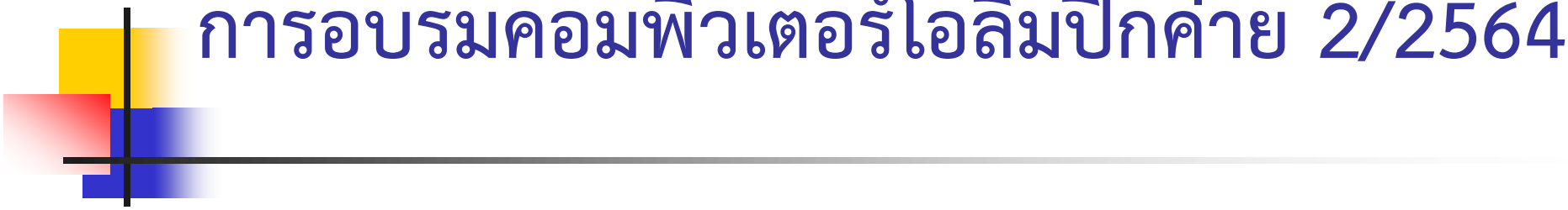
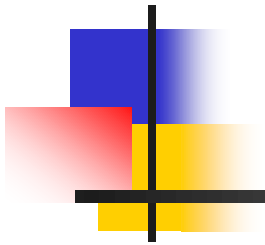


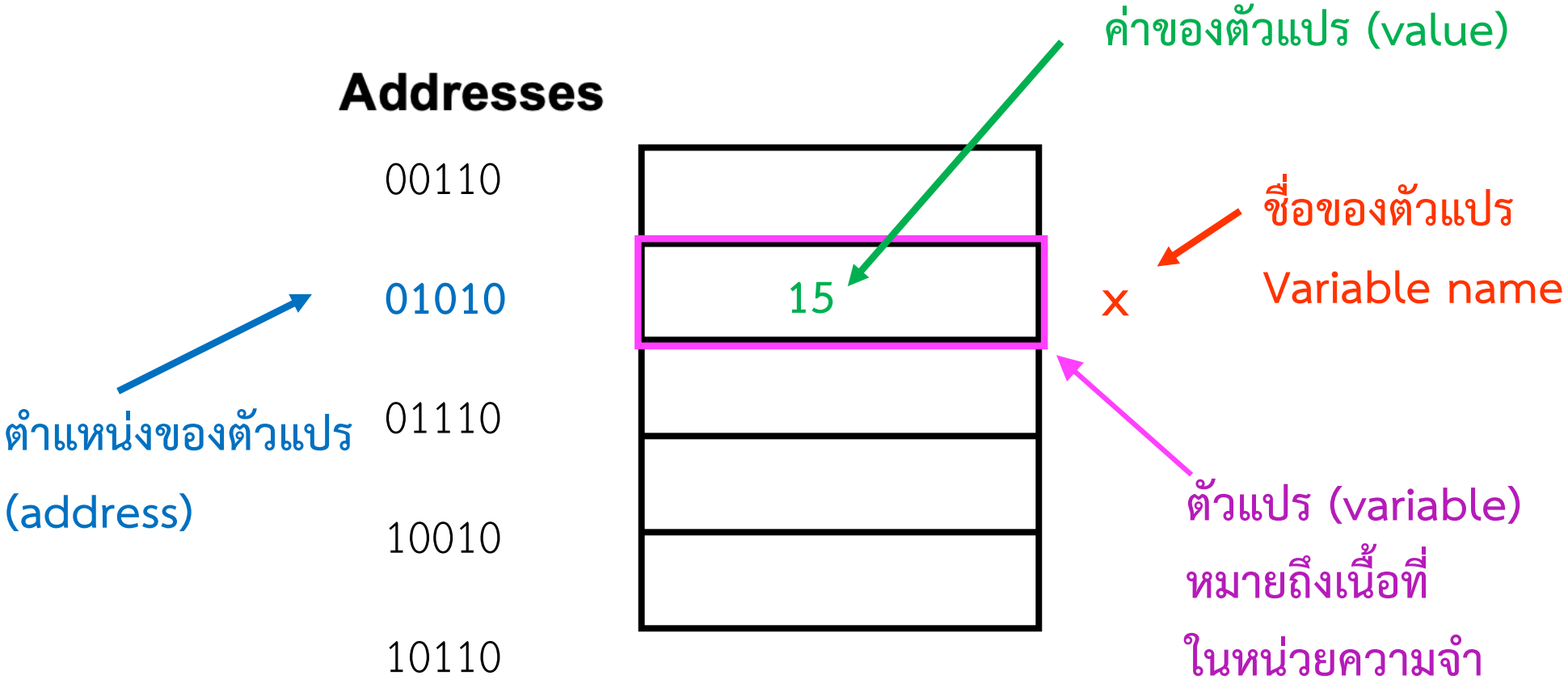
การอบรมคอมพิวเตอร์โอลิมปิกค่าย 2/2564

- ทบทวนการใช้ pointer ในภาษา C/C++
- ข้อมูลนามธรรม (Data Abstraction)



