BIẾN VÀ ĐỊNH DANH BIẾN

# **Biến (variable) trong C++ và tác dụng của biến**

Biến (variable) trong C++ là một khối lưu trữ trong bộ nhớ được dùng để lưu trữ giá trị. Biến có thể được sử dụng để lưu trữ các kiểu dữ liệu khác nhau như số nguyên, số thực, chuỗi, ký tự, v.v.

Các tác dụng của biến trong C++ bao gồm:

* Lưu trữ giá trị: Biến được sử dụng để lưu trữ giá trị của một biểu thức hoặc một giá trị đơn lẻ. Ví dụ, int a = 5; sẽ khai báo biến a là một số nguyên và gán giá trị 5 cho nó.
* Tính toán: Biến có thể được sử dụng để tính toán kết quả của một biểu thức. Ví dụ, nếu có hai biến a và b, ta có thể tính toán tổng của chúng bằng cách sử dụng phép tính a + b.
* Truyền tham số: Biến có thể được sử dụng để truyền giá trị cho một hàm. Ví dụ, khi gọi một hàm với tham số là một biến, giá trị của biến đó sẽ được truyền vào hàm.
* Sử dụng trong các câu lệnh điều kiện: Biến có thể được sử dụng trong các câu lệnh điều kiện để kiểm tra giá trị của nó. Ví dụ, nếu có một biến a, ta có thể kiểm tra nó có lớn hơn hoặc nhỏ hơn một giá trị khác bằng cách sử dụng câu lệnh if (a > 10) hoặc if (a < 10).
* Lưu trữ tạm thời: Biến có thể được sử dụng để lưu trữ tạm thời một giá trị hoặc kết quả của một phép tính để sử dụng trong các phép tính khác. Ví dụ, nếu có một biến a và b, ta có thể lưu trữ kết quả của a + b vào một biến khác để sử dụng cho các phép tính khác.
  + Nói ngắn ngọn & dễ hiểu hơn : biến (variable) như một vùng nhớ để lưu trữ dữ liệu, giá trị xuyên suốt quá trình chương trình chạy & thực thi.
  + Khi chương trình chạy thì biến đã khai báo sẽ đóng vai trò như một vùng nhớ để lưu trữ các dữ liệu & giá trị và khi chương trình kết thúc thì biến cũng kết thúc.
  + Lưu ý : biến chiếm một vùng trong bộ nhớ, chúng ta có thể sử dụng toán tử &<tên biến> để truy xuất vùng nhớ (hay còn gọi là address) của biến đấy trong vùng nhớ trên máy tính.
  + Ví dụ : Ví dụ : &age = 00000022FEFFFB24 (là dãy số thập lục phân).

# **Các khai báo biến trong C++**

* Khai báo (declaration) biến trong C++ thường sẽ khai báo kiểu dữ liệu trước rồi sau đó sẽ là tên biến.
* Ví dụ cụ thể : int age = 19 -> int (Data Type) kiểu số nguyên (integer) và tên biến là age. Chúng ta gán cho biến age với toán tử gán “=” 19. Giờ đây, biến age có giá trị = 19.
* Chúng ta có thể khai báo nhiều biến trên cùng một kiểu dữ liệu. Ví dụ int age, tuoi, year… mỗi tên biến cách nhau bằng một dấu “,”.

# **Quy tắc đặt tên biến trong C++ (identifier) – định danh :**

1. Biến không được bắt đầu bằng số.
2. Biến không được có ký tự đặc biệt nào khác ngoài “\_”.
3. Biến không được trùng với keyword.
4. Biến phân biệt chữ hoa và chữ thường (age – aGe) là 2 biến khác nhau.

KIỂU DỮ LIỆU TRONG C/C++ LÀ GÌ?

# **Kiểu dữ liệu trong C++ là gì?**

Kiểu dữ liệu trong C/C++ là một khái niệm quan trọng trong lập trình, nó xác định kiểu giá trị mà một biến có thể chứa và loại bỏ những giá trị không phù hợp với kiểu dữ liệu đó. Một số kiểu dữ liệu thông thường trong C/C++ bao gồm:

1. Kiểu số nguyên (integer): kiểu dữ liệu này được sử dụng để lưu trữ các giá trị số nguyên, ví dụ như -1, 0, 1, 2, 3,... Kiểu số nguyên có kích thước khác nhau, phổ biến nhất là int, long và short.
2. Kiểu số thực (floating-point): kiểu dữ liệu này được sử dụng để lưu trữ các giá trị số thực, ví dụ như 3.14, 1.23e-5, 2.0, ... Kiểu số thực có kích thước khác nhau, phổ biến nhất là float và double.
3. Kiểu ký tự (character): kiểu dữ liệu này được sử dụng để lưu trữ một ký tự, ví dụ như 'a', 'b', 'c', ... Kiểu ký tự trong C/C++ có kích thước bằng 1 byte.
4. Kiểu boolean (bool): kiểu dữ liệu này được sử dụng để lưu trữ giá trị true hoặc false.
5. Kiểu con trỏ (pointer): kiểu dữ liệu này được sử dụng để lưu trữ địa chỉ bộ nhớ của một biến khác trong bộ nhớ máy tính.

Ngoài các kiểu dữ liệu trên, C/C++ còn có nhiều kiểu dữ liệu khác như enum, struct, union, và array, và các kiểu dữ liệu được tạo ra bởi người dùng (user-defined types).

# **Kiểu dữ liệu nguyên thuỷ (Primitive) và kiểu dữ liệu mở rộng (derived)**

* Trong lập trình, kiểu dữ liệu là một đặc tính của một biến, xác định loại giá trị mà biến đó có thể lưu trữ. Trong nhiều ngôn ngữ lập trình, kiểu dữ liệu được chia thành hai loại chính: kiểu dữ liệu nguyên thuỷ (primitive) và kiểu dữ liệu mở rộng (derived).
* Kiểu dữ liệu nguyên thuỷ là các kiểu dữ liệu đơn giản nhất và cơ bản nhất được hỗ trợ bởi ngôn ngữ lập trình. Các kiểu dữ liệu nguyên thuỷ bao gồm các loại số, ký tự và boolean. Các kiểu dữ liệu này thường được lưu trữ trong bộ nhớ của máy tính dưới dạng nhị phân và được xử lý nhanh chóng.
* Các kiểu dữ liệu mở rộng là các kiểu dữ liệu được tạo ra bằng cách kết hợp hoặc mở rộng các kiểu dữ liệu nguyên thuỷ. Các kiểu dữ liệu mở rộng bao gồm mảng, chuỗi, cấu trúc, đối tượng, lớp, union và enum. Các kiểu dữ liệu này có thể được định nghĩa và tùy chỉnh bởi người dùng hoặc các nhà phát triển để đáp ứng nhu cầu cụ thể của chương trình.
* Sự khác biệt chính giữa các kiểu dữ liệu nguyên thuỷ và mở rộng là kiểu dữ liệu nguyên thuỷ được hỗ trợ bởi ngôn ngữ lập trình và được xử lý nhanh chóng, trong khi kiểu dữ liệu mở rộng được tạo ra bởi người dùng hoặc các nhà phát triển và thường được xử lý chậm hơn.

# **Kiểu nguyên thuỷ (Primitive data types)**

Là kiểu dữ liệu cơ bản nhất được hỗ trợ trong C++, là những kiểu dữ liệu không thể băm nhỏ ra nữa.

1. **Kiểu số nguyên (Integer)** : được sử dụng để lưu trữ các số nguyên âm, dương và số 0.
   * Lưu ý : kiểu dữ liệu số nguyên (Integer) có kích thước là 4 byte (32 bit) với giới hạn là : -2^31 đến 2^31-1 tương đương với -2,147,483,648 đến 2,147,483,647.
2. **Kiểu số thực (Floating – point)** : kiểu dữ liệu này dùng để lưu trữ các số thực (real – number) bao gồm cả số thập phân.
   * Lưu ý : Kiểu dữ liệu số thực (float) có kích thước là 4 byte với giới hạn là xấp xỉ 7 chữ số thập phân.
3. **Kiểu số thực (Double)** : kiểu dữ liệu này cũng giống float nhưng sẽ có giới hạn lưu trữ rộng hơn cũng như kích thước lớn hơn float.
   * Lưu ý : kiểu dữ liệu số thực (double) có kích thước 8 byte (64 bit) được lưu trữ theo chuẩn IEEE – 754. Với giới hạn là xấp xỉ 15 chữ số sau phần thập phân.
4. **Kiểu ký tự (Character – Char)** : kiểu dữ liệu này dùng để lưu trữ các ký tự.
   * Lưu ý : kiểu dữ liệu char chữ lưu trữ được 1 ký tự và ký tự đó được đặt trong cặp dấu nháy ‘đơn’ 🡪 ‘A’.
   * Kích thước của kiểu dữ liệu char là 1 byte.
5. **Kiểu dữ liệu logic (boolean – bool)** : kiểu dữ liệu này chỉ có thể nhận giá trị TRUE or FALSE. Tức là chỉ nhận giá trị 0 (false) và giá trị 1 (true).
   * Lưu ý : kiểu dữ liệu bool thường được sử dụng trong các điều kiện logic như là cấu trúc rẽ nhánh ...
   * Kích thước của bool là 1 byte.

# **Kiểu mở rộng – Derived (dẫn xuất)**

**Pointer (con trỏ)**

Kiểu dữ liệu con trỏ là một kiểu dữ liệu đặc biệt trong các ngôn ngữ lập trình, cho phép ta truy cập và thao tác trên vùng nhớ trong chương trình. Một con trỏ thực chất là một biến lưu trữ địa chỉ của một vùng nhớ khác, và có thể truy cập vào giá trị của vùng nhớ đó thông qua con trỏ.

**Reference (only for C++)**

Trong C++, kiểu dữ liệu reference được sử dụng để tạo một biến tham chiếu (reference variable), cho phép ta truy cập trực tiếp đến giá trị của một biến khác mà không cần phải sao chép giá trị đó. Có thể coi kiểu dữ liệu reference là một cái tên khác cho biến gốc (original variable), vì khi ta thay đổi giá trị của biến tham chiếu, giá trị của biến gốc cũng thay đổi theo.

**Function**

Trong C++, function được coi là một kiểu dữ liệu đặc biệt, cụ thể là kiểu dữ liệu con trỏ hàm.

**Array, struct, class**

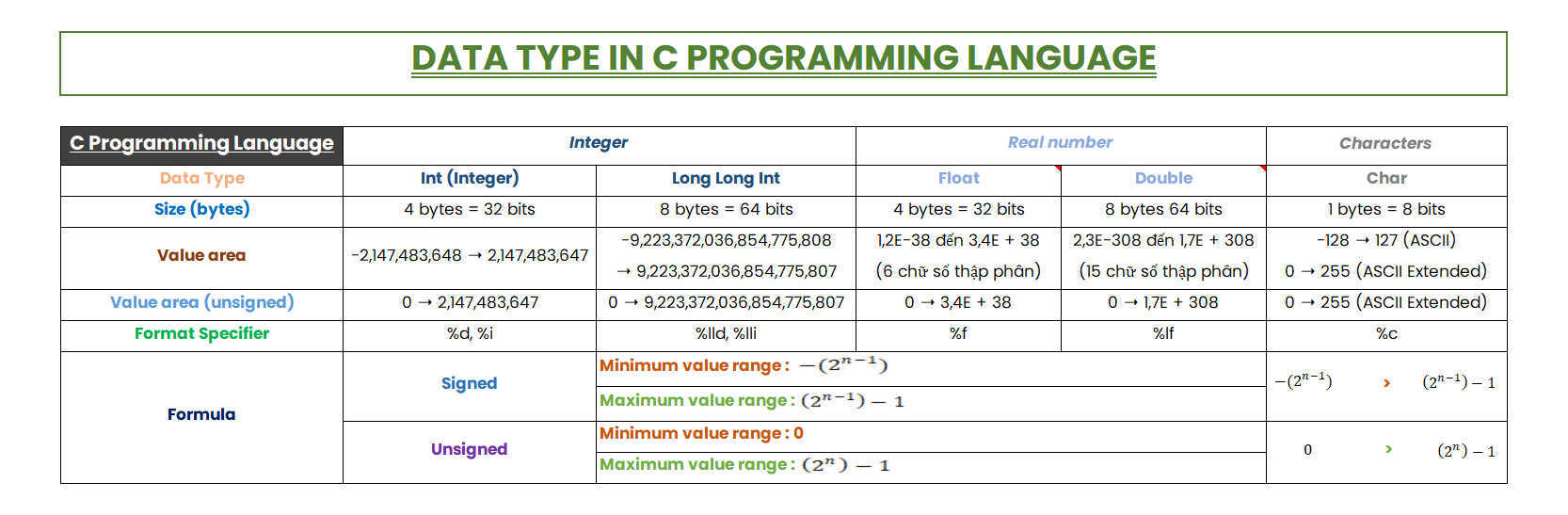
Kiểu dữ liệu này được tìm hiểu ở những bài tiếp theo

**Thư viện STL : std::string – kiểu dữ liệu dạng chuỗi**

Kiểu dữ liệu string được hỗ trợ bởi thư viện STL (Standard Template Library) với khai báo thư viện <string> trong C++.

Kiểu dữ liệu string được lưu trữ một chuỗi ký tự không giới hạn ký tự. Và các chuỗi ký tự được đặt trong cặp nháy “”. Ví dụ “Huynh Khuan Coder”.

**Các kiểu dữ liệu trong C++**

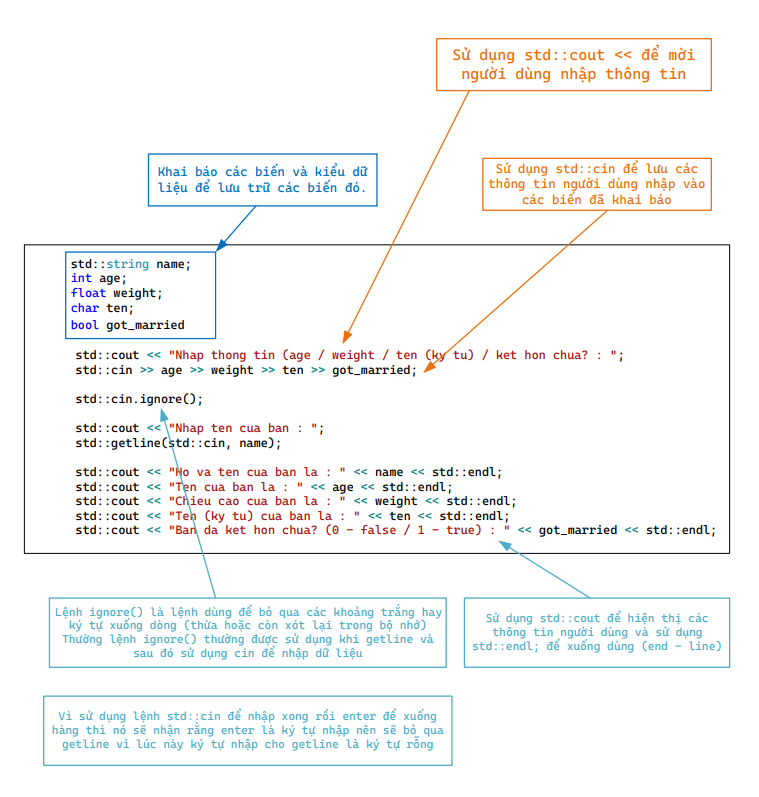


NHẬP XUẤT DỮ LIỆU TRONG C++

# Nhập / xuất dữ liệu là gì?

* Trong C++, nhập/xuất (input/output) là quá trình đưa dữ liệu vào hoặc lấy dữ liệu ra từ một chương trình thông qua các thiết bị như bàn phím, màn hình, tệp tin,... Đây là hoạt động cực kỳ quan trọng trong lập trình để tương tác với người dùng hoặc đọc/ghi dữ liệu vào các tệp tin.
* Nhập là quá trình đưa dữ liệu từ môi trường bên ngoài (bàn phím, màn hình …) vào môi trường console.
* Xuất là quá trình đưa dữ liệu từ môi trường console ra môi trường bên ngoài và hiển thị ra ngoài màn hình.
* Quá trình nhập xuất được tuân theo mô hình chung là IPO (Input – Processing – Output) tức là dữ liệu vào – xử lý dữ liệu – dữ liệu kết quả.

# Quá trình nhập – xuất dữ liệu

Quá trình nhập xuất dữ liệu bao gồm :

1. Dữ liệu vào (input).
2. Xử lý dữ liệu vào (Process).
3. Dữ liệu ra (output).

* Đầu tiên khai báo các biến để lưu trữ dữ liệu.
* Sử dụng lệnh cout để mời người dùng nhập thông tin.
* Sử dụng lệnh cin để nhập dữ liệu từ người dùng và lưu vào các biến.
* Sử dụng lệnh cout để trả ra kết quả người dùng nhập bằng cách gọi các biến (variable) đã được khởi tạo để lưu trữ thông tin người dùng.
* Nếu muốn nhập một chuỗi ký tự dài như là string thì sử dụng hàm getline của thư viện STL (Standard Template Library) vì khi nhập một chuỗi ký tự thì sẽ có dấu cách, mà C++ sẽ ngầm tưởng dấu “space” đó là một ngăn cách giữa các lần nhập dữ liệu, nên sử dụng hàm getline() để lấy dữ liệu trong 1 hàng, tức là enter thì sẽ kết thúc nhập dữ liệu.
* Một lưu ý là : Khi sử dụng cin >> trước để nhập dữ liệu hay sử dụng getline() trước thì khi enter chương trình sẽ hiểu là ký tự enter đó là ký tự cho dòng getline() hoặc là dùng cin >> tức là hệ thống sẽ nhận một ký tự rỗng khi nhập dữ liệu trong hàng getline() vì khi đã enter thì hệ thống nhận enter đó là một ký tự (trong trường hợp này sẽ là ký tự rỗng).
* Để khắc phục trường hợp này thì nên sử dụng lệnh ignore() giữa cin >> và getline() để xoá bỏ ký tự khoảng trắng hoặc ký tự enter (new line).
* Đây là một lỗi rất hay mắc phải khi nhập dữ liệu kiểu cin >> sau đó là nhập dữ liệu kiểu chuỗi getline().

# Tóm lại :

* Nhập là cin >> (Viết tắt : Character Input). Và nhập vào sẽ là một Character, tức là sẽ là ký tự chứ không phải là số, nhưng hệ thống sẽ tự chuyển đổi một ký tự sang kiểu số nguyên, số thức, bool tuỳ vào kiểu dữ liệu (data type) mà chúng ta khai báo cho biến đó.
* Vì trong ngôn ngữ C++ (không chỉ C++ mà hầu hết ngôn ngữ khác đều vậy) là ký tự hoặc chuỗi ký tự với số là khác nhau.
* Ví dụ : “30” khác với 30 🡪 Vì “30” là chuỗi ký tự 30 được đánh dấu bởi cặp dấu nháy kép. Còn 30 là hệ thống sẽ hiểu là một dạng số nguyên hay số thực tuỳ vào kiểu dữ liệu (data type) mà chúng ta khai báo cho biến đó để lưu trữ dữ liệu.
* Xuất là cout << (Viết tắt : Character Output). Là lệnh xuất các chuỗi ký tự ra màn hình console và cout << có thể nối các biến (variable) đã khai báo bằng << biến << để nối tiếp nhau.
* Chúng ta có thể xuất ra một biến bằng lệnh cout.
* Ví dụ : std::cout << “Tuoi cua ban la :” << age << std::endl;
* Lệnh cout có thể hỗ trợ xuất nhiều dữ liệu, nhiều biến khác nhau được ngăn cách bởi hai dấu “<<”.

Vùng nhớ trong C++

# **Vùng nhớ trong C++ là gì**

* Trong C++, vùng nhớ là một khu vực của bộ nhớ trong máy tính được sử dụng để lưu trữ dữ liệu trong quá trình thực thi chương trình. Các biến, mảng, đối tượng và các kiểu dữ liệu khác được lưu trữ trong vùng nhớ.
* Một vùng nhớ được khởi tạo khi chương trình được thực thi [khởi chạy] và kết thúc khi chương trình kết thúc.
* Vùng nhớ được đánh số [Địa chỉ - Address] từng ô, và mỗi ô như vậy có kích thước là 1 byte. Với những kiểu dữ liệu như là int (integer) hay float có kích thước 4 byte thế nên sẽ chiếm một vị trí trong vùng nhớ là 4 byte, và sẽ lấy địa chỉ đầu hoặc cuối làm đại diện cho vùng nhớ đó.
* Lưu ý : Mỗi chương trình được khởi tạo trên hệ điều hành đều chiếm một vùng nhớ trên máy tính, thế như những vùng nhớ này được khởi tạo khi chương trình hoạt động và kết thúc khi chương trình đó kết thúc. Ví dụ như khi khởi chạy chrome thì một vùng nhớ tạm thời được khởi tạo để lưu các thông tin chúng ta truy cập và vùng nhớ được khởi tạo ấy sẽ kết thúc khi chúng ta tắt chương trình.
* Lưu ý : Mỗi vùng nhớ của từng chương trình là khác nhau, độc lập với nhau và không ảnh hưởng gì với nhau. Mỗi một vùng nhớ khi chương trình được thực thi thì các không gian vùng nhớ [memory space] được tạo ra tuỳ thuộc vào từng chương trình dịch khác nhau.
* Những vùng nhớ này được gọi là vùng nhớ ảo [virtual memory] được hệ điều hành cấp phát (tạo ra).
* Mỗi một ô [1 byte] trong vùng nhớ có liên kết tới 1 ô nhớ thật trong RAM và không truy cập trực tiếp vào phần cứng [KHÔNG TRUY CẬP TRỰC TIẾP VÀO PHẦN CỨNG] mà những vùng nhớ ảo này được tạo ra để phục vụ chương trình ấy hoạt động nhờ hệ điều hành cấp phát.

# **Các vùng nhớ tiêu biểu**

## *Vùng nhớ Stack / Automatic Memory (Cơ chế thu hồi vùng nhớ)*

* Đây là vùng nhớ được sử dụng để lưu trữ các biến cục bộ và các tham số của hàm. Khi một hàm được gọi, các biến cục bộ và các tham số của hàm được tạo ra trên stack. Khi hàm kết thúc, các biến cục bộ và các tham số của hàm được hủy bỏ và vùng nhớ trên stack được giải phóng.
* Vùng nhớ Stack được hệ điều hành cung cấp [cấp sẵn] cho vùng nhớ là 1 Mega – Byte tương ứng với ~ 1.000.000 byte (còn tuỳ thuộc vào từng hệ điều hành khác nhau).
  + Lưu ý : Khi chúng ta truy cập và sử dụng hết vùng nhớ Stack mà hệ điều hành cung cấp sẵn cho chương trình thì chương trình sẽ CHẾT. Hay còn lại là lỗi tràn bộ nhớ (Stack – Overflow error). Khi gặp phải lỗi này thì chương trình dừng hoạt động.
* Chúng ta có thể thấy được giới hạn bộ nhớ của vùng nhớ Stack là quá thấp so với các chương trình lớn. Khi gặp các chương trình lớn vượt ngoài bộ nhớ Stack có thể chịu đựng thì chương trình sẽ không hoạt động.

