตัวแปรทั้งสี่ตัวก็ต้องการแสดงให้เห็นว่าหน่วยความจำสำหรับตัวแปร a ,b ,c และ d ทั้งสี่ ตัวนี้ อยู่เรียงต่อกันไปตามลำดับ และไม่มีช่องว่างระหว่างหน่วยความของตัวแปรแต่ละตัวโดยสังเกตจากหมายเลขที่อยู่ของตัวแปรแต่ละตัวซึ่งมีระยะห่างกันเท่ากับขนาดของข้อมูลแต่ละตัว (ในกรณีนี้ตัวแปรแบบ int มีขนาดเท่ากับ 2 ไบต์)

ในตัวอย่างนี้ เราจะเห็นได้ว่า คอมไพล์เลอร์จะสร้างโปรแกรมที่ใช้หน่วยความจำสำหรับตัวแปรสถิต a , b , c และ d โดยเรียงลำดับหมายเลขที่อยู่จากน้อยไปมาก ในกรณีนี้เราก็จะได้หน่วยความจำของตัวแปรเรียงต่อกันไปตามลำดับตามที่เราได้แจ้งใช้จริง ดังนั้น ถ้าเราใช้พอยน์เตอร์ ptr และกำหนดให้ชี้ไปยังที่อยู่ของตัวแปร a เราก็สามารถบังคับให้ ptr ชี้ไปยังแหล่งที่อยู่ของตัวแปรตัวใดตัวหนึ่งได้ ในวงวนแบบ for เราได้ใช้โอเปอร์เรเตอร์ ++ กับพอยน์เตอร์ ptr ผลก็คือ ptr จะมีค่าเพิ่มขึ้นเรื่อยๆ ทุกครั้งที่มีการวนลูป โปรดสังเกตว่า โอเปอร์เรเตอร์ ++ ที่ใช้กับพอยน์เตอร์ ptr นี้ จะทำให้ค่าของ ptr เพิ่มจากค่าเดิมเท่ากับขนาดของข้อมูลที่พอยน์เตอร์กำลังชี้ไป ซึ่งก็คือ int

เราลองเปรียบเทียบกับอีกตัวอย่างหนึ่ง ในกรณีนี้เราแจ้งใช้ตัวแปรแบบ double ธรรมดาแต่มิได้กำหนดให้เป็น static จากผลของโปรแกรมที่แสดงข้อมูลออกทางจอภาพ เราจะเห็นได้ว่าคอมไพล์เลอร์จะกำหนดให้ตัวแปรใช้หน่วยความจำจากหมายเลขมากไปน้อย ซึ่งแตกต่างจากในโปรแกรมที่แล้ว

```
#include <stdio.h>

int main()
{
     int i;
     double a=1.0, b=2.0, c=3.0,d=4.0;
     double *ptr;

     printf ("&a = %p\n", &a);
     printf ("&b = %p\n", &b);
     printf ("&c = %p\n", &c);
     printf ("&d = %p\n", &d);

     ptr = &a;
     for (i=0; i<4; i++)
     {
          double *address = ptr;
          double value = *ptr--;
          printf ("ptr = %p, *ptr = %lf\n", address, value);
     }
  return 0;
}
```

ผลของโปรแกรม

```
     &a = 22CA
     &b = 22C2
     &c = 22BA
     &d = 22B2
     ptr = 22CA, *ptr = 1.000000
     ptr = 22C2, *ptr = 2.000000
     ptr = 22BA, *ptr = 3.000000
     ptr = 22B2, *ptr = 4.000000
```

ในตอนแรกเรากำหนดให้พอยน์เตอร์ ptr ชี้ไปยังที่อยู่ของตัวแปร a แต่เนื่องจากว่า a มีหมายเลขที่อยู่มากกว่าหมายเลขที่อยู่ของตัวแปรตัวอื่นๆอีกสามตัว ดังนั้น ถ้าเราจะเข้าถึงตัวแปรสามตัวที่เหลือนั้น เราจะต้องกำหนดให้ค่าของ ptr มีค่าน้อยลงตามลำดับ โดยใช้โอเปอร์เรเตอร์ -- และภายในวงวน for ค่าของ ptr จะลดลงทีละแปดไบต์ซึ่งเป็นค่าที่เท่ากับขนาดของข้อมูลแบบ double คือ แปดไบต์

คำถาม : ลองติดตั้งค่าเริ่มต้นเฉพาะกับตัวแปรบางตัวเท่านั้น เช่น

```
double a=1.0, b, c=3.0, d;
```

แล้วดูว่าหมายเลขที่อยู่ของตัวแปรแต่ละตัวจะเป็นอย่างไร จะ เรียงต่อกันตามลำดับที่ได้แจ้งใช้หรือไม่

ประโยชน์ของการใช้งานพอยน์เตอร์ในโปรแกรมภาษาซีคือการผ่านที่อยู่ของแหล่งข้อมูลแทนที่จะผ่านค่าของข้อมูลให้เป็นพารามิเตอร์ของฟังก์ชัน การทำงานของฟังก์ชัน `scanf()` ก็อาศัยหลักการนี้เช่นกันเพราะเราต้องผ่านที่อยู่ของแหล่งข้อมูลให้แก่ฟังก์ชันเพื่อที่จะสามารถใช้เก็บค่าที่ได้จากฟังก์ชันได้

ส่วนใหญ่แล้วพอยน์เตอร์จะเกี่ยวข้องกับการเรียกใช้ฟังก์ชันโดยการผ่านตัวอ้างอิง ตัวอย่างเช่น เราต้องการสร้างฟังก์ชันที่มีพารามิเตอร์สองตัวเป็นที่อยู่ของตัวแปรใดๆแบบ `int` สองตัว และหน้าที่ของฟังก์ชันนี้ก็คือการเพิ่มค่าของตัวแปรตามที่อยู่ที่เป็นพารามิเตอร์แต่ละตัวโดยการยกกำลังสองของค่าเดิมและเขียนค่าใหม่แทนที่ค่าเก่าของตัวแปรแต่ละตัวนั้น เราก็สามารถนิยามฟังก์ชันดังกล่าวได้ดังนี้

```
void  square (int *a, int *b)
{
   *a *= *a;
   *b *= *b;
}
```

หรือถ้าแจ้งใช้พารามิเตอร์ของฟังก์ชันแบบเก่าก็จะเป็น

```
void  square (a, b)
int *a;
int *b;
{
   *a *= *a;
   *b *= *b;
}
```

ตามปรกติแล้วเราจะไม่แจ้งใช้พารามิเตอร์ของฟังก์ชันในรูปแบบเก่าตามตัวอย่างนี้ แต่ที่ได้ยกมาเป็นตัวอย่างมีจุดประสงค์คือบางทีผู้อ่านบางท่านอาจจะไปอ่านหรือพบเจอโปรแกรมภาษาซีที่มีการแจ้งใช้พารามิเตอร์ในรูปแบบเก่าก็จะได้คุ้นเคยและไม่เกิดความสงสัยว่าทำไมเขาจึงเขียนโปรแกรมโค้ดในลักษณะนี้

เราลองเปรียบเทียบตัวอย่างข้างบนกับฟังก์ชันที่มีรูปแบบดังต่อไปนี้

```
void square (int *a, int *b)
{
   a *= a;
   b *= b;
}
```