ทบทวนการใช้ Pointer ในภาษา C/C++

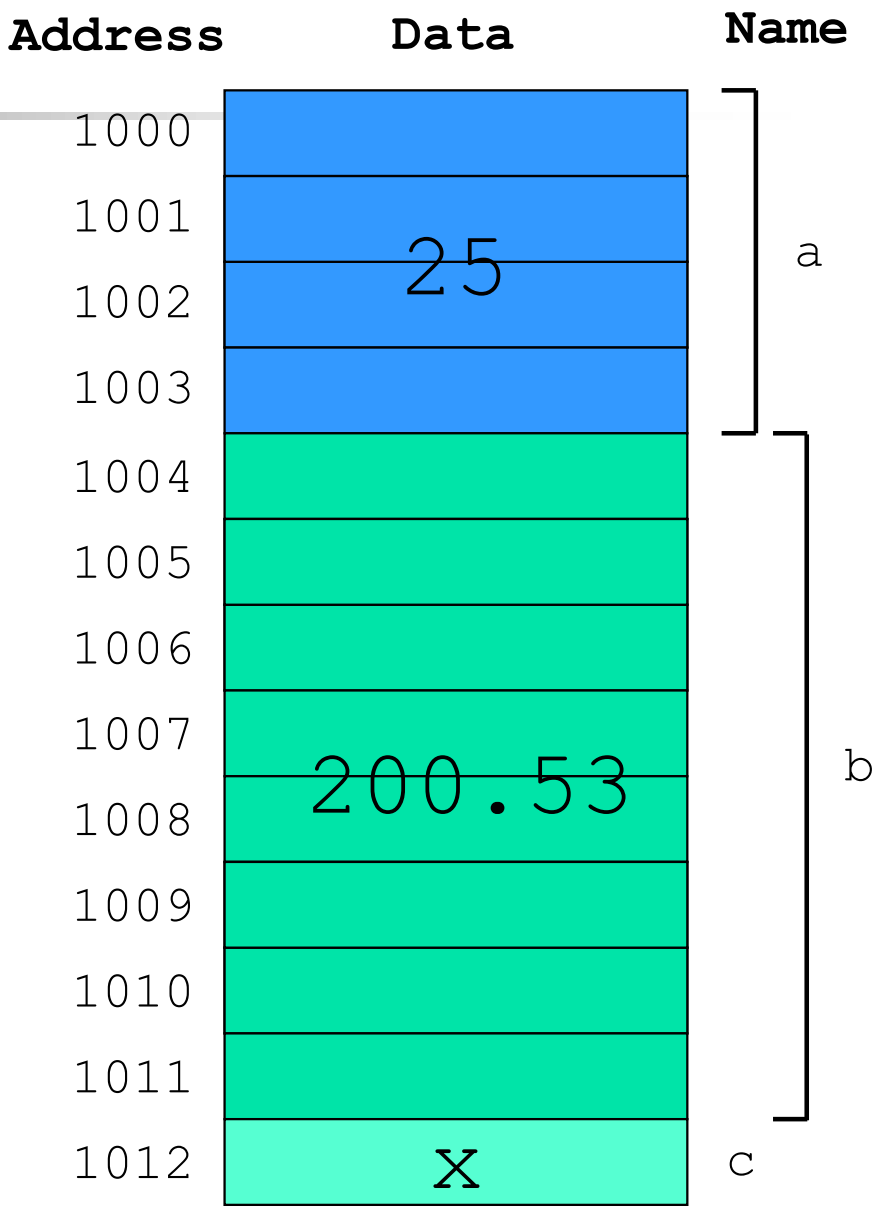
Terms ที่เกี่ยวข้องกับหน่วยความจำ



หน่วยความจำ

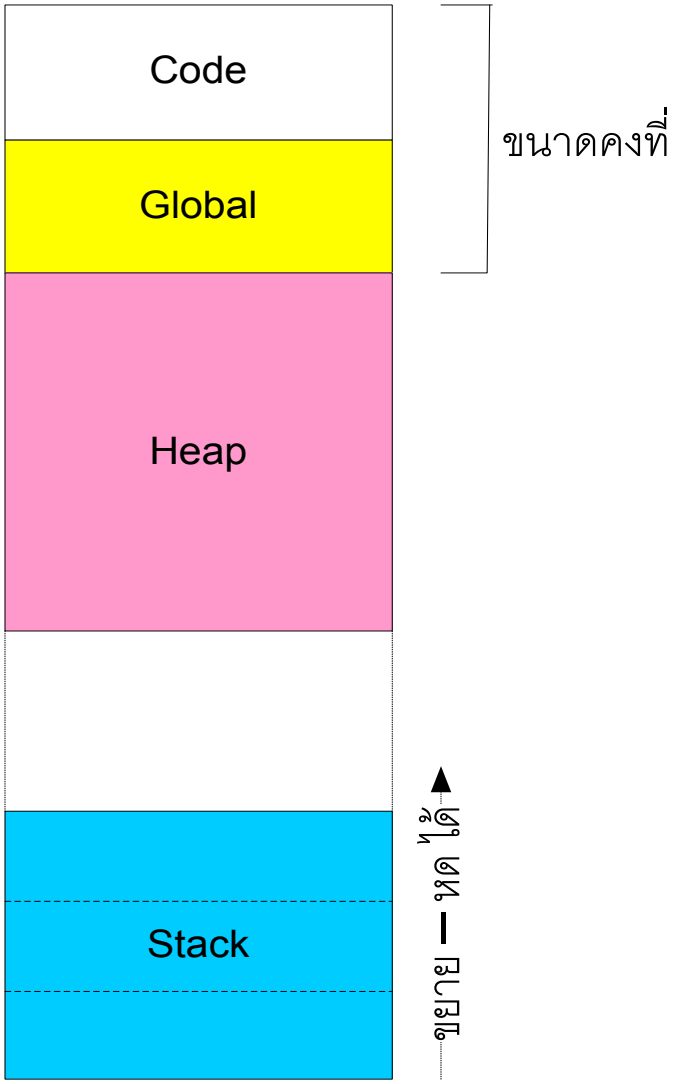
- **หน่วยความจำ**มีลักษณะเป็นชั้นเก็บข้อมูลหนึ่งมิติ มีช่องเก็บข้อมูล (Data) หลายช่องติดกัน
- แต่ละช่องมีขนาดการเก็บข้อมูลเท่ากัน คือเก็บข้อมูล 1 ไบต์ หรือ 8 บิต
- แต่ละช่องมีเลขที่ (Address) กำกับ
- โปรแกรมเมอร์ใช้ชื่อ (Name หรือ Identifier) ในการเรียกใช้ช่องเก็บข้อมูลเหล่านี้
- ชื่อหนึ่งชื่อ หมายถึงข้อมูลหนึ่งชิ้น อาจใช้ช่องเก็บข้อมูลติดกันหลายๆ ช่อง

```
int a = 25;
double b = 200.53;
char c = 'x';
```



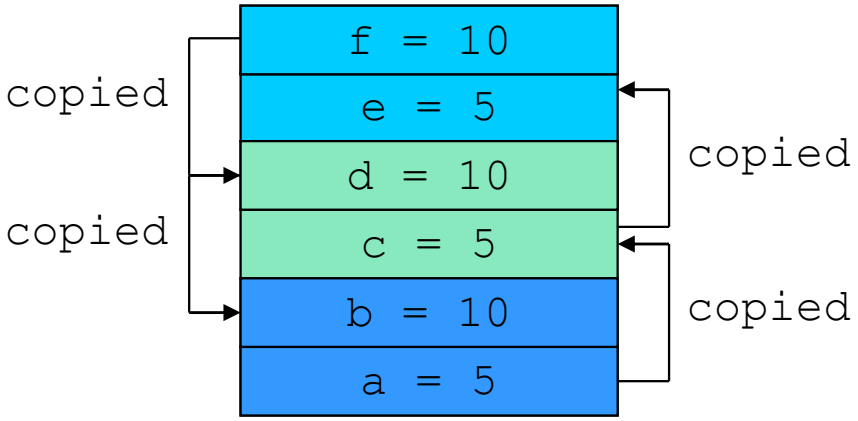
การใช้งานหน่วยความจำของโปรแกรม

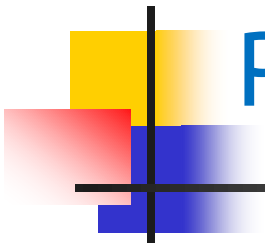
- ตอนที่โปรแกรมกำลังทำงาน หน่วยความจำที่โปรแกรมใช้แบ่งส่วนได้ดังนี้
  - **Code** ส่วนของคำสั่ง หรือ executable ที่ถูกคอมไพล์มาแล้ว
  - **Global** ส่วนของข้อมูลของตัวแปรที่เป็น static หรือ global
  - **Heap** ส่วนของข้อมูลที่โปรแกรมขอพื้นที่ขณะกำลังทำงาน (dynamic allocation)
  - **Stack** ส่วนของข้อมูลของตัวแปร local



หน่วยความจำ Stack

```
int func2(int e) {
    int f;
    f = e*2;
    return f;
    }

int func1(int c) {
    int d;
    d = func2(c)
    return d;
    }

    int main() {
    int a;
    int b;
    a = 5;
    b = func1(a);
    }
```





Pointers

- A *pointer* หมายถึงตัวแปรที่เก็บค่า address หรือตำแหน่งในหน่วยความจำ
- การประกาศต้องกำหนดชนิดของข้อมูลที่จะเก็บในตำแหน่งที่ pointer เก็บอยู่ เช่น

```
int* x;      // ประกาศว่า x เป็น pointer ที่สามารถชี้ไปที่ข้อมูลไทป์ int เท่านั้น
```

