

# CS 216 Data Structures & Algorithms

---

## Pointer, Dynamic Arrays, Vector, C++

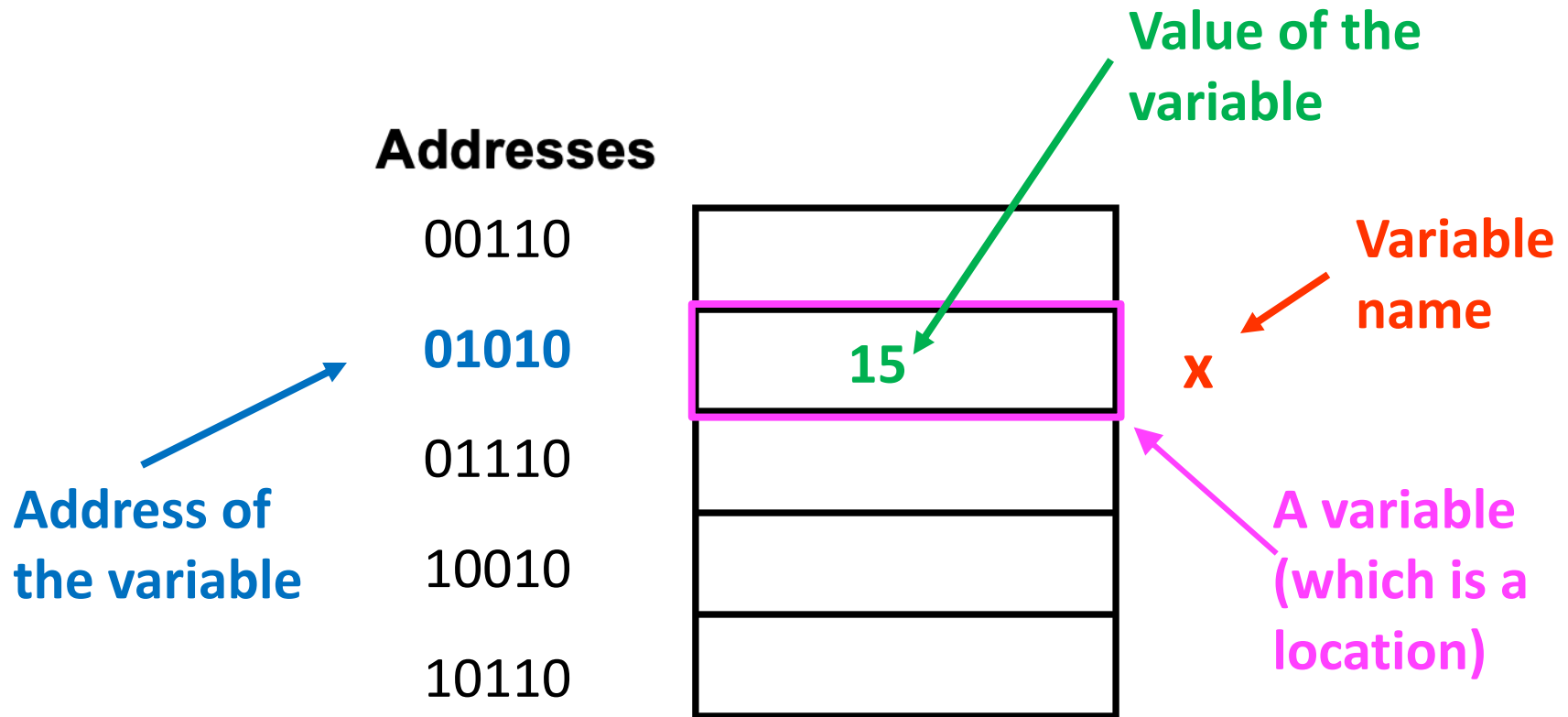
Wirat Jareevongpiboon

- Ratchata Peachavanish (ทบทวนภาษา C/C++)
- Yusuf Sahillioğlu, Department of Computer Engineering, Middle East Technical University
- Douglas Wilhelm Harder, M.Math.LEL, Department of Electrical and Computer Engineering, University of Waterloo

# Memory Terminology

- variable name
- variable
- value
- address – a binary number used by the operating system to identify a memory cell of RAM
- It is important to know the precise meanings of these terms

