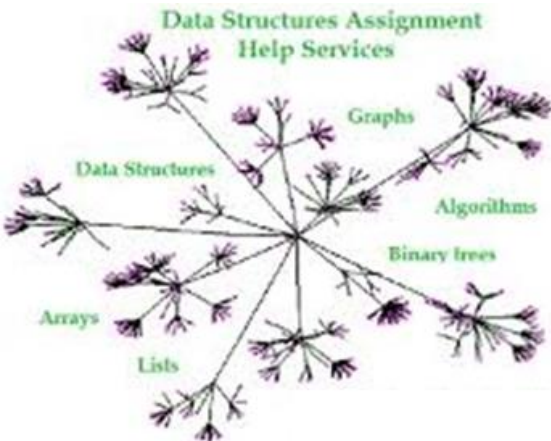
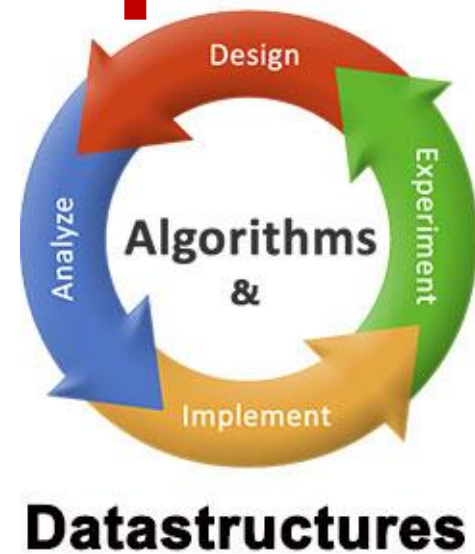


# CẤU TRÚC DỮ LIỆU & GIẢI THUẬT



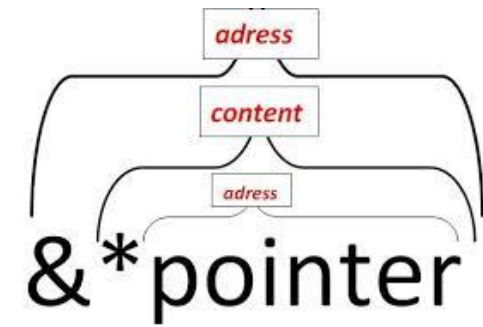
Lê Văn Hạnh

levanhanhvn@gmail.com

# NỘI DUNG MÔN HỌC

- Chương 1: Ôn tập ngôn ngữ lập trình C (*Review*) 3t
- **Chương 2: Kiểu dữ liệu con trỏ** (*Pointer Data Types*) 3t
- Chương 3: Tổng quan về cấu trúc dữ liệu và giải thuật (*Overview*) 3t
- Chương 4: Danh sách kê (Danh sách tuyến tính - *Contiguous List*) 3t
- Chương 5: Các giải thuật tìm kiếm trên danh sách kê (*The algorithms of Search*) 3t
- Chương 6: Các giải thuật sắp xếp trên danh sách kê (*The algorithms of Sort*) 6t
- Chương 7: Danh sách liên kết động (*Linked List*) 3t
- Chương 8: Ngăn xếp (*Stack*) 3t
- Chương 9: Hàng đợi (*Queue*) 3t
- Chương 10: Cây nhị phân tìm kiếm (*BST-Binary Search Tree*) 3t
- Chương 11: Cây nhị phân tìm kiếm cân bằng (*Balanced binary search tree*) 3t
- Chương 12: Bảng băm (*Hash Table*) 6t

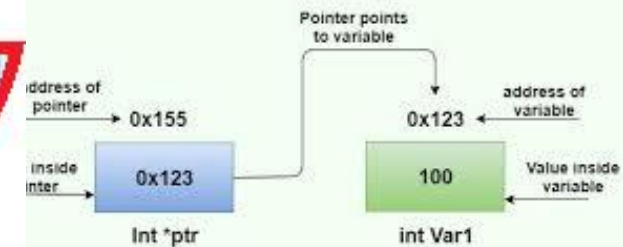
## CHƯƠNG 2



# Kiểu dữ liệu con trỏ (*Pointer Data Types*)



### Pointers in C++



# MỤC TIÊU

1. Hiểu khái niệm về con trỏ;
2. Biết cách khai báo và sử dụng biến kiểu con trỏ;
3. Biết xử lý các phép toán và các giải thuật trên:
  - Mảng một chiều theo kiểu con trỏ;
  - Mảng hai chiều theo kiểu con trỏ;
  - Con trỏ với kiểu dữ liệu có cấu trúc.
4. Hiểu được về con trỏ và cấp phát động.
  - Áp dụng con trỏ trong cấp phát mảng.
  - Áp dụng con trỏ và tham số của hàm.
  - Áp dụng con trỏ và cấu trúc.

# NỘI DUNG CHƯƠNG

1. Khái niệm về địa chỉ ô nhớ và con trỏ
2. Khai báo và sử dụng biến con trỏ
3. Các phép toán trên con trỏ
4. Sử dụng con trỏ để cấp phát và thu hồi bộ nhớ động
5. Con trỏ và hàm
6. Cấp phát bộ nhớ động
7. Con trỏ và mảng một chiều
8. Con trỏ và mảng hai chiều
9. Con trỏ với kiểu dữ liệu có cấu trúc (struct)
10. Câu hỏi ôn tập
11. Bài tập
12. Vấn đề mở rộng

# 1. ĐẶT VẤN ĐỀ

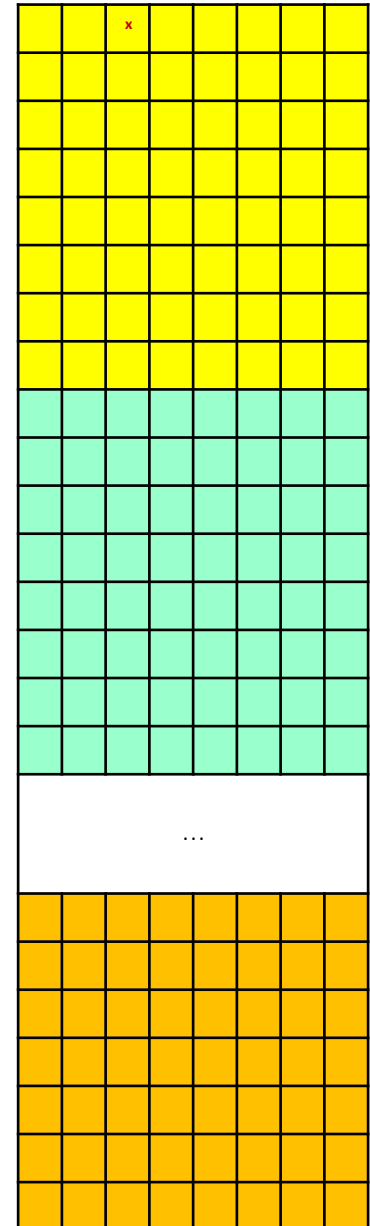
**Chương trình C/C++ quản lý các biến  
như thế nào?**



# 1. KHÁI NIỆM VỀ ĐỊA CHỈ Ô NHỚ VÀ CON TRỎ

## 1.1. Bộ nhớ chính

- *Bộ nhớ chính*: là thiết bị lưu trữ duy nhất để thông qua đó CPU có thể trao đổi thông tin với môi trường bên ngoài.
- Bộ nhớ chính được tổ chức như một mảng 1 chiều các từ nhớ (word), mỗi từ nhớ có 1 địa chỉ

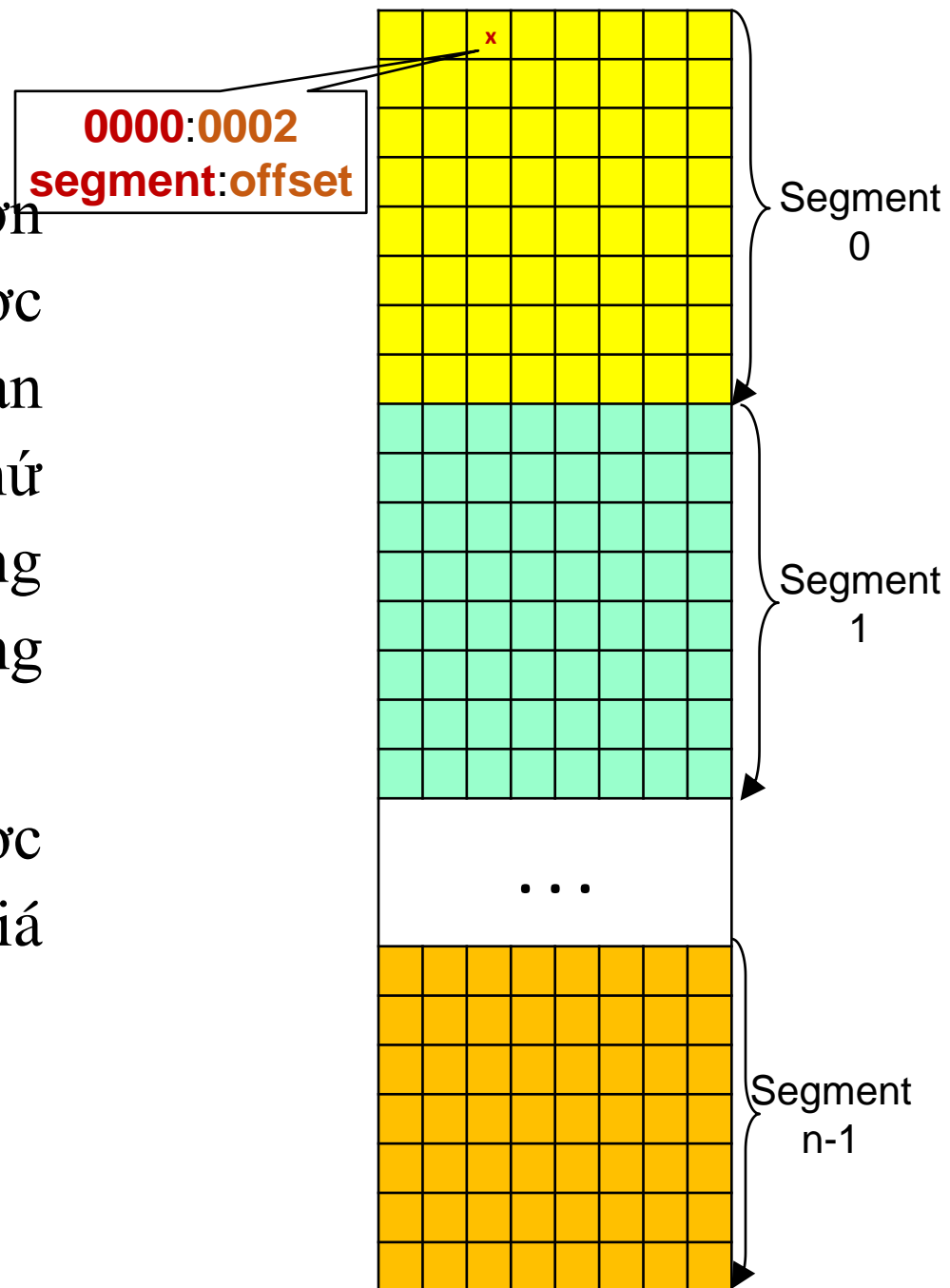


## 1. Khái niệm về địa chỉ ô nhớ và con trỏ

### 1.1. Bộ nhớ chính

- Do số lượng ô nhớ quá lớn nên nhiều ô nhớ sẽ được nhóm thành 1 đoạn (*segment*) được đánh số thứ tự từ 0 trở đi. Các ô trong mỗi đoạn (*offset*) cũng được đánh số từ 0 trở đi.
- Vậy địa chỉ mỗi ô nhớ được xác định thông qua cặp giá trị:

**segment: offset**





### 1.2. Biến tĩnh (Static variable) và vùng nhớ

- **Khi khai báo biến**: máy tính sẽ **dành riêng một vùng nhớ** để lưu biến đó. Thông tin máy tính cần lưu trữ về biến gồm:
  - **Tên biến**
  - **Kiểu dữ liệu** của biến
  - **Giá trị** của biến
  - **Địa chỉ vùng nhớ** nơi chứa giá trị của biến: nếu kích thước của biến gồm nhiều byte thì máy tính sẽ cấp phát một dãy các byte liên tiếp nhau, địa chỉ của biến sẽ **là địa chỉ byte đầu tiên** trong dãy các byte này.
- **Khi tên biến được gọi**: máy tính sẽ thực hiện 2 bước sau:
  - **B1**: Tìm kiếm địa chỉ ô nhớ của biến.
  - **B2**: Truy xuất hoặc thiết lập giá trị của biến được lưu trữ tại ô nhớ đó.