**Vậy câu hỏi đặt ra ở đây là chúng ta có thể tăng giới hạn cũng như là dung lượng của vùng nhớ từ 1 MB tăng lên được không?**

* Câu trả lời là CÓ. Chúng ta hoàn toàn có thể tự xin hệ điều hành, hoặc được gọi là xin cấp phát THỦ CÔNG từ các lập trình viên, và vùng nhớ có thể thực hiện điều tương tự là vùng nhớ HEAP.

**Ưu / nhược điểm của vùng nhớ Stack**

* **Ưu điểm :** Tốc độ truy cập nhanh [Vì các vùng nhớ hay các ô nhớ trong vùng nhớ ảo của vùng nhớ Stack đã được hệ điều hành cấp sẵn cho và tuỳ vào từng hệ điều hành khác nhau mà cho ra dung lượng khác nhau]. Vì đã có sẵn dung lượng mà chúng ta không cần hệ điều hành cấp phát thêm dung lượng vùng nhớ nữa nên tốc độ truy cập cao hơn, cần khai báo hàm, biến hay là một mảng, lớp cũng được truy cập vào vùng nhớ nhanh hơn.
* **Nhược điểm :** Bộ nhớ được cấp phát sẵn, không linh động trong việc dung lượng vùng nhớ thế nên khi gặp các chương trình / dự án lớn khi vượt quá hạn mức dung lượng cho phép thì chương trình sẽ CHẾT.

## *Vùng nhớ Heap [Bộ nhớ linh động]*

* Vùng nhớ HEAP : được sử dụng để cấp phát động bộ nhớ và lưu trữ các đối tượng. Vùng nhớ này được quản lý bởi người lập trình và phải được giải phóng bằng tay khi không còn cần sử dụng.
* Đây là vùng nhớ lớn hơn vùng nhớ Stack, nó rất là lớn đấy. Và kích thước dung lượng của nó không giới hạn, nếu hết dung lượng thì có thể can thiệp và xin cấp phát thêm vùng nhớ / dung lượng từ hệ điều hành.
* Các đối tượng được tạo ra trên heap phải được giải phóng bởi người lập trình bằng cách sử dụng toán tử "delete". Nếu không giải phóng các đối tượng này, chương trình sẽ gặp phải vấn đề về rò rỉ bộ nhớ.
* Bộ nhớ HEAP không cố định về dung lượng, linh hoạt trong việc xin cấp phát vùng nhớ, và có thể xin cấp phát cho tới khi hết RAM trong phần cứng máy tính của chúng ta.
* Chúng ta có thể thấy được sự to lớn về dung lượng cũng như việc linh hoạt xin thêm dung lượng từ hệ điều hành của nó, linh hoạt của vùng nhớ HEAP. Thế nhưng việc xin cấp phát từ hệ điều hành chiếm một tốc độ không nhỏ thế nên việc xin cấp phát cũng kiến tốc độ truy cập vùng nhớ bị chậm lại.

**Ưu / nhược điểm của vùng nhớ Heap**

* **Ưu điểm :** Không giới hạn vùng nhớ, khi hết có thể xin cấp phát thêm từ hệ điều hành.
* **Nhược điểm :** Tốc độ không cao vì tính linh hoạt của nó là hết thì xin cấp phát thêm, khác với Stack là được dựng sẵn vùng nhớ, cần thì truy cập mà không cần can thiệp để cấp phát thêm vùng nhớ.

**Chú ý :** Chỉ có 2 kiểu vùng nhớ **Stack** và **Heap** mới có thể truy cập thủ công vào ĐỊA CHỈ VÙNG NHỚ trên máy tính. Địa chỉ vùng nhớ của một biến hay một mảng sẽ được hiển thị ở dạng **Hexa Decimal** (dạng thập lục phân) – tức là một số trong dãy đó sẽ có 16 ký tự.

## *Vùng nhớ TĨNH [Static Memory] hay [Data Segment]*

* **Vùng nhớ Static** là vùng nhớ tĩnh được lưu trữ suốt quá trình thực thi.
* Vậy khác nhau giữa **memory Static và Stack / Heap** ở chỗ nào?
* Đó là Static được khởi tạo một lần và duy nhất, được nằm yên trong vùng nhớ và không linh động cấp phát tuỳ trường hợp của 2 loại kia. Được lưu trữ và nằm yên trong suốt thời gian chương trình thực thi.
* **Vùng nhớ Static** hay còn gọi là **Data Segment** thường được sử dụng để lưu trữ các biến toàn cục **[global variable]** và các hằng số, khi chương trình kết thúc thì vùng nhớ này cũng được giải phóng .

## *Vùng nhớ Code Segment [Vùng nhớ chứa mã máy]*

* Vùng nhớ dùng để lưu trữ các đoạn mã máy mà máy tính sử dụng để thực thi chương trình chạy.Thường dưới dạng nhị phân [0, 1] mà máy tính có thể hiểu và thực thi.
* Vùng nhớ Code Segment là một vùng nhớ trong bộ nhớ của chương trình được sử dụng để lưu trữ mã máy được biên dịch từ các lệnh và chức năng trong chương trình.
* Vùng nhớ này là một phần của bộ nhớ chỉ đọc, nghĩa là không thể ghi vào đó. Mã máy trong Code Segment được tổ chức và lưu trữ theo cách thức đảm bảo tính liên tục và sắp xếp theo thứ tự thực thi của các lệnh trong chương trình.
* Khi chương trình được khởi động, nó sẽ được nạp vào bộ nhớ và mã máy sẽ được thực thi dựa trên luồng điều khiển của chương trình. Code Segment được quản lý bởi hệ điều hành và không thể thay đổi nội dung trong quá trình chạy chương trình.
* Tóm lại là vùng nhớ Code Segment là vùng nhớ dùng để lưu trữ các mã máy của chương trình mà máy tính có thể hiểu và thực thi.

**Chú ý quan trọng :** Là các vùng nhớ rất quan trọng trong việc cấp phát cũng như là lưu trữ các dữ liệu quan trọng bởi vì tuỳ vào từng vùng nhớ khác nhau và chương trình được khởi động khác nhau mà truy cập vào vùng nhớ ảo khác nhau, nên sử dụng hợp lí và linh hoạt tránh gây ra xung đột vùng nhớ.

BIẾN VÀ PHẠM VI CỦA BIẾN / PHẠM VI SỬ DỤNG BIẾN

# **Giới hạn của biến là gì?**

* Mỗi một biến có một giới hạn riêng hay còn gọi là {block} và những giá trị hay biến được khai báo sẽ nằm bên trong giới hạn {block} ấy. Giới hạn trong {Block} ấy được gọi là [Scope].
* Khi chương trình được viết lên từ ngôn ngữ C++, hàm main() sẽ được khai báo và có cặp dấu {} là một khối của hàm main() những biến được khai báo trong đây sẽ được gọi là **biến cục bộ [Block Scope] hay là [Local Scope / Local Variable]**. Và các biến hay các hằng số được khai báo bên ngoài hàm main() sẽ được gọi là **biến toàn cục [Global Scope / Global Variable]**.

# **Block / Local Scope / Biến cục bộ**

* Là các biến được khai báo trong {block} và được lưu trữ trong vùng nhớ Stack (vùng nhớ Stack có cơ chế tự động thu hồi vùng nhớ) khi block ấy kết thì chương trình sẽ giải phóng vùng nhớ ấy.
* Tức là khi chúng ta khai báo {khối 1} chứa các hàm và khi kết thúc block này / thoát ra thì chương trình sẽ giải phóng và tự động thu hồi vùng nhớ đấy.
* Biến cục bộ được lưu trữ trong bộ nhớ Stack (có cơ chế tự động thu hồi vùng nhớ) và mỗi khi thoát khỏi phạm vi được giới hạn trong {block} đó thì bộ nhớ Stack tự động thu hồi các biến và các dữ liệu được khai báo trong phạm vi đó.
* Chúng ta có thể triển khai một {block} khác nằm trong {block} trước đó và có thể trùng tên mà không gây ra lỗi trùng lặp tên. Nhưng trong thực tế không nên khai báo trùng tên biến dễ gây hiểu lầm và khó hiểu.
* Biến cục bộ thường được sử dụng nhiều trong điều khiển luồng trong C++, lợi ích của biến cục bộ là khai báo ngay trên {block} đơn giản và dễ khai báo, khi hoàn thành xong {block} đó thì biến ấy sẽ được thu hồi.
* Nhược điểm của biến cục bộ là sau khi thoát khỏi block thì biến hay các mảng sẽ bị thu hồi thế nên chúng ta phải khai báo một biến chính bên ngoài là con của hàm main().

# **Global Scope / Biến toàn cục**

* Là biến được khai báo ngoài hàm main() nằm ở bên ngoài chương trình, biến toàn cục sẽ không bị thu hồi trong suốt thời gian chương trình thực thi.
* Biến toàn cục được triển khai cho các hằng số hay là các dữ liệu có giá trị không đổi, được gán giá trị tồn tại trong suốt quá trình chương trình thực thi.
* Biến toàn cục khác với biến cục bộ ở chỗ là biến cục bộ lưu trữ trong bộ nhớ Stack còn biến toàn cục được lưu trữ trong bộ nhớ Static (bộ nhớ tĩnh), bộ nhớ này sẽ tồn tại xuyên suốt quá trình chương trình thực thi mà không bị thu hồi, nằm tĩnh và ở yên đó.

# **Sự khác nhau giữa Block / Local Scope và Global Scope**

1. Block / Local Scope phải được định nghĩa trong {block}, còn Global Scope thì không cần định nghĩa trong cặp {block} mà nằm ngoài hàm main().
2. Block / Local Scope sử dụng vùng nhớ Stack (vùng nhớ sẽ thu hồi khi khi thoát khỏi phạm vi block đó. Còn Global Scope thì tồn tại xuyên suốt quá trình chương tình thực thi và được lưu trữ trong bộ nhớ Static, thường được sử dụng để khai báo các hàm hay biến có dữ liệu không đổi, hằng số…
3. Block / Local Scope định nghĩa trong hàm main() sử dụng cho các biến khai báo bình thường, còn Global Scope thường được định nghĩa cho các giá trị không đổi, hằng số.

# **Lợi ích của việc sử dụng biến toàn cục**

* Khởi tạo một biến tồn tại xuyên suốt quá trình chương trình thực thi mà không cần khai báo thêm trong chương trình, giá trị của biến đó là các số không đổi, cố định.
* Thuận lợi trong việc sử dụng, nhanh và dễ dàng hơn khi chỉ cần gọi ra và sử dụng biến đó trong chương trình hàm main().

# **Nhược điểm của việc sử dụng biến toàn cục**

* Khởi tạo một biến toàn cục là chiếm một dung lượng trong bộ nhớ ảo máy tính, được liên kết tới RAM trên máy tính, việc khởi tạo nhiều biến toàn cục sẽ khiến chương trình nặng hơn vì các biến ấy được lưu trữ trong bộ nhớ Static, luôn tồn tại đến khi chương trình kết thúc.

**Kết luận : Hãy sử dụng Block / Local Scope và Global Scope tuỳ vào từng mục đích và yêu cầu, nên sử dụng một cách thận trọng, và tối ưu nhất.**

TOÁN TỬ TRONG C++

TOÁN TỬ SỐ HỌC (ARITHMETIC OPERATOR)

# **TOÁN TỬ SỐ HỌC (ARITHMETIC OPERATOR)**

* + Toán tử số học là các ký hiệu hay còn được gọi là các toán tử (**Operator**) và tham gia cùng các toán hạng (**Operand**) để tạo thành một biểu thức (**Expression**) hay nhiều biểu thức kết hợp lại với nhau (**Expressions**).
  + Tóm lại : Toán tử số học (**Arithmetic Operator**) dùng để thực hiện các phép toán số học, để tính toán cho ra một kết quả nào đó (**return value**) giữa các giá trị số.
  + **Toán tử số học (Arithmetic Operator)** bao gồm :
    1. **Phép cộng (+)** : Thực hiện cộng giữa hai giá trị số hoặc biểu thức.
       - **Lưu ý :** Ngoài ra phép cộng còn có thể sử dụng để nối lớp chuỗi **std::string** hoặc sử dụng hàm nối chuỗi (**concatenation function**­) 🡺 Các chuỗi kí tự cần nối phải được lưu trong một biến và sử dụng toán tử (**Operator**) để thực hiện nối giữa hai biến của lớp **std::string**.
    2. **Phép trừ (-)** : Thực hiện trừ giữa hai giá trị số hoặc các biểu thức.
    3. **Phép nhân (\*)** : Thực hiện nhân giữa hai giá trị số hoặc biểu thức.
    4. **Phép chia (/)** : Thực hiện phép chia giữa hai giá trị, kết quả là một giá trị kiểu số thực nếu ít nhất một trong hai giá trị là số thực.
    5. **Phép chia dư (%)** : Thực hiện phép chia giữa hai giá trị và trả về số dư của phép chia.
  + Trong C++ bạn cung có thể sử dụng một số hàm toán học có sẵn trong thư viện (**cmath**) của **C++** và (**math.h**) của **C thuần**.
    1. **Hàm sqrt() – {Square root}** : Tính căn bậc 2 của một số cố định (**fixed value**) và các biến lưu trữ giá trị (**variable**). Trong hàm (**function**) sẽ không gọi là các toán hạng (**operand**) mà gọi là đối số (**argument**) được truyền vào các hàm (**function**) để trả về một giá trị.
    2. **Hàm pow() – {Power}** : Dùng để tính luỹ thừa giữa hai đối số (**Arguments**), hàm **pow**() truyền vào 2 đố số. Đối số 1 là là cơ số, đối số 2 là là số mũ.
    3. Hàm **cbrt**() : dùng để trả về căn bậc 3 của một đối số.
    4. Hàm **abs**() : Trả về trị tuyệt đối của một số.

# **VÍ DỤ VỀ CÁC TOÁN TỬ SỐ HỌC (ARITHMETIC OPERATOR)**

**Ví dụ về các phép toán số học**

**int a = 10, b = 5;**

**int c = a + b; // c = 15**

**int d = a - b; // d = 5**

**int e = a \* b; // e = 50**

**int f = a / b; // f = 2**

**int g = a % b; // g = 0**

**Ví dụ về các hàm (function) toán học trong thư viện C++**

|  |
| --- |
| **double x = 27.0;**  **double result = cbrt(x); // result = 3.0**  **cout << "cbrt(" << x << ") = " << result << endl;** |
|  |
| **double x = 16.0;**  **double result = sqrt(x); // result = 4.0**  **cout << "sqrt(" << x << ") = " << result << endl;** |
|  |
| **int x = -3;**  **int result = abs(x); // result = 3**  **cout << "abs(" << x << ") = " << result << endl;** |
|  |
| **double x = 2.0;**  **double y = 3.0;**  **double result = pow(x, y); // result = 8.0**  **cout << x << "^" << y << " = " << result << endl;** |

**Ví dụ về phép + nối chuỗi trong C++**

**string s1 = "Hello";**

**string s2 = " world!";**

**string s3 = s1 + s2; // s3 = "Hello world!"**

**cout << s3 << endl;**

# **CÁC LƯU Ý QUAN TRỌNG CỦA TOÁN TỬ SỐ HỌC (ARITHMETIC OPERATOR)**

* + **Phép chia** (**/**) giữa hai số nguyên sẽ trả về một **số nguyên**, loại bỏ phần thập phân. Nếu bạn muốn kết quả là số thực, ít nhất một trong hai số phải là số thực.
  + **Phép chia lấy dư (%)** chỉ áp dụng cho các **số nguyên**. Kết quả là số dư của phép chia.
  + Trong trường hợp sử dụng toán tử + để thực hiện phép cộng hai chuỗi ký tự, toán tử sẽ nối chuỗi thay vì thực hiện phép cộng số học.
  + Nếu một trong hai toán hạng trong phép tính số học là số thực, thì kết quả sẽ là số thực. Trong trường hợp này, toán tử sẽ thực hiện phép tính số thực.
  + Các phép toán số học sẽ được thực hiện theo thứ tự ưu tiên toán học. Để đảm bảo kết quả chính xác, bạn nên sử dụng các dấu ngoặc “()” để xác định thứ tự các phép tính hoặc tăng độ ưu tiên cho biểu thức.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Toán tử số học** | **Cú pháp (Syntax)** | **Số toán hạng tham gia (Ngôi)** | **Độ ưu tiên (Precedence)** | **Thứ tự thực hiện (Associativity)** | **Giá trị trả về (Return Value)** |
| **+** | Là các toán tử bao gồm các toán hạng để tạo thành một biểu thức | Các phép toán số học như : +, -, \*, /, % là 2 toán hạng  trở lên. | Độ ưu tiên trong các phép toán số học (Arithmetic  Operator) là : \* / % trước, + - sau.  Có thể tăng độ ưu tiên cho các phép toán số học khi  thêm cặp ngoặc “()” để tăng độ ưu tiên | Thứ tự thực hiện ➔ Left to Right “➔” | Giá trị trả về của số nguyên thì là số nguyên, của số  thực thì là số thực, giá trị trả về của sqrt() thì là căn  bậc 2 của số.  pow() là luỹ thừa của một số, abs() là trị tuyệt đối của một số. |
| **-** |
| **\*** |
| **/** |
| **%** |
| **Các hàm số học (pow, sqrt, abs…)** | Là các hàm (function) trong các thư viện toán học  trong C++ [cmath…] | Các hàm (function) như sqrt(), abs() sẽ là 1 toán hạng  hay được gọi là đối số (Argument) |

TOÁN TỬ QUAN HỆ (RELATION OPERATOR)

# **TOÁN TỬ QUAN HỆ (RELATION OPERATOR)**

* + Toán tử quan hệ trong C++ là các toán tử được sử dụng để so sánh giá trị của hai toán hạng và trả về một giá trị **boolean** (**TRUE** hoặc **FALSE**) để chỉ ra mối quan hệ giữa chúng.
  + Các toán tử quan hệ bao gồm:
    1. **Toán tử so sánh bằng (==)** : Trả về true nếu hai toán hạng bằng nhau, false nếu ngược lại.
    2. **Toán tử so sánh khác (!=)** : Trả về true nếu hai toán hạng khác nhau, false nếu ngược lại.
    3. **Toán tử lớn hơn (>)** : Trả về true nếu toán hạng trái lớn hơn toán hạng phải, false nếu ngược lại.
    4. **Toán tử nhỏ hơn (<)** : Trả về true nếu toán hạng trái nhỏ hơn toán hạng phải, false nếu ngược lại.
    5. **Toán tử lớn hơn hoặc bằng (>=)** : Trả về true nếu toán hạng trái lớn hơn hoặc bằng toán hạng phải, false nếu ngược lại.
    6. **Toán tử nhỏ hơn hoặc bằng (<=)** : Trả về true nếu toán hạng trái nhỏ hơn hoặc bằng toán hạng phải, false nếu ngược lại.
  + Toán tử quan hệ trong C++ thường được sử dụng để kiểm tra mối quan hệ giữa các giá trị, ví dụ như kiểm tra xem một số có lớn hơn hay nhỏ hơn một số khác, hoặc xem hai chuỗi có giống nhau hay khác nhau.
  + Toán tử quan hệ (**Relation Operator**) thường được sử dụng trong cấu trúc rẽ nhánh (**if – else**) hay toán tử 3 ngôi để so sánh các giá trị, biểu thức hay các biến. Nếu là TRUE thì ta thực hiện một hành động nào đó, ngược lại FALSE thì thực hiện hành động khác.

# **CÁC LƯU Ý TRONG TOÁN TỬ QUAN HỆ (RELATION OPERATOR)**

* + Toán tử quan hệ chỉ có thể áp dụng cho các kiểu dữ liệu cơ bản (**primitive data types**) như int, float, double, char, bool, và các kiểu dữ liệu đối tượng (**object types**) được định nghĩa để hỗ trợ toán tử quan hệ.
  + Toán tử quan hệ trả về giá trị **boolean** (**TRUE** hoặc **FALSE**) để chỉ ra kết quả của phép so sánh.
  + **Toán tử so sánh bằng (==)** được sử dụng để so sánh giá trị của hai toán hạng. Để so sánh địa chỉ của hai biến, ta cần sử dụng toán tử con trỏ.
  + **Khi sử dụng toán tử so sánh khác (!=)**, ta phải chắc chắn rằng toán hạng trái và toán hạng phải đều có cùng kiểu dữ liệu, vì nếu không, sẽ xảy ra lỗi.
  + Toán tử quan hệ lớn hơn (>) và nhỏ hơn (<) chỉ áp dụng được cho các kiểu dữ liệu có thể so sánh được như int, float, double và char.
  + Khi sử dụng toán tử quan hệ trong các câu lệnh điều kiện, ta phải luôn luôn sử dụng dấu ngoặc đơn ( ) để bảo đảm thứ tự ưu tiên của các phép tính.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Toán tử quan hệ** | **Cú pháp (Syntax)** | **Số toán hạng tham gia (Ngôi)** | **Độ ưu tiên (Precedence)** | **Thứ tự thực hiện (Associativity)** | **Giá trị trả về (Return Value)** |
| **> ; < (lớn hơn ; nhỏ hơn)** | Là các toán tử (Operator) và thực hiện với 2 toán hạng trở lên (Oprand) để trả về giá trị Logic (TRUE or FALSE)  🡺 Lưu ý : Nếu có 2 toán hạng trở lên thì ta có thể sử dụng toán tử logic để nối các toán hạng lại với nhau. | Phép toán quan hệ (Relation  Operator) có 2 toán hạng tham gia,  hoặc 2 biểu thức để so sánh với nhau. | Độ ưu tiên của phép toán quan hệ <,  <=, >, >= là 9  Còn !=, == là 10 vì vậy chương trình sẽ  thực hiện các phép toán so sánh lớn  hơn, nhỏ hơn trước sau đó mới so  sánh bằng hoặc so sánh khác.  Nếu trong phép toán quan hệ có 2 biểu  thức thì nó sẽ thực hiện biểu thức  (Expression) bên trái xong mới thực  hiện biểu thức bên phải, sau cùng là so  sánh hai biểu thức với nhau. | Thứ tự thực hiện là từ trái sang phải.  ➔ Left to right.  Thực hiện bên trái xong tới thực hiện  bên phải. | Giá trị trả về của các phép toán quan hệ (Relation Operator) là các giá trị 0  và 1, tức là False và True. Các giá trị này sẽ được làm việc nhiều hơn trong các phép toán logic và cấu trúc rẽ nhánh.  ➔ Giá trị trả về luôn là giá trị logic  [0,1]. |
| **(>= ; <=) : lớn hơn, nhỏ hơn hoặc bằng** |
| **==, != : so sánh bằng và so sánh khác** |
| **Toán tử quan hệ trả về TRUE hoặc FALSE (là các giá trị logic)** |

TOÁN TỬ LOGIC (LOGICAL OPERATOR)

# **TOÁN TỬ LOGIC (LOGICAL OPERATOR)**

* + Trong C++, các toán tử **LOGIC** được sử dụng để thực hiện các phép toán logic giữa các giá trị **BOOL** (**TRUE**) or (**FALSE**).
  + Toán tử **LOGIC** bao gồm :
    1. **Phép HỘI (&&) hay còn gọi là toán tử AND** : Được sử dụng để kiểm tra một biểu thức logic (**Logical Expression**) và trả về giá trị TRUE nếu cả 2 biểu thức đều đúng, có 1 biểu thức FALSE thì trả về FALSE nguyên biểu thức đó.
    2. **Phép tuyển (||) hay còn gọi là toán tử OR** : Được sử dụng để kiểm tra biểu thức Logic (**Logical Expression**) và trả về giá trị TRUE nếu 1 trong 2 biểu thức đó đúng, tức là 1 trong 2 là TRUE thì nguyên biểu thức đó trả về giá trị TRUE.
    3. **Phép phủ định (!) hay còn gọi là toán tử NOT** : Được sử dụng để đảo ngược giá trị của một biểu thức. Nếu biểu thức đó là TRUE, thì phủ định của TRUE là FALSE, và ngược lại như thế.
       - **Lưu ý :** Toán tử NOT (Phép phủ định) chỉ làm việc với một toán hạng, phủ định của một toán hạng, trả về TRUE nếu toán hạng logic là FALSE, ngược lại trả về FALSE nếu toán hạng logic đó là TRUE.

