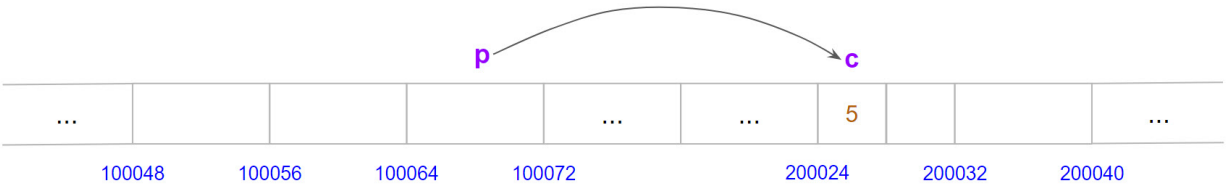
สัปดาห์ที่ 7 Pointers และ Dynamic Memory Allocation

เรียบเรียงโดย ชาคริต วัชโรภาส

วิชา 01418113 Computer Programming

1. Pointers and Addresses

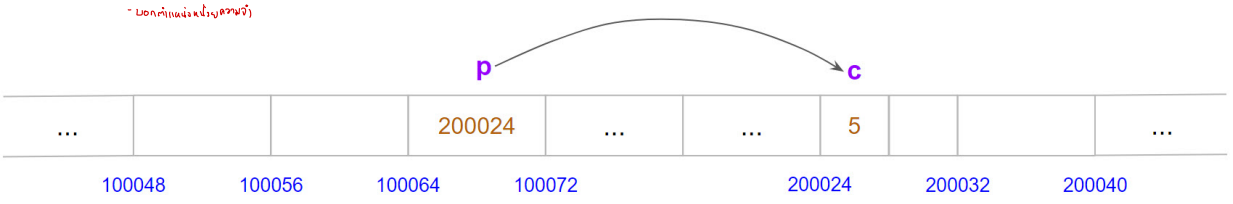
- ภาษา C มีความสามารถเด่นที่แตกต่างจากภาษาโปรแกรมโดยทั่วไปคือ โปรแกรมเมอร์สามารถเข้าถึงและจัดการหน่วยความจำ (memory) ได้โดยตรง ด้วยการใช้ pointers
- Pointer เป็นตัวแปรที่ใช้เก็บค่าตำแหน่งหรือ address ในหน่วยความจำ (ที่ต้องการอ้างถึง)
- ค่าของตำแหน่งหรือ address ในหน่วยความจำก็คือค่าจำนวนเต็มบวกที่เริ่มจาก 0 นั่นเอง
- แต่ปกติเวลาเราใช้งานตัวแปร pointer เพื่อเก็บค่าตำแหน่งในหน่วยความจำ เรามักจะบอกกับคอมไพเลอร์ให้ทราบว่า ตำแหน่งในหน่วยความจำนั้นมีการเก็บข้อมูลประเภทใดไว้



- จากรูปด้านบน สมมติเราประกาศตัวแปร **c** ที่มีประเภทข้อมูลเป็น *int* และเก็บค่า 5 เอาไว้ ตำแหน่งในหน่วยความจำของตัวแปร **c** อยู่ในตำแหน่งในหน่วยความจำดังรูป
- จากรูป เรากำหนดให้ตัวแปร pointer **p** อ้างอิงหรือชี้ (point) ไปที่หน่วยความจำที่ตัวแปร **c** ถูกจัดเก็บอยู่
- หากเราต้องการเข้าถึงหน่วยความจำที่ถูกอ้างอิงผ่านค่าในตัวแปร **p** หน่วยความจำบริเวณนั้นจะเก็บข้อมูลประเภท int (ถ้าอ่านแล้วไม่เข้าใจ ให้ถามผู้สอน)
- การเขียนโค้ดที่จะให้ตัวแปร pointer **p** อ้างอิงไปถึงตำแหน่งในหน่วยความจำที่เก็บค่าของตัวแปร **c** สามารถเขียนได้ในลักษณะข้างล่างนี้

```
int c;
int *p;

p = &c;
```



2. Pointer Operators

- ในการใช้ pointer เรามีตัวดำเนินการอีก 2 ตัวถูกนิยามการใช้งานเพิ่มขึ้นมาจากตัวดำเนินการอื่นที่เราเคยเรียนผ่านมาแล้ว