เนื่องจากว่า a และ b เป็นพารามิเตอร์ที่ทำหน้าที่เป็นพอยน์เตอร์ ดังนั้น ถ้าเรามิได้ใส่โอเปอร์เรเตอร์ * ไว้ข้างหน้าจึงหมายความว่า เราได้อ่านค่าของพอยน์เตอร์ a และ b ซึ่งเป็นที่อยู่ของแหล่งข้อมูลใดๆ แบบ int (ค่าของที่อยู่เหล่านี้ได้ตอนที่เราเรียกใช้ฟังก์ชัน) ในกรณีนี้เราได้พยายามอ่านค่าของ a และคูณกับค่าของตัวมันเองแล้วเก็บผลคูณที่ได้ไว้ใน a อีกครั้ง การกระทำเช่นนี้ถือว่าไม่ผิดแปลกอะไรถ้า a เป็นตัวแปรธรรมดาและมิใช่เป็นพอยน์เตอร์ แต่เนื่องจากว่า a เป็นพอยน์เตอร์ การคูณพอยน์เตอร์ไม่ว่าจะเป็นการคูณกับตัวเองหรือคูณกับพอยน์เตอร์ตัวอื่น ถือว่าไม่ใช่โอเปอร์เรเตอร์ที่ถูกต้องตามหลักของเลขคณิตตัวชี้ (Pointer Arith-metic) ซึ่งจะมีเฉพาะการลบระหว่างพอยน์เตอร์เท่านั้น ดังนั้นการคูณกันระหว่างพอยน์เตอร์จึงไม่ถูกต้องและฟังก์ชันในลักษณะนี้จึงผิดและไร้ความหมาย สมมุติว่าถ้าภาษาซีอนุญาตให้มีการคูณกันระหว่างพอยน์เตอร์ได้ ประโยคคำสั่งทั้งสองก็จะถูกต้อง

```
a *= a;
b *= b;
```

แม้กระนั้นค่าของ a และ b ที่ได้ใหม่ก็ไม่มีผลอะไรเมื่อฟังก์ชันจบการทำงาน เพราะ a และ b เป็นเพียงพอยน์เตอร์ที่เก็บหมายเลขที่อยู่ของแหล่งข้อมูลแบบ int ใดๆสองตัว ตราบเท่าที่เรายังมิได้เข้าถึงแหล่งข้อมูลโดยการใช้โอเปอร์เรเตอร์ * และอ่านหรือเปลี่ยนแปลงค่าของข้อมูลเหล่านั้น พอยน์เตอร์ก็ไม่มีประโยชน์อะไร และเราก็จะเกี่ยวข้องกับเรื่องของที่อยู่เท่านั้น ถ้าเราเปลี่ยนแปลงค่าของ a และ b (ดังในตัวอย่างนี้) ก็จะเป็นการกำหนดให้พอยน์เตอร์ a และ b ชี้ไปยังที่อยู่อื่นและจะเป็นค่าที่ถูกต้องหรือไม่ เป็นอีกคำถามหนึ่ง เพราะว่า a และ b เป็นตัวแปรพอยน์เตอร์ที่ใช้ภายในฟังก์ชันเท่านั้น เมื่อจบการทำงานของฟังก์ชันตัวแปรพอยน์เตอร์ a และ b นี้ก็จะถูกทำลายไป เพราะทั้งสองก็ถือว่าเป็นตัวแปรภายในของฟังก์ชัน ดังนั้นถ้าเราเปลี่ยนแปลงค่าของ a และ b เท่านั้น ก็จะไม่มีผลกับแหล่งข้อมูลที่พอยน์เตอร์ทั้งสองได้ชี้ไปในตอนแรก

เมื่อสร้างฟังก์ชันได้ถูกต้องแล้วก็ต้องเรียกใช้ให้ถูกต้องด้วย ตัวอย่างทั้งที่ถูกและผิด เช่น

```
int x=10, y=7;
int *p1=&x, *p2=&y;

square(&x, &y);    /* O.K. */
square(x, y);      /* ILLEGAL */

square(p1, p2);    /* O.K. */
square(*p1, *p2);  /* ILLEGAL */
square(&p1, &p2);  /* ILLEGAL */
```

ถ้าฟังก์ชันต้องการพารามิเตอร์ที่เป็นที่อยู่ เราก็ต้องให้ที่อยู่ของตัวแปร และไม่ใช่ค่าของตัวแปร

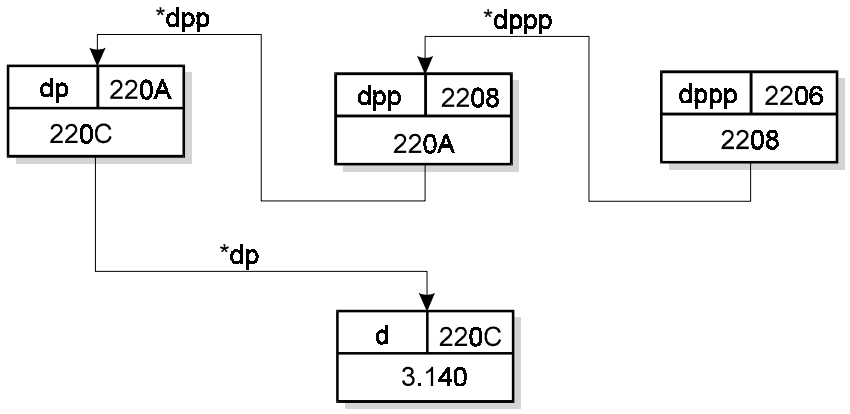
นอกจากที่เราจะกำหนดให้ตัวแปรที่เป็นพอยน์เตอร์ชี้ไปยังที่อยู่ของหน่วยความจำสำหรับพื้นฐานได้แล้วเรายังสามารถแจ้งใช้ตัวแปรพอยน์เตอร์ที่ชี้ไปยังพอยน์เตอร์ก็ได้ ตัวอย่างเช่น

```
int      **ipp;
char     **chpp;
double  ***dbpp;
void   ****ptr;
```

โปรดสังเกตจำนวนของเครื่องหมายดอกจันทร์ที่อยู่ข้างหน้าชื่อตัวแปรทั้งหลายเพราะเครื่องหมายดอกจันทร์ที่อยู่ข้างหน้าชื่อของตัวแปรพอยน์เตอร์เหล่านี้ บ่งบอกถึงลักษณะและคุณสมบัติของพอยน์เตอร์ เช่น `ipp` จากตัวอย่างข้างบนหมายถึงพอยน์เตอร์ที่ชี้ไปยังพอยน์เตอร์ที่ชี้ ต่อไปยังข้อมูลแบบ `int`

```
#include <stdio.h>

int main()
{
     double d = 3.14;

     double   *dp   = &d;
     double  **dpp  = &dp;
     double ***dppp = &dpp;

     printf(" &d\t d\n");
     printf(" %x\t %.3lf\n\n",
               &d, d);

     printf(" &dp\t dp\t *dp\n");
     printf(" %x\t %x\t %.3lf\n\n",
               &dp, dp, *dp);

     printf(" &dpp\t dpp\t *dpp\t **dpp\n");
     printf(" %x\t %x\t %x\t %.3lf\n\n",
               &dpp, dpp, *dpp, **dpp);

     printf(" &dppp\t dppp\t *dppp\t **dppp\t ***dppp\n");
     printf(" %x\t %x\t %x\t %x\t %.3lf\n\n",
               &dppp, dppp, *dppp, **dppp, ***dppp);

     return 0;
}
```