# **NHỮNG LƯU Ý CỦA TOÁN TỬ LOGIC (LOGICAL OPERATOR)**

* + **Toán tử NOT (!)** chỉ áp dụng cho một giá trị bool duy nhất.
  + **Toán tử AND (&&) và OR (||)** sẽ chỉ kiểm tra giá trị của biểu thức thứ hai nếu cần thiết. Nếu biểu thức thứ nhất đã xác định kết quả của toán tử, thì biểu thức thứ hai sẽ không được thực thi. Điều này được gọi là "**short-circuit evaluation**".
  + **Khi sử dụng toán tử AND (&&)**, nếu biểu thức đầu tiên là sai, thì kết quả của toán tử sẽ luôn là sai. Do đó, nếu bạn sử dụng nhiều biểu thức AND, hãy đặt biểu thức có thể trả về false trước tiên để tối ưu hiệu suất.
  + Tương tự, khi sử dụng **toán tử OR (||),** nếu biểu thức đầu tiên là đúng, thì kết quả của toán tử sẽ luôn là đúng. Do đó, nếu bạn sử dụng nhiều biểu thức OR, hãy đặt biểu thức có thể trả về true trước tiên để tối ưu hiệu suất.
  + Nếu bạn sử dụng các biểu thức phức tạp trong toán tử logic, hãy sử dụng các dấu ngoặc đơn hoặc ngoặc kép để xác định thứ tự ưu tiên của các phép toán. Ví dụ: (a && b) || c.
  + Lưu ý rằng các toán tử logic chỉ làm việc với giá trị bool và trả về một giá trị bool.
  + Các toán tử Logic thường được sử dụng nhiều trong các cấu trúc rẽ nhánh hay cấu trúc lặp.

|  |  |  |  |
| --- | --- | --- | --- |
| **Toán tử Logic (Logical Operator)** | **Phép HỘI – Toán tử {&&}** | **Phép TUYỂN – Toán tử {||}** | **Phép PHỦ ĐỊNH – Toán tử {!}** |
| **Cú pháp (Syntax)** | Cú pháp của phép toán logic bao gồm : phép hội (AND), phép tuyển (OR) và phép phủ định (!). | | |
| **Toán hạng tham gia (ngôi)** | Phép toán logic có thể có 1 toán hạng tham gia, ví dụ !cuoi\_vo;  Hoặc có 2 toán hạng nhưu các phép hội (AND) và phép tuyển (OR).  Có thể có nhiều hơn 2 toán hạng tham gia và được nối với nhau giữa các phép toán Logic. | | |
| **Độ ưu tiên (Precedence)** | Độ ưu tiên của phép toán logic (Logical Operator) thì độ ưu tiên lần lượt như sau : (! 🡺 && 🡺 ||)  Phép phủ định ➔ Phép hội ➔ Phép tuyển.  Nếu trong phép toán có 3 biểu thức thì chương trình sẽ xét theo độ ưu tiên để thực hiện. Ví dụ : a || b && c.  ➔ Chương trình sẽ thực hiện b && c trước rồi sau đó ra giá trị logic rồi so sánh với a. | | |
| **Thứ tự thực hiện (Associativity)** | Thứ tự thực hiện là từ trái sang phải. Left ➔ Right. | | |
| **Giá trị trả về (Return Value)** | Giá trị trả về của phép toán Logic là giá trị Logic, tức là True hoặc False [1, 0]. Những giá trị logic này sẽ được sử dụng nhiều trong cấu trúc rẽ nhánh trong C++. | | |

TOÁN TỬ SO SÁNH BIT (BITWISE OPERATORS)

# **TOÁN TỬ SO SÁNH BIT (BITWISE OPERATORS)**

* + Trong C++, toán tử so sánh bit được sử dụng để thực hiện các phép toán so sánh bit trên các biến số nguyên. Toán tử này chỉ hoạt động trên các bit đơn lẻ của hai số và thực hiện các phép toán như AND, OR, XOR, NOT trên các bit tương ứng.
  + Các toán tử so sánh bit bao gồm:
    1. **& (AND bit)** : trả về 1 nếu cả hai bit tương ứng đều là 1, ngược lại trả về 0.
    2. **| (OR bit) :** trả về 1 nếu ít nhất một trong hai bit tương ứng là 1, ngược lại trả về 0.
    3. **^ (XOR bit)** : trả về 1 nếu hai bit tương ứng khác nhau, ngược lại trả về 0.
    4. **~ (NOT bit) :** đảo ngược tất cả các bit trong biến.

# **VÍ DỤ VỀ TOÁN TỬ SO SÁNH BIT**

**unsigned int a = 5; // 101 (trong hệ nhị phân)**

**unsigned int b = 3; // 011 (trong hệ nhị phân)**

**Thực hiện so sánh bit**

**unsigned int c = a & b; // 001 (AND bit)**

**unsigned int d = a | b; // 111 (OR bit)**

**unsigned int e = a ^ b; // 110 (XOR bit)**

**unsigned int f = ~a; // 11111111111111111111111111111010 (NOT bit)**

* **Trong ví dụ trên, biến c sẽ có giá trị 1 (101 AND 011 = 001), biến d sẽ có giá trị 7 (101 OR 011 = 111), biến e sẽ có giá trị 6 (101 XOR 011 = 110), và biến f sẽ có giá trị 4294967290 (được tính bằng cách NOT từ 00000000000000000000000000000101).**

# **NHỮNG LƯU Ý VỀ TOÁN TỬ SO SÁNH BIT**

* + Toán tử so sánh bit chỉ hoạt động trên các biến số nguyên, và chỉ áp dụng cho các phép toán AND, OR, XOR, NOT trên các bit của hai số tương ứng.
  + Toán tử NOT bit được áp dụng trên từng bit của biến số, đảo ngược giá trị của từng bit. Nó không đảo ngược toàn bộ giá trị của biến số.
  + Các toán tử so sánh bit sẽ trả về giá trị kiểu nguyên, có thể được gán vào biến kiểu nguyên hoặc sử dụng trong biểu thức.
  + Nếu hai biến số có độ dài khác nhau, các toán tử so sánh bit sẽ hoạt động trên các bit tương ứng trong phạm vi của biến số có độ dài nhỏ hơn.
  + Các toán tử so sánh bit không nên sử dụng trong các tình huống cần tính toán số nguyên lớn hoặc số âm, vì các bit cao nhất có thể được sử dụng để đại diện cho dấu của số nguyên.
  + Nên sử dụng các toán tử so sánh bit khi cần thực hiện các phép toán AND, OR, XOR, NOT trên các bit đơn lẻ của hai số. Ví dụ, các toán tử này có thể được sử dụng để thao tác với các cờ bit (flags) hoặc để mã hóa dữ liệu.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Toán tử so sánh bit** | **& (AND bit)** | **| (OR bit)** | **^ (XOR bit)** | **~ (NOT bit)** |
| **Cú pháp (Syntax)** | Cú pháp của phép toán logic bao gồm : & (AND bit) | (OR bit) ^ (XOR bit) ~ (NOT bit) | | | |
| **Toán hạng tham gia (ngôi)** | Toán hạng (operand) của các toán tử này phải là các biến số nguyên (int) hoặc các biểu thức (expression) trả về giá trị nguyên. Toán tử NOT bit (~) không có toán hạng tham gia vì nó đảo ngược từng bit của biến. | | | |
| **Độ ưu tiên (Precedence)** | Độ ưu tiên của các toán tử so sánh bit là thấp hơn so với các toán tử số học và toán tử quan hệ. Để thay đổi độ ưu tiên của các toán tử so sánh bit, bạn có thể sử dụng các dấu ngoặc đơn hoặc dấu ngoặc kép để xác định thứ tự thực hiện của các biểu thức. | | | |
| **Thứ tự thực hiện (Associativity)** | Thứ tự thực hiện của các toán tử so sánh bit là từ trái sang phải, và các toán tử này thực hiện trên từng bit đơn lẻ của hai số tương ứng.   * + - * + Ví dụ, khi sử dụng toán tử AND bit (&), toán tử này sẽ thực hiện phép toán AND trên bit thứ nhất của hai số, sau đó sẽ thực hiện phép toán trên bit thứ hai, và tiếp tục như vậy đến hết tất cả các bit của hai số. | | | |
| **Giá trị trả về (Return Value)** | Giá trị trả về của các toán tử so sánh bit là một số nguyên, có giá trị tương ứng với kết quả của phép toán trên các bit tương ứng của hai số.   * + - * + Ví dụ, nếu bạn sử dụng toán tử AND bit (&) trên hai số 5 và 3, kết quả sẽ là 1 (101 AND 011 = 001), và giá trị trả về của toán tử này sẽ là 1. | | | |

TOÁN TỬ GÁN (ASSIGNMENT OPERATOR)

# **TOÁN TỬ GÁN (ASSIGNMENT OPERATOR)**

* + **Toán tử gán (assignment operator)** trong C++ là một toán tử được sử dụng để gán giá trị cho một biến. Toán tử này có dạng "=" và được đặt giữa tên biến và giá trị muốn gán cho biến đó.
  + **Phép gán (Assigment Operator)** : là một toán tử thực hiện Copy giá trị hay dữ liệu từ bên phải sang bên trái. Hay còn được gọi là Copy Assignment Operator.
  + Ví dụ : a = b (gán biến b cho a). x = 10 (gán fixed value 10 cho biến x).

# **NHỮNG LƯU Ý VỀ TOÁN TỬ GÁN (ASSIGNMENT OPERATOR)**

* + **Toán tử gán "="** được sử dụng để gán giá trị, khác với toán tử quan hệ **“==”** dùng để so sánh bằng.
  + **Toán tử gán** có thể được sử dụng với hầu hết các kiểu dữ liệu trong C++, bao gồm cả các kiểu dữ liệu nguyên thủy (int, float, double...) và kiểu dữ liệu đối tượng.
  + **Phép gán (Assignment Operator)** chỉ thực hiện Copy giá trị từ bên phải sang một **BIẾN** (**VARIABLE**) bên trái. Tức là chúng ta không thể để giá trị hoặc biểu thức ở bên trái được.
  + **Vị trí** của 2 phép gán cũng không được đảo lại, đảo vị trí lại có thể không sai về mặt cú pháp (**syntax**) như sai về mặt gán các biến.
  + **Phép toán gán cũng được gán cho một biểu thức gán.** Ví dụ : x = y = 5 thì thực hiện gán biểu thức (y = 5) gán 5 cho y sao đó thực hiện gán y cho biến x.
  + Khi khai báo một kiểu dữ liệu là **int** nhưng gán một kiểu dữ liệu khác với int (integer) thì chương trình sẽ **tự động chuyển về kiểu dữ liệu (type casting)** đã được khai báo (**declare**).
  + Khi chúng ta khai báo một biến hằng (**const**) thì giá trị sẽ không thể thay đổi được ➔ **Fixed Value** (**giá trị cố định**). Nếu chúng ta thực hiện gán một giá trị mới (**new value**) cho một const thì chương trình sẽ báo lỗi (**errors**).

|  |  |
| --- | --- |
| **Toán tử gán (Assignment Operator)** | **Phép gán “=” hay còn gọi là Copy Assignment Operator** |
| **Cú pháp (Syntax)** | Cú pháp của phép gán là một toán tử (Operator) được ký hiệu “=”.   * + - * + Lưu ý : Phép gán “=” khác với toán tử quan hệ “==”. Một cái là phép gán, một cái là so sánh bằng. |
| **Toán hạng tham gia (ngôi)** | Số toán hạng tham gia vào phép gán là 2 toán hạng (2 Operand), là toán tử 2 ngôi.   * + - * + Thực hiện gán giá trị bên phải sang bên trái nên được xem là toán tử 2 ngôi và 2 toán hạng tham gia. |
| **Độ ưu tiên (Precedence)** | Toán tử “=” có độ ưu tiên là 16, gần như là cuối cùng. Tức là phép gán sẽ thực hiện sau cùng.   * + - * + Ví dụ : x = y + 10 \* 20; ➔ Thì chương trình sẽ thực hiện biểu thức (y + 10 \* 20) trước rồi mới gán cho biến x. |
| **Thứ tự thực hiện (Associativity)** | Thứ tự thực hiện của phép gán là “Right – to – left” “Phải ➔ trái”. |
| **Giá trị trả về (Return Value)** | Giá trị trả về cho chương trình chính là giá trị mà nó được gán. |

TOÁN TỬ HỖN HỢP (MISC OPERATORS)

# **TOÁN TỬ HỖN HỢP (MISC OPERATORS)**

* + **Toán tử hỗn hợp (Misc Operators) bao gồm :**
    - **Sizeof operator :** Để xác định kích thước của một kiểu dữ liệu trên một máy tính cụ thể. Toán tử sizeof là toán tử một ngôi, nhận vào một kiểu dữ liệu hoặc một biến, và trả về kích thước (byte) của kiểu dữ liệu hoặc biến đó.
      * int (integer), float: 4 byte.
      * long long int, double : 8 byte.
      * char, bool : 1 byte.
    - **Comma operator** : Toán tử comma được sử dụng để tách các biểu thức trong một câu lệnh hoặc tham số trong một hàm. Toán tử này thực hiện từ trái sang phải và trả về giá trị của biểu thức cuối cùng trong danh sách. Toán tử comma thường được sử dụng để viết các biểu thức ngắn gọn hơn trong một câu lệnh hoặc để truyền nhiều tham số vào một hàm.
      * **Chú ý**: Dấu phẩy có ưu tiên thấp nhất trong tất cả các toán tử.
    - **Conditional operator** : Toán tử điều kiện (**?:**) là toán tử 3 ngôi duy nhất trong C++ (**có 3 toán hạng**), tương đương với câu điều kiện (**if/else statements**).
      * **Chú ý**: Chỉ sử dụng toán tử điều kiện với những câu điều kiện đơn giản.
      * **Chú ý**: Toán tử điều kiện **(?:)** có thể là một biểu thức (**expression**), trong khi câu điều kiện (**if**/**else**) chỉ là một câu lệnh (**statements**).
      * Toán tử 3 ngôi có thể thực hiện để trả về một giá trị nếu biểu thức điều kiện là **TRUE**, ngược lại là **FALSE**, và cũng có thể trả về một biến và ta có thể thực hiện các câu lệnh cho biến đó.

# **NHỮNG LƯU Ý CỦA TOÁN TỬ HỖN HỢP**

* + **Toán tử sizeof:**
    - Toán tử **sizeof** không thể sử dụng để tính toán kích thước của một đối tượng được cấp phát động trong chương trình.
    - Khi sử dụng với một mảng, sizeof trả về kích thước của mảng trong bộ nhớ, không phải số lượng phần tử trong mảng.
    - Khi sử dụng với con trỏ, sizeof trả về kích thước của con trỏ, không phải kích thước của đối tượng mà con trỏ trỏ tới.
  + **Toán tử comma:**
    - Toán tử **comma** không được sử dụng để thay thế cho dấu chấm phẩy trong các câu lệnh.
    - Toán tử **comma** thường được sử dụng để viết các biểu thức ngắn gọn hơn trong một câu lệnh hoặc để truyền nhiều tham số vào một hàm.
    - Giá trị của biểu thức trước dấu **comma** trong danh sách sẽ được tính toán trước, sau đó giá trị của biểu thức sau dấu **comma** sẽ được tính toán.
  + **Toán tử điều kiện:**
    - Toán tử điều kiện không nên được sử dụng để thay thế cho câu lệnh if/else trong các trường hợp phức tạp.
    - Điều kiện được đưa ra trong toán tử điều kiện phải có kiểu boolean, nếu không, trình biên dịch sẽ tự động chuyển đổi kiểu.
    - Cả hai giá trị được trả về bởi toán tử điều kiện phải có cùng kiểu dữ liệu.

|  |  |  |  |
| --- | --- | --- | --- |
| **Toán tử hỗn hợp (Misc Operators)** | **Toán tử sizeof** | **Toán tử Comma** | **Toán tử điều kiện** |
| **Cú pháp (Syntax)** | Cú pháp của toán tử hỗn hợp là :   * + - * + Toán tử : **sizeof**(**argument**),         + Toán tử **Commna** : **expression1**, **expression2**   Toán tử điều kiện : **condition** ? **expression1** : **expression2**; | | |
| **Toán hạng tham gia (ngôi)** | Toán hạng tham gia :   * + - * + Toán tử **sizeof**() : chỉ có 1 toán hạng, hay còn gọi là đối số.         + Toán tử **comma** : 2 toán hạng, biểu thức và có thể có nhiều hơn 2.         + **Toán tử điều kiện** : 3 toán hạng, gọi là toán tử 3 ngôi. Bao gồm điều kiện, biểu thức 1 và biểu thức 2. | | |
| **Độ ưu tiên (Precedence)** | Toán tử sizeof có độ ưu tiên cao nhất trong các toán tử này.   * + - * + Chỉ có 1 đối số nên không cần quan tâm lắm đến độ ưu tiên của **sizeof**().   Toán tử **comma** có độ ưu tiên thấp hơn toán tử sizeof.  Toán tử **conditional operator** **(?:)** có độ ưu tiên cao hơn toán tử comma.   * + - * + Thực hiện tính giá trị biểu thức, trả về một giá trị bool **TRUE** or **FALSE** sau đó nếu là TRUE thì thực hiện **expression** 1, **FALSE** thì thực hiện **expression** **2**. | | |
| **Thứ tự thực hiện (Associativity)** | Toán tử **sizeof**() : chỉ có 1 đối số, không có thứ tự thực hiện.  Toán tử **Comma** : từ trái sang phải.  Toán tử điều kiện : từ trái sang phải. | | |
| **Giá trị trả về (Return Value)** | Toán tử sizeof trả về kích thước (số byte) của kiểu dữ liệu hoặc biểu thức được truyền vào.  Toán tử comma trả về giá trị của biểu thức cuối cùng.  Toán tử conditional operator trả về giá trị của biểu thức thứ hai nếu điều kiện là true, ngược lại trả về giá trị của biểu thức thứ ba. | | |

CẤU TRÚC ĐIỀU KHIỂN, CÂU LỆNH RẼ NHÁNH CẤU TRÚC IF

# **CẤU TRÚC IF LÀ GÌ?**

* Cấu trúc rẽ nhánh trong C++ là một cách để thực hiện các hành động khác nhau dựa trên điều kiện được xác định trước đó. Cấu trúc rẽ nhánh có thể được thực hiện bằng cách sử dụng các câu lệnh điều kiện **if-else** hoặc **switch-case**.

# **Cú pháp của cấu trúc if**

if (condition) {

**// Code sẽ được thực thi nếu điều kiện là true**

}

* Hầu hết các câu lệnh khác của C++ thực hiện theo tuần tự, khác với các câu lệnh đó, cấu trúc rẽ nhánh if thực hiện theo một biểu thức điều kiện, nếu **biểu thức** (**expression**) đó đúng thì chương trình sẽ thực hiện các dòng lệnh trong cặp ngoặc {**block**}. Ngược lại nếu điều kiện đó sai thì sẽ kết thúc câu lệnh if và thực hiện dòng tiếp theo hoặc dòng kế tiếp.
* Điều kiện của một cấu trúc rẽ nhánh if là một giá trị **LOGIC** (**bool**), chỉ có True or False. Các biểu thức Logic trong điều kiện của câu lệnh if có thể kể đến như : **==, <, <=, >, >=, !=**. Chúng ta có thể kết hợp giữa hai biểu thức (expression) bằng các toán tử, hay các **phép HỘI (&&) hay là phép TUYỂN (||) hoặc là phép phủ định (!)**.

# **Khi nào nên sử dụng cấu trúc rẽ nhánh if và ứng dụng**

* Khi chúng ta đứng trước một bài toán, chúng ta có nhiều điều kiện cần kiểm tra, nếu điều kiện là True thì ta thực hiện một hành động gì đó, nếu là False thì ta thực hiện một hành động khác.
* **Ví dụ :** Nếu tôi đạt điểm kỳ thi trên 9 thì tôi đạt được điểm A+, ngược lại dưới 9 thì tôi chỉ đạt A hoặc B+.
* **Ví dụ :** Nếu hôm nay trời mưa, tôi sẽ không đi chơi, còn nếu là trời nắng, tôi sẽ đi chơi.
* Nhiều bài toán có hoặc nhiều hơn 2 điều kiện cần kiểm tra, như vậy cấu trúc rẽ nhánh if sinh ra để chúng ta kiểm tra một điều kiện gì đó đúng thì trả về kết quả tương ứng giống như một con Robot đơn giản, chúng ta đặt một câu hỏi gì cho nó sau đó nó tính toán và đưa ra câu trả lời đúng và gần sát với câu hỏi mà chúng ta đưa ra.
* Ngoài ra cấu trúc if còn có thể sử dụng trong vòng lặp để kiểm tra điều kiện gì đó trước khi lặp một giá trị hay là một biểu thức cụ thể.

