

KỸ THUẬT LẬP TRÌNH C/C++

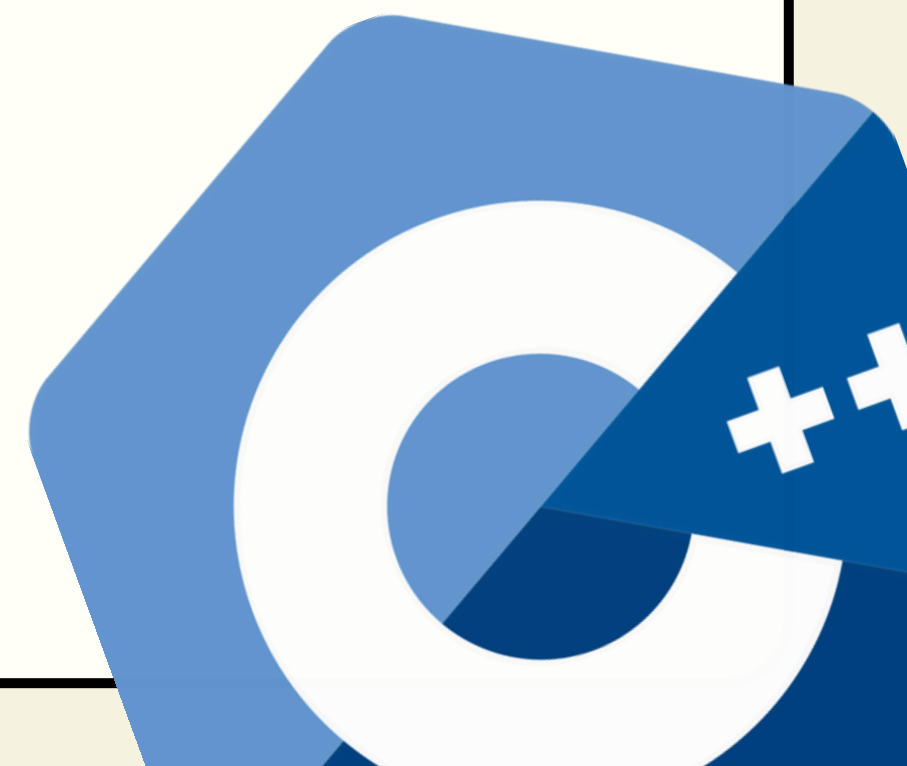
Dạy bởi 1 SVBK

Đại học Bách Khoa Hà Nội
Kỹ thuật Điện tử - Viễn thông

2024-25

NỘI DUNG

- Địa chỉ bộ nhớ (**Memory address**)
- Con trỏ (**Pointer**)
- Tham chiếu và tham trị trong C
- Con trỏ làm tham số cho hàm
- Con trỏ và mảng 1 chiều
- Cấp phát động, malloc(), calloc(), free(), realloc()
- Con trỏ hàm (**Function pointer**)



GIỚI THIỆU CHUNG

Một **con trỏ (pointer)** là một biến lưu trữ **địa chỉ bộ nhớ của một biến khác**. Thay vì chứa một giá trị trực tiếp, nó **giữ địa chỉ** nơi giá trị được lưu trữ trong bộ nhớ. Có hai toán tử quan trọng mà chúng ta sẽ sử dụng trong khái niệm con trỏ:

- **Toán tử giải tham chiếu (*)**

- Được sử dụng để khai báo biến con trỏ.
- Dùng để truy cập giá trị được lưu trữ tại địa chỉ mà con trỏ trỏ đến.

- **Toán tử địa chỉ (&)**

- Trả về **địa chỉ của một biến**.
- Được sử dụng để **gán địa chỉ** của một biến cho một con trỏ.

ĐỊA CHỈ BỘ NHỚ

Khi một biến được tạo trong C, một **địa chỉ bộ nhớ** sẽ được **cấp phát** cho biến đó.

Địa chỉ bộ nhớ là vị trí nơi biến được lưu trữ trong máy tính.

Khi chúng ta gán một giá trị cho biến, giá trị đó sẽ được lưu trữ tại địa chỉ bộ nhớ này.