การเขียน **int* x** มีความหมายเหมือน **int *x** ทุกประการ

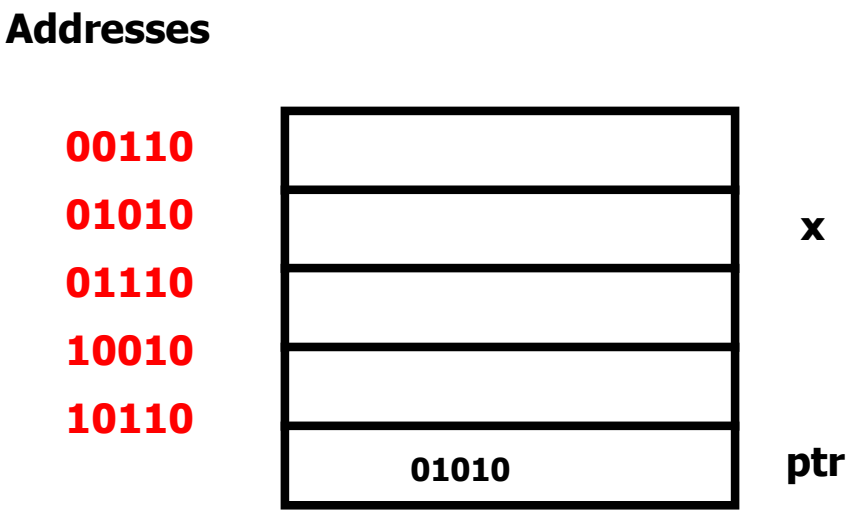
- Pointer สามารถประกาศให้ชี้ไปที่ข้อมูลได้ทุกชนิด

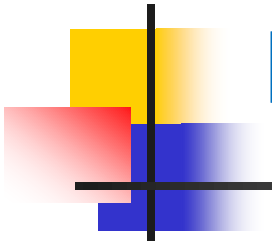
```
double* x;          // ชี้ไปที่ double
double** x;         // ชี้ไปที่ Pointer ที่ชี้ไปที่ double
MyDataType* x;      // ชี้ไปที่ MyDataType
```

Address-of operator

- เราสามารถใส่ค่า address ในตัวแปร pointer ด้วยการใช้ *address-of* operator (&)

```
int *ptr;

int x;

ptr = &x;
```



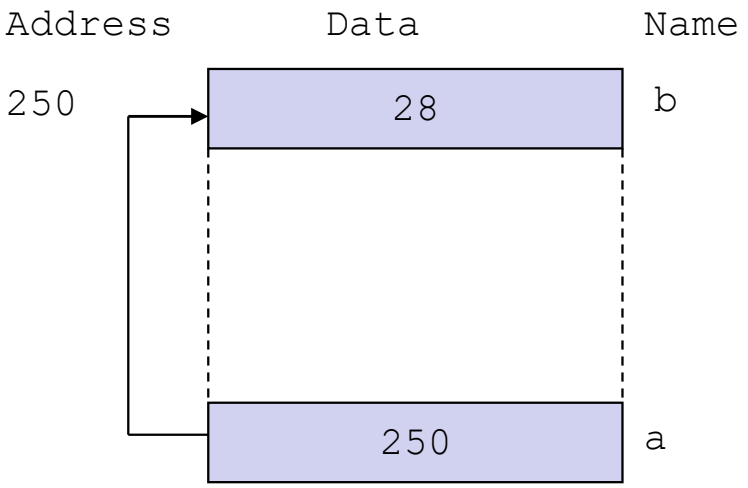


Dereference Operator

- The *dereference* operator (*)
- ผลลัพธ์ของการใช้ dereference คือเนื้อที่ที่ตำแหน่ง ที่ pointer เก็บอยู่
- ความหมายของ dereference จะขึ้นอยู่กับตำแหน่งที่มันปรากฏในโปรแกรม

ตัวอย่างการใช้ & และ *

```
int* a;       // a ประกาศว่าชี้ไปที่ int
int  b = 28;
int  c;

a = &b;       // a มีค่า 250 (address ของ b)
c = *a;       // c มีค่า 28 (ข้อมูลของสิ่งที่ a ชี้)
```

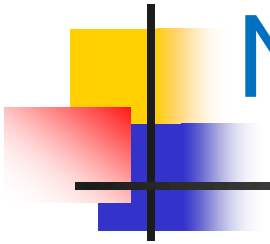


- operator & คืนค่า address ของตัวแปร

  &b คืนค่า address ของหน่วยความจำชื่อ b

- operator * ใช้กับตัวแปร pointer คืนข้อมูลของสิ่งที่ตัวแปรนั้นชี้

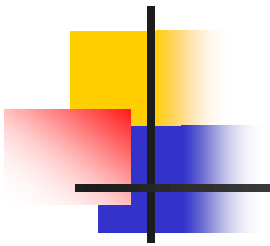
  a คืนข้อมูลที่เก็บในหน่วยความจำที่ a กำลังชี้



NULL

- pointer มีไว้เก็บ address แต่หากเราต้องการระบุว่า pointer ตัวนี้ไม่ได้ชี้ไปที่อะไรเลย เราจะใส่ค่า NULL (หมายถึงเลข 0)
  - Pointer ที่มีค่า NULL ไม่สามารถถูก Dereference ได้ โปรแกรมจะ Crash ทันทีถ้าหากถูก Dereference
  - เราใช้ NULL ในการระบุว่า Pointer ตัวนี้ยังไม่มีค่าที่สามารถใช้ได้หรือเป็นค่าเริ่มต้น

```
int* x = NULL;
```



Arrays

- address ของอะเรย์ก็คือ address ของข้อมูลตัวแรกในอะเรย์
- ชื่ออะเรย์ (array name) ที่ไม่มี index กำกับจะบอกตำแหน่งของอะเรย์
- address ของอะเรย์สามารถ assign ให้กับ pointer โดยใช้ชื่อของอะเรย์
- ชื่ออะเรย์ (array name) ไม่เป็น pointer เพราะ address ของอะเรย์ไม่สามารถเปลี่ยนแปลงได้ (address ที่เก็บใน pointer สามารถเปลี่ยนได้).