ตัวอย่างผลการทำงานของโปรแกรม เราสามารถเขียนให้อยู่ในรูปภาพได้ดังนี้



6.5 ขนาดหน่วยความจำของพอยน์เตอร์

โปรแกรมต่อไปนี้เราจะใช้ในการหาขนาดของหน่วยความจำที่ใช้สำหรับตัวแปรพอยน์เตอร์ซึ่งชี้ไปยังแหล่งข้อมูลที่แตกต่างกัน

```
#include <stdio.h>

int main()
{
     char         * chptr;
     int          * iptr;
     long int     * liptr;
     float        * fptr;
     double       * dptr;
     long double  * ldptr;
     void         * vptr;

     printf("%d %d %d %d %d %d %d\n",
           sizeof (chptr),
           sizeof (iptr),
           sizeof (liptr),
           sizeof (fptr),
           sizeof (dptr),
           sizeof (ldptr),
           sizeof (vptr)
     );
     return 0;
}
```

ผลของโปรแกรมคือ

```
2 2 2 2 2 2 2
```

เราจะเห็นได้ว่าขนาดของพอยน์เตอร์ไม่ว่าจะชี้ไปยังข้อมูลพื้นฐานแบบใดจะมีขนาดเดียวกันหมด ซึ่งก็ไม่น่าแปลกใจ เพราะตามที่เราทราบพอยน์เตอร์แต่ละตัวจะเก็บค่าที่เป็นที่อยู่นั้นเองซึ่งมีขนาดเท่ากับขนาดของ int (2 ไบต์สำหรับเครื่องพีซีธรรมดา และ 4 ไบต์สำหรับเครื่องแบบ UNIX) พอยน์เตอร์ที่ชี้ไปยังพอยน์เตอร์ก็มีขนาดเท่ากัน

6.6 การใช้ const กำกับพอยน์เตอร์

คำว่า const เราจะใช้ตอนที่แจ้งใช้ตัวแปรใดๆ และหลังจากได้แจ้งใช้และติดตั้งค่าเริ่มต้นแล้วไม่สามารถเปลี่ยนแปลงได้และเราก็เรียกตัวแปรลักษณะนี้ว่าตัวแปรคงที่ เช่น

```
const int number = 1024;
```

นอกจากนี้เราสามารถใช้ const ร่วมกับการแจ้งใช้ตัวแปรพอยน์เตอร์ได้ เช่น

```
const int *iptr;
```

หมายความว่า พอยน์เตอร์ iptr ถ้าชี้ไปยังแหล่งข้อมูลใดแล้ว (ข้อมูลแบบ int) เราไม่สามารถอาศัยพอยน์เตอร์นี้ในการเปลี่ยนแปลงค่าของแหล่งข้อมูลนั้นโดยทางอ้อมได้เหมือนที่เราได้เรียนรู้ไป ดังนั้นเราสามารถใช้ iptr ในการเข้าถึงข้อมูลแบบ int ที่พอยน์เตอร์กำลังชี้ไปและใช้สำหรับอ่านค่าจากแหล่งข้อมูลเท่านั้น ตัวอย่างที่ผิดเช่น

```
/*1*/     int x = 10, y = 102;
/*2*/     const int *iptr = &x;    /* iptr points at x */
/*3*/     *iptr = 1001;            /* ILLEGAL*/
/*4*/     iptr  = &y;              /* O.K. */
```

จะเห็นได้ว่า เราใช้ iptr ในการเก็บที่อยู่ของตัวแปร x ซึ่งเป็นข้อมูลแบบ int ที่มีค่าเท่ากับ 10 ในบรรทัดที่ 3 จากตัวอย่างนี้ เราพยายามเขียนค่า 1001 ทับค่าเดิมของตัวแปร x โดยอาศัยพอยน์เตอร์ iptr ซึ่งไม่ถูกต้อง ในขณะที่เราสามารถบังคับให้ iptr ชี้ไปยังที่อยู่ของข้อมูลใหม่ได้ ซึ่งในบรรทัดที่ 4 นี้เรากำหนดให้พอยน์เตอร์ iptr ชี้ไปยังที่อยู่ของตัวแปร y ซึ่งตัวแปรนี้มีค่าเท่ากับ 102

แต่ถ้าเราเขียนคำว่า const ไว้หลังเครื่องหมายดอกจันทร์ตามตัวอย่างต่อไปนี้

```
/*1*/     int x = 10, y = 1001;
/*2*/     int * const iptr = &x;   /* iptr points to x */
/*3*/     *iptr = 1001;            /* O.K.*/
/*4*/     iptr  = &y;              /* ILLEGAL */
```

ในกรณีนี้เราไม่สามารถกำหนดให้พอยน์เตอร์ iptr ชี้ไปยังแหล่งข้อมูลอื่นนอกเหนือจากที่ได้กำหนดไว้ตั้งแต่ตอนแจ้งใช้ ดังนั้นการแจ้งใช้ตัวแปรพอยน์เตอร์ในลักษณะนี้ เราจะต้องติดตั้งค่าของพอยน์เตอร์พร้อมกับการแจ้งใช้ ซึ่งค่าเริ่มต้นของพอยน์เตอร์ iptr นี้จะต้องเป็นที่อยู่ของข้อมูลแบบ int ตัวใดตัวหนึ่ง สำหรับกรณีนี้เราสามารถเปลี่ยนแปลงค่าของแหล่งข้อมูลที่พอยน์เตอร์กำลังชี้ไปได้ ดังนั้นบรรทัดที่ 3 จึงไม่ผิดหลักไวยากรณ์ ในขณะที่เราพยายามเปลี่ยนค่าของ iptr ในบรรทัดที่ 4 ซึ่งไม่ถูกต้อง

```
/*1*/     int x = 10, y = 1001;
/*2*/     const int * const iptr = &x;  /* iptr points at x */
/*3*/     *iptr = 1001;                 /* ILLEGAL */
/*4*/     iptr = &y;                    /* ILLEGAL */
```

อีกกรณีหนึ่งคือการเขียนคำว่า const ไว้ทั้งสองตำแหน่ง ซึ่งเป็นการรวมสองกรณีแรกที่เราได้เห็นตัวอย่างไปแล้ว ดังนั้นในกรณีนี้เราไม่สามารถเปลี่ยนแปลงค่าของพอยน์เตอร์ได้อีก หลังจากที่ได้แจ้งใช้แล้ว และเราก็ไม่สามารถเปลี่ยนแปลงค่าของแหล่งข้อมูลที่พอยน์เตอร์กำลังชี้ไป

แต่อย่างไรก็ตามเราสามารถใช้วิธีการแปลงแบบสำหรับพอยน์เตอร์ ในการเปลี่ยนแปลงค่าของแหล่งข้อมูลโดยทางอ้อมได้โดยใช้พอยน์เตอร์แม้ว่าเราจะกำหนดให้พอยน์เตอร์ที่เราใช้นี้เป็น const ก็ตาม ตัวอย่างเช่น

```
#include <stdio.h>

int main()
{
     const int x = 10;
     const int *iptr1 = &x;
     int * iptr2;

     x = -1110;                          /* ILLEGAL!!! */
     printf(" x = %d\n", x);

     *iptr1 = -1111;                     /* ILLEGAL!!! */
     printf(" *iptr1 = %d\n", *iptr1);

     iptr2  = (int *)iptr1;              /* Type Conversion */
     *iptr2 = -1112;                     /* O.K. */
     printf(" *iptr2 = %d\n", *iptr2);

     return 0;
}
```

แต่อย่างไรก็ตาม วิธีการนี้ทำให้คุณสมบัติของ const สำหรับพอยน์เตอร์เสียไป ดังนั้นควรจะระมัดระวังเวลาใช้วิธีการแปลงแบบสำหรับพอยน์เตอร์ในลักษณะนี้