Để **truy cập địa chỉ** bộ nhớ của biến, chúng ta sử dụng **toán tử địa chỉ (&)**, và kết quả trả về chính là vị trí mà biến được lưu trữ.

```
int n = 10;  
printf("%p", &n);
```



0X0061FF1C

ĐỊA CHỈ BỘ NHỚ

LƯU Ý

- Địa chỉ bộ nhớ được biểu diễn dưới dạng **hệ thập lục phân (hexadecimal)** (có dạng 0x..). Ta có thể sẽ **không nhận được cùng một kết quả** trong chương trình của mình, vì điều này **phụ thuộc vào vị trí mà biến được lưu trữ trên máy tính của bạn**.
- Bạn cũng nên lưu ý rằng &myAge thường được gọi là **một "con trỏ" (pointer)**.
- Một con trỏ về cơ bản là một **biến lưu trữ địa chỉ bộ nhớ** của một biến khác dưới dạng giá trị của nó.
- Để in giá trị của con trỏ, chúng ta sử dụng **định dạng %p** trong printf. (1 số trường hợp nó vẫn là hệ hexa nhưng không có đầu 0x - ta cần đổi thành %#p nếu muốn in ra đầy đủ).

CON TRỎ

KHAI BÁO & KHỞI TẠO

Một biến con trỏ (pointer variable) trỏ đến một kiểu dữ liệu (như int) cùng loại và được tạo bằng **toán tử ***.

Địa chỉ của biến mà bạn đang làm việc sẽ được gán cho con trỏ.

Kiểu dữ liệu của con trỏ sẽ tương ứng với kiểu dữ liệu nó trỏ đến.

Cú pháp:

```
kiểu* tên_con_trỏ
```

VÍ DỤ

```
int n = 100;  
int* ptr = &n;
```

Giá trị **&n** được gán cho biến con trỏ **ptr** khi khởi tạo có nghĩa là: con trỏ ptr này **trỏ tới biến n**/con trỏ ptr **đang lưu trữ địa chỉ của biến n**.

Nếu không **khởi tạo ngay** khi khai báo, con trỏ sẽ chứa **giá trị rác** và **có thể gây lỗi khi sử dụng**.

CON TRỎ

GIẢI THAM CHIẾU CON TRỎ

Khi con trỏ **ptr** trở tới hay **tham chiếu (reference)** tới **biến N** thì thông qua con trỏ ptr ta có thể **truy xuất, thay đổi giá trị của biến N mà không cần dùng N**.

Để truy xuất tới giá trị của biến mà con trỏ đang trở tới, ta dùng **toán tử giải tham chiếu * (dereference)**.

Sau khi con trỏ ptr trở tới biến N thì **N** và ***ptr là một**, đều **truy xuất đến ô nhớ** mà N đang chiếm để **lấy giá trị tại ô nhớ đó**.

Lưu ý là bạn cần phân biệt dấu ***** **khi khai báo con trỏ ptr** và dấu ***** **khi giải tham chiếu con trỏ ptr**. Dấu ***** khi khai báo **thể hiện ptr là một con trỏ** còn dấu ***** trước ptr ở những câu lệnh sau là **toán tử giải tham chiếu**.

CON TRỎ

GIẢI THAM CHIẾU CON TRỎ

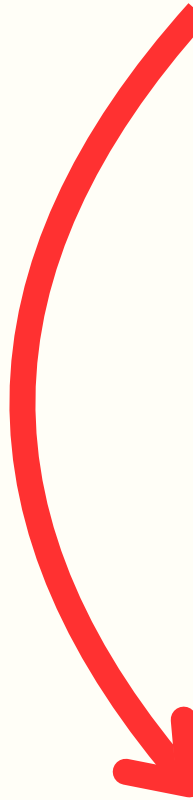
TỔNG KẾT

- **int* ptr:** dấu sao là dấu thể hiện rằng pointer **là con trỏ khi khai báo.**
- ***ptr:** dấu sao là **dấu giải tham chiếu**, giúp truy xuất **giá trị tại vùng nó trỏ tới**, đóng vai trò tương tự khi dùng tên biến để truy xuất giá trị.
- **Dấu giải tham chiếu khác dấu sao khi khai báo.**
- **ptr:** không có dấu sao → giá trị của con trỏ là **địa chỉ của biến nó trỏ tới**, đóng vai trò tương tự khi dùng &tên_biến để truy xuất địa chỉ.