# **Những lưu ý trong cấu trúc if**

* Cấu trúc **if** kiểm tra biểu thức sau đó trả về một giá trị boolean (**true** hoặc **false**), chứ không phải là một giá trị nào khác.
* Tầm quan trọng của việc **đúng sai** của một biểu thức trong bài toán quyết định rất lớn tới luồng điều khiển mà chương trình thực hiện, nếu chúng ta đặt sai biểu thức thì cấu trúc chương trình sẽ cho ra kết quả khác, vì cấu trúc rẽ nhánh không thực hiện **tuần tự**, mà thực hiện theo các nhánh True or False. Trừ khi kết thúc câu lệnh if thì chương trình mới thực hiện các lệnh kế tiếp nếu còn các dòng lệnh.
* Nếu hành động trong ngoặc {**block**} chỉ có 1 câu lệnh thì chúng ta hoàn toàn có thể không cần sử dụng đến ngoặc {block} đó mà hoàn toàn có thể lược bỏ đi cũng như có thể ghi trên cùng 1 dòng.
* **Ví dụ :** if (age > 18) cout << “Ban da truong thanh!” << endl;
* Còn nếu nhiều hơn một dòng lệnh thì chúng ta phải bọc nó trong cặp ngoặc nhọn {**block**}, để chương trình hiểu đây là một khối lệnh của cấu trúc **if**.
* Khi chúng ta cần nối nhiều điều kiện với nhau, nhiều biểu thức lại với nhau để cùng so sánh vừa đưa ra kết quả cuối cùng chỉ có 1 giá trị (**True** / **False**) thì chúng ta có thể sử dụng các toán tử **Logic** : **Phép hội (&&), phép tuyển (||), phép phủ định (!)**.
* Nếu gặp một biểu thức so sánh có độ khó phức tạp hoặc nhiều biểu thức với nhau thì nên sử dụng cặp ngoặc () để tăng độ ưu tiên của các **biểu thức** (**expression**) hay là các **toán hạng** (**operand**).

# **Câu hỏi đặt ra về vùng nhớ, khối lệnh**

Khi chúng ta khởi tạo một cấu trúc if thì sẽ có một khối {block} để chứa các câu lệnh cho cấu trúc if thực hiện. Vậy câu hỏi đặt ra ở đây là khi chúng ta khởi tạo một biến (variable) trong {block} của cấu trúc if đó khi kết thúc cấu trúc if thì có thể sử dụng được không?

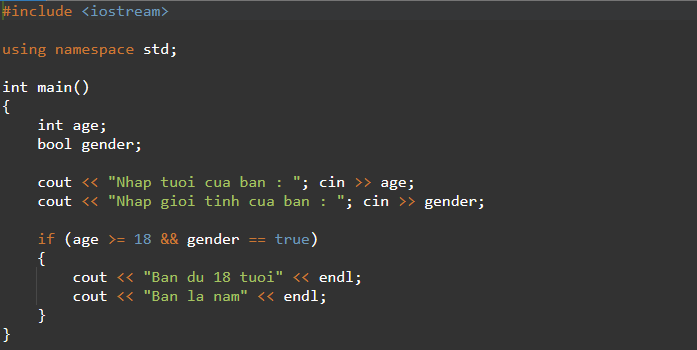
Câu trả lời là KHÔNG, vì như chúng ta đã biết, khi chúng ta khởi tạo một {block} đồng nghĩa chúng ta đang làm việc với bộ nhớ STACK, khi chúng ta khởi tạo một hàm main() và tạo các biến giá trị nằm ở trong đấy thì những biến (variable) đó đều được khởi tạo và nằm trong bộ nhớ STACK, khi chúng ta kết thúc một {block} thì những biến (variable), giá trị (value) trong khối lệnh đó cũng bị thu hồi. Ngoại trừ trường hợp chúng ta khai báo biến với [Global Scope] tức là [Biến toàn cục] thì chúng ta hoàn toàn có thể sử dụng biến (variable) đó ở bất kì đâu trong chương trình, ngoài {block} hay trong {block} khi kết thúc cũng không bị thu hồi. Vì các biến đó được lưu trữ trong vùng nhớ STATIC, vùng nhớ tĩnh và nằm yên ở đó đến khi chương trình kết thúc thì mới được giải phóng.

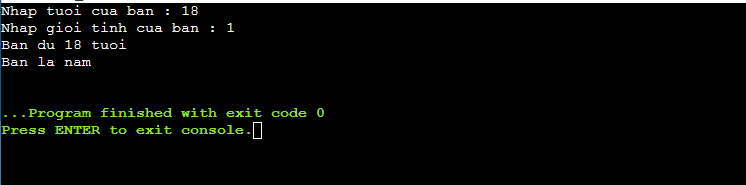
Kết lại, thì chúng ta nên lưu ý việc sử dụng biến trong {block} trong cấu trúc if. Nếu chúng ta muốn thay đổi một biến (variable) thì chúng ta nên khai báo một biến ngoài hàm main() để có thể sử dụng trong cấu trúc if và sau khi thoát khỏi if thì giá trị của biến đó sẽ được cập nhật.

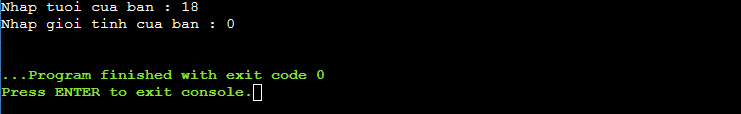
**Như chúng ta có thể thấy, việc sử dụng biến và các khối lệnh là quan trọng quyết định tính chất cũng như đúng sai của một chương trình, cấu trúc if cung cấp cho chúng ta những luồng điều khiển chương trình đi theo từng hướng, những giá trị trả về chỉ có True or False.**

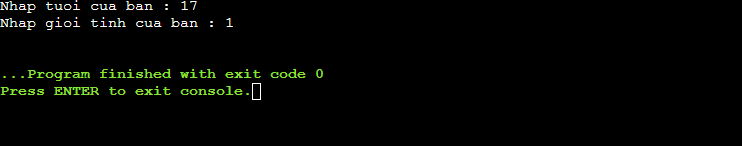
**Tóm lại, cấu trúc rẽ nhánh (if) trong C++ là một cách để thực hiện các hành động khác nhau dựa trên điều kiện được xác định trước đó.**

# **Các ví dụ về cấu trúc if trong C++**









CẤU TRÚC ĐIỀU KHIỂN, CÂU LỆNH RẼ

NHÁNH CẤU TRÚC IF ELSE VÀ IF ELSE IF

# **CẤU TRÚC IF ELSE LÀ GÌ?**

* Cấu trúc rẽ nhánh trong C++ là một cách để thực hiện các hành động khác nhau dựa trên điều kiện được xác định trước đó. Cấu trúc rẽ nhánh có thể được thực hiện bằng cách sử dụng các câu lệnh điều kiện **if-else** hoặc **switch-case**.
* Cấu trúc if else có thể đưa ra kết quả khi mà biểu thức câu lệnh if sai hoặc điều kiện không thoả mãn câu lệnh **IF**. Khi đó ta sẽ sử dụng cấu trúc **if – else** (thường thì cấu trúc if sẽ hay đi với else để tạo thành một cấu trúc rẽ nhánh hoàn chỉnh). Nếu biểu thức if đúng, trả về giá trị True, thì khối lệnh của if sẽ được thực hiện. Ngược lại, nếu biểu thức **if** **sai**, thì ta sẽ trả về kết quả của false, khi mà biểu thức **if** **sai**.
* **Ví dụ :**

#include <iostream>

int main()

{

int age;

std::cout << "Nhap tuoi : "; std::cin >> age;

if (age >= 18)

{

std::cout << "Ban da du 18 tuoi!" << std::endl;

}

else

{

std::cout << "Ban chua du 18 tuoi!" << std::endl;

}

return 0;

}

* + Khi nhập vào số **tuổi >= 18** thì chương trình sẽ thực hiện khối lệnh if.
  + Ngược lại, khi người dùng nhập số **tuổi < 18 tức là 17** trở xuống thì chương trình sẽ thực hiện cấu trúc else.
* Cấu trúc rẽ nhánh **if else if** là cấu trúc rẽ nhánh có thể cho phép nhiều điều kiện trong một cấu trúc rẽ nhánh, trên 2 điều kiện, khi đó có thể sử dụng cấu trúc if, else if.
* **Ví dụ :**

#include <iostream>

int main()

{

int age;

std::cout << "Nhap tuoi : "; std::cin >> age;

if (age >= 18)

{

std::cout << "Ban da du 18 tuoi!" << std::endl;

}

else if (age > 15 && age < 18)

{

std::cout << "Ban du tuoi vi thanh nien!" << std::endl;

}

else

{

std::cout << "Ban chua du 18 tuoi!" << std::endl;

}

return 0;

}

* + Khi nhập số **tuổi >= 18** thì khối lệnh if được thực hiện, ngược lại khi nhập từ 15 < age < 18 thì khối lệnh else if được thực hiện, ngược lại dưới 16 tuổi thì khối lệnh else được thực hiện.
* ***Lưu ý*** *: Có thể sử dụng nhiều cấu trúc else if trong cùng một cấu trúc rẽ nhánh. Và if luôn nằm cuối cùng, else luôn nằm ở sau cùng.*

# **Cú pháp của cấu trúc if**

if (condition) {

**// Code sẽ được thực thi nếu điều kiện là true**

}

* Hầu hết các câu lệnh khác của C++ thực hiện theo tuần tự, khác với các câu lệnh đó, cấu trúc rẽ nhánh if thực hiện theo một biểu thức điều kiện, nếu **biểu thức** (**expression**) đó đúng thì chương trình sẽ thực hiện các dòng lệnh trong cặp ngoặc {**block**}. Ngược lại nếu điều kiện đó sai thì sẽ kết thúc câu lệnh if và thực hiện dòng tiếp theo hoặc dòng kế tiếp.
* Điều kiện của một cấu trúc rẽ nhánh if là một giá trị **LOGIC** (**bool**), chỉ có True or False. Các biểu thức Logic trong điều kiện của câu lệnh if có thể kể đến như : **==, <, <=, >, >=, !=**. Chúng ta có thể kết hợp giữa hai biểu thức (expression) bằng các toán tử, hay các **phép HỘI (&&) hay là phép TUYỂN (||) hoặc là phép phủ định (!)**.

# **Cú pháp của cấu trúc if – else**

if (condition) {

**// Code sẽ được thực thi nếu điều kiện là true**

}

else {

**// Code sẽ được thực hiện nếu điều kiện của if là false**

}

# **Cú pháp của cấu trúc if – else if – else**

if (condition) {

**// Code sẽ được thực thi nếu điều kiện là true**

}

else if (condition) {

**// Code sẽ được thực hiện nếu điều kiện của if là false**

}

else {

**// Code sẽ được thực hiện nếu điều kiện của else if là false**

}

* Các điều kiện của cấu trúc if else if phải luôn không trùng với nhau.
* Khi thực hiện thì chương trình sẽ chạy tuần tự từ trên xuống dưới, nếu có điều kiện nào đúng thì nó thực hiện khối lệnh của điều kiện đó sau đó thoát luôn ra khỏi cấu trúc.
* Cấu trúc if luôn được đặt trên cùng để kiểm tra biểu thức điều kiện đầu tiên, sau đó tiếp đến các điều kiện tiếp theo đến điều kiện (n). Cho tới khi kết thúc mới là else luôn được đặt ở sau cùng.

# **Khi nào nên sử dụng cấu trúc rẽ nhánh if và ứng dụng**

* Khi chúng ta đứng trước một bài toán, chúng ta có nhiều điều kiện cần kiểm tra, nếu điều kiện là True thì ta thực hiện một hành động gì đó, nếu là False thì ta thực hiện một hành động khác.
* **Ví dụ :** Nếu tôi đạt điểm kỳ thi trên 9 thì tôi đạt được điểm A+, ngược lại dưới 9 thì tôi chỉ đạt A hoặc B+.
* **Ví dụ :** Nếu hôm nay trời mưa, tôi sẽ không đi chơi, còn nếu là trời nắng, tôi sẽ đi chơi.
* Nhiều bài toán có hoặc nhiều hơn 2 điều kiện cần kiểm tra, như vậy cấu trúc rẽ nhánh if sinh ra để chúng ta kiểm tra một điều kiện gì đó đúng thì trả về kết quả tương ứng giống như một con **Robot** đơn giản, chúng ta đặt một câu hỏi gì cho nó sau đó nó tính toán và đưa ra câu trả lời đúng và gần sát với câu hỏi mà chúng ta đưa ra.
* Ngoài ra cấu trúc if còn có thể sử dụng trong vòng lặp để kiểm tra điều kiện gì đó trước khi lặp một giá trị hay là một biểu thức cụ thể.

# **Khi nào nên sử dụng cấu trúc rẽ nhánh if else if else và ứng dụng**

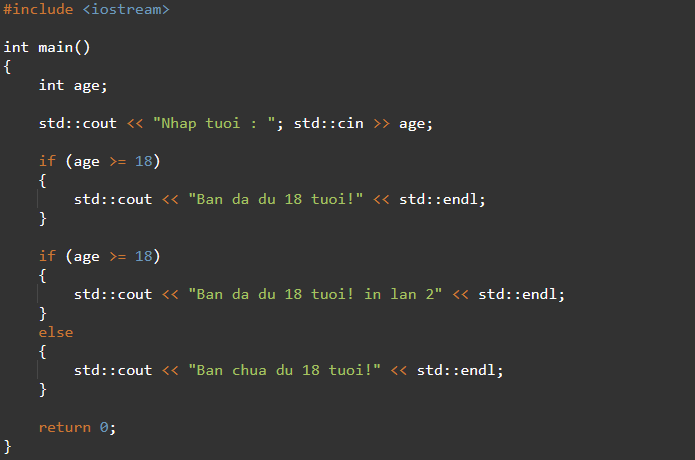
* Khi chúng ta cần kiểm tra nhiều điều kiện mà cấu trúc if không thể đáp ứng, khi đó ta sử dụng cấu trúc if else if để kiểm tra lần lượt các điều kiện khi có điều kiện đúng thì thực hiện khối lệnh của điều kiện đó và thoát khỏi cấu trúc rẽ nhánh.
* **Ví dụ :** Xét thang điểm từ **0 🡪 10**.
  + **Từ 8 trở lên là HSG.**
  + **Từ 6.5 đến dưới 8 là HSK.**
  + **Từ 5.0 đến dưới 6.5 là HSTB.**
  + **Dưới 5.0 là HSY.**
* Khi đó ta sử dụng cấu trúc **if else if** để xét lần lượt các điều kiện rồi đưa ra kết quả cuối cùng 🡺 Ví dụ khi nhập 6.4 thì sẽ thực hiện khối lệnh từ 5.0 đến dưới 6.5.
* Cấu trúc này thực hiện tuần tự từ trên xuống dưới, khi có điều kiện thoả mãn thì thực hiện và thoát cấu trúc, nếu không có điều kiện nào trả về **True** thì chương trình sẽ thực hiện nhánh **else**.

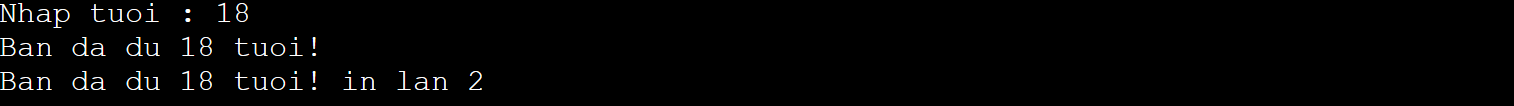
# **Những lưu ý trong cấu trúc if**

* Cấu trúc **if** kiểm tra biểu thức sau đó trả về một giá trị boolean (**true** hoặc **false**), chứ không phải là một giá trị nào khác.
* Tầm quan trọng của việc **đúng sai** của một biểu thức trong bài toán quyết định rất lớn tới luồng điều khiển mà chương trình thực hiện, nếu chúng ta đặt sai biểu thức thì cấu trúc chương trình sẽ cho ra kết quả khác, vì cấu trúc rẽ nhánh không thực hiện **tuần tự**, mà thực hiện theo các nhánh True or False. Trừ khi kết thúc câu lệnh if thì chương trình mới thực hiện các lệnh kế tiếp nếu còn các dòng lệnh.
* Nếu hành động trong ngoặc {**block**} chỉ có 1 câu lệnh thì chúng ta hoàn toàn có thể không cần sử dụng đến ngoặc {block} đó mà hoàn toàn có thể lược bỏ đi cũng như có thể ghi trên cùng 1 dòng.
* **Ví dụ :** if (age > 18) cout << “Ban da truong thanh!” << endl;
* Còn nếu nhiều hơn một dòng lệnh thì chúng ta phải bọc nó trong cặp ngoặc nhọn {**block**}, để chương trình hiểu đây là một khối lệnh của cấu trúc **if**.
* Khi chúng ta cần nối nhiều điều kiện với nhau, nhiều biểu thức lại với nhau để cùng so sánh vừa đưa ra kết quả cuối cùng chỉ có 1 giá trị (**True** / **False**) thì chúng ta có thể sử dụng các toán tử **Logic** : **Phép hội (&&), phép tuyển (||), phép phủ định (!)**.
* Nếu gặp một biểu thức so sánh có độ khó phức tạp hoặc nhiều biểu thức với nhau thì nên sử dụng cặp ngoặc () để tăng độ ưu tiên của các **biểu thức** (**expression**) hay là các **toán hạng** (**operand**).

# **Những lưu ý trong cấu trúc if else if**

* Cấu trúc if else if được sử dụng để kiểm tra các trường hợp đặc biệt hay các điều kiện phức tạp do đó các điều kiện khi chúng ta xét phải đúng khi sử dụng các toán tử Logic hay xét từ khoảng nào đến khoảng nào.
* Cấu trúc rẽ nhánh thực hiện tuần từ đầu trên xuống do đó nếu có điều kiện nào thoả mãn thì thực hiện khối lệnh của nhánh đó và thoát khỏi cấu trúc rẽ nhánh.
* **Nếu bạn sử dụng 2 điều kiện giống nhau :** 
  + **Ví dụ :** if (age >= 18) ; else if (age >= 18).
  + Thì chương trình sẽ thực hiện nhánh if sau đó thoát khỏi cấu trúc vì cấu trúc if else if thực hiện tuần tự từ trên xuống. Nên khi bạn để 2 điều kiện giống nhau thì sẽ không có thực hiện được cả 2 dòng lệnh rẽ nhánh.
* Khi chúng ta sử dụng 2 if, thì 2 khối lệnh if này thực hiện độc lập nhau, tách rời nhau hoàn toàn và không ảnh hưởng gì đến nhau. Khi thực hiện xong if đầu tiên thì chương trình tiếp tục xét đến if tiếp theo rồi thực hiện.
* **Ví dụ :**





* Khi đó chương trình sẽ thực hiện tuần tự từ trên xuống dưới, 2 lệnh if là hoàn toành tách biệt nhau, cho nên 2 if này được xem độc lập và đều được xét điều kiện khi chương trình chạy.

# **Câu hỏi đặt ra về vùng nhớ, khối lệnh**

Khi chúng ta khởi tạo một cấu trúc if thì sẽ có một khối {block} để chứa các câu lệnh cho cấu trúc if thực hiện. Vậy câu hỏi đặt ra ở đây là khi chúng ta khởi tạo một biến (variable) trong {block} của cấu trúc if đó khi kết thúc cấu trúc if thì có thể sử dụng được không?

Câu trả lời là KHÔNG, vì như chúng ta đã biết, khi chúng ta khởi tạo một {block} đồng nghĩa chúng ta đang làm việc với bộ nhớ STACK, khi chúng ta khởi tạo một hàm main() và tạo các biến giá trị nằm ở trong đấy thì những biến (variable) đó đều được khởi tạo và nằm trong bộ nhớ STACK, khi chúng ta kết thúc một {block} thì những biến (variable), giá trị (value) trong khối lệnh đó cũng bị thu hồi. Ngoại trừ trường hợp chúng ta khai báo biến với [Global Scope] tức là [Biến toàn cục] thì chúng ta hoàn toàn có thể sử dụng biến (variable) đó ở bất kì đâu trong chương trình, ngoài {block} hay trong {block} khi kết thúc cũng không bị thu hồi. Vì các biến đó được lưu trữ trong vùng nhớ STATIC, vùng nhớ tĩnh và nằm yên ở đó đến khi chương trình kết thúc thì mới được giải phóng.

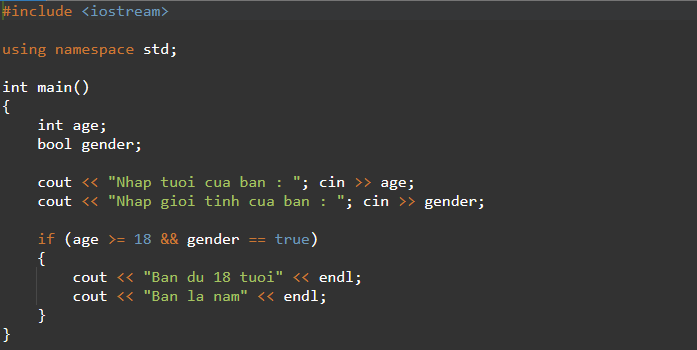
Kết lại, thì chúng ta nên lưu ý việc sử dụng biến trong {block} trong cấu trúc if. Nếu chúng ta muốn thay đổi một biến (variable) thì chúng ta nên khai báo một biến ngoài hàm main() để có thể sử dụng trong cấu trúc if và sau khi thoát khỏi if thì giá trị của biến đó sẽ được cập nhật.

