



Chương 07 CON TRỎ

Lê Thành Sách

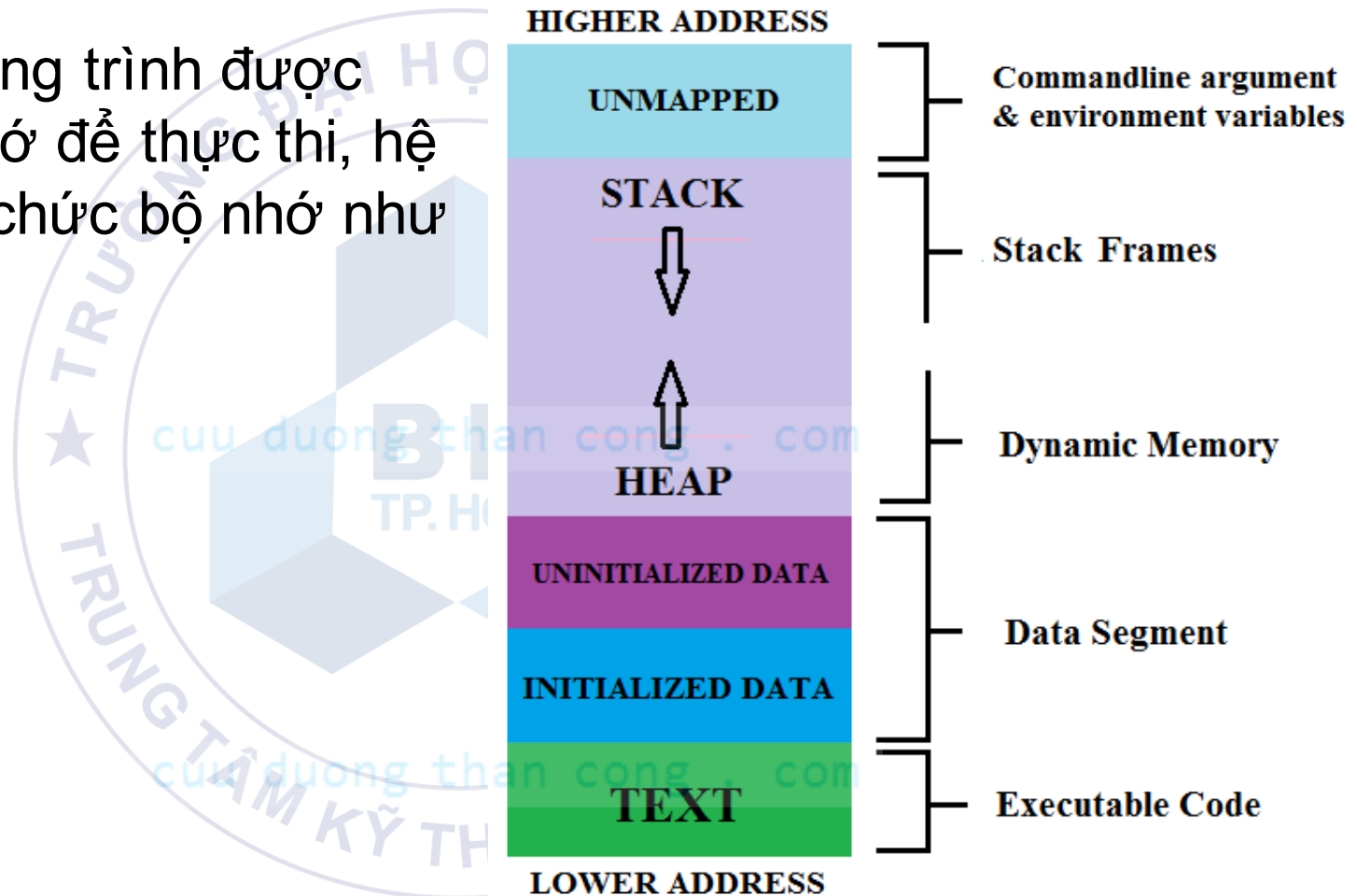
Nội dung

- Tổ chức bộ nhớ thực thi
- Ứng dụng của con trỏ
- Mô hình của con trỏ
- Toán tử &
- Khai báo trỏ
- Toán tử *
- Các phép toán trên con trỏ
- Con trỏ và mảng
- Cấp phát bộ nhớ động
- Con trỏ và cấu trúc, toán tử ->
- Các chủ đề nâng cao với con trỏ
 - Thứ tự đánh giá * và ++, --
 - Con trỏ và const
 - Con trỏ đến con trỏ
 - Con trỏ void



Tổ chức bộ nhớ thực thi

- Khi chương trình được lên bộ nhớ để thực thi, hệ thống tổ chức bộ nhớ như hình vẽ

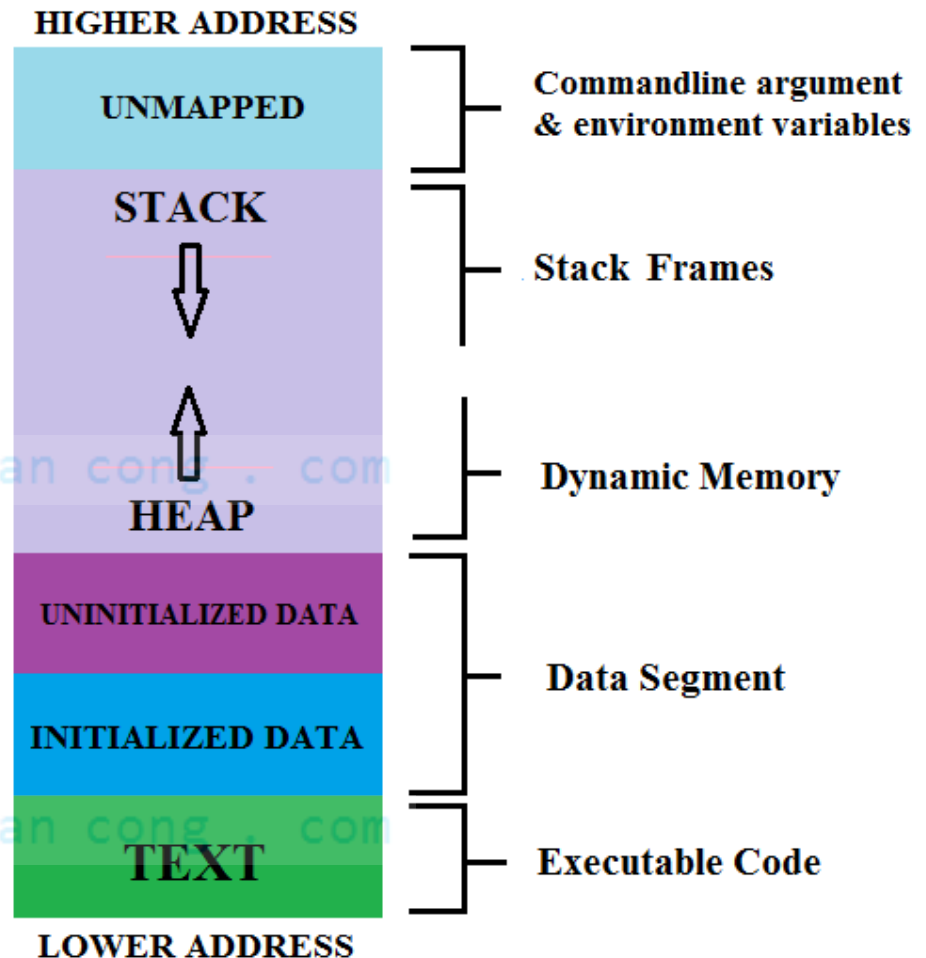


(Nguồn: <http://proprogramming.org/>)

Tổ chức bộ nhớ thực thi

■ Vùng “text”

- Chứa mã thực thi của chương trình
- Vùng này chỉ đọc
- Vùng này có thể dùng chung trong trường hợp chương trình thực thi thường xuyên