# Memory Terminology



# หน่วยความจำ

- หน่วยความจำมีลักษณะเป็นชั้นเก็บข้อมูลหนึ่งมิติ 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;
```

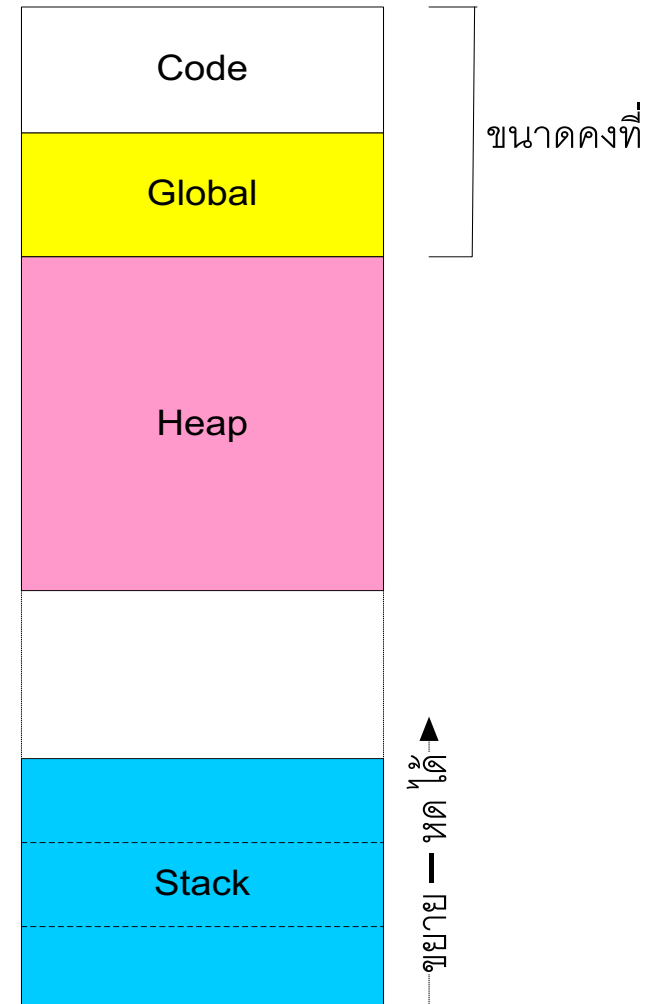
```
double b = 200.53;
```

```
char c = 'x' ;
```

Address	Data	Name
1000	25	a
1001		
1002		
1003		
1004	200.53	b
1005		
1006		
1007		
1008		
1009		
1010		
1011		
1012	X	c

# การใช้งานหน่วยความจำของโปรแกรม

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

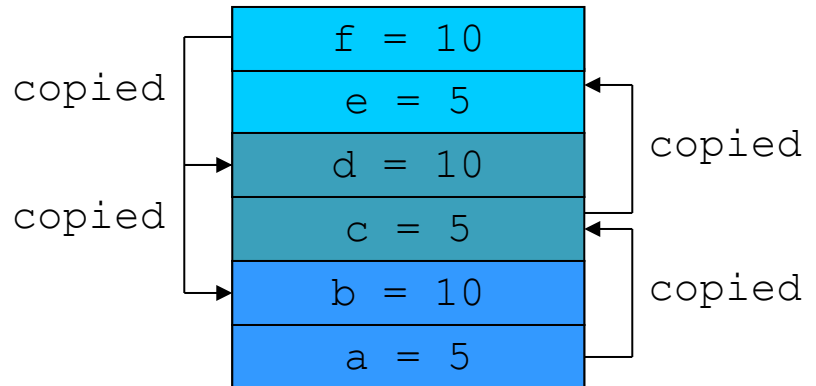


# หน่วยความจำ Stack