  1. & เรียกว่า address operator ซึ่งเป็น unary operator ที่ต้องการเพียง 1 operand โดยที่ operand นั้นมักอยู่ในรูปของตัวแปร
     - กล่าวคือ & เป็น operator ที่ใช้หาค่าตำแหน่งในหน่วยความจำของ operand ดังที่เราเคยเห็นผ่านตัวอย่างที่ใช้ในคำสั่ง

```
p = &c;
```

ซึ่งเป็นการกำหนดค่าให้กับตัวแปร p ให้มีค่าเท่ากับ address ของตัวแปร c

2. * เรียกว่า dereferencing operator หรือ indirection operator ซึ่งเป็น unary operator โดยที่ * ถูกใช้ในการเข้าถึงข้อมูลที่ถูกอ้างอิงโดยค่าตำแหน่งในหน่วยความจำที่มีค่าเป็น operand

In [ ]:
```c
#include <stdio.h>

int main()
{  int c = 5;
   int *p;

   p = &c;
   printf(" c = %d\n", c);
   printf("*p = %d\n", *p);
}
```

ลองดูโค้ดข้างล่างนี้

In [ ]:
```c
#include <stdio.h>

int main()
{  int x = 1, y = 2, z[10];
   int *ip;         // ip is a pointer to int

   ip = &x;         // ip now points to x
   y = *ip;
   *ip = 0;
   ip = &z[0];      // ip now points to z[0]

   // เมื่อโปรแกรมทำงานถึงบรรทัดนี้ ตัวแปร x และ y มีค่าเป็นเท่าไร
}
```

In [ ]:

```c
#include <stdio.h>

int main ()
{  int *a_ptr, a;

   a = 10;
   a_ptr = &a;

   printf("(1) a = %d, *a_ptr = %d\n", a, *a_ptr);

   *a_ptr *= 2;

   printf("(2) a = %d, *a_ptr = %d\n\n", a, *a_ptr);

   printf("The address of a is %p\n"
          "The value of a_ptr is %p\n\n", &a, a_ptr);

   printf("&*a_ptr = %p\n", &*a_ptr);
   printf("*&a_ptr = %p\n", *&a_ptr);
}
```

2.1 ฟังก์ชัน scanf()

- ฟังก์ชัน scanf() เปรียบเทียบได้คล้ายกับฟังก์ชัน printf() เพียงแต่ทำงานในลักษณะตรงกันข้าม ตรงที่ printf() เป็นการแสดงผลออกไปที่ standard output (เช่น หน้าจอ) แต่ scanf() เป็นการรับข้อมูลเข้ามาในโปรแกรมผ่าน standard input (เช่น คีย์บอร์ด)
- ตัวอย่างการใช้งาน scanf() ที่รับค่าจำนวนเต็มเข้ามาในโปรแกรม

```c
int input;

scanf("%d", &input);
```

```
SCANF(3P)                  POSIX Programmer's Manual                 SCANF(3P)

PROLOG
       This  manual  page is part of the POSIX Programmer's Manual.  The Linux
       implementation of this interface may differ (consult the  corresponding
       Linux  manual page for details of Linux behavior), or the interface may
       not be implemented on Linux.

NAME
       scanf - convert formatted input

SYNOPSIS
       #include <stdio.h>

       int scanf(const char *restrict format, ...);
```

3. Pointers กับ Arrays

- อาเรย์สามารถใช้เก็บข้อมูลมากกว่า 1 จำนวนที่เป็นประเภทเดียวกันเข้าไว้ด้วยกันภายใต้ตัวแปรเดียวกันได้
- ต.ย.

```
int a[10];
```

- a[i] อ้างอิงขึ้นข้อมูลตัวที่ i ในอาเรย์ a โดยข้อมูลตัวแรกในอาเรย์คือ a[0]
- ชื่อตัวแปรอาเรย์เป็นนิพจน์ที่บอกถึงค่า base address ของอาเรย์ กล่าวคือ a จะให้ค่าตำแหน่งในหน่วยความจำที่ใช้เก็บข้อมูลตัวแรกในอาเรย์
- ดังนั้น a และ &a[0] จึงให้ค่าเท่ากัน
- รวมถึง *(a + i) และ a[i] ที่เป็นนิพจน์ที่บอกถึงสิ่งเดียวกัน