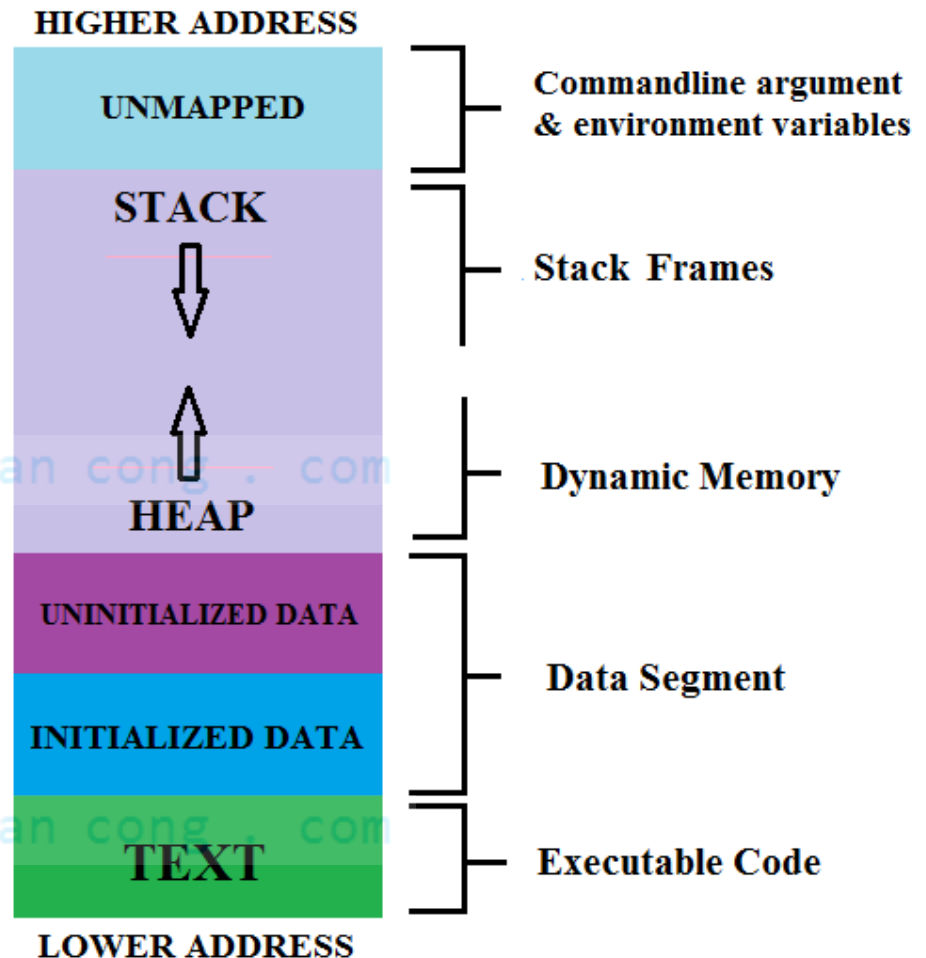
(Nguồn: <http://proprogramming.org/>)

Tổ chức bộ nhớ thực thi

■ Vùng “Data”

■ Gồm:

- Dữ liệu được khởi động (bởi người lập trình)
- Dữ liệu không được khởi động (bởi người lập trình)



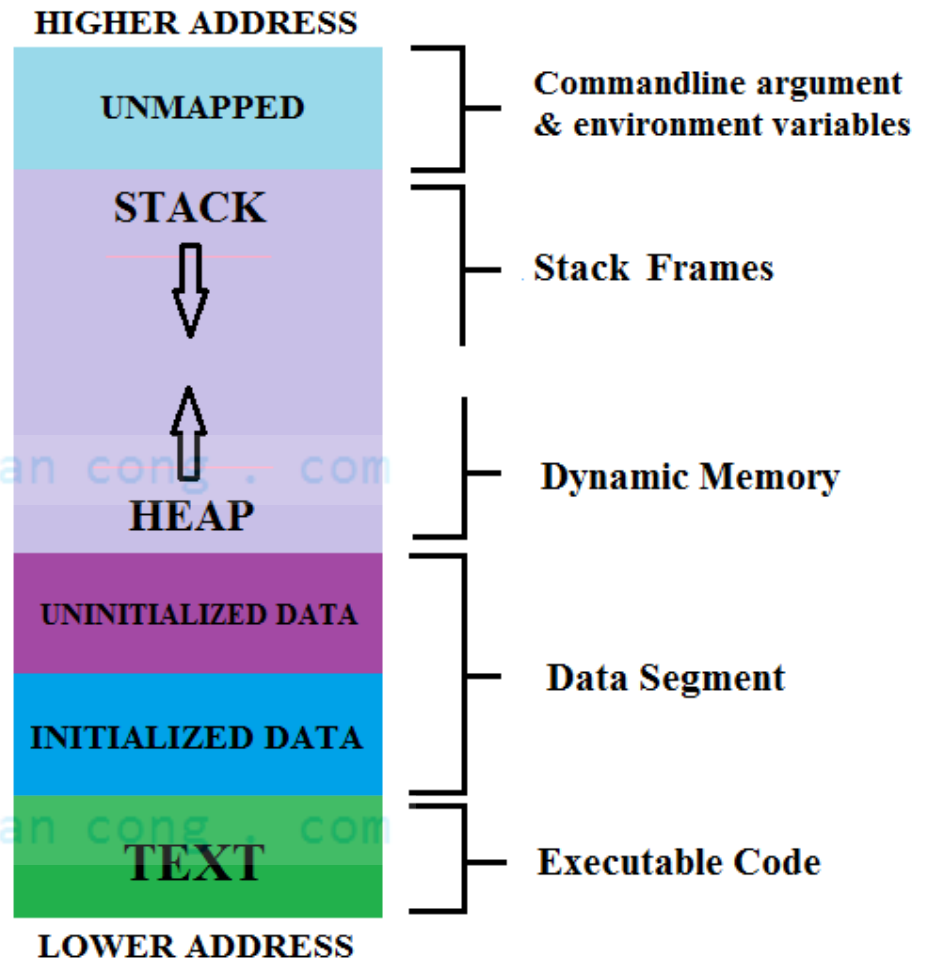
(Nguồn: <http://proprogramming.org/>)

Tổ chức bộ nhớ thực thi

■ Vùng “Data”

■ Gồm:

- Dữ liệu được khởi động (bởi người lập trình)
 - Biến toàn cục
 - Biến tĩnh (static)
- Vùng này gồm hai vùng con:
 - Chỉ đọc
 - Ví dụ: Hằng chuỗi
 - Đọc/ghi
 - Các biến static và global không hằng



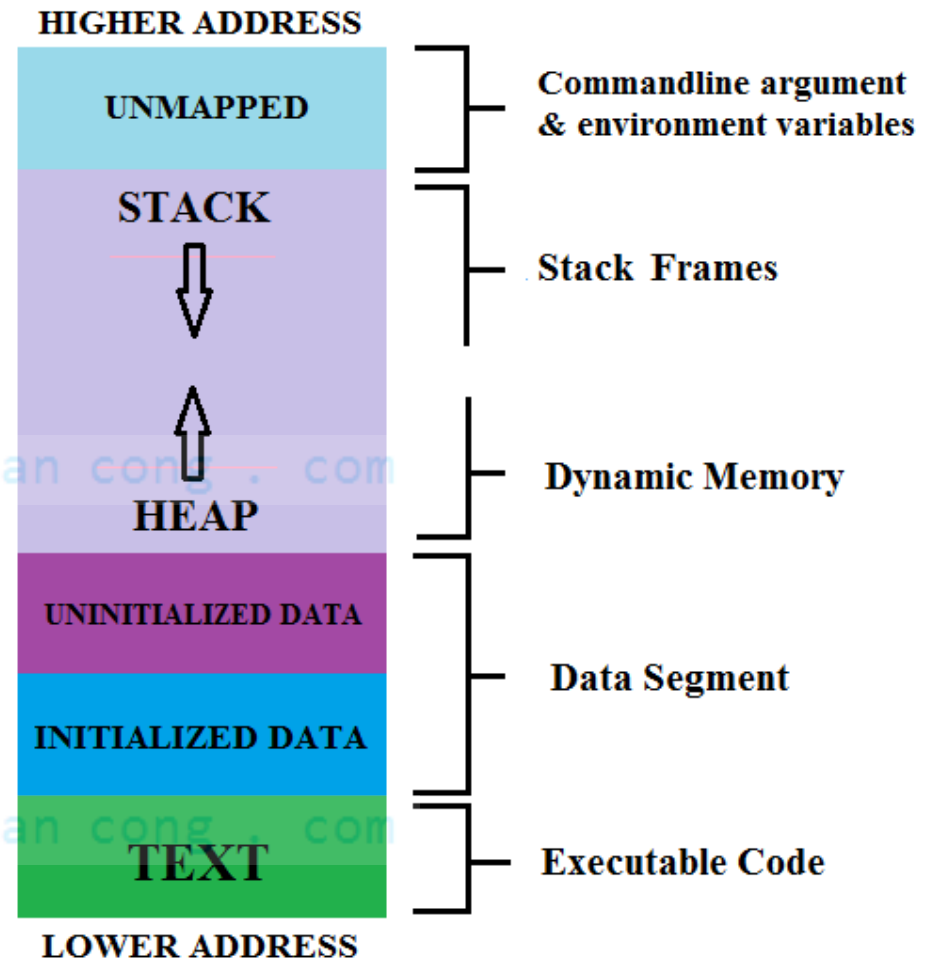
(Nguồn: <http://proprogramming.org/>)

Tổ chức bộ nhớ thực thi

■ Vùng “Data”

■ Gồm:

- Dữ liệu được khởi động
- Dữ liệu không khởi động bởi người lập trình
 - Biến toàn cục
 - Biến tĩnh (static)
- Hệ thống khởi động về 0 (số) cho các biến không được người lập trình chủ động khởi động