## 1. Khái niệm về địa chỉ ô nhớ và con trỏ

### 1.2. Biến tĩnh (Static variable) và vùng nhớ

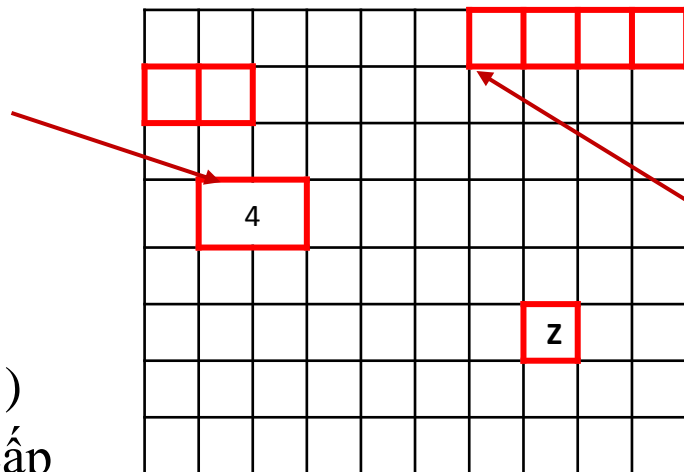
— VD:

```
void main()  
{  int x=4;  
    char a[6];  
    char c='Z';  
    ...  
}
```

Sau khi khai báo biến **x**,  
địa chỉ ô nhớ hệ điều  
hành cấp cho biến **x** là:

**A2B9:417C**

Giá trị của biến **x** (=4)  
được lưu tại địa chỉ đã cấp  
cho biến **x** (**A2B9:417C**)




Địa chỉ ô nhớ hệ  
điều hành cấp  
cho mảng **a** là:

**A2B9:3F8E**

*Memory Layout*

### 1.3. Toán tử & và \*

- **Toán tử &**: (*Address of Operator*) đặt trước tên biến và cho biết địa chỉ của vùng nhớ của biến.

 Địa chỉ của biến luôn luôn là một số nguyên (hệ thập lục phân) dù biến đó chứa giá trị là số nguyên, số thực hay ký tự, ...

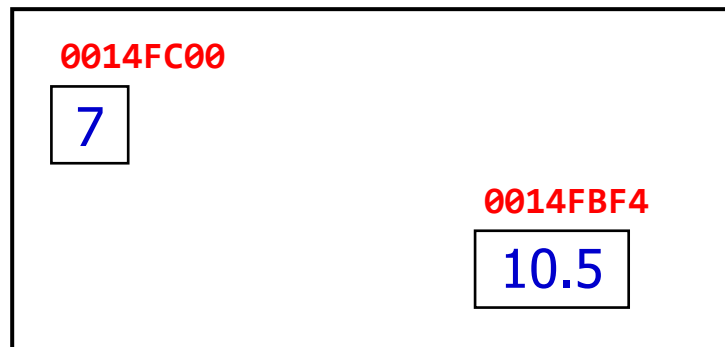
- **Toán tử \*** (*Dereferencing Operator* hay *Indirection Operator*) đặt trước một địa chỉ và cho biết giá trị lưu trữ tại địa chỉ đó.

## 1. Khái niệm về địa chỉ ô nhớ và con trỏ

### 1.3. Toán tử & và \*

— Ví dụ:

```
void main()  
{  
    int x = 7;  
    float y = 10.5;  
    printf("Gia tri bien x=%d, duoc luu tai dia chi %x\n",x,&x);  
    printf("Gia tri bien y=%.1f, duoc luu tai dia chi %x",*(&y),&y);  
}
```



*Memory Layout*

Các giá trị này có thể thay đổi trong mỗi lần chương trình được thực hiện

#### Kết quả

Gia tri bien x=7, duoc luu tai dia chi **0014FC00**  
Gia tri bien y=10.5, duoc luu tai dia chi **0014FBF4**

## 1.4. Một số hạn chế của biến tĩnh

- Cấp phát ô nhớ dư, gây ra lãng phí ô nhớ.

VD: `char a[20]; int n=3;`



- Cấp phát ô nhớ thiếu, chương trình thực thi bị lỗi.

VD: `char a[20]; int n=3;`

`a[20]=5; // báo lỗi`



- Các biến tĩnh sẽ tồn tại trong suốt thời gian thực thi chương trình (dù biến chỉ sử dụng 1 lần rồi bỏ)

## **1.5. Biến “động” (*Dynamic variable*)**

Để khắc phục hạn chế của biến tĩnh, ngôn ngữ C cung cấp một loại biến đặc biệt gọi là biến động với các đặc điểm sau:

- Chỉ phát sinh trong quá trình thực hiện chương trình (biến tĩnh phát sinh lúc bắt đầu chương trình).
- Khi chương trình đang thực thi, **kích thước của biến, vùng nhớ và địa chỉ vùng nhớ** được cấp phát cho biến **có thể thay đổi**.
- Sau khi sử dụng xong có thể giải phóng để tiết kiệm bộ nhớ.

### 1.6. Con trỏ (*pointer*)

- Do biến động không có địa chỉ nhất định nên người lập trình không thể truy cập đến chúng được.
- Để khắc phục tình trạng này, ngôn ngữ C cung cấp thêm một loại biến đặc biệt nữa là biến con trỏ (*pointer*).
- Con trỏ (*pointer*) là một **biến lưu trữ địa chỉ** của một **địa chỉ bộ nhớ**. Địa chỉ này thường là địa chỉ của một biến khác.

VD: Biến ***x*** lưu giữ (chứa) địa chỉ của biến ***y***. Vậy ta nói biến ***x*** “***trỏ tới***” ***y***.

## ***1.6. Con trỏ (pointer)***

### ***– Các đặc điểm của biến con trỏ***

- Biến con trỏ không chứa dữ liệu mà chỉ chứa địa chỉ của ô nhớ chứa dữ liệu.
- Kích thước của biến con trỏ không phụ thuộc vào kiểu dữ liệu, luôn có **kích thước cố định là 2 byte**.

### ***- Phân loại con trỏ***

- Con trỏ kiểu ***int*** dùng để chứa địa chỉ của các biến kiểu ***int***.
- Tương tự ta có con trỏ kiểu ***float***, ***double***, ...



1. Khái niệm về địa chỉ ô nhớ và con trỏ

1.7. Bảng các thuật ngữ liên quan nội dung con trỏ

Thuật ngữ tiếng Việt	Thuật ngữ tiếng Anh
Con trỏ	<i>Pointer</i>
Hằng con trỏ	<i>Constant pointer</i>
Địa chỉ bộ nhớ	<i>Memory Address</i>
Toán tử &	<i>Address-of Operator</i>
Toán tử *	<i>Dereferencing Operator, or: Indirection Operator</i>
Cấp phát bộ nhớ	<i>Memory Allocation</i>
Giải phóng bộ nhớ	<i>De-Allocate Memory</i>
Cấp phát tĩnh	<i>Static Memory Allocation</i>
Cấp phát động	<i>Dynamic Memory Allocation</i>
Biến động	<i>Dynamic Variable</i>
Phép toán số học trên con trỏ	<i>Pointer Arithmetic</i>

## 2. KHAI BÁO VÀ SỬ DỤNG BIẾN CON TRỞ

### 2.1. Khai báo biến con trỏ

- **Cú pháp:** `<Kiểu_dữ_liệu> * <Tên_con_trỏ>;`
- **Ý nghĩa:** Khai báo một biến có tên là `Tên_con_trỏ` dùng để chứa địa chỉ của các biến có thuộc `Kiểu_dữ_liệu`.
- **Ví dụ**
  - Khai báo 2 biến `a`, `x` có kiểu `int` và 2 biến `pa`, `px` là 2 biến con trỏ

```
int a, x=3, *pa, *px;
```
  - Khai báo biến `y` kiểu `float` và biến `py` là con trỏ `float`

```
float y=7.15, *py;
```
  - Sử dụng:

```
px = &x;  
py = &y;  
px = &y; // lỗi do sai kiểu dữ liệu
```

## 2. Khai báo và sử dụng biến con trỏ

### 2.1. Khai báo biến con trỏ

– Toán tử **&** dùng để định vị con trỏ đến địa chỉ của một biến đang làm việc.

– **Cú pháp:** **<Tên\_biến\_con\_trỏ> = & <Tên\_biến>**

– **Ví dụ:**

- Khai báo biến **int \*px , x = 5 ;**

- Gán địa chỉ của biến x cho con trỏ xp. **px = &x;**



- Lúc này

```
printf("x=%d\n", x);  
printf("*px=%d\n", *px);
```

```
printf("&x=%ld\n",&x);  
printf("px=%ld\n", px);
```

## 2. Khai báo và sử dụng biến con trỏ

### 2.2. Thao tác trên biến con trỏ

#### 2.2.2. Lấy (hoặc thay đổi) giá trị của biến qua con trỏ tham chiếu tới biến

– **Cú pháp:**     \***<Tên biến con trỏ>**

– **Ví dụ 1**

```
int x=4, *px;  
px= &x;  
int y= *px;  
*px = 10;
```

1C2E: 

4
---

  
x

A06D: 

1C2E
------

  
px

8F2E: 

4
---

  
y

– **Lưu ý**

- Có thể truy nhập tới biến bằng 1 trong 2 cách:
  - Truy nhập trực tiếp tới biến.
  - Truy nhập gián tiếp thông qua biến con trỏ.
- Khi gán địa chỉ của một biến cho một biến con trỏ, mọi sự thay đổi trên nội dung ô nhớ con trỏ chỉ tới sẽ làm giá trị của biến thay đổi theo.

2. Khai báo và sử dụng biến con trỏ

2.2. Thao tác trên biến con trỏ

2.2.2. Lấy giá trị của biến con trỏ chỉ tới

– Ví dụ 2

```
#include <stdio.h>
#include <conio.h>
int main()
{
    int a=2, *pa;
    pa = &a;

    printf("\nGia tri cua bien a (su dung 'a')    = %d \n", a) ;           //⇒2
    printf("\nGia tri cua bien a (su dung '*pa') = %d \n", *pa);           //⇒2

    printf("\nDia chi bien a (su dung '&a')      = %d\n", &a); //⇒ 11204948
    printf("\nDia chi bien a (su dung 'pa')      = %x\n", pa); //⇒ aaf954
    printf("\nDia chi bien a (su dung '&(*pa)')= %d\n", &(*pa)); //⇒ 11204948
    printf("\nDia chi bien a su dung '*(&pa)'   = %x\n", *(&pa)); //⇒ aaf954
    *pa = 10;

    printf("\nSau gan *pa = 10, gia tri cua a (su dung 'a')    = %d \n", a);
    printf("\nSau gan *pa = 10, gia tri cua a (su dung '*pa') = %d \n", *pa);

    a = 5;
    printf("\nSau gan a=5, gia tri cua a (su dung 'a')    = %d \n", a); //⇒5
    printf("\nSau gan a=5, gia tri cua a (su dung '*pa') = %d \n", *pa); //⇒5
    printf("\nDung '&pa' de biet dia chi cua bien con tro pa = %x \n", &pa); //⇒ aaf948

    getch();
}
```

## 2. Khai báo và sử dụng biến con trỏ

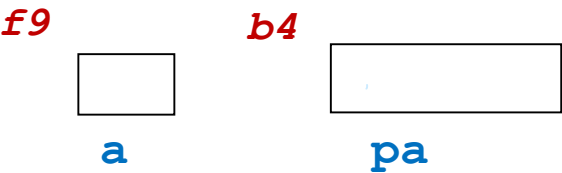
### 2.2. Thao tác trên biến con trỏ

#### 2.2.2. Lấy giá trị của biến con trỏ chỉ tới

##### – Thực hành

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

```
int main()
{   int a=4, *pa;
    pa = &a;
```



```
printf("\nGia tri cua bien a (su dung 'a')    = %d \n", a)  ;
printf("\nGia tri cua bien a (su dung '*pa') = %d \n", *pa);
```

```
printf("\nDia chi bien a (su dung '&a')      = %d\n", &a);
printf("\nDia chi bien a (su dung 'pa')      = %x\n", pa);
printf("\nDia chi bien a (su dung '&(*pa)')= %d\n", &(*pa));
printf("\nDia chi bien a su dung '*(&pa)'   = %x\n", *(&pa));
```

```
*pa = 7;
printf("\nSau gan *pa = 10, gia tri cua a (su dung 'a')  = %d \n", a);
printf("\nSau gan *pa = 10, gia tri cua a (su dung '*pa') = %d \n", *pa);
```

```
a = *pa*2;
printf("\nSau gan a=5, gia tri cua a (su dung 'a')  = %d \n", a);
printf("\nSau gan a=5, gia tri cua a (su dung '*pa') = %d \n", *pa);
printf("\nDung '&pa' de biet dia chi cua bien con tro pa = %x \n", &pa);
getch();
}
```