ประโยชน์ของการใช้พอยน์เตอร์ที่ชี้ไปยังพอยน์เตอร์ จะเห็นได้จากตัวอย่างต่อไปนี้

สมมุติว่า เราต้องการผ่านพอยน์เตอร์แบบ char เป็นพารามิเตอร์ของฟังก์ชันและต้องการให้ฟังก์ชันเปลี่ยนแปลงค่าของพอยน์เตอร์ตัวนี้ เราก็ต้องใช้พอยน์เตอร์ที่ชี้ไปยังพอยน์เตอร์ ลองเปรียบเทียบการทำงานของสองฟังก์ชัน

```
#include <stdio.h>

void func1(char *p)
{
     static char ch = 'B';
     p = &ch;
}

void func2(char **p)
{
     static char ch = 'C';
     *p = &ch;
}

int main()
{
     char   ch  = 'A';
     char *cptr = &ch;

     func1(cptr);
     printf("1) *cptr = %c\n", *cptr);

     func2(&cptr);
     printf("2) *cptr = %c\n", *cptr);

     return 0;
}
```

ผลของโปรแกรมคือ

```
     1) *cptr = A
     2) *cptr = C
```

เราจะเห็นได้ว่า เราได้ผ่านค่าของพอยน์เตอร์ cptr ให้เป็นพารามิเตอร์ของฟังก์ชัน func1() ดังนั้นจึงเราจึงไม่สามารถเปลี่ยนแปลงค่าของ cptr ในฟังก์ชันหลักได้ และเมื่อฟังก์ชันจบการทำงาน cptr ก็ยังคงมีค่าเท่าเดิม ซึ่งเป็นที่อยู่ของตัวแปร ch สำหรับการเรียกใช้ฟังก์ชัน func2() เราจะต้องผ่านที่อยู่ของ

พอยน์เตอร์ ดังนั้นฟังก์ชันจึงสามารถเปลี่ยนแปลงค่าของพอยน์เตอร์ cptr และผลที่ได้คือ cptr จะชี้ไปยังตัวแปรสถิต ch ที่อยู่ภายในฟังก์ชัน func2() ซึ่งมีค่าเท่ากับ 'C' สรุปได้ว่า ถ้าต้องการจะเปลี่ยนค่าของพอยน์เตอร์ เราก็ต้องผ่านที่อยู่ของพอยน์เตอร์ไปให้ฟังก์ชัน ซึ่งหมายความว่า เราจะต้องใช้พอยน์เตอร์ที่ชี้ไปยังพอยน์เตอร์เป็นพารามิเตอร์ของฟังก์ชัน

เราได้เรียนรู้ไปแล้วว่า เราสามารถแจ้งใช้ตัวแปรซึ่งเป็นพอยน์เตอร์ที่ชี้ไปยังข้อมูลพื้นฐานใดๆได้ นอกจากนี้ ภาษาซียังเปิดโอกาสให้เราใช้ ตัวแปรที่เป็นพอยน์เตอร์และชี้ไปยังฟังก์ชันใดๆตามรูปแบบของฟังก์ชันที่เราต้องการได้ และแบบของฟังก์ชันจะถูกกำหนดโดยพารามิเตอร์และแบบข้อมูลที่ให้กลับคืนของฟังก์ชัน

6.7 วิธีการแจ้งใช้พอยน์เตอร์ที่ชี้ไปยังฟังก์ชัน

เราลองเปรียบเทียบสามตัวอย่างต่อไปนี้ ซึ่งตัวอย่างแรกเป็นการแจ้งใช้ฟังก์ชัน ที่ให้ค่าที่เป็นข้อมูลแบบ int กลับคืน ตัวอย่างที่สองเป็นการแจ้งใช้ฟังก์ชันที่ให้พอยน์เตอร์แบบ int กลับคืน และตัวอย่างที่สามเป็นการแจ้งใช้ตัวแปรที่ทำหน้าที่เป็นพอยน์เตอร์ซึ่งชี้ไปยังแบบฟังก์ชันที่ให้ค่าแบบ int กลับคืน

```
int (f1)(void); /* f1 is a function returning an int. */

int *f2(void);  /* f2 is a function returning a pointer
                           to int. */

int (*pf)(void); /* pf is a pointer to a function having
                    no argument and returning an int. */
```

ในตัวอย่างแรก เราเห็นได้ว่า ชื่อของฟังก์ชัน ซึ่งก็คือ f1 เขียนอยู่ระหว่างเครื่องหมายวงเล็บเปิดปิด แต่ตามปรกติแล้วฟังก์ชันธรรมดาทั่วๆไป เวลานิยามหรือแจ้งใช้เราจะไม่ใส่เครื่องหมายวงเล็บให้แก่ชื่อของฟังก์ชัน แต่ถ้าต้องการใส่ก็ได้เพียงแต่จะต้องใส่ให้ถูกต้อง ถ้าเราเขียนว่า

```
int (f1)(void);
int  f1 (void);
```

อย่างใดอย่างหนึ่ง จึงไม่แตกต่างกัน ซึ่งหมายถึงการแจ้งใช้ฟังก์ชัน f1() และนอกจากนั้นเรายังได้กำหนดให้ฟังก์ชันนี้ไม่มีอาร์กิวเมนท์ใดๆ คือเป็น void ในทำนองเดียวกันถ้าเราต้องการจะใส่เครื่องหมายวงเล็บให้แก่ชื่อของฟังก์ชันในตัวอย่างที่สอง เราก็ต้องใส่ให้ถูกต้องเพื่อมิให้คุณสมบัติของฟังก์ชันเปลี่ยนแปลงไป

```
int *(f2)(void);
```

ซึ่งจะแตกต่างจากตัวอย่างที่สามได้อย่างชัดเจน (ตัวอย่างที่สองก็ยังคงเป็นการแจ้งใช้ฟังก์ชันชื่อ f2)

```
int (*pf)(void);
```

เพราะ f2 เป็นฟังก์ชันแบบ void ที่ให้พอยน์เตอร์ที่ชี้ไปยังข้อมูลแบบ int กลับคืน ในขณะที่ pf ทำหน้าที่เป็นพอยน์เตอร์ (ตัวอย่างที่สามเป็นการแจ้งใช้พอยน์เตอร์มิใช่การแจ้งใช้ฟังก์ชันเหมือนสองตัวอย่างแรก) และเมื่อเป็นพอยน์เตอร์แล้วก็ต้องชี้ไปยังสิ่งใดสิ่งหนึ่ง ในกรณีนี้ pf ชี้ไปยังฟังก์ชันที่ให้ค่าแบบ int กลับคืน เราอาจจะกล่าวได้ว่า ฟังก์ชันก็เป็นแบบข้อมูลอย่างหนึ่งตามความหมายที่เราใช้กับข้อมูลพื้นฐานทั่วๆไป

```
/*1*/     #include <stdio.h>
/*2*/
/*3*/     int f1(void)
/*4*/     {
/*5*/          return 1;
/*6*/     }
/*7*/
/*8*/     int f2(void)
/*9*/     {
/*10*/         return 2;
/*11*/    }
/*12*/
/*13*/    int main()
/*14*/    {
/*15*/         int (f1)(void);  /* function prototype */
/*16*/         int (f2)(void);  /* function prototype */
/*17*/         int (*pf)(void);
/*18*/         pf = f1;
/*19*/         printf("f1 returns %d.\n", (*pf)() );
/*20*/         printf("f1 returns %d.\n", pf() );
/*21*/         pf = f2;
/*22*/         printf("f2 returns %d.\n", (*pf)() );
/*23*/         printf("f2 returns %d.\n", pf() );
/*24*/         return 0;
/*25*/    }
```

