MỤC LỤC

[Phần 1. C BASIC 1](#_Toc142380269)

[1.1. Complier C. 1](#_Toc142380270)

[1.1.1. 4 bước chuyển đổi từ C sang file chạy exe,out,hex 1](#_Toc142380271)

[1.2. File C. 2](#_Toc142380272)

[1.3. Token 4](#_Toc142380273)

[1.3.1. Keywords (từ khóa) 4](#_Toc142380274)

[1.3.2. Identifiers (Định danh) 4](#_Toc142380275)

[1.3.3. Constants (Hằng) 5](#_Toc142380276)

[1.3.4. Strings (Chuỗi) 5](#_Toc142380277)

[1.3.5. Special Symbols (Kí tự đặc biệt) 5](#_Toc142380278)

[1.3.6. Operators (Toán tử) 5](#_Toc142380279)

[Phần 2. C Variables and Constants 6](#_Toc142380280)

[2.1. Variable (Biến) 6](#_Toc142380281)

[2.1.1. Định nghĩa 6](#_Toc142380282)

[2.1.2. C Variable Syntax (Cú phát khái báo biến) 6](#_Toc142380283)

[2.2. Basic Data Types (Kiểu dữ liệu). 9](#_Toc142380284)

[2.2.1. Overview: 9](#_Toc142380285)

[2.3. Storage Classes in C (Lớp lưu trữ) 10](#_Toc142380286)

[2.3.1. Định nghĩa 10](#_Toc142380287)

[2.3.2. Scope: 10](#_Toc142380288)

[2.3.3. Giải thích Store Class 13](#_Toc142380289)

[2.4. Key word with variable (Từ khóa) 14](#_Toc142380290)

[2.4.1. Static: 14](#_Toc142380291)

[2.4.2. Extern: 15](#_Toc142380292)

[2.4.3. Register: 16](#_Toc142380293)

[Phần 3. Structure data type (Cấu trúc dữ liệu ): 17](#_Toc142380294)

[3.1. Structure 17](#_Toc142380295)

[3.1.1. Định nghĩa 17](#_Toc142380296)

[3.1.2. Khai báo 17](#_Toc142380297)

[3.1.3. Truy xuất dữ liệu 17](#_Toc142380298)

[3.1.4. C Structure Padding and Packing. 18](#_Toc142380299)

[3.1.5. Bit Field: 22](#_Toc142380300)

[3.2. Union 23](#_Toc142380301)

[3.2.1. Định nghĩa 23](#_Toc142380302)

[3.2.2. Size of Union 23](#_Toc142380303)

[3.2.3. Difference between C Structure and C Union 24](#_Toc142380304)

[3.3. Enum 25](#_Toc142380305)

[3.3.1. Định nghĩa 25](#_Toc142380306)

[3.3.2. Size of Enum: 25](#_Toc142380307)

[3.3.3. Note: 25](#_Toc142380308)

[Phần 4. C Pointers 26](#_Toc142380309)

[4.1. Định nghĩa 26](#_Toc142380310)

[4.1.1. Size (kích thước) : 26](#_Toc142380311)

[4.1.2. Syntax(Cú pháp): 26](#_Toc142380312)

[4.2. How to Use Pointers ( Cách dùng ) ? 27](#_Toc142380313)

[4.2.1. Pointer Declaration: 27](#_Toc142380314)

[4.2.2. Pointer Initialization: 27](#_Toc142380315)

[4.2.3. Dereferencing: 28](#_Toc142380316)

[4.2.4. Pointer Arithmetics (Toán tử) 28](#_Toc142380317)

[4.3. Types of Pointers: 31](#_Toc142380318)

[4.3.1. Array Pointer: 32](#_Toc142380319)

[4.3.2. Function Pointer : 39](#_Toc142380320)

[4.3.3. Double pointer: 43](#_Toc142380321)

[4.3.4. NULL Pointer 44](#_Toc142380322)

[4.3.5. Void Pointer 44](#_Toc142380323)

[4.3.6. Wild Pointers 44](#_Toc142380324)

[4.3.7. Constant Pointers (Hằng con trỏ) 44](#_Toc142380325)

[4.3.8. Pointer to Constant (Con trỏ hằng) 46](#_Toc142380326)

[4.4. Dynamic Memory Allocation 48](#_Toc142380327)

[4.4.1. Syntax 48](#_Toc142380328)

[4.5. Ưu điểm của con trỏ 52](#_Toc142380329)

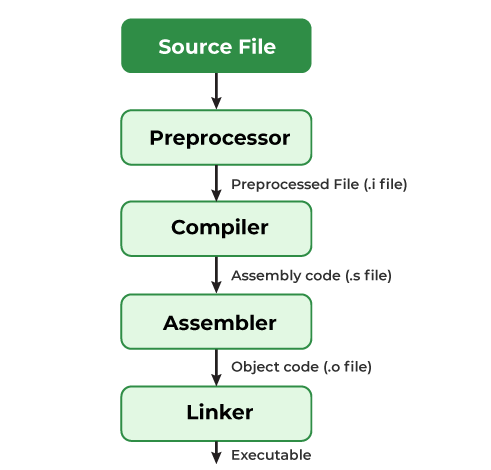
[4.6. Nhược điểm của con trỏ 52](#_Toc142380330)

# Phần 1. C BASIC

## Complier C.

### 4 bước chuyển đổi từ C sang file chạy exe,out,hex

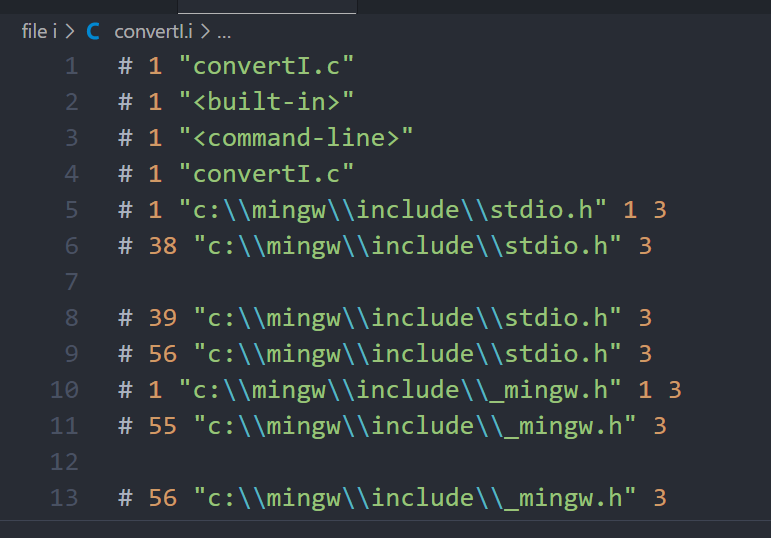
* + Pre-processing
    - Xóa comment .
    - Chèn các macro .
    - Chèn các file header .
    - Thay điều kiện tiền xử lí.
  + Compilation
  + Assembly
  + Linking

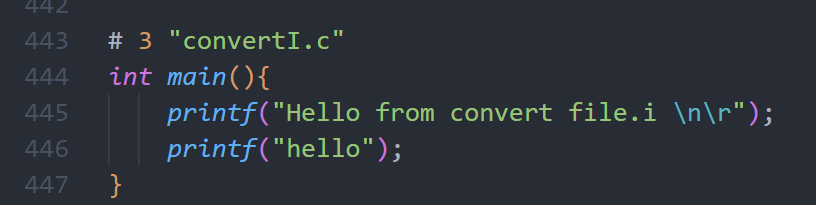


## File C.

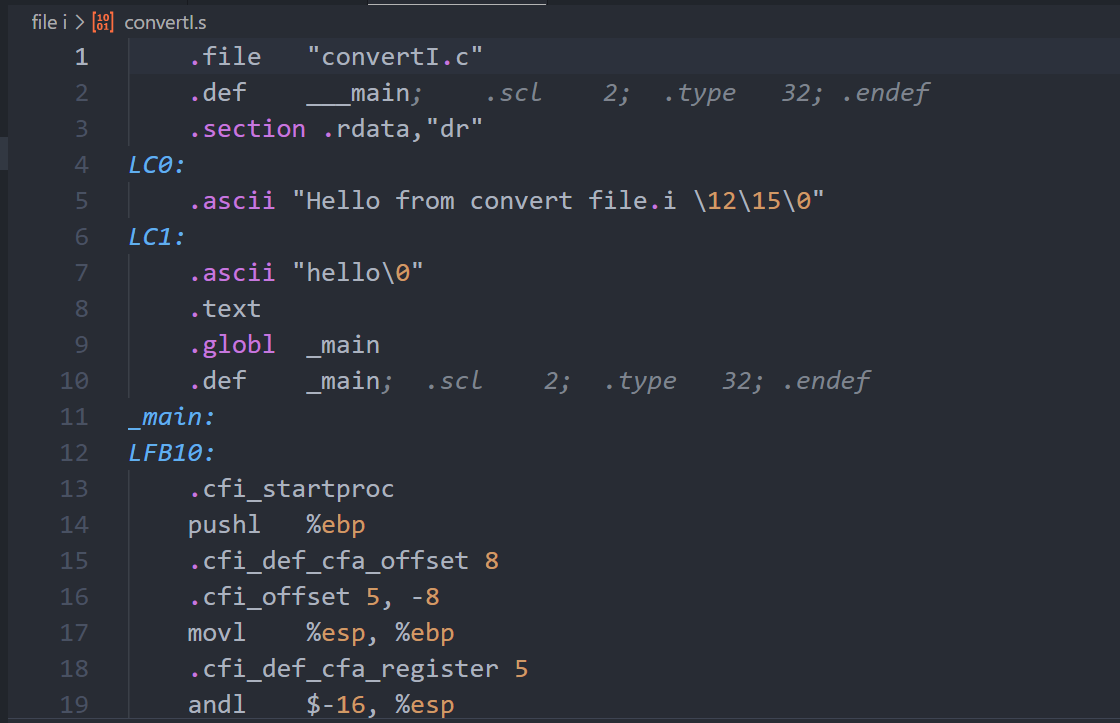


Chuyển sang file .i ( cmd: gcc -E convertI.c -o convertI.i )

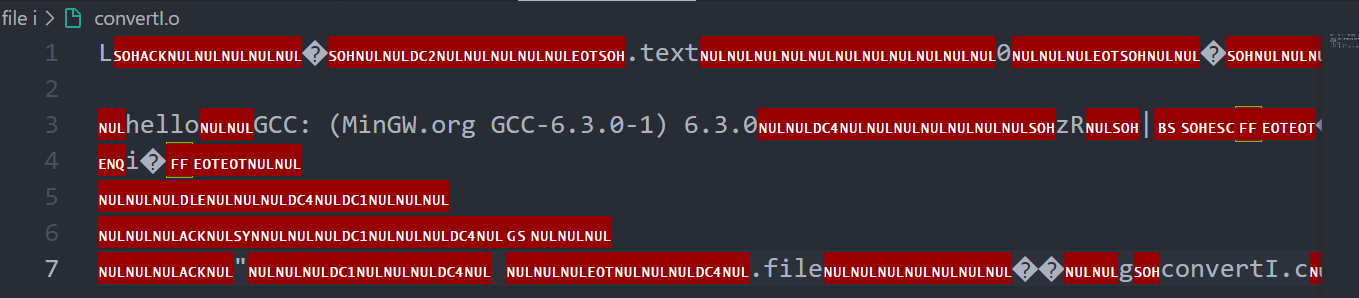




Chuyển sang file .s (cmd gcc -S convertI.i -o convertI.s)



Chuyển sang mã máy (cmd as convertI.s -o convertI.o)

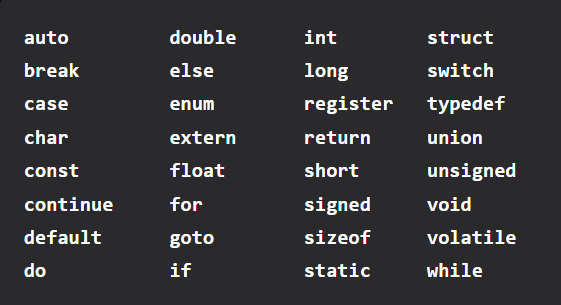


Reference: [Thử thực hiện 4 Stage khi Compile C bằng GCC | TopDev](https://topdev.vn/blog/thu-thuc-hien-4-stage-khi-compile-c-bang-gcc/)

## Token

  
  
Token trong C gồm :

### Keywords (từ khóa)

* + Mỗi từ khóa có nghĩa là để thực hiện một chức năng cụ thể trong một chương trình.
  + 

### Identifiers (Định danh)

* + Định danh được sử dụng làm thuật ngữ chung để đặt tên biến, hàm và mảng.
  + Ví dụ,
    - main: Hàm.
    - a: Biến.

### Constants (Hằng)

* + const int c\_var = 20;
  + const int\* const ptr = &c\_var;

### Strings (Chuỗi)

* + Chuỗi làm một mảng kí tự nhưng chứa kí tự kết thúc “/0”.
    - char string[20] = {‘g’, ’e’, ‘e’, ‘k’, ‘k’, ‘s’,‘\0’};
    - char string[20] = “geeksforgeeks”;
    - char string [] = “geeksforgeeks”;

### Special Symbols (Kí tự đặc biệt)

* + Brackets[]. /ˈbrækɪt/
  + Parentheses() /pəˈrenθəsɪs/
  + Braces{} /breɪs/
  + Comma (, ) /ˈkɒmə/
  + Colon (:) /ˈkəʊlən/
  + Semicolon(;) /ˌsemiˈkəʊlən/
  + Asterisk (\*) /ˈæstərɪsk/
  + Pre-processor (#)
  + Period (.) /ˈpɪəriəd/
  + Tilde(~) /ˈtɪldə/

### Operators (Toán tử)

# Phần 2. C Variables and Constants

## 2.1. Variable (Biến)

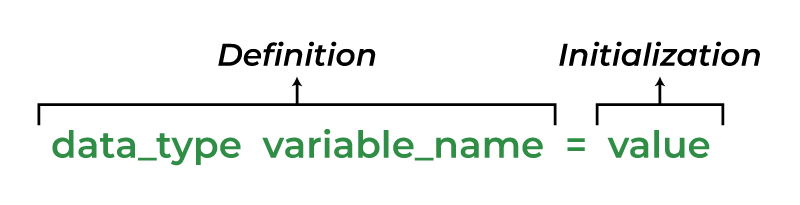
### Định nghĩa

* Một biến trong C là gán một địa chỉ bằng tên để dễ dàng truy cập và lưu trữ.



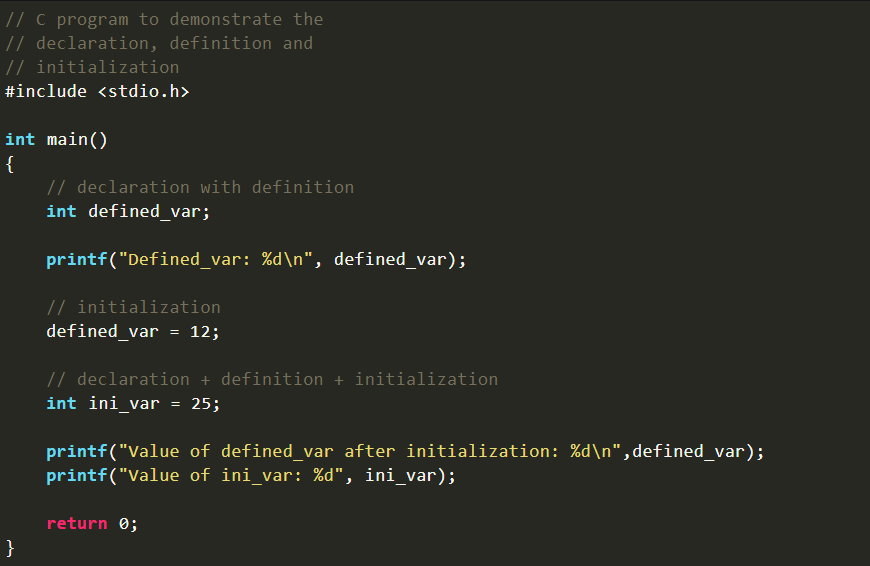
### C Variable Syntax (Cú phát khái báo biến)

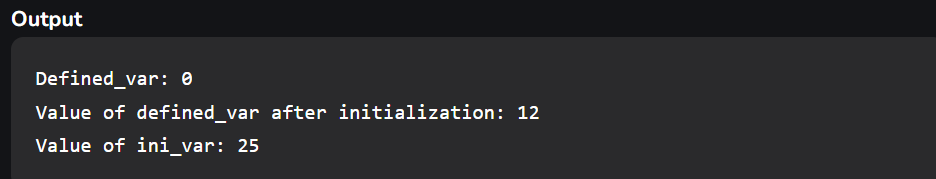
* data\_type variable\_name = value;
* data\_type variable\_name1, variable\_name2;
* int a = 5;
* int a,b,c;
  + data\_type: Kiểu dữ liệu (int , float ..).
  + variable\_name: tên biến .
  + value:Giá trị gán vào .



Có 3 dạng định nghĩa biến trong C:

* Variable Declaration
  + Hay còn được gọi là khai báo biến. Có nghĩa báo cho trình biên dịch biết về sự tồn tại của biến với tên và kiểu dữ liệu đã cho. Không có bộ nhớ nào được cấp phát cho một biến trong phần khai báo
* Variable Definition ( Định nghĩa )
  + Giống như khai báo biến như khác biệt là đã cấp phát bộ nhớ (tức vị trí cho biến đó ).
  + Note: Hầu hết ở C thì việc chúng ta khai báo và khởi tạo sẽ là 1 cú pháp.
  + Ví dụ:
    - int var;
    - char var2;
* Variable Initialization ( Gán giá trị )
  + Gán giá trị cho biến đó
  + Ví dụ:
    - int var; // Định nghĩa biến ( hay cũng có thể gọi là khai báo vì với biến thì khai báo và định nghĩa là 1)
    - var = 10; // initialization
    - int var = 10; // variable declaration and definition.

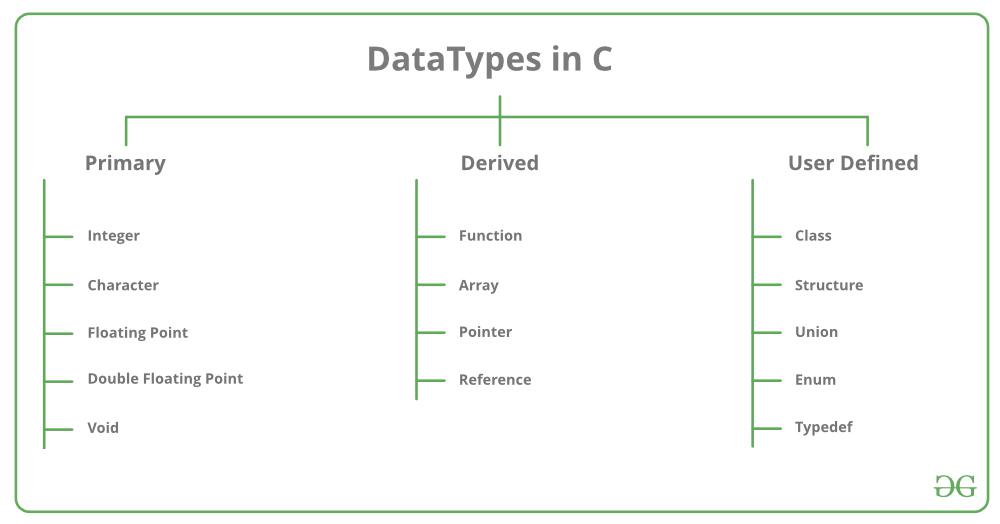




## 2.2. Basic Data Types (Kiểu dữ liệu).

### Overview:

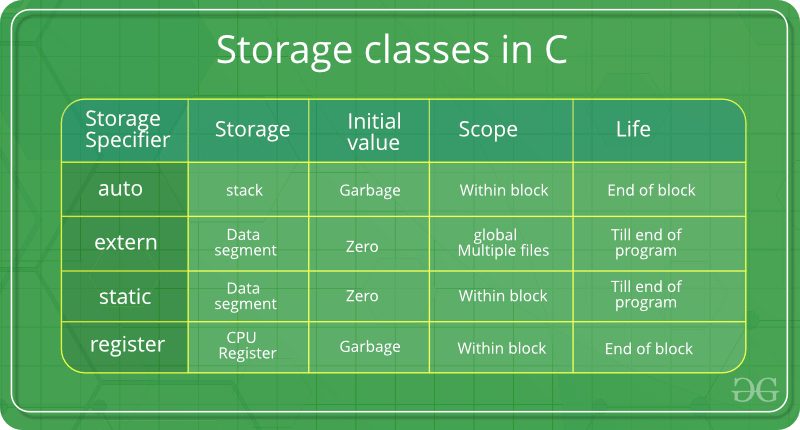
* Primitive Data Types (Kiểu dữ liệu nguyên thủy) .
* User Defined Data Types (Kiểu dữ liệu người dùng tự định nghĩa)
* Derived Types (Kiểu dữ liệu mở rộng)



## 2.3. Storage Classes in C (Lớp lưu trữ)

### Định nghĩa

Lớp lưu trữ C được sử dụng để mô tả các tính năng của một biến/hàm. Các tính năng này về cơ bản bao gồm phạm vi, khả năng hiển thị và tuổi thọ giúp theo dõi sự tồn tại của một biến cụ thể trong thời gian chạy của chương trình.