```
→ 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

`int* x;`

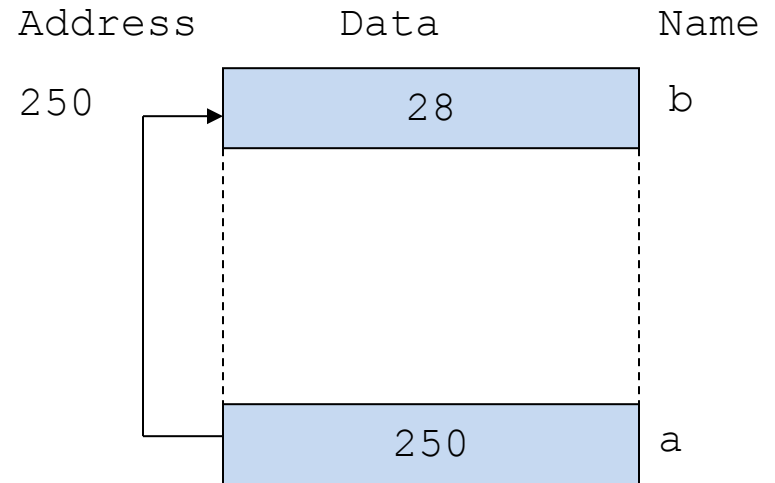
- ประกาศว่า **x** เป็นไพบี **pointer** ที่สามารถชี้ไปที่ข้อมูลไพบี **int** เท่านั้น
  - การเขียน `int* x` มีความหมายเหมือน `int *x` ทุกประการ
  - **x** ใช้เนื้อที่ **32** บิตหรือ **4** ไบท์
  - **x** เก็บค่า **address**
- **Pointer** สามารถประกาศให้ชี้ไปที่ข้อมูลได้ทุกชนิด

<code>double* x;</code>	<code>// ชี้ไปที่ double</code>
<code>double** x;</code>	<code>// ชี้ไปที่ Pointer ที่ชี้ไปที่ double</code>
<code>MyDataType* x;</code>	<code>// ชี้ไปที่ MyDataType</code>

# Pointers

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

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



- **operator &** คำนวณค่า address ของตัวแปร  
    **&b** คำนวณค่า address ของหน่วยความจำชื่อ b
- **operator \*** ใช้กับตัวแปร pointer คำนวณข้อมูลของสิ่งที่ตัวแปรนั้นชี้  
    **\*a** คำนวณข้อมูลที่เก็บในหน่วยความจำที่ a กำลังชี้  
    กระบวนการนี้เรียกว่า Dereferencing



# NULL

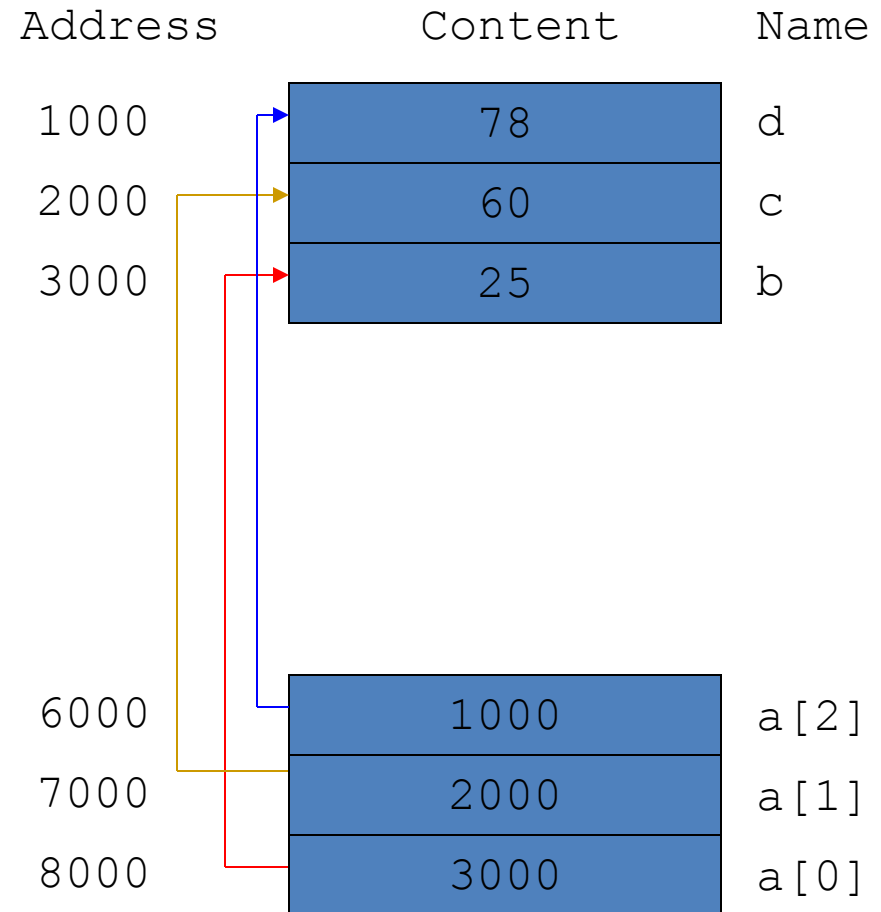
- **pointer** มีไว้เก็บ **address** แต่หากเราต้องการระบุว่า **pointer** ตัวนี้ไม่ได้ชี้ไปที่อะไรเลย เราจะใส่ค่า **NULL** (หมายถึงเลข 0)
  - **Pointer** ที่มีค่า **NULL** ไม่สามารถถูก **Dereference** ได้ โปรแกรมจะ **Crash** ทันทีถ้าหากถูก **Dereference**
  - เราใช้ **NULL** ในการระบุว่า **Pointer** ตัวนี้ยังไม่มีค่าที่สามารถใช้ได้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