The [ ] Operator

- [ ] ใช้ระบุช่องของข้อมูลในอะเรย์ เช่น

  `nums[3]` ให้ค่าเช่นเดียวกับ `*(nums+3)`

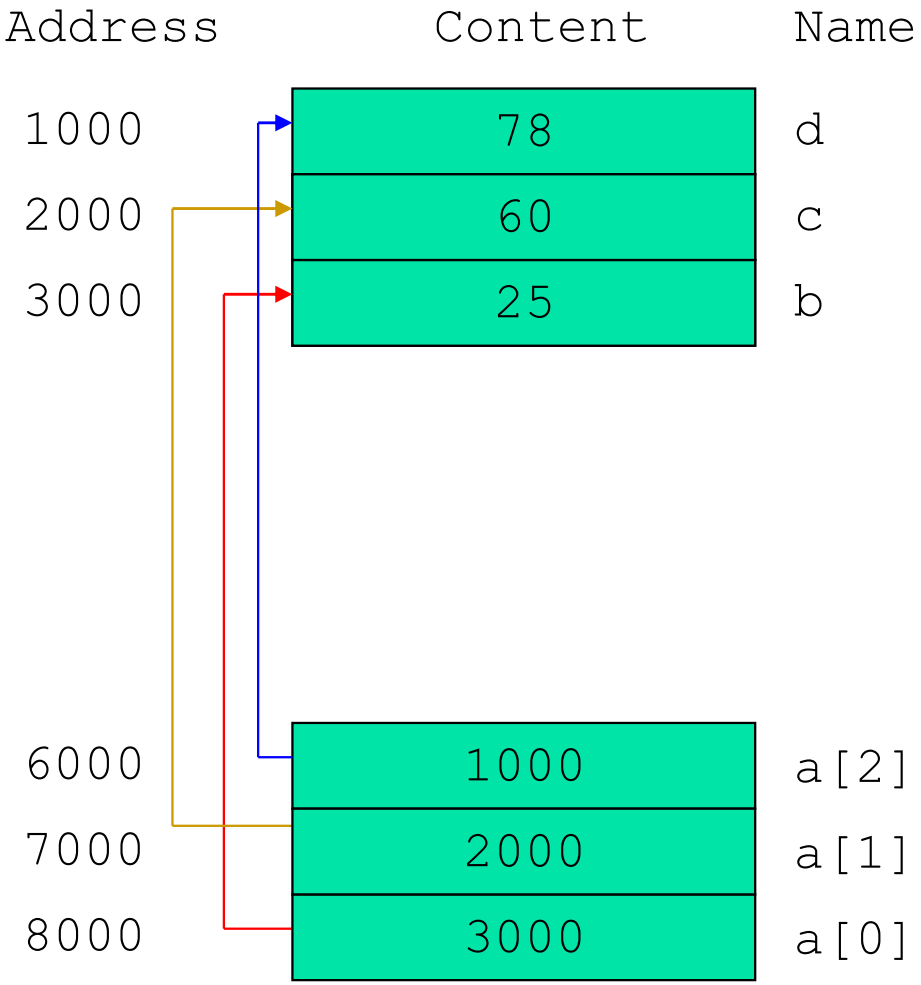
- การคำนวณหา address จาก nums + 3 ทำได้โดย
  - หาชนิดของข้อมูลที่ nums เก็บอยู่
  - นำขนาด (bytes) ของชนิดข้อมูลนั้นคูณด้วย 3
  - นำผลคูณที่ได้บวกกับ address ที่เก็บอยู่ใน nums
- เมื่อได้ address ที่เป็นผลลัพธ์ ก็จะเข้าไปหาข้อมูลที่ address นี้ได้

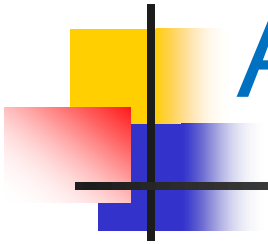
Array ของ pointers

```
int* a[3];
int  b=25, c=60, d=78;

a[0]=&b;
a[1]=&c;
a[2]=&d;
```

// a เป็น array มีขนาด 3 ช่อง

// แต่ละช่องเก็บข้อมูลชนิด pointer





Assignment

- เครื่องหมาย = หรือ assignment หมายถึงการนำค่าที่อยู่ฝั่งขวา ไปใส่ในหน่วยความจำที่ระบุโดยค่าที่อยู่ฝั่งซ้าย

```
x = y;      // copy ข้อมูลที่เก็บในหน่วยความจำชื่อ y ไปใส่ในหน่วยความจำชื่อ x
x = 2;      // นำเลข 2 ไปใส่ในหน่วยความจำชื่อ x
x = 2+y;    // คำนวณฝั่งขวาออกมาเป็นค่า (ตัวเลข) แล้วนำผลลัพธ์ไปใส่ในหน่วยความจำชื่อ x
x = *p;     // copy ค่าที่ได้จากการ Dereference p ไปใส่ในหน่วยความจำชื่อ x
*p = x;     // copy ข้อมูลที่เก็บในหน่วยความจำชื่อ x ไปใส่ในหน่วยความจำที่ชี้โดย p
p = q;      // copy ข้อมูลที่เก็บในหน่วยความจำชื่อ q ไปใส่ในหน่วยความจำชื่อ p
            // หาก p และ q เป็น pointer ทั้งคู่ นั่นก็หมายความว่าทั้ง p และ q เก็บ
            // address ค่าเดียวกัน แปลว่า p และ q ชี้ไปที่หน่วยความจำก้อนเดียวกัน
```

การคำนวณ Address

การคำนวณ Address จะยึดขนาดของไทป์ที่ Pointer ชี้เป็นหลัก เช่น

```
int* x; x++;      //แปลว่า x+4
double* x; x++; // แปลว่า x+8
```

```
int  a[10];
int  b;
int* c;

a[3] = 5;

c = &a[0];          // c ชี้ไปที่ a[0] มีค่าเท่ากับ c = a

c += 2;             // c ชี้ไปที่ a[2]

c++;                // c ชี้ไปที่ a[3]

b = *c;             // ข้อมูลที่ a[3] ถูก copy ไปใส่ที่ b
                    // b มีค่าเท่ากับ 5

*c = 10;            // ข้อมูลที่ a[3] มีค่าเท่ากับ 10
(*c)++;             // ข้อมูลที่ a[3] มีค่าเท่ากับ 11
```

การใช้ pointer กับฟังก์ชัน

```
void increment(int p)
{
    p = p + 1;
}
main()
{
    int a = 1;
    increment(a);

    // main ส่ง copy ของค่า a
    // หลังเรียกฟังก์ชัน a ไม่เปลี่ยนแปลง
}
```

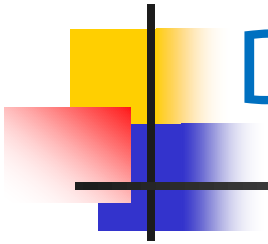
```
void increment(int *p)
{
    *p = *p + 1;
}
main()
{
    int a = 1;
    increment(&a);

    // main ส่ง address ของ a
    // ฟังก์ชันเปลี่ยนค่า a กลายเป็น 2
}
```

Exercise

What is the output from this code?

```
void F (int a, int *b)
{
        a = 7 ;
        *b = a ;
        b = &a ;
        *b = 4 ;
        printf("%d, %d\n", a, *b) ;
}

int main()
{
        int m = 3, n  = 5;
        F(m, &n) ;
        printf("%d, %d\n", m, n) ;
        return 0;
}
```

Dynamic memory allocation



- โดยปกติ โปรแกรมเมอร์ไม่มีทางรู้ล่วงหน้าว่าโปรแกรมจะต้องการใช้หน่วยความจำเท่าใด เช่น โปรแกรม Photoshop ไม่มีทางรู้ล่วงหน้าว่าผู้ใช้จะเปิดไฟล์รูปใหญ่เท่าใด
- ในภาษา C รุ่นที่นิยมใช้ การจองพื้นที่ array ไม่สามารถเปลี่ยนขนาดได้ตอนที่โปรแกรมกำลังทำงาน

```
int x[100];     // ขนาด 100 เปลี่ยนแปลงไม่ได้
```

- หากต้องการมากกว่า 100 ภายหลัง ไม่สามารถทำได้
- หากใช้ไม่ถึง 100 ก็จะเป็นการ "กั๊ก" หน่วยความจำไว้โดยผู้อื่นไม่สามารถใช้ได้
- การแก้ปัญหาทำได้โดยใช้ Dynamic Memory Allocation ซึ่งก็คือการ ขอใช้-ให้คืน หน่วยความจำในขณะที่โปรแกรมกำลังทำงาน

การขอเนื้อที่ในภาษา C ด้วย *malloc()*

- ในการขอเนื้อที่ให้กับอะเรย์ของเลขจำนวนเต็มขนาด n ตัว โดยค่า *n* จะรู้ต่อเมื่อมีการรันโปรแกรม

```
int *a;

a  = malloc(n *sizeof(int));
```

- ฟังก์ชัน *sizeof* ใช้ในการคำนวณขนาดเนื้อที่ที่ต้องการ (bytes)
- เมื่อตำแหน่งของเนื้อที่ที่ขอถูกเก็บไว้ที่ *a* แล้ว เราสามารถใช้ *a* ได้เหมือนกับการใช้ชื่อของอะเรย์ทั่วไป

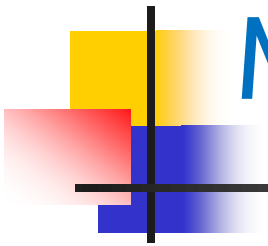
```
for (i = 0; i < n; i++)
   a[i] = 0;
```

ตัวอย่าง : การขอเนื้อที่สำหรับ Dynamic Arrays

```
1   void foo( )
2   {
3       int numElements;
4       printf("How many elements would you like the array to have? ");
6       scanf("%d",&numElements);
7       float *ptrArr = malloc(numElements * (sizeof(float));
8
9       /* the array is processed here
10      output to the user is provided here */
11  }
```

จะเกิดอะไรขึ้นเมื่อฟังก์ชันนี้ทำงานจบ ?