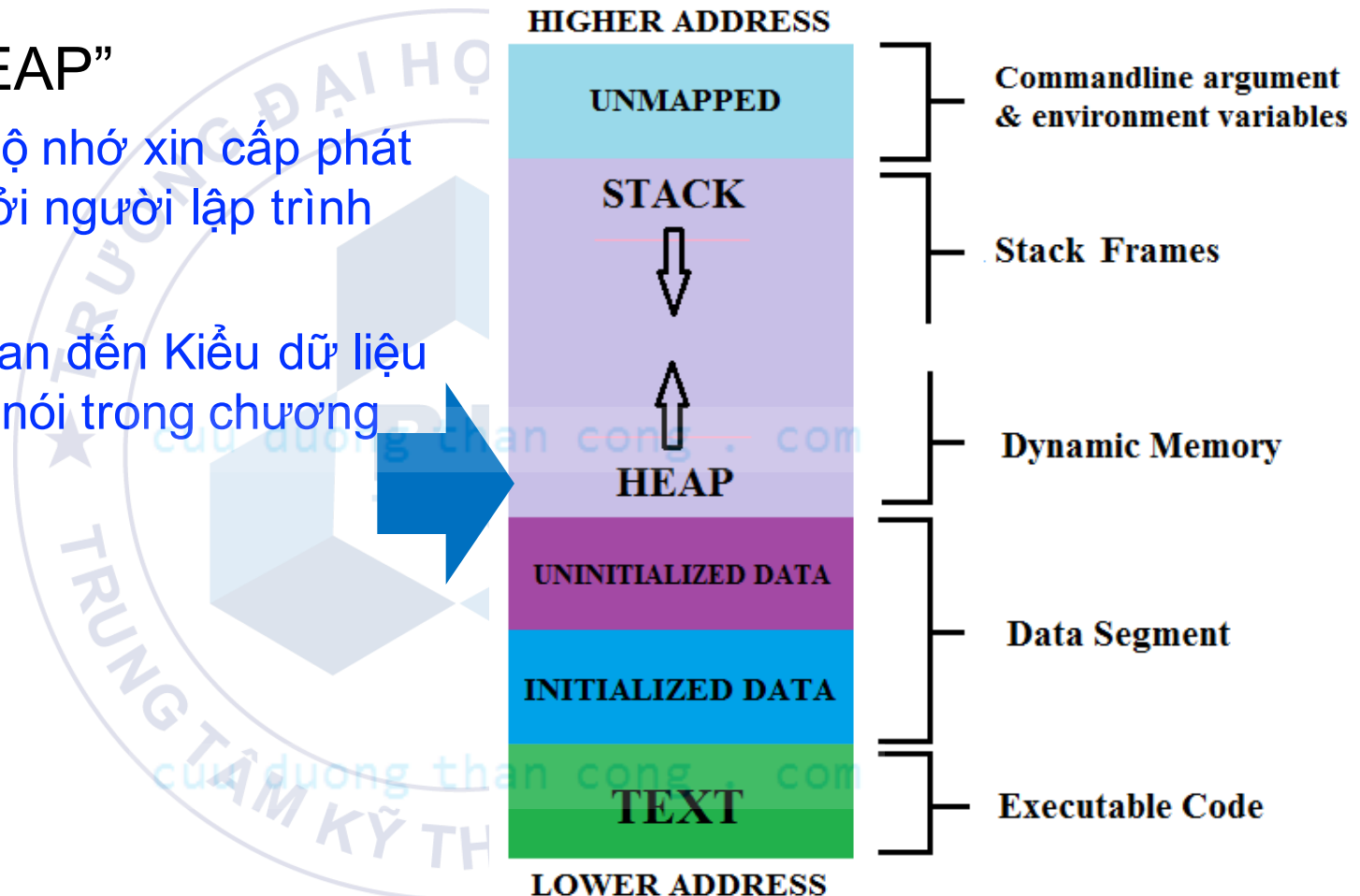


(Nguồn: <http://proprogramming.org/>)

Tổ chức bộ nhớ thực thi

■ Vùng “HEAP”

- Chứa bộ nhớ xin cấp phát động bởi người lập trình
- Liên quan đến Kiểu dữ liệu con trỏ nói trong chương này



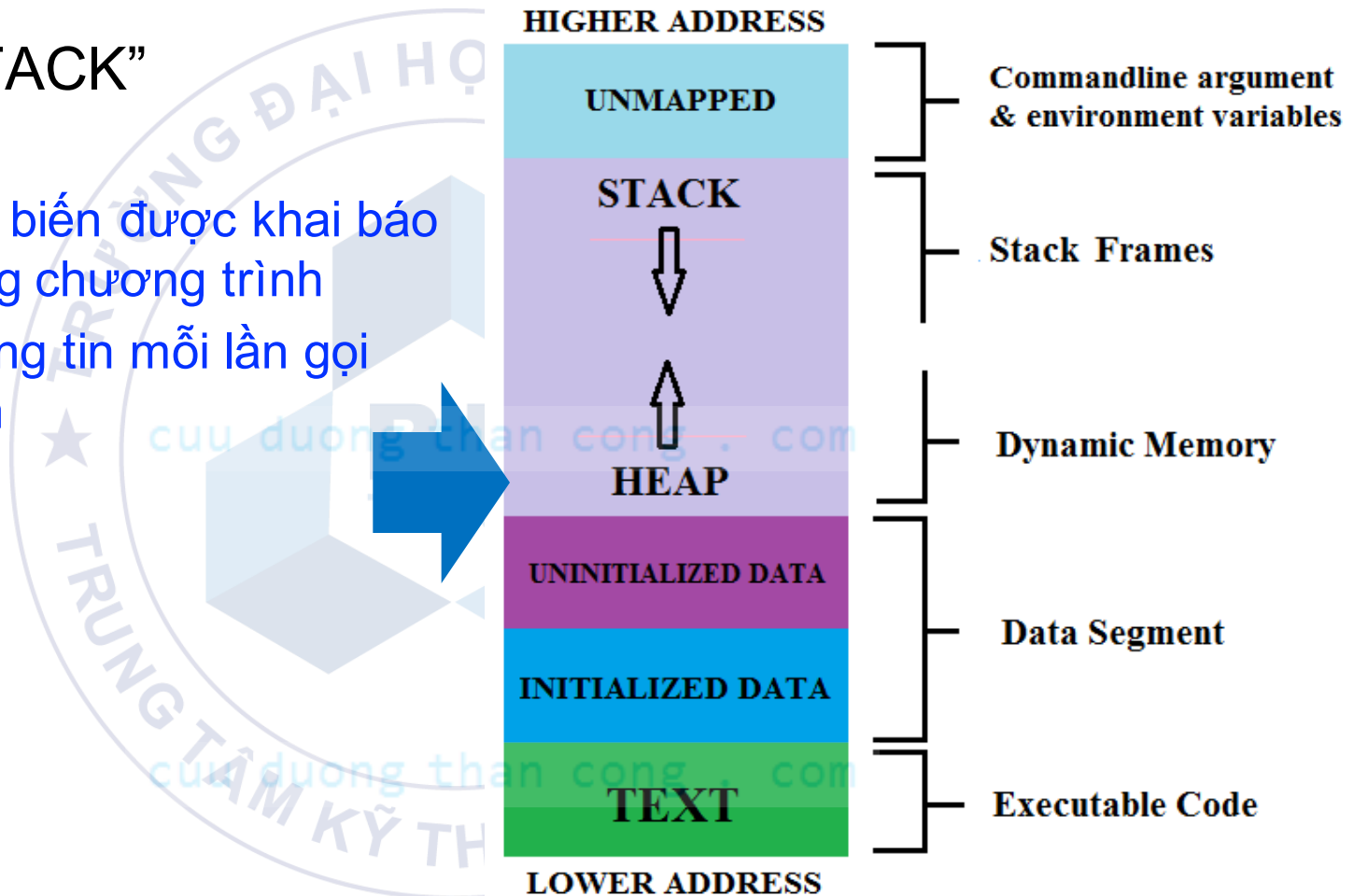
(Nguồn: <http://proplearning.org/>)

Tổ chức bộ nhớ thực thi

■ Vùng “STACK”

■ Chứa

- Các biến được khai báo trong chương trình
- Thông tin mỗi lần gọi hàm



(Nguồn: <http://phoptelegraph.org/>)

Ứng dụng của con trỏ

■ Mảng trong C

- Phải biết trước số lượng phần tử tại thời điểm viết chương trình
- Do đó, cần phải khai báo một số lượng lớn các ô nhớ để sẵn. Tuy nhiên, tại một thời điểm nào đó, chương trình có thể sẽ sử dụng ít hơn rất nhiều → lãng phí
- Yêu cầu: Có thể nào dùng mảng với số lượng phần tử chỉ cần biết lúc chương trình đang chạy?
- => Cần con trỏ