Scope (Phạm vi truy cập của biến);

Lifetime (Thời gian tồn tại của biến);

### Scope:

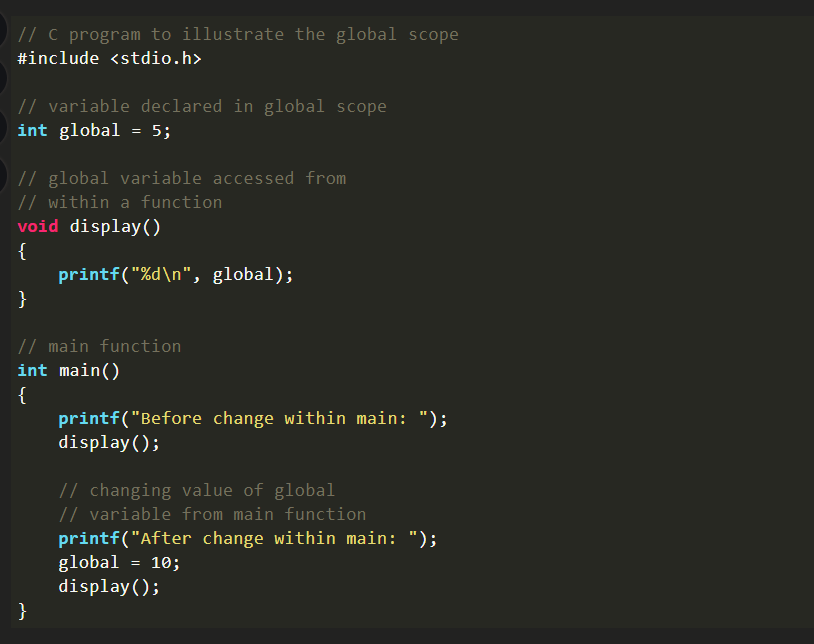
#### Định nghĩa

* Phạm vi của biến trong C là khối hoặc vùng trong chương trình nơi biến được khai báo, định nghĩa và sử dụng. Bên ngoài khu vực này, không thể truy cập vào biến và nó được coi là một biến không được khai báo.
* Phạm vi là khu vực mà một biến có thể truy cập được.
* Chúng ta chỉ có thể tham chiếu đến một biến trong phạm vi của nó.

#### Các loại phạm vi truy cập của biến:

1. Global Scope (Toàn cục)
2. Local Scope (Cục bộ)
3. Global Scope in C:

* Phạm vi toàn cục đề cập đến khu vực bên ngoài bất kỳ khối ( khối là {} ) hoặc function ( hàm ) nào.
* Các biến toàn cục có thể nhìn thấy trong mọi phần của chương trình.
* Toàn cục còn được gọi là phạm vi file vì phạm vi của một biến bắt đầu ở đầu file và kết thúc ở cuối file .



* Biến global được gọi là biến toàn cục :
  1. Biến được khai báo ngoài phạm vi các hàm.
  2. Biến này được truy cập từ bất kì nơi nào ( tức Scope – phạm vi truy cập của biến là toàn bộ file này ).
* Linkage of Variables in Global Scope ( Phạm vi các bến toàn cục có extern theo mặc định. Nó có nghĩa là các biến được khai báo trong phạm vi toàn cục có thể được truy cập trong một tệp nguồn C khác. Chúng ta phải sử dụng từ khóa extern cho mục đích đó).
* Ưu điểm của biến toàn cầu
  1. Các biến toàn cục có thể được truy cập bởi tất cả các hàm có trong chương trình.
  2. Chỉ cần khai báo một lần.
  3. Biến toàn cục rất hữu ích nếu tất cả các hàm đang truy cập cùng một dữ liệu.
* Nhược điểm của biến toàn cầu
  1. Giá trị của một biến toàn cục có thể được thay đổi vô tình vì nó có thể được sử dụng bởi bất kỳ hàm nào trong chương trình.
  2. Nếu chúng ta sử dụng một số lượng lớn các biến toàn cục, thì có khả năng cao tạo ra lỗi trong chương trình

1. Local Scope (cục bộ)

* Phạm vi cục bộ đề cập đến khu vực bên trong một khối hoặc một hàm. Đó là khoảng trống được bao bọc giữa { }.
* Các biến được khai báo trong phạm vi cục bộ được gọi là biến cục bộ.
* Các biến cục bộ có thể nhìn thấy trong khối mà chúng được khai báo và các khối khác lồng bên trong khối đó.
* Reference : [Scope rules in C - GeeksforGeeks](https://www.geeksforgeeks.org/scope-rules-in-c/)

### Giải thích Store Class

#### Auto

Đây là lớp lưu trữ mặc định cho tất cả các biến được khai báo bên trong một hàm hoặc một khối. Do đó, từ khóa auto hiếm khi được sử dụng trong khi viết chương trình bằng ngôn ngữ C. Các biến auto chỉ có thể được truy cập trong khối / hàm mà chúng đã được khai báo chứ không phải bên ngoài chúng (xác định phạm vi của chúng). Tất nhiên, chúng có thể được truy cập trong các khối lồng nhau trong khối / hàm cha trong đó biến tự động được khai báo.

#### Extern

Lớp lưu trữ extern chỉ đơn giản cho chúng ta biết rằng biến được định nghĩa ở nơi khác và có thể gọi trong một file khác trong cùng folder của nó

Ngoài ra, một biến toàn cục bình thường cũng có thể được tạo ra bên ngoài bằng cách đặt từ khóa 'extern' trước khai báo / định nghĩa của nó trong bất kỳ hàm / khối nào. Điều này về cơ bản biểu thị rằng chúng ta không khởi tạo một biến mới mà thay vào đó, chúng ta chỉ sử dụng / truy cập biến toàn cục. Mục đích chính của việc sử dụng các biến extern là chúng có thể được truy cập giữa hai tệp khác nhau là một phần của một chương trình lớn.

#### Static

Lớp lưu trữ này được sử dụng để khai báo các biến tĩnh được sử dụng phổ biến trong khi viết chương trình bằng ngôn ngữ C. Các biến tĩnh có đặc tính bảo toàn giá trị của chúng ngay cả sau khi chúng nằm ngoài phạm vi của chúng! Do đó, các biến tĩnh bảo toàn giá trị của lần sử dụng cuối cùng trong phạm vi của chúng. Vì vậy, chúng ta có thể nói rằng chúng chỉ được khởi tạo một lần và tồn tại cho đến khi chấm dứt chương trình. Do đó, không có bộ nhớ mới nào được phân bổ vì chúng không được khai báo lại.

Phạm vi của chúng là cục bộ cho chức năng mà chúng được xác định. Các biến tĩnh toàn cục có thể được truy cập ở bất cứ đâu trong chương trình. Theo mặc định, chúng được trình biên dịch gán giá trị 0.

#### Register

Lớp lưu trữ này khai báo các biến thanh ghi có chức năng tương tự như các biến auto. Sự khác biệt duy nhất là trình biên dịch cố gắng lưu trữ các biến này trong thanh ghi của bộ vi xử lý nếu có sẵn thanh ghi miễn phí. Điều này làm cho việc sử dụng các biến thanh ghi nhanh hơn nhiều so với các biến được lưu trữ trong bộ nhớ trong thời gian chạy của chương trình.

Thông thường, một vài biến được truy cập rất thường xuyên trong một chương trình được khai báo với từ khóa đăng ký giúp cải thiện thời gian chạy của chương trình. Một điểm quan trọng và thú vị cần lưu ý ở đây là chúng ta không thể lấy địa chỉ của một biến thanh ghi bằng cách sử dụng con trỏ.

Cú pháp

storage\_class var\_data\_type var\_name;

extern int a;

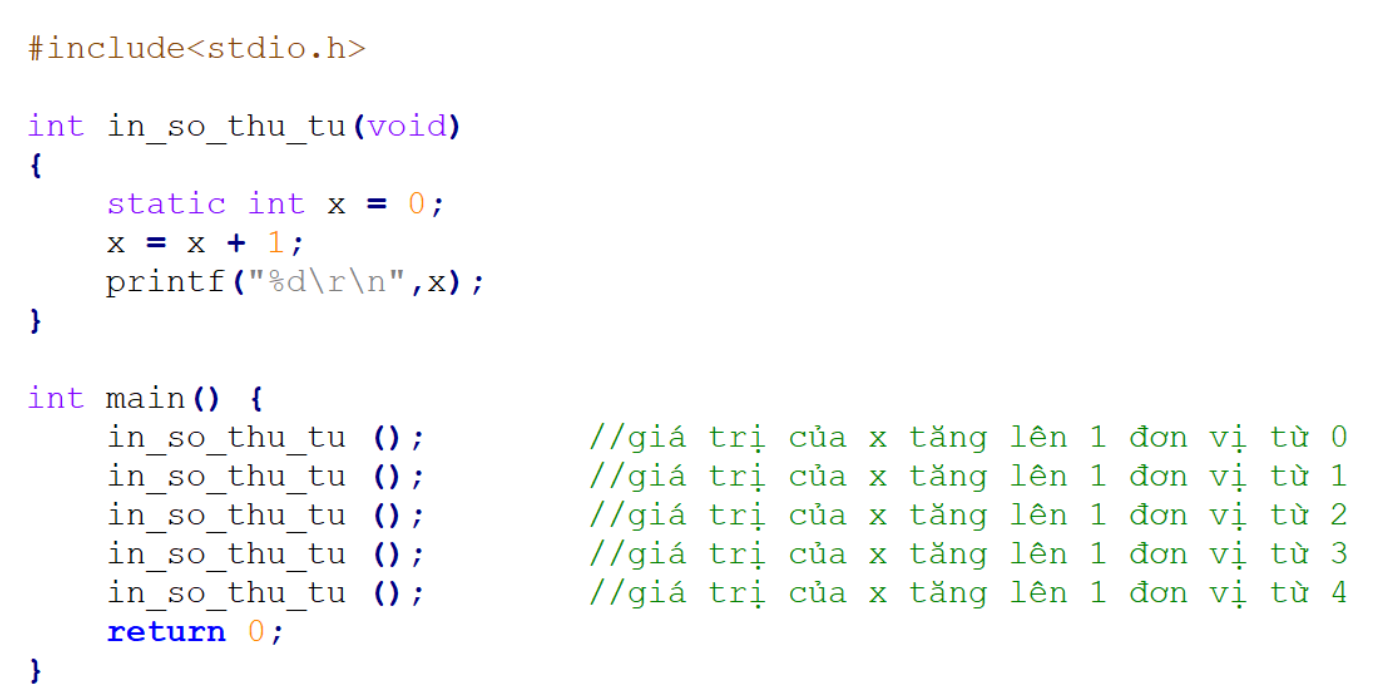
References: Storage Classes in C - GeeksforGeeks

## Key word with variable (Từ khóa)

### Static:

* Biến static

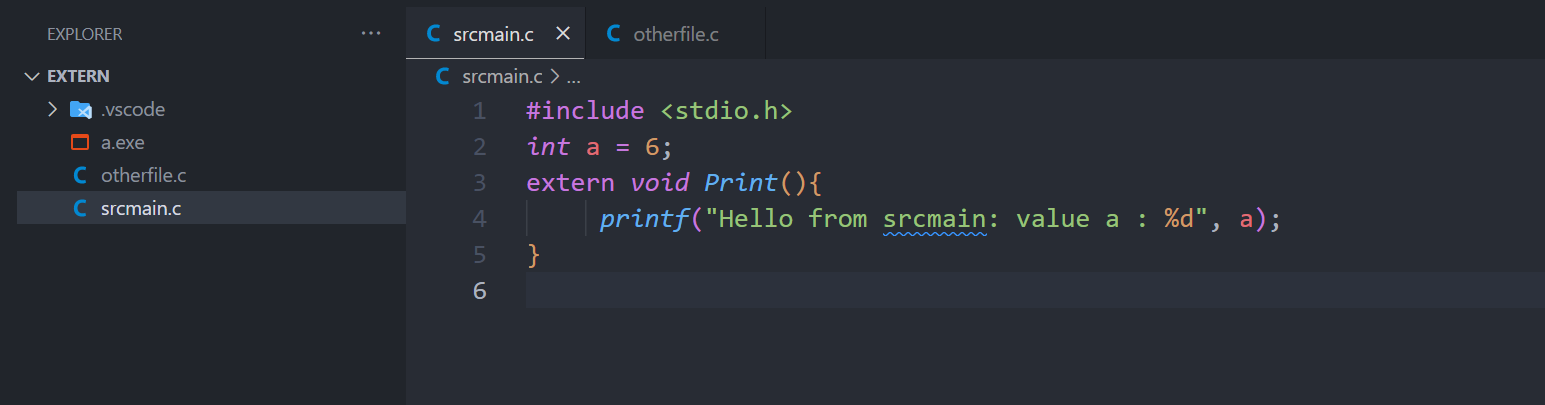
1. Biến static trong khai báo biến cục bộ.
   * Được khởi tạo 1 lần duy nhất trong cả chương trình.
   * Giá trị được giữ nguyên cả khi ra khỏi hàm, được lưu vào vùng nhớ Data Segment.
   * Chỉ được sử dụng bên trong hàm khai báo nó.
2. Biến static trong khai báo biến toàn cục
   * Sử dụng tương tự như biến toàn cục.
   * Chỉ được sử dụng trong File khai báo nó, các
   * File khác không được truy cập.
3. Biến static trong khai báo hàm
   * Chỉ được sử dụng trong File khai báo nó, các
   * File khác không được truy cập.



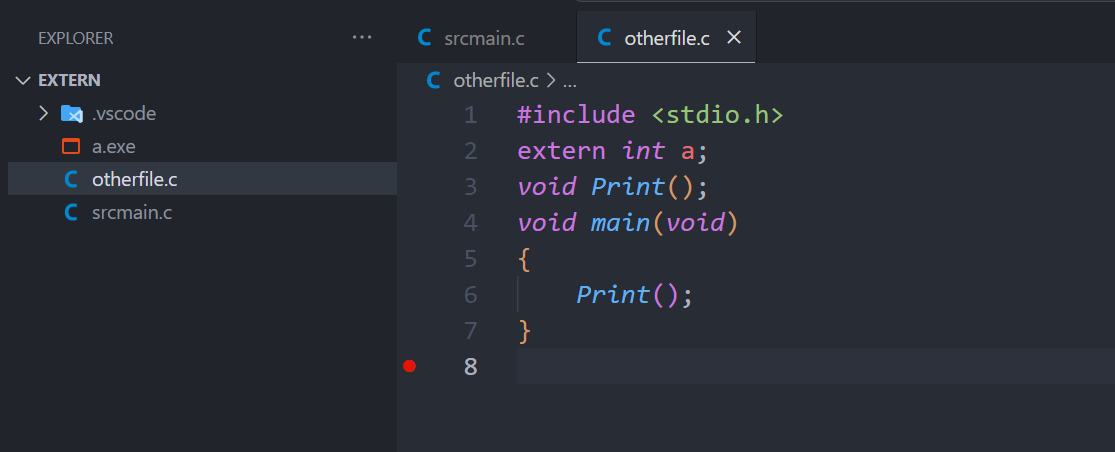
### Extern:

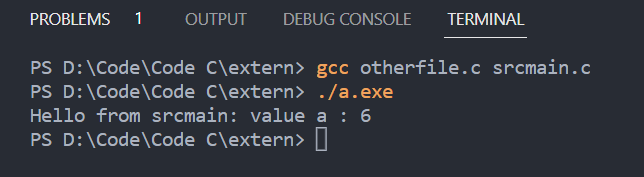
Để có thể truy cập giá trị một biến toàn cục, một mảng, hoặc một hàm ở một File khác, chúng ta sử dụng từ khóa extern.

Ví dụ:



otherfile.c





### Register:

(Đọc lại Store Class).

#### Volatile:

Từ khóa Volatile nhằm ngăn trình biên dịch áp dụng bất kỳ tối ưu hóa nào trên các đối tượng có thể thay đổi theo cách mà trình biên dịch không thể xác định được.

# Phần 3. Structure data type (Cấu trúc dữ liệu ):

1. Structure variable
2. Union
3. Enum

## Structure

### Định nghĩa

* + - Là một kiểu dữ liệu tự định nghĩa bao gồm nhiều thành phần có thể thuộc nhiều kiểu dữ liệu khác nhau cùng liên quan tới một đối tượng.

### Khai báo

* Các thành viên cấu trúc không thể được khởi tạo với khai báo.
* Ví dụ: chương trình C sau đây bị lỗi trong quá trình biên dịch

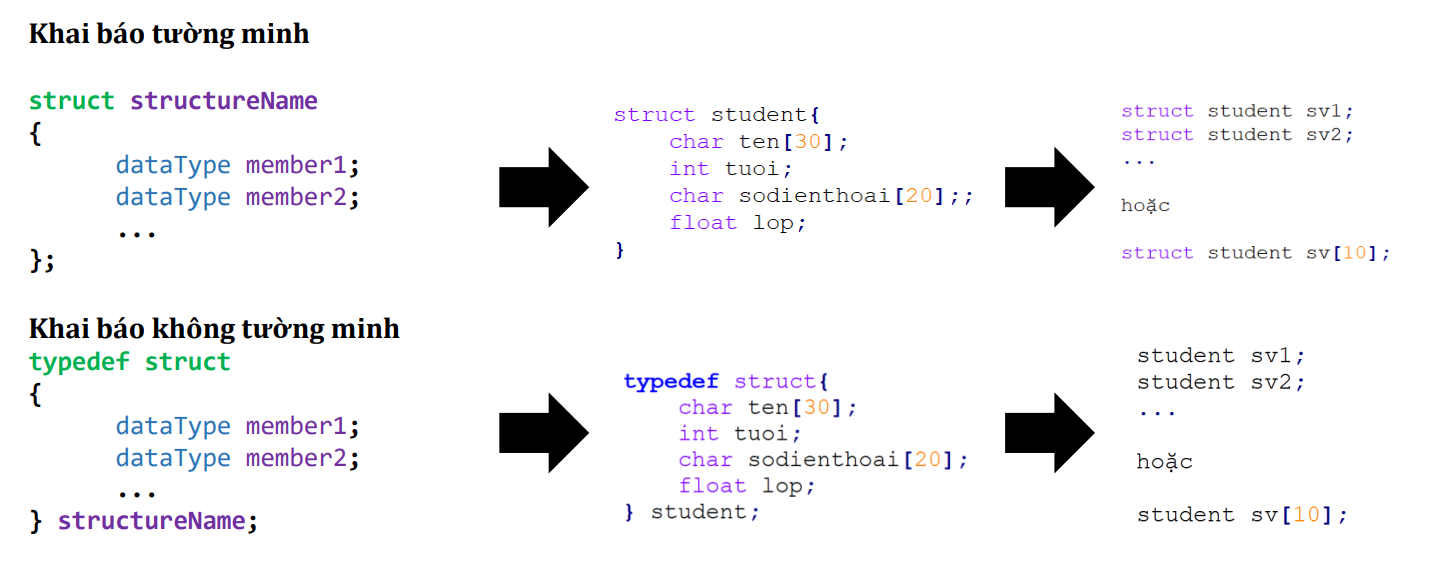
struct Point

{

int x = 0; *// COMPILER ERROR: cannot initialize members here*

int y = 0; *// COMPILER ERROR: cannot initialize members here*

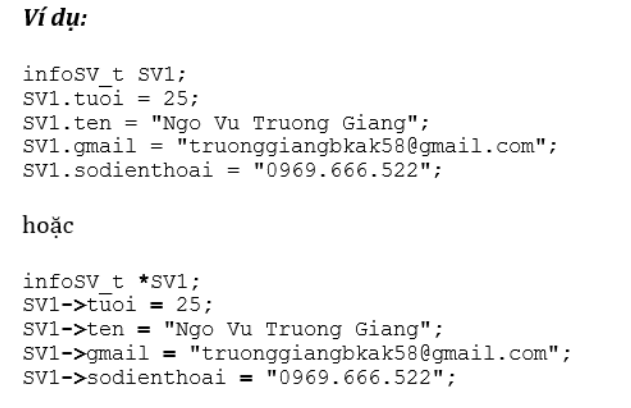
};



### Truy xuất dữ liệu

#### Cú pháp khai báo

* Sử dụng “.” Toán tử truy xuất tới thành viên khi khai báo biến bình thường.
* Sử dụng “→ Toán tử truy xuất tới thành viên khi biến là con trỏ.