---

4. การจำลองการเรียกฟังก์ชันแบบ Call by Reference

- โดยทั่วไป การส่งผ่านค่าพารามิเตอร์ไปยังฟังก์ชันสามารถทำได้ 2 ลักษณะ ซึ่งเรียกว่า call by value และ call by reference ภาษา c ไม่สามารถทำได้
- ภาษา C การส่งค่าพารามิเตอร์ไปยังฟังก์ชันผ่านการเรียกใช้ฟังก์ชันล้วนเป็น call by value แต่เราสามารถใช้ pointer ช่วยจำลองการส่งผ่านค่าแบบ call by reference ได้

In [ ]:

```c
#include <stdio.h>

void squareByValue(int);          /* function prototype */
void squareByReference(int *);    /* function prototype */

int main()
{  int a = 3;
   int b = 4;

   printf("ค่าของ a คือ %d\n", a);
   squareByValue(a);
   printf("ค่าของ a หลังเรียกฟังก์ชัน squareByValue(a) คือ %d\n\n", a);

   printf("ค่าของ b คือ %d\n", b);
   squareByReference(&b);
   printf("ค่าของ b หลังเรียกฟังก์ชัน squareByReference(&b) คือ %d\n", b);
}

void squareByValue(int val)
{
   val = val * val;
}

void squareByReference(int *val)
{
   *val = *val * *val;
}
```

In [ ]:

```c
#include <stdio.h>

void swap(int x, int y)   // ถูกหรือไม่
{  int temp;

   temp = x;
   x = y;
   y = temp;
}

int main()
{  int x = 3, y = 4;

   printf("ก่อน x = %d, y = %d\n", x, y);
   swap(x, y);
   printf("หลัง x = %d, y = %d\n", x, y);
}
```

In [ ]:

```c

```

In [ ]:

```c
#include <stdio.h>

void swap(int *px, int *py)
{  int temp;

   // เติมโค้ดให้หน่อย
}

int main()
{  int x = 3, y = 4;

   printf("ก่อน x = %d, y = %d\n", x, y);
   swap(&x, &y);
   printf("หลัง x = %d, y = %d\n", x, y);
}
```

---

5. การใช้ const กับ pointers

- ใน ANSI C Standard คีย์เวิร์ด const สามารถใช้ร่วมกับ pointer เพื่อแจ้งให้คอมไพเลอร์ทราบว่าค่าของตัวแปรนั้นไม่สามารถถูกแก้ไขได้
- *ข้อควรระวัง* การใช้ const ร่วมกับ pointer ในบางระบบอาจไม่เป็นไปตามที่ ANSI C กำหนดไว้

In [ ]:

```c
#include <stdio.h>

int f1(const int *arr, const int len)
{  int i, sum;


   for (i=sum=0; i < len; i++)
      sum += arr[i];

   // ลองเอาคอมเม้นต์คำสั่งข้างล่างนี้ออกแล้วดูว่าคอมไพเลอร์ยอมหรือไม่
   // *(arr+5) = 10;

   return sum;
}

int main ()
{  int data[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
   int n=10, sum;

   sum = f1(data, n);
   printf("    sum = %d\n",sum);
   printf("data[5] = %d\n",data[5]);
}
```

---

6. นิพจน์ใน pointer และการใช้ตัวดำเนินการคณิตศาสตร์กับ pointer

(Expressions and Pointer Arithmetic)

ลองพิจารณาโค้ดต่อไปนี้

```
int v[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int *pv;

pv = v;
pv = pv + 2;
printf("%d\n, pv - v);
pv++;
printf("%d\n, pv - v);
```

- ตัวแปร pointer สามารถเป็น operand ให้กับนิพจน์เหล่านี้ได้
  - นิพจน์ทางคณิตศาสตร์
  - นิพจน์ในการกำหนดค่า
  - นิพจน์ในการเปรียบเทียบ
  - หากแต่ว่า ไม่ใช่ทุกตัวดำเนินการที่ใช้ในนิพจน์เหล่านี้จะสามารถใช้ร่วมกับ pointer ได้ทั้งหมด

ความหมายของนิพจน์ที่มี pointer กับตัวดำเนินการคณิตศาสตร์

- โดยทั่วไปแล้ว นิพจน์ทางคณิตศาสตร์ เช่น 2000 + 4 จะมีค่าเท่ากับ 2004
- แต่เมื่อนำมาใช้กับ pointer แล้วจะไม่เป็นเช่นนั้น (ซะทีเดียว)
- ตัวอย่างเช่น

```
float f[4] = {0.0, 1.0, 2.0, 3.0};
float *pf;

pf = &f[0];
pf = pf + 2;
```