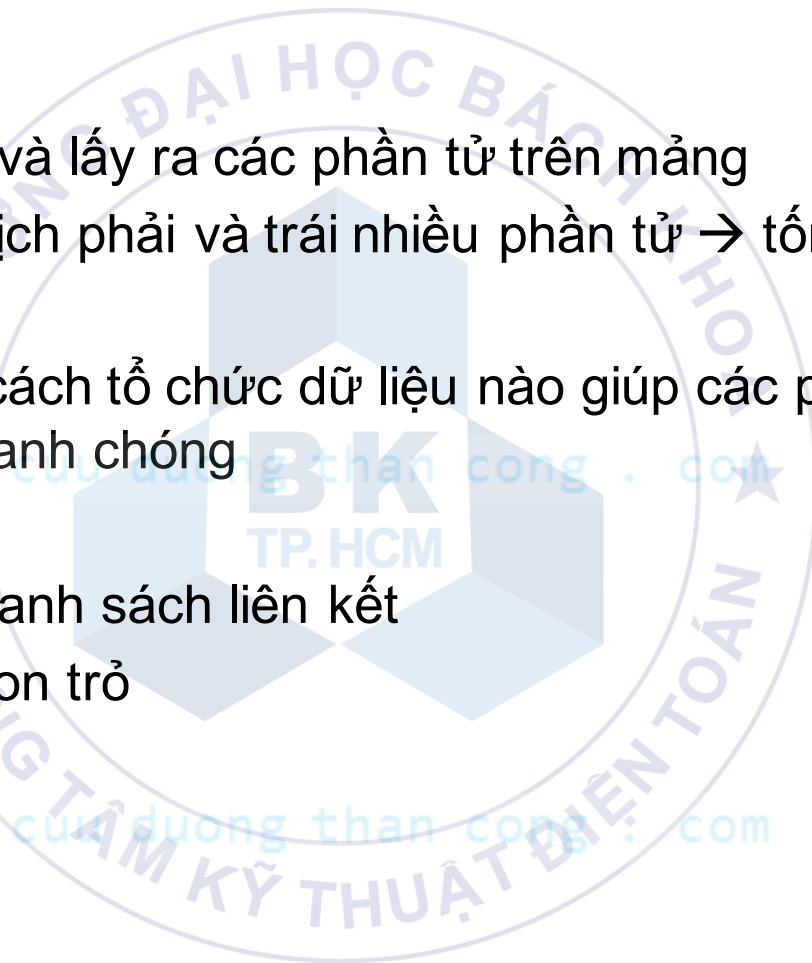
cuduongthanhcong.com

cuduongthanhcong.com

Ứng dụng của con trỏ

■ Mảng trong C

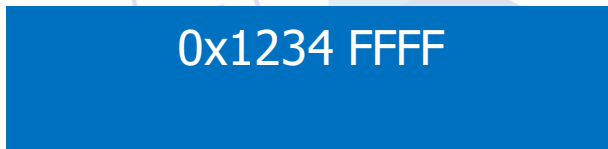
- Khi thêm vào và lấy ra các phần tử trên mảng
- => cần phải dịch phải và trái nhiều phần tử → tốn nhiều thời gian
- Yêu cầu: Có cách tổ chức dữ liệu nào giúp các phép quản lý phần tử nói trên nhanh chóng
- => Sử dụng danh sách liên kết
- => Cần đến con trỏ



Mô hình của con trỏ



Biến a: là biến có kiểu nào đó bất kỳ
Ô nhớ bắt đầu của a có địa chỉ là: (ví dụ)
0x1234 FFFF