### C Structure Padding and Packing.

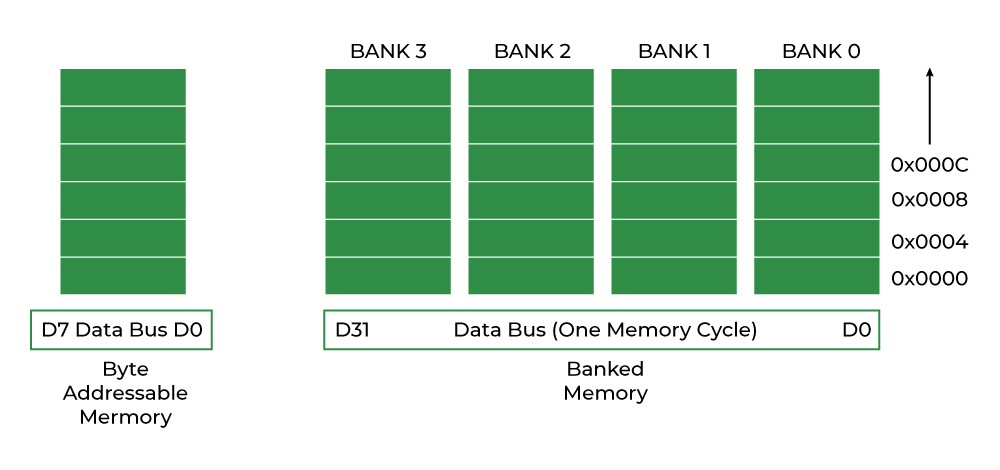
Về mặt kỹ thuật, kích thước của cấu trúc trong C phải là tổng kích thước của các thành viên của nó. Nhưng nó có thể không đúng với hầu hết các trường hợp. Lý do cho điều này là Structure Padding.

Structure Padding là khái niệm thêm nhiều byte trống trong cấu trúc để căn chỉnh tự nhiên các thành viên dữ liệu trong bộ nhớ. Nó được thực hiện để giảm thiểu chu kỳ đọc CPU để truy xuất các thành viên dữ liệu khác nhau trong cấu trúc

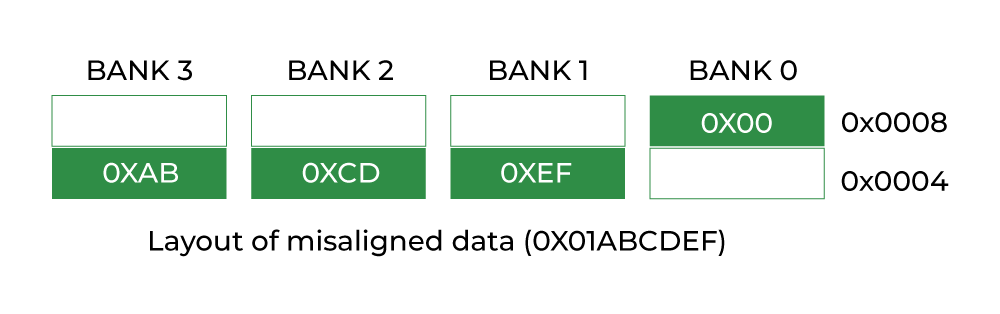
1. Using #pragma pack(1)
2. Using \_\_attribute((packed))\_\_

#### Structure Alignment (căn chỉnh bộ nhớ ):

Mỗi kiểu dữ liệu trong C sẽ có các yêu cầu alignment (trên thực tế, nó được ủy quyền bởi kiến trúc bộ xử lý, không phải theo ngôn ngữ). Một bộ xử lý sẽ có độ dài từ xử lý như kích thước bus dữ liệu. Trên máy 32 bit, kích thước từ xử lý sẽ là 4 byte ( 32/8[bits] = 4) .



Có thể hiểu một chu kì máy thì CPU có thể đọc được 4 bytes (4 banks).

Nếu một int 4 byte được phân bổ trên địa chỉ X (X là bội số của 4), bộ xử lý chỉ cần một chu kỳ bộ nhớ để đọc toàn bộ int . Trong khi đó, nếu số nguyên được phân bổ tại một địa chỉ khác với bội số của 4, nó trải dài trên hai hàng của các banks thể hiện trong hình dưới đây. Một int như vậy đòi hỏi hai chu kỳ đọc bộ nhớ để lấy dữ liệu. 

Tương tự, căn chỉnh int của một số short int là 2 byte. Nó có nghĩa là một số short int có thể được lưu trữ trong cặp bank 0 - bank 1 hoặc bank 2 - bank 3 ( bội số của 2). Một double yêu cầu 8 byte và chiếm hai hàng trong bank bộ nhớ. Bất kỳ sự sai lệch nào của double sẽ buộc nhiều hơn hai chu kỳ đọc để lấy dữ liệu kép.

|  |  |
| --- | --- |
|  | Những gì bạn tính là 1+4+8=13 bytes, nhưng kết quả lại là 16, lạ phải không nào?  Đầu tiên compiler sẽ tìm xem trong struct, thành viên nào có kích thước lớn nhất, sau đó sẽ cấp phát một block có kích thước tương ứng với member lớn nhất, rồi điền các vùng nhớ của các members theo thứ tự khai báo trong struct, khi nào hết một block, sẽ cấp thêm block mới.  Cụ thể với ví dụ trên: Compiler sẽ xác định member gpa kiểu double là thành viên có kích thước lớn nhất trong struct, kích thước là 8 bytes.  Sau đó, compiler sẽ đẩy 1 byte của id vào block. Lúc này còn 7 bytes trống, compiler sẽ xác định thành viên có kích thước bé tiếp theo rồi đẩy vào, ở đây sẽ là 4 bytes của age, compiler sẽ padding 3 bytes và đẩy age vào 4 bytes còn lại.  Vậy là hết block đầu tiên, còn thành viên gpa chưa được cấp bộ nhớ, do đó compiler sẽ cấp phát tiếp 1 block 8 bytes rồi đẩy gpa vào 8 bytes đó |

Để có thể làm được việc đó chúng ta cần hiểu 2 khái niệm sau:

1. Data alignment: sắp xếp data sao cho địa chỉ của các biến luôn là số chẵn và phù hợp với hệ thống ( Địa chỉ của biến sẽ chia hết cho kích thước của biến đó).
2. Data padding: để làm được việc alignment như ở trên chúng ta cần phải “padding” (đệm) thêm một số byte vào sau biến “char id” để khi đó biến “int age” có thể ở địa chỉ chẵn. char có thể bắt đầu trên bất cứ byte địa chỉ nào, short-2 bytes chỉ bắt đầu bằng các bytes địa chỉ chẵn, int-4 bytes hoặc float-4 bytes bắt đầu tại các bytes địa chỉ chia hết cho 4, long-8 bytes hoặc double-8 bytes bắt đầu tại các byte địa chỉ chia hết cho 8. Không có sự khác biệt giữa các kiểu có dấu và không dấu

Ví dụ

*// C program to illustrate structure padding and packing*

#include <stdio.h>

*// structure with padding*

struct str1 {

char c;

int i;

};

struct str2 {

char c;

int i;

} \_\_attribute((packed)) \_\_; *// using structure packing*

*// driver code*

int main()

{

printf("Size of str1: %d\n", sizeof(struct str1));

printf("Size of str2: %d\n", sizeof(struct str2));

return 0;

}

Size of str1: 8.

Size of str2: 5.

#### Ưu điểm :

* Truy cập nhanh.

#### Nhược điểm:

* Do phải padding ( thêm các byte,bit vào nên tăng kích thước code )

Reference: [Data structure alignment - Viblo](https://viblo.asia/p/data-structure-alignment-Az45bjag5xY), [Structure Padding in C - javatpoint](https://www.javatpoint.com/structure-padding-in-c)

### Bit Field:

Bit Field được sử dụng để chỉ định độ dài của các thành viên cấu trúc theo bit. Khi chúng ta biết độ dài tối đa của thành viên, chúng ta có thể sử dụng các trường bit để chỉ định kích thước và giảm mức tiêu thụ bộ nhớ.

Cú pháp

struct structure\_name {

data\_type member\_name: width\_of\_bit-field;

};

*// C Program to illustrate bit fields in structures*

#include <stdio.h>

*// declaring structure for reference*

struct str1 {

int a;

char c;

};

*// structure with bit fields*

struct str2 {

int a : 24; *// size of 'a' is 3 bytes = 24 bits*

char c;

};

*// driver code*

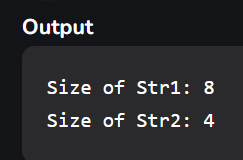
int main()

{

printf("Size of Str1: %d\nSize of Str2: %d",sizeof(struct str1), sizeof(struct str2));

return 0;

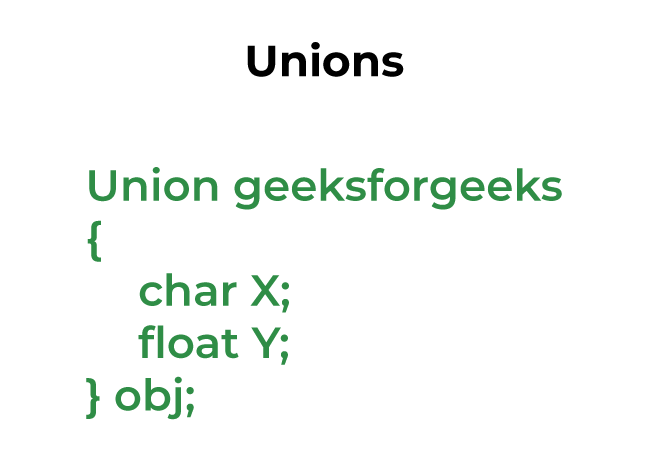
}



## Union

### Định nghĩa

* Union là một kiểu dữ liệu người dùng tự định nghĩa
* Tất cả các thành viên trong union C được lưu trữ trong cùng một vị trí bộ nhớ. Do đó, chỉ có một thành viên có thể lưu trữ dữ liệu tại thời điểm nhất định ( kích thước của union sẽ là biến có giá trị bytes cao nhất )



Định nghĩa Union Variable với khai báo

union union\_name {

datatype member1;

datatype member2;

...

} var1, var2, ...;

Định nghĩa Union Variable sau khai báo

union union\_name var1,var2;

### Size of Union

Kích thước của union là kích thước của phần tử có kích thước lớn nhất

Ví dụ:

union union\_name {

int member1;

char member2;

...

} var1, var2, ...;

Kích thước của union trên là 4 vì int là kiểu dữ liệu có kích thước lớn nhất.

### Difference between C Structure and C Union

|  |  |
| --- | --- |
| Struct | Union |
| Kích thước của Struct bằng hoặc lớn hơn tổng kích thước của tất cả các thành viên của nó | Quy mô của union là size của thành viên lớn nhất của nó. |
| Struct có thể chứa dữ liệu trong nhiều thành viên cùng một lúc. | Chỉ một thành viên có thể chứa dữ liệu cùng một lúc. |
| Nó được khai báo bằng cách sử dụng từ khóa struct. | Nó được khai báo bằng cách sử dụng từ khóa union |

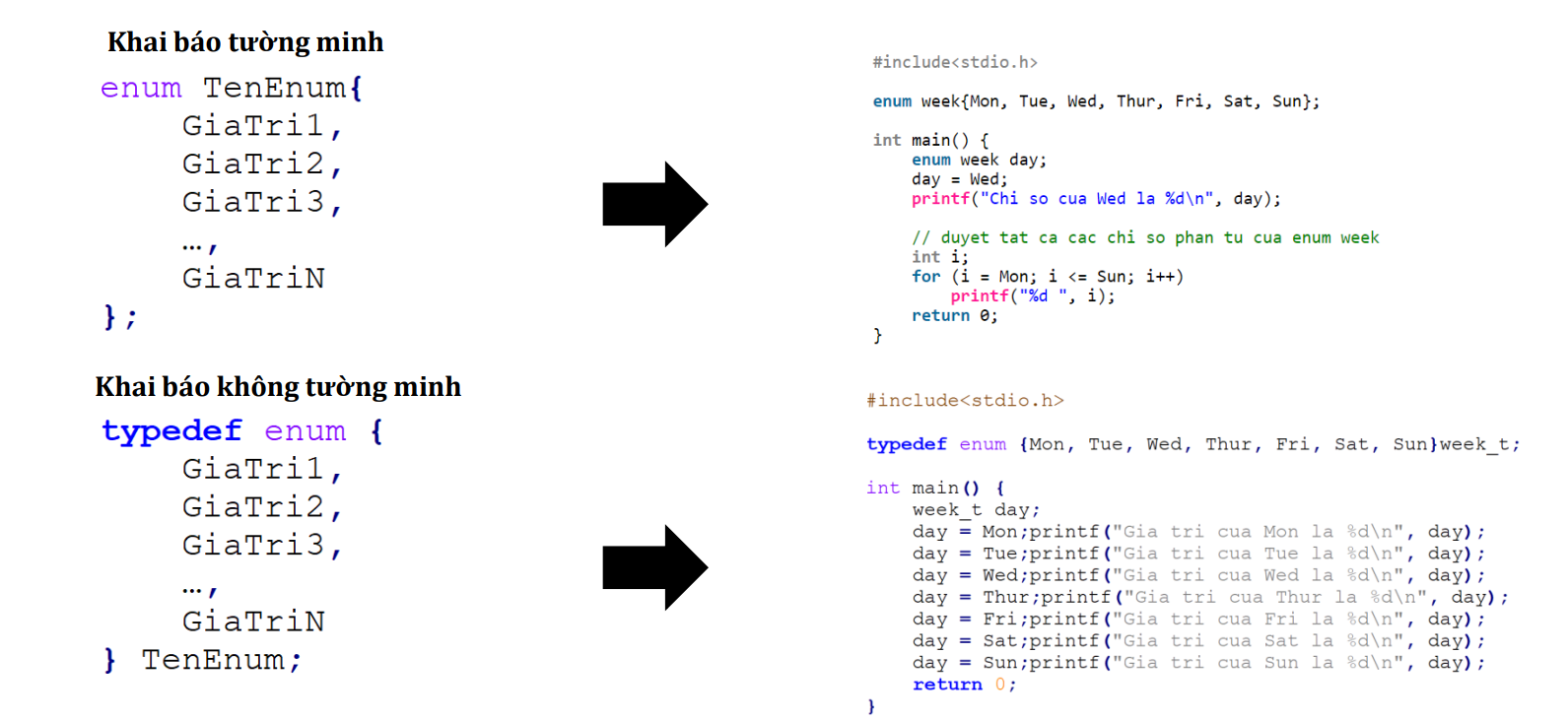
## Enum

### Định nghĩa

Enum là 1 kiểu dữ liệu người dùng tự định nghĩa. Dùng để gán tên cho các hằng số. Được sử dụng để bạn có thể kiểm soát các hằng số không muốn bị nhầm lẫn trong quá trình sử dụng. Enum được ứng dụng để định nghĩa các mã lỗi, các mã trả về, các biến hằng cần kiểm soát.

### Size of Enum:

Có thể là int hoặc long int tùy theo hằng số mà người dùng gán.



### Note:

* Chúng ta cũng có thể tự do định nghĩa giá trị bất kỳ cho các hằng số.
* Nếu chúng ta không gán giá trị cho tên enum một cách rõ ràng, trình biên dịch theo mặc định gán giá trị bắt đầu từ 0.
* Chúng ta có thể gán 1 giá trị bất kỳ cho một phần tử của enum. Tất cả các tên chưa được chỉ định nhận giá trị là giá trị của tên trước đó cộng với một.

# Phần 4. C Pointers

## 4.1. Định nghĩa

Con trỏ được định nghĩa là một kiểu dữ liệu dẫn xuất có thể lưu trữ địa chỉ của các biến C khác hoặc vị trí bộ nhớ. Chúng ta có thể truy cập và thao tác dữ liệu được lưu trữ trong vị trí bộ nhớ đó bằng cách sử dụng con trỏ.

### 4.1.1. Size (kích thước) :

Khi con trỏ lưu trữ địa chỉ bộ nhớ, kích thước của chúng độc lập với loại dữ liệu mà chúng đang trỏ đến. Kích thước con trỏ này chỉ phụ thuộc vào kiến trúc hệ thống.

### 4.1.2. Syntax(Cú pháp):

datatype \* ptr;

## 4.2. How to Use Pointers ( Cách dùng ) ?

Chia thành 3 bước:

1. Pointer Declaration ( Khai báo)
2. Pointer Initialization (Gán giá trị)
3. Dereferencing (Tham chiếu)

### 4.2.1. Pointer Declaration:

Trong khai báo con trỏ, chúng ta chỉ khai báo con trỏ chứ không khởi tạo nó. Để khai báo một con trỏ, chúng ta sử dụng toán tử dereference ( \* ) trước tên của nó.

#### Ví dụ

int \*ptr;

Con trỏ được khai báo ở đây sẽ trỏ đến một số địa chỉ bộ nhớ ngẫu nhiên vì nó không được khởi tạo. Những con trỏ như vậy được gọi là con tr NULL.

### 4.2.2. Pointer Initialization:

Khởi tạo con trỏ là quá trình mà chúng ta gán một số giá trị ban đầu cho biến con trỏ. Chúng ta thường sử dụng toán tử ( & ) addressof để lấy địa chỉ bộ nhớ của một biến và sau đó lưu trữ nó trong biến con trỏ.

Ví dụ

int var = 10;

int \* ptr;

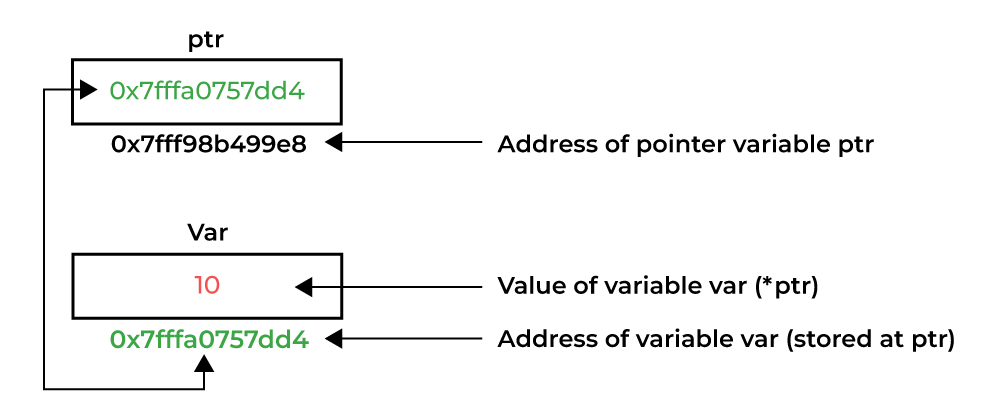
ptr = &var;

Chúng ta cũng có thể khai báo và khởi tạo con trỏ trong một bước duy nhất. Phương thức này được gọi là định nghĩa con trỏ vì con trỏ được khai báo và khởi tạo cùng một lúc.

Ví dụ

int \* ptr = &var;

### 4.2.3. Dereferencing:

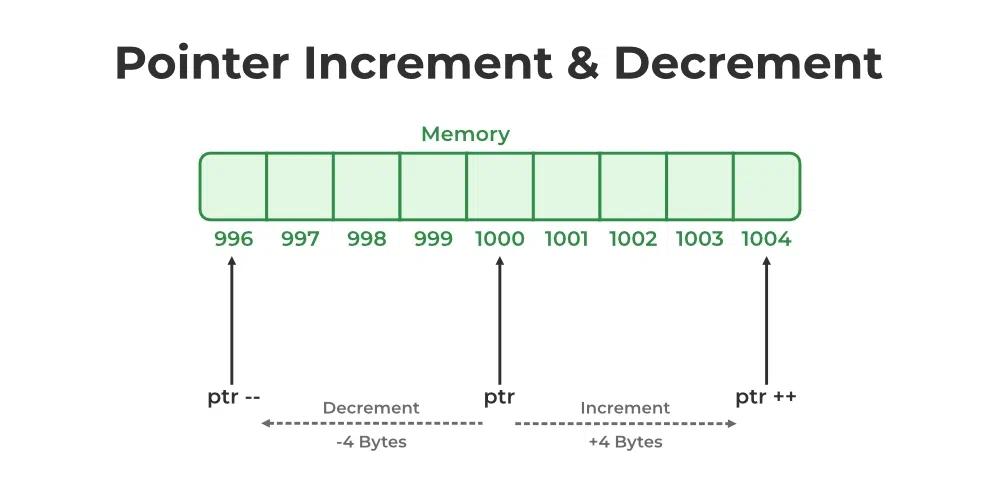


Dereferencing con trỏ là quá trình truy cập giá trị được lưu trữ trong địa chỉ bộ nhớ được chỉ định trong con trỏ. Chúng ta sử dụng cùng một toán tử dereferencing ( \*) mà chúng ta đã sử dụng trong khai báo con trỏ.