และผลที่ได้คือ

```
f1 returns 1.
f1 returns 1.
f2 returns 2.
f2 returns 2.
```

โปรดสังเกตว่า ในบรรทัดที่ 15 หรือ 16 เป็นการแจ้งใช้ต้นแบบฟังก์ชัน f1() และ f2() หรือเรียกว่า Function Prototype และตามปรกติแล้วจะให้ผลเหมือนกับกรณีที่เราเขียนคำว่า extern ไว้ข้างหน้าดังนี้

```
extern    int (f1)(void);  /* function prototype */
extern    int (f2)(void);  /* function prototype */
```

จากผลการทำงานของโปรแกรม ทำให้เรามองเห็นได้ว่า คำสั่งในบรรทัดที่ 19 และ 20 ให้ผลเหมือนกัน และในบรรทัดที่ 22 และ 23 ก็ให้ผลเช่นเดียวกัน

```
(*pf)()
 pf()
```

pf เป็นพอยน์เตอร์ที่ชี้ไปยังฟังก์ชัน ซึ่งถ้าต้องการเรียกใช้ฟังก์ชันดังกล่าว เราก็จะใช้เครื่องหมายดอกจันทร์วางไว้ข้างหน้าชื่อของพอยน์เตอร์ดังกล่าวหรือไม่ใช้ก็ได้ และตามด้วยพารามิเตอร์ของฟังก์ชันที่อยู่ระหว่างเครื่องหมายวงเล็บเปิดปิดถัดไป การเข้าถึงตัวฟังก์ชันโดยอาศัยพอยน์เตอร์จึงเป็นไปในลักษณะเช่นเดียวกับเวลาที่ใช้พอยน์เตอร์ในเข้าถึงข้อมูลพื้นฐานแบบใดแบบหนึ่ง เช่น int โดยทางอ้อม โปรดพิจารณาตารางเปรียบเทียบความเหมือนและความแตกต่าง

p พอยน์เตอร์ชี้ไปยัง int	pf พอยน์เตอร์ชี้ไปยังฟังก์ชัน	คำอธิบาย
int *p;	int (*pf)(void);	การแจ้งใช้
*p	(*pf)()	*p ให้ค่าของข้อมูลที่ p กำลังชี้ไป *pf เป็นการอ้างถึงตัวฟังก์ชันที่ pf กำลัง ชี้ไป และ (*pf)() เป็นการเรียกใช้ฟังก์ชัน
p	(pf)()	p ให้ที่อยู่ของข้อมูลแบบ int (pf)() ให้ผลเหมือนกับ (*pf)()
	pf	pf ให้ที่อยู่ของฟังก์ชันที่ pf กำลังชี้ไป
&p	&pf	&p และ &pf ให้ที่อยู่ของพอยน์เตอร์ p และ pf ตามลำดับ

ตัวอย่างการเรียกใช้ฟังก์ชันโดยผ่านพอยน์เตอร์ pf ที่ไม่ถูกต้อง เช่น

```
(&pf)();
```

เป็นการหาที่อยู่ของพอยน์เตอร์ pf ซึ่งได้เป็นค่าคงที่แบบ int แต่มีวงเล็บที่ตามมาจึงหมายถึงค่าคงที่ตามด้วยวงเล็บซึ่งเข้าทำนองเดียวกันกับการที่เราเขียนว่า

```
(16037)();
```

ซึ่งผิดหลักไวยากรณ์ในภาษาซี หรือถ้าเราเขียนว่า

```
&pf();
```

ก็หมายถึงการหาที่อยู่ของค่ากลับคืนจากการเรียกใช้ฟังก์ชัน ในกรณีนี้เราใช้โอเปอร์เรเตอร์ & สำหรับหาค่าคงที่แบบ int (ค่าที่ได้จากเรียกใช้ฟังก์ชันโดยอาศัยพอยน์เตอร์ pf) และไม่ถูกต้อง ซึ่งเหมือนกับการเขียนว่า

```
&1;
```

ถ้าสมมุติว่า ฟังก์ชันให้ค่าเท่ากับ 1 กลับคืน ดังนั้นจึงไม่ถูกต้องเพราะเราไม่สามารถหาที่อยู่ของค่าคงที่ได้

ตัวอย่างต่อไปจะคล้ายกับตัวอย่างที่แล้วเพียงแต่คราวนี้เรานิยามฟังก์ชันและพอยน์เตอร์ใหม่โดยที่เรากำหนดให้ฟังก์ชันมีพารามิเตอร์สองตัวแบบ int

```
/*1*/     #include <stdio.h>
/*2*/
/*3*/     int f1(int i, int j)
/*4*/     {
/*5*/          return i*i - j*j;
/*6*/     }
/*7*/
/*8*/     int f2(int i, int j)
/*9*/     {
/*10*/         return i*i + j*j;
/*11*/    }
/*12*/
/*13*/    int main()
/*14*/    {
/*15*/         int (f1)(int, int); /* function prototype */
/*16*/         int (f2)(int, int); /* function prototype */
/*17*/         int (*pf)(int, int);
/*18*/         pf = f1;
/*19*/         printf("f1 returns %d.\n", (*pf)(5,3) );
/*20*/         printf("f1 returns %d.\n", pf(5,3) );
/*21*/         pf = f2;
/*22*/         printf("f2 returns %d.\n", (*pf)(5,3) );
/*23*/         printf("f2 returns %d.\n", pf(5,3) );
/*24*/         return 0;
/*25*/    }
```

การใช้พอยน์เตอร์ที่ชี้ไปยังฟังก์ชัน เรามักจะไม่พบการทำงานในลักษณะนี้ได้บ่อยเหมือนการใช้พอยน์เตอร์กับตัวแปร อย่างไรก็ตามในบางสถานการณ์เทคนิคนี้ก็มีประโยชน์มาก เช่น การผ่านฟังก์ชันเป็น

พารามิเตอร์ของฟังก์ชันอื่นหรือฟังก์ชันใดอาจจะให้ฟังก์ชันตัวอื่นกลับคืนก็ได้ ซึ่งหมายถึงการให้พอยน์เตอร์ที่ชี้ไปยังฟังก์ชันตัวอื่น ตัวอย่างเช่นการจับคู่ระหว่างข้อมูลที่ใช้เป็นพารามิเตอร์และฟังก์ชันที่เราต้องการผ่านข้อมูล ไปให้ เราลองมาพิจารณาตัวอย่างต่อไปนี้

```
int assign (int, int, int (*func)(int, int) );
```

ฟังก์ชัน `assign()` เป็นฟังก์ชันที่มีพารามิเตอร์สามตัว สองตัวแรกเป็นข้อมูลแบบ `int` พารามิเตอร์ตัวที่สามเป็นพอยน์เตอร์ที่ชี้ไปยังฟังก์ชัน ซึ่งจะเป็นฟังก์ชันที่มีพารามิเตอร์สองตัวแบบ `int` และ ให้ค่าแบบ `int` กลับคืน

```
#include <stdio.h>

int assign (int a, int b, int (*func)(int, int) )
{
   return (*func)(a, b);
}

int getMin(int a, int b)
{
  return (a > b) ? b : a;
}

int getMax(int a, int b)
{
  return (a > b) ? a : b;
}


int main()
{
   extern int getMin(int, int);
   extern int getMax(int, int);

   int x=10, y= -3;
   int min, max;

   max = assign (x,y, getMax);
   min = assign (x,y, getMin);

   printf("max{%d,%d} = %d\n", x,y,max);
   printf("min{%d,%d} = %d\n", x,y,min);

   return 0;
}
```