Memory Leak

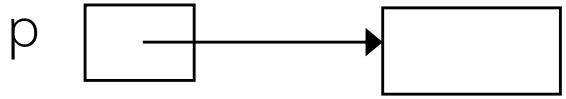


- เมื่อฟังก์ชันทำงานจบ เนื้อที่ของตัวแปรทุกตัวในฟังก์ชัน และค่าที่ตัวแปรเก็บอยู่จะถูกคืนไป (numElements and ptrArr)
- address ของ dynamic array ก็หายไปด้วย ทั้งๆ ที่ dynamic array ยังไม่ถูกคืนเนื้อที่ และเนื้อที่ส่วนนี้จะไม่สามารถนำกลับมาใช้ได้อีก
- ปัญหานี้เรียกว่า ***memory leak*** (จะเกิดขึ้นขณะที่โปรแกรมทำงาน และหายเมื่อโปรแกรมทำงานจบ)
- ถ้าโปรแกรมปล่อยให้เกิด memory leaks อาจทำให้พื้นที่ในส่วนของ heap memory ถูกใช้งานจนหมดและโปรแกรมไม่สามารถทำงานต่อได้
- ปัญหา Memory leak สามารถป้องกันได้

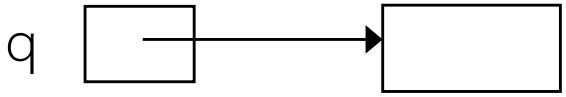
การคืนเนื้อที่ใน heap

- เพื่อป้องกัน memory leak เราควรคืนเนื้อที่ให้กับระบบเมื่อไม่ใช้งานเนื้อที่นั้นแล้ว

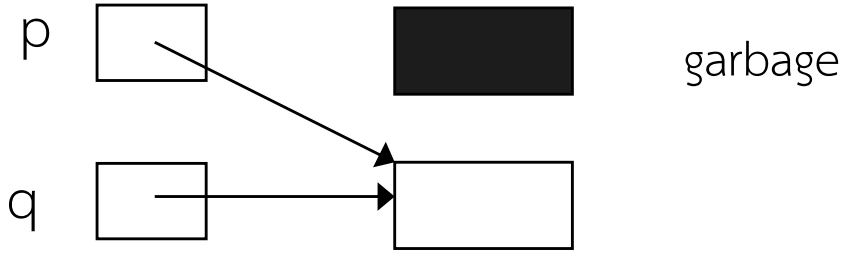
```
p = malloc (...);
```




```
q = malloc(...);
```




```
p = q;
```

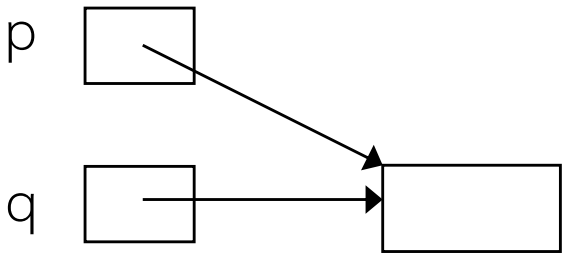


การใช้คืนเนื้อที่ในภาษา C ด้วย *free()*

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

- ฟังก์ชัน *free* จะคืนเนื้อที่หน่วยความจำตำแหน่งที่ตัวแปร p ชี้อยู่
- เนื้อที่ที่คืนไปนั้น ระบบจะสามารถนำกลับมาใช้ได้อีกสำหรับการเรียกฟังก์ชัน malloc ในครั้งต่อๆ ไป

```
p = malloc (...);
q = malloc(...);
free(p)
p = q;
```



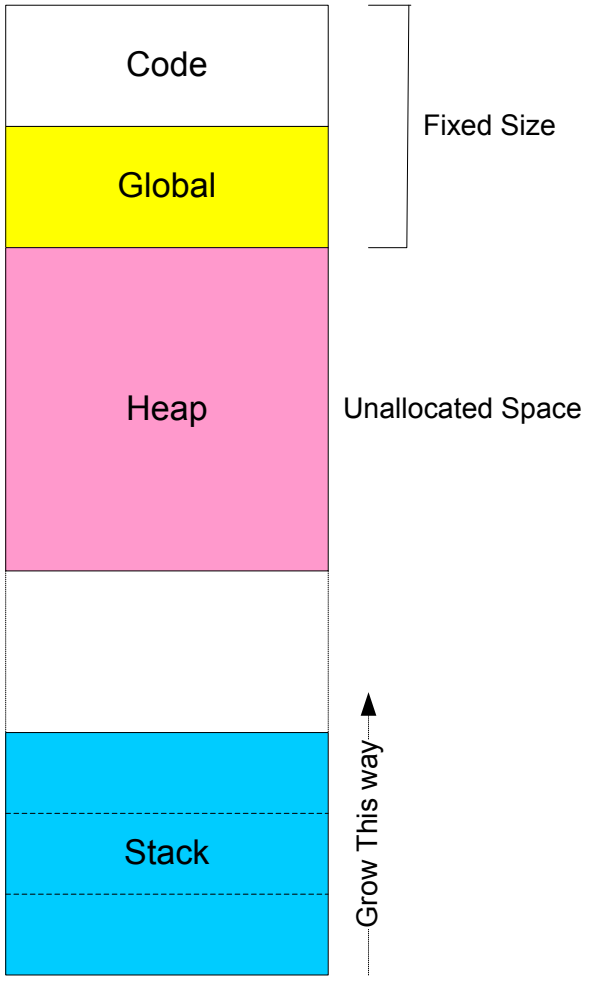
ตัวอย่าง : การคืนเนื้อที่สำหรับ Dynamic Arrays

```
1    void foo( )
2    {
3          int numElements;
4          printf(“How many elements would you like the array to have? “);
6          scanf(“%d”,numElements);
7          float *ptrArr = malloc(numElements * (sizeof(float));
8
9          /* the array is processed here
10         output to the user is provided here */
11         free (ptrArr);
12   }
```