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

# Array ของ pointers

```
int* a[3];  
int b=25, c=60, d=78;  
  
a[0]=&b;  
a[1]=&c;  
a[2]=&d;
```

// a เป็นไทป์ array มีขนาด 3 ช่อง  
// แต่ละช่องเก็บข้อมูลไทป์ 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

```
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
```

การคำนวณ Address จะยึดขนาดของไทป์ที่ Pointer ชี้เป็นหลัก เช่น

**int\* x; x++;** แปลว่า **x+4**

**double\* x; x++;** แปลว่า **x+8**

# ส่งข้อมูลให้ฟังก์ชัน

```
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
}
```

# Dynamic memory allocation

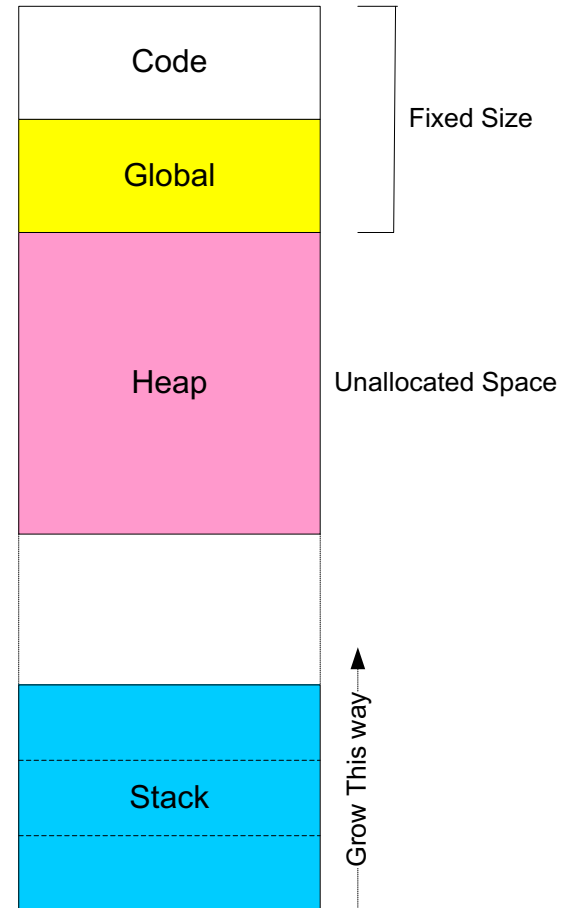
- โดยทั่วไป โปรแกรมเมอร์ไม่มีทางรู้ล่วงหน้าว่าโปรแกรมจะต้องการใช้หน่วยความจำเท่าใด เช่น โปรแกรม **Photoshop** ไม่มีทางรู้ล่วงหน้าว่าผู้ใช้จะเปิดไฟล์รูปใหญ่เท่าใด
- ในภาษา **C** รุ่นที่นิยมใช้ การจองพื้นที่ **array** ไม่สามารถเปลี่ยนขนาดได้ตอนที่โปรแกรมกำลังทำงาน

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

- หากต้องการมากกว่า **100** ภายหลัง ไม่สามารถทำได้
- หากใช้ไม่ถึง **100** ก็จะเป็นการ "กัก" หน่วยความจำไว้โดยผู้อื่นไม่สามารถใช้ได้
- การแก้ปัญหาก็ทำได้โดยใช้ **Dynamic Memory Allocation** ซึ่งก็คือการ ขอใช้-ให้คืน หน่วยความจำในขณะที่โปรแกรมกำลังทำงาน

# Heap

- เราขอใช้หน่วยความจำจาก **Operating System** จากส่วนที่เรียกว่า **Heap**
- ภาษา **C++** ใช้ **new** ในการขอใช้หน่วยความจำ
- ใช้ **delete** ในการคืนเมื่อใช้เสร็จ