CON TRỎ

GIẢI THAM CHIẾU CON TRỎ

```
int N = 1000;
printf("Dia chi cua N : %d\n", &N);
int *ptr = &N; // ptr trỏ tới N
printf("Gia tri cua ptr : %d\n", ptr);
//Toán tử giải tham chiếu
printf("Gia tri cua bien ma con tro ptr tro toi : %d\n", *ptr);
//Thay đổi N bằng ptr, *ptr và N là một
*ptr = 280;
printf("Gia tri cua N sau thay doi : %d %d\n", N, *ptr);
```



```
Dia chi cua N : 6422296
Gia tri cua ptr : 6422296
Gia tri cua bien ma con tro ptr tro toi : 1000
Gia tri cua N sau thay doi : 280 280
```

OUTPUT

CON TRỎ

PHÂN LOẠI CON TRỎ

Trong ngôn ngữ C, kiểu dữ liệu của con trỏ được xác định bằng **kiểu dữ liệu của biến mà nó trỏ đến**. Điều này giúp trình biên dịch biết con trỏ sẽ **truy xuất dữ liệu theo kiểu nào** và **chiếm bao nhiêu byte trong bộ nhớ**.

Phân loại:

- Con trỏ trỏ về các **kiểu dữ liệu nguyên thủy**: int, char, double,...
- Con trỏ **không kiểu**: void
- Con trỏ **trỏ về hàm, mảng, cấu trúc**,...
- Con trỏ trỏ về hằng số.
- Con trỏ **“hoang dã”**: con trỏ không có giá trị được khởi tạo.

CON TRỎ

pass by reference



fillCup(*)*

THAM CHIẾU VÀ THAM TRỊ

pass by value



fillCup(*)*

Tham chiếu (Pass by Reference) và Tham trị (Pass by Value) trong C

Trong ngôn ngữ lập trình C, khi **truyền biến vào hàm (truyền đối số)**, có hai cách chính:

- **Truyền theo giá trị** (Pass by Value - Tham trị)
- **Truyền theo địa chỉ** (Pass by Address - Tham chiếu gián tiếp, vì **C không có pass by reference thực sự như C++**)

CON TRỎ

THAM CHIẾU VÀ THAM TRỊ

TRUYỀN THAM TRỊ

Khi truyền theo giá trị, một **bản sao** của biến được tạo ra trong bộ nhớ và được sử dụng trong hàm. Do đó, mọi thay đổi bên trong hàm sẽ **không ảnh hưởng đến biến gốc**.

```
#include <stdio.h>

void thamTri(int a) {
    a = 10;
}

int main()
{
    int a = 5;
    printf("Truoc: %d", a);
    thamTri(a);
    printf("Sau: %d", a);
    return 0;
}
```



Truoc: 5
Sau: 5

OUTPUT

Biến gốc không bị ảnh hưởng sau khi gọi hàm thamTri()

CON TRỎ

THAM CHIẾU VÀ THAM TRỊ

TRUYỀN THEO ĐỊA CHỈ (THAM CHIẾU GIÁN TIẾP)

C không hỗ trợ tham chiếu thực sự như C++, nhưng có thể **truyền địa chỉ của biến vào hàm bằng con trỏ**. Điều này giúp **thay đổi giá trị của biến gốc**.

```
#include <stdio.h>

void thamChieuGianTiep(int *a) {
    *a = 10;
}

int main()
{
    int a = 5;
    printf("Truoc: %d\n", a);
    thamChieuGianTiep(&a);
    printf("Sau: %d", a);
    return 0;
}
```



Truoc: 5
Sau: 10

CON TRỎ

THAM CHIẾU VÀ THAM TRỊ

KHI NÀO DÙNG?

- **Dùng Pass by Value (Tham trị)** nếu không muốn hàm thay đổi giá trị biến gốc.
- **Dùng Pass by Address (Tham chiếu gián tiếp - con trỏ)** khi cần thay đổi giá trị của biến gốc hoặc làm việc với mảng/struct lớn để tránh sao chép tốn tài nguyên.

Con trỏ và hàm trong C

Trong ngôn ngữ lập trình C, **con trỏ (pointer)** và **hàm** có thể kết hợp với nhau theo nhiều cách để **tối ưu hóa bộ nhớ, tăng hiệu suất chương trình** và **truyền dữ liệu hiệu quả hơn**.