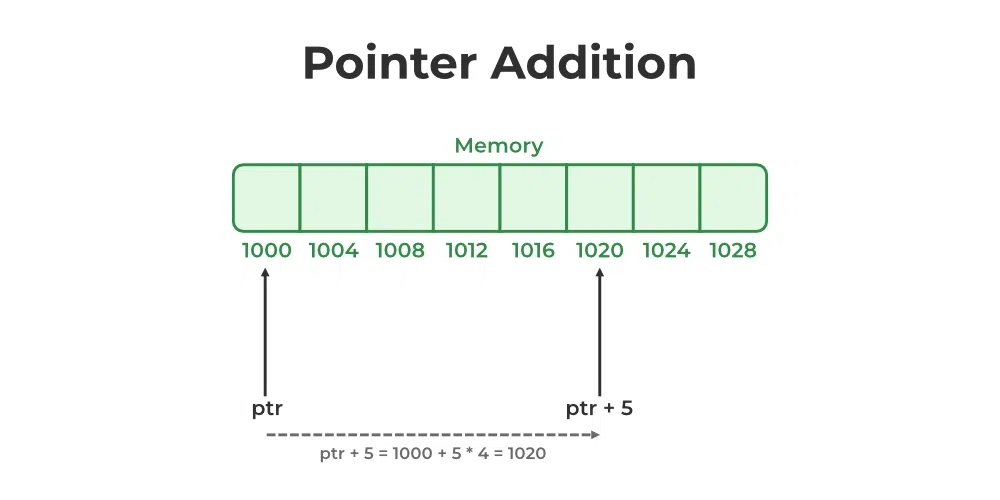
### 4.2.4. Pointer Arithmetics (Toán tử)

1. Increment/Decrement of a Pointer
2. Addition of integer to a pointer
3. Subtraction of integer to a pointer
4. Subtracting two pointers of the same type
5. Comparison of pointers of the same type.

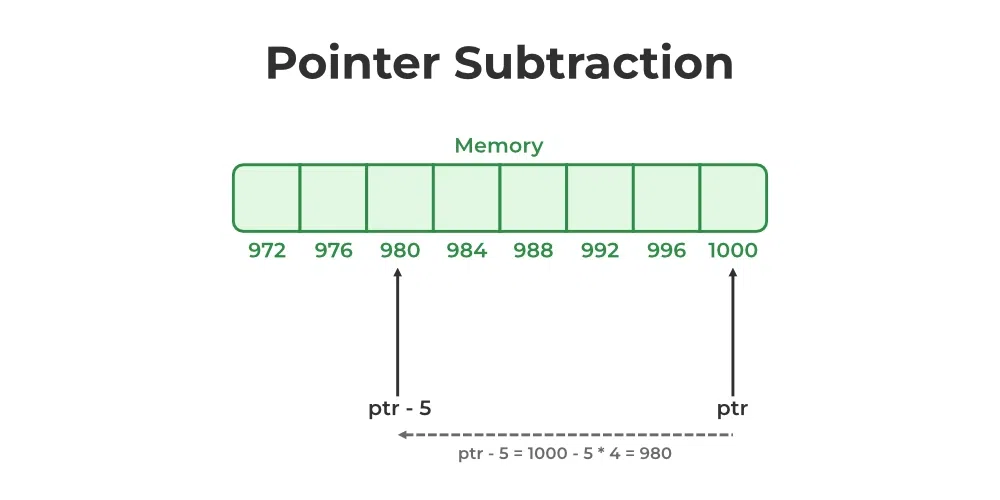
#### Increment/Decrement of a Pointer



#### Addition of integer to a pointer:



#### Subtraction of Integer to Pointer



## 4.3. Types of Pointers:

1. Integer Pointers:
2. Array Pointer:
3. Structure Pointer: (Xem lại ở bài Structure).
4. Function Pointers:
5. Double Pointers:
6. NULL Pointer
7. Void Pointer
8. Wild Pointers
9. Constant Pointers
10. Pointer to Constant

Integer Pointers:

* + - Syntax
      * int \*ptr;

### 4.3.1. Array Pointer:

Con trỏ và Mảng có liên quan chặt chẽ với nhau. Ngay cả tên mảng cũng là con trỏ đến phần tử đầu tiên của nó. Chúng còn được gọi là Pointer to Arrays. Chúng ta có thể tạo một con trỏ đến một mảng bằng cú pháp đã cho.

char \*ptr = &array\_name;

Ví dụ:

#include<stdio.h>

int main()

{

int arr[5] = { 1, 2, 3, 4, 5 };

int \*ptr = arr;

printf("%p\n", ptr);

return 0;

}

#### Note:

* + - * Tên của mảng sẽ là con trỏ trỏ tới địa chỉ đầu tiên của mảng.

#### Different between int \* p and int \*ptr[];

* Với int \*p nó trỏ tới phần tử đầu tiên của mảng . Các phép toán tử ++ hay – thì nó sẽ truy cập tới phần tử tiếp theo.
* Với int \* ptr[5] nó cũng trỏ tới phần tử đầu tiên của mảng nó được gán nhưng có sự khác biệt là khi dùng toán tử ++ hay – thì phần tử tiếp theo nó truy cập không phải là ptr[1] mà là ptr[6]

#include<stdio.h>

*int* *main*()

{

*int* \*p;

*int* (\*ptr)[5];

*int* arr[5];

    p = arr;

    ptr = &arr;

*printf*("p = %p, ptr = %p\n", p, ptr);

    p++;

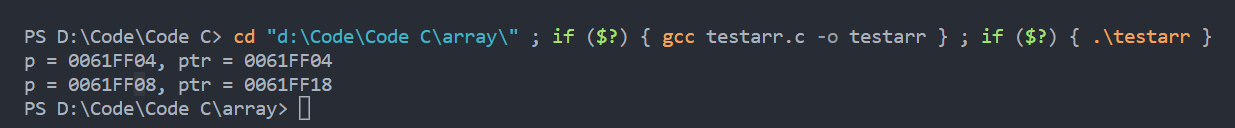
    ptr++;

*printf*("p = %p, ptr = %p\n", p, ptr);

    return 0;

}

OUTPUT:



Other

#include <stdio.h>

*int* *main*()

{

*int* arr[] = {3, 5, 6, 7, 9};

*int* \*p = arr;

*int*(\*ptr)[5] = &arr;

*printf*("p = %p, ptr = %p\n", p, ptr);

*printf*("\*p = %d, \*ptr = %p\n", \*p, \*ptr);

*printf*("sizeof(p) = %lu, sizeof(\*p) = %lu\n",

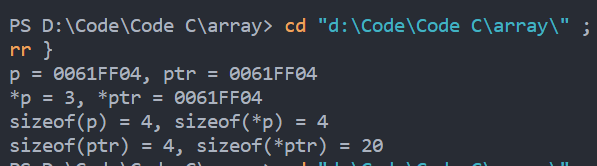
           sizeof(p), sizeof(\*p));

*printf*("sizeof(ptr) = %lu, sizeof(\*ptr) = %lu\n",

           sizeof(ptr), sizeof(\*ptr));

    return 0;

}



#include <stdio.h>

*int* *main*()

{

*int* arr[] = {3, 5, 6, 7, 9};

*int* \*p = arr;

*int*(\*ptr)[5] = &arr;

*printf*("p = %p, ptr = %p\n", p, ptr);

*printf*("\*p = %d, \*ptr = %p\n", \*p, \*(*int* \*)(\*ptr));

*printf*("sizeof(p) = %lu, sizeof(\*p) = %lu\n",

           sizeof(p), sizeof(\*p));

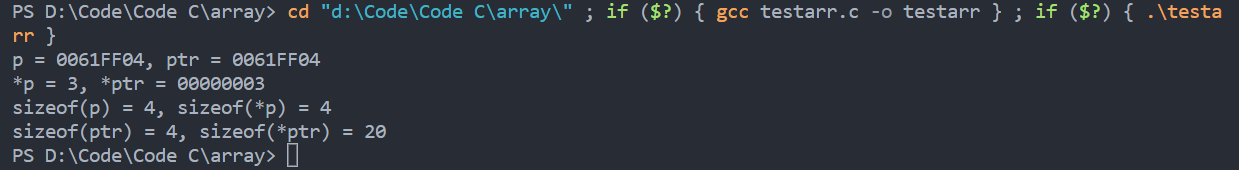
*printf*("sizeof(ptr) = %lu, sizeof(\*ptr) = %lu\n",

           sizeof(ptr), sizeof(\*ptr));

    return 0;

}

OUT:

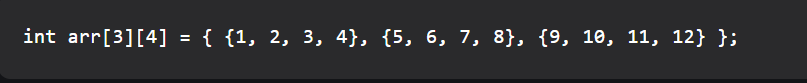


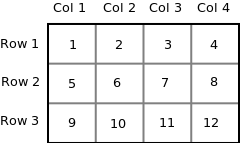
#### Pointer to Multidimensional Arrays:

Con trỏ và Mảng hai chiều: Trong mảng hai chiều, chúng ta có thể truy cập từng phần tử bằng cách sử dụng hai chỉ số dưới, trong đó chỉ số dưới đầu tiên đại diện cho số hàng và chỉ số dưới thứ hai đại diện cho số cột. Các phần tử của mảng 2-D cũng có thể được truy cập với sự trợ giúp của ký hiệu con trỏ. Giả sử arr là một mảng 2-D, chúng ta có thể truy cập bất kỳ phần tử nào arr[i][j] của mảng bằng cách sử dụng biểu thức con trỏ \*(\*(arr + i) + j). Bây giờ chúng ta sẽ xem biểu thức này có thể được bắt nguồn như thế nào.

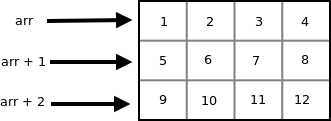
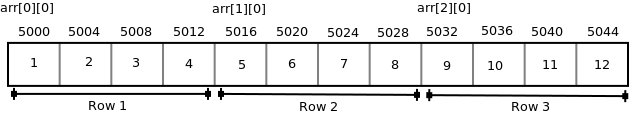
***Ví dụ trang 33***

Chúng ta hãy lấy một mảng hai chiều arr[3][4]:





Vì bộ nhớ trong máy tính được tổ chức tuyến tính, không thể lưu trữ mảng 2-D theo hàng và cột. Khái niệm hàng và cột chỉ là lý thuyết, trên thực tế, một mảng 2-D được lưu trữ theo thứ tự hàng chính, tức là các hàng được đặt cạnh nhau. Hình dưới đây cho thấy mảng 2-D ở trên sẽ được lưu trữ như thế nào trong bộ nhớ.



#include<stdio.h>

*int* *main*()

{

*int* arr[2][3][2] = {

                       {

                         {5, 10},

                         {6, 11},

                         {7, 12},

                       },

                       {

                         {20, 30},

                         {21, 31},

                         {22, 32},

                       }

                     };

*int* i, j, k;

 for (i  = 0 ; i *<* 2;i++){

    for(j = 0 ; j *<*3 ; j ++){

        for(k = 0 ; k *<* 2;k++){

*printf*("%d ",\*(\*(\*(arr + i)+ j)+k));

        }

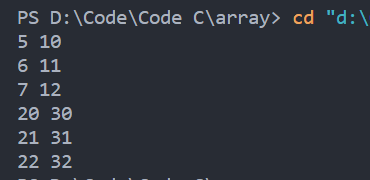
*printf*("\n\t\r");

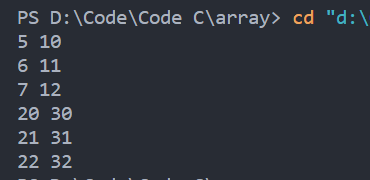
    }

 }

}

====================================





#include <stdio.h>

*int* *main*()

{

*int* arr[3][4] = {

        {10, 11, 12, 13},

        {20, 21, 22, 23},

        {30, 31, 32, 33}};

*int*(\*ptr)[4];

    ptr = arr;

*printf*("%p %p %p\n", ptr, ptr + 1, ptr + 2);

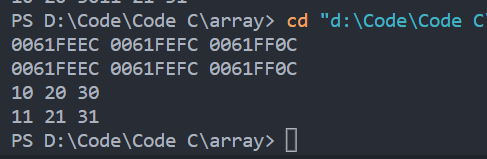
*printf*("%p %p %p\n", \*(ptr), \*(ptr + 1), \*(ptr + 2));

*printf*("%d %d %d\n", \*(\*(ptr)), \*(\*(ptr + 1)), \*(\*(ptr + 2)));

*printf*("%d %d %d\n", \*(\*(ptr) + 1), \*(\*(ptr + 1) + 1), \*(\*(ptr + 2) + 1));

    return 0;

}



### 4.3.2. Function Pointer :

Trong C, giống như các con trỏ dữ liệu thông thường (int \*, char \*, v.v.), chúng ta có thể có con trỏ đến các hàm. Sau đây là một ví dụ đơn giản cho thấy khai báo và cuộc gọi hàm bằng cách sử dụng con trỏ hàm.

#include <stdio.h>

*void* *fun*(*int* *a*)

{

*printf*("Value of a is %d\n", *a*);

}

*int* *main*()

{

*void* (\*fun\_ptr)(*int*) = &*fun*;

    (\*fun\_ptr)(10);

    return 0;

}

Tại sao chúng ta cần thêm một dấu ngoặc xung quanh các con trỏ hàm như fun\_ptr trong ví dụ trên?

Nếu chúng ta loại bỏ dấu ngoặc, thì biểu thức "void (\*fun\_ptr)(int)" trở thành "void \*fun\_ptr(int)" là khai báo của một hàm trả về con trỏ void.

1) Không giống như con trỏ thông thường, con trỏ hàm trỏ đến đầu code của hàm, không phải dữ liệu. Thông thường, một con trỏ hàm lưu trữ phần đầu của mã thực thi.

2) Không giống như con trỏ thông thường, ta không phân bổ bộ nhớ bỏ phân bổ bằng cách sử dụng con trỏ hàm.

3) Tên của một hàm cũng có thể được sử dụng để lấy địa chỉ của hàm. Ví dụ: trong chương trình dưới đây, đã xóa toán tử địa chỉ '&' trong bài tập. Chúng tôi cũng đã thay đổi lệnh gọi hàm bằng cách xóa \*, chương trình vẫn hoạt động.

#include <stdio.h>

*void* *fun*(*int* *a*)

{

*printf*("Value of a is %d\n", *a*);

}

*int* *main*()

{

*void* (\*fun\_ptr)(*int*) = *fun*; *// & removed*

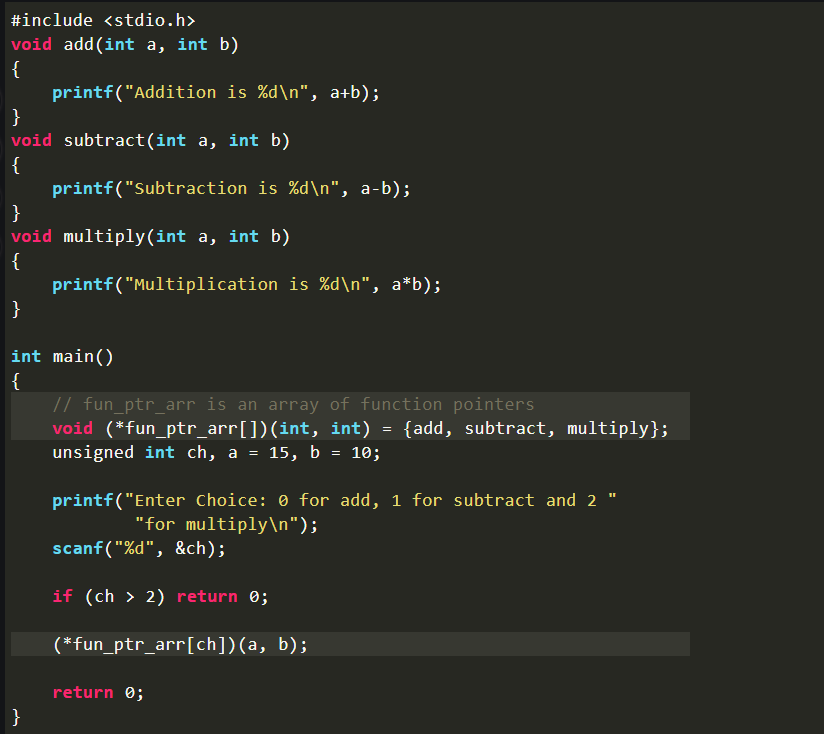
*fun\_ptr*(10); *// \* removed*

    return 0;

}

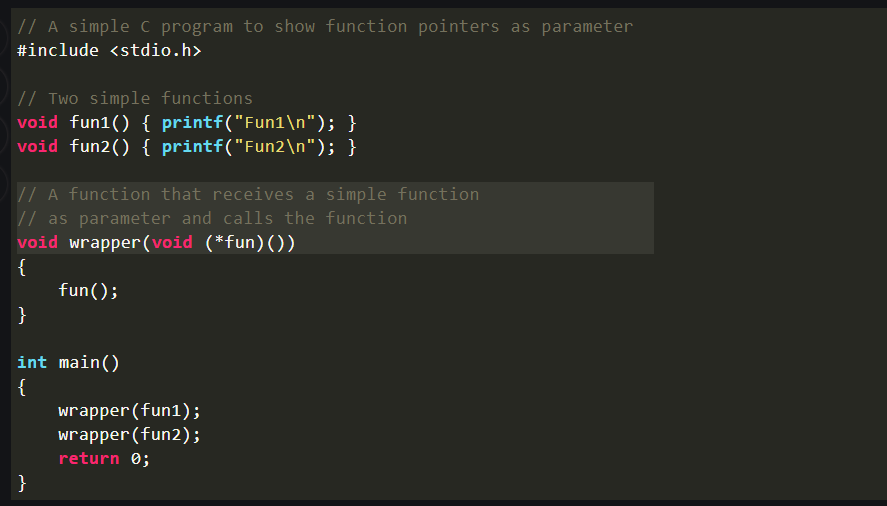
4) Giống như con trỏ bình thường, ta có thể có một mảng các con trỏ hàm. Ví dụ dưới đây ở điểm 5 cho thấy cú pháp cho mảng con trỏ.

5) Con trỏ chức năng có thể được sử dụng thay cho trường hợp chuyển đổi. Ví dụ: trong chương trình dưới đây, người dùng được yêu cầu lựa chọn giữa 0 và 2 để thực hiện các tác vụ khác nhau.

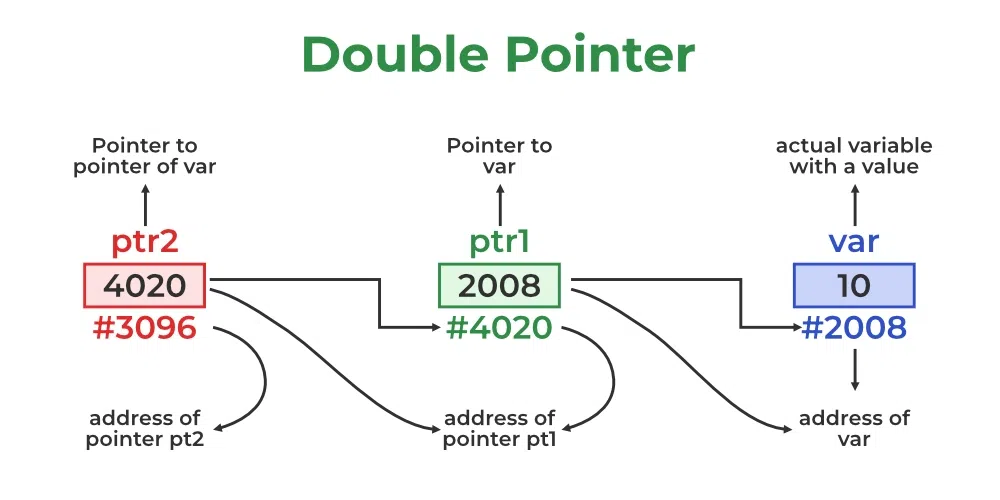


6) Giống như con trỏ dữ liệu thông thường, một con trỏ hàm có thể được truyền dưới dạng đối số và cũng có thể được trả về từ một hàm.