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

```
void func() {...}
```

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



```
    p = new int;
```

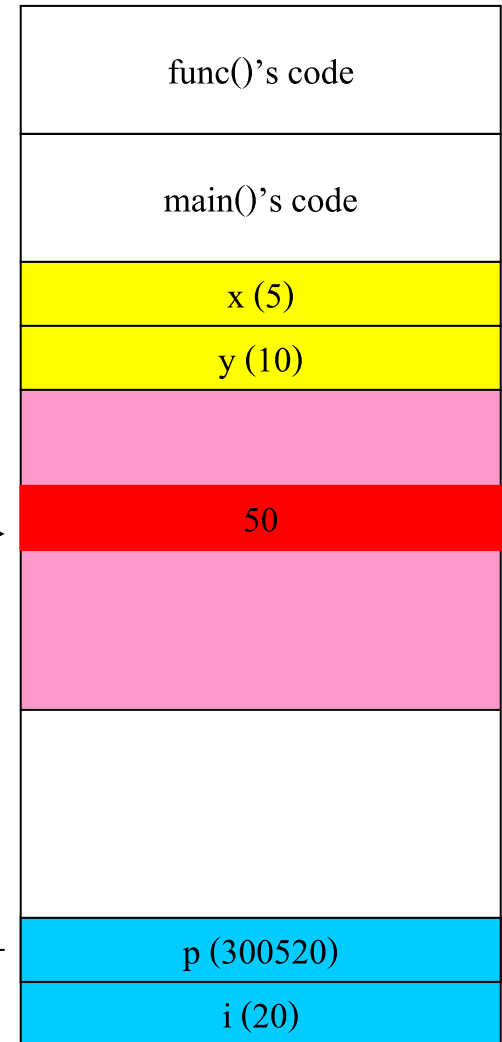
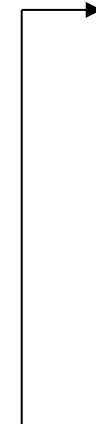
```
    *p = 50;
```

```
    delete(p);
```

```
    return 0;
```

```
}
```

300520





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

```
void func() {...}
```

```
int main()  
{
```

```
→ int i = 20;
```

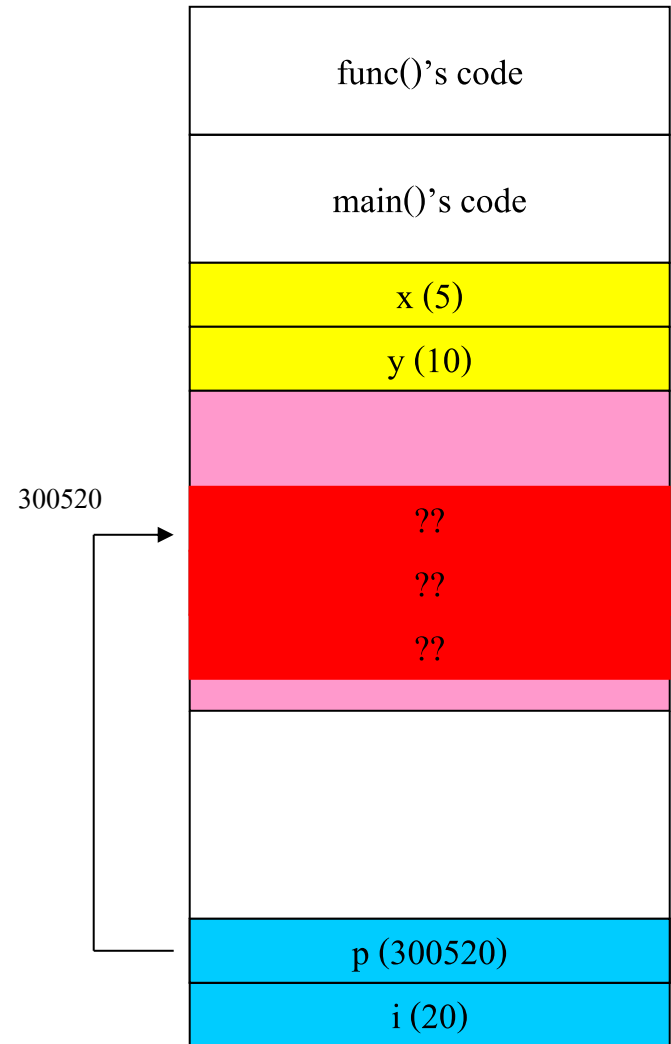
```
→ int* p = new int[3];
```

```
→ p[1] = 50;
```

```
→ delete(p);
```

```
    return 0;
```

```
}
```