Con trỏ được sử dụng trong cấu trúc của hàm trong các trường hợp sau:

- Hàm trả về một địa chỉ (hoặc mảng)
- Hàm có nhiều đầu ra
- Đầu vào có cấu trúc mảng
- Địa chỉ của một hàm (con trỏ hàm) trong cơ chế callback

CON TRỎ

CON TRỎ VÀ HÀM

CON TRỎ LÀM THAM SỐ CHO HÀM

- Như trong phần trước, để thay đổi giá trị của 1 biến sau khi hàm kết thúc thì việc **truyền tham trị là không hợp lý**, thay vì đó bạn hãy sử dụng con trỏ với mục đích là thay đổi giá trị của biến thông qua con trỏ (**truyền địa chỉ – tham chiếu gián tiếp**).
- Khi hàm có **tham số là một con trỏ** thì khi gọi hàm bạn cần truyền vào một giá trị phù hợp, có thể là **địa chỉ của 1 biến hoặc một con trỏ khác**.

CON TRỎ

CON TRỎ VÀ HÀM

CON TRỎ LÀM THAM SỐ CHO HÀM

```
void swap(int *a, int *b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}  
  
int main() {  
    int a = 5, b = 10;  
    printf("Truoc: %d %d\n", a, b);  
    swap(&a, &b);  
    printf("Sau: %d %d", a, b);  
    return 0;  
}
```



Truoc: 5 10
Sau: 10 5

Khi **tham số là con trỏ**, **đối số** được truyền vào sẽ là **địa chỉ của các biến**.
Việc **truyền địa chỉ** giúp chúng ta có thể **thao tác với biến gốc**.

CON TRỎ

Con trỏ và mảng trong C

Trong ngôn ngữ lập trình C, con trỏ và mảng có mối liên hệ chặt chẽ với nhau. Hiểu về cách chúng hoạt động giúp ta quản lý bộ nhớ hiệu quả hơn, tối ưu hóa hiệu suất của chương trình.

CON TRỎ VÀ MẢNG 1 CHIỀU

KHÁI QUÁT

- Khi bạn khai báo một mảng, **tên mảng chính là địa chỉ của phần tử đầu tiên.**
(arr == &arr[0])
- Con trỏ có thể được sử dụng để **truy cập và duyệt qua** các phần tử của mảng.
- Có thể sử dụng con trỏ để **cấp phát động** mảng thay vì dùng mảng tĩnh.

CON TRỎ

```
int arr[5] = {10, 20, 30, 40, 50};  
int *ptr = arr;  
  
printf("%d\n", *ptr);  
printf("%d\n", *(ptr + 1));
```



10
20

CON TRỎ VÀ MẢNG 1 CHIỀU

- Ta khởi tạo con trỏ ptr với giá trị là **arr – tên của mảng**. Khi tên của mảng đứng 1 mình với vai trò là 1 giá trị, nó sẽ tương ứng với **&arr[0]** → **địa chỉ của phần tử đầu tiên trong mảng**.
- Để truy cập phần tử ở **index “n”** ta sẽ sử dụng ***(ptr + n)**.

CON TRỎ

```
int arr[5] = {10, 20, 30, 40, 50};  
  
int *ptr = arr;  
for (int i = 0; i < 5; i++)  
{  
    printf("%d ", *(ptr + i));  
}
```



10 20 30 40 50

CON TRỎ VÀ MẢNG 1 CHIỀU

Trong ví dụ trên, ta duyệt mảng arr dùng con trỏ. Ta khởi tạo con trỏ ptr với giá trị là địa chỉ phần tử đầu tiên, rồi duyệt dần lên arr[1], arr[2],... qua phép (ptr + i)
→ ngoài ra cũng có thể viết là ptr++

CON TRỎ

CẤP PHÁT ĐỘNG

CẤP PHÁT ĐỘNG CHO MẢNG

- **Cấp phát động (Dynamic memory allocation)** là một kỹ thuật giúp ta có thể xin cấp phát một vùng nhớ phù hợp với nhu cầu của bài toán trong lúc thực thi thay vì phải khai báo cố định (khác với kiểu stack-frame thông thường).
- Cấp phát động thường được sử dụng để **cấp phát mảng động** hoặc sử dụng trong các **cấu trúc dữ liệu**.
- Có nhiều cấu trúc dữ liệu quan trọng dựa trên kỹ thuật này như: **Danh sách liên kết, cây nhị phân** hay các cấu trúc dữ liệu **dạng mảng động**.
- Khi sử dụng cấp phát động thì vùng nhớ cấp phát sẽ là **vùng nhớ heap**.