- นิพจน์ **pf = pf + 2** ไม่ได้หมายความว่าให้เพิ่มค่า pf ขึ้นไปอีก 2 หน่วย
- นิพจน์ **pf = pf + 2** หมายถึงให้เพิ่มค่า pf ขึ้นไปเป็นจำนวน 2 เท่าของขนาดประเภทข้อมูลที่ pf อ้างถึง (ซึ่งในกรณีนี้ pf อ้างถึงประเภทข้อมูลที่เป็น float ซึ่งมีขนาด 4 byte)
- ดังนั้น 2 เท่าของขนาดประเภทข้อมูล float จะมีค่าเป็น 2*4 = 8 byte นั่นเอง
- นิพจน์ **pf = pf + 2** จะเพิ่มค่า pf ขึ้นไปอีก 8 หน่วยนั่นเอง

ทดลองรันโค้ดข้างล่างนี้ดู

In [ ]:
```c
#include <stdio.h>

int main ()
{  float f[4] = {0.0, 1.0, 2.0, 3.0};
   float *pf;

   pf = &f[0];
   printf("before pf = %p\n", pf);
   pf = pf + 2;
   printf("after1 pf = %p\n", pf);
   pf = pf - 1;
   printf("after2 pf = %p\n", pf);
}
```

ดูตัวอย่าง fgets.c

7. การใช้ pointer อ้างไปที่ฟังก์ชัน (Pointers to Functions)

- เราสามารถใช้ตัวแปร pointer อ้างไปยังตำแหน่งในหน่วยความจำ (base address) ของฟังก์ชันที่ต้องการได้ (ฟังก์ชันใดที่จะถูกเรียกใช้งานจะถูกโหลดเข้ามาในหน่วยความจำ)
- โดยปกติแล้ว ชื่อฟังก์ชันเป็นนิพจน์ที่ให้ค่า base address ของฟังก์ชัน (ซึ่งมีลักษณะเดียวกับอาเรย์ที่ชื่ออาเรย์เป็นนิพจน์ที่ให้ค่า base address ของอาเรย์)

การประกาศตัวแปร pointer ที่จะใช้อ้างไปยังฟังก์ชันในรูปแบบที่ต้องการ

- อ่านหัวข้อแล้วคงงง
- ลองทบทวนดู การประกาศตัวแปรมีขึ้นเพื่อระบุประเภทข้อมูลให้กับตัวแปรก่อนการใช้งาน
- แล้วประเภทข้อมูลที่จะใช้เก็บตำแหน่งฟังก์ชันมีหน้าตาเป็นอย่างไรหละ เพราะเราไม่เคยเห็นมาก่อน
- ส่วนนี้เป็นสิ่งที่เราจะต้องรู้เพิ่มเติม
- สมมติสถานการณ์ว่า เราต้องการประกาศตัวแปร pfunc ที่จะใช้เก็บ base address ของฟังก์ชันที่มีการส่งค่าของประเภทข้อมูล double กลับไปยังผู้เรียก และมีการรับค่าพารามิเตอร์ 2 ตัวที่มีประเภทข้อมูลเป็น int และ float ตามลำดับ
- การประกาศตัวแปร pfunc จะมีลักษณะการเขียนดังนี้

```
double (*pfunc)(int, float)
```

ทดลองรันโค้ดข้างล่างนี้ดู

In [ ]:

```c
#include <stdio.h>

double f1(int x, float y)
{
   return x + y;
}

double f2(int x, float y)
{
   return x * y;
}

int main ()
{  double (*pfunc)(int, float);

   pfunc = f1;
   printf("Return %lf\n", (*pfunc)(3, 2.5));
   pfunc = f2;
   printf("Return %lf\n", (*pfunc)(3, 2.5));
}
```

การนำมาใช้งานโดยทั่วไป

In [ ]:

```c
#include <stdio.h>

void action(double (*pfunc)(int, float), int x, float y)
{
   printf("Return %lf\n", (*pfunc)(x, y));
}

double f1(int x, float y)
{
   return x + y;
}

double f2(int x, float y)
{
   return x * y;
}

int main ()
{
   action(f1, 3, 2.5);
   action(f2, 3, 2.5);
}
```