## 2. Khai báo và sử dụng biến con trỏ

### 2.2. Thao tác trên biến con trỏ

#### 2.2.2. Lấy giá trị của biến con trỏ chỉ tới

##### — Ví dụ 3

```
#include <stdio.h>
#include <conio.h>
int main()
{
```

```
    int x, y, *px , *py;
```

```
    x  = 5;    // Truy nhập trực tiếp tới biến x
```

```
    px = &x;
```

```
    py = px;
```

```
    y  = *px;
```

```
    *py =17;
```

```
}
```

1C2E:

17

x

8F2E:

5

y

A06D:

1C2E

px

A06D:

1C2E

py

## 2. Khai báo và sử dụng biến con trỏ

### 2.2. Thao tác trên biến con trỏ

#### 2.2.2. Lấy giá trị của biến con trỏ chỉ tới

- **Ví dụ 4:** viết hàm hoán vị 2 biến a và b bằng 2 cách dùng và không dùng biến con trỏ

##### //sử dụng con trỏ

```
#include <stdio.h>
#include <conio.h>
void HoanVi(int *a, int *b)
{
    int tam;
    tam = *a;
    *a = *b;
    *b = tam;
}
void main()
{
    int x = 3, y = 7;
    printf("TRUOC khi hoan vi,
           x= %d, y=%d", x, y);
    HoanVi(&x, &y);
    printf("\nSAU khi hoan vi,
           x= %d, y=%d", x, y);
    getch();
}
```

##### //sử dụng tham chiếu

```
#include <stdio.h>
#include <conio.h>
void HoanVi(int &a, int &b)
{
    int tam;
    tam = a;
    a = b;
    b = tam;
}
void main()
{
    int x = 3, y = 7;
    printf("TRUOC khi hoan vi,
           x= %d, y=%d", x, y);
    HoanVi(x, y);
    printf("\nSAU khi hoan vi
           x= %d, y=%d", x, y);
    getch();
}
```

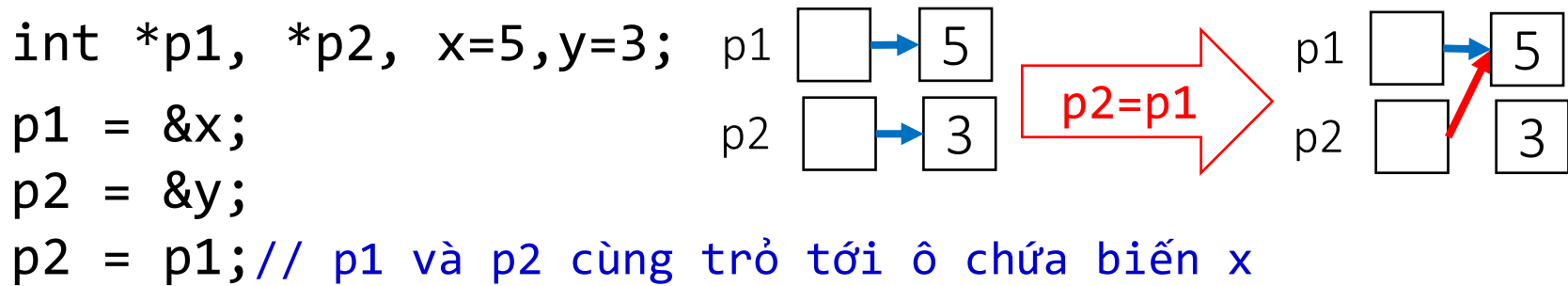


## 2. Khai báo và sử dụng biến con trỏ

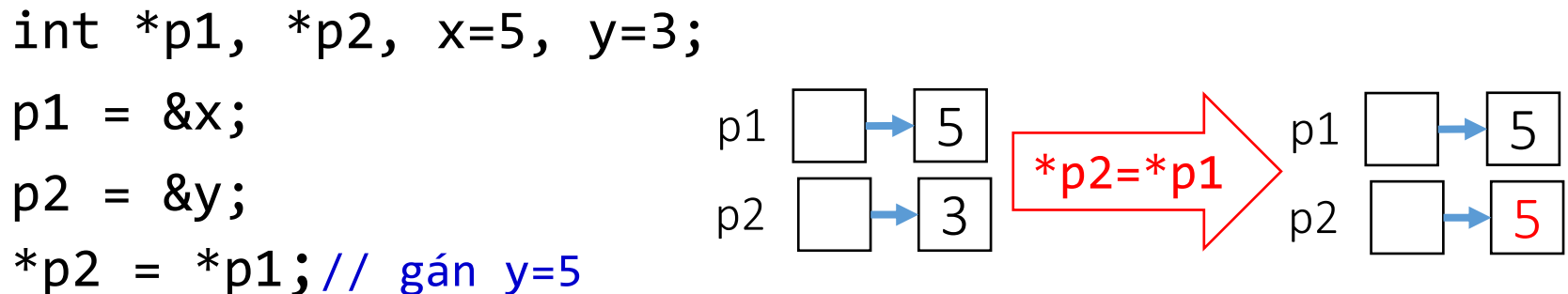
### 2.2. Thao tác trên biến con trỏ

#### 2.2.3. Phép gán con trỏ

- Có thể gán biến con trỏ có cùng kiểu:



- Dễ bị lẫn với:  $*p2 = *p1; //$  Gán “giá trị trỏ bởi p1” cho “giá trị trỏ bởi p2”



### 3. Con trỏ kiểu void

- Con trỏ kiểu void là một loại con trỏ có thể trỏ đến các biến có kiểu bất kỳ.
- Cú pháp khai báo: `void *<tên biến con trỏ>;`

– Ví dụ:

```
void main()
{
    void *p; //p là con trỏ kiểu void
    char c = 'a';
    int d = 10;
    float f = 30.50;
    p = &c;
    /*Phải ép kiểu khi sử dụng vì p là con trỏ kiểu void*/
    printf("%c\n", *((char *)p));           /*ép về kiểu char*/
    p = &d;
    printf("%d\n", *((int *)p));             /*ép về kiểu int*/
    p = &f;
    printf("%0.2f\n", *((float *)p));        /*ép về kiểu float*/
}
```

## 4. CÁC PHÉP TOÁN TRÊN CON TRỞ

Có bốn phép toán liên quan đến con trỏ và địa chỉ:

- Phép gán
- Phép tăng giảm địa chỉ
- Phép truy nhập bộ nhớ
- Phép so sánh

## 4. Các phép toán trên con trỏ

### 4.1. Phép gán

- Chỉ nên thực hiện phép gán cho các con trỏ cùng kiểu

Ví dụ: `int x=1, *pi, *qi;`

`pi= &x;`

`qi=pi; /*sao chép nội dung của pi vào qi`

`⇒pi và qi cùng trỏ đến biến x)*/`

**1C2E:**

1

**x**

**A06D:**

1C2E

**pi**

**A06D:**

1C2E

**qi**

- Muốn gán các con trỏ khác kiểu phải dùng phép ép kiểu.

Ví dụ: `int x;`

`char *pc;`

`pc=(char *) (&x); //ép kiểu`

#### 4. Các phép toán trên con trỏ

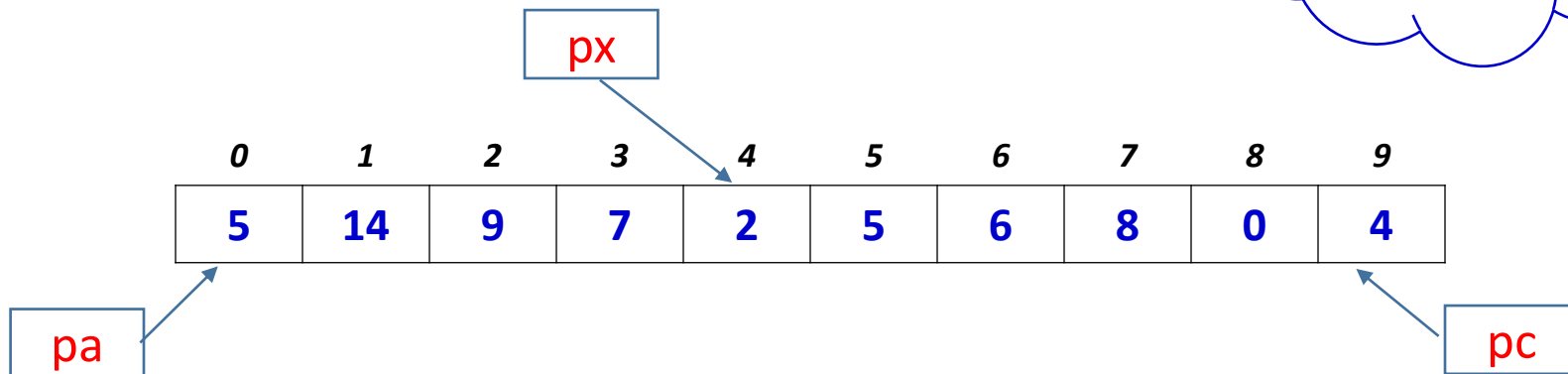
### 4.2. Phép tăng giảm địa chỉ

- Có thể cộng (+) hoặc trừ (-) **1 con trỏ** với 1 **số nguyên n**: kết quả trả về là 1 con trỏ. Con trỏ này chỉ đến vùng nhớ cách vùng nhớ của con trỏ hiện tại **n** phần tử.

Ví dụ: `float a[10], *px, *pa, *pc, i=1;`  
`pa = &a[0];` //px trỏ đến phần tử a[0]  
`pc = &a[9];` //px trỏ đến phần tử a[9]  
`px = &a[4];` //px trỏ đến phần tử a[4]

Khi đó: **px++** trỏ đến phần tử **a[5]**  
**px+i** trỏ đến phần tử **a[4+i]**  
**px-i** trỏ đến phần tử **a[4-i]**

Sẽ phát sinh  
lỗi khi  
 $4 - i < 0$   
hoặc  $4 + i > n - 1$



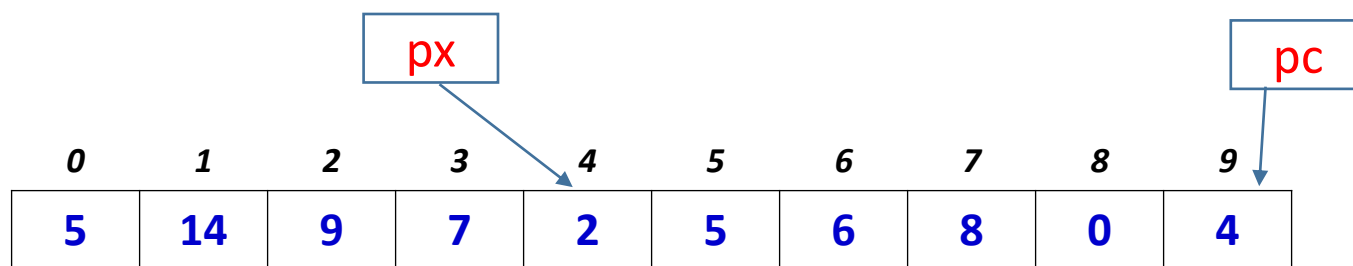
#### 4. Các phép toán trên con trỏ

### 4.2. Phép tăng giảm địa chỉ

- Phép trừ 2 **con trỏ cùng kiểu** sẽ trả về 1 **giá trị nguyên** (**int**). Đây chính là khoảng cách (số phần tử) giữa 2 con trỏ đó.

Chẳng hạn, trong ví dụ trên,  $pc - px = 5$

- **Chú ý**: 2 phép toán trên không dùng cho con trỏ kiểu void
- Con trỏ **NULL**: là con trỏ không chứa địa chỉ nào cả. Có thể gán giá trị **NULL** cho 1 con trỏ có kiểu bất kỳ.