CON TRỎ

CẤP PHÁT ĐỘNG

CẤP PHÁT ĐỘNG CHO MẢNG

- Để có thể thao tác với việc **cấp phát động vùng nhớ** và **giải phóng vùng nhớ sau khi sử dụng**, ta sẽ làm quen với 4 hàm sau:
 - *malloc()*
 - *calloc()*
 - *free()*
 - *realloc()*
- Các hàm này sẽ có trong **thư viện <stdlib.h>**.

CON TRỎ

CẤP PHÁT ĐỘNG

HÀM MALLOC()

Hàm **malloc()** (viết tắt của **memory allocation**) được sử dụng để cấp phát một khối bộ nhớ liên tục trên **heap** trong **thời gian chạy (runtime)**. Bộ nhớ được cấp phát bởi **malloc()** không được khởi tạo, có nghĩa là nó **chứa các giá trị rác**.

Cú pháp:

```
cast_type* ptr = (cast_type *)malloc(số_phần_tử * sizeof(cast_type))
```



ép kiểu thành **int*** do giá trị trả về của mảng là **con trỏ kiểu void***

CON TRỎ

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

int main() {
    int n = 10;
    int *ptr = (int *)malloc(n * sizeof(int));
    if (ptr == NULL) {
        printf("Cap phat khong thanh cong");
    }
    for (int i = 0; i < n; i++) {
        ptr[i] = i;
    }

    for (int i = 0; i < n; i++) {
        printf("%d ", ptr[i]);
    }
}
```

CẤP PHÁT ĐỘNG - MALLOC()

Toán tử **sizeof()** trả về kích thước dữ liệu, khi nhân với số phần tử, ta sẽ có **tổng dung lượng cần cấp phát**.

Nếu **không đủ bộ nhớ để cấp phát**, sẽ trả về **NULL**.

Thao tác với mảng như bình thường.

0 1 2 3 4 5 6 7 8 9

CON TRỎ

CẤP PHÁT ĐỘNG

HÀM CALLOC()

Hàm **calloc()** - **contagious allocation** thực hiện cấp phát bộ nhớ và **khởi tạo tất cả các ô nhớ có giá trị bằng 0**. Vì thế nên hàm **calloc** sẽ cần thời gian thực thi **lâu hơn malloc()**.

Chú ý: trong **malloc()** có **1 tham số**, trong **calloc()** có **2 tham số**.

Cú pháp:

```
cast_type* ptr = (cast_type *)calloc(số_phần_tử, sizeof(cast_type))
```

ép kiểu thành **int*** do giá trị trả về của mảng là **con trỏ kiểu void***

CON TRỎ

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

int main() {
    int n = 10;
    int *ptr = (int *)calloc(n, sizeof(int));
    if (ptr == NULL) {
        printf("Cap phat khong thanh cong");
    }
    printf("Mang khi khoi tao: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", ptr[i]);
    }
}
```

Mang khi khoi tao: 0 0 0 0 0 0 0 0 0 0

CẤP PHÁT ĐỘNG - CALLOC()

Toán tử **sizeof()** trả về kích thước dữ liệu, khi nhân với số phần tử, ta sẽ có **tổng dung lượng cần cấp phát**.

Nếu **không đủ bộ nhớ để cấp phát**, sẽ trả về **NULL**.

In ra các phần tử ban đầu được khởi tạo. (Tất cả có giá trị khởi tạo là 0).

CON TRỎ

CẤP PHÁT ĐỘNG

HÀM FREE()

Bộ nhớ được cấp phát bằng các hàm malloc() và calloc() **không tự động giải phóng**. Hàm **free()** được sử dụng để **trả lại bộ nhớ được cấp phát động cho hệ điều hành**. Việc **giải phóng bộ nhớ không còn cần thiết** là rất quan trọng để **tránh rò rỉ bộ nhớ (memory leaks)**.