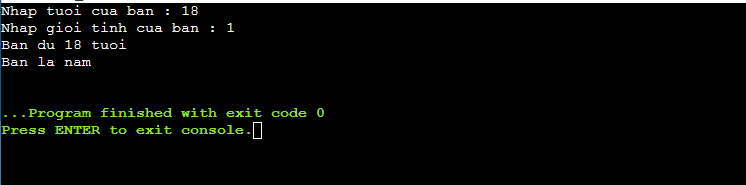
**Như chúng ta có thể thấy, việc sử dụng biến và các khối lệnh là quan trọng quyết định tính chất cũng như đúng sai của một chương trình, cấu trúc if cung cấp cho chúng ta những luồng điều khiển chương trình đi theo từng hướng, những giá trị trả về chỉ có True or False.**

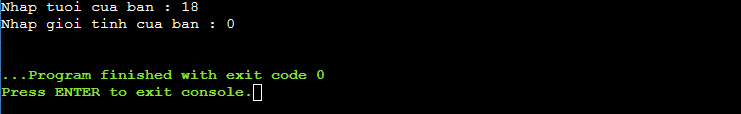
**Tóm lại, cấu trúc rẽ nhánh (if) trong C++ là một cách để thực hiện các hành động khác nhau dựa trên điều kiện được xác định trước đó.**

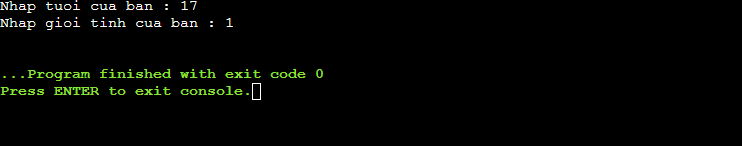
**Cấu trúc if else if được sử dụng để xét nhiều điều kiện cùng lúc, xử lý một chương trình điều kiện phức tạp mà 1 cấu trúc if else không thể xử lý.**

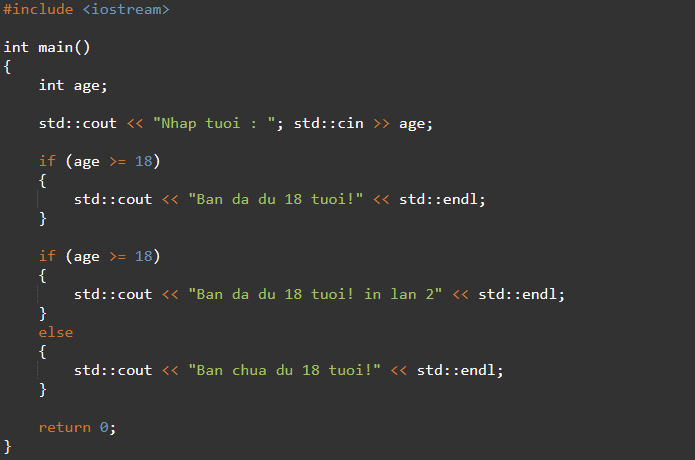
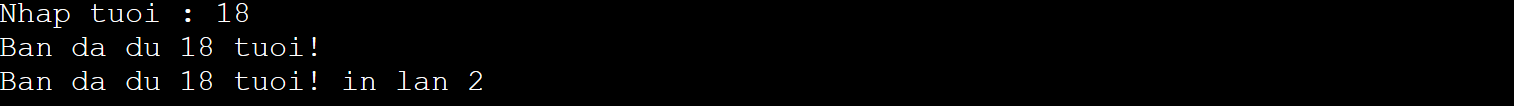
# **Các ví dụ về cấu trúc if trong C++**









**Tổng kết về cấu trúc if else : Cấu trúc rẽ nhánh if else if dùng để xử lý các luồng điều khiển phức tạp, có nhiều điều kiện cần xét và trả về nhiều giá trị khác nhau.**

**Cấu trúc if else if có thể sử dụng để kiểm tra số âm, dương, tính chẵn – lẻ, hay một chuỗi ký tự hoặc là mảng.**

**Một chương trình có nhiều luồng điều khiển phức tạp thì không thể không kể đến cấu trúc rẽ nhánh if else được. Tầm quan trọng của nó đã được nêu rõ như cái tên của nó.**

CẤU TRÚC ĐIỀU KHIỂN, CÂU LỆNH RẼ

NHÁNH CẤU TRÚC IF ELSE VÀ IF ELSE IF

# **CẤU TRÚC IF ELSE LÀ GÌ?**

* Cấu trúc rẽ nhánh trong C++ là một cách để thực hiện các hành động khác nhau dựa trên điều kiện được xác định trước đó. Cấu trúc rẽ nhánh có thể được thực hiện bằng cách sử dụng các câu lệnh điều kiện **if-else** hoặc **switch-case**.
* Cấu trúc if else có thể đưa ra kết quả khi mà biểu thức câu lệnh if sai hoặc điều kiện không thoả mãn câu lệnh **IF**. Khi đó ta sẽ sử dụng cấu trúc **if – else** (thường thì cấu trúc if sẽ hay đi với else để tạo thành một cấu trúc rẽ nhánh hoàn chỉnh). Nếu biểu thức if đúng, trả về giá trị True, thì khối lệnh của if sẽ được thực hiện. Ngược lại, nếu biểu thức **if** **sai**, thì ta sẽ trả về kết quả của false, khi mà biểu thức **if** **sai**.
* **Ví dụ :**

#include <iostream>

int main()

{

int age;

std::cout << "Nhap tuoi : "; std::cin >> age;

if (age >= 18)

{

std::cout << "Ban da du 18 tuoi!" << std::endl;

}

else

{

std::cout << "Ban chua du 18 tuoi!" << std::endl;

}

return 0;

}

* + Khi nhập vào số **tuổi >= 18** thì chương trình sẽ thực hiện khối lệnh if.
  + Ngược lại, khi người dùng nhập số **tuổi < 18 tức là 17** trở xuống thì chương trình sẽ thực hiện cấu trúc else.
* Cấu trúc rẽ nhánh **if else if** là cấu trúc rẽ nhánh có thể cho phép nhiều điều kiện trong một cấu trúc rẽ nhánh, trên 2 điều kiện, khi đó có thể sử dụng cấu trúc if, else if.
* **Ví dụ :**

#include <iostream>

int main()

{

int age;

std::cout << "Nhap tuoi : "; std::cin >> age;

if (age >= 18)

{

std::cout << "Ban da du 18 tuoi!" << std::endl;

}

else if (age > 15 && age < 18)

{

std::cout << "Ban du tuoi vi thanh nien!" << std::endl;

}

else

{

std::cout << "Ban chua du 18 tuoi!" << std::endl;

}

return 0;

}

* + Khi nhập số **tuổi >= 18** thì khối lệnh if được thực hiện, ngược lại khi nhập từ 15 < age < 18 thì khối lệnh else if được thực hiện, ngược lại dưới 16 tuổi thì khối lệnh else được thực hiện.
* ***Lưu ý*** *: Có thể sử dụng nhiều cấu trúc else if trong cùng một cấu trúc rẽ nhánh. Và if luôn nằm cuối cùng, else luôn nằm ở sau cùng.*

# **Cú pháp của cấu trúc if**

if (condition) {

**// Code sẽ được thực thi nếu điều kiện là true**

}

* Hầu hết các câu lệnh khác của C++ thực hiện theo tuần tự, khác với các câu lệnh đó, cấu trúc rẽ nhánh if thực hiện theo một biểu thức điều kiện, nếu **biểu thức** (**expression**) đó đúng thì chương trình sẽ thực hiện các dòng lệnh trong cặp ngoặc {**block**}. Ngược lại nếu điều kiện đó sai thì sẽ kết thúc câu lệnh if và thực hiện dòng tiếp theo hoặc dòng kế tiếp.
* Điều kiện của một cấu trúc rẽ nhánh if là một giá trị **LOGIC** (**bool**), chỉ có True or False. Các biểu thức Logic trong điều kiện của câu lệnh if có thể kể đến như : **==, <, <=, >, >=, !=**. Chúng ta có thể kết hợp giữa hai biểu thức (expression) bằng các toán tử, hay các **phép HỘI (&&) hay là phép TUYỂN (||) hoặc là phép phủ định (!)**.

# **Cú pháp của cấu trúc if – else**

if (condition) {

**// Code sẽ được thực thi nếu điều kiện là true**

}

else {

**// Code sẽ được thực hiện nếu điều kiện của if là false**

}

# **Cú pháp của cấu trúc if – else if – else**

if (condition) {

**// Code sẽ được thực thi nếu điều kiện là true**

}

else if (condition) {

**// Code sẽ được thực hiện nếu điều kiện của if là false**

}

else {

**// Code sẽ được thực hiện nếu điều kiện của else if là false**

}

* Các điều kiện của cấu trúc if else if phải luôn không trùng với nhau.
* Khi thực hiện thì chương trình sẽ chạy tuần tự từ trên xuống dưới, nếu có điều kiện nào đúng thì nó thực hiện khối lệnh của điều kiện đó sau đó thoát luôn ra khỏi cấu trúc.
* Cấu trúc if luôn được đặt trên cùng để kiểm tra biểu thức điều kiện đầu tiên, sau đó tiếp đến các điều kiện tiếp theo đến điều kiện (n). Cho tới khi kết thúc mới là else luôn được đặt ở sau cùng.

# **Khi nào nên sử dụng cấu trúc rẽ nhánh if và ứng dụng**

* Khi chúng ta đứng trước một bài toán, chúng ta có nhiều điều kiện cần kiểm tra, nếu điều kiện là True thì ta thực hiện một hành động gì đó, nếu là False thì ta thực hiện một hành động khác.
* **Ví dụ :** Nếu tôi đạt điểm kỳ thi trên 9 thì tôi đạt được điểm A+, ngược lại dưới 9 thì tôi chỉ đạt A hoặc B+.
* **Ví dụ :** Nếu hôm nay trời mưa, tôi sẽ không đi chơi, còn nếu là trời nắng, tôi sẽ đi chơi.
* Nhiều bài toán có hoặc nhiều hơn 2 điều kiện cần kiểm tra, như vậy cấu trúc rẽ nhánh if sinh ra để chúng ta kiểm tra một điều kiện gì đó đúng thì trả về kết quả tương ứng giống như một con **Robot** đơn giản, chúng ta đặt một câu hỏi gì cho nó sau đó nó tính toán và đưa ra câu trả lời đúng và gần sát với câu hỏi mà chúng ta đưa ra.
* Ngoài ra cấu trúc if còn có thể sử dụng trong vòng lặp để kiểm tra điều kiện gì đó trước khi lặp một giá trị hay là một biểu thức cụ thể.

# **Khi nào nên sử dụng cấu trúc rẽ nhánh if else if else và ứng dụng**

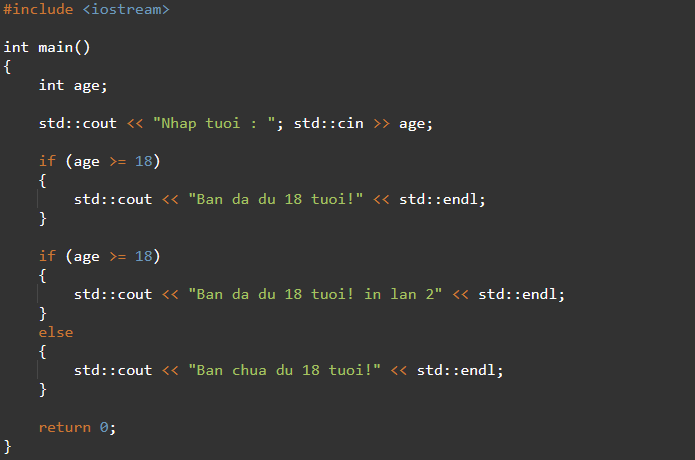
* Khi chúng ta cần kiểm tra nhiều điều kiện mà cấu trúc if không thể đáp ứng, khi đó ta sử dụng cấu trúc if else if để kiểm tra lần lượt các điều kiện khi có điều kiện đúng thì thực hiện khối lệnh của điều kiện đó và thoát khỏi cấu trúc rẽ nhánh.
* **Ví dụ :** Xét thang điểm từ **0 🡪 10**.
  + **Từ 8 trở lên là HSG.**
  + **Từ 6.5 đến dưới 8 là HSK.**
  + **Từ 5.0 đến dưới 6.5 là HSTB.**
  + **Dưới 5.0 là HSY.**
* Khi đó ta sử dụng cấu trúc **if else if** để xét lần lượt các điều kiện rồi đưa ra kết quả cuối cùng 🡺 Ví dụ khi nhập 6.4 thì sẽ thực hiện khối lệnh từ 5.0 đến dưới 6.5.
* Cấu trúc này thực hiện tuần tự từ trên xuống dưới, khi có điều kiện thoả mãn thì thực hiện và thoát cấu trúc, nếu không có điều kiện nào trả về **True** thì chương trình sẽ thực hiện nhánh **else**.

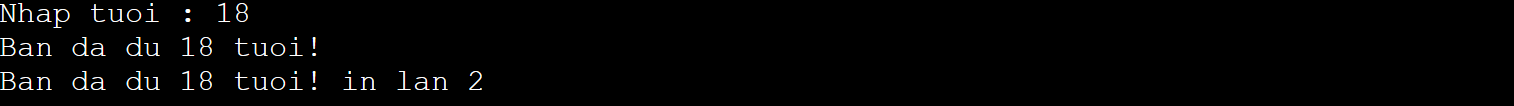
# **Những lưu ý trong cấu trúc if**

* Cấu trúc **if** kiểm tra biểu thức sau đó trả về một giá trị boolean (**true** hoặc **false**), chứ không phải là một giá trị nào khác.
* Tầm quan trọng của việc **đúng sai** của một biểu thức trong bài toán quyết định rất lớn tới luồng điều khiển mà chương trình thực hiện, nếu chúng ta đặt sai biểu thức thì cấu trúc chương trình sẽ cho ra kết quả khác, vì cấu trúc rẽ nhánh không thực hiện **tuần tự**, mà thực hiện theo các nhánh True or False. Trừ khi kết thúc câu lệnh if thì chương trình mới thực hiện các lệnh kế tiếp nếu còn các dòng lệnh.
* Nếu hành động trong ngoặc {**block**} chỉ có 1 câu lệnh thì chúng ta hoàn toàn có thể không cần sử dụng đến ngoặc {block} đó mà hoàn toàn có thể lược bỏ đi cũng như có thể ghi trên cùng 1 dòng.
* **Ví dụ :** if (age > 18) cout << “Ban da truong thanh!” << endl;
* Còn nếu nhiều hơn một dòng lệnh thì chúng ta phải bọc nó trong cặp ngoặc nhọn {**block**}, để chương trình hiểu đây là một khối lệnh của cấu trúc **if**.
* Khi chúng ta cần nối nhiều điều kiện với nhau, nhiều biểu thức lại với nhau để cùng so sánh vừa đưa ra kết quả cuối cùng chỉ có 1 giá trị (**True** / **False**) thì chúng ta có thể sử dụng các toán tử **Logic** : **Phép hội (&&), phép tuyển (||), phép phủ định (!)**.
* Nếu gặp một biểu thức so sánh có độ khó phức tạp hoặc nhiều biểu thức với nhau thì nên sử dụng cặp ngoặc () để tăng độ ưu tiên của các **biểu thức** (**expression**) hay là các **toán hạng** (**operand**).

# **Những lưu ý trong cấu trúc if else if**

* Cấu trúc if else if được sử dụng để kiểm tra các trường hợp đặc biệt hay các điều kiện phức tạp do đó các điều kiện khi chúng ta xét phải đúng khi sử dụng các toán tử Logic hay xét từ khoảng nào đến khoảng nào.
* Cấu trúc rẽ nhánh thực hiện tuần từ đầu trên xuống do đó nếu có điều kiện nào thoả mãn thì thực hiện khối lệnh của nhánh đó và thoát khỏi cấu trúc rẽ nhánh.
* **Nếu bạn sử dụng 2 điều kiện giống nhau :** 
  + **Ví dụ :** if (age >= 18) ; else if (age >= 18).
  + Thì chương trình sẽ thực hiện nhánh if sau đó thoát khỏi cấu trúc vì cấu trúc if else if thực hiện tuần tự từ trên xuống. Nên khi bạn để 2 điều kiện giống nhau thì sẽ không có thực hiện được cả 2 dòng lệnh rẽ nhánh.
* Khi chúng ta sử dụng 2 if, thì 2 khối lệnh if này thực hiện độc lập nhau, tách rời nhau hoàn toàn và không ảnh hưởng gì đến nhau. Khi thực hiện xong if đầu tiên thì chương trình tiếp tục xét đến if tiếp theo rồi thực hiện.
* **Ví dụ :**





* Khi đó chương trình sẽ thực hiện tuần tự từ trên xuống dưới, 2 lệnh if là hoàn toành tách biệt nhau, cho nên 2 if này được xem độc lập và đều được xét điều kiện khi chương trình chạy.

# **Câu hỏi đặt ra về vùng nhớ, khối lệnh**

Khi chúng ta khởi tạo một cấu trúc if thì sẽ có một khối {block} để chứa các câu lệnh cho cấu trúc if thực hiện. Vậy câu hỏi đặt ra ở đây là khi chúng ta khởi tạo một biến (variable) trong {block} của cấu trúc if đó khi kết thúc cấu trúc if thì có thể sử dụng được không?

Câu trả lời là KHÔNG, vì như chúng ta đã biết, khi chúng ta khởi tạo một {block} đồng nghĩa chúng ta đang làm việc với bộ nhớ STACK, khi chúng ta khởi tạo một hàm main() và tạo các biến giá trị nằm ở trong đấy thì những biến (variable) đó đều được khởi tạo và nằm trong bộ nhớ STACK, khi chúng ta kết thúc một {block} thì những biến (variable), giá trị (value) trong khối lệnh đó cũng bị thu hồi. Ngoại trừ trường hợp chúng ta khai báo biến với [Global Scope] tức là [Biến toàn cục] thì chúng ta hoàn toàn có thể sử dụng biến (variable) đó ở bất kì đâu trong chương trình, ngoài {block} hay trong {block} khi kết thúc cũng không bị thu hồi. Vì các biến đó được lưu trữ trong vùng nhớ STATIC, vùng nhớ tĩnh và nằm yên ở đó đến khi chương trình kết thúc thì mới được giải phóng.

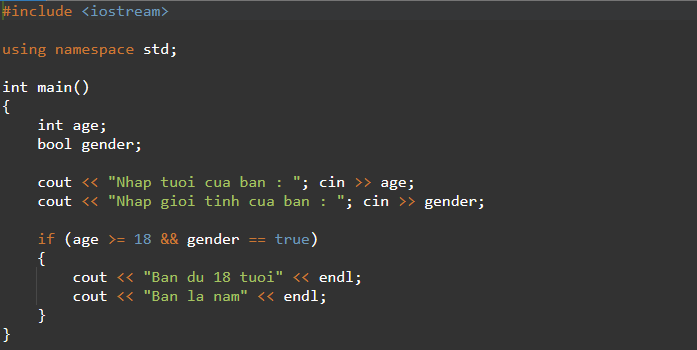
Kết lại, thì chúng ta nên lưu ý việc sử dụng biến trong {block} trong cấu trúc if. Nếu chúng ta muốn thay đổi một biến (variable) thì chúng ta nên khai báo một biến ngoài hàm main() để có thể sử dụng trong cấu trúc if và sau khi thoát khỏi if thì giá trị của biến đó sẽ được cập nhật.

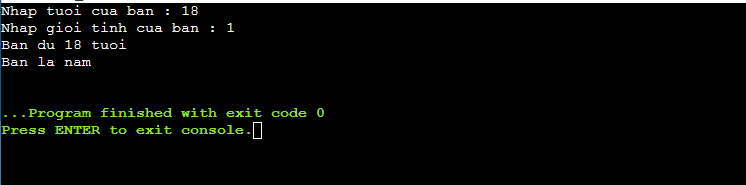
**Như chúng ta có thể thấy, việc sử dụng biến và các khối lệnh là quan trọng quyết định tính chất cũng như đúng sai của một chương trình, cấu trúc if cung cấp cho chúng ta những luồng điều khiển chương trình đi theo từng hướng, những giá trị trả về chỉ có True or False.**

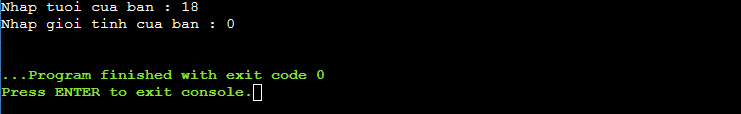
**Tóm lại, cấu trúc rẽ nhánh (if) trong C++ là một cách để thực hiện các hành động khác nhau dựa trên điều kiện được xác định trước đó.**

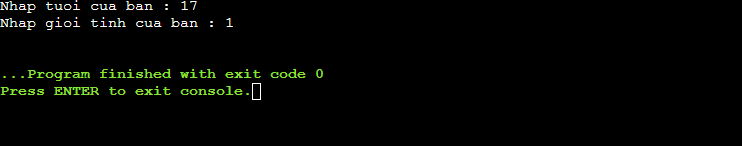
**Cấu trúc if else if được sử dụng để xét nhiều điều kiện cùng lúc, xử lý một chương trình điều kiện phức tạp mà 1 cấu trúc if else không thể xử lý.**

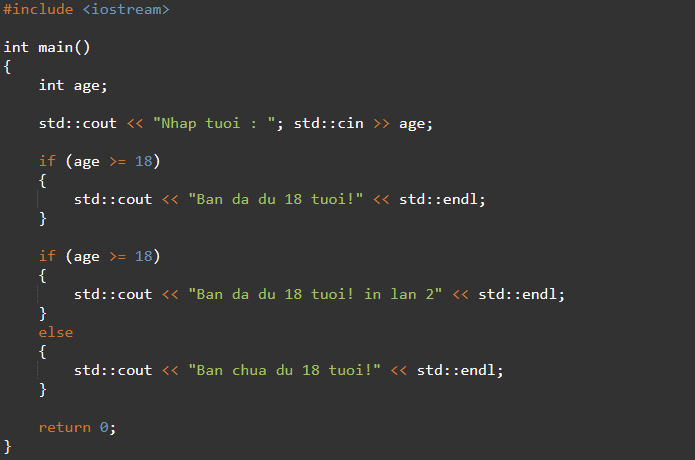
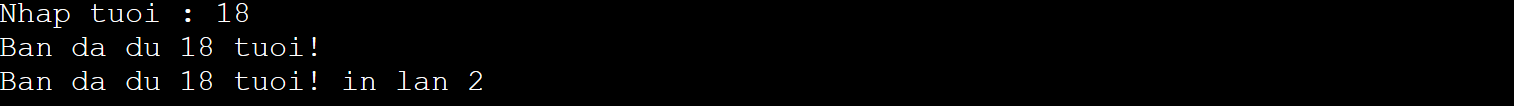
# **Các ví dụ về cấu trúc if trong C++**









**Tổng kết về cấu trúc if else : Cấu trúc rẽ nhánh if else if dùng để xử lý các luồng điều khiển phức tạp, có nhiều điều kiện cần xét và trả về nhiều giá trị khác nhau.**

**Cấu trúc if else if có thể sử dụng để kiểm tra số âm, dương, tính chẵn – lẻ, hay một chuỗi ký tự hoặc là mảng.**

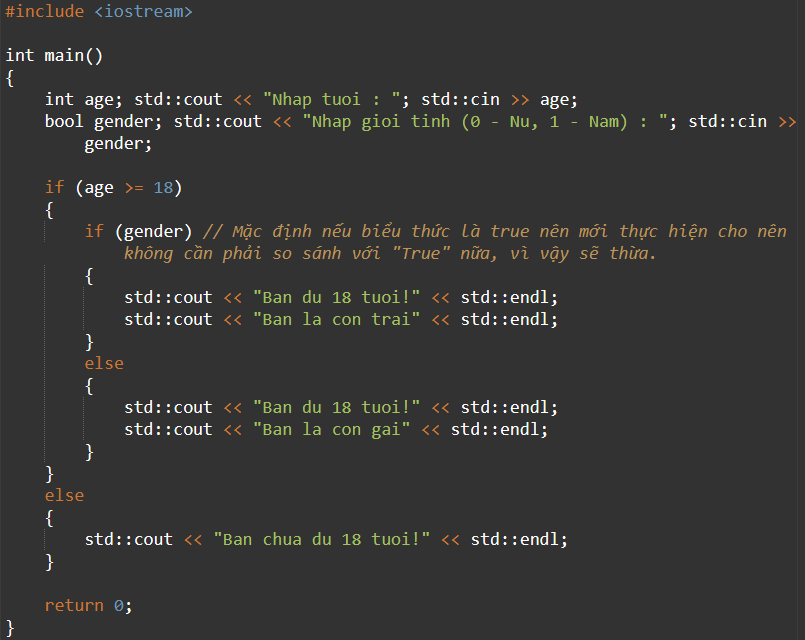
**Một chương trình có nhiều luồng điều khiển phức tạp thì không thể không kể đến cấu trúc rẽ nhánh if else được. Tầm quan trọng của nó đã được nêu rõ như cái tên của nó.**

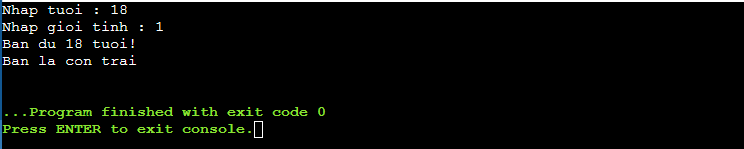
CẤU TRÚC RẼ NHÁNH – CẤU TRÚC IF ELSE LỒNG NHAU

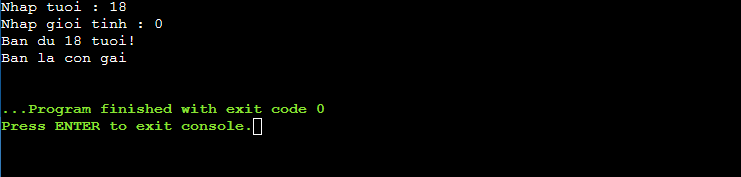
# **Cấu trúc if else lồng nhau là gì**

* + Cấu trúc rẽ nhánh trong C++ là một cách để thực hiện các hành động khác nhau dựa trên điều kiện được xác định trước đó. Cấu trúc rẽ nhánh có thể được thực hiện bằng cách sử dụng các câu lệnh điều kiện if-else hoặc switch-case.
  + Cấu trúc if else sử dụng để lý các điều kiện rẽ nhánh phức tạp, như là các điều kiện xét độ chẵn lẻ của một số hay số đó có phải số nguyên tố hay không. Cấu trúc **if else** sử dụng nhiều cho các chương trình rẽ nhánh có độ phức tạp. Thế nhưng một chương trình rẽ nhánh phức tạp hơn thì có một cấu trúc if else lồng nhau, cấu trúc if else lồng nhau thường dùng để xử lý các bài toán khi xét một điều kiện đúng sau đó vào trong biểu thức if trước đó xét thêm một biểu thức điều kiện nữa.
  + Cấu trúc **if else lồng nhau** nghe như vậy có vẻ phức tạp, nhưng thật sự không phức tạp chút nào đâu, sau đây sẽ là ví dụ thật dễ hiểu.