#### 4. Các phép toán trên con trỏ

### *4.3. Phép truy nhập bộ nhớ*

#### *— Nguyên tắc:*

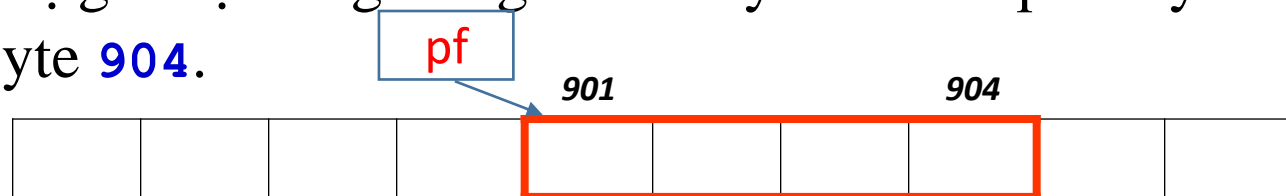
- Con trỏ `float` truy nhập **4** byte.
- Con trỏ `int` truy nhập **2** byte.
- Con trỏ `char` truy nhập **1** byte

### 4.3. Phép truy nhập bộ nhớ

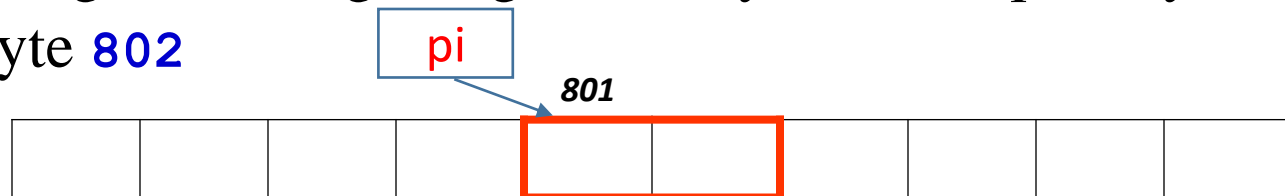
Ví dụ      `float *pf ; int      *pi ; char      *pc ;`

Khi đó:

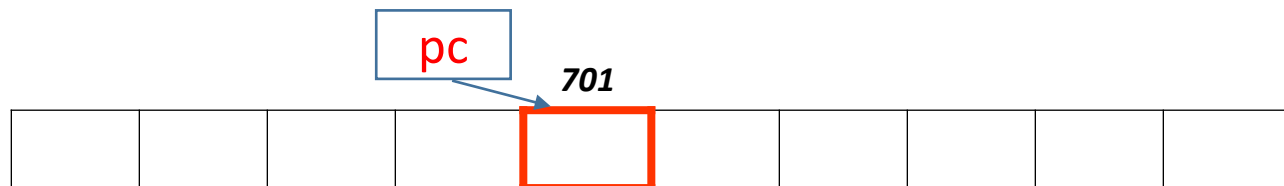
- Nếu **pf** trỏ đến byte thứ **10001**  
⇒ **\*pf** biểu thị giá trị trong vùng nhớ **4** byte liên tiếp từ byte **901** đến byte **904**.



- Nếu **pi** trỏ đến byte thứ **801**  
⇒ **\*pi** biểu thị giá trị trong vùng nhớ **2** byte liên tiếp từ byte **801** đến byte **802**



- Nếu **pc** trỏ đến byte thứ **701**  
⇒ **\*pc** biểu thị giá trị trong vùng nhớ **1** byte là byte **701**.





## 4. Các phép toán trên con trỏ

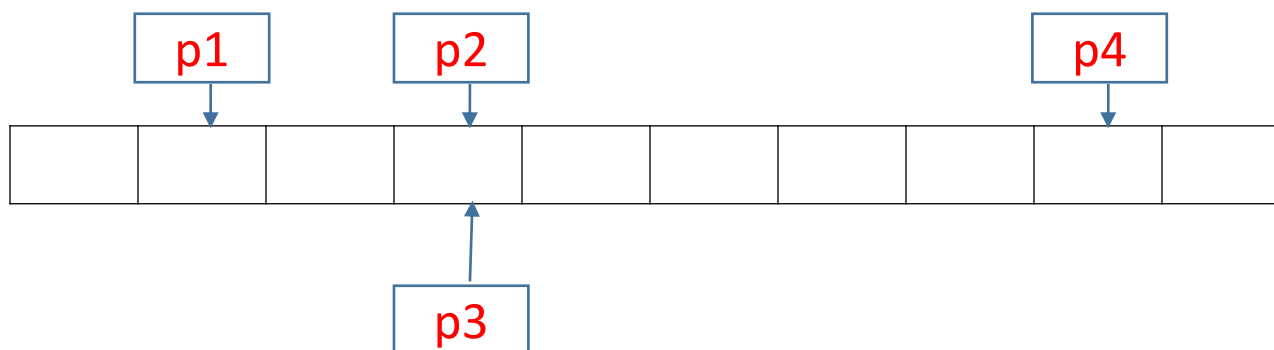
### 4.4. Phép so sánh

Dùng cho phép so sánh 2 con trỏ cùng kiểu

- $p1 < p2$  nếu địa chỉ  $p1$  trỏ tới **thấp** hơn địa chỉ  $p2$  trỏ tới
- $p1 = p2$  nếu địa chỉ  $p1$  trỏ tới **bằng** địa chỉ  $p2$  trỏ tới
- $p1 > p2$  nếu địa chỉ  $p1$  trỏ tới **cao** hơn địa chỉ  $p2$  trỏ tới.

Ví dụ: cho minh họa như hình vẽ, biểu thức so sánh

- `if (p1 < p2)` trả về kết quả **true**
- `if (p1 == p2)` trả về kết quả **false**
- `if (p1 > p2)` trả về kết quả **false**



## 5. CON TRỎ VÀ HÀM

- Con trỏ là kiểu dữ liệu hoàn chỉnh có thể dùng nó như các kiểu khác
- Con trỏ có thể là tham số của hàm
- Có thể là kiểu trả về của hàm

Ví dụ: `int* findOtherPointer(int* p);`

Hàm này khai báo:

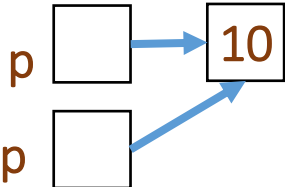
- Có tham số kiểu con trỏ trỏ tới int
- Trả về biến con trỏ trỏ tới kiểu dữ liệu int

## 5. Con trỏ và hàm

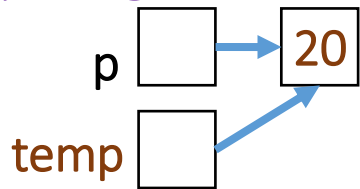
### Ví dụ

```
typedef int *IntPtr;
void Input(IntPtr temp)
{
    *temp = 20;
    printf("Trong ham goi *temp = %d\n", *temp);
}
int main() {
    IntPtr p = new int;
    *p = 10;
    printf("Truoc khi goi ham, *p = %d\n", *p);
    Input(p);
    printf("Sau khi ket thuc ham, *p = %d\n", *p);
}
```

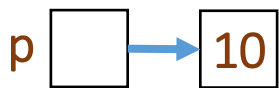
2. Giá trị của p sẽ được truyền vào temp



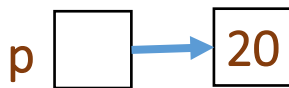
3. Thay đổi giá trị \*temp



1. Trước khi gọi hàm Input



4. Sau khi kết thúc gọi hàm Input



Truoc khi goi ham, \*p = 10  
Trong ham goi \*temp = 20  
Sau khi ket thuc ham, \*p = 20

## 5. Con trỏ và hàm

**Bài tập** Khai báo kiểu con trỏ kiểu int. Viết hàm cấp phát bộ nhớ cho 1 con trỏ và gán con trỏ vừa cấp phát trỏ vào giá trị 15 theo 2 cách:

- i. Hàm trả về con trỏ kiểu int vừa định nghĩa, hàm không nhận tham số.
- ii. Kiểu trả về của hàm là void. Hàm nhận tham số là kiểu con trỏ vừa định nghĩa dưới dạng tham biến.

### Bài giải

```
// Cách 1
#include <stdio.h>
typedef int *IntPtreter;
IntPtreter Input()
{   IntPtreter temp = new int;
    *temp = 15;
    return temp;
}
int main()
{   IntPtreter p;
    p = Input();
    printf("Sau khi ket thuc
        ham, *p = %d\n", *p);
}
```

```
// Cách 2
#include <stdio.h>
typedef int *IntPtreter;
void Input(IntPtreter &temp)
{   temp = new int;
    *temp = 15;
}
int main()
{   IntPtreter p;
    Input(p);
    printf("Sau khi ket thuc ham, *p
        = %d\n", *p);
}
```

## 6. CẤP PHÁT BỘ NHỚ ĐỘNG

- **Cấp phát bộ nhớ tĩnh** (*static memory allocation*)

- Khai báo biến, cấu trúc, mảng, ...
- Bắt buộc phải biết trước cần bao nhiêu bộ nhớ lưu trữ

⇒ tồn bộ nhớ, không thay đổi được kích thước, ...

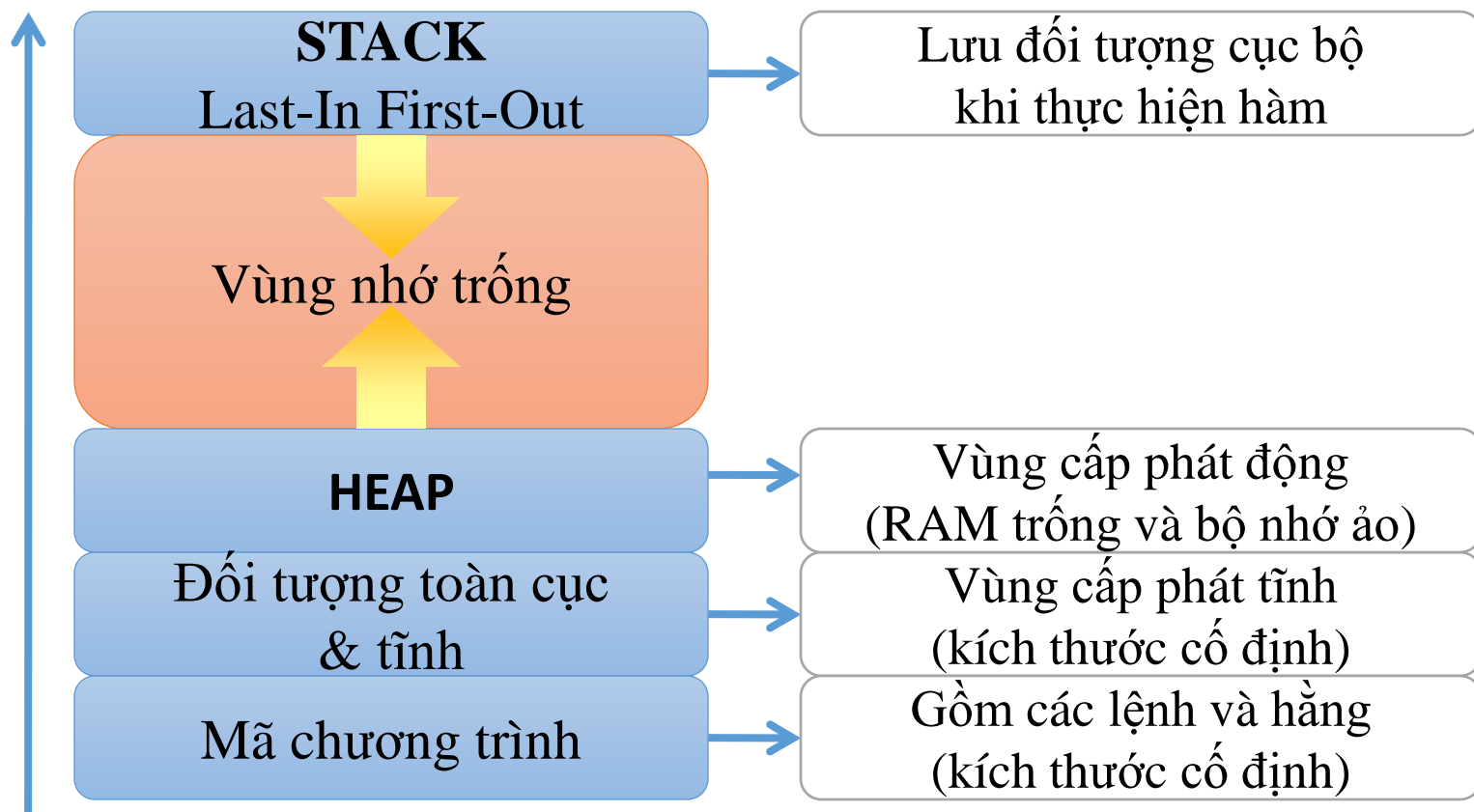
- **Cấp phát động** (*dynamic memory allocation*)

- Cần bao nhiêu cấp phát bấy nhiêu.
- Có thể giải phóng nếu không cần sử dụng.
- Sử dụng vùng nhớ ngoài chương trình (cả bộ nhớ ảo *virtual memory*).

## 6. Cấp phát bộ nhớ động

### 6.1. Cấu trúc một CT C++ trong bộ nhớ

- Toàn bộ tập tin chương trình sẽ được nạp vào bộ nhớ tại vùng nhớ còn trống, gồm 4 phần:



## 6. Cấp phát bộ nhớ động

### 6.2. Cấp phát và giải phóng bộ nhớ

#### - *Cấp phát bộ nhớ*

- Trong C: Hàm *malloc*, *calloc*, *realloc* (<stdlib.h> hoặc <alloc.h>)
- Trong C++: Toán tử *new*

#### - *Giải phóng bộ nhớ*

- Trong C: Hàm *free*
- Trong C++: Toán tử *delete*

## 6. Cấp phát bộ nhớ động

### 6.2. Cấp phát và giải phóng bộ nhớ

#### 6.2.1. Hàm cấp phát bộ nhớ malloc

- *Cú pháp*: `void *malloc(size_t n);`
- Hàm xin cấp phát vùng nhớ cho n phần tử, mỗi phần tử có kiểu dữ liệu (kích thước) là `size_t`.
- *Kết quả trả về của hàm*:
  - *Thành công*: hàm trả về địa chỉ đầu vùng nhớ được cấp phát.
  - *Không thành công*: hàm trả về trị `NULL`.