ในตัวอย่างนี้เราสร้างฟังก์ชัน `assign()` ที่ใช้ในการจับคู่ระหว่างพารามิเตอร์ `x` และ `y` กับฟังก์ชันใดๆที่มีรูปแบบตามที่กำหนดคือฟังก์ชันจะต้องรับพารามิเตอร์แบบ `int` สองตัวและให้ค่าแบบ `int` กลับคืนและจากตัวอย่างเราสามารถเข้าใจการทำงานของฟังก์ชันได้ดังนี้

```
max = assign (x,y, getMax);
min = assign (x,y, getMin);
```

จะให้ผลลัพธ์เหมือนกับ

```
max = getMax(x,y);
min = getMin(x,y);
```

จากตัวอย่างที่ผ่านๆมาเราสามารถกล่าวได้ว่า ตัวระบุชื่อของฟังก์ชันก็คือชื่อของพอยน์เตอร์ที่ชี้ไปยังฟังก์ชันนั้นเองและเป็นพอยน์เตอร์แบบคงที่ (Constant Pointer) และที่น่าสนใจก็คือ การเรียกใช้ฟังก์ชัน `f1()` ทั้งสี่รูปแบบต่อไปนี้

```
f1();
(f1)();
(*f1)();
(***********f1)();
```

ต่างก็ให้ผลเหมือนกัน

พอยน์เตอร์ที่ชี้ไปยังฟังก์ชันรูปแบบต่าง ๆ	***คำอธิบาย***
`int (*pf)(int, int);`	พอยน์เตอร์ที่ชี้ไปยังฟังก์ชันที่มีพารามิเตอร์สองตัวแบบ `int` และให้ค่าแบบ `int` กลับคืน
`float (*pf)(float);`	พอยน์เตอร์ที่ชี้ไปยังฟังก์ชันที่มีพารามิเตอร์หนึ่งตัวแบบ `float` และให้ค่าแบบ `float` กลับคืน
`char *(*pf)(char *, int);`	พอยน์เตอร์ที่ชี้ไปยังฟังก์ชันที่มีพารามิเตอร์สองตัว พารา-มิเตอร์ตัวแรกเป็นพอยน์เตอร์สำหรับข้อมูลแบบ `char` พารามิเตอร์ตัวที่สองเป็นข้อมูลแบบ `int` และ ฟังก์ชันให้พอยน์เตอร์ที่ชี้ไปยังข้อมูลแบบ `char` กลับคืน
`void (*pf)(void);`	พอยน์เตอร์ที่ชี้ไปยังฟังก์ชันที่ไม่มีพารามิเตอร์และไม่ให้ข้อมูลใดๆกลับคืน
`void *(*pf)(void);`	พอยน์เตอร์ที่ชี้ไปยังฟังก์ชันที่ไม่มีพารามิเตอร์ใดๆแต่ให้พอยน์เตอร์แบบ `void` กลับคืน

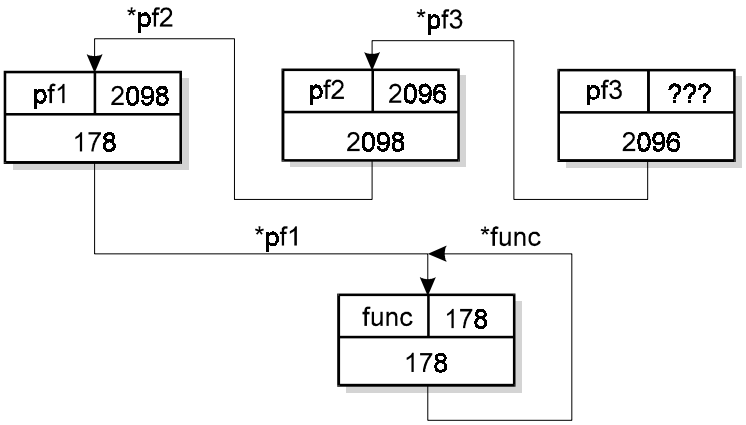
`char **(*pf)(void);`	พอยน์เตอร์ที่ชี้ไปยังฟังก์ชันที่ไม่มีพารามิเตอร์ใดๆแต่ให้พอยน์เตอร์ที่ชี้ไปยังพอยน์เตอร์แบบ `void` กลับคืน (Pointer-to-void-Pointer)

ตัวอย่างข้างล่างนี้ แสดงให้เห็นการใช้พอยน์เตอร์ที่ชี้ไปยังพอยน์เตอร์ และใช้กับฟังก์ชัน

```
#include <stdio.h>

int main()
{
     extern void func(void);

     void   (*pf1)(void)  = func;
     void  (**pf2)(void)  = &pf1;
     void (***pf3)(void)  = &pf2;

     (*pf1)();
     (**pf2)();
     (***pf3)();
     printf("\n=========================\n");
     printf("    func : %X \n",  func);
     printf("     pf1 : %X \n",   pf1);
     printf("    *pf1 : %X \n",  *pf1);
     printf("     pf2 : %X \n",   pf2);
     printf("    *pf2 : %X \n",  *pf2);
     printf("   **pf2 : %X \n", **pf2);
     printf("     pf3 : %X \n",   pf3);
     printf("    *pf3 : %X \n",  *pf3);
     printf("   **pf3 : %X \n", **pf3);
     printf("  ***pf3 : %X \n",***pf3);
     printf("\n-------------------------\n");
     printf("    func : %X \n",    func);
     printf("   *func : %X \n",   *func);
     printf("  **func : %X \n",  **func);
     printf(" ***func : %X \n", ***func);
     return 0;
}

void func(void)
{
     printf("Function....Hello World!\n");
}
```

นิพจน์	*ค่าของนิพจน์*
func	178
pf1	178
*pf1	178
pf2	2098
*pf2	178
**pf2	178
pf3	2096
*pf3	2098
**pf3	178
***pf3	178
func	178
*func	178
**func	178
***func	178

ตารางแสดงตัวอย่างค่าของนิพจน์ที่ได้จากทำงานของโปรแกรม



6.8 การสร้างฟังก์ชันที่ให้ที่อยู่ของฟังก์ชันกลับคืน

ในตัวอย่างที่แล้วเราได้เห็นวิธีการนิยามพอยน์เตอร์ที่ชี้ไปยังฟังก์ชันที่ให้ข้อมูลพื้นฐานแบบ `int` กลับคืนและต่อไปนี้เราจะทำความเข้าใจโดยอาศัยตัวอย่างในเรื่องของการแจ้งใช้ รวมถึงวิธีใช้พอยน์เตอร์ที่ชี้ไปยังฟังก์ชันซึ่งให้พอยน์เตอร์ที่ชี้ไปยังฟังก์ชันกลับคืน

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

void f1(char *s)
{
     char *p = s;
```

```
        while (*p)
          printf("%c", toupper(*p++));
        printf("\n");
}

void f2(char *s)
{
        char *p = s;
        while (*p)
           printf("%c", tolower(*p++));
        printf("\n");
}