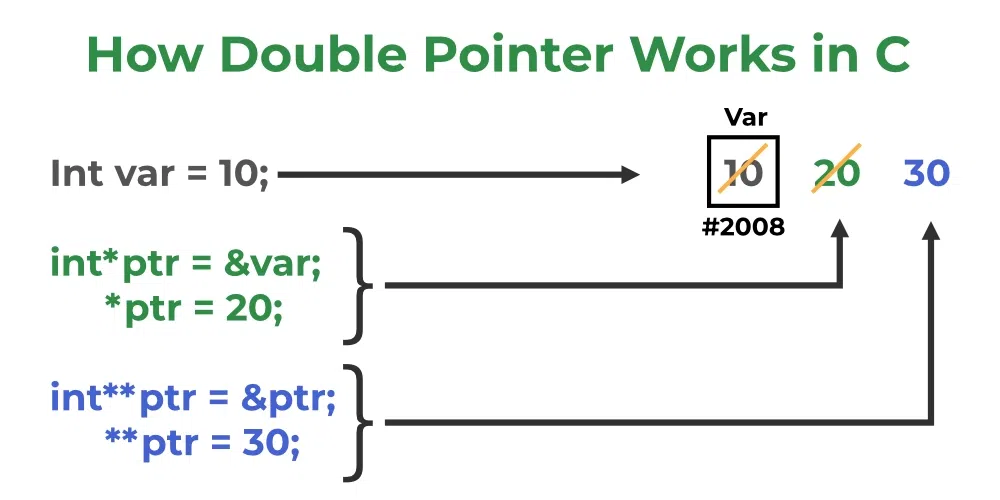
Ví dụ, hãy xem xét chương trình C sau đây trong đó wrapper() nhận được một void fun() làm tham số và gọi hàm được truyền.



### 4.3.3. Double pointer:



#### How Double Pointer Works?



### 4.3.4. NULL Pointer

NULL Pointer là những con trỏ không trỏ đến bất kỳ vị trí bộ nhớ nào. Chúng có thể được tạo bằng cách gán giá trị NULL cho con trỏ. Một con trỏ thuộc bất kỳ loại nào cũng có thể được gán giá trị NULL.

* data\_type \*pointer\_name = NULL;

or

* pointer\_name = NULL

### 4.3.5. Void Pointer

Các Void Pointer trong C là các con trỏ của loại void. Điều đó có nghĩa là chúng không có bất kỳ kiểu dữ liệu liên quan nào. Chúng còn được gọi là con trỏ chung vì chúng có thể trỏ đến bất kỳ loại nào và có thể được đánh máy cho bất kỳ loại nào.

* void \* pointer\_name;

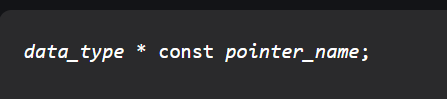
### 4.3.6. Wild Pointers

Wild Pointers là những con trỏ chưa được khởi tạo với một cái gì đó. Những loại con trỏ C này có thể gây ra sự cố trong các chương trình và cuối cùng có thể khiến chúng gặp sự cố.

* int \*ptr;
* char \*str;

### 4.3.7. Constant Pointers (Hằng con trỏ)

Đặc điểm của con trỏ này là nó chỉ có thể trỏ đến 1 địa chỉ duy nhất và sau đó không thể thay đổi địa chỉ trỏ được nữa. Khác với con trỏ hằng thì hằng con trỏ có thể đọc ghi giá trị vùng nhớ thông qua chính bản thân con trỏ đó.



#include <stdio.h>

*void*  *main*()

{

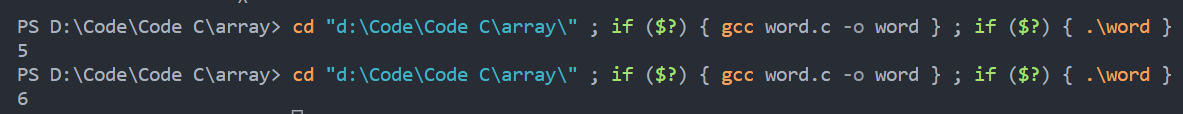
*int* b = 5;

*int* \*const a = &b;

   \*a = 6;

*printf*("%d", \*a);

}



#include <stdio.h>

*void*  *main*()

{

*int* b = 5;

*int* c = 5;

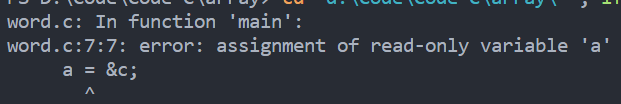
*int* \*const a = &b;

    a = &c;

   \*a = 6;

*printf*("%d", \*a);

}



### 4.3.8. Pointer to Constant (Con trỏ hằng)

Con trỏ hằng là con trỏ có thể trỏ đến 1 vùng nhớ hằng. Đặc điểm của con trỏ này là nó là con trỏ chỉ đọc (read-only), người dùng có thể thông qua nó đọc giá trị vùng nhớ mà nó trỏ đến nhưng không thể thông qua nó ghi lại giá trị vào vùng nhớ đó.



#include <stdio.h>

*void*  *main*()

{

*int* b = 5;

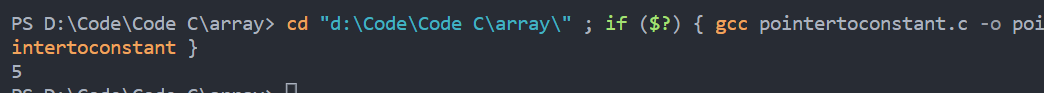
*int* c = 5;

    const *int* \* a = &b;

    a = &c;

*printf*("%d", \*a);

}



=||==||==||

#include <stdio.h>

*void*  *main*()

{

*int* b = 5;

*int* c = 5;

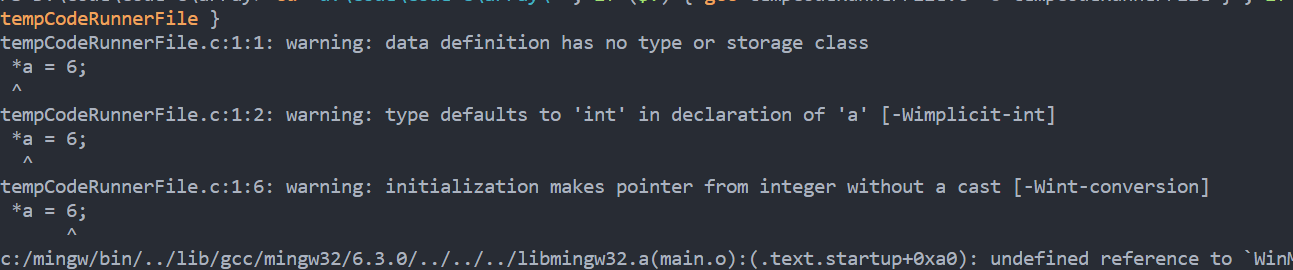
    const *int* \* a = &b;

    a = &c;

   \*a = 6;

*printf*("%d", \*a);

}



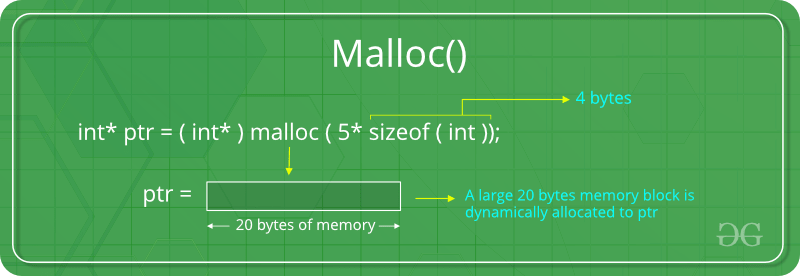
## 4.4. Dynamic Memory Allocation

Ở đây mình đã khai báo 1 vùng nhớ gồm 10 phần tử int (mỗi phần tử 4 byte). Vùng nhớ này được cấp phát tĩnh ngay khi chương trình bắt đầu chạy và tồn tại tới khi chương trình kết thúc. Như vậy nếu như trong quá trình chạy chúng ta cần sử dụng 1 vùng nhớ với mục đích riêng, lúc này chúng ta phải sử dụng kỹ thuật cấp phát động để có thể được cấp phát cho 1 vùng nhớ trên RAM để sử dụng. Vùng nhớ này cũng sẽ không tự mất đi mà chỉ khi chúng ta tự giải phóng nó mới mất.

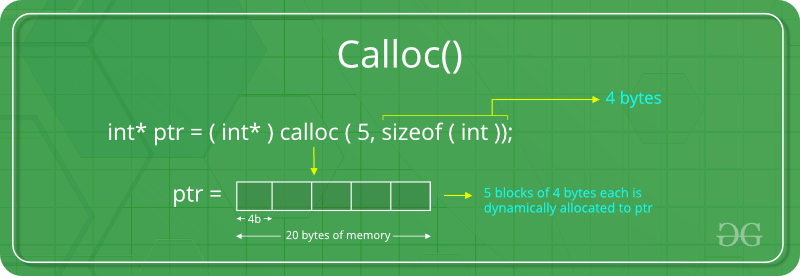
### 4.4.1. Syntax

1. malloc()
2. calloc()
3. free()
4. realloc()

#### Malloc()

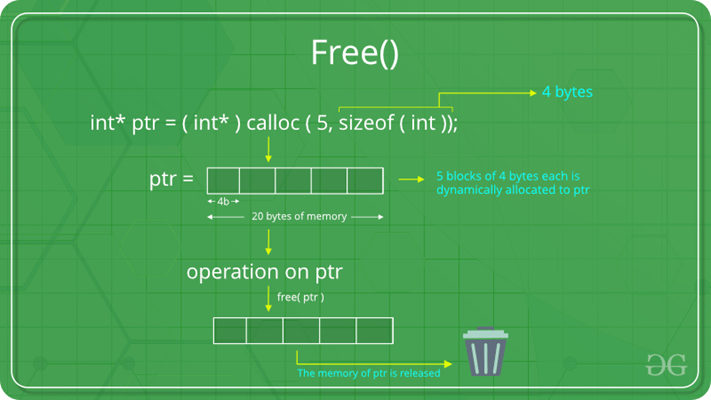


#### Calloc()



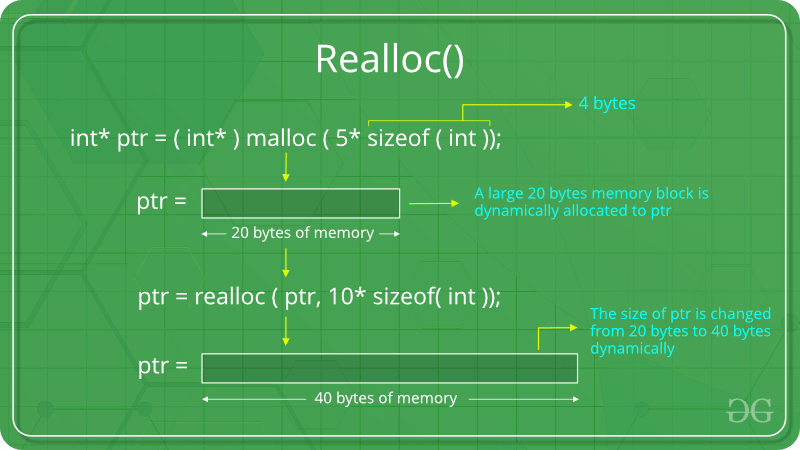
#### free():

Note: Luôn luôn đưa con trỏ về vị trí ban đầu cấp phát nếu dùng các toán tử ++,--



#### realloc()

Phương pháp "realloc" hoặc " reallocation" trong C được sử dụng để tự động thay đổi phân bổ bộ nhớ của bộ nhớ được cấp phát trước đó. Nói cách khác, nếu bộ nhớ được cấp phát trước đó với sự trợ giúp của malloc hoặc calloc là không đủ, realloc có thể được sử dụng để tự động phân bổ lại bộ nhớ. Phân bổ lại bộ nhớ duy trì giá trị đã có và các khối mới sẽ được khởi tạo với giá trị rác mặc định.



|  |  |  |
| --- | --- | --- |
| S.No. | Malloc() | calloc() |
| 1. | malloc() là một hàm tạo ra một khối bộ nhớ có kích thước cố định. | calloc() là một hàm gán một số khối bộ nhớ cụ thể cho một biến duy nhất. |
| 2. | malloc() chỉ lấy một đối số | calloc() có hai đối số. |
| 3. | malloc() nhanh hơn calloc. | calloc() chậm hơn malloc() |
| 4 | malloc() có hiệu quả thời gian cao | calloc() có hiệu quả thời gian thấp |
| 5 | malloc() được sử dụng để chỉ ra sự phân bổ bộ nhớ | calloc() được sử dụng để chỉ ra phân bổ bộ nhớ liền kề |
| 6 | Cú pháp : void\* malloc(size\_t size); | Cú pháp : void\* calloc(size\_t num, size\_t size); |
| 7 | malloc() không khởi tạo bộ nhớ về 0 | calloc() khởi tạo bộ nhớ về 0 |
| 8 | malloc() không thêm bất kỳ chi phí bộ nhớ bổ sung nào | calloc() thêm một số chi phí bộ nhớ bổ sung |

## 4.5. Ưu điểm của con trỏ

Sau đây là những ưu điểm chính của con trỏ trong C:

1. Con trỏ được sử dụng để phân bổ bộ nhớ động và phân bổ bộ nhớ.
2. Một mảng hoặc một cấu trúc có thể được truy cập hiệu quả với các con trỏ
3. Con trỏ rất hữu ích để truy cập các vị trí bộ nhớ.
4. Con trỏ được sử dụng để tạo thành các cấu trúc dữ liệu phức tạp như danh sách được liên kết, đồ thị, cây, v.v.
5. Con trỏ cũng làm giảm độ dài của chương trình và thời gian thực hiện của nó.

## 4.6. Nhược điểm của con trỏ

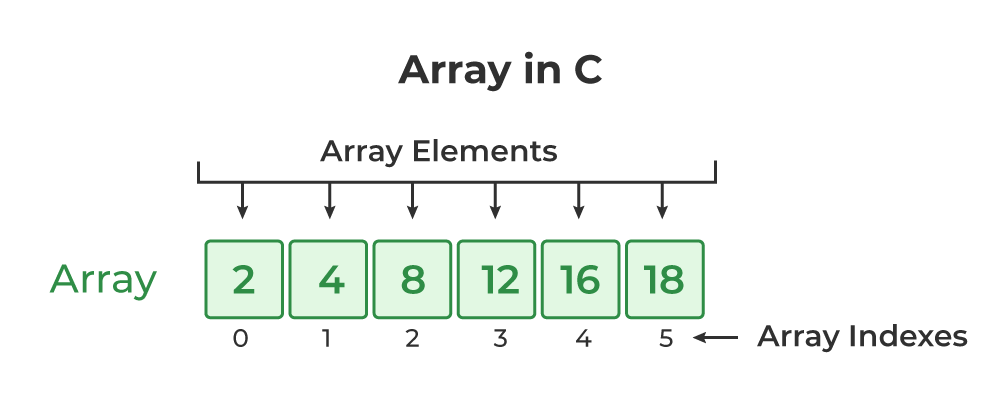
Con trỏ dễ bị lỗi và có những nhược điểm sau:

1. Hỏng bộ nhớ có thể xảy ra nếu một giá trị không chính xác được cung cấp cho con trỏ.
2. Con trỏ hơi phức tạp để hiểu.
3. Con trỏ chịu trách nhiệm chính cho memory leaks in C.
4. Con trỏ tương đối chậm hơn các biến trong C.
5. Con trỏ chưa khởi tạo có thể gây ra lỗi phân đoạn.

# Array

## Define ( Mảng là gì ?)

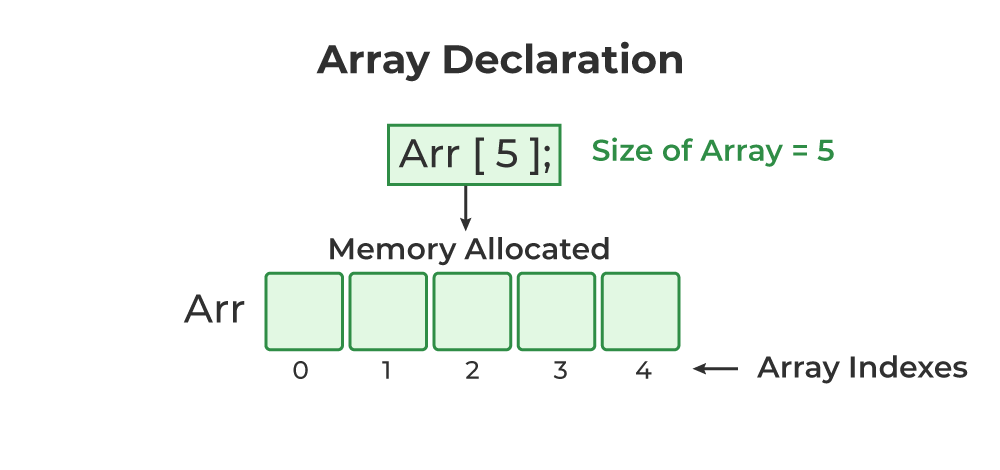
Một mảng trong C là một tập hợp kích thước cố định của các mục dữ liệu được lưu trữ ở các vị trí bộ nhớ liền kề. Nó có thể được sử dụng để lưu trữ tập hợp các kiểu dữ liệu nguyên thủy như int, char, float, v.v., cũng như các kiểu dữ liệu có nguồn gốc và do người dùng xác định như con trỏ, structure, v.v

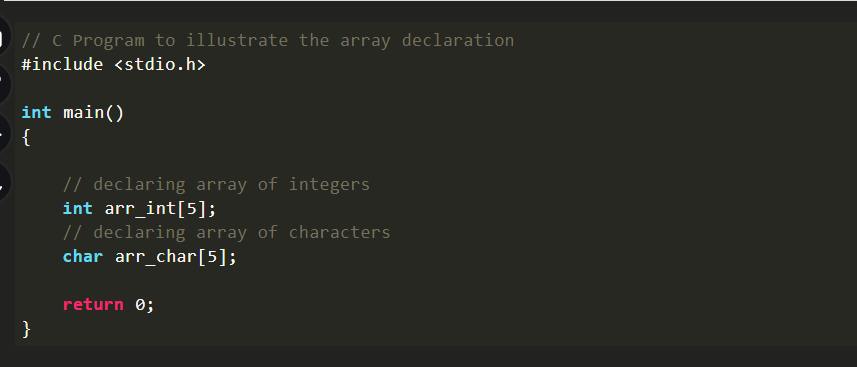


## C Array Declaration (Khai báo)

### Syntax of Array Declaration

* + - * data\_type array\_name [size];
        + Ex: int Arr[50];
      * or
      * data\_type array\_name [size1] [size2]...[sizeN];
        + int Arr[5][5];

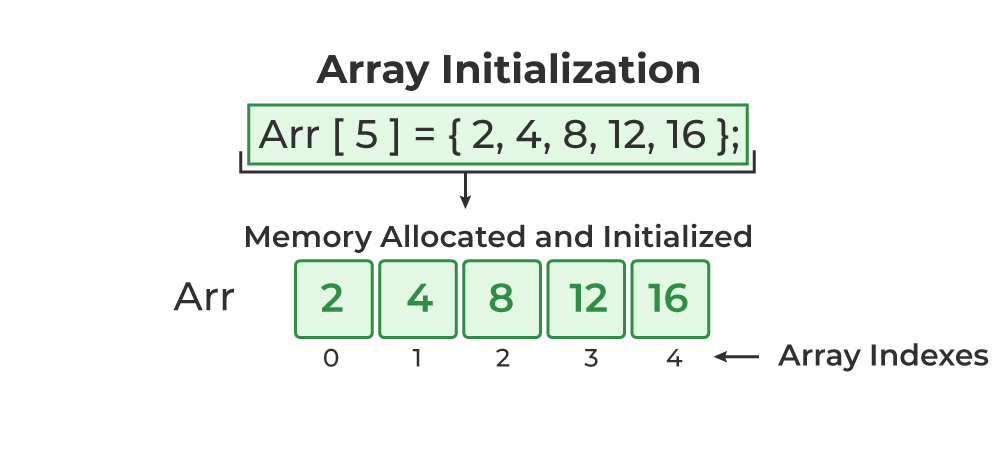




## C Array Initialization (Khởi tạo giá trị )

### Array Initialization with Declaration ( Khai báo và khởi tạo)

* data\_type array\_name [size] = {value1, value2, ... valueN};