# 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;
```

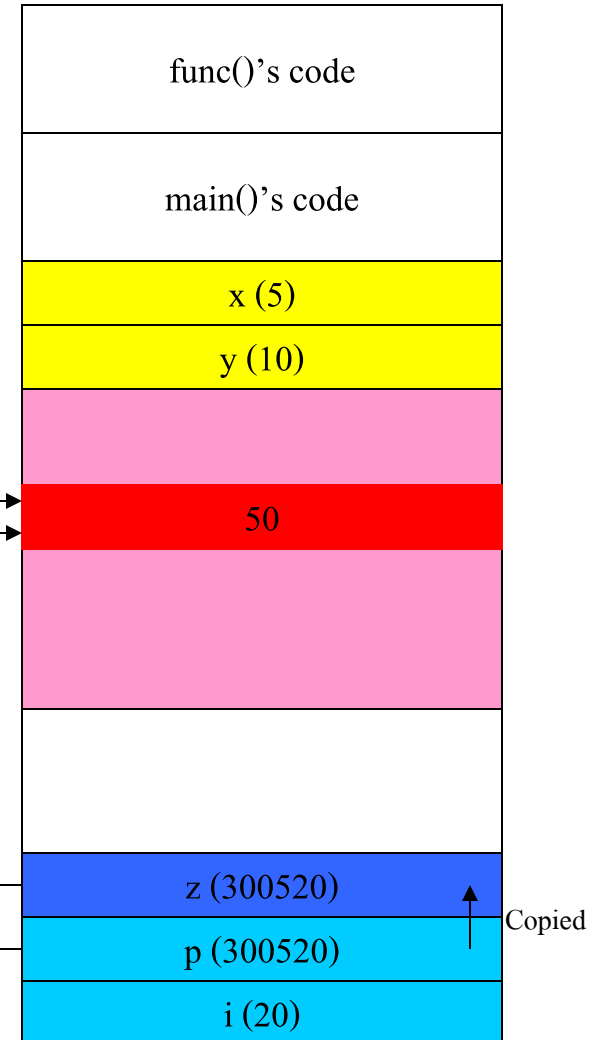
```
→ func(p);
```

```
→ i = *p;
```

**BAD!**

```
    return 0;  
}
```

300520



# Memory leak

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

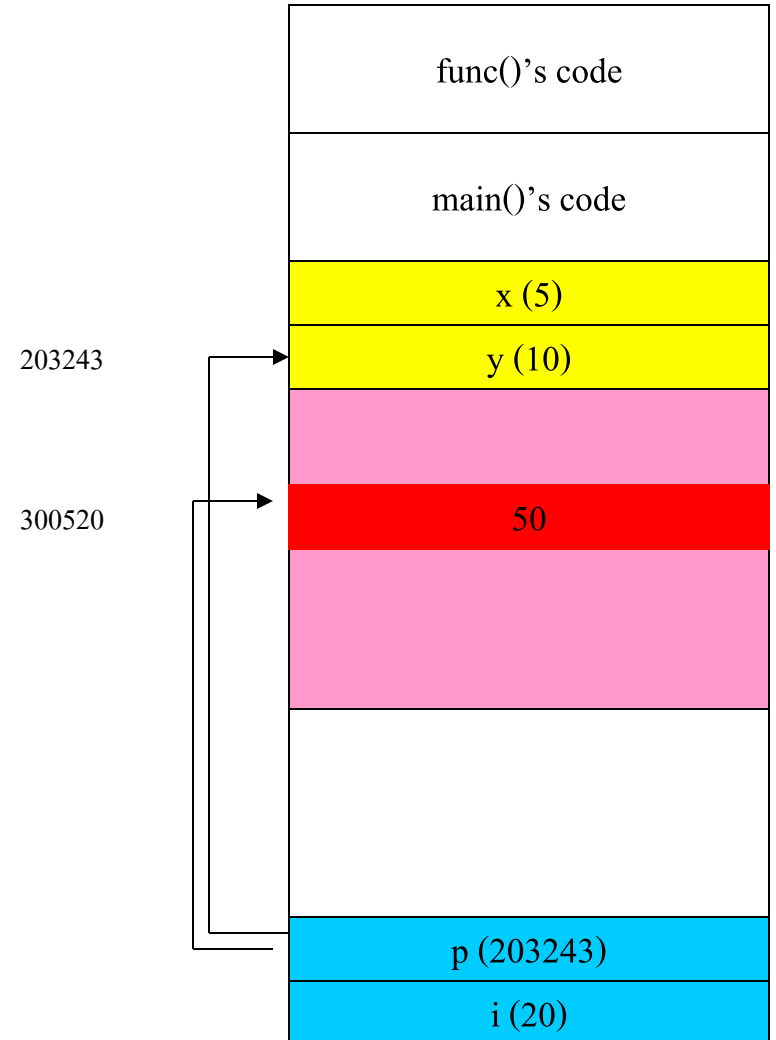
int main()
{
    int i = 20;
    int* p;

    → p = new int;
    → *p = 50;

    → p = &y;

    return 0;
}
```