Cú pháp:

```
free(ptr)
```

Sau khi gọi free(), con trỏ **vẫn trỏ đến vùng nhớ đã bị giải phóng**, dẫn đến **lỗi nếu truy cập**. Gán con trỏ về NULL **giúp tránh con trỏ treo (dangling pointer)** và ngăn chặn **hành vi không xác định (undefined behavior)**.

CON TRỎ

```
int main()
{
    int n = 10;
    int *ptr = (int *)calloc(n, sizeof(int));
    if (ptr == NULL)
    {
        printf("Cap phat khong thanh cong");
    }
    printf("Mang khi khoi tao: ");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", ptr[i]);
    }

    free(ptr);
    ptr = NULL;
}
```

CẤP PHÁT ĐỘNG - FREE()

Giải phóng bộ nhớ sử dụng free().

Sau khi gọi free(), **con trỏ vẫn trỏ đến vùng nhớ đã bị giải phóng**, dẫn đến **lỗi nếu truy cập**. Gán con trỏ về **NULL** giúp **tránh con trỏ treo (dangling pointer)** và ngăn chặn hành vi không xác định (**undefined behavior**).

CON TRỎ

CẤP PHÁT ĐỘNG

HÀM REALLOC()

Hàm **realloc()** được sử dụng để **thay đổi kích thước** của một khối bộ nhớ **đã được cấp phát trước đó**. Nó cho phép thay đổi kích thước vùng nhớ động mà **không cần trực tiếp giải phóng và cấp phát lại thủ công**.

Cú pháp:

```
ptr = (cast_type*)realloc(ptr, số_phần_tử_mới * sizeof(cast_type))
```

Hàm này **trả về một con trỏ** trỏ đến **vùng nhớ mới được cấp phát**, hoặc **NULL** nếu việc cấp phát lại **thất bại**. Nếu thất bại, vùng nhớ ban đầu **vẫn giữ nguyên**. **Các giá trị trong ô nhớ cũ vẫn giữ nguyên** nếu **realloc()** thành công, nhưng vùng nhớ mới chưa được khởi tạo **có thể chứa giá trị rác**.

CON TRỎ

```
int main()
{
    int n = 10;
    int *a = (int *)malloc(n * sizeof(int));
    printf("Truoc: ");
    for (int i = 0; i < n; i++)
    {
        a[i] = i;
        printf("%d ", a[i]);
    }
    printf("\n");
    a = (int *)realloc(a, (n + 5) * sizeof(int));
    printf("Sau: ");
    for (int i = 0; i < n + 5; i++)
    {
        printf("%d ", a[i]);
    }
    free(a);
    a = NULL;
    return 0;
}
```

CẤP PHÁT ĐỘNG - REALLOC()

Sử dụng **realloc()** để cấp phát lại.
Có thể thêm bước **kiểm tra** nếu
realloc() thất bại.

Giải phóng bộ nhớ bằng **hàm free()**.
Gán cho con trỏ giá trị NULL để tránh
con trỏ treo.

CON TRỎ

CẤP PHÁT ĐỘNG - REALLOC()

Mảng động trước khi cấp phát vùng nhớ mới.

Trước: 0 1 2 3 4 5 6 7 8 9

Sau: 0 1 2 3 4 5 6 7 8 9 301989906 54624 14499560 14483648 1869767529

Các phần tử trong vùng nhớ mới **mang giá trị rác** do **chưa khởi tạo giá trị**.

CON TRỎ

CON TRỎ HÀM

Trong ngôn ngữ C, **con trỏ hàm (function pointer)** là một loại con trỏ lưu trữ **địa chỉ của một hàm**, cho phép truyền hàm như một đối số và gọi hàm một cách linh hoạt. Nó hữu ích trong các kỹ thuật như **hàm gọi lại (callback functions)**, **lập trình hướng sự kiện (event-driven programs)** và **đa hình (polymorphism)** – một khái niệm mà trong đó một hàm hoặc toán tử có thể hoạt động khác nhau tùy theo ngữ cảnh.

CON TRỎ

CON TRỎ HÀM

Cú pháp:

```
cast_type (*ptr)(parameters_type) = &tên_hàm
```

Dấu ngoặc đơn xung quanh ptr là cần thiết, nếu không, nó sẽ bị **hiểu nhầm** như một khai báo hàm với **kiểu trả về là return_type*** và **tên là ptr**.