เมื่อคืนเนื้อที่แล้ว เนื้อที่นี้สามารถใช้ได้อีกเมื่อมีการขอด้วย malloc

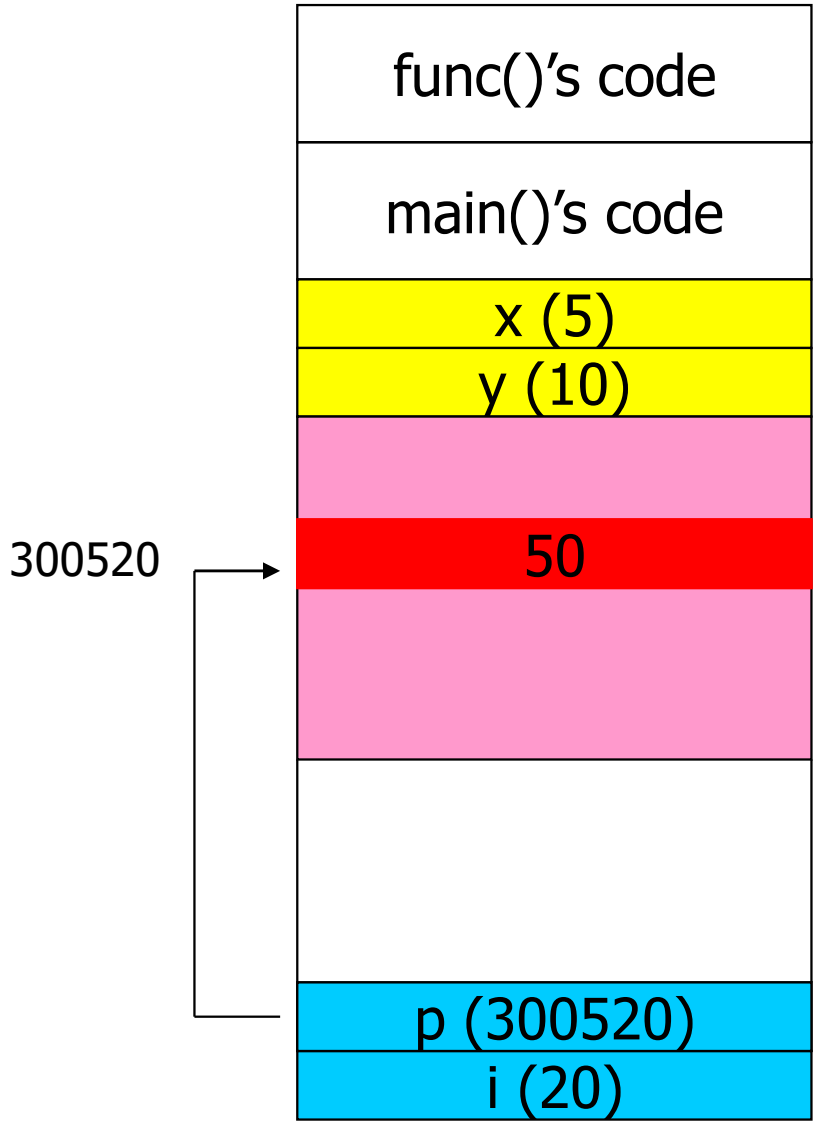
การขอเนื้อที่และคืนเนื้อที่ในภาษา C++

- ภาษา C++ ใช้ new ในการขอยืมใช้หน่วยความจำ
- ใช้ delete ในการคืนเมื่อใช้เสร็จ

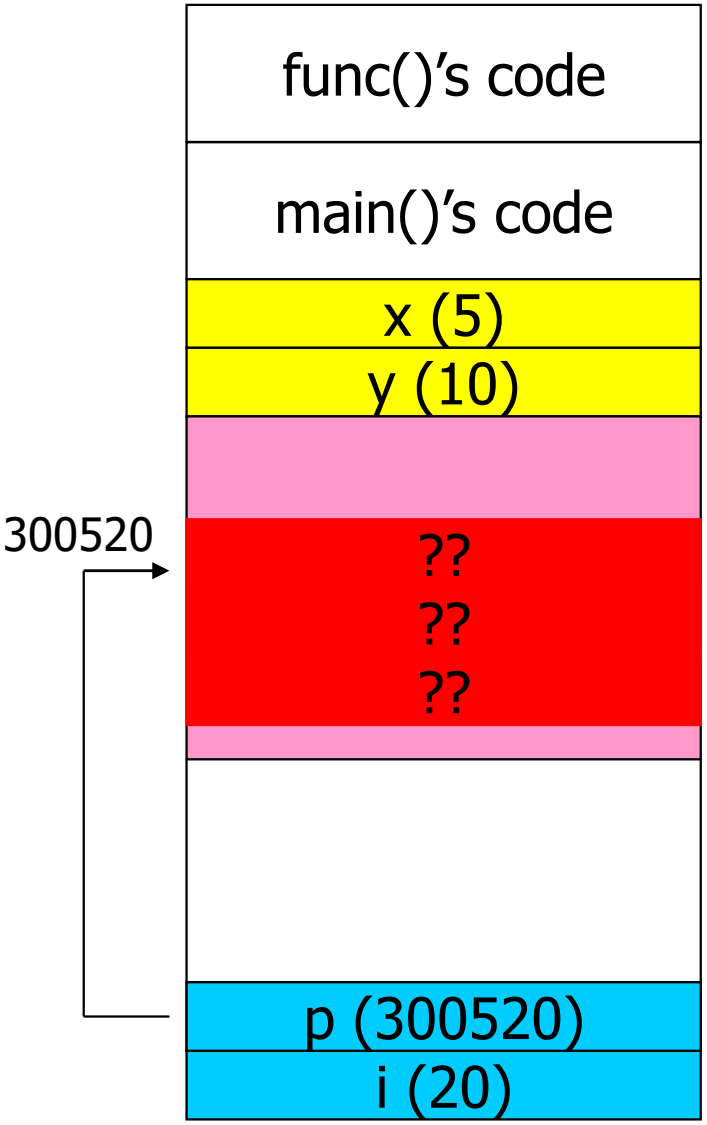


ตัวอย่างการใช้ new และ delete

```
int x = 5;
int y = 10;

void func() {…}

int main()
{
   int i = 20;
   int* p;

   p = new int;
   *p = 50;

   delete(p);

   return 0;
}
```



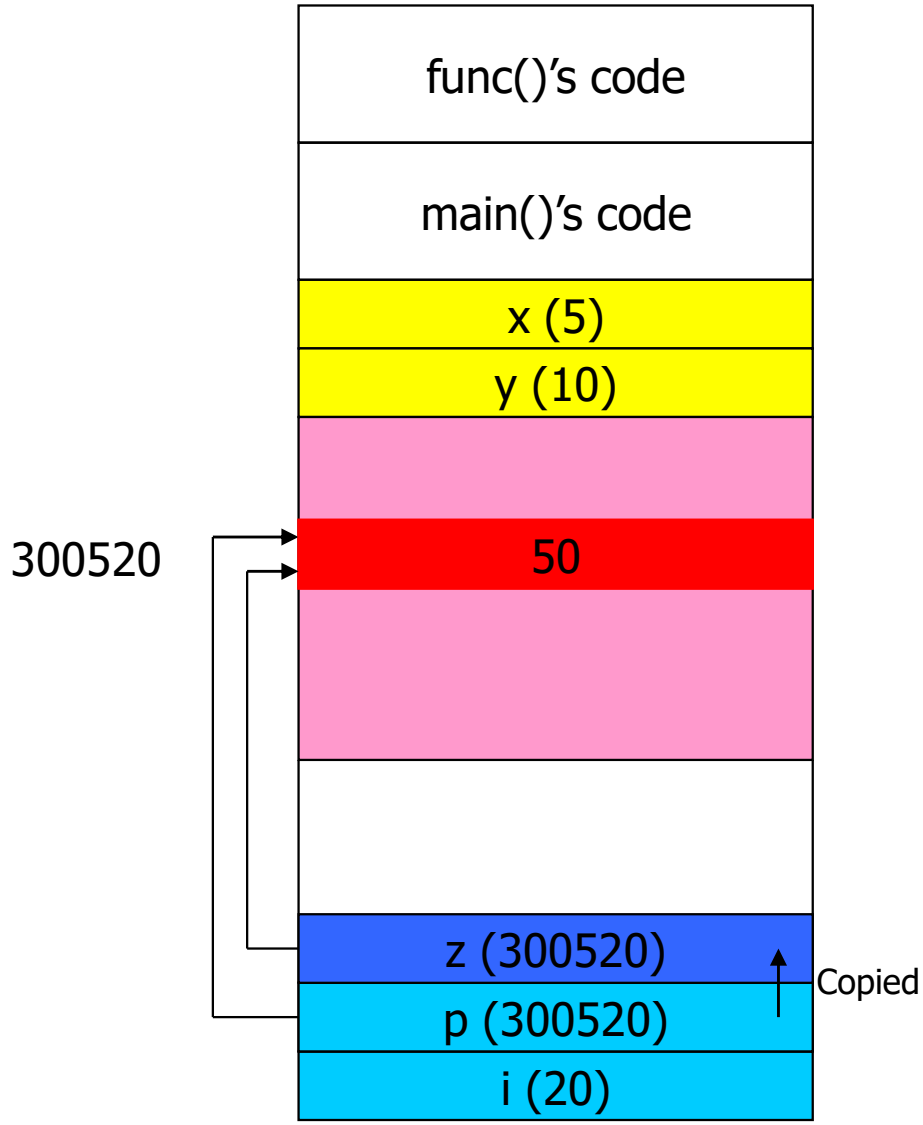
```
int x = 5;
int y = 10;

void func() {…}

int main()
{
    int i = 20;
    int* p = new int[3];

    p[1] = 50;

    delete([]p);

    return 0;
}
```