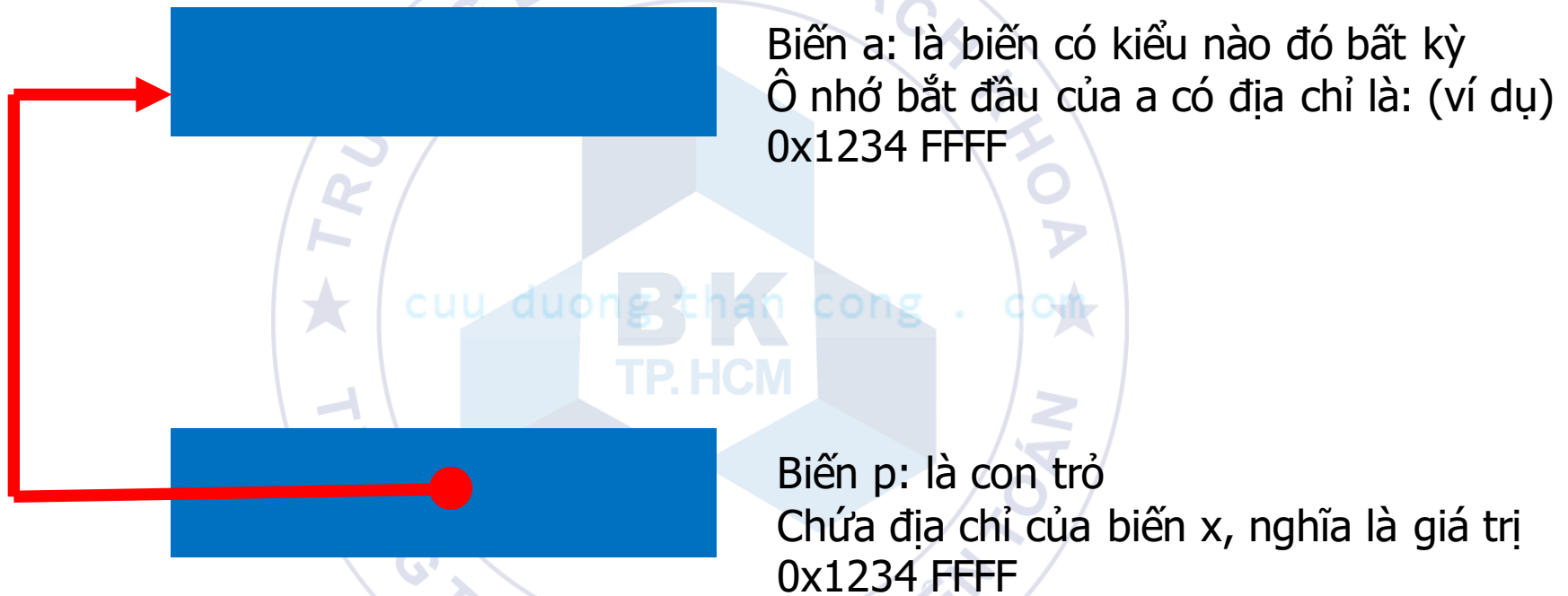


Biến p: là con trỏ
Chứa địa chỉ của biến x, nghĩa là giá trị
0x1234 FFFF

TRƯỜNG ĐẠI HỌC BÁCH KHOA
BKT
TP. HCM
cuuduongthancong.com

TRUNG TÂM KỸ THUẬT ĐIỆN TOÁN
cuuduongthancong.com

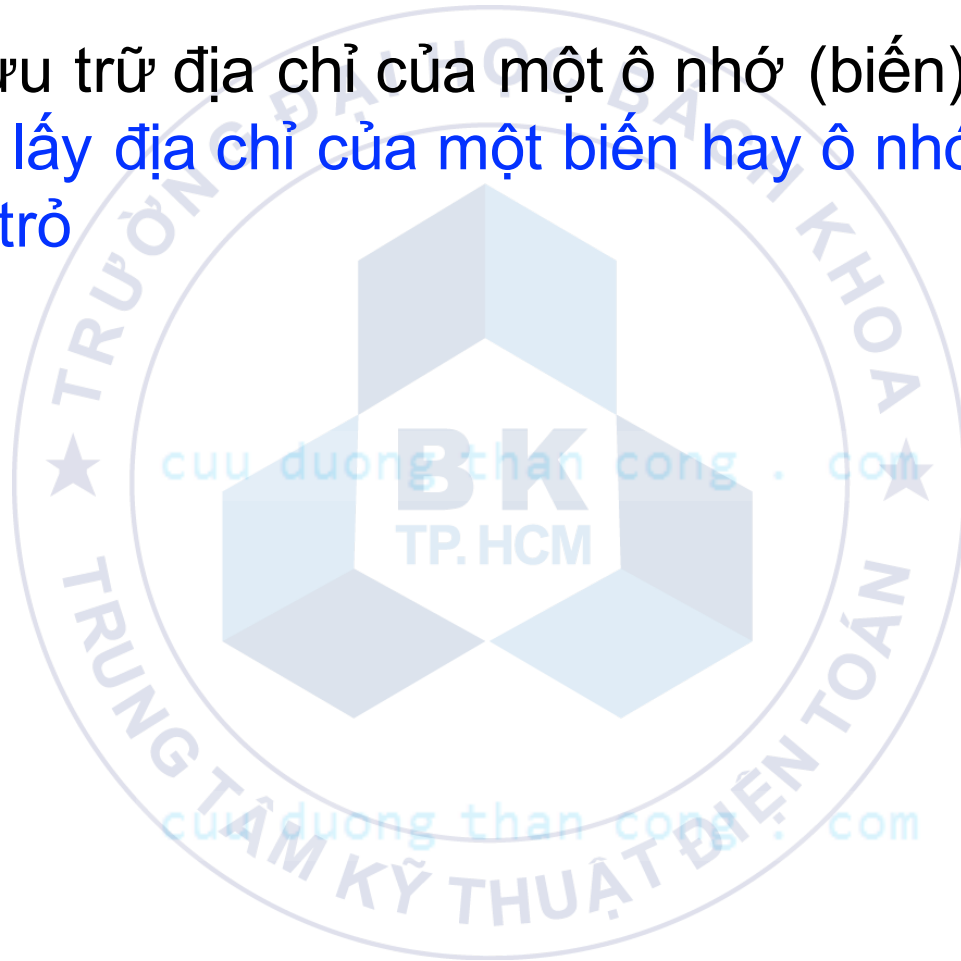
Mô hình của con trỏ



Minh họa con trỏ bởi tên từ ô nhớ biến p
CHỈ ĐẾN (point to) ô nhớ biến x

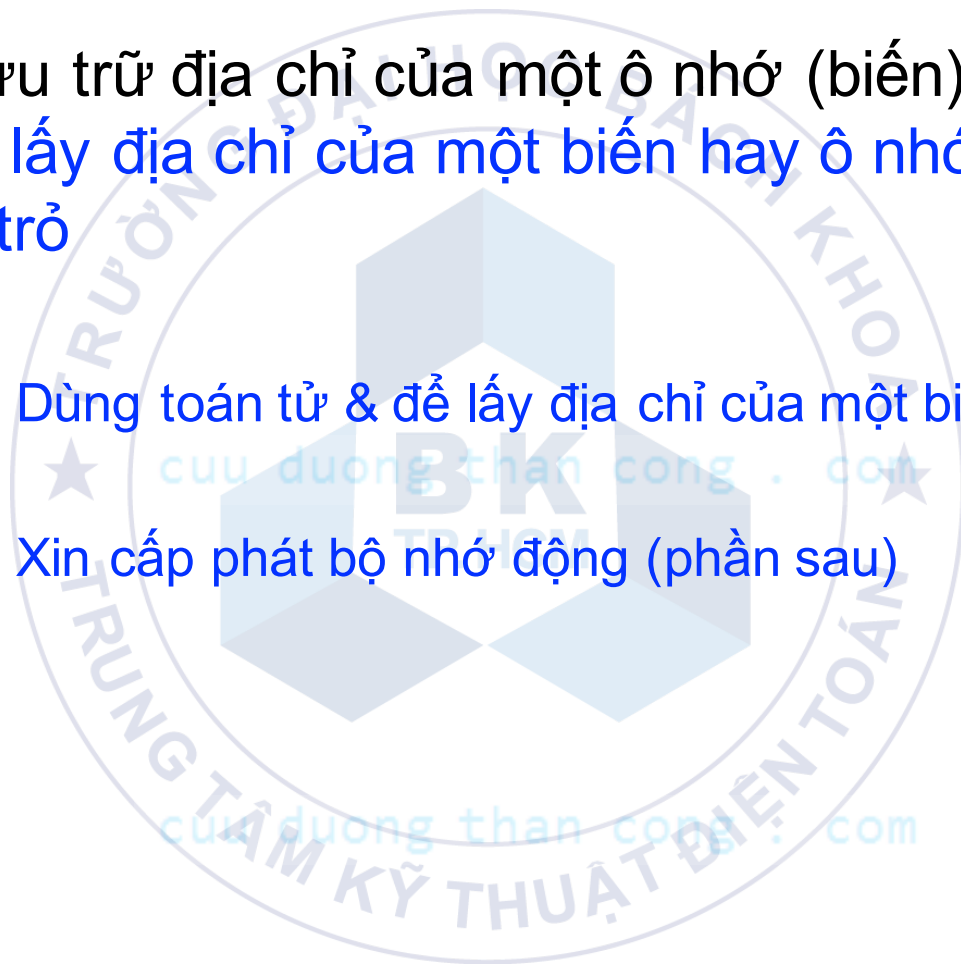
Toán tử &

- Con trỏ lưu trữ địa chỉ của một ô nhớ (biến) khác → Bằng cách nào lấy địa chỉ của một biến hay ô nhớ để gán cho biến con trỏ



Toán tử &

- Con trỏ lưu trữ địa chỉ của một ô nhớ (biến) khác → Bằng cách nào lấy địa chỉ của một biến hay ô nhớ để gán cho biến con trỏ
 - Cách 1: Dùng toán tử & để lấy địa chỉ của một biến đang có
 - Cách 2: Xin cấp phát bộ nhớ động (phần sau)



Toán tử &

- Toán tử & trả về địa chỉ của một biến
- Ví dụ

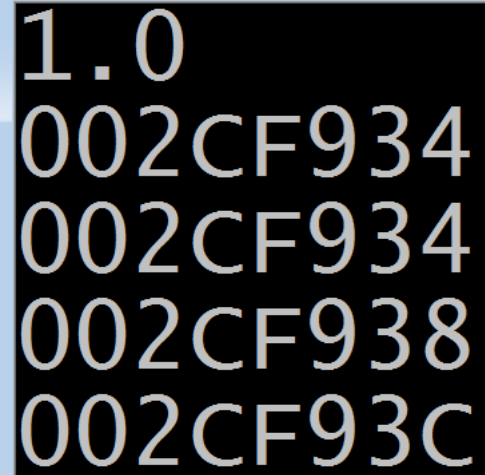
```
#include <stdio.h>
#include <stdlib.h>

int main(){
    int a = 100;
    printf("%d\n", a); ← In ra giá trị của a
    printf("%p\n", &a); ← In ra địa chỉ của a
    system("pause");
    return 0;
}
```

Toán tử &

- Toán tử & trả về địa chỉ của một biến
- Ví dụ

```
#include <stdio.h>
#include <stdlib.h>
typedef struct sPoint3D{float x, y, z;} Point3D;
int main(){
    Point3D p1 = {1.0f, 2.0f, 3.0f};
    printf("%-5.1f\n", p1.x);
    printf("%p\n", &p1);
    printf("%p\n", &p1.x);
    printf("%p\n", &p1.y);
    printf("%p\n", &p1.z);
    system("pause");
    return 0;
}
```



```
1.0
002CF934
002CF934
002CF938
002CF93C
```

← In ra giá trị của p1.x

← In ra địa chỉ của p1

← In ra địa chỉ của p1.x

← In ra địa chỉ của p1.y

← In ra địa chỉ của p1.z

Khai báo con trỏ

Cú pháp

<Tên kiểu> * <tên biến>;
<Tên kiểu> * <tên biến a> = 0;
<Tên kiểu> * <tên biến a> = &<tên biến b>;

<tên biến b>: Phải có kiểu <Tên kiểu>, hoặc có kiểu chuyển đổi qua được <Tên kiểu>

0: Hằng số, gọi là NULL

Khai báo con trỏ

Cú pháp

```
int a;  
int *p1;  
int *p2 = 0;  
int *p3 = &a;
```

a: là số nguyên

p1: con trỏ đến số nguyên, giá trị chưa xác định

p2: con trỏ đến số nguyên, giá trị là **NULL**

p3: con trỏ đến số nguyên, giá trị chính là địa chỉ của số nhớ a

```
double d;  
double *pd1;  
double *pd2 = 0;  
double *pd3 = &d;
```

f: là số float

pf1: con trỏ đến số float, giá trị chưa xác định

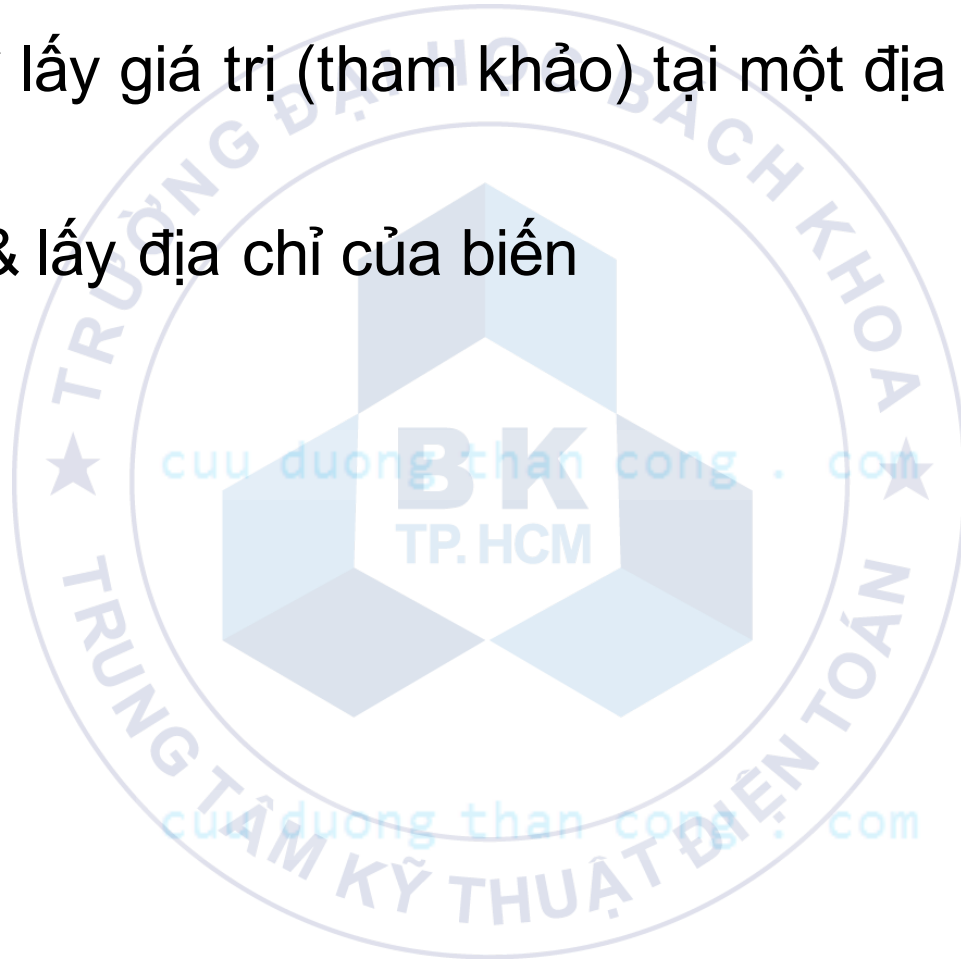
pf2: con trỏ đến số float, giá trị là **NULL**

pf3: con trỏ đến số float, giá trị chính là địa chỉ của số nhớ f

```
Point3D p1 = {1.0f, 2.0f, 3.0f};  
Point3D *pp1;  
Point3D *pp2 = 0;  
Point3D *pp3 = &p1;
```