ทำ address หายไป!



# References


- **References** are a type of C++ variable that act as an *alias* to another variable.
- A **reference variable** acts just like the original variable it is referencing.
- **References** are declared by using an **ampersand (&)** between the reference type and the variable name.

# Example

```
int n = 5, m = 6;
```

```
int &rn = n;
```

You cannot declare a reference without giving a value.



```
n = 6;
```


```
rn = 7,
```

```
cout << n << rn << m << endl; //776
```

```
rn = m ;
```

```
cout << n << rn << m << endl; //666
```

Makes n equal to m  
(doesn't make rn refer to m)



# const Reference

- A `const` reference will not let you change the value it references:
- Example:

```
int n = 5;  
const int &rn = n;
```

```
rn = 6; // error!!
```

- `const` reference is like a `const` pointer to a `const` object.

# References vs Pointers

- Everything that is accomplished by *references* can be accomplished by *pointers* but the syntax of *references* is simpler:
- Example:

```
int n= 5;  
int &rn = n;  
int *const p = &n;  
*p = 6;  
rn = 6;
```



Same effect

# Pointers and `const`

There are two different ways that pointers and `const` can be intermixed:

1. Constant pointer
2. Pointer to a constant variable



# Constant Pointer

- A `const` pointer must be initialized to a value upon declaration, and its value can not be changed.
- However, because the value being pointed to is still `non-const`, it is possible to change the value being pointed to via dereferencing the pointer:

```
int *const p = &i;  
*p = 6;           // it is O.K.  
p = &j;           // NOT O.K.
```

# Pointer to a `const` variable

- It is also possible to declare a pointer to a constant variable by using the `const` before the data type:

```
int i;  
const int * p = &i;  
*p = 6;  // it is NOT O.K., because i is  
          //treated as constant when accessed by p.
```

- However, it can be changed independently:

```
i = 6;  // It is O.K.
```

- It is also possible to declare a `const` pointer to a constant value:

```
const int n = 5;  
const int * const p = &n;
```

# Parameter Passing

In C, all parameters are passed by value (call by value). But C++ offers three options:

- **Call by value**
  - Copy of data passed to function
  - Changes to copy do not change original
- **Call by reference**
  - Uses &
  - Avoids a copy and allows changes to the original
- **Call by constant reference**
  - Uses `const&`
  - Avoids a copy and guarantees that actual parameter will not be changed

# Example

```
#include <iostream>
using std::cout;
using std::endl;

int squareByValue( int ); // pass by value
void squareByReference( int & ); // pass by reference
int squareByConstReference ( const int & ); // const ref.

int main()
{   int x = 2, z = 4, r1, r2;

    r1 = squareByValue(x);
    squareByReference( z );
    r2 = squareByConstReference(x);

    cout << "x = " << x << " z = " << z << endl;
    cout << "r1 = " << r1 << " r2 = " << r2 << endl;
    return 0;
}
```

# Example (cont.)

```
int squareByValue( int a )
{
    return a *= a;    // caller's argument not modified
}

void squareByReference( int &cRef )
{
    cRef *= cRef;    // caller's argument modified
}

int squareByConstReference (const int& a )
{
    // a *= a;  not allowed (compiler error)
    return a * a;
}
```