/* f is a function that returns a pointer to
 * a void-function with (char *) as a parameter.
 */
void  (*f(int i)) (char *)
{
        extern void f1 (char *);
        extern void f2 (char *);

        return ((i==1) ? f1 : f2);
}

int main()
{
        (f(1))("Hello World!");
        (f(2))("Hello World!");
        printf("--------------------\n");
          f(1)("Hello World!");
          f(2)("Hello World!");
        printf("--------------------\n");
         (*f1)("Hello World!");
         (*f2)("Hello World!");
        printf("--------------------\n");
          f1("Hello World!");
          f2("Hello World!");
        printf("--------------------\n");
        return 0;
}
```

ในตัวอย่างนี้ `f()` เป็นฟังก์ชันที่ให้พอยน์เตอร์ที่ชี้ไปยังฟังก์ชันที่มีพารามิเตอร์แบบ `char *` และไม่ให้ค่าใดกลับคืน(`void`) และสำหรับ `f1()` ก็ต้องการพารามิเตอร์หนึ่งตัวแบบ `int` สำหรับใช้ภายในฟังก์ชันซึ่งมีลักษณะดังนี้

```
void (* f(int i))(char *);
```

โดยที่ประโยคคำสั่ง

```
(f(1))("Hello World!");
```

```
(f(2))("Hello World!");
```

และ

```
f(1)("Hello World!");
f(2)("Hello World!");
```

ให้ผลเหมือนกัน และ หมายถึง

```
(*f1)("Hello World!");
(*f2)("Hello World!");
```

หรือ

```
f1("Hello World!");
f2("Hello World!");
```

ตามลำดับ ซึ่งเราจะเห็นได้ว่า `1` หรือ `2` เป็นพารามิเตอร์แบบ `int` สำหรับฟังก์ชัน `f(int)(char *)` และ `f(1)(char *)` คือฟังก์ชัน `f1(char *)` และ `f(2)(char *)` คือฟังก์ชัน `f2(char *)` ตามลำดับ ในขณะที่ `"Hello World!"` เป็นพารามิเตอร์สำหรับฟังก์ชัน `f1()` หรือ `f2()` และ ฟังก์ชัน `f1()` พิมพ์ข้อความที่เป็นพารามิเตอร์โดยเปลี่ยนให้เป็นตัวพิมพ์ใหญ่และฟังก์ชัน `f2()` ก็ทำงานเหมือนกับ `f1()` เพียงแต่พิมพ์ข้อความให้เป็นตัวพิมพ์เล็กทั้งหมด

เราสรุปได้ว่า เราสามารถใช้พอยน์เตอร์ที่ชี้ไปยังที่อยู่ของฟังก์ชันใดๆที่มีรูปแบบตามที่เราได้กำหนดไว้ในการผ่านเป็นพารามิเตอร์ของฟังก์ชันหรือเป็นพอยน์เตอร์ที่ฟังก์ชันให้เป็นค่ากลับคืนจากการเรียกใช้

การใช้พอยน์เตอร์กับฟังก์ชัน ก็มีเงื่อนไขเช่นเดียวกันกับการใช้พอยน์เตอร์กับตัวแปร ซึ่งก็คือการกำหนดค่าของพอยน์เตอร์จะต้องเป็นไปตามแบบที่ๆเราได้กำหนดไว้ และเราก็ไม่สามารถกำหนดให้พอยน์เตอร์ตัวหนึ่งชี้ไปยังแหล่งข้อมูลเดียวกันกับที่พอยน์เตอร์อีกตัวหนึ่งกำลังชี้ไปได้ ถ้าพอยน์เตอร์ทั้งสองต่างแบบกัน ซึ่งจะเห็นได้จากตัวอย่างต่อไปนี้

```
extern int   f1();       /* function prototype */
extern int (*f2())();    /* function prototype */

int (*f3())()
{
     int (*pf)();        /* pointer to a function
                            that returns an int. */

     pf = f1;           /* O.K.    */
     pf = f2;           /* ILLEGAL */
     return f2;         /* ILLEGAL */
}
```

ฟังก์ชัน f3() เป็นฟังก์ชันที่ให้พอยน์เตอร์ที่ชี้ไปยังที่อยู่ของฟังก์ชันที่ให้ค่าแบบ int ฟังก์ชัน f1() เป็นฟังก์ชันที่ให้ค่าแบบ int และ ฟังก์ชัน f2() เป็นฟังก์ชันที่มีรูปแบบเหมือน f3() ในตัวอย่างนี้บรรทัดคำสั่งที่เป็นปัญหาคือ

```
pf = f2;            /* ILLEGAL */
return f2;          /* ILLEGAL */
```

เพราะ pf เป็นพอยน์เตอร์ที่ชี้ไปยังฟังก์ชันที่ให้ค่าแบบ int เช่น ฟังก์ชัน f1() แต่ f2() เป็นฟังก์ชันอีกลักษณะหนึ่ง ดังนั้นเราจึงไม่สามารถกำหนดให้ pf ชี้ไปยังที่อยู่ของฟังก์ชัน f2()ได้

6.9 การใช้ typedef สร้างแบบของพอยน์เตอร์

เราสามารถกำหนดแบบของพอยน์เตอร์โดยใช้ typedef ได้ ซึ่งช่วยให้เราเขียนคำสั่งแจ้งใช้พอยน์เตอร์ได้สั้นและสะดวกมากขึ้น ตัวอย่างเช่น

```
typedef  char *      p_char;
typedef  int *       p_int;
typedef  float *     p_float;
typedef  double *    p_double;
```

เวลาใช้เราก็เขียนเฉพาะตัวระบุชื่อที่เรานิยามขึ้น เช่น

```
p_char    ptr;
p_int     ip1, ip2;
p_double  dp1, dp2, dp3;
```

หรือ อาจจะเป็นพอยน์เตอร์ที่ชี้ไปยังพอยน์เตอร์ก็ได้ เช่น

```
typedef  char **    pp_char;
typedef  int **     pp_int;
typedef  float **   pp_float;
typedef  double **  pp_double;
```

นอกจากนี้เราสามารถใช้ typedef กำหนดแบบของพอยน์เตอร์ที่ชี้ไปยังฟังก์ชันก็ได้ เช่น

```
typedef int  (*func_1)(int);
typedef int  (*func_2)(int *, int *);
typedef void (*func_3)(void *);
```

ถ้าเราแจ้งใช้พอยน์เตอร์ดังต่อไปนี้

```
func_1  pf1;
func_2  pf2;
func_3  pf3;
```

ก็จะหมายความว่า pf1 เป็นพอยน์เตอร์ที่ทำหน้าที่ชี้ไปยังที่อยู่ของฟังก์ชันที่มีพารามิเตอร์แบบ int หนึ่งตัว และให้ค่าแบบ int กลับคืน pf2 เป็นพอยน์เตอร์ที่ชี้ไปยังที่อยู่ของฟังก์ชันที่มีพารามิเตอร์เป็นพอยน์เตอร์แบบ int สองตัวและให้ค่าแบบ int กลับคืน pf3 เป็นพอยน์เตอร์ที่ชี้ไปยังที่อยู่ของฟังก์ชันที่มีพารามิเตอร์เป็นพอยน์เตอร์แบบ void และไม่ให้ค่าใดกลับคืน

ความเข้าใจในการใช้พอยน์เตอร์ถือว่าเป็นสิ่งสำคัญ เพราะเป็นพื้นฐานสำคัญที่นำไปสู่ความรู้ความเข้าใจในการเขียนโปรแกรมภาษาซีระดับสูง เช่น การใช้อาร์เรย์และสายอักขระเป็นต้น ซึ่งเราจะได้เรียนรู้ในบทต่อๆไป

แบบฝึกหัดท้ายบท

1. จงหาค่าของนิพจน์ต่อไปนี้ เมื่อโปรแกรมทำงานจบบรรทัดคำสั่งที่มีหมายเลขอยู่ในตารางข้างล่างนี้

บรรทัดที่	นิพจน์	ค่าของนิพจน์
3	(p1==p2) (*p1==*p2)	
5	(p1==p2) (*p1==*p2)	
6	(p1==p2) (*p1==*p2)	
7	(p1==p2) (*p1==*p2)	