Toán tử *

- Toán tử * lấy giá trị (tham khảo) tại một địa chỉ
- Toán tử & lấy địa chỉ của biến

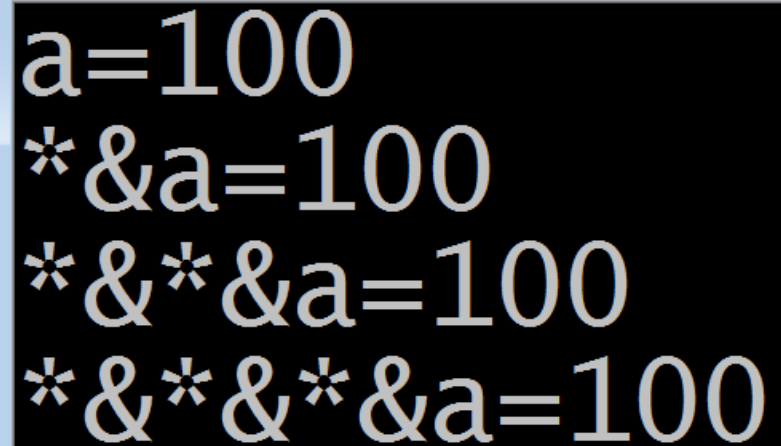


Toán tử *

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int a = 100;

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

    system("pause");
    return 0;
}
```



```
a=100
*&a=100
*&&a=100
*&&&a=100
```

Các phép toán trên con trỏ

- Tăng và Giảm: ++, --
- Cộng và trừ: +, -
- Cộng và trừ kết hợp gán: +=, -=
- So sánh: ==, !=

Gọi p là con trỏ có kiểu T: **T *p;**

Các phép cộng và trừ: làm con trỏ p tăng hay giảm một bội số của kích thước kiểu T.

Con trỏ và mảng

- Con trỏ và mảng có nhiều điểm giống nhau
 - Con trỏ & mảng: giữ địa chỉ của ô nhớ
 - Con trỏ: giữ địa chỉ của ô nhớ nào đó (của biến khác, của bộ nhớ động)
 - Mảng: giữ địa chỉ của phần tử đầu tiên
 - => có thể gán mảng vào con trỏ
 - Tuy nhiên, gán con trỏ vào mảng là không được

cuuduongthancong.com

cuuduongthancong.com

Con trỏ và mảng

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int a[5];
    int *p = a;

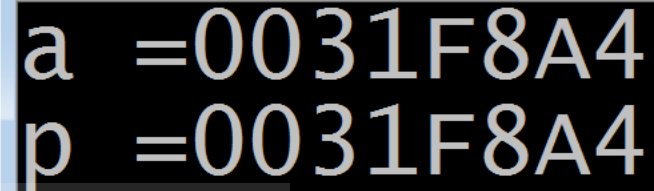
    printf("a =%p\n", a);
    printf("p =%p\n", p);

    system("pause");
    return 0;
}
```

Có thể gán con trỏ
mảng vào con trỏ

=>

A và p có giữ cùng
địa chỉ: địa chỉ của ô
đầu tiên trên mảng



```
a =0031F8A4
p =0031F8A4
```

Con trỏ và mảng

- Con trỏ và mảng có nhiều điểm giống nhau
 - Có cùng cách truy cập các ô nhớ
 - Dùng toán tử []
 - Dùng toán tử * và +

```
int a[5];  
int *p = a;  
int id = 2;
```

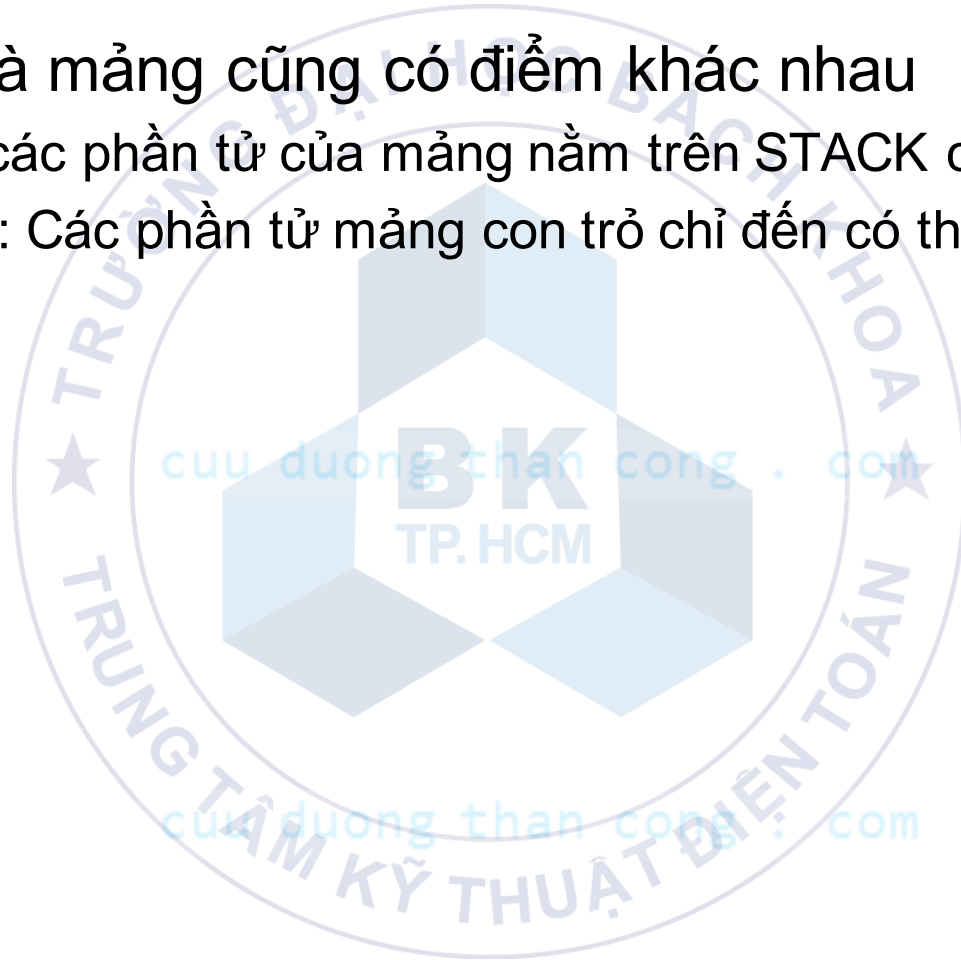
```
a[id] = 100;  
p[id] = 100;
```

```
*(a + id) = 100;  
*(p + id) = 100;
```

Giống nhau

Con trỏ và mảng

- Con trỏ và mảng cũng có điểm khác nhau
 - Mảng: các phần tử của mảng nằm trên STACK của chương trình
 - Con trỏ: Các phần tử mảng con trỏ chỉ đến có thể trên STACK hay HEAP



Cấp phát bộ nhớ động

- Giúp người lập trình tạo ra mảng động. Không cần xác định số lượng phần tử của mảng động tại thời điểm biên dịch như mảng tĩnh
- Mảng động sẽ được cấp phát trên HEAP

Mỗi khi xin cấp phát bộ nhớ
CẦN PHẢI giải phóng vùng nhớ xin được khi dùng xong

Cấp phát bộ nhớ động

- Hàm xin bộ nhớ

- `malloc`
- `calloc`
- `realloc`

- Hàm giải phóng bộ nhớ

- `free`



Cấp phát bộ nhớ động

malloc

```
#include <stdio.h>
#include <stdlib.h>
typedef struct sPoint3D{float x, y, z;} Point3D;
int main(){
    int *p1;
    float *p2;
    Point3D *p3;
    int num = 100;

    p1 = (int*)malloc(num*sizeof(int));
    p2 = (float*)malloc(num*sizeof(float));
    p3 = (Point3D*)malloc(num*sizeof(Point3D));

    free(p1); free(p2); free(p3);
    return 0;
}
```

Các biến con trỏ

Xin cấp bộ nhớ

Giải phóng bộ nhớ

Cấp phát bộ nhớ động

malloc

```
p1 = (int*)malloc(num*sizeof(int));
```

num: số con số int cần xin

sizeof(int): kích thước của mỗi số nguyên.