# Improving the Complex Class

Old class:

```
class Complex
{
    float re, im; // by default private
public:
    Complex(float x = 0, float y = 0)
        : re(x), im(y) { }

    Complex operator*(Complex rhs) const;
    float modulus() const;
    void print() const;
};
```

```
#ifndef _Complex_H
#define _Complex_H

using namespace std;
class Complex
{
    float re, im; // by default private
public:
    Complex(float x = 0, float y = 0)
        : re(x), im(y) { }

    Complex operator*(const Complex& rhs) const;
    float modulus() const;
    void print() const;
};

#endif
```

Complex class Interface in the file *Complex.h*

# Improving the Complex Class

*Complex.cpp*

```
#include <iostream>
#include <cmath>
#include "Complex.h"
```

```
Complex Complex::operator*(const Complex& rhs) const
{
    Complex prod;
    prod.re = (re*rhs.re - im*rhs.im);
    prod.im = (re*rhs.im + im*rhs.re);
    return prod;
}
```

```
float Complex::modulus() const
{
    return sqrt(re*re + im*im);
}
```

```
void Complex::print() const
{
    std::cout << "(" << re << "," << im << ")" << std::endl;
}
```

# The uses of keyword **const**

- **Const reference parameter:**

```
Complex operator*(const Complex& rhs) const;
```

In this case it means the parameter cannot be modified.

- **Const member function:**

```
Complex operator*(const Complex& rhs) const;
```

In this case it means the function cannot modify class members.

- **Const object/variable:**

```
const Complex c1(3, 4);
```

In this case it means the object cannot be modified.



# Memory Management

- In C++, we use **new** and **delete** instead of **malloc** and **free** used in C
  - **new** - automatically creates object of proper size, calls constructor, returns pointer of the correct type
  - **delete** - destroys object (calls the destructor) and frees space
- Example:

```
int* pi = new int(6);  
Complex *pc = new Complex(3, 5);  
delete pi;  
delete pc;
```

```
// Allocate an array of complex objects (calls the default  
// constructor for each object).
```

```
Complex *ptr1 = new Complex [10];
```

```
for (int i = 0; i < 10; ++i)  
    ptr[i]->print();
```

```
delete[] ptr1;           // note the delete[] syntax
```

```
int* ptr2 = new int[12];  // similar for int
```

```
delete [] ptr2;          // free up the dynamically  
                          // allocated array
```

# static Class Members

- Shared by all objects of a class
  - Normally, each object gets its own copy of each variable
- Efficient when a single copy of data is enough
  - Only the static variable has to be updated
- May seem like global variables, but have class scope
  - Only accessible to objects of same class
- Exist even if no instances (objects) of the class exist
- Can be variables or functions
  - public, private, or protected

# Example:

## Complex.h

```
private:
    static int count;
    ...
public:
    static int
    getCount();
    ...
```

## Complex.cpp

```
int Complex::count = 0;

int Complex::getCount()
{
    return count;
}

Complex::Complex()
{
    Re = 0;
    Imag = 0;
    count ++;
}
```

```
cout << Complex :: getCount() << endl;
Complex c1;
cout << c1.getCount();
```

## Driver program

# C++ Error Handling

- In C, errors are reported by returning error codes from functions:

```
int read(const char* filename, char data[])
{
    FILE* fp = fopen(filename, "r");
    if (fp == NULL)
        return -1;           // case of error in open

    // read file contents into data
    ...
}
```

# C++ Error Handling

- In C++, we have a more advanced mechanism called exceptions
- It uses three keywords: **throw**, **catch**, **try**
- The function that encounters an error throws an exception:

```
int read(const char* filename, char data[])
{
    FILE* fp = fopen(filename, "r");
    if (fp == NULL)
        throw "file open error"; // indicate error

    // read file contents into data
    ...
}
```

# C++ Error Handling

- This exception must be caught, otherwise the program will abnormally terminate:

```
int main()
{
    char data[128];
    try {
        read("test.txt", data);
        ... // some other code
    }
    catch(const char* error) {
        // if read throws an exception,
        // program will continue executing from here
        cout << "Error message:" << error << endl;
    }
}
```

Note that we throw an object or a variable, and we catch an object or a variable. These types should match for the exception to be caught

# Another Example

```
class FileReadError
{
};

int read(const char* filename, char data[])
{
    FILE* fp = fopen(filename, "r");
    if (fp == NULL)
        throw FileReadError(); // indicate error

    // read file contents into data
    ...
}

int main()
{
    char data[128];
    try {
        read("test.txt", data);
    }
    catch(FileReadError error) {
        // if read throws an exception,
        // we will come here
    }
}
```