```
/* 1 */    char x = 'A', y = 'B';
/* 2 */    char *p1 = &x;
/* 3 */    char *p2 = &y;
/* 4 */
/* 5 */     p1 =  p2;
/* 6 */     p1 = &x;
/* 7 */    *p1 = *p2;
```

2. ถ้า a มีหน่วยความจำอยู่ที่หมายเลข 20D8 และมีขนาดของข้อมูลเท่ากับสองไบต์

```
int a = 10;
int *ptr = &a;
int **ptrptr = &ptr;
```

จงหาค่าของนิพจน์ต่อไปนี้ (ถ้านิพจน์เหล่านี้ถูกต้องตามหลักไวยากรณ์ของภาษาซี)

1) `*ptr`

2) `&*ptr`

3) `&(*ptr)`

4) `*(&ptr)`

5) `*ptrptr`

6)`**(&ptrptr)`

3. จงอธิบายความแตกต่างระหว่างฟังก์ชันทั้งสอง และตรวจสอบดูว่า ฟังก์ชันใดที่ทำงานไม่ถูกต้อง ถ้าเราต้องการใช้ฟังก์ชันใดฟังก์ชันหนึ่ง ตามรูปแบบข้างล่างนี้ในการสลับค่าของตัวแปรสองตัวและให้เหตุผลด้วยว่า ทำไมจึงทำงานไม่ถูกต้อง

```
void swap1(int *a, int *b)
{ int t = *a; *a = *b; *b = t; }

void swap2(int *a, int *b)
{ int *t = a; a = b; b = t; }
```

4. จงสร้างฟังก์ชันชื่อ mem_swap() ซึ่งมีรูปแบบดังต่อไปนี้

```
void mem_swap (void *p1, void *p2, unsigned n);
```

และมีหน้าที่สลับเปลี่ยนค่าระหว่างพื้นที่ของหน่วยความจำสองแห่งที่พอยน์เตอร์ p1 และ p2 ชี้ไปยังหมายเลขที่อยู่เริ่มต้นของพื้นที่หน่วยความจำทั้งสองนี้ตามลำดับ โดยที่พารามิเตอร์ n เป็นตัวกำหนดความยาวของพื้นที่หน่วยความจำทั้งสองในหน่วยของไบต์ ซึ่งหมายความว่าจะมีการสลับเปลี่ยนข้อมูลทั้งหมด n ไบต์

5. จงให้เหตุผลว่า ทำไมการแจ้งใช้และติดตั้งค่าของตัวแปรพอยน์เตอร์ dptr จึงไม่ถูกต้อง

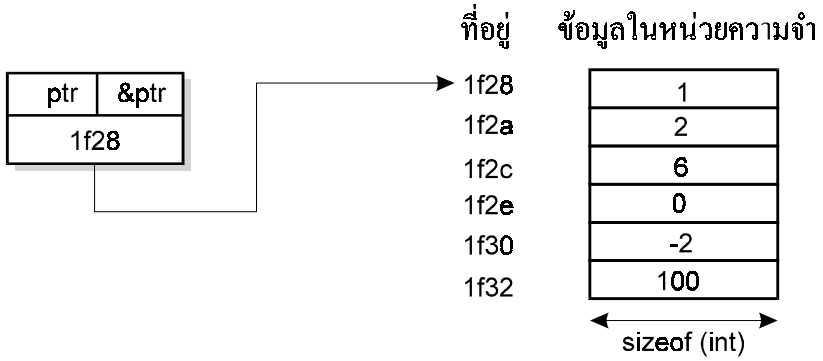
```
const double x = 10.35;
double *dptr = &x;
printf ("x = %g\n", *dptr);
```

6. จงเขียนฟังก์ชัน pack()ที่มีหน้าที่เก็บพารามิเตอร์สี่ตัวที่มีแบบข้อมูลเป็น char ลงในข้อมูลแบบ long ขนาด 32 บิต โดยข้อมูลแบบ char ตัวแรกเก็บไว้ที่บิตหมายเลข 0 ถึง 7 ข้อมูลแบบ char ตัวที่สองเก็บไว้ที่บิตหมายเลข 8 ถึง 15 ไปเรื่อยๆจนครบทั้งสี่ตัว และฟังก์ชัน unpack() ที่มีหน้าที่อ่านข้อมูลแบบ long ที่เก็บข้อมูลแบบ char สี่ตัวไว้ในหน่วยความจำของตนเอง ฟังก์ชันทั้งสองจะมีรูปแบบตามลำดับต่อไปนี้

```
void pack  (long *x, char  a, char  b, char  c, char  d);
void unpack (long x, char *a, char *b, char *c, char *d);
```

ฟังก์ชัน pack() จะต้องเก็บค่าของพารามิเตอร์ a , b, c, และ d ไว้ในที่อยู่ที่พอยน์เตอร์ x กำลังชี้ไป ฟังก์ชัน unpack() จะต้องอ่านค่าของพารามิเตอร์ x และ เก็บค่าของข้อมูลแต่ละตัวที่อ่านได้จากค่าของ x ไว้ในที่อยู่ที่พอยน์เตอร์ a , b, c, และ d กำลังชี้ไปตามลำดับ

7. จากภาพพอยน์เตอร์ ptr เป็นพอยน์เตอร์แบบ int ที่ชี้ไปยังพื้นที่ของหน่วยความจำที่เก็บข้อมูลแบบ int จำนวนหกตัวเรียงต่อกันไป และถ้ากำหนดให้ขนาดของข้อมูลแบบ int มีขนาดเท่ากับ 2 ไบต์ จงหาค่าของ x ซึ่งเป็นตัวแปรแบบ int จากประโยคคำสั่งต่อไปนี้




```
ptr = (int *)0x1f28;
x = ++*ptr;
ptr = (int *)0x1f28;
x = *++ptr;
ptr = (int *)0x1f28;
x = (*ptr)++;
ptr = (int *)0x1f28;
x = *(++ptr);

ptr = (int *)0x1f28;
x = *(ptr++);
x = *ptr++;
x = (*ptr++)++;
x = (*ptr++)++;
x = *--ptr;
x = *(ptr-1);
```

8. จงหาว่าประโยคคำสั่งหรือนิพจน์ใดในโปรแกรมต่อไปนี้ไม่ถูกต้องตามหลักไวยากรณ์ในภาษาซี

```
{
  extern int f1(void)
  int (*pf)(void) = f1;

  pf = f1();
  pf = &f1();
  pf = (*f1)();
  pf = (**f1);
  pf = &f1;
}
```

9. จงยกตัวอย่างรูปแบบของพอยน์เตอร์ที่ชี้ไปยังฟังก์ชันซึ่งมีคุณสมบัติ ตามหัวข้อต่อไปนี้

1) มีพารามิเตอร์เป็นพอยน์เตอร์ที่ชี้ไปยังฟังก์ชันซึ่งมีข้อมูลแบบ `double` เป็นพารามิเตอร์

2) และให้พอยน์เตอร์ที่ชี้ไปยังฟังก์ชันที่มีข้อมูลแบบ `int` เป็นพารามิเตอร์และให้ค่าแบบ `int` กลับคืน