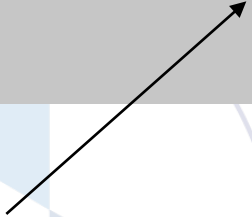
→ **num*sizeof(int)**: số bytes cần thiết để xin

(Đối số truyền vào hàm malloc là số bytes bộ nhớ cần xin)

Cấp phát bộ nhớ động

malloc

```
p1 = (int*)malloc(num*sizeof(int));
```



Hàm malloc trả về kiểu void* (kiểu vô định).
Cần ép vào kiểu của bên tay trái
p1: kiểu int* → ép vào int* bằng (int*)

cuiduongthancong.com

Cấp phát bộ nhớ động

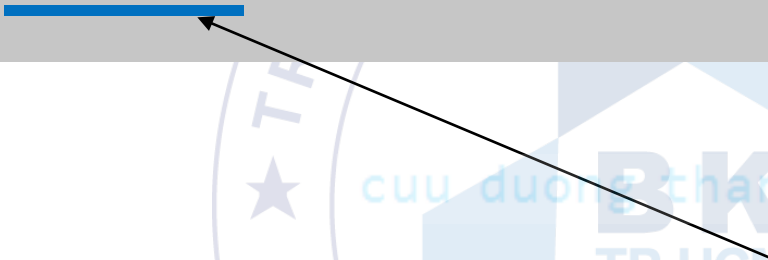
malloc

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int num = 100;
    int *p1 = (int*)malloc(num*sizeof(int));
    if(p1 == NULL){
        printf("Xin khong duoc!\n");
        exit(1);
    }
    else{
        //Thuc hien cong viec tai day
        free(p1);
    }
    return 0;
}
```

Cấp phát bộ nhớ động

malloc

```
int *p1 = (int*)malloc(num*sizeof(int));  
  
if(p1 == NULL){ ... } else { ... }
```



Hàm malloc trả về NULL nếu không xin được.

Lúc đó, không thể dùng bộ nhớ được!

Do đó, LUÔN LUÔN kiểm tra xem malloc có trả về NULL hay không

Cấp phát bộ nhớ động

Ví dụ

Khai báo biến con trỏ đến các kiểu + xin cấp phát bộ nhớ động

```
typedef struct{
    float x, y, z;
} Point3D;
int main(){
    int num = 20;
    int *int_ptr = (int*)malloc(num*sizeof(int));
    char *str = (char*)malloc(num*sizeof(char));
    double *double_ptr = (double*)malloc(num*sizeof(double));
    Point3D *p_ptr = (Point3D*)malloc(num*sizeof(Point3D));
    //Sử dụng
    free(int_ptr);
    free(str);
    free(double_ptr);
    free(p_ptr);

    return 0;
}
```

Giải phóng các vùng nhớ sau khi sử dụng

Con trỏ và cấu trúc

Khai báo

```
typedef struct{  
    float x, y, z;  
} Point3D;
```

← **(1)** Định nghĩa kiểu cấu trúc: **Point3D**

```
Point3D *p_ptr = (Point3D*)malloc(sizeof(Point3D));
```

// (4) Sử dụng

```
free(p_ptr);
```

(2) Khai báo con trỏ đến một mảng

(3) Xin cấp phát bộ nhớ trên HEAP,
p_ptr: giữ địa chỉ của ô nhớ đầu tiên trong
vùng được cấp

(5) Giải phóng vùng nhớ

Con trỏ và cấu trúc

Truy cập biến thành viên cấu trúc qua con trỏ

Ví dụ: gán các biến thành viên của cấu trúc Point3D

```
(*p_ptr).x = 4.5f; (*p_ptr).y = 5.5f; (*p_ptr).z = 6.5f;
```

```
p_ptr->x = 7.5f; p_ptr->y = 8.5f; p_ptr->z = 9.5f;
```

p_ptr : Ô nhớ (biến) chứa địa chỉ của một cấu trúc Point3D

(*p_ptr) : Nghĩa là vùng nhớ của cấu trúc Point3D

(*p_ptr).x : Nghĩa là vùng nhớ chứa biến x của cấu trúc Point3D

p_ptr->x : Nghĩa là vùng nhớ chứa biến x của cấu trúc Point3D, truy cập thông qua toán tử **->** từ con trỏ **p_ptr**

Con trỏ và cấu trúc

Truy cập biến thành viên cấu trúc qua con trỏ

Ví dụ: gán các biến thành viên của cấu trúc Point3D

```
(*p_ptr).x = 4.5f; (*p_ptr).y = 5.5f; (*p_ptr).z = 6.5f;
```

```
p_ptr->x = 7.5f; p_ptr->y = 8.5f; p_ptr->z = 9.5f;
```

Tổng quát:

<con trỏ> **->** <Biến thành viên của cấu trúc>

Như:

p_ptr->x

Con trỏ và cấu trúc

Chương trình minh họa

```
# include <stdio.h>
# include <stdlib.h>
typedef struct{
    float x, y, z;
} Point3D;
int main(){
    Point3D p = {1.5f, 2.5f, 3.5f};
    Point3D *p_ptr = (Point3D*)malloc(sizeof(Point3D));
    (*p_ptr).x = 4.5f; (*p_ptr).y = 5.5f; (*p_ptr).z = 6.5f;
    p_ptr->x = 7.5f; p_ptr->y = 8.5f; p_ptr->z = 9.5f;

    printf("p = [%-4.1f, %-4.1f, %4.1f]\n",
           p.x, p.y, p.z);
    printf("*p_ptr = [%-4.1f, %-4.1f, %4.1f]\n",
           (*p_ptr).x, (*p_ptr).y, (*p_ptr).z);
    printf("*p_ptr = [%-4.1f, %-4.1f, %4.1f]\n",
           p_ptr->x, p_ptr->y, p_ptr->z);
    free(p_ptr);
    system("pause");
    return 0;
}
```

Thứ tự các phép toán *, ++ và --

❖ *p++ // *(p++)
❖ *++p // *(++p)
❖ ++*p // ++(*p)
❖ (*p)++ // Tăng vùng nhớ do con trỏ p chỉ đến

Khi nghi ngờ, hoặc không nhớ ... hãy dùng toán tử
() để phân giải độ ưu tiên

Con trỏ và const

```
int a = 20, b = 30, c = 40;
```

```
const int * ptr1 = &a;  
//int const * ptr1 = &a;
```

```
int* const ptr2 = &b;
```

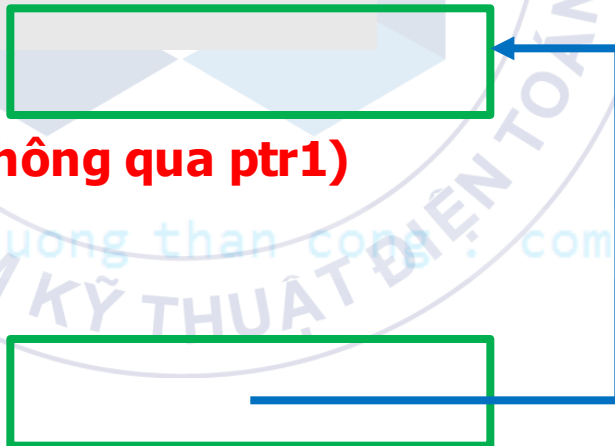
ptr1: có thể thay đổi được.

Giá trị mà **ptr1** chỉ đến không thể thay đổi được

Ô nhớ ptr1 chỉ đến

(Không thể thay đổi được thông qua ptr1)

ptr1:



Con trỏ và const

```
int a = 20, b = 30, c = 40;
```

```
const int * ptr1 = &a;
```

```
int* const ptr2 = &b; →
```

ptr2: Không thể thay đổi được giá trị của ptr2 = không thể làm ptr2 chỉ đến ô nhớ nào khác sau dòng này.

Giá trị mà **ptr2** chỉ đến có thể thay đổi được qua **tr**.

Ô nhớ ptr2 chỉ đến



(Không thể thay đổi được ptr2)

ptr2:



Con trỏ và const

Các lỗi thông dụng

```
int main(){  
    int a = 20, b = 30, c = 40;  
    const int * ptr1;  
    int* const ptr2 = &a;  
    int* const ptr3;
```

Error: const variable "ptr3" requires an initializer

Ptr3: là con trỏ hằng nhưng không được khởi động tương tự như cho ptr2

Con trỏ và const

Các lỗi thông dụng

```
int a = 20, b = 30, c = 40;  
const int * ptr1;  
int* const ptr2 = &a;  
int* const ptr3;  
*ptr1 = 100;
```

Error: expression must be a modifiable lvalue

Giá trị mà ptr1 chỉ đến không thay đổi được qua con trỏ ptr1.
Do đó, nó không thể nằm bên trái biểu thức gán

Con trỏ và const

Các lỗi thông dụng

```
int a = 20, b = 30, c = 40;  
const int * ptr1;  
int* const ptr2 = &a;  
int* const ptr3;  
*ptr1 = 100;  
ptr2 = &c;
```

Error: expression must be a modifiable lvalue

Con trỏ ptr2 là hằng số, nó chỉ nhận giá trị khởi động
Sau đó, không thể làm ptr2 chỉ đến đối tượng nào khác

Con trỏ và const

Các lỗi thông dụng

```
int a = 20, b = 30, c = 40;
const int * ptr1;
int* const ptr2 = &a;
int* ptr3 = &c;
ptr1 = ptr3;
ptr3 = ptr1;
```

Error: a value of type "const int *" cannot be assigned to an entity of type "int *"

ptr3: con trỏ bình thường, thay đổi được nó và giá trị nó chỉ đến

Gán con trỏ ptr1 vào ptr3: khiến cho giá trị mà ptr1 chỉ đến có thể thay đổi được
→ bộ biên dịch không cho phép.

Vì nếu cho phép thì ý nghĩa của ptr1 không còn.

Người lập trình luôn luôn có thể thay đổi nội dung mà ptr1 chỉ đến, bằng cách dùng con trỏ phụ.

Con trỏ và const

Các lỗi thông dụng