ตัวอย่างปัญหา Dangling pointer

```
int x = 5;
int y = 10;

void func(int* z)
{
   delete(z);
}

int main()
{
   int i = 20;
   int* p;

   p = new int;
   *p = 50;

   func(p);

   i = *p;

   return 0;
}
```

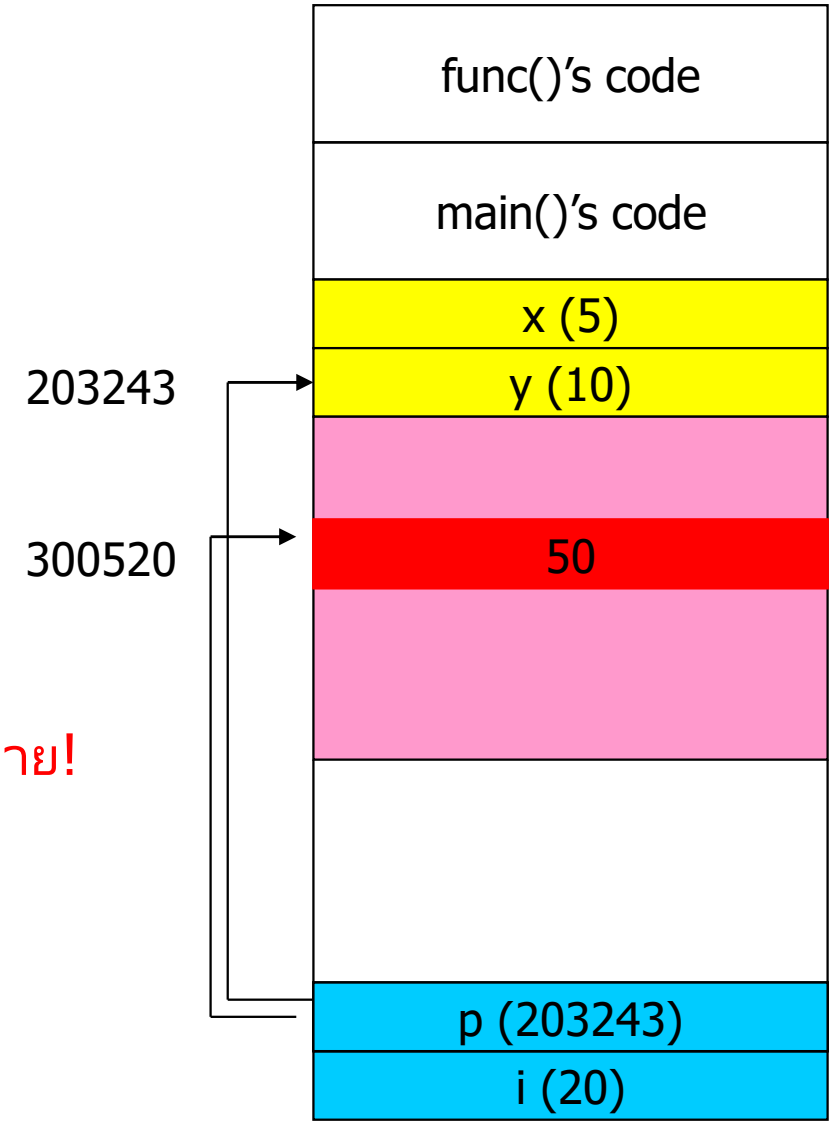
BAD!



ตัวอย่างปัญหา Memory leak

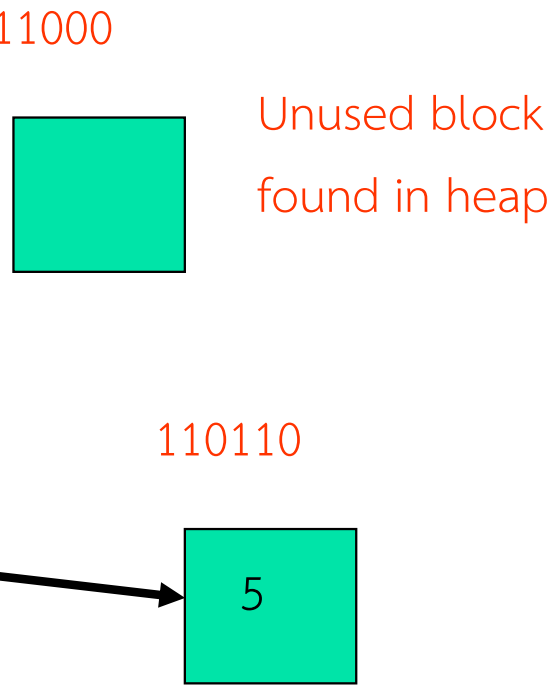
```
int x = 5;
int y = 10;

int main()
{
   int i = 20;
   int* p;

   p = new int;
   *p = 50;

   p = &y;

   return 0;
}
```

ทำ address หาย!