# **Ví dụ về cấu trúc rẽ nhánh if else lồng nhau**







* + Với ví dụ ở trên, ta có thể thấy một cấu trúc if else lồng nhau. Khi nhập tuổi trên hoặc bằng 18 thì cấu trúc sẽ chạy tiếp vào if thứ 2 để xét giới tính, nếu là nam thì thực hiện khối lệnh if 2, nếu là false thì thực hiện else và xuất ra kết quả.
  + Khi ta nhập tuổi < 18 thì chương trình sẽ chạy vào nhánh else ở if 1, xuất ra kết quả là “Ban chua du 18 tuoi!”.

# **Cú pháp của cấu trúc if else lồng nhau**

if (*condition*)

if (*condition*) **// Thực hiện khi if 1 true**

**// Thực hiện khi if 2 true**

else

**// thực hiện khi if 2 false**

else

**// Thực hiện khi if 1 false**

…

…

* + Cú pháp của cấu trúc **if else lồng nhau** khá đơn giản, đây là cú pháp, cấu trúc chung của một cấu trúc if else lồng nhau, trong thực thế, ta tuỳ thuộc vào độ khó hay độ phức tạp của một bài toán mà triển khai cấu trúc if else sao cho **tối ưu, hoàn hảo nhất.**

# **Những lưu ý của cấu trúc if else lồng nhau**

* + ***Tuần tự :*** Cấu trúc của chương trình sẽ chạy tuần tự từ trên xuống dưới, khi có một điều kiện thoả mãn thì cấu trúc if rẽ vào điều kiện đó và xét tiếp, nếu đúng thì tiếp tục rẽ nhánh vào cho tới khi kết thúc nhánh cấu trúc if else.
    - ***Lưu ý :*** Nếu có 2 điều kiện giống nhau, ví dụ như 2 if đầu tiên điều kiện giống nhau thì chương trình sẽ thực hiện nhánh trên, vì tính tuần tự, chương trình thực hiện từ trên xuống.
  + ***Dữ liệu trả về :*** Dữ liệu trả về của một biểu thức if else lúc nào cũng là dữ liệu kiểu bool (boolean) – chỉ có True – Fasle và không là điều kiện nào khác.
  + ***Phạm vi truy cập của biến :*** Khi ta sử dụng if thì có một cặp ngoặc {block} được tạo ra, khi ra tạo một biến trong cặp ngoặc trong một biểu thức điều kiện if thì sau khi thoát khỏi if thì không thể sử dụng lại biến vừa khai báo trong câu lệnh if đó nữa.
    - ***Lưu ý :*** Nếu muốn sử dụng biến đã khai báo trong Scope của if thì không còn cách nào khác ngoài việc khai báo ở bên ngoài.
      * ***Có 2 cách :***
        1. Khai báo biến ở trong hàm main, nằm trước những Scope đã tạo trước đó (cách này thường xuyên được sử dụng để khai báo và khởi tạo giá trị cho các biến mà sử dụng xuyên suốt chương trình).

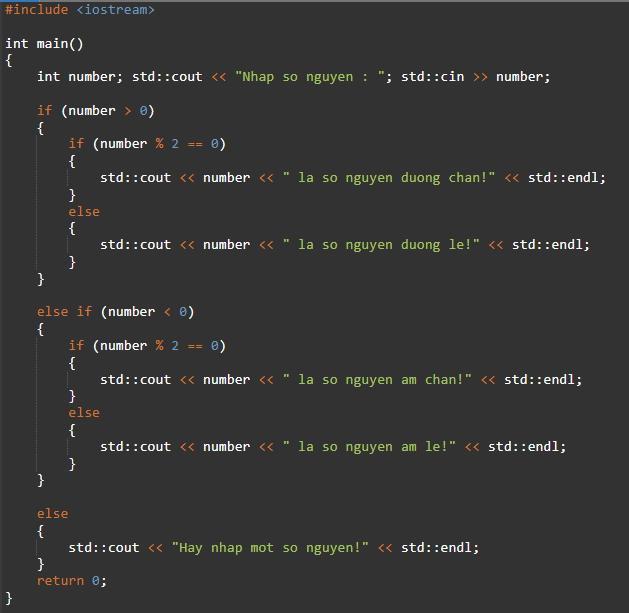
***Lưu ý :*** Việc khai báo biến trong hàm main() cũng chỉ là khai báo biến cục bộ (Local Scope / Local Variable) nếu chúng ta muốn sử dụng biến đó ở ngoài hàm main() là một hàm chúng ta tự định nghĩa thì việc này bất khả thi.

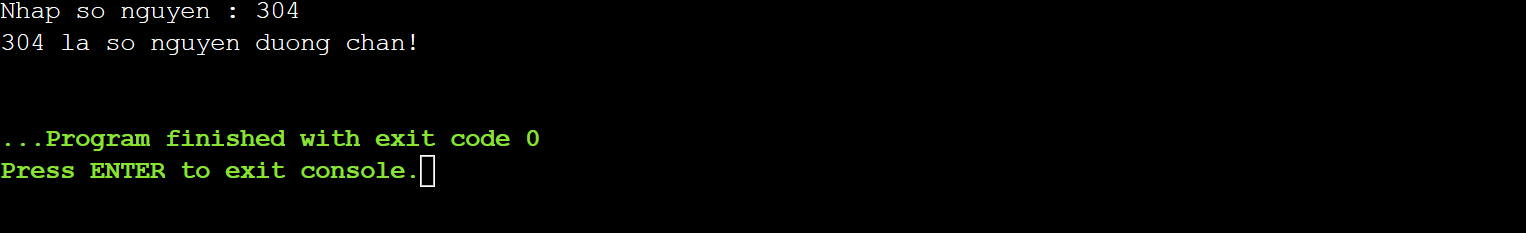
* + - * 1. Khai báo biến ở trên cùng của chương trình, dưới phần khai báo các thư viện (cách này không được sử dụng nhiều).

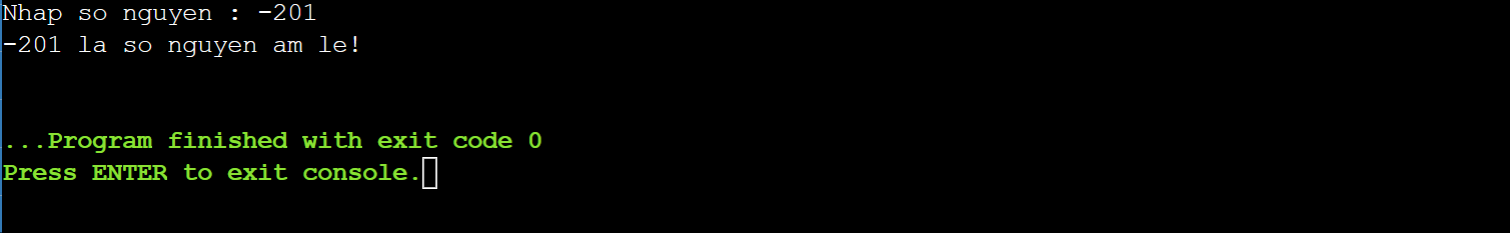
***Lưu ý :*** Khai báo biến ở ngoài hàm main() còn được gọi là biến toàn cục (Global Scope / Global Variable). Biến này được lưu trữ trong bộ nhớ Static (bộ nhớ tĩnh) được khởi tạo và tồn tại xuyên suốt chương trình hoạt động, cho tới khi ta kết thúc chương trình. Thế nhưng cách này không được hay sử dụng vì việc khai báo biến toàn cục chỉ hay sử dụng cho các giá trị bất biến (không đổi) xuyên suốt quá trình hoạt động, có thể đó là một hằng số hay một giá trị cố định mà bài toán cần thiết.

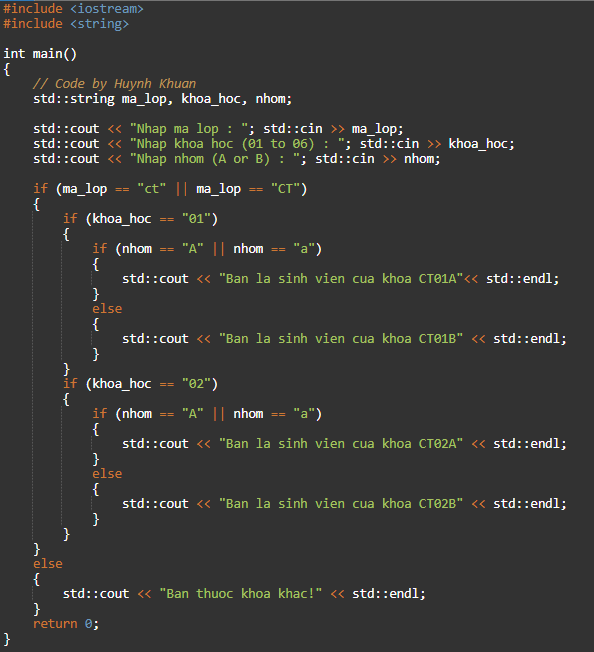
* + if else lồng nhau và if if là khác nhau : if else lồng nhau là một cấu trúc rẽ nhánh hoàn chỉnh, nhiều nhánh để rẽ tuỳ vào độ đúng sai của biểu thức Logic mà chúng ta khởi tạo và sử dụng cho cấu trúc rẽ nhánh. Thế nhưng if if là 2 cấu trúc rẽ nhánh độc lập nhau, không ảnh hưởng gì tới nhau.
    - ***Lưu ý :*** Khi chúng ta muốn xét một chương trình mà nhập số, sau đó kiểm tra số nào nhỏ hơn thì cập nhật số đó thì ta nên sử dụng if if (2 cấu trúc độc lập), vì giả sử chương trình có 3 số thế nhưng khi chúng ta nhập một số nhỏ thì chương trình rẽ nhánh mà không kiểm tra 2 số còn lại thì việc này vô lý.

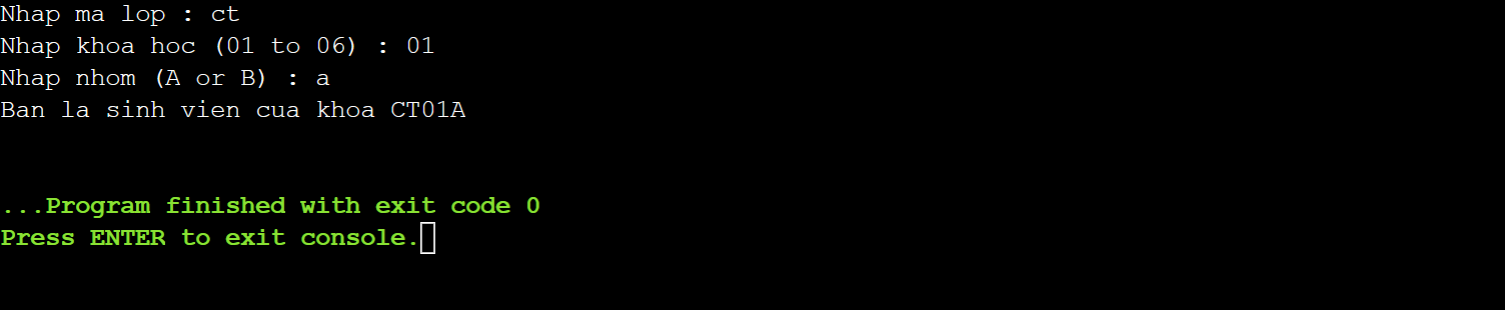
# **Bài toán vận dụng cấu trúc if else lồng nhau**

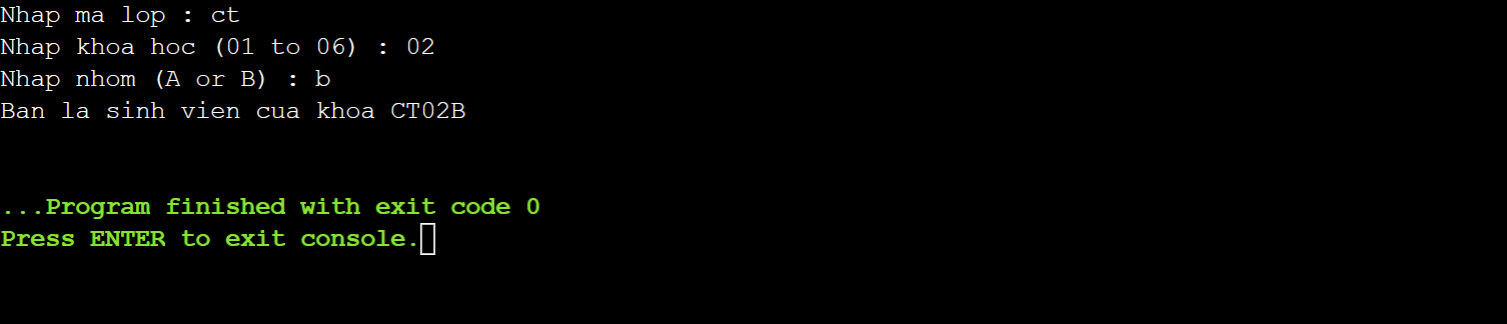


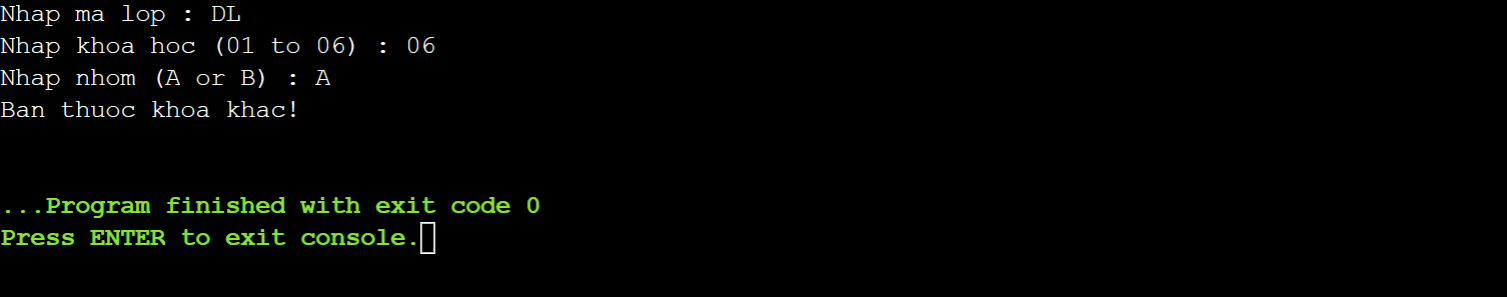












# **Sơ đồ về cấu trúc if lồng nhau**



TOÁN TỬ ĐIỀU KIỆN (CONDITIONAL OPERATOR)

# **Toán tử điều kiện (Conditional Operator) là gì?**

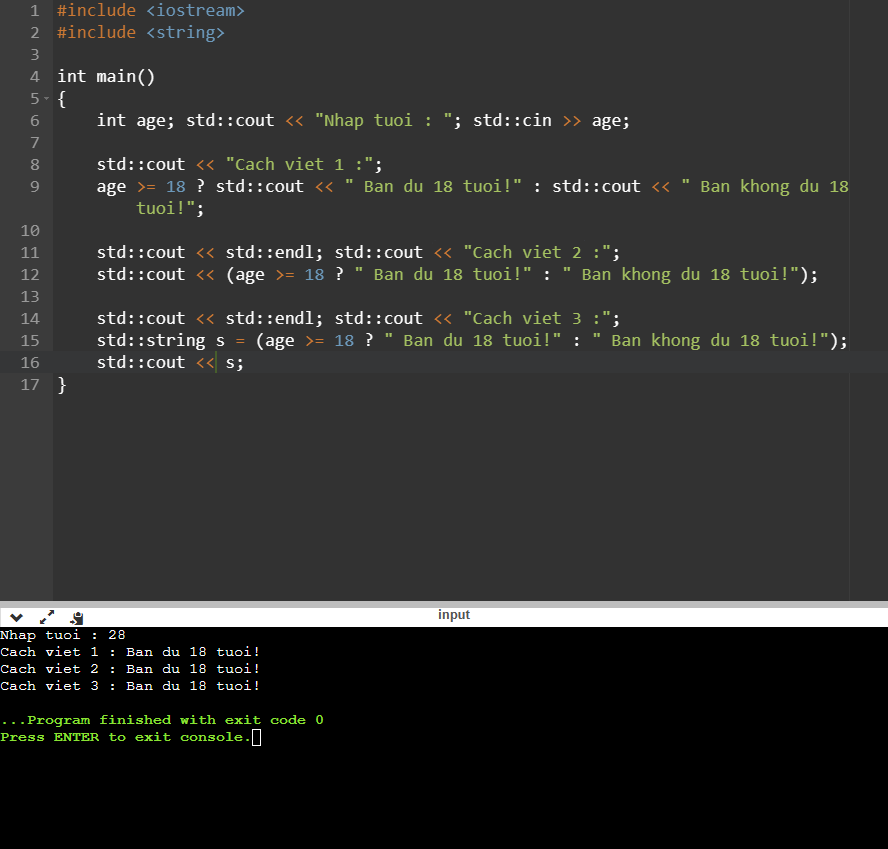
* + **Toán tử điều kiện (Conditional Operator)** là một biểu thức điều kiện hay còn gọi là **toán tử 3 ngôi (Ternary Operator)** dùng để xử lý các điều kiện đơn giản, ngắn ngọn mà không cần dùng tới **cấu trúc rẽ nhánh if else**.
  + Cách viết của **toán tử điều kiện (Conditional Operator)** tương đối đơn giản, viết trên cùng một dòng và có **2 toán tử** là **“?” và “:”**.
  + Khi ta muốn xử lý một điều kiện ngắn ngọn mà không cần dùng tới **cấu trúc rẽ nhánh if else** phức tạp thì ta có thể lựa chọn toán tử điều kiện hay **toán tử 3 ngôi**.
  + Toán tử điều kiện có thể xử lý các cặp lệnh hoặc có thể trả về một giá trị hay một biến, với kiểu trả về giá trị ta có thể gán giá trị đó cho một biến hoặc một cặp lệnh cụ thể.

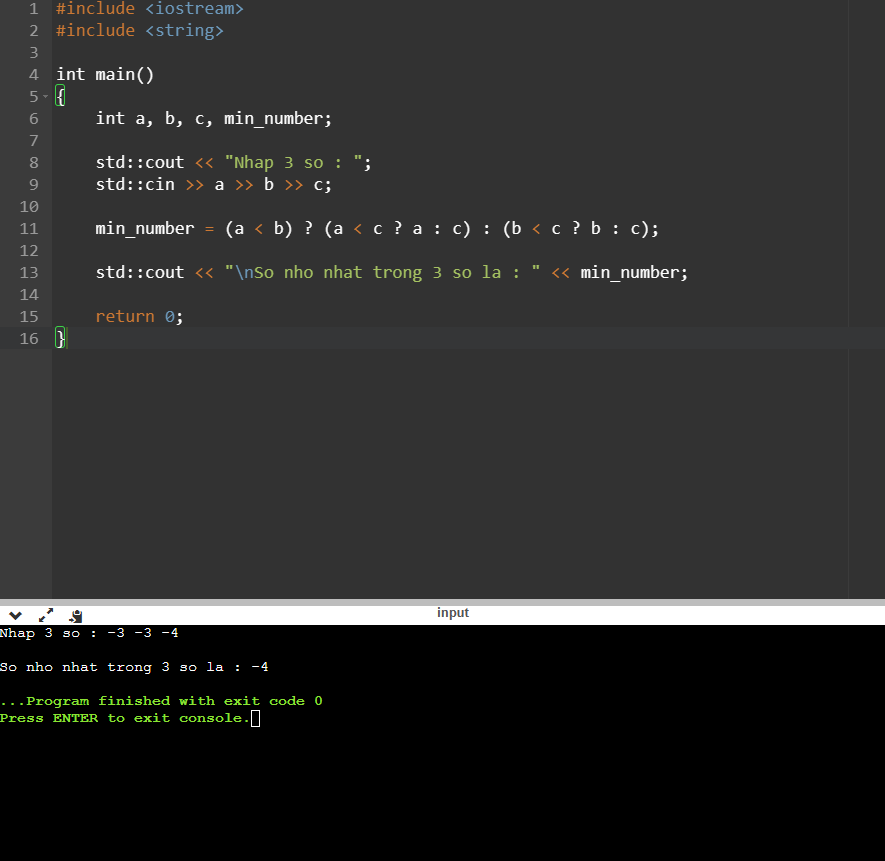
# **Cú pháp của toán tử điều kiện (Conditional Operator)**

condition ? expression1 : expression2

* + **Trong đó :**
    - **Condition** là một biểu thức Logic trả về giá trị **True** or **False**.
    - **expression1** là giá trị trả về nếu biểu thức logic là **True**.
    - **expression2** là giá trị trả về nếu biểu thức logic là **False**.
      * **expression1** và **expression2** có thể là các giá trị cụ thể (**fixed value**) hoặc các biến (**variable**) hoặc các cặp lệnh để thực hiện. Nếu là giá trị cụ thể ta có thể nối thêm một cặp lệnh trước toán tử điều kiện để thực hiện gán giá trị đó hoặc thực hiện giá trị đó.
  + Cú pháp của toán tử điều kiện (**Condition Operator**) thường rất đơn giản, gồm **2 toán tử** là **“?”** và **“:”** và **3 toán hạng** (**conditional**, **expression1**, **expression2**).
  + Các biểu thức trả về có thể là một giá trị hay biến và ta có thể thực hiện gán các giá trị đó cho các cặp lệnh cụ thể.