Kiểu của một hàm được xác định bởi **kiểu trả về, số lượng và kiểu** của các tham số. Vì vậy, con trỏ hàm cần được khai báo sao cho phù hợp với chữ ký của hàm mà nó sẽ trỏ đến sau này.

CON TRỎ

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

```
int dienTich(int x, int y)
{
    return x * y;
}
```

```
int main()
{
```

```
    int (*fptr)(int, int) = &dienTich;
    printf("%d", fptr(10,10));
}
```

100

CON TRỎ HÀM

Tạo hàm tính diện tích dienTich() với **2 tham số x và y** mang kiểu **int**.

Khai báo con trỏ hàm **trỏ tới hàm dienTich()**. Chú ý tham số.

Trong C, khi gọi một con trỏ hàm, ta **có thể bỏ qua** dấu giải tham chiếu (*) vì bản chất của con trỏ hàm khi được sử dụng trong **ngữ cảnh gọi hàm** sẽ **tự động được giải tham chiếu**.

CON TRỎ

CON TRỎ HÀM

THUỘC TÍNH CON TRỎ HÀM

Con trỏ hàm trỏ đến mã thực thi thay vì dữ liệu, vì vậy nó có một số hạn chế so với các con trỏ khác. Dưới đây là một số thuộc tính quan trọng của con trỏ hàm:

- **Trỏ đến địa chỉ bộ nhớ** của một hàm trong **phân đoạn mã (code segment)**.
- Yêu cầu **chữ ký hàm** chính xác (**kiểu trả về và danh sách tham số**).
- Có thể trỏ đến các **hàm khác nhau với cùng một chữ ký**.
- **Không thể thực hiện** các phép toán như tăng (++) hoặc giảm (--).
- Hỗ trợ chức năng giống như mảng để **tạo bảng con trỏ hàm**.

CON TRỎ

CON TRỎ HÀM

ỨNG DỤNG CON TRỎ HÀM

Con trỏ hàm giúp **tăng tính linh hoạt và mở rộng khả năng lập trình**. Dưới đây là một số ứng dụng quan trọng của con trỏ hàm:

- Callback function (**hàm gọi lại**)
- Xây dựng menu chương trình động
- Giao tiếp với phần cứng & hệ điều hành

CON TRỎ

CON TRỎ HÀM - CALLBACK FUNCTION

```
#include <stdio.h>

int dienTich(int x, int y) {
    return x * y;
}

int chuVi(int x, int y) {
    return (x + y) * 2;
}

void hcn(int x, int y, int (*op)(int, int)) {
    printf("%d\n", op(x, y));
}

int main() {
    hcn(5, 10, dienTich);
    hcn(5, 10, chuVi);
    return 0;
}
```

Tạo 2 hàm tính diện tích và chu vi hình chữ nhật.

Bản chất của **callback function** là truyền hàm **như tham số cho 1 hàm khác**. Ta tạo hàm **hcn()** với tham số là 1 con trỏ hàm để có thể truyền các hàm **dienTich()**, **chuVi()** vào và **thực thi một cách linh hoạt hơn**.

50
30

CON TRỎ

CON TRỎ HÀM - MẢNG CÁC HÀM

```
#include <stdio.h>

float cong(float a, float b) {
    return a + b;
}
float tru(float a, float b) {
    return a - b;
}
float nhan(float a, float b) {
    return a * b;
}
float chia(float a, float b) {
    return a / b;
}

int main () {
    float (*tinh[4])(float, float) = {cong, tru, nhan, chia};
    float x = 10.0, y = 3.0;
    printf("Tong 2 so: %.f\n", tinh[0](x,y));
    printf("Hieu 2 so: %.f\n", tinh[1](x,y));
    printf("Tich 2 so: %.f\n", tinh[2](x,y));
    printf("Thuong 2 so: %.3f\n", tinh[3](x,y));
    return 0;
}
```

Tạo 4 hàm thực hiện các phép toán cơ bản.

tinh[] là một **mảng các con trỏ hàm**, mỗi phần tử của mảng này **chứa địa chỉ** của một hàm thực hiện phép toán (**cong, tru, nhan, chia**).

```
Tong 2 so: 13
Hieu 2 so: 7
Tich 2 so: 30
Thuong 2 so: 3.333
```