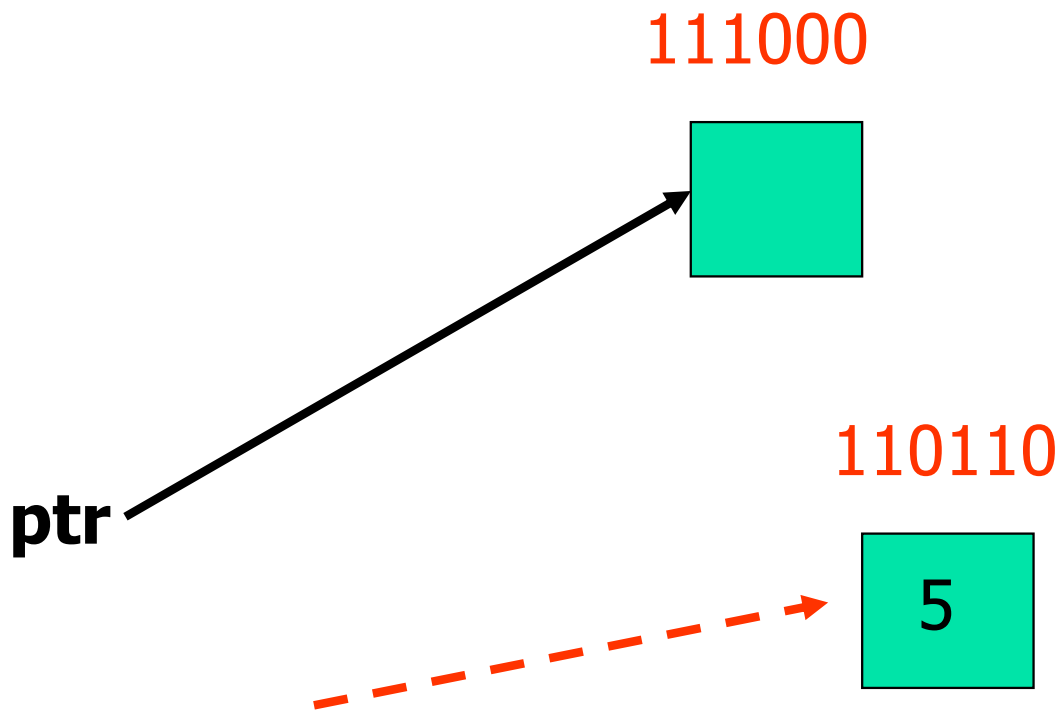


สาเหตุอื่นของ Memory Leak

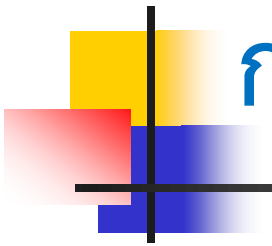
```
ptr = malloc(sizeof(int));
*ptr = 5;
```



ptr = 111000;

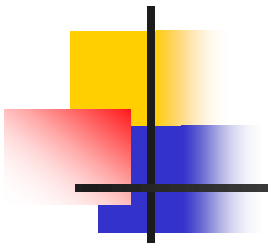


address ของเนื้อที่นี้หายไป และยังไม่ได้ถูกคืน (memory leak)



การหลีกเลี่ยงปัญหา Memory Leak

- เมื่อต้องการเปลี่ยน address ที่เก็บอยู่ใน pointer ให้คิดก่อนว่า address ที่เก็บอยู่ถูกใช้สำหรับ dynamically-allocated memory หรือไม่
- ถ้าใช่ (และเนื้อที่นั้นไม่ถูกใช้งานแล้ว) ใช้คำสั่ง free() คืนเนื้อที่ก่อนที่จะเปลี่ยนค่า address ใน pointer



ข้อมูลนามธรรม (Data Abstraction)

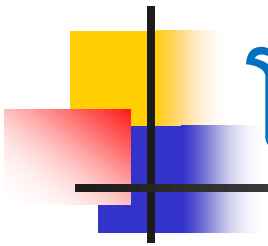
- ชนิดของข้อมูล (Data Type)
- โครงสร้างข้อมูล (Data Structure)
- ชนิดข้อมูลนามธรรม (Abstract Data Type)

ชนิดของข้อมูล (Data Type)

ชนิดของข้อมูล คือ ช่วงของค่าที่ตัวแปรสามารถใช้ได้

- นอกจากชนิดของข้อมูลจะบอกถึงช่วงของข้อมูลแล้วยังบอกถึงสิ่งที่สามารถกระทำได้กับข้อมูลนั้น ๆ (operations)
- ชนิดของข้อมูลโดยทั่วไปแล้วจะแตกต่างกันในแต่ละภาษาโปรแกรม
- ตัวอย่างของชนิดของข้อมูลได้แก่

integer	ชนิดของข้อมูลที่มีค่าเป็นตัวเลขจำนวนเต็ม มักจะมีอยู่ในทุกภาษาโปรแกรม
boolean	ชนิดของข้อมูลที่เป็นจริงหรือเท็จมีในบางภาษาเท่านั้น
string	ชนิดของข้อมูลที่เก็บข้อความหรือตัวอักษรตั้งแต่ 1ตัวขึ้นไป



โครงสร้างข้อมูล (Data Structures)

โครงสร้างข้อมูลเป็นที่เก็บข้อมูลหลาย ๆ ตัวไว้ด้วยกัน โดยที่ข้อมูลเหล่านั้นอาจจะเป็นข้อมูลที่มีชนิดเดียวกันหรือต่างชนิดกันก็ได้ มักมีอยู่ในทุกภาษาโปรแกรม ตัวอย่างเช่น

array	เป็นโครงสร้างข้อมูลที่เก็บข้อมูลชนิดเดียวกันไว้ด้วยกัน
record	เป็นโครงสร้างข้อมูลที่เก็บข้อมูลต่างชนิดกันหรือชนิดเดียวกันไว้ด้วยกัน

ชนิดข้อมูลนามธรรม (Abstract Data Type)

ชนิดข้อมูลนามธรรม เป็นชนิดของข้อมูลหรือโครงสร้างข้อมูลที่ไม่มีอยู่ในภาษาโปรแกรม ผู้เขียนโปรแกรมสามารถสร้างขึ้นมาเพื่อแก้ปัญหาโดยยังไม่ต้องคำนึงว่าจะเขียนเป็นโปรแกรมอย่างไร เมื่อมีการสร้างหรือกำหนดขึ้นมาผู้สร้างจำเป็นต้องกำหนดสิ่งที่สามารถจะกระทำได้ (operations) กับข้อมูลที่สร้างใหม่นี้ด้วย

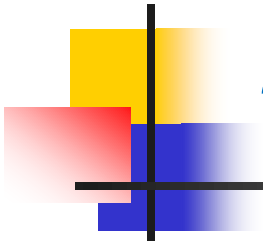
Example : Set

สมมติให้ A, B และ C มีชนิดเป็นเซตของตัวเลขจำนวนเต็ม

MAKENULL(A) ทำให้เซต A กลายเป็นเซตว่าง

UNION(A,B,C) ทำการ union เซต A กับ เซต B ให้ผลลัพธ์อยู่ในเซต C

SIZE (A) ส่งกลับขนาดหรือจำนวนสมาชิกของเซต A



การสร้างข้อมูลนามธรรม

- เราไม่สามารถที่จะออกแบบภาษาโปรแกรมที่ประกอบไปด้วย ชนิดของข้อมูลที่สามารถแก้ไขปัญหาได้ทุกปัญหาในโลก
- ในแต่ละภาษาโปรแกรมมีเพียงชนิดของข้อมูลหรือโครงสร้างของข้อมูลหลักๆ ซึ่งบางครั้งไม่เพียงพอในการแก้ปัญหาที่ซับซ้อน
- ผู้พัฒนาโปรแกรมต้องสร้างชนิดของข้อมูลพิเศษขึ้นมาใช้เอง ชนิดของข้อมูลที่สร้างขึ้นมานี้ก็คือ ข้อมูลนามธรรม (Abstract Data Type) นั่นเอง
- เมื่อสร้างข้อมูลนามธรรมขึ้นมาใหม่ผู้สร้างก็ต้องกำหนด operations ที่สามารถทำงานได้กับข้อมูลชนิดนี้ขึ้นมาด้วย เช่นเดียวกันกับการสร้างชนิดข้อมูล SET ตามตัวอย่าง

ความสัมพันธ์ของ ADTและโครงสร้างข้อมูล