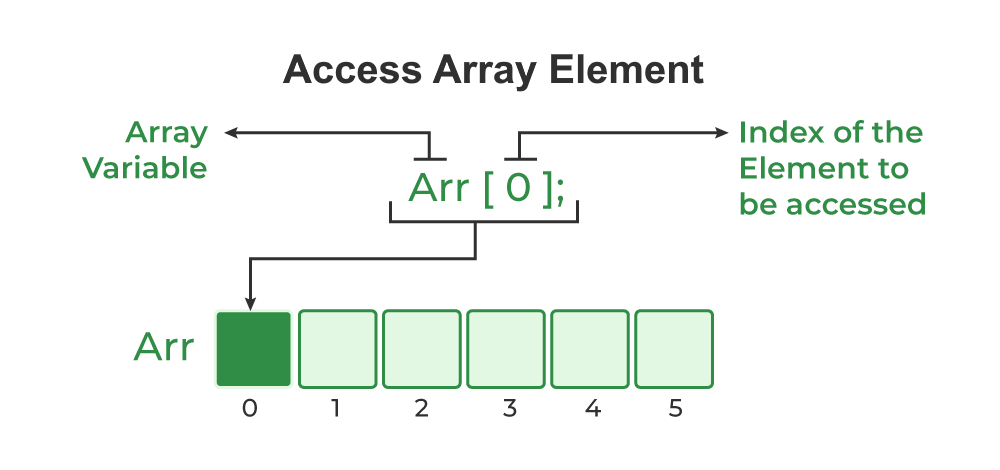
### Array Initialization with Declaration without Size ( Khởi tạo và khai báo không cần kích thước )

* data\_type array\_name[] = {1,2,3,4,5};

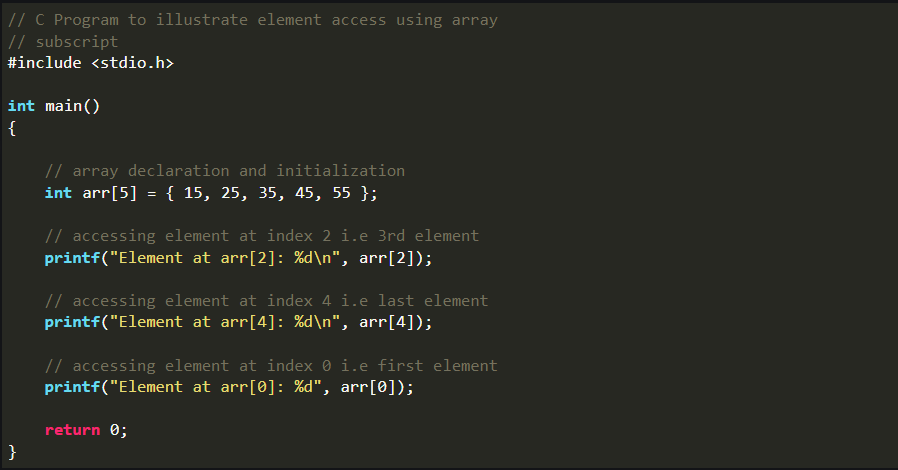
## Access Array Elements ( Truy cập )

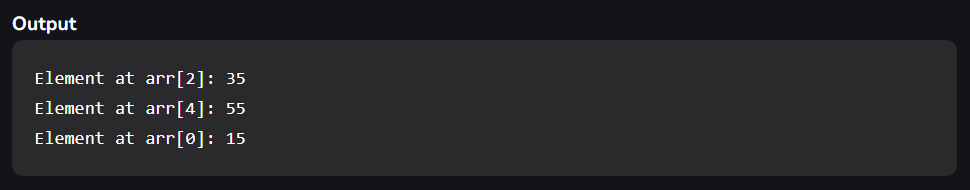
Chúng ta có thể truy cập bất kỳ phần tử nào của một mảng trong C bằng cách sử dụng toán tử chỉ số dưới mảng [ ] và giá trị chỉ mục i của phần tử.





Một điều cần lưu ý là việc lập chỉ mục trong mảng luôn bắt đầu bằng 0, tức là phần tử đầu tiên nằm ở chỉ số 0 và phần tử cuối cùng ở N – 1 trong đó N là số phần tử trong mảng.





## Update Array Element ( Thay đổi giá trị)

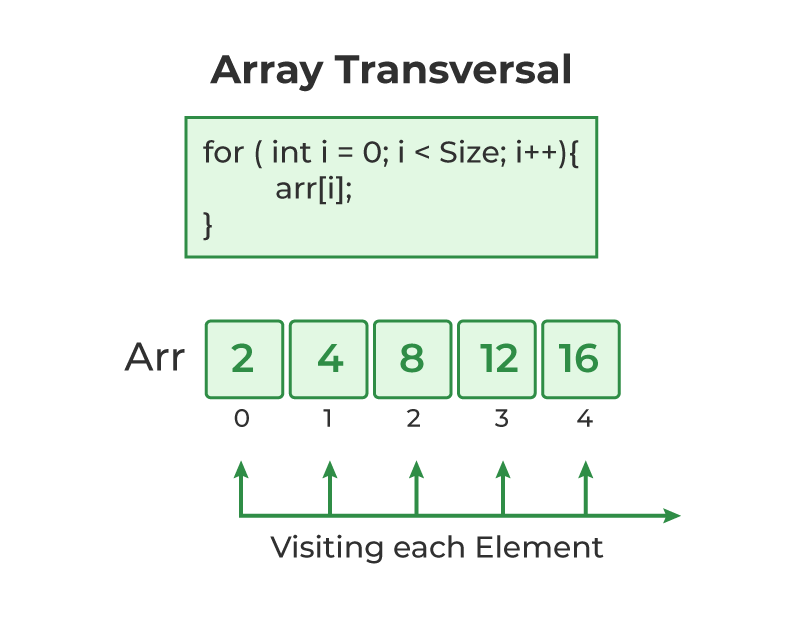
Chúng ta có thể cập nhật giá trị của một phần tử tại chỉ mục i đã cho theo cách tương tự như truy cập một phần tử bằng cách sử dụng toán tử chỉ số dưới mảng [ ] và toán tử gán =.



## C Array Traversal (Duyệt mảng )

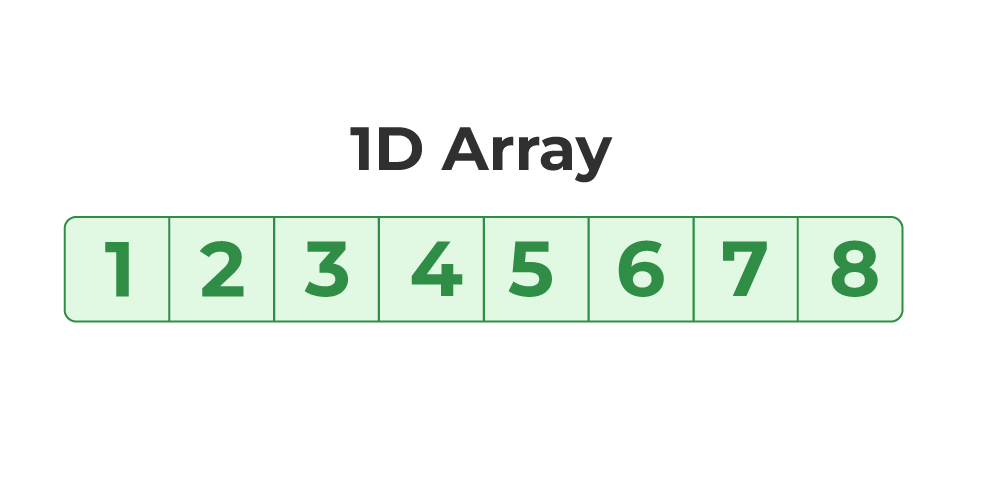
Traversal là quá trình mà chúng ta truy cập dữ liệu của cấu trúc dữ liệu. Đối với truyền qua mảng C, chúng ta sử dụng các vòng lặp để lặp qua từng phần tử của mảng.

Array Traversal sử dụng cho Loop



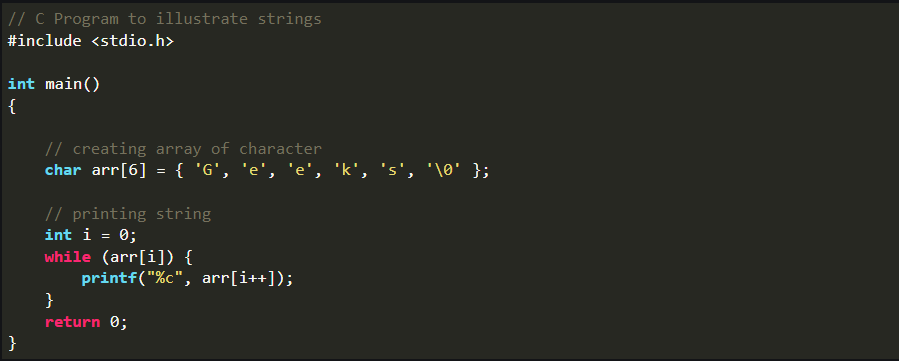
## Types of Array in C ( Các loại mảng trong C)

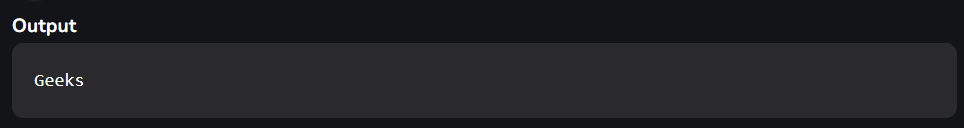
### One Dimensional Arrays (1D Array)



#### Array of Characters (Strings)

Trong C, chúng ta lưu trữ các từ, tức là một chuỗi các ký tự dưới dạng một mảng các ký tự được chấm dứt bởi một ký tự “\0”. Chúng được gọi là chuỗi trong ngôn ngữ C.





To know more about strings, refer to this article– [Strings in C](https://www.geeksforgeeks.org/strings-in-c/)

### Multidimensional Arrays (Mảng nhiều chiều )

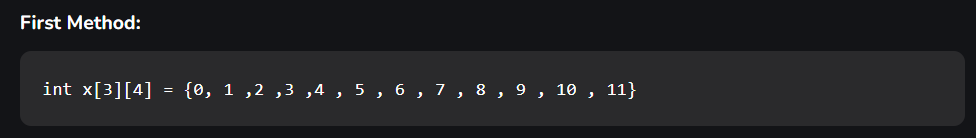
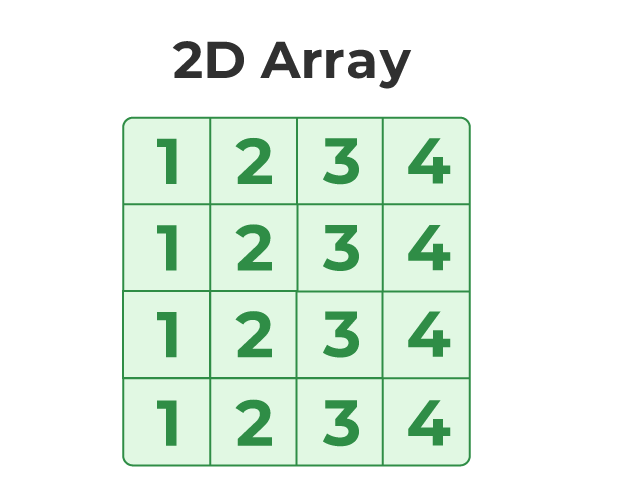
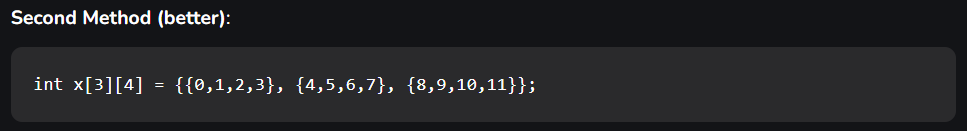
Mảng đa chiều trong C là những mảng có nhiều hơn một chiều. Một số mảng đa chiều phổ biến là mảng 2D và mảng 3D. Chúng ta có thể khai báo các mảng có nhiều kích thước hơn mảng 3d nhưng chúng được tránh vì chúng rất phức tạp và chiếm một lượng lớn không gian

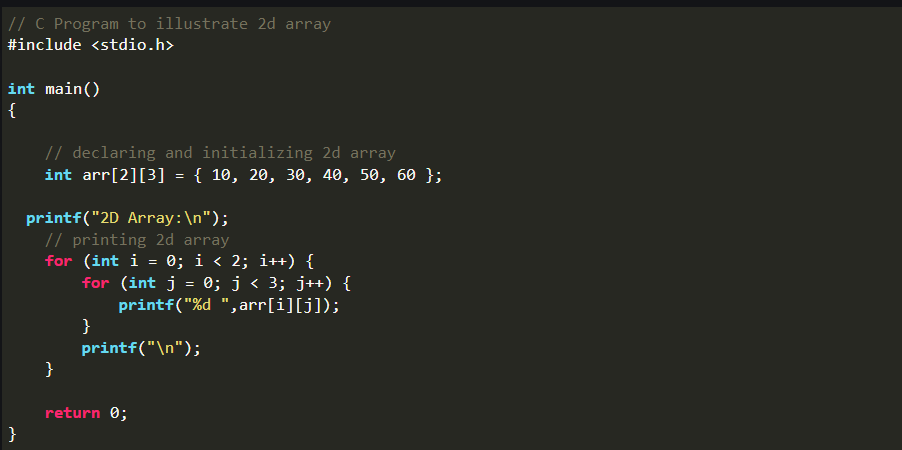
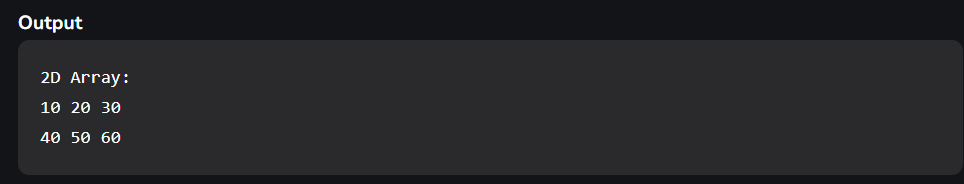
#### Two-Dimensional Array in C( mảng 2 chiều)

Mảng hai chiều hoặc mảng 2D trong C là một mảng có chính xác hai chiều. Chúng có thể được hình dung dưới dạng các hàng và cột được tổ chức trong một mặt phẳng hai chiều.

##### Syntax of 2D Array in C (Cú pháp)



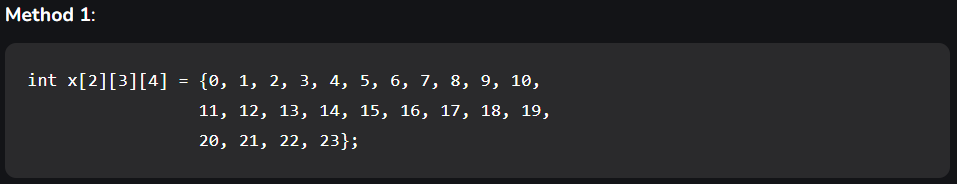
 

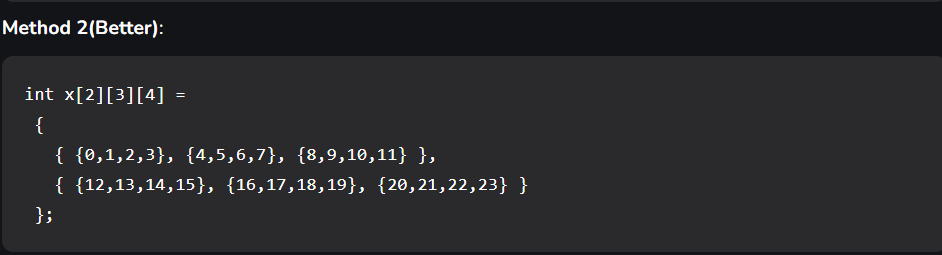
#### Three-Dimensional Array in C ( mảng 3 chiều )

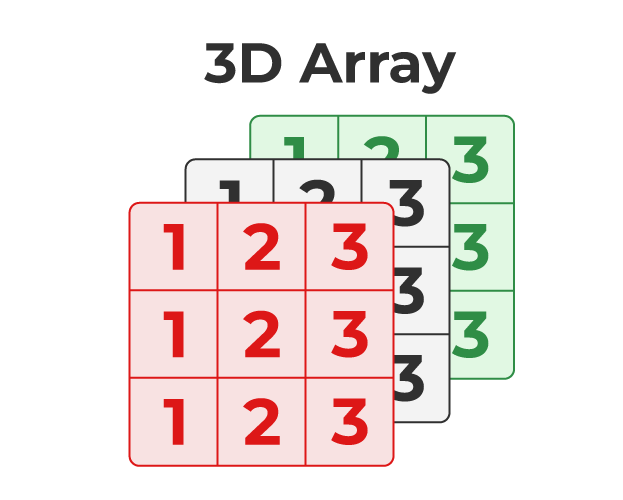
Một dạng phổ biến khác của mảng đa chiều là Mảng ba chiều hoặc Mảng 3D. Một mảng 3D có chính xác ba chiều. Nó có thể được hình dung như một tập hợp các mảng 2D xếp chồng lên nhau để tạo ra chiều thứ ba.

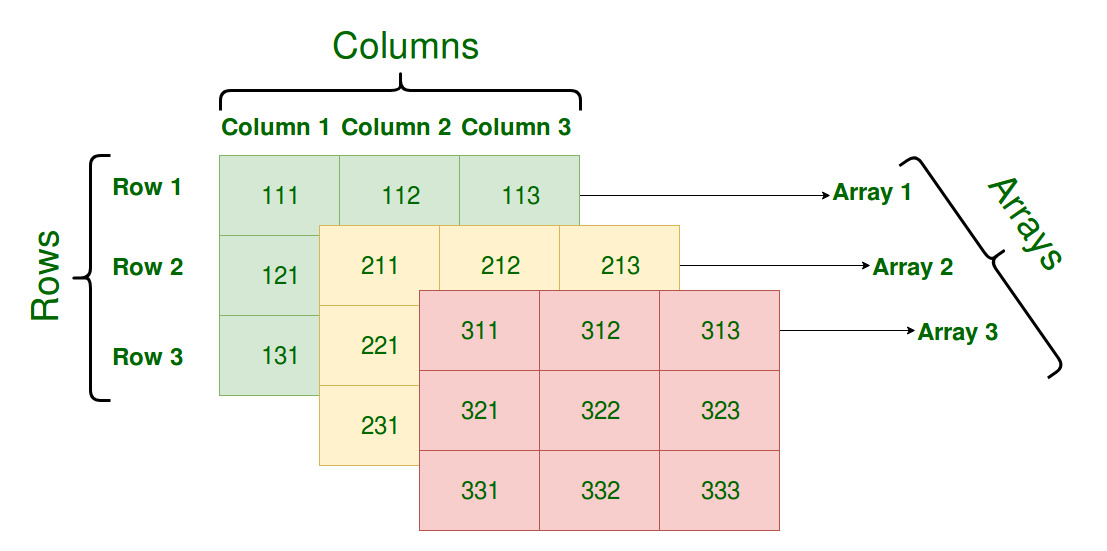
##### Syntax of 3D Array in C







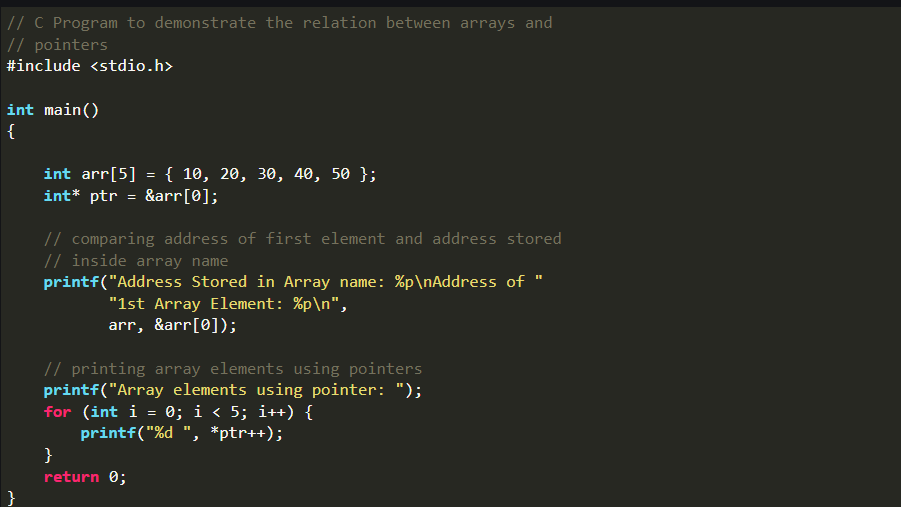


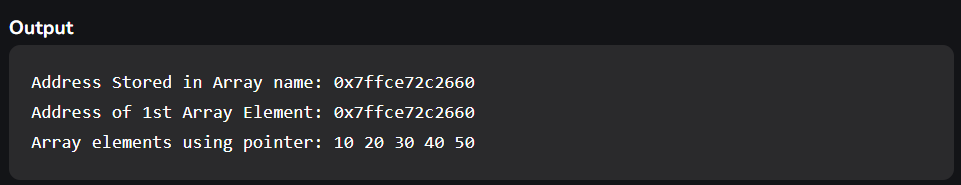


## Relationship between Arrays and Pointers ( Con trỏ và mảng )

*Xem lại Array Pointer*

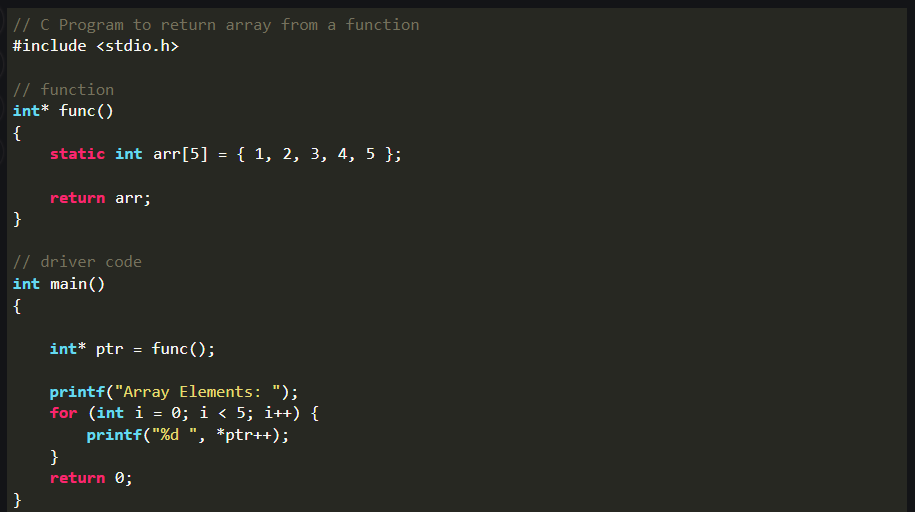
Mảng và Con trỏ có liên quan chặt chẽ với nhau để chúng ta có thể sử dụng con trỏ để thực hiện tất cả các hoạt động có thể có của mảng. Tên mảng là một con trỏ hằng số đến phần tử đầu tiên của mảng và mảng phân rã thành các con trỏ khi được truyền đến hàm.