# **Ví dụ về toán tử điều kiện (Conditional Operator)**





# **Những lưu ý về toán tử điều kiện (Conditional Operator)**

* + **Sử dụng cho các trường hợp điều kiện ngắn ngọn**, trả về môt giá trị hoặc một biến ngắn ngọn.
  + **Sử dụng đơn giản** hơn cấu trúc rẽ nhánh if else, nhưng lại không sử lý được các trường hợp có nhánh cấu trúc phức tạp.
  + **Thích hợp** với các điều kiện kiểm tra ngắn ngọn và đơn giản.
  + Có thể **kiểm tra thêm điều kiện** trong 1 biểu thức trả về.
  + **Lưu ý** rằng toán tử **“?”** luôn lúc nào cũng **trước toán tử** **“:”**.

# **Những điều cần lưu ý trong toán tử điều kiện (Conditional Operator)**

* + **Cú pháp (syntax)** : là các toán tử hay còn được gọi là ký hiệu : “?” “:”.
  + **Số toán hạng (operand) tham gia** : 3 toán hạng tham gia tạo thành toán tử 3 ngôi gồm : biểu thức điều kiện, biểu thức trả về nếu là True, biểu thức trả về nếu là False.
  + **Độ ưu tiên (Precedence)** : Toán tử 3 ngôi luôn kiểm tra biểu thức điều kiện trước rồi trả về giá trị True or False.
  + **Thứ tự thực hiện (Associativity)** : Từ trái sang phải, nếu có thêm một cặp lệnh bên ngoài thì ta nên bọc toán tử điều kiện trong cặp ngoặc tròn “()” để đảm bảo độ ưu tiên cũng như là thứ tự thực hiện.
  + **Giá trị trả về (Return value)** : Các giá trị trả về của toán tử điều kiện có thể là giá trị (value) hoặc các biến (variable) hoặc là lệnh để biểu thức đó thực hiện.

**Tóm lại, toán tử điều kiện (Conditional Operator) hay còn được gọi là toán tử 3 ngôi (Ternary Conditional Operator) thường được sử dụng cho các trường hợp ngắn ngọn, không quá dài, và tăng tính thẩm mỹ cũng như chuyên nghiệp cho chương trình nếu vận dụng tốt.**

CẤU TRÚC SWITCH - CASE

# Cấu trúc switch case là gì

* + Cấu trúc switch case trong C++ được sử dụng để kiểm tra một biến hoặc biểu thức có giá trị tương ứng với một trong các giá trị cụ thể được liệt kê trong các trường hợp (case) được định nghĩa. Nó có thể được sử dụng để thay thế cho một chuỗi lệnh if else phức tạp và giúp mã chương trình dễ đọc và dễ bảo trì hơn.
  + Cấu trúc switch case dùng để kiểm tra các điều kiện có dạng so sánh bằng với điều kiện mà ta khởi tạo, giống nhau thay vì viết if (age == 18) thì ta có thể thay thế bằng cấu trúc switch case để nhìn ngắn ngọn và dễ đọc hơn.

# Cú pháp của cấu trúc switch case

**switch** (**variable or expression**)

{

**case** **value1**:

**// Các lệnh sẽ được thực hiện nếu biến hoặc biểu thức có giá trị bằng giá\_trị\_1**

**break**;

**case** **value2**:

**// Các lệnh sẽ được thực hiện nếu biến hoặc biểu thức có giá trị bằng giá\_trị\_2**

**break**;

**default**:

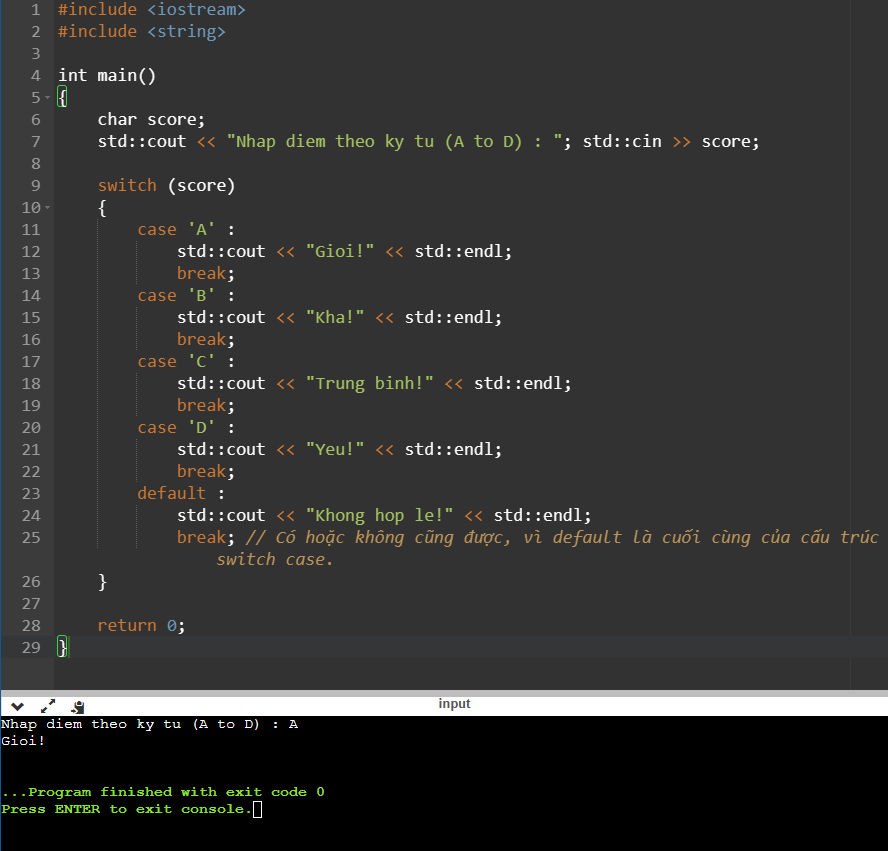
**// các lệnh sẽ được thực hiện nếu không có trường hợp nào khớp với giá trị của biến hoặc biểu thức**

**break**;

}

* + Trong đó:
    - biến hoặc biểu thức là biến hoặc biểu thức cần được kiểm tra.
    - giá\_trị\_1, giá\_trị\_2, ... là các giá trị cần so sánh với biến hoặc biểu thức.
    - Mỗi case là một trường hợp có thể xảy ra, tương ứng với một giá trị cụ thể.
    - default là một trường hợp mặc định, sẽ được thực hiện nếu không có trường hợp nào khớp với giá trị của biến hoặc biểu thức.
    - break được sử dụng để thoát khỏi switch case sau khi thực hiện các lệnh trong trường hợp tương ứng.

# Các ví dụ về cấu trúc switch case



# Những lưu ý về cấu trúc switch case

* + Cấu trúc switch case sử dụng cho các trường hợp so sánh với biểu thức hoặc giá trị hoặc là các biến một cách nhanh chóng thay vì sử dụng cấu trúc rẽ nhánh if else và phải đặt điều kiện là (variable == value).
  + Cấu trúc switch case phải có break; nằm ở cuối 1 case vì không có break thì cấu trúc switch case sẽ chạy cho tới khi gặp được case nào có break thì mới dừng lại, các giá trị vẫn sẽ được thực hiện cho tới khi break. Và nếu không có break thì cấu trúc switch case sẽ chạy cho tới khi kết thúc cấu trúc switch case thì thoát ra.
  + Cấu trúc switch case và trong 1 case có thể thực hiện nhiều cặp lệnh và cũng có thể lồng thêm các cấu trúc switch case trong 1 case, gọi là switch case lồng nhau.
  + Khi khởi tạo một biến trong 1 case nên lưu ý rằng nên tạo một scope để chứa biến đó, nếu không sẽ bị lỗi.

# Một số bài tập sử dụng cấu trúc switch case

* + Switch case lồng nhau
  + Link : [BmT4Cu - Online C++0x Compiler & Debugging Tool - Ideone.com](https://www.ideone.com/BmT4Cu)

**Tóm lại, cấu trúc switch case là một cấu trúc so sánh bằng với điều kiện hoặc một biểu thức mà không cần sử dụng cấu trúc rẽ nhánh if else. Sử lý nhanh chóng và có thể lồng thêm cấu trúc switch case và tiếp tục so sánh với biểu thức điều kiện của switch case 2.**

CẤU TRÚC VÒNG LẶP - CẤU TRÚC FOR

# **VÒNG LẶP FOR LÀ GÌ**

* + Thực tế một chút, khi bạn muốn thực hiện một hành động mà lặp đi lặp lại nhiều lần, có thể đếm được hoặc có những hành động không đếm được. Vậy thì chúng ta lặp đi lặp lại hành động đó được gọi là “Vòng lặp”.
  + Vòng lặp for trong C++ là một cấu trúc lặp được sử dụng để thực hiện một tập hợp các câu lệnh nhiều lần cho một khoảng giá trị được xác định trước đó. Vòng lặp for được sử dụng khi số lần lặp cần được biết trước và có thể được xác định bởi một biểu thức điều kiện.

# **CÚ PHÁP CỦA VÒNG LẶP FOR**

for (**khởi tạo**; **điều kiện**; **bước nhảy**) {

***// Các câu lệnh được lặp lại trong vòng lặp***

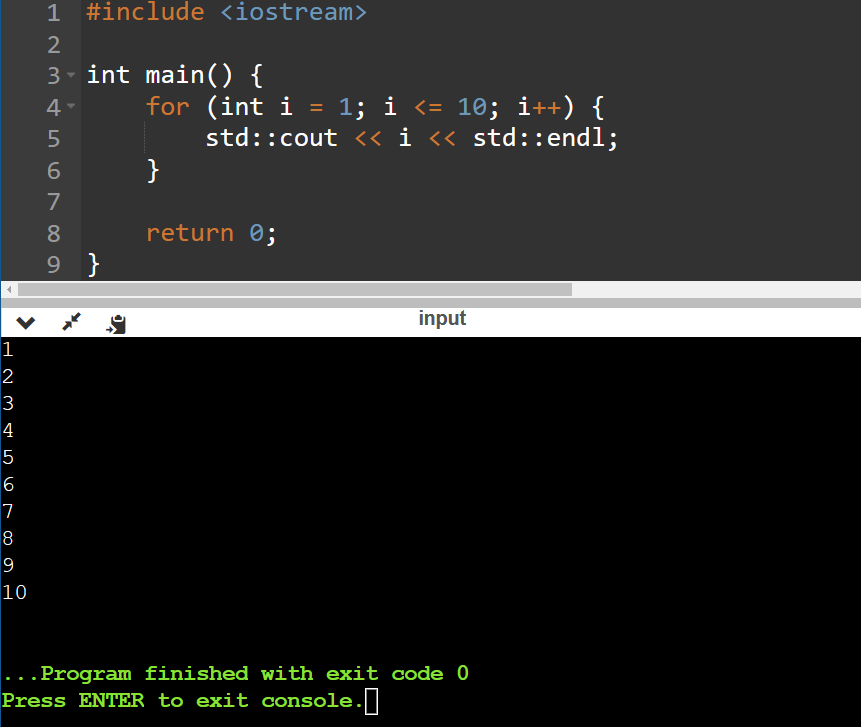
}

**Trong đó:**

* + **Khởi tạo :** là một biểu thức được sử dụng để khởi tạo giá trị ban đầu cho biến lặp.
  + **Điều kiện :** là một biểu thức điều kiện được sử dụng để kiểm tra xem liệu vòng lặp cần được tiếp tục thực hiện hay không.
  + **Bước nhảy :** là một biểu thức được sử dụng để thay đổi giá trị của biến lặp trong mỗi lần lặp lại.

Khi vòng lặp được thực hiện, biểu thức khởi tạo được tính toán một lần duy nhất, sau đó biểu thức điều kiện được kiểm tra. Nếu biểu thức điều kiện là đúng, các câu lệnh trong thân vòng lặp được thực hiện và sau đó biểu thức bước nhảy được tính toán để thay đổi giá trị của biến lặp. Quá trình này được lặp lại cho đến khi biểu thức điều kiện trả về giá trị sai.

# **Ví dụ về vòng lặp For**



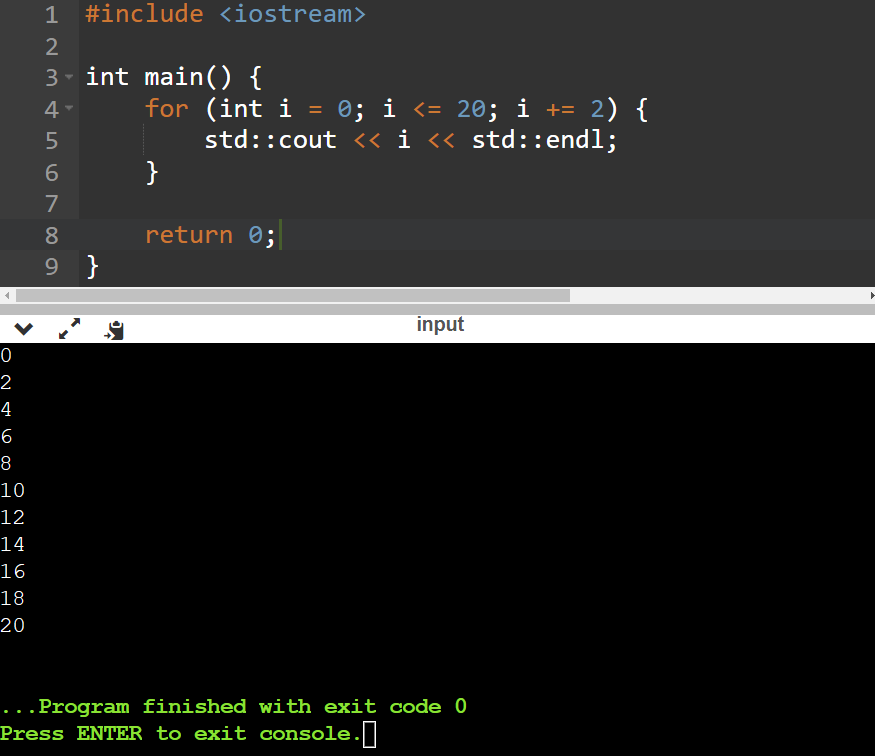
* + **Phân tích ví dụ trên :**
    - Khởi tạo vòng lặp for với biến khởi tạo i = 1.
    - Điều kiện để tiếp tục vòng lặp là i <= 10 🡺 i chạy từ 1 tới 10.
    - Bước nhảy là mỗi lần lặp như vậy là tăng i lên 1 đơn vị với “i++”.
  + **Cụ thể quá trình hoạt động như sau :**
    - Đầu tiên vòng lặp for sẽ kiểm tra i khởi tạo = 1 có <= 10 hay không, nếu có thì lặp lần thứ 1.
    - Sau đó sẽ chạy lên tăng 1 đơn vị với bước nhảy i++ và chạy qua kiểm tra điều kiện vòng lặp 🡺 i = 2 <= 10 🡺 tiếp tục vòng lặp.
    - Vòng lặp sẽ tiếp tục chạy theo quy trình trên và lặp tới khi 10. Lúc này sẽ kiểm tra điều kiện i <= 10 🡺 10 <= 10 🡺 Lặp lần thứ 10.
    - Tiếp theo tăng i lên 1 đơn vị 🡺 i = 11 🡺 i = 11 <= 10 (False) 🡺 Kết thúc vòng lặp, không tiếp tục lặp và thoát khỏi vòng lặp.
    - **Lưu ý :** Khi thoát khỏi vòng lặp (tức là vòng lặp dừng lại) thì lúc này i được khởi tạo ban đầu là i = 1 và khi dừng vòng lặp thì i = 11 chứ không phải 10 nhé. Vì khi tăng i lên 11 thì vòng lặp dừng nên biến i ở đây là 11.

# **Những lưu ý về vòng lặp For**

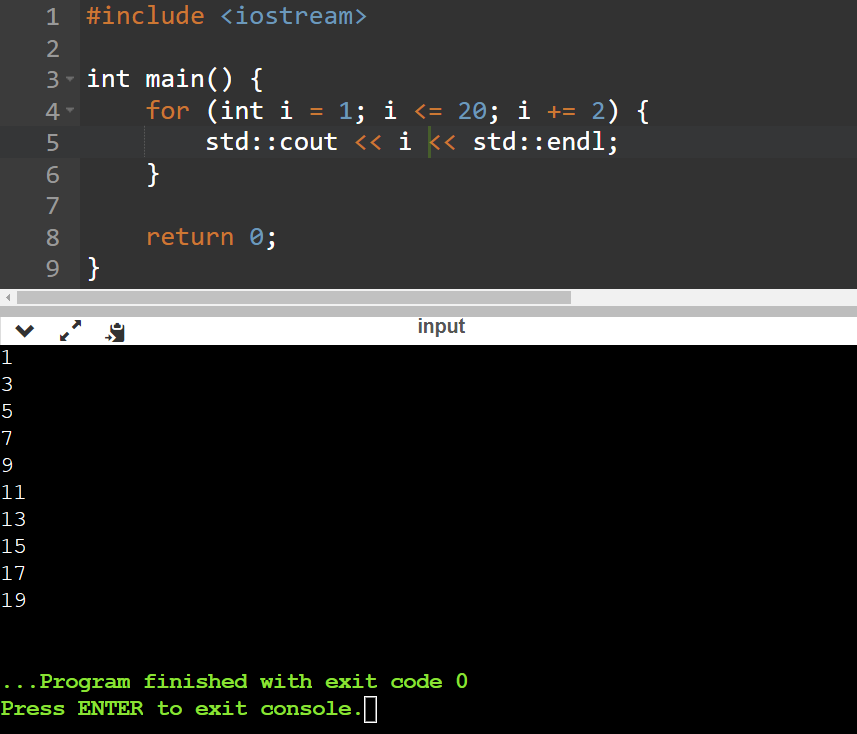
* + Vòng lặp có thể khuyết các giá trị khởi tạo; điều kiện vòng lặp; và bước nhảy hay không?
    - **Câu trả lời là CÓ.**
      1. **Khuyết giá trị khởi tạo** : Nếu không cung cấp giá trị khởi tạo trong vòng lặp for, bạn có thể khởi tạo giá trị ban đầu cho biến lặp ở bất kỳ đâu trong chương trình trước khi bắt đầu vòng lặp.
      2. **Khuyết điều kiện vòng lặp** : Nếu không có điều kiện vòng lặp thì điều kiện ở đây sẽ luôn luôn là TRUE thì vòng lặp sẽ được lặp trong vô hạn và không bao giờ dừng lại đến khi tràn RAM hoặc hết bộ nhớ.
      3. **Khuyết bước nhảy** : Nếu không có bước nhảy thì chương trình sẽ lặp đi lặp lại một giá trị mà chúng ta khởi tạo ban đầu. Ví dụ : for (int i = 1; i <= 10; ) 🡺 Thì chương trình sẽ chạy vô hạn giá trị i = 1 và lặp vô hạn tới khi tràn bộ nhớ thì kết thúc. Vì không có giá trị bước nhảy để cập nhật vòng lặp nên vòng lặp sẽ lặp vô hạn và giữ nguyên giá trị ban đầu.
         * Thế nhưng ta có thể di chuyển BƯỚC NHẢY đến cuối vòng lặp, vì mỗi khi thực hiện tất cả trong vòng lặp xong thì sẽ cập nhật giá trị khởi tạo ban đầu. Cho nên ta có thể di chuyển BƯỚC NHẢY (i++) xuống cuối vòng lặp vẫn được nhá.
  + **Biến và phạm vi của biến** : Các biến i (i viết tắt của **index**) khi khởi tạo trong vòng lặp thì chỉ có giá trị trong phạm vi vòng lặp đó (Variable Scope) được quy định trong một khối {block} khi thoát khỏi vòng lặp mà ta gọi biến đó thì sẽ bị lỗi (Undeclared Variable). Tương tự các biến (Variable) khi được khởi tạo trong Scope của vòng lặp For thì chỉ có giá trị trong Scope đó, khi thoát khỏi thì không sử dụng được.
  + Ta có thể **lồng nhiều vòng lặp for** với nhau để tạo ra một vòng lặp phức tạp để xử lý các bài toán có độ khó cao hơn.

# **Các bài toán về vòng lặp For**

* + **Chương trình in ra các số chẵn từ 0 🡪 20.**



* + **Chương trình in ra số lẻ từ 1 🡪 20.**



# **Các bài tập nâng cao về For :**

* + **Bài in ra bảng số từ 1 tới 100.**
* Link : [jbzIRY - Online C++ Compiler & Debugging Tool - Ideone.com](https://ideone.com/jbzIRY)
  + **Bài in ra bảng nhân từ 0 tới 10.**
* Link : [9PCu8b - Online C++ Compiler & Debugging Tool - Ideone.com](https://ideone.com/9PCu8b)
  + **Bài in bảng cửu chương từ 1 tới 10 theo dạng bảng**
* Link : [FSSfBi - Online C++ Compiler & Debugging Tool - Ideone.com](https://ideone.com/FSSfBi)
  + **Bài in ra dãy số chẵn và dãy số lẻ từ 1 tới n**
* Link : [ypG5e2 - Online C++ Compiler & Debugging Tool - Ideone.com](https://ideone.com/ypG5e2)

CẤU TRÚC VÒNG LẶP – CẤU TRÚC LẶP WHILE

# **Vòng lặp While là gì?**

* + Vòng lặp while trong C++ là một cấu trúc điều khiển lặp được sử dụng để lặp lại một khối mã cho đến khi một điều kiện được thỏa mãn.
  + Vòng lặp **While** sử dụng lặp đi lặp lại một hành động hay một khối lệnh cụ thể và với điều kiện (**condition**) luôn bằng **TRUE** thì vòng lặp sẽ được chạy mãi cho tới khi điều kiện bằng **FALSE** hoặc ngoài ra còn một kỹ thuật để thoát khỏi vòng lặp while đó chính là **BREAK**;
  + **Vậy vòng lặp while khác với vòng lặp for ở điểm nào?**
    - Điểm khác nhau giữa hai vòng lặp thì không có nhiều điểm khác nhau lắm, khi cả 2 vòng lặp đều sử dụng để lặp đi lặp lại một khối lệnh hay hành động cụ thể.
    - Điểm khác nhau giữa hai vòng lặp chính là điều kiện triển khai giữa hai vòng lặp.
      * **Vòng lặp for** : Cần 3 điều kiện trong vòng lặp for chính là : giá trị khởi tạo, điều kiện, bước nhảy. Cả 3 điều kiện này cùng nhau tạo nên một vòng lặp for hoàn hảo, nhưng 3 điều kiện này có thể khuyết (tức là không có cũng được). Thế nhưng hậu quả khuyết 1 trong 3 hoặc khuyết tất cả các điều kiện của vòng lặp for có thể gây ra các lỗi không mong muốn như lặp vô hạn, lặp vô hạn một giá trị hay một lệnh cụ thể.
      * **Vòng lặp while** : Chỉ cần 1 điều kiện cho vòng lặp while chính là (**condition**). Điều kiện của vòng lặp While sẽ dưới dạng bool (**boolean**) chỉ trả về giá trị **TRUE** or **FALSE**. Nếu là giá trị **TRUE** thì vòng lặp sẽ được tiếp tục lặp, FALSE thì vòng lặp sẽ kết thúc. Ta có thể thấy rằng vòng lặp While đơn giản hơn vòng lặp for khá nhiều.
    - Điểm khác nhau tiếp theo là nằm ở cách sử dụng cũng như mục đích sử dụng 2 vòng lặp này.
      * **Vòng lặp for** : Vòng lặp for sử dụng khi ta xác định được điểm đầu (start) và điểm cuối (end). Ví dụ như ta lặp một số từ 1 🡪 n thì ta xác định được điểm đầu là 1 và điểm cuối là n (với n được nhập từ bàn phím). Vòng lặp for được sử dụng khi ta biết được số lần lặp, có thể nói là số lần lặp hữu hạn hoặc số lần lặp xác định.
      * **Vòng lặp while** : Vòng lặp while sử dụng khi ta muốn lặp đi lặp lại một khối lệnh nào đó vô hạn hoặc không xác định số lần lặp là bao nhiêu. Thì khi đó ta sử dụng vòng lặp while. Ví dụ như ta tạo một chương trình cho người dùng nhập số từ bàn phím với số lần nhập không xác định và nhập cho tới khi người dùng nhập vào số 0 thì dừng.