```
int a = 20, b = 30, c = 40;
const int * ptr1;
int* const ptr2 = &a;
int* ptr3 = &c;
ptr1 = ptr3; -> OK
ptr3 = ptr1; -> LỖI
```

Error: a value of type "const int *" cannot be assigned to an entity of type "int *"

ptr3: con trỏ bình thường, thay đổi được nó và giá trị nó chỉ đến

Gán con trỏ ptr1 vào ptr3: khiến cho giá trị mà ptr1 chỉ đến có thể thay đổi được
→ bộ biên dịch không cho phép.

Vì nếu cho phép thì ý nghĩa của ptr1 không còn.

Người lập trình luôn luôn có thể thay đổi nội dung mà ptr1 chỉ đến, bằng cách dùng con trỏ phụ.

Con trỏ và const

Các lỗi thông dụng

```
int a = 20, b = 30, c = 40;  
const int * ptr1;  
int* const ptr2 = &a;  
int* ptr3 = &c;  
ptr2 = ptr3; -> LỖI
```

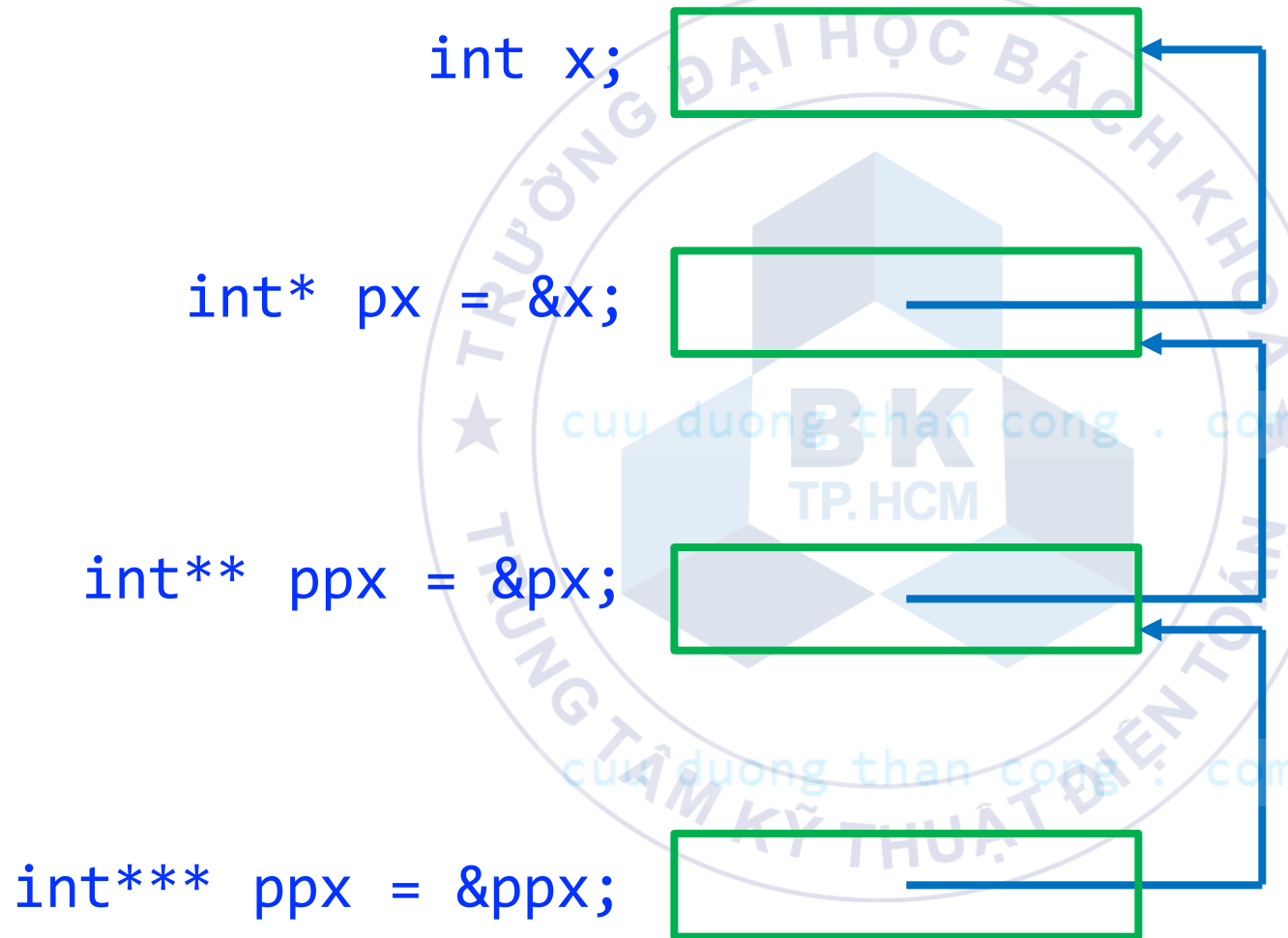
int *const ptr2

Error: expression must be a modifiable lvalue

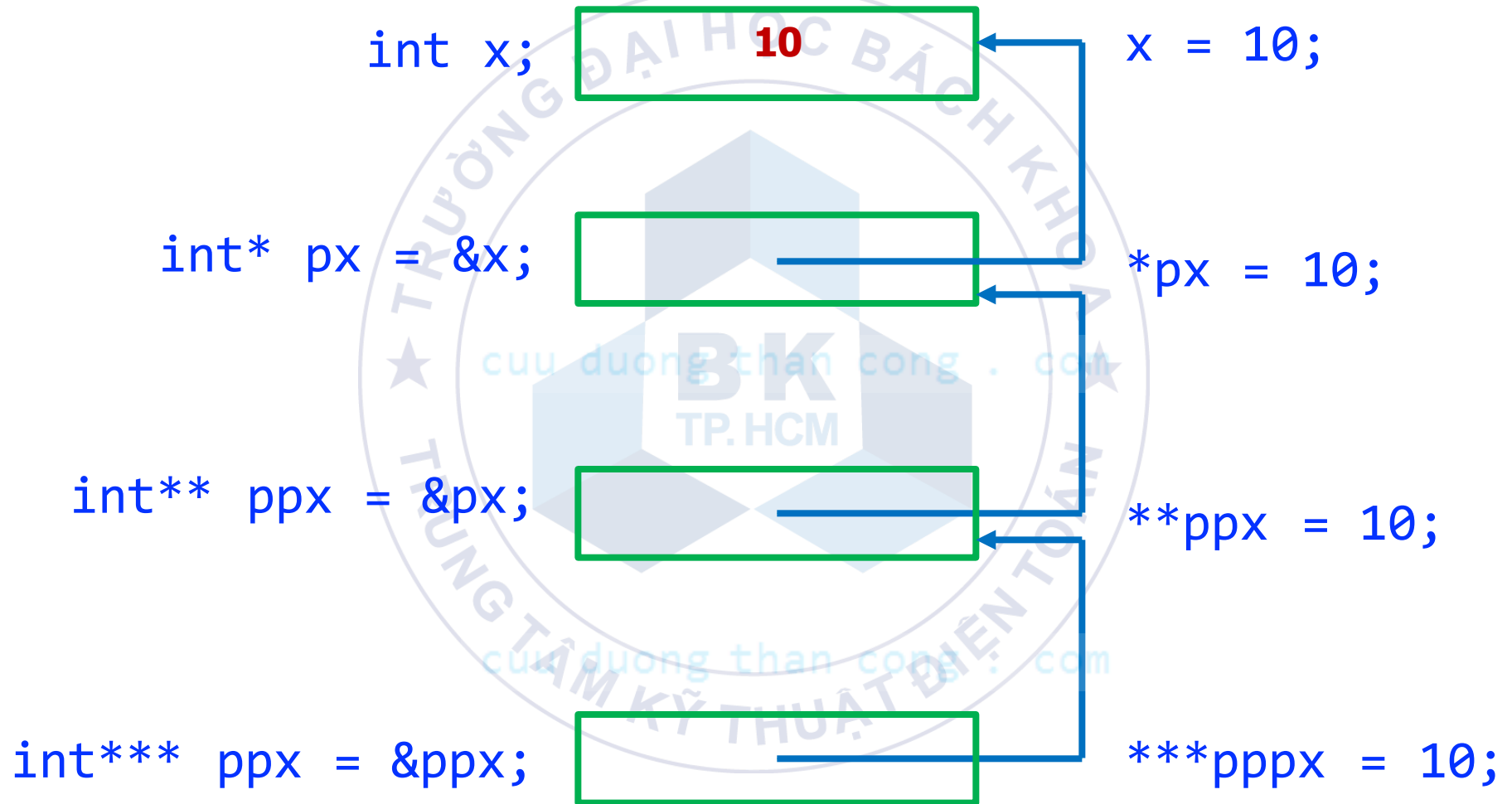
```
ptr3 = ptr2; -> OK
```

ptr2: không thể thay đổi giá trị được

Con trỏ đến con trỏ



Con trỏ đến con trỏ



Con trỏ void

- void *ptr: là con trỏ chưa định kiểu
 - Có thể được ép kiểu về kiểu mong muốn
 - Như các hàm malloc và free
 - Con trỏ void giúp chương trình uyển chuyển,
 - Nhưng rủi ro đi kèm: bộ biên dịch không thể kiểm tra tương thích kiểu tại thời điểm biên dịch

cuduongthancong.com

Bài tập

- Hiện thực lại các bài tập về array nhưng dữ liệu các array nằm trên bộ nhớ HEAP