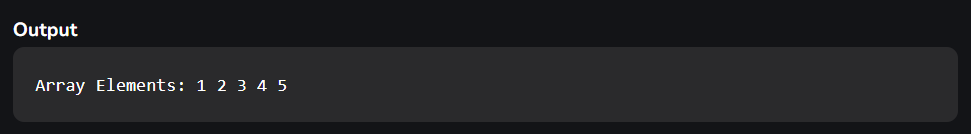




## Return an Array from a Function in C ( Hàm trả về mảng )

Trong C, chúng ta chỉ có thể trả về một giá trị duy nhất từ một hàm. Để trả về nhiều giá trị hoặc phần tử, chúng ta phải sử dụng con trỏ. Chúng ta có thể trả về một mảng từ một hàm bằng cách sử dụng con trỏ đến phần tử đầu tiên của mảng đó.





## Properties of Arrays in C (Thuộc tính của Mảng)

1. Kích thước cố định
2. Dữ liệu trong mảng cùng định dạng ( cùng kiểu dữ liệu )
3. Thứ tự của mảng ( Index)
4. Địa chỉ các phần tử trong mảng là liên tiếp nhau .
5. Truy cập ngẫu nhiên

## Advantages of Array in C (Ưu điểm)

1. Truy cập ngẫu nhiên và nhanh chóng các phần tử bằng cách sử dụng chỉ mục mảng (index).
2. Việc đi qua mảng trở nên dễ dàng bằng cách sử dụng một vòng lặp duy nhất.

## Disadvantages of Array in C (Nhược điểm)

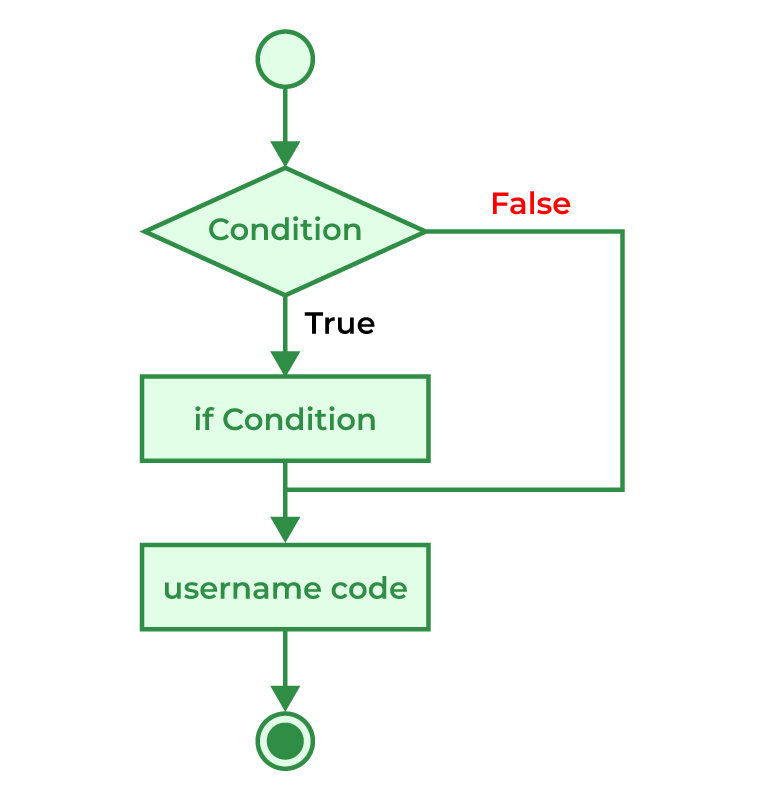
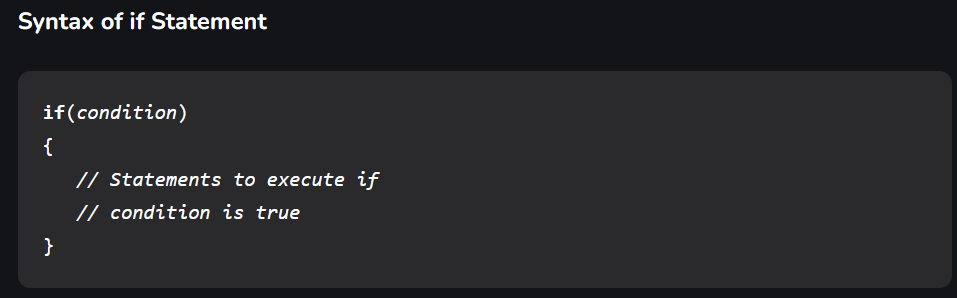
1. Cho phép nhập một số lượng cố định các yếu tố được quyết định tại thời điểm khai báo. Không giống như một linked list , một mảng trong C không động.
2. Việc chèn và xóa các phần tử có thể tốn kém vì các phần tử cần được sắp xếp lại sau khi chèn và xóa.

|  |  |
| --- | --- |
| Pointer | Array |
| Con trỏ là kiểu dữ liệu dẫn xuất có thể lữu trữ địa chỉ của biến khác | Mảng là tập hợp những dữ liệu có cùng kiểu dữ liệu |
| Con trỏ có thể cấp phát trong lúc chạy | Khai báo trước giá trị |
| Biến Dynamic | Biến tĩnh |

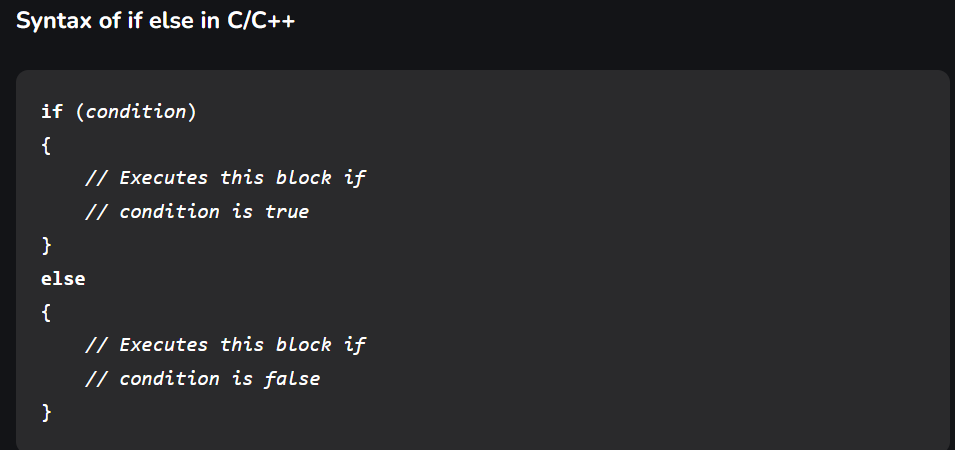
# Decision Making in C (Điều kiện)

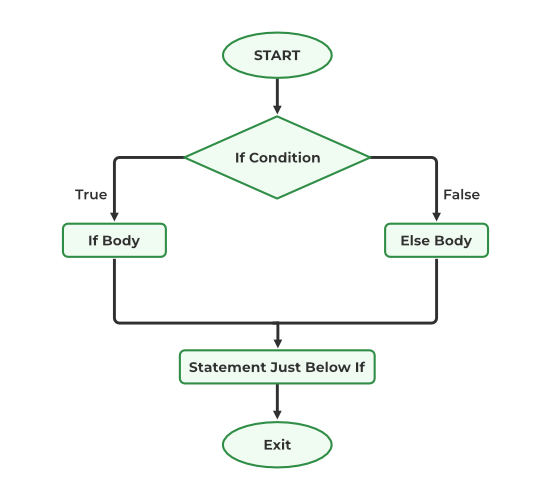
## Need of Conditional Statements

### if in C/C++

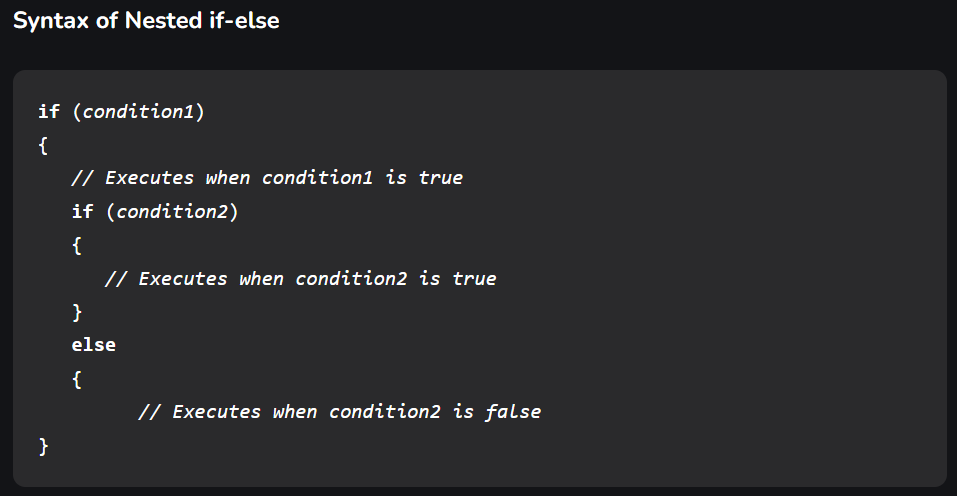


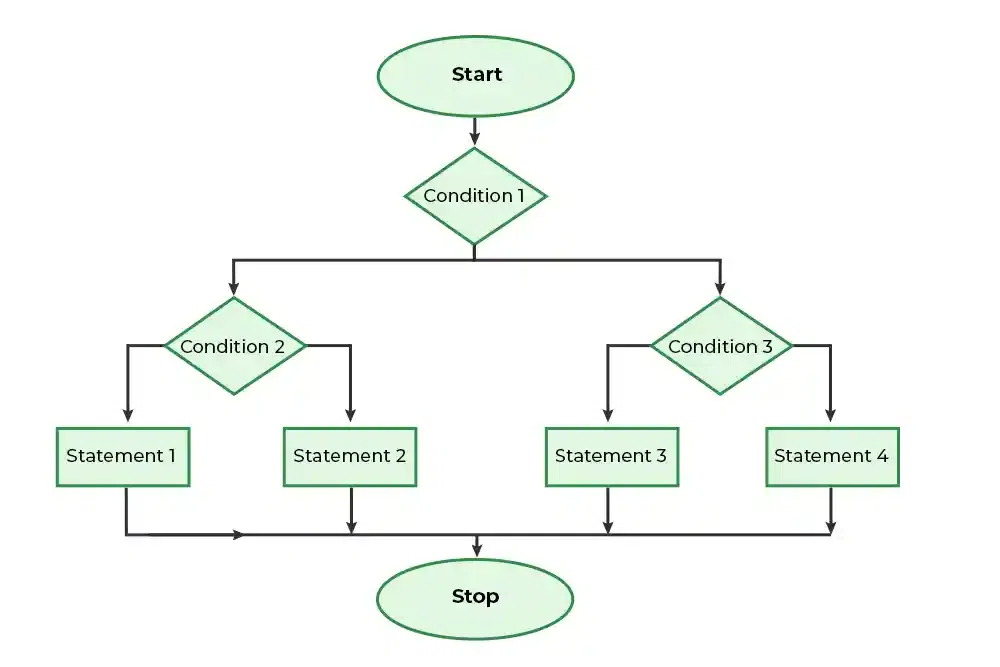
### if-else in C/C++





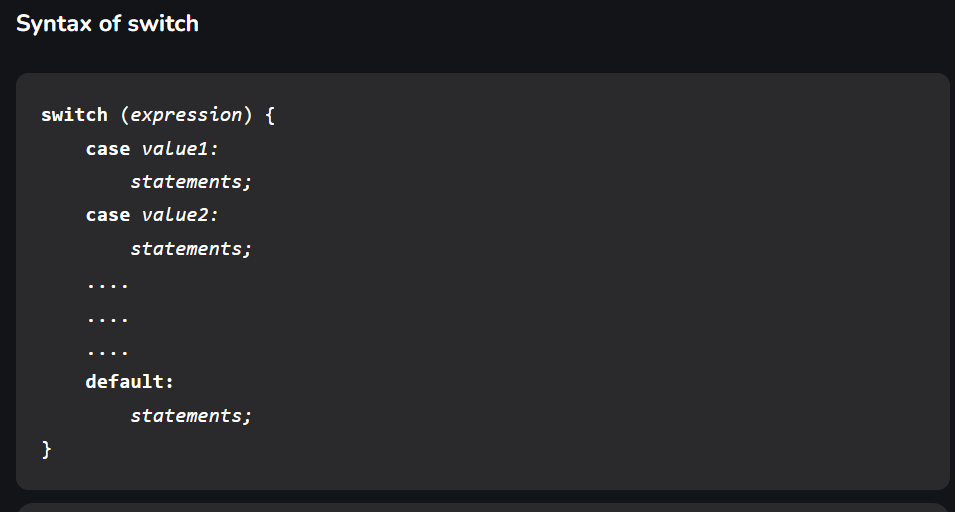
### Nested if-else in C/C++ (If else lồng nhau)

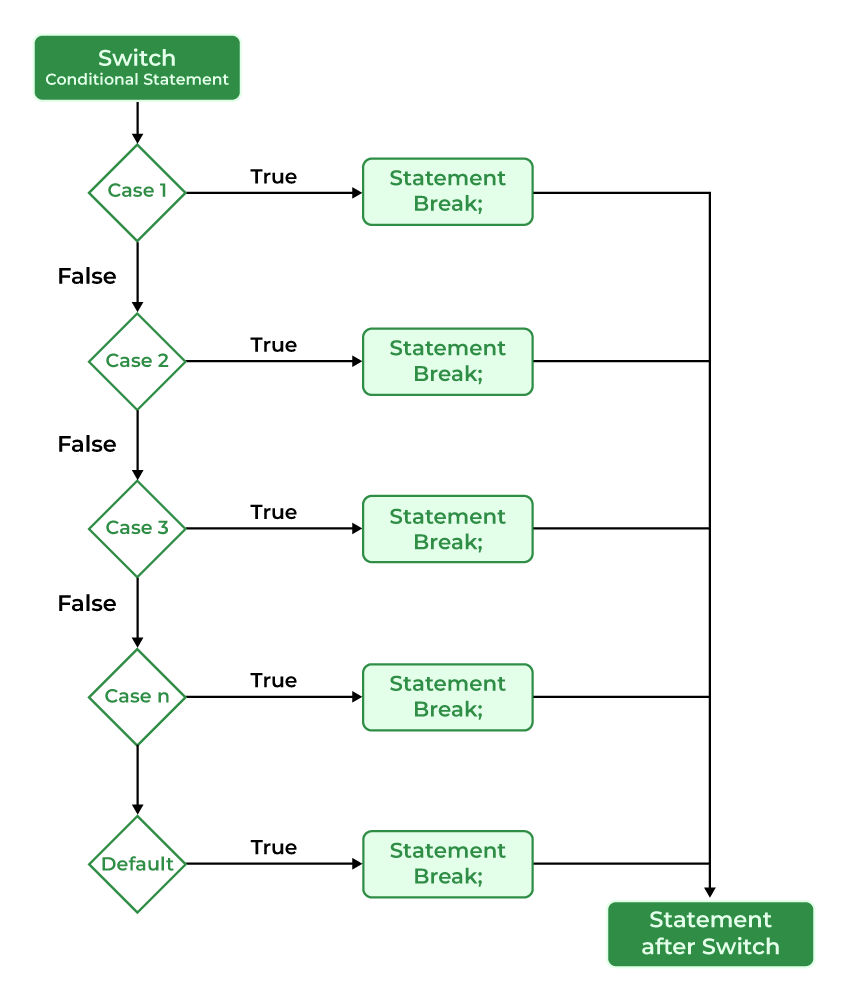




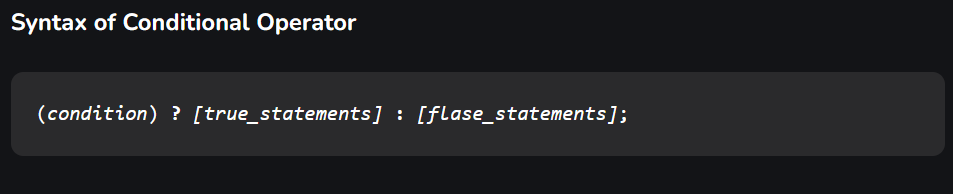
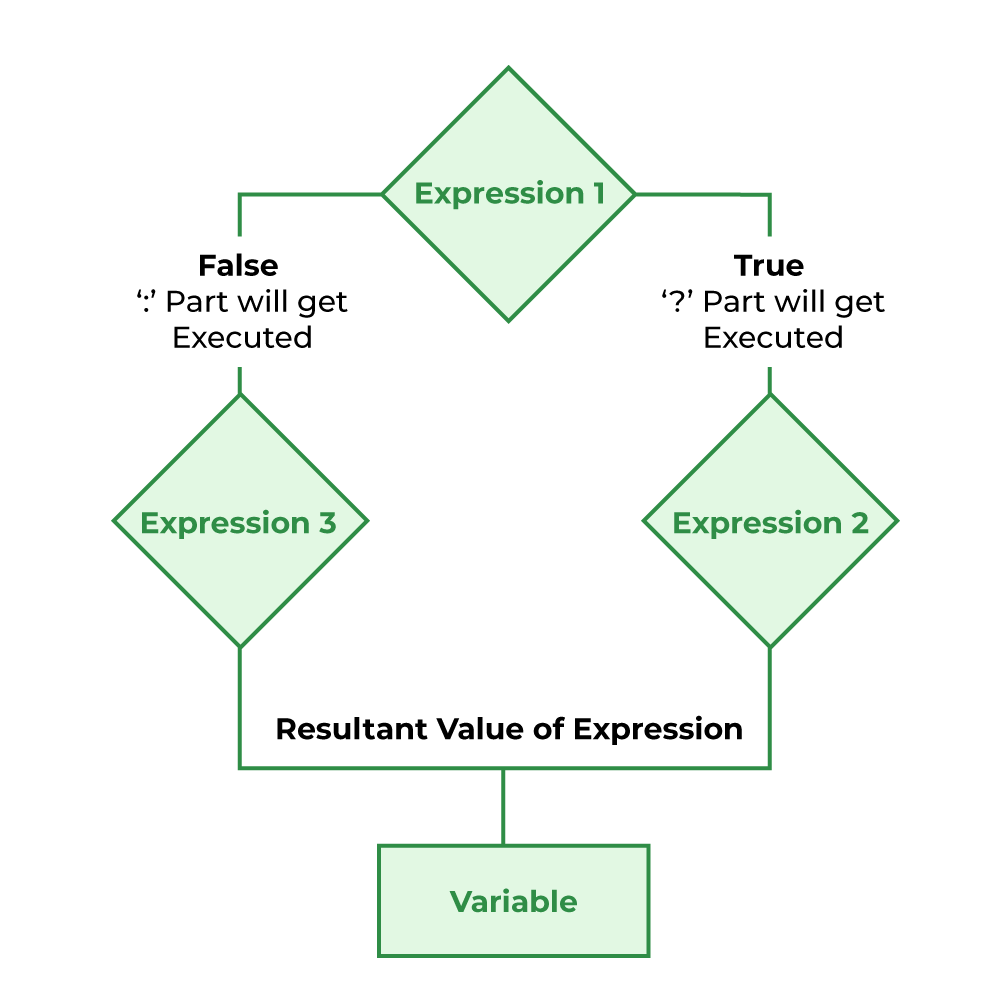
### Switch Statement in C