## 6. Cấp phát bộ nhớ động

### 6.2. Cấp phát và giải phóng bộ nhớ

#### 6.2.1. Hàm cấp phát bộ nhớ malloc

Ví dụ : dùng hàm malloc

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include <process.h>
int main()
{
    char *str;
    str = (char *) malloc(10); /*allocate memory for string*/
    if(str == NULL)
    {
        printf("Not enough memory to allocate
                                                    buffer\n");
        exit(1); /*terminate program if out of memory*/
    }
    strcpy(str, "Hello"); /*copy "Hello" into string*/
    printf("String is %s\n", str); /*display string*/
    free(str); /* free memory */
    return 0;
}
```

## 6. Cấp phát bộ nhớ động

### 6.2. Cấp phát và giải phóng bộ nhớ

#### 6.2.2. Hàm cấp phát bộ nhớ `calloc`

- **Cú pháp:** `void *calloc(size_t nItems, size_t size);`
- **Giải thích:** Cấp phát vùng nhớ `nItems*` với kích thước `size` byte. Nếu thành công hàm trả về địa chỉ đầu vùng nhớ được cấp phát. Khi không đủ bộ nhớ để cấp phát hàm trả về giá trị `NULL`.
- **Lưu ý**
  - Hàm `calloc` cấp phát vùng nhớ và khởi tạo tất cả các bit trong vùng nhớ mới cấp phát về 0.
  - Hàm `malloc` chỉ cấp phát vùng nhớ.

## 6. Cấp phát bộ nhớ động

### 6.2. Cấp phát và giải phóng bộ nhớ

#### 6.2.2. Hàm cấp phát bộ nhớ calloc

##### — Ví dụ

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>
int main(void)
{
    char *str = NULL;
    /*allocate memory for string*/
    str = (char *) calloc(10, sizeof(char));
    strcpy(str, "Hello"); /*copy "Hello" into string*/
    printf("String is %s\n", str); /*display string*/
    free(str);             /*free memory*/
    return 0;
}
```

## 6. Cấp phát bộ nhớ động

### 6.2. Cấp phát và giải phóng bộ nhớ

#### 6.2.3. Hàm cấp phát bộ nhớ realloc

- **Công dụng:** Hàm thay đổi kích thước vùng nhớ đã cấp phát trước đó. Vùng nhớ mới có thể có địa chỉ khác so với vùng nhớ cũ. Phần dữ liệu trên vùng nhớ cũ được chuyển đến vùng nhớ mới.
- **Cú pháp:** `void* realloc(void *ptr, unsigned size);`
- **Giải thích**
  - `ptr`: trỏ đến vùng nhớ đã được cấp phát trước đó.
  - `size`: là số byte cần cấp phát lại.

## 6. Cấp phát bộ nhớ động

### 6.2. Cấp phát và giải phóng bộ nhớ

#### 6.2.3. Hàm cấp phát bộ nhớ realloc

— Ví dụ

```
int a, *pa;
```

```
/*Xin cấp phát vùng nhớ có kích thước (10x2) byte*/
```

```
pa = (int*) malloc (10);
```

```
pa = realloc (pa, 16);
```

```
/*Xin cấp phát lại vùng nhớ có kích thước (16x2) byte*/
```

### 6.3. Biến cục bộ và Biến cấp phát động

#### - *Biến cục bộ*

- Khai báo bên trong định nghĩa hàm
- Sinh ra khi hàm được gọi
- Hủy đi khi hàm kết thúc
  - Thường gọi là biến tự động nghĩa là được trình biên dịch quản lý một cách tự động.

#### - *Biến cấp phát động*

- Sinh ra bởi cấp phát động
- Sinh ra và hủy đi khi chương trình đang chạy
- Biến cấp phát động hay Biến động là biến con trỏ trước khi sử dụng được cấp phát bộ nhớ.

### 6.4. Toán tử new

- Vì con trỏ có thể tham chiếu tới biến nhưng không thực sự cần phải có định danh cho biến đó.
- Có thể cấp phát động cho biến con trỏ bằng toán tử **new**. Toán tử new sẽ tạo ra biến “không tên” cho con trỏ trỏ tới.
- Cú pháp: **<type> \*<pointerName> = new <type>**
- Ví dụ: **int \*ptr = new int;**
  - Tạo ra một biến “không tên” và gán ptr trỏ tới nó
  - Có thể làm việc với biến “không tên” thông qua **\*ptr**

## 6. Cấp phát bộ nhớ động

### 6.5. Kiểm tra việc cấp phát có thành công hay không?

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{    int *p = new int;
```

```
    if (p == NULL)
```

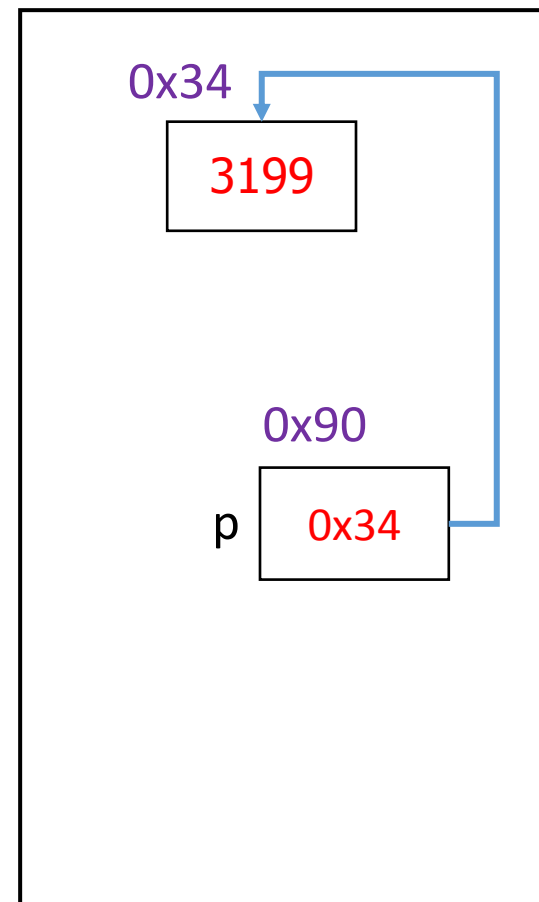
```
    {    printf("Error: Không đủ bộ nhớ.\n");
```

```
        exit(1);
```

```
    }
```

```
    *p = 3199;
```

```
}
```





## 6. Cấp phát bộ nhớ động

### 6.6. Khởi tạo giá trị trong cấp phát động

- Cú pháp: `<type> pointer = new <type> (value)`
- Ví dụ:

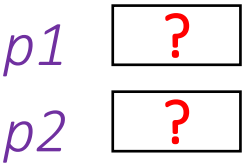
```
#include <iostream>
using namespace std;
int main()
{
    int *p;
    p = new int(99); // initialize with 99
    printf("%d" , *p); // displays 99
    return 0;
}
```

6. Cấp phát bộ nhớ động

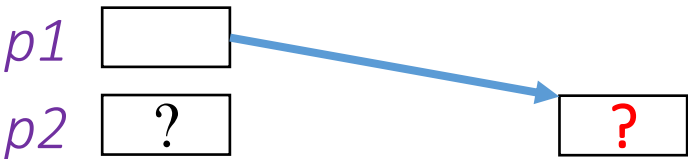
6.6. Khởi tạo giá trị trong cấp phát động

Ví dụ

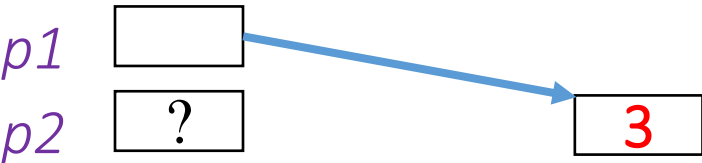
```
1. int *p1, *p2;
```



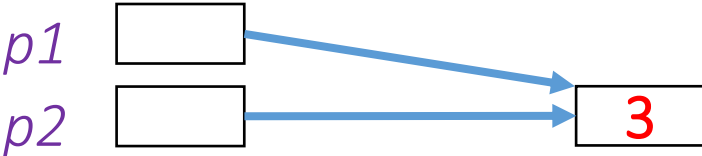
```
2. int *p1 = new int;
```



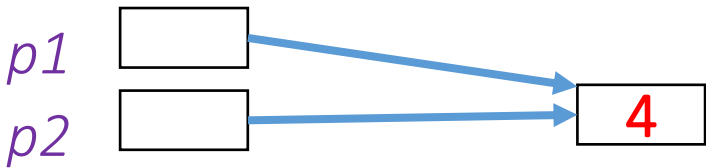
```
3. *p1 = 3;
```



```
4. p2 = p1;
```



```
5. *p2 = 4;
```



```
6. p1 = new int;
```



```
7. *p1 = 5;
```



## 6. Cấp phát bộ nhớ động

### 6.7. Giải phóng vùng nhớ đã cấp phát

- Khi sử dụng con trỏ để xin cấp phát bộ nhớ thì khi dùng xong người lập trình bắt buộc phải trả lại bộ nhớ. Để thu hồi bộ nhớ ta có thể dùng hàm *free* hoặc toán tử *delete*.
- Để thu hồi bộ nhớ:
  - Sử dụng hàm *free*: khi sử dụng các hàm *malloc*, *calloc*, *realloc* cấp phát bộ nhớ cho 1 biến con trỏ.
  - Sử dụng toán tử *delete*: khi sử dụng toán tử *new* để xin cấp phát bộ nhớ.

### 6.7. Giải phóng vùng nhớ đã cấp phát

- Toán tử *delete* dùng để giải phóng vùng nhớ trong HEAP do con trỏ trỏ tới (con trỏ được cấp phát bằng toán tử *new*).

- *Ghi chú*: Sau khi gọi toán tử *delete* thì con trỏ vẫn trỏ tới vùng nhớ trước khi gọi hàm *delete*. Ta gọi là “*con trỏ lạc*”. Ta vẫn có thể gọi tham chiếu trên con trỏ, tuy nhiên:

- ⇒ Hãy tránh con trỏ lạc bằng cách gán con trỏ bằng ***NULL*** sau khi ***delete***.

52

## 6. Cấp phát bộ nhớ động

### 6.8. Định nghĩa kiểu con trỏ

#### 6.8.1. Từ khóa typedef

- Từ khóa typedef dùng để định nghĩa 1 tên mới hay gọi là một biệt danh (alias) cho tên kiểu dữ liệu có sẵn.
- Ví dụ: `typedef int SONGUYEN;`

Sau đó, các khai báo sau là tương đương:

```
int a;
```

```
SONGUYEN a;
```

## 6. Cấp phát bộ nhớ động

### 6.8. Định nghĩa kiểu con trỏ

#### 6.8.2. Định nghĩa kiểu dữ liệu con trỏ

- Có thể đặt tên cho kiểu dữ liệu con trỏ
- Để có thể khai báo biến con trỏ như các biến khác (giúp loại bỏ \* trong khai báo con trỏ)

Ví dụ: Định nghĩa một tên khác cho kiểu dữ liệu con trỏ

```
typedef int* IntPtr;
```

Au khi khai báo, các khai báo sau là tương đương:

```
IntPtr p;
```

```
int *p;
```

## 6.8. Toán tử sizeof

- Toán tử **sizeof** cho biết kích thước (byte) của một kiểu dữ liệu chuẩn (như **int**, **float**, ...) hay kiểu dữ liệu được định nghĩa trong chương trình (như **typedef**, **enum**, **struct**, **union**, ...).
- Đối tượng dữ liệu bao gồm tên biến, tên mảng, biến struct, ...
- Khai báo:     **sizeof (<đối tượng dữ liệu>)**  
                hoặc         **sizeof (<kiểu dữ liệu>)**
- Ví dụ: 

```
struct DiemThi    {    char    masv[8];  
                  char    mamh[5];  
                  int     lanthi;  
                  float   diem;  
                  } x;  
  
sizeof (int)      // cho trị 2  
sizeof (float)    // cho trị 4  
sizeof (x)        // cho trị 19
```

# 7. CON TRỎ VÀ MẢNG MỘT CHIỀU

## 7.1. Nhắc lại

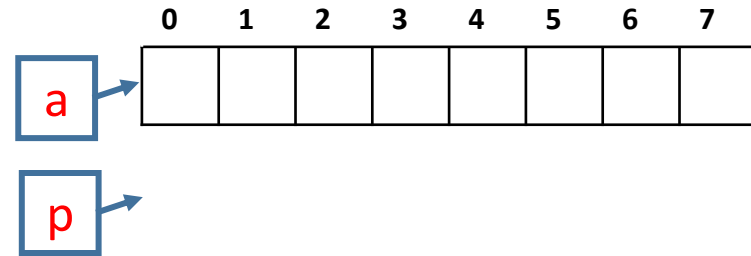
- Mảng lưu trong các ô nhớ liên tiếp trong bộ nhớ máy tính
- Biến mảng tham chiếu tới phần tử đầu tiên.
- Biến mảng là một con trỏ hằng.

- Ví dụ:

```
int a[8];
```

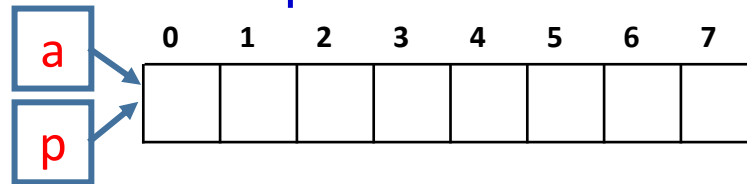
```
typedef int* IntPtr;
```

```
IntPtr p;
```

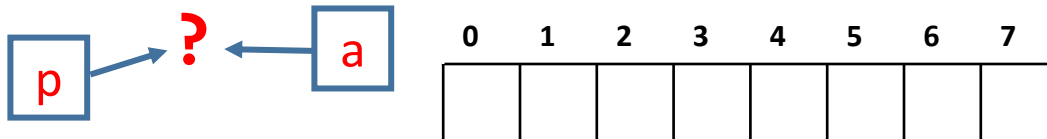


⇒ a và p cùng là các biến con trỏ.

- *Phép gán hợp lệ* **p = a;** /\* cho biết p sẽ trỏ tới nơi a trỏ, tức là tới phần tử đầu tiên của mảng a\*/



- *Phép gán không hợp lệ* **a = p;** /\* do con trỏ mảng là con trỏ hằng\*/





## 7.2. Mảng động

### - *Mảng tĩnh*

- Mảng lưu trong các ô nhớ liên tiếp trong bộ nhớ máy tính
- Hạn chế của mảng tĩnh: Bắt buộc phải biết trước cần bao nhiêu bộ nhớ lưu trữ  $\Rightarrow$  tốn bộ nhớ, không thay đổi được kích thước, ...

$\Rightarrow$  dùng **Mảng động**

### - *Mảng động*

- Kích thước không xác định ở thời điểm lập trình
- Mà xác định khi chạy chương trình

## 7. Con trỏ và mảng một chiều

### 7.3. Tạo mảng động bằng toán tử new

- Cấp phát động cho biến con trỏ
- Sau đó dùng con trỏ như mảng tĩnh
- Cú pháp:

`<type> <pointer> = new <type> [<number_of_elements>]`

Ví dụ:

```
typedef double * doublePtr;  
doublePtr d;  
d = new double[10];
```

⇒ Tạo biến mảng cấp phát động d có 10 phần tử, kiểu cơ sở là double.

## 7. Con trỏ và mảng một chiều

### 7.4. Xóa mảng động

- Dùng toán tử **delete[]** để xóa mảng động.

Ví dụ:

```
double *d = new double[10];  
//... Processing  
delete[] d;
```

⇒ Giải phóng tất cả vùng nhớ của mảng động này

⇒ Cặp ngoặc vuông báo hiệu có mảng

⇒ **Nhắc lại:** **d** vẫn trỏ tới vùng nhớ đó. Vì vậy sau khi  
**delete,** cần gán **d = NULL;**

# 7.5. Truy cập các phần tử mảng theo dạng con trỏ

Quy tắc:

Sử dụng		Tương đương với	
Cú pháp	Ví dụ	Cú pháp	Ví dụ
&<Tên mảng>[0]	&a[0]	<Tên mảng>	a
&<Tên mảng> [<Vị trí>]	&a[2]	<Tên mảng> + <Vị trí>	a+2
<Tên mảng>[<Vị trí>]	a[3]	*(<Tên mảng> + <Vị trí>)	*(a+3)

**p** 1024

STT phần tử	0	1	2	3	4	5	6	7	8	n-1
Sử dụng mảng	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	...	a[n-1]
Sử dụng con trỏ	p	p+1	p+2	p+3	p+4	p+5	p+6	p+7	...	p+n-1

## 7. Con trỏ và mảng một chiều

### 7.5. Truy cập các phần tử mảng theo dạng con trỏ

#### — Ví dụ

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

/* Nhập mảng bình thường*/
void NhapMang (int a[], int n)
{
    for(int i=0;i<n ;i++)
    {   printf("Nhap a[%d]: ",i);
        scanf("%d",&a[i]);
    }
}

void main()
{   int A[10], n;
    //nhập n
    ...
    NhapMang (A, n) ;
    ...
}
```

```
/* Nhập mảng theo dạng con trỏ*/
void NhapMangPointer (int *a, int n)
{
    for(int i=0;i<n ;i++)
    {   printf("Nhap a[%d]: ",i);
        scanf("%d",a+i);
    }
}

void main()
{   int A[10], n;
    //nhập n
    ...
    NhapMang (A, n) ;
    ...
}
```

## 7. Con trỏ và mảng một chiều

### 7.5. Truy cập từng phần tử đang được quản lý bởi con trỏ theo dạng mảng

#### — Ví dụ

```
#include <stdio.h>
#include <alloc.h>
#include <conio.h>
int main()
{ int *a, i;
  a = (int*)malloc (10);
  for(i=0;i<10;i++)
    a[i] = 2*i;
  printf("Truy cap theo kieu mang: ");
  for(i=0;i<10;i++)
    printf("%d ",a[i]);
  printf("\nTruy cap theo kieu con tro: ");
  for(i=0;i<10;i++)
    printf("%d ",*(a+i));
  getch();
  return 0;
}
```

## 7. Con trỏ và mảng một chiều

### Hàm trả về kiểu mảng

- Ta không được phép trả về kiểu mảng trong hàm.

Ví dụ:

```
int[] someFunction(); // Không hợp lệ!
```

- Có thể thay bằng trả về con trỏ tới mảng có cùng kiểu cơ sở:

```
int* someFunction(); // Hợp lệ!
```

### Thực hành sử dụng con trỏ

- i. Viết hàm nhận tham số là số nguyên  $n$ . Hàm tạo mảng 1 chiều có  $n$  phần tử với giá trị ngẫu nhiên từ 0 đến 9. Hàm trả về mảng vừa tạo.

```
int* Input(int n) { ... }
```

- ii. Viết hàm xuất mảng. Với tham số đầu vào là biến con trỏ kiểu `int p` và biến  $n$  có kiểu `int` cho biết số lượng phần tử có trong mảng.

```
void Output1(int *p, int n) { ... }
```

- iii. Viết hàm nhận tham số là con trỏ của mảng  $A$  chứa các số nguyên và số nguyên  $n$  ( $0 < n \leq 1000$ ) cho biết số lượng phần tử của mảng  $A$ . Biết rằng các giá trị trong mảng  $A$  nằm trong khoảng từ 0 đến 9. Hàm thực hiện tạo mảng 1 chiều  $B$  chứa tần suất xuất hiện của các giá trị trong mảng  $A$ . Hàm trả về mảng  $B$  vừa tạo.

```
int* TanSuat(int *p, int n) { ... }
```

- iv. Viết hàm xuất kết quả thực hiện của câu iii. Yêu cầu chỉ xuất những giá trị có tần suất  $> 0$ .

```
void Output2(int *p, int n) { ... }
```



## 7. Con trỏ và mảng một chiều

# Hàm trả về kiểu mảng

## Lời giải

```
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
void Output1(int *p, int n);
int* Input(int n);

int main()
{
    int *a, n;
    printf("Nhap n: ");
    scanf("%d", &n);
    a= Input(n);
    Output1(a,n);
    getch();
}
```

```
int* Input(int n)
{
    int *p;
    p = new int[n];
    for (int i = 0; i < n; i++)
        *(p+i) = rand() % 10;
        //tương đương với lệnh sau
        //p[i] = rand() % 10;
    return p;
}

void Output1(int *p, int n)
{
    .
    .
    .
}

}
```

## 7. Con trỏ và mảng một chiều

# Hàm trả về kiểu mảng

## Lời giải

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
void Output1(int *p, int n);
int* Input(int n);
int* TanSuat(int *a, int n);
void Output2(int *p, int n);

int main()
{
    int *a, n, *kq;
    printf("Nhap n: ");
    scanf("%d", &n);
    a = Input(n);
    Output1(a, n);
    kq=TanSuat(a, n);
    Output2(kq, n);
    delete[] a;
    delete[] kq;
    getch();
}
```

```
void Output2(int *p, int n)
{
    printf("\nTan suat xuat hien cua
                                     cac so:\n");
    for (int i = 0; i<10; i++)
        if (*(p+i)>0)
            printf("So %d xuat hien %d
                    lan\n", i, *(p + i));
}

int* TanSuat(int *a,int n)
{
    int *b;
    b = new int[10];
    for (int i = 0; i < 10; i++)
        b[i] = 0;
    for (int i = 0; i < n; i++)
        b[* (a+i)]++;
    // tuong duong voi lenh sau
    //*(b + a[i])+=1;
    return b;
}
```

## Hàm trả về kiểu mảng

### Lời giải

#### //Cách 2

```
#include <stdio.h>
#include <conio.h>
void Output(int *p, int n);
void Input2(int *&p, int n);
int main()
{   int *a, n;
    printf("Nhap n: ");
    scanf("%d", &n);
    Input2(a,n);
    Output(a,n);
    getch();
}
void Output(int *p, int n)
{   printf("\n Xuat mang 1 chieu: ");
    for (int i = 0; i < n; i++)
    {   printf("%5d", p[i]);
    }
}
```

#### //Cách 2

```
void Input2(int *&p, int n)
{
    p = new int[n];
    for (int i = 0; i < n; i++)
    {   printf("Nhap so thu %d: ", i + 1);
        scanf("%d", &p[i]);
    }
}
```

## 8. CON TRỎ VÀ MẢNG HAI CHIỀU

### 8.1. Sử dụng mảng con trỏ

- Nhìn mảng 2 chiều dưới cách nhìn “*mảng 2 chiều là mảng của mảng*”.
- Sử dụng định nghĩa kiểu con trỏ giúp hiểu rõ hơn:
- Ví dụ: cần tạo ra mảng 2 chiều động kích thước 3 x 4

```
typedef int* IntArrayPtr;
```

```
IntArrayPtr *m = new IntArrayPtr[3];
```

⇒ Tạo ra mảng gồm 3 phần tử, mỗi phần tử là 1 con trỏ (1 mảng)

⇒ Sau đó biến mỗi phần tử con trỏ thành mảng gồm 4 biến kiểu int