8. Complicated Declarations

- ลองพิจารณาการประกาศทั้ง 2 นี้

```
int *f1();     // เป็นการประกาศฟังก์ชัน f1 ที่ส่งค่า pointer ไปยัง int กลับ
```

```
int (*f2)();  // เป็นการประกาศตัวแปร pointer f2 ที่สามารถเก็บค่าตำแหน่งของฟังก์ชันที่ส่งค่า int กลับ
```

```
int *(*f3)();  // เป็นการประกาศตัวแปร pointer f3 ที่สามารถเก็บค่าตำแหน่งของฟังก์ชันที่ส่งค่า pointer ไปยัง int กลับ
```

```
int *daytab[13];  // เป็นการประกาศตัวแปรอาเรย์ daytab (จำนวนสมาชิก 13 ตัว) ที่เก็บ pointer ไปยัง int
```

---

9. การจองหน่วยความจำ (Dynamic Memory Allocation)

- ที่ผ่านมา เราจองหน่วยความจำผ่านการประกาศตัวแปร โดยที่การประกาศตัวแปรจะเป็นการจองและใช้พื้นที่ในหน่วยความจำเพื่อใช้จัดเก็บค่าของตัวแปร
- ในการจองที่ผ่านมา ขนาดพื้นที่จะคงที่และไม่สามารถเปลี่ยนแปลงได้ตลอดการทำงานของโปรแกรม
- แต่ในการทำงานโดยทั่วไปแล้ว บางครั้งเราต้องการความสามารถในการจองพื้นที่ในหน่วยความจำตามขนาดที่เราต้องการในขณะที่โปรแกรมกำลังทำงานอยู่ได้ รวมถึงความสามารถในการคืนพื้นที่ที่เคยจองไว้หากเราไม่ได้งานอีกต่อไป
- เราสามารถใช้ฟังก์ชัน *malloc()* และ *free()* เพื่อจองและคืนพื้นที่ในหน่วยความจำได้
- จากที่เรียนผ่านมา ตัวแปรที่มีคลาสเป็น auto จะถูกจัดเก็บอยู่ในหน่วยความจำที่เป็นส่วนของ Stack segment และตัวแปร global จะถูกจัดเก็บอยู่ในหน่วยความจำที่เป็นส่วนของ Data segment
- หน่วยความจำที่ถูกจองด้วย malloc() จะเก็บอยู่ในหน่วยความจำที่เป็นส่วนของ Heap segment

การใช้ malloc()

```
int *data;

data = (int *)malloc(1000 * sizeof(int));
```

- พารามิเตอร์ของ malloc() จะเป็นจำนวน byte ที่ใช้ในการจองพื้นที่
- ประเภทข้อมูลในการส่งกลับของ malloc() เป็น (void *) ดังนั้นหากเราต้องการพื้นที่เพื่อใช้จัดเก็บข้อมูลประเภทอื่น เราจำเป็นต้อง explicit cast ให้เป็นประเภทข้อมูลที่ต้องการใช้
- ค่าที่ malloc() ส่งกลับเป็นตำแหน่งในหน่วยความจำ (base address) ที่ระบบปฏิบัติการจองพื้นที่ในหน่วยความจำไว้ให้
- หากหน่วยความจำในระบบไม่มีเพียงพอให้จองได้ malloc() จะส่งค่า NULL กลับมายังผู้เรียก

การใช้ free()

```
free(data);
```

- พารามิเตอร์ของ free() จะเป็นตำแหน่งในหน่วยความจำ (base address) ที่เราต้องการคืนพื้นที่ให้กับระบบ

ตระกูลฟังก์ชันการจองพื้นที่

```
#include <stdlib.h>

void *malloc(size_t n);
void *calloc(size_t n, size_t size);
void *realloc(void *ptr, size_t size);
void free(void *ptr); คืนค่า
```

10. Segmentation fault (core dumped)

- หากเราพยายามเข้าถึงหน่วยความจำที่เราไม่ได้จองไว้หรือหน่วยความจำที่นอกเหนือจากตัวแปรที่เราประกาศไว้ เรามีโอกาศที่จะถูกระบบปฏิบัติการจะหยุดการทำงานของโปรแกรมของเรา โดยในระบบ UNIX มักจะแสดงข้อความว่า Segmentation fault (core dumped) ออกมาบนหน้าจอให้โปรแกรมเมอร์ได้รับรู้

```
#include <stdio.h>

int main()
{  int data[] = {1, 2, 3, 4, 5};
   int i;

   for (i=0; i < 5000; i++)
      printf("Data element #%d is %d\n", i, data[i]);
}
```