* expression phải bắt buộc là giá trị hằng, có thể là biểu thức nhưng kết quả cần là hằng số. Trong đó, expression sẽ được so sánh với các giá trị của các case.
* Khả năng xử lí của switch có thể nhanh hơn if else vì khi gặp expression thì nó sẽ nhảy trực tiếp vào điều kiện hàm đó ( có thể coi các value1,2,3 là các index như mảng ).
* Nếu có 1 case nào đó khớp giá trị, các khối lệnh tương ứngsau case đó sẽ được thực hiện cho tới khi gặp lệnh break. Do đó, nếu chúng ta không sử dụng break thì tất cả các case kể từ case khớp giá trị đều được thực hiện.
* Case default sẽ được thực hiện nếu không có case nào khớp giá trị với expression





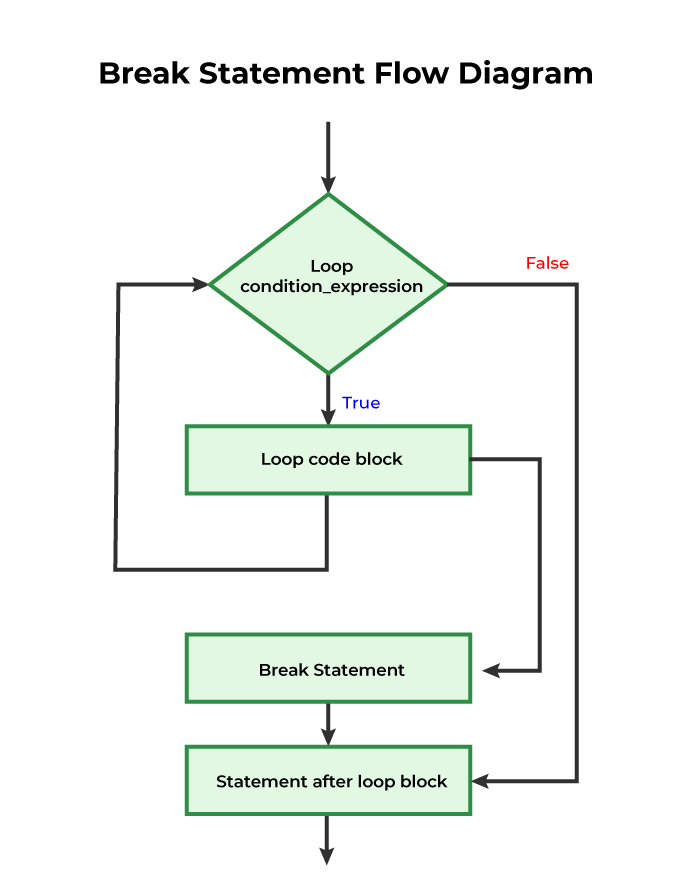
### Conditional Operator in C/C++ (Toán tử 3 ngôi)

### Jump Statements in C/C++

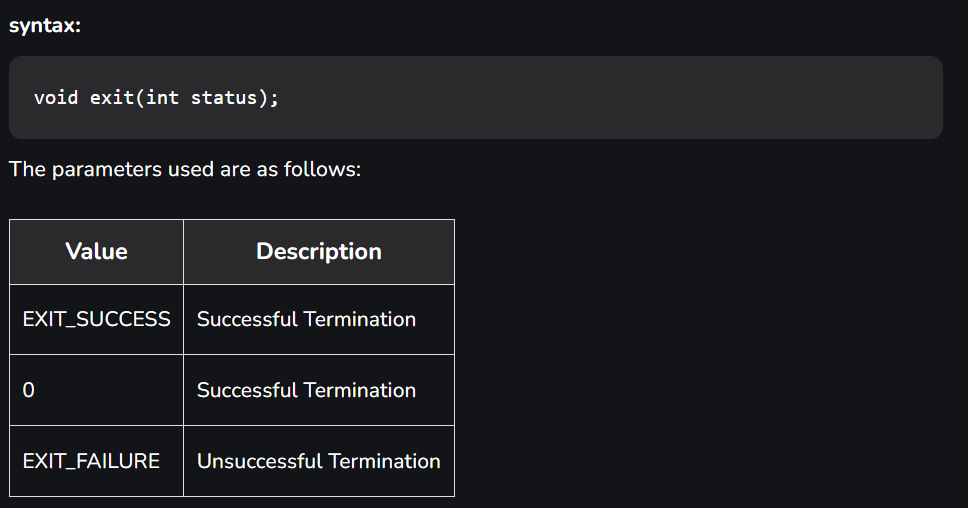
#### Break ( Thoát vòng lặp )

Một vòng lặp nếu đang thực hiện và gặp lệnh break sẽ ngay lập tức thoát ra khỏi vòng lặp. Vòng lặp ở đây có thể là vòng lặp for, vòng lặp while…



#### Exit()

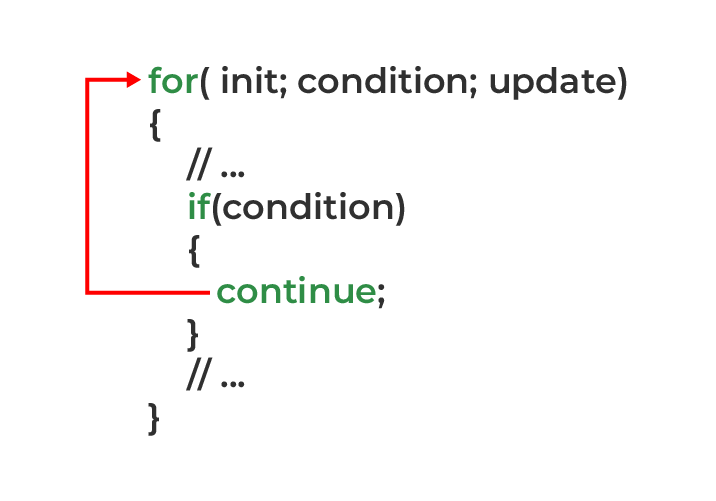
* Khi người dùng muốn thoát khỏi một chương trình từ chức năng này được sử dụng.
* Nó là một hàm loại trả về vô hiệu gọi tất cả các hàm được đăng ký tại lối ra và chấm dứt chương trình.
* Bộ đệm tệp được xóa, luồng bị đóng và các tệp tạm thời bị xóa và do đó bộ nhớ được giải phóng.



|  |  |
| --- | --- |
| break() | exit() |
| Đó là một từ khóa | Nó là một hàm được xác định trước. |
| Nó không yêu cầu bất kỳ tệp tiêu đề nào vì nó được xác định trước trong tệp tiêu đề stdio.h trong C. | Nó yêu cầu stdlib.h chỉ cho C, |
| Nó chấm dứt vòng lặp. | Nó chấm dứt chương trình. |
| Nó thường chỉ được sử dụng trong câu lệnh trường hợp vòng lặp và chuyển đổi. | Nó thường được sử dụng bất cứ nơi nào trong chương trình. |
| Nó không thể được sử dụng như một tên biến vì nó là một từ dành riêng trong ngôn ngữ C. | Nó không phải là một từ dành riêng vì vậy, nó thường được sử dụng như một tên biến. |
| Trong một chương trình C, nhiều hơn một câu lệnh ngắt có thể được thực thi. | Trong chương trình C, chỉ một hàm thoát sẽ được thực thi. |

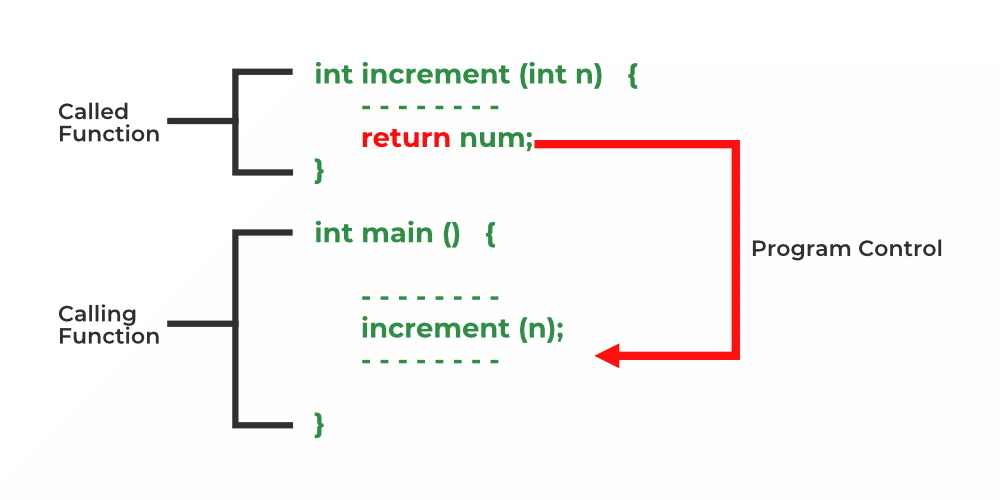
#### Continue ( bỏ qua lệnh sau vòng lặp )

Nếu một vòng lặp đang chạy mà gặp lệnh continue, tất cả các lệnh trong thân vòng lặp nằm phía dưới lệnh continue sẽ bị bỏ qua ở lần lặp hiện tại. Vòng lặp sẽ chuyển sang kiểm tra điều kiện và tiếp tục lặp ở lần tiếp theo(nếu điều kiện lặp còn thỏa mãn).

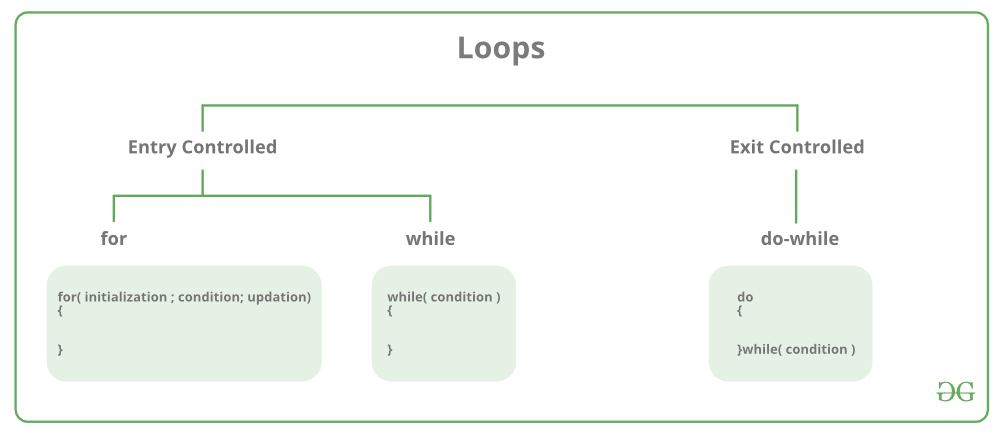


#### Return

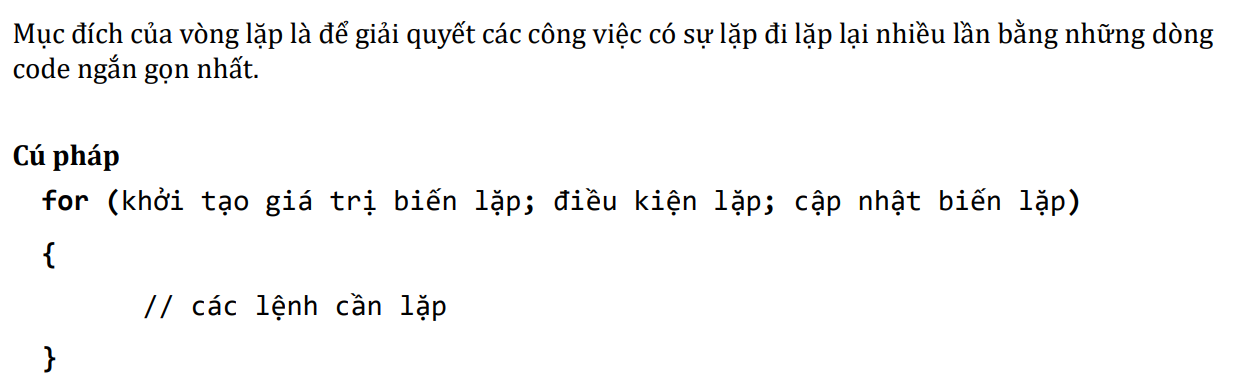
Giả về giá trị cho hàm.

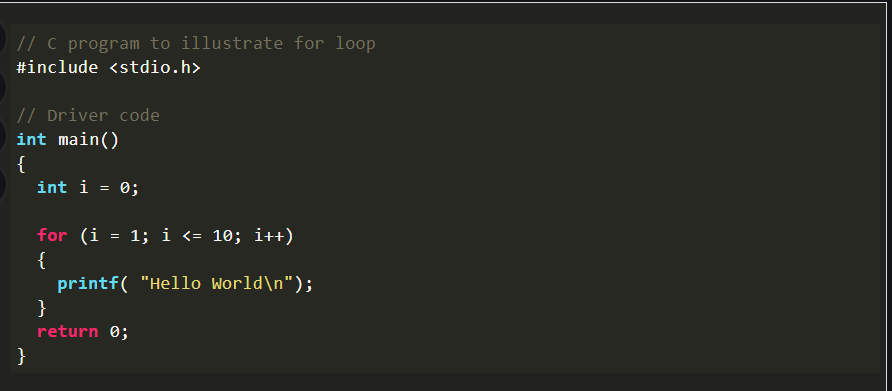


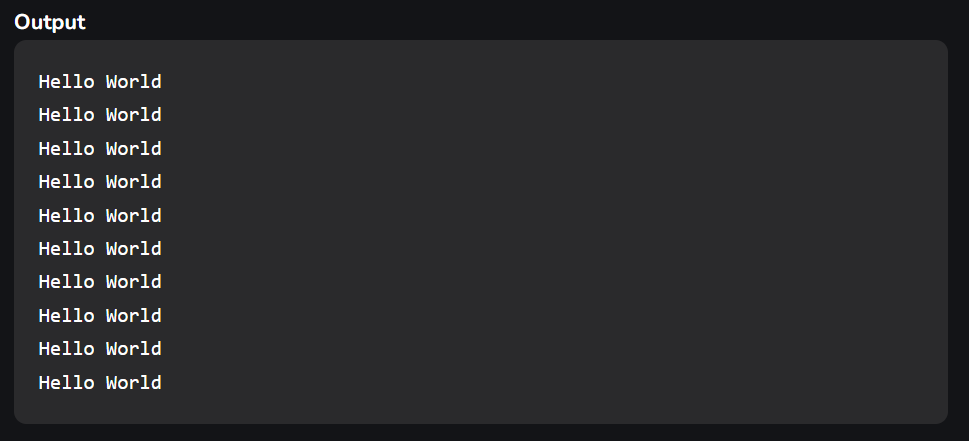
# LOOP ( Hàm lặp)



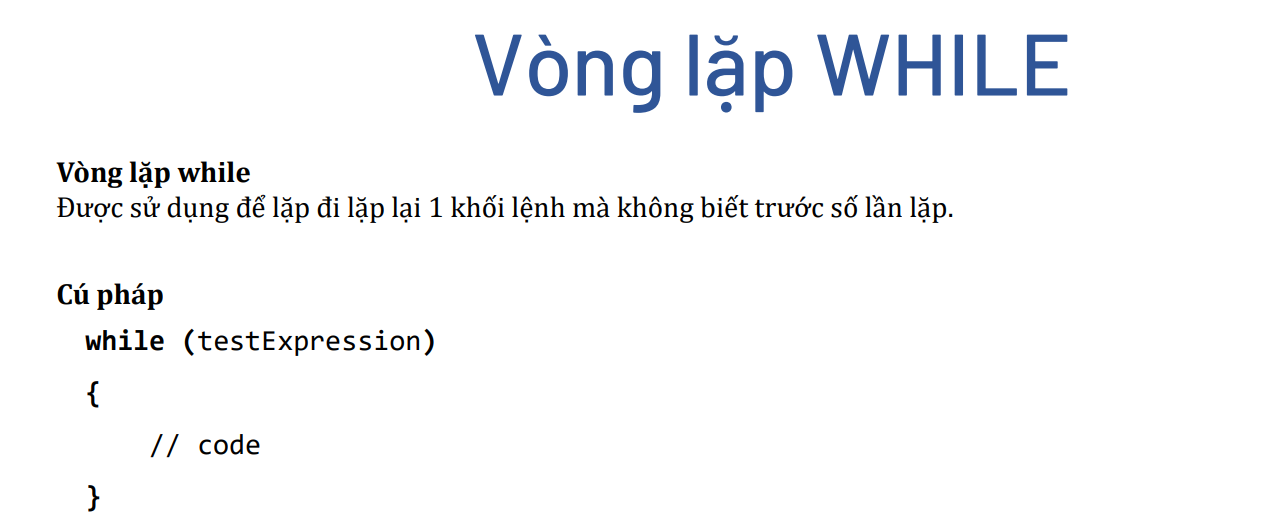
## For

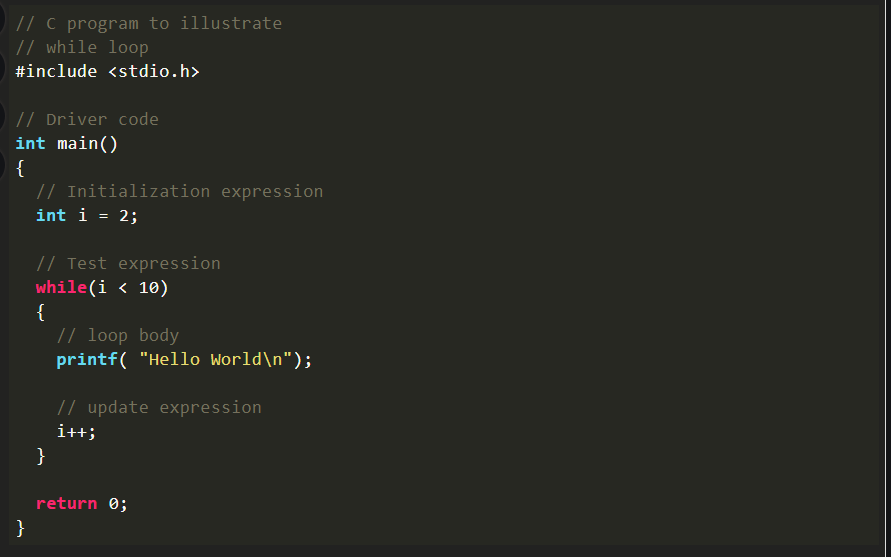


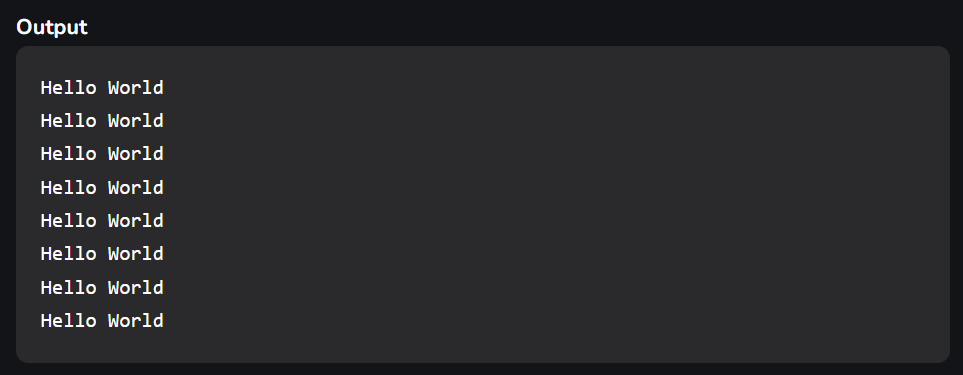




## While







## do-while Loop