```
for (int i = 0; i < 3; i++)
```

```
    m[i] = new int[4];
```

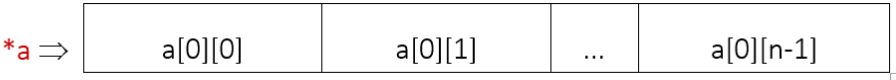
⇒ Kết quả là mảng động 3 x 4

8. Con trỏ và mảng hai chiều

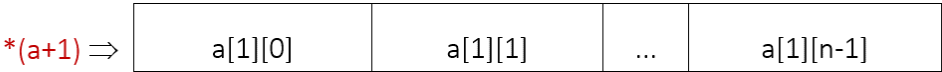
8.1. Sử dụng mảng con trỏ

Nhận xét:

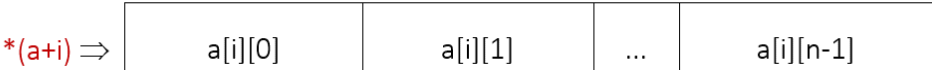
**\*a** quản lý các phần tử dòng **0**: **a[0][0], a[0][1], a[0][2],..., a[0][n-1]**



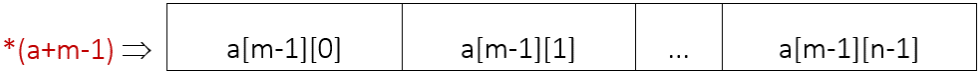
**\*(a+1)** quản lý các phần tử dòng **1**: **a[1][0], a[1][1], a[1][2],..., a[1][n-1]**



**\*(a+i)** quản lý các phần tử dòng **i**: **a[i][0], a[i][1], a[i][2],..., a[i][n-1]**



**\*(a+m-1)** quản lý các phần tử dòng **m-1** : **a[m-1][0], a[m-1][1], a[m-1][2],..., a[m-1][n-1]**




8. Con trỏ và mảng hai chiều

8.1. Sử dụng mảng con trỏ

Nhận xét: Xin cấp phát cho con trỏ **\*\*a** để quản lý mảng các con trỏ: **\*a, \*(a+1), \*(a+2)...**, **\*(a+m-1)**.

<b>**a</b>		0	1	2	...	n-1
<b>*(a)</b>	⇒	0	1	2	...	n-1
<b>*(a+1)</b>	⇒	N	N+1	N+2	...	2n-1
<b>*(a+2)</b>	⇒	2n	2n+1	2n+2	...	3n-1
<b>...</b>		...	...	...	...	...
<b>*(a+m-1)</b>	⇒	(m-1)n	(m-1)n+1	(m-1)n+2	...	(mn)-1

 Con trỏ kép là một loại con trỏ được dùng để chứa địa chỉ của một con trỏ khác.

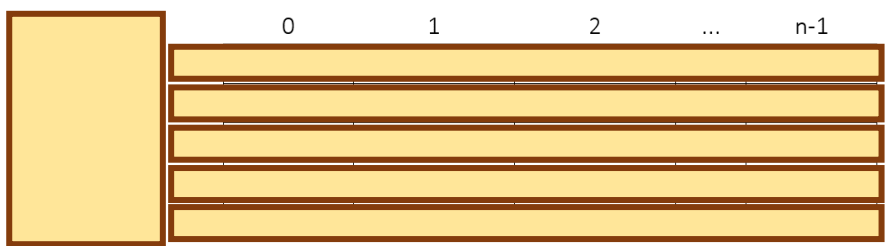
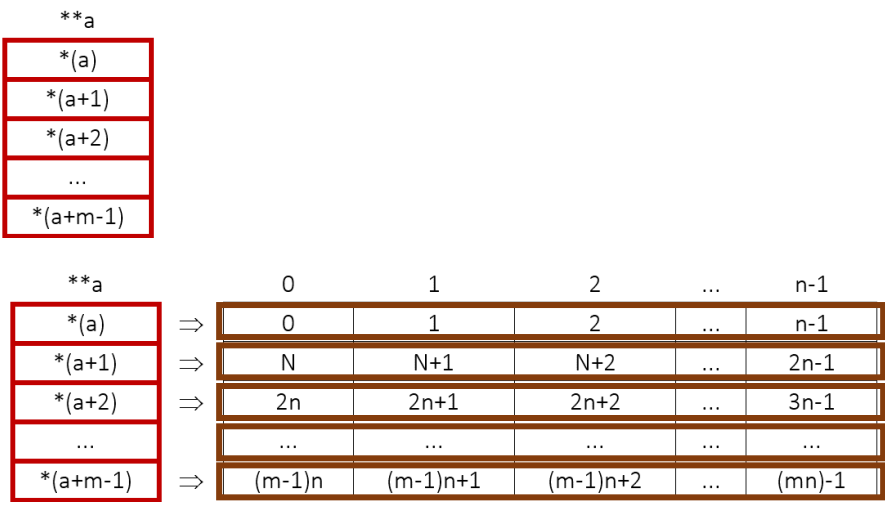
**Cú pháp khai báo:** <Kiểu> **\*\*** <Tên biến con trỏ>;

## 8. Con trỏ và mảng hai chiều

### 8.1. Sử dụng mảng con trỏ

- **Thực hành 1:** Tạo mảng 2 chiều bằng con trỏ.
- Mã lệnh

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
int main()
{
    int row = 2, col = 3;
    // Cấp phát vùng nhớ
    int **p = new int*[row];
    if (p == NULL)
        exit(1);
    for (int i = 0; i < row; i++)
    {
        p[i] = new int[col];
        if (p[i] == NULL)
            exit(1);
    }
    // Giải phóng vùng nhớ
    for (int i = 0; i < row; i++)
    {
        delete[] p[i];
    }
    delete[] p;
    return 0;
}
```



## 8. Con trỏ và mảng hai chiều

### 8.1. Sử dụng mảng con trỏ

#### *Thực hành 2:*

- i. Viết chương trình nhập vào một ma trận số nguyên gồm  $m$  dòng,  $n$  cột ( $m, n \leq 100$ ).
- ii. Xuất ra ma trận số nguyên vừa nhập.
- iii. Tính tổng các phần tử có trong ma trận.
- iv. Sắp xếp mảng 2 chiều tăng dần từ trái sang phải và từ trên xuống dưới.



## 8. Con trỏ và mảng hai chiều

### 8.1. Sử dụng mảng con trỏ

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
// hàm xin cấp phát bộ nhớ
void KhoiTao (int **a , int m ,int n)
{
    a = (int**) calloc (m, sizeof (int*));
    if(a==NULL)
    {
        printf("Khong du bo nho");
        getch();
        exit(1);
    }
    for(int i=0;i<m;i++)
    {
        a[i] = (int *) calloc (n, sizeof (int));
        if (a[i] == NULL)
        {
            printf(" ko du bo nho");
            getch();
            exit(1);
        }
    }
}
```

## 8. Con trỏ và mảng hai chiều

### 8.1. Sử dụng mảng con trỏ

// hàm nhập ma trận

```
void NhapMang (int** a , int m , int n)
{
    for (int i=0 ; i<m ; i++)
        for (int j=0 ; j<n ; j++)
        {
            printf(" nhap a[%d][%d]:", i , j);
            scanf (" %d ", (*(a+i) + j));
        }
}
```

// hàm xuất ma trận

```
void XuatMang (int ** a, int m , int n)
{
    for (int i=0 ; i<m ; i++)
    {
        for (int j=0 ; j<n ; j++)
            printf ("%4d", (*(a+i) +j));
        printf("\n");
    }
}
```

## 8. Con trỏ và mảng hai chiều

### 8.1. *Sử dụng mảng con trỏ*

// hàm tính tổng các phần tử trong ma trận

```
long TinhTong (int ** a, int m , int n)
```

```
{
```

```
    long s=0;
```

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

```
        for (int j=0 ; j<n ; j++)
```

```
            s=s+ *(*(a+i) +j) ;
```

```
    return s;
```

```
}
```

// hàm giải phóng bộ nhớ

```
void myfree (int **a , int m)
```

```
{
```

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

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

```
    free (a);
```

```
}
```

## 8. Con trỏ và mảng hai chiều

### 8.1. *Sử dụng mảng con trỏ*

// hàm sắp xếp giá trị các phần tử trong ma trận tăng dần

```
void SortTang(int ** a, int m, int n)
```

```
{
```

```
    int i,j,tam, sl=m*n-1;
```

```
    for (i=0; i<sl-1; i++)
```

```
        for (j=i+1; j<sl; j++)
```

```
            if ( *(a+i) > *(a+j) )
```

```
            {
```

```
                tam      = *(a+i) ;
```

```
                *(a+i)   = *(a+j) ;
```

```
                *(a+j)   = tam;
```

```
            }
```

```
}
```

## 8. Con trỏ và mảng hai chiều

### 8.1. *Sử dụng mảng con trỏ*

```
void main(int a[][COLS], int m, int n)
{
    int *a // khai báo con trỏ a để quản lý ma trận
    a = (int *) malloc (m*n); //xin cấp phát bộ nhớ cho ma trận
    int i,j,tam, sl=m*n-1;
    NhapDongCot(m,n);
    KhoiTaoMang(a,m,n);
    NhapMang(a,m,n);
    printf("Ma tran ban dau:\n");
    XuatMang(a,m,n);
    printf("Tong cac so co trong ma tran= %ld", TinhTong(a,m,n));
    SortTang(a,m,n);
    printf("Ma tran sau khi sort tang:\n");
    XuatMang(a,m,n);
    getch();
}
```

```
* (a + (row*n) + col);
```

## 8. Con trỏ và mảng hai chiều

### 8.2. Sử dụng con trỏ

#### *Thực hành 2: thực hiện lại qua việc sử dụng con trỏ*

- i. Viết chương trình nhập vào một ma trận số nguyên gồm  $m$  dòng,  $n$  cột ( $m, n \leq 100$ ).
- ii. Xuất ra ma trận số nguyên vừa nhập.
- iii. Tính tổng các phần tử có trong ma trận.
- iv. Sắp xếp mảng 2 chiều tăng dần từ trái sang phải và từ trên xuống dưới.

## 8. Con trỏ và mảng hai chiều

### 8.2. *Sử dụng con trỏ*

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
// hàm nhập dòng và cột
void NhapDongCot (int &m , int &n)
{
    do
    {
        printf (" nhap so dong: ");
        scanf(" %d ", &m);
        if (m<=0 || m >100)
            printf (" nhap sai, nhap lai ");
    } while(m<=0 || m>100);
    do
    {
        printf (" nhap so cot: ");
        scanf (" %d ", &n);
        if(n<=0 || n>100)
            printf (" nhap sai, nhap lai ") ;
    } while(n<=0 || n>100);
}
```



## 8. Con trỏ và mảng hai chiều

### 8.2. *Sử dụng con trỏ*

```
// hàm xin cấp phát bộ nhớ cho ma trận
void KhoiTaoMang (int *a , int m ,int n)
{
    a = (int*) calloc (m * n , sizeof (int));
    if(a==NULL)
    {
        printf("khong du bo nho");
        getch();
        exit(1);
    }
}

// hàm nhập ma trận
void NhapMang (int *a , int m , int n)
{
    for (int i=0 ; i<m ; i++)
        for (int j=0 ; j<n ; j++)
        {
            printf(" nhap a[%d][%d]:", i , j);
            scanf (" %d ", (a+ i*n + j));
        }
}
```

## 8. Con trỏ và mảng hai chiều

### 8.2. *Sử dụng con trỏ*

// hàm xuất ma trận

```
void XuatMang (int *a, int m , int n)
{
    for (int i=0 ; i<m ; i++)
    {
        for (int j=0 ; j<n ; j++)
            printf ("%4d", *(a+i *n +j));
        printf("\n");
    }
}
```

// hàm tính tổng các phần tử trong ma trận

```
long TinhTong (int *a, int m , int n)
{
    for (int i=0 ; i<m ; i++)
        for (int j=0 ; j<n ; j++)
            s = s+ *(a+i *n +j) ;
}
```

## 8. Con trỏ và mảng hai chiều

### 8.2. *Sử dụng con trỏ*

```
void SortTang(int a[][COLS], int m, int n)
{
    int i,j,tam, sl=m*n-1;
    int *p;
    p=&a[0][0]; // p chỉ vào phần tử đầu tiên của ma trận;
    for (i=0; i<sl-1; i++)
        for (j=i+1; j<sl; j++)
            if (* (p+i) > * (p+j) )
            {
                tam      = * (p+i) ;
                * (p+i)  = * (p+j) ;
                * (p+j)  = tam;
            }
}
```

## 9. CON TRỎ VỚI KIỂU DỮ LIỆU CÓ CẤU TRÚC (*Struct*)

### 9.1. Truy cập đến các thành phần của struct

Chọn một trong hai cách sau:

- *Cách 1*

- ❑ Cú pháp `<tên con trỏ> -> <tên thành phần>`

- ❑ Ví dụ: `printf("%f", p->diem);`

- *Cách 2*

- ❑ Cú pháp `(*<tên con trỏ>).<tên thành phần>`

- ❑ Ví dụ: `printf("%f ", (*p).diem);`

## 9. Con trỏ với kiểu dữ liệu có cấu trúc (Struct)

### 9.2. Ví dụ

Viết chương trình nhập vào điểm, và xuất kết quả ra màn hình.

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

struct DIEM
{
    char masv[8];
    char mamh[5];
    float diem;
};

void main()
{
    p=&x;
    struct DIEM *p, x; /* p là con trỏ kiểu struct và x là biến struct */
    p=&x;                /* con trỏ p chứa địa chỉ của biến x */
    printf("\nNhap ma so sinh vien :");
    gets(p->masv);
    printf("Nhap ma mon hoc : ");
    gets(p->mamh);
    printf("diem :");
    scanf("%f ", &tam);
    p->diem=tam;
    printf("\nKet qua:");
    printf("\nMa so sinh vien :%s", (*p).masv);
    printf("\nMa mon hoc : %s", (*p).mamh);
    printf("\ndiem :%.2f ", (*p).diem);
    getch();
}
```

## 9. Con trỏ với kiểu dữ liệu có cấu trúc (Struct)

### 9.2. Truyền structure sang hàm

Viết chương trình nhập vào điểm, và xuất kết quả ra màn hình.

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

```
#include <conio.h>
```

```
struct DIEM
```

```
{ char masv[8];
```

```
  char mamh[5];
```

```
  float diem;
```

```
};
```

```
void nhap(DIEMTHI *px);
```

```
void xuat(DIEMTHI x);
```

```
void main()
```

```
{  
    DIEMTHI x;  
    px=&x;  
    printf("\nNhap diem thi");  
    nhap(x);  
    printf("\nXuat diem thi");  
    xuat(x);  
    getch();  
}
```

```
void xuat(DIEMTHI x)
```

```
{  
    printf("\nMa sinh vien :%s", x.masv);  
    printf("\nMa mon hoc:%s", x.mamh);  
    printf("\nLan thi:%d", x.lanthi);  
    printf("\nDiem thi:%.2f", x.diem);  
}
```

```
void nhap(DIEMTHI *px)
```

```
{  
    float tam;  
    printf("\nMa sinh vien :");  
    gets(px->masv);  
    printf("Ma mon hoc:");  
    gets(px->mamh);  
    printf("Lan thi:");  
    scanf("%d%c", &px->lanthi);  
    printf("Diem thi:");  
    scanf("%f%c", &tam);  
    px->diem=tam;  
}
```

## 9. Con trỏ với kiểu dữ liệu có cấu trúc (Struct)

### 9.3. Cấu trúc đệ quy (tự trỏ)

- Ngôn ngữ lập trình C cho phép khai báo dạng Cấu trúc đệ quy (tự trỏ)

```
struct PERSON
{
    char hoten[30];
    struct PERSON *father, *mother;
};
```

```
struct NODE
{
    int value;
    struct NODE *pNext;
};
```

## 10. CÂU HỎI ÔN TẬP

**Câu 1:** Tại sao cần phải giải phóng khối nhớ được cấp phát động?

⇒ **Khối nhớ không tự giải phóng sau khi sử dụng** nên sẽ làm giảm tốc độ thực hiện chương trình hoặc tràn bộ nhớ nếu tiếp tục cấp phát

**Câu 2:** Điều gì xảy ra nếu ta nối thêm một số ký tự vào một chuỗi (được cấp phát động trước đó) mà không cấp phát lại bộ nhớ cho nó?

⇒ Nếu chuỗi đủ lớn để chứa thêm thông tin thì không cần cấp phát lại. Ngược lại phải cấp phát lại để có thêm vùng nhớ.



## 10. Câu hỏi ôn tập

*Câu 3:* Ta thường dùng phép ép kiểu trong những trường hợp nào?

⇒ Lấy phần nguyên của số thực hoặc lấy phần thực của phép chia hai số nguyên, ...

– *Câu 4:* Giả sử **c** kiểu **char**, **i** kiểu **int**, **l** kiểu **long**. Hãy xác định kiểu của các biểu thức sau:

- $(c + i + l)$
- $(i + 'A')$
- $(i + 32.0)$
- $(100 + 1.0)$

## 10. Câu hỏi ôn tập

*Câu 5:* Việc cấp phát động nghĩa là gì?

⇒ Bộ nhớ được cấp phát động là bộ nhớ được cấp phát trong khi chạy chương trình và có thể thay đổi độ lớn vùng nhớ.

*Câu 6:* Cho biết sự khác nhau giữa **malloc** và **calloc**?

⇒ **malloc**: cấp phát bộ nhớ cho một đối tượng.

⇒ **calloc**: cấp phát bộ nhớ cho một nhóm đối tượng.

## 10. Câu hỏi ôn tập

*Câu 7:* Viết câu lệnh sử dụng hàm **malloc** để cấp phát **1000** số kiểu **long**.

```
long *ptr;  
ptr = (long *)malloc(1000 * sizeof(long));
```

– *Câu 8:* Giống bài 7 nhưng dùng **calloc**

```
long *ptr;  
ptr = (long *)calloc(1000, sizeof(long));  
ptr = (long *)calloc(sizeof(long), 1000); !!!
```

## 10. Câu hỏi ôn tập

*Câu 9:* Cho biết kết quả

```
void func()  
{   int n1 = 100, n2 = 3;  
    float ketqua = n1 / n2;  
    printf("%d / %d = %f", n1, n2, ketqua);  
}
```

– *Câu 10:* Tìm lỗi

```
int main()  
{   void *p;  
    p = (float *)malloc(sizeof(float));  
    *p = 1.23;  
}
```

## 10. Câu hỏi ôn tập

*Câu 11:* Cho biết kết quả chương trình sau. Giải thích tại sao ta có được kết quả này.

```
void hamf(int*a)
```

```
{  a = new int[5];  
    for (int i = 0; i<5; i++)  
        a[i] = i + 1;  
}
```

```
void main()
```

```
{  int n = 5;  
    int *a = &n;  
    printf ("Giá trị *a = %d",*a);  
    hamf(a);  
    printf("Giá trị *a = %d", *a);  
}
```

## 10. Câu hỏi ôn tập

*Câu 12:* Cho biết kết quả chương trình sau. Giải thích tại sao ta có được kết quả này.

```
void hamf(int*&a)
{
    a = new int[5];
    for (int i = 0; i<5; i++)
        a[i] = i + 1;
}

void main()
{
    int n = 5;
    int *a = &n;
    printf("Gia tri *a = ", *a);
    hamf(a);
    printf("Gia tri *a = ", *a);
}
```

## 11. BÀI TẬP

1. Viết chương trình nhập một dãy số hữu tỉ tùy ý (sử dụng con trỏ và sự cấp phát động), xuất ra dãy gồm tất cả các số nhỏ hơn 1 có trong dãy được nhập vào, tính tổng và tích của dãy số hữu tỉ.
2. Viết chương trình khai báo mảng hai chiều có  $12 \times 12$  phần tử kiểu char. Gán ký tự 'X' cho mọi phần tử của mảng này. Sử dụng con trỏ đến mảng để in giá trị các phần tử mảng lên màn hình ở dạng lưới.

## 11. BÀI TẬP

3. Viết chương trình khai báo mảng 10 con trỏ đến kiểu float, nhận 10 số thực từ bàn phím, sắp xếp lại và in ra màn hình dãy số đã sắp xếp.
4. Chương trình cho phép người dùng nhập các dòng văn bản từ bàn phím đến khi nhập một dòng trống. Chương trình sẽ sắp xếp các dòng theo thứ tự alphabet rồi hiển thị chúng ra màn hình.



## 12. VẤN ĐỀ MỞ RỘNG

- i. Các thao tác trên khối nhớ
- ii. Tham khảo cấp phát động bằng hàm `malloc`

## 12. Vấn đề mở rộng

### 12.1. Thao tác trên các khối nhớ

- Thuộc thư viện `<string.h>`
  - *memset*: gán giá trị cho tất cả các byte nhớ trong khối.
  - *memcpy*: sao chép khối.
  - *memmove*: di chuyển thông tin từ khối này sang khối khác.

## 12. Vấn đề mở rộng

### 12.1. Thao tác trên các khối nhớ

```
void *memset(void *dest, int c, size_t count)
```

Gán **count** (bytes) đầu tiên của vùng nhớ mà **dest** trỏ tới bằng giá trị **c** (từ 0 đến 255)

Thường dùng cho vùng nhớ kiểu char còn vùng nhớ kiểu khác thường đặt giá trị zero.

Trả về: Con trỏ **dest**.

```
char str[] = "Hello world";  
printf("Truoc khi memset: %s\n", str);  
memset(str, '*', strlen(str));  
printf("Sau khi memset: %s\n", str);
```

Truoc khi memset: Hello world

Sau khi memset: \*\*\*\*\*

## 12. Vấn đề mở rộng

### 12.1. Thao tác trên các khối nhớ

```
void *memcpy(void *dest, void *src, size_t count)
```

Sao chép chính xác **count** byte từ khối nhớ **src** vào khối nhớ **dest**.

Nếu hai khối nhớ đè lên nhau, hàm sẽ làm việc không chính xác.

Trả về: Con trỏ **dest**.

```
char src[] = "*****";  
char dest[] = "0123456789";  
memcpy(dest, src, 5);  
memcpy(dest + 3, dest + 2, 5);  
printf("dest: %s\n", dest);
```

dest: \*\*\*\*\*5689

## 12. Vấn đề mở rộng

### 12.1. Thao tác trên các khối nhớ

```
void *memmove(void *dest, void *src, size_t count)
```

Sao chép chính xác **count** byte từ khối nhớ **src** vào khối nhớ **dest**.

Nếu hai khối nhớ đè lên nhau, hàm vẫn thực hiện chính xác.

Trả về: Con trỏ **dest**.

```
char src[] = "*****";  
char dest[] = "0123456789";  
memmove(dest, src, 5);  
memmove(dest + 3, dest + 2, 5);  
printf("dest: %s\n", dest);
```

Kết quả đoạn code: dest: \*\*\*\*\*5689

## 12. Vấn đề mở rộng

### 12.2. Cấp phát động bằng hàm malloc trong C

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

Cấp phát trong HEAP một vùng nhớ **size** (bytes)  
**size\_t** thay cho unsigned (trong **<stddef.h>**)

Trả về:

- **Thành công**: Con trỏ đến vùng nhớ mới được cấp phát.
- **Thất bại**: **NULL** (không đủ bộ nhớ).

```
int *p = (int *)malloc(sizeof(int));  
// int *p = new int;
```

```
int *p = (int *)malloc(10 * sizeof(int));  
// int *p = new int[10];  
if (p == NULL) printf("Khong du bo nho!");
```

## 12. Vấn đề mở rộng

### 12.2. Cấp phát động bằng hàm malloc trong C

```
void *calloc(size_t num, size_t size)
```

Cấp phát vùng nhớ gồm **num** phần tử trong HEAP, mỗi phần tử kích thước **size** (bytes)

Trả về:

- Thành công: Con trỏ đến vùng nhớ mới được cấp phát.
- Thất bại: **NULL** (không đủ bộ nhớ).

```
int *p = (int *)calloc(10, sizeof(int));
```

```
if (p == NULL)
```

```
    printf("Khong du bo nho!");
```

## 12. Vấn đề mở rộng

### 12.2. Cấp phát động bằng hàm malloc trong C

```
void *realloc(void *block, size_t size)
```

Cấp phát lại vùng nhớ có kích thước **size** do **block** trỏ đến trong vùng nhớ HEAP.

**block == NULL** → sử dụng **malloc**

**size == 0** → sử dụng **free**

Trả về:

- **Thành công**: Con trỏ đến vùng nhớ mới được cấp phát.
- **Thất bại**: **NULL** (không đủ bộ nhớ).

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

```
p = (int *)realloc(p, 20 * sizeof(int));
```

```
if (p == NULL)
```

```
    printf("Khong du bo nho!");
```



## 12. Vấn đề mở rộng

### 12.2. Cấp phát động bằng hàm malloc trong C

```
void free(void *ptr)
```

Giải phóng vùng nhớ do **ptr** trỏ đến, được cấp bởi các hàm malloc(), calloc(), realloc().  
Nếu **ptr** là NULL thì không làm gì cả.

Trả về: Không có.

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

```
free(p);
```

*// Tương đương:*

```
int *p = new int[10];
```

```
delete []p;
```

## 12. Vấn đề mở rộng

### 12.3. Bài tập Tạo mảng 2 chiều bằng con trỏ.

#### Lời giải (sử dụng hàm malloc)

```
int main()
{
    int m = 4, n = 4;
    int kt;
    int **a = (int **)malloc(m * sizeof(int *));
    if (a != NULL) /* kiểm tra sự cấp phát thành công */
    {
        kt = 0;
        for (int i = 0; i < m; i++)
        {
            if (kt == 1)
                break;
            a[i] = (int *) malloc(n*sizeof(int));
            if (a[i] == NULL)
                kt = 1;
        }
    }
}
```

## 12. Vấn đề mở rộng

### 12.3. Bài tập Tạo mảng 2 chiều bằng con trỏ.

Lời giải (sử dụng hàm malloc)

```
if (kt == 0)
{
    /* sử dụng được a[i][j] */
    /* Giải phóng vùng nhớ được cấp phát bằng malloc*/
    for (int i = 0; i < m; i++)
        if (a[i] != NULL)
            free(a[i]);
    free(a);
}
```

## 12. Vấn đề mở rộng

### 12.4. Lưu ý

- **Không cần** kiểm tra con trỏ có **NULL** hay không trước khi **free** hoặc **delete**.
- Cấp phát bằng **malloc**, **calloc** hay **realloc** thì giải phóng bằng **free**, cấp phát bằng **new** thì giải phóng bằng **delete**.
- Cấp phát bằng **new** thì giải phóng bằng **delete**, cấp phát mảng bằng **new[]** thì giải phóng bằng **delete[]**.

## 12. Vấn đề mở rộng

### 12.5. Câu hỏi ôn tập

**Câu 1:** Ưu điểm của việc sử dụng các hàm thao tác khối nhớ? Ta có thể sử dụng một vòng lặp kết hợp với một câu lệnh gán để khởi tạo hay sao chép các byte nhớ hay không?

⇒ Việc sử dụng các hàm thao tác khối nhớ như **memset**, **memcpy**, **memmove** giúp khởi tạo hay sao chép/di chuyển vùng nhớ nhanh hơn.

⇒ Trong một số trường hợp chỉ có thể sử dụng vòng lặp kết hợp với lệnh gán để khởi tạo nếu như các byte nhớ cần khởi tạo khác giá trị.

## 12. Vấn đề mở rộng

### 12.5. Câu hỏi ôn tập

**Câu 2:** Cho biết sự khác nhau giữa memcpy và memmove

⇒ Hàm memmove cho phép sao chép hai vùng nhớ **chồng lên nhau** trong khi hàm memcpy làm việc không chính xác trong trường hợp này

**Câu 3:** Trình bày 2 cách khởi tạo mảng float **data[1000];** với giá trị **0**.

⇒ C1: `for (int i=0; i<1000; i++) data[i] = 0;`

⇒ C2: `memset(data, 0, 1000*sizeof(float));`