# **Cú pháp của vòng lặp while**

**while (condition)**

**{**

**// Khối mã cần được lặp lại**

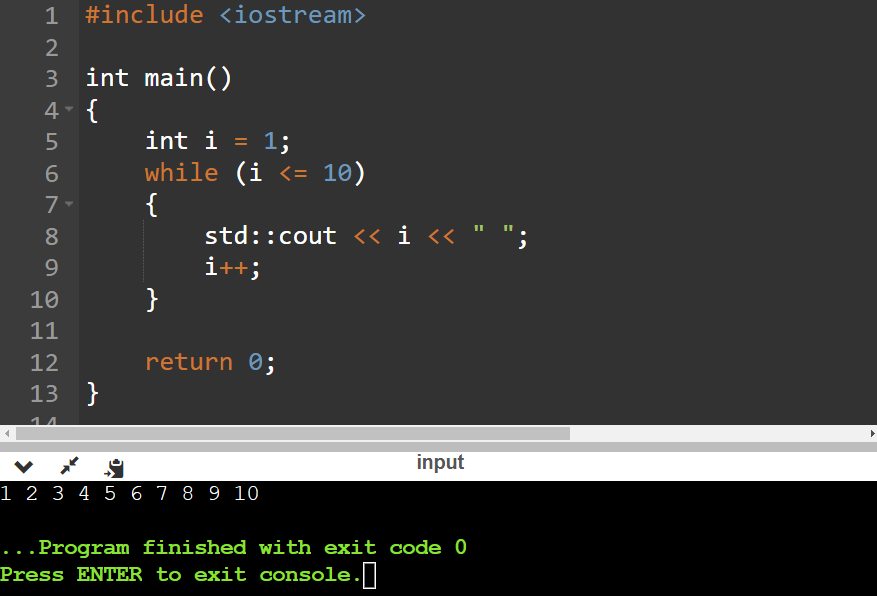
**// Cập nhật giá trị lặp (i++) để điều kiện FALSE để kết thúc vòng lặp**

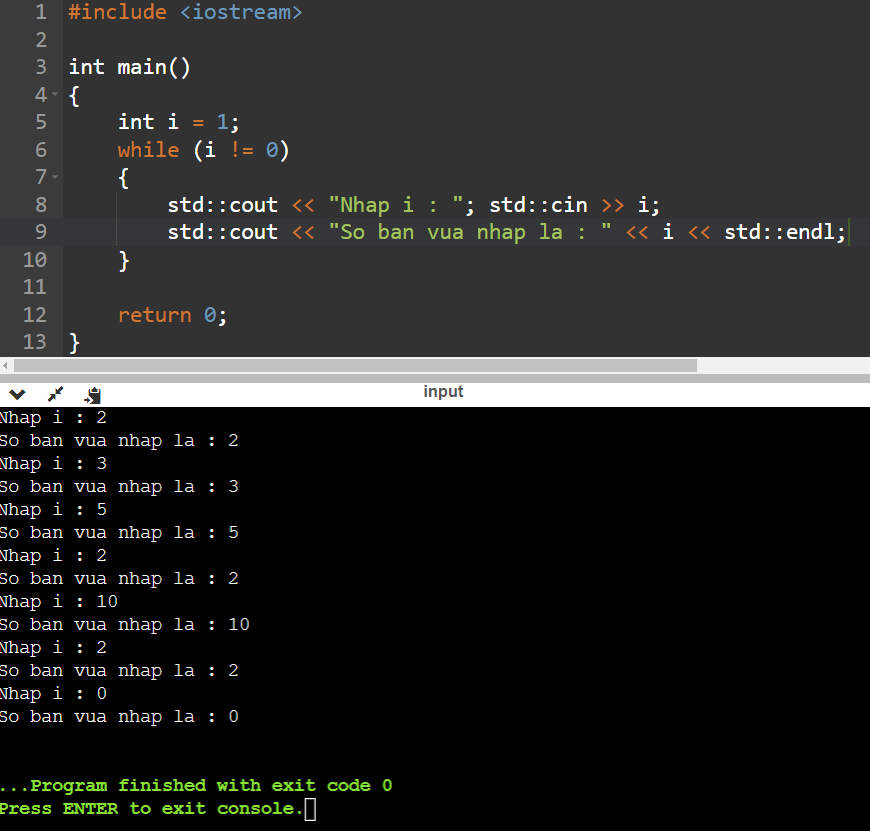
**// Hoặc sử dụng break;**

**}**

* + Cú pháp của vòng lặp while khá đơn giản, chỉ cần một điều kiện để **DUY** **TRÌ** vòng lặp while liên tục chạy và cho tới khi điều kiện **DUY TRÌ** đó **FALSE** thì vòng lặp sẽ ngay lập tức kết thúc.
  + Điều ta cần lưu ý ở đây chính là vậy nếu vòng lặp while vô hạn thì có cách nào dừng không?
    - **Câu trả lời là CÓ.**
      * **Ví dụ : i = 1; while (i <= 5) // giá trị.**
      * Thì lúc này vòng lặp while sẽ lặp vô hạn giá trị hoặc khối mã đó và giá trị khởi tạo i ở đây lúc nào cũng bằng 1 và while sẽ xuất ra màn hình i = 1 cho tới khi tràn RAM thì dừng chương trình.
      * Nếu muốn thoát khỏi vòng lặp while thì ta chỉ cần thêm bước nhảy vào cuối vòng lặp (tức là sau các khối mã) 🡺 i++.
      * Hoặc ta có thể sử dụng câu lệnh **break**; để kết thúc vòng lặp while và thoát ra ngoài.
    - Nếu không có biến cập nhật giá trị (**bước nhảy**) và break thì sao?
      * Thì vòng lặp sẽ lặp liên tục mà không kết thúc, không có điểm dừng chương trình, vòng lặp sẽ chạy cho tới khi tràn bộ nhớ thì thôi.

# **Ví dụ về vòng lặp while**





# **Những lưu ý về vòng lặp while**

* + Vòng lặp while dùng để lặp đi lặp đi khối mã với số lần không xác định hoặc lặp **vô hạn**.
  + Vòng lặp **while** được sử dụng để yêu cần người dùng nhập một chuỗi ký tự hoặc một số cho tới khi đúng thì mới dừng lại.
  + Các biến trong phạm vi **while** cũng giống như for, khi biến được khởi tạo trong vòng lặp thì khi sử dụng bên ngoài phạm vi (**Scope**) thì sẽ bị lỗi khai báo.
  + Nếu không muốn lặp vô hạn và hay bị lỗi thì chúng ta nên để một biến cập nhật giá trị ở cuối vòng lặp while chẳng hạn như i++…
  + Vòng lặp **while** lặp cho tới khi điều kiện sai cho nên việc sử dụng vòng lặp while cũng phải thật cẩn thận.

# **Các bài toán sử dụng vòng lặp while.**

* + Chương trình in ra dãy số Fibonacci
    - **Link** : [hD0r8i - Online C++ Compiler & Debugging Tool - Ideone.com](https://ideone.com/hD0r8i)
  + Chương trình nhập số và tính tổng số đã nhập cho tới khi nhập số âm thì dừng chương trình
    - **Link** : [Ti4abL - Online C++ Compiler & Debugging Tool - Ideone.com](https://ideone.com/Ti4abL)

CẤU TRÚC VÒNG LẶP – CẤU TRÚC LẶP LỒNG RẼ NHÁNH

# **Cấu trúc lặp lồng rẽ nhánh là gì**

* + Khi ta có một điều kiện cần xét khi lặp một dãy số thì cấu trúc rẽ nhánh lồng với cấu trúc lặp sẽ được áp dụng. Gọi là cấu trúc lặp lồng cấu trúc rẽ nhánh.
  + Ví dụ ta có dãy số từ 0 🡪 20, ta muốn in ra các số chẵn, và các số lẻ thì ta có thể sử dụng cấu trúc lặp lồng rẽ nhánh.
  + Tương tư như vậy, cấu trúc rẽ nhánh có thể lồng trong vòng lặp for, vòng lặp while và do while.

# **Cú pháp của cấu trúc lặp lồng rẽ nhánh**

**for (giá trị khởi tạo; điều kiện; bước nhảy)**

**{**

**if (condition)**

**{**

**// True thì thực hiện khối lệnh if này**

**}**

**else**

**{**

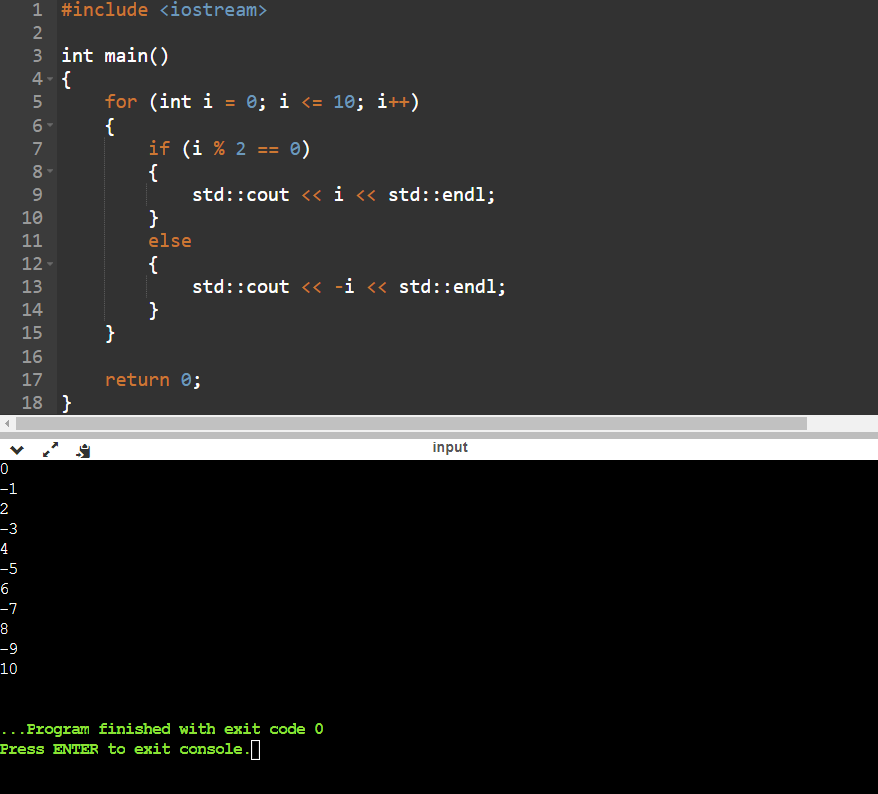
**// False thì thực hiện khối lệnh này**

**}**

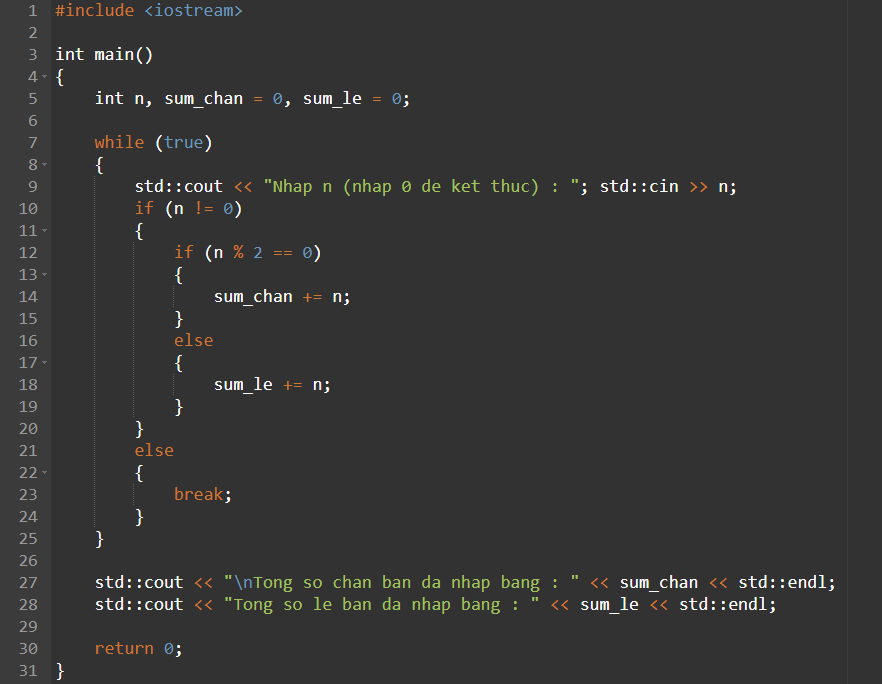
**}**

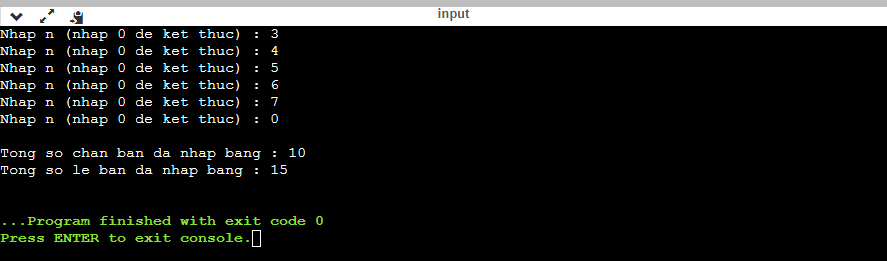
* + Cấu trúc lặp lồng rẽ nhánh gồm vòng lặp bọc bên ngoài và bên trong là cấu trúc rẽ nhánh if else để kiểm tra các điều kiện, nếu là True thì in ra if và ngược lại thì thực hiện khối lệnh của else.
  + Nhìn vào cấu trúc như vậy khá đơn giản nhưng vào triển khai các bài toán thì không đơn giản một chút nào.

# **Các ví dụ về cấu trúc lặp lồng rẽ nhánh**



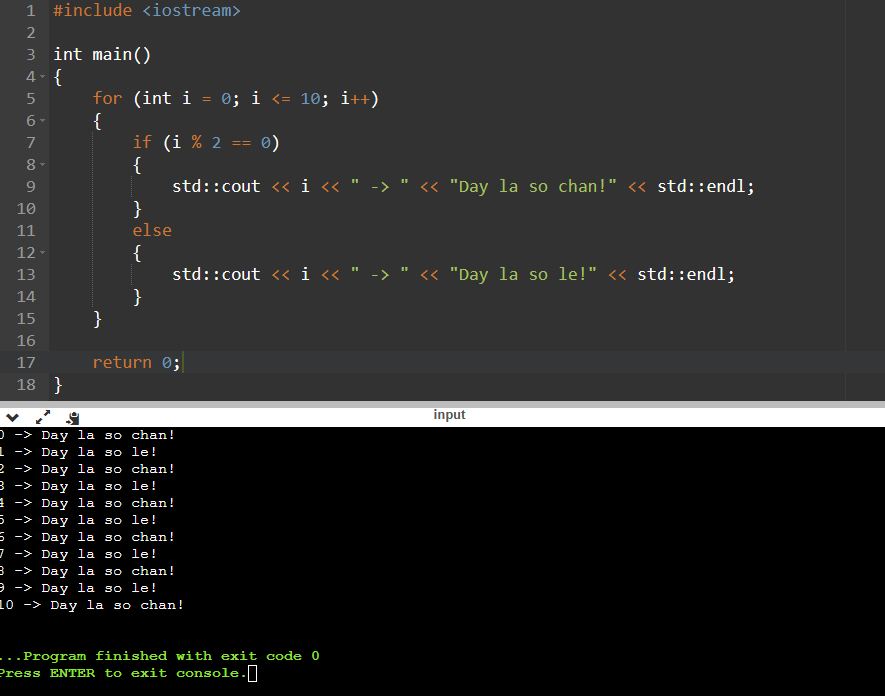
* + Trên ví dụ trên, ta khởi tạo một vòng lặp chạy từ 0 tới 10, và mỗi lần chạy như vậy tăng i lên 1 đơn vi (i++).
    - Trong đó khi mỗi lần chạy vòng lặp thì ta kiểm tra điều kiện i % 2 == 0 (i có chia hết cho 2) nếu là True thì in ra i (khi in ra i ở đây, i là số chẵn, vì lặp từ 0 tới 10 chỉ có số chẵn mới chia hết cho 2).
    - Và nếu không chia hết cho 2 là trường hợp còn lại thì sẽ rơi vào trường hợp số lẻ thì ta cho in đan xe bằng cách thêm dấu “-“.





* + Theo ví dụ trên thì ta khởi tạo các biến n để người dùng nhập, biến sum\_chan dùng để cộng các giá trị chẵn, biến sum\_le dùng để lưu tổng các biến số lẻ.
  + Khởi tạo một vòng lặp while với điều kiện luôn đúng, để người dùng nhập các số n tới khi người dùng nhập 0 thì dừng.
  + Khởi tạo một cấu trúc rẽ nhánh với điều kiện khác 0 thì vẫn nhập cho tới khi người dùng nhập 0 thì kết thúc chương trình.
  + Trong if (n khác 0) thì ta khởi tạo một if lồng trong đó và kiểm tra các điều kiện bên trong là chẵn thì thực hiện sum\_chan = sum\_chan + n. Ngược lại là số lẻ thì sum\_le += n;
  + Tới khi người dùng nhập 0 thì vòng lặp kết thúc và in ra sum\_chan và sum\_le.

# **Các bài toán sử dụng vòng lặp lồng rẽ nhánh**



**Tóm lại, cấu trúc lặp lồng rẽ nhánh sử dụng cho các cấu trúc lặp cần điều kiện để xét trong mỗi lần lặp, có thể lồng nhiều rẽ nhánh trong cấu trúc lặp để mở rộng khả năng của bài toán, thế nhưng phải chú ý việc xét điều kiện của rẽ nhánh vì nó quyết định tính đúng sai của bài toán.**

CẤU TRÚC VÒNG LẶP – CẤU TRÚC LẶP LỒNG NHAU

# **CẤU TRÚC LẶP LỒNG NHAU LÀ GÌ**

* + Cấu trúc lặp lồng nhau là cấu trúc lặp trong một cấu trúc lặp (có thể là for, for hoặc là while while) để xử lý các bài toán nâng cao.
  + Ví dụ rõ nét nhất của vòng lặp lồng nhau chính là các bài toán dạng vẽ hình.
  + Cấu trúc lặp lồng nhau có thể phụ thuộc vào các yếu tố bên ngoài nhưng cũng có cấu trúc lặp lồng nhau phụ thuộc vào nhau, ví dụ như vòng lặp j phụ thuộc vào vòng lặp i.
  + Chú ý quan trọng nhất của cấu trúc lặp lồng nhau chính là vòng lặp bên trong thực hiện xong rồi mới thoát ra ngoài thực hiện vòng lặp bên ngoài rồi tiếp tục vào lặp bên trong tới khi thoả điều kiện thì dừng.

# **CÚ PHÁP VỀ CẤU TRÚC LẶP LỒNG NHAU**

**for (int i = 1; i <= 5; i++)**

**{**

**for (int j = 1; j <= 5; j++)**

**{**

**// Khối lệnh cần lặp**

**}**

**}**

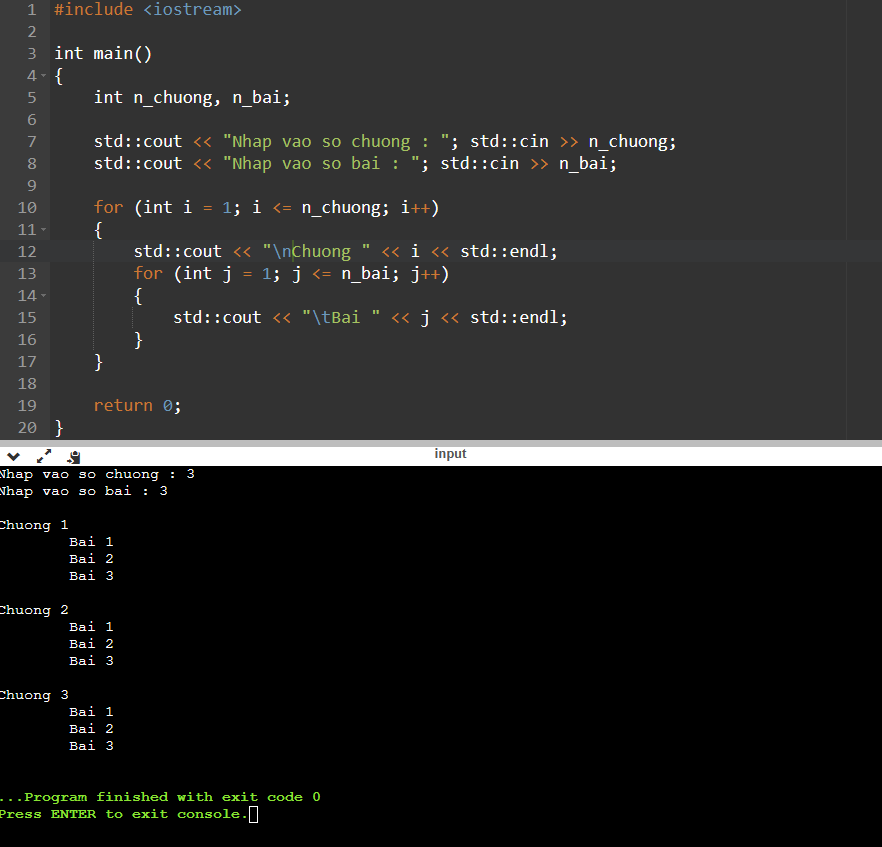
* + Cú pháp của cấu trúc lặp lồng nhau chính là vòng lặp lồng vòng lặp, với i = 1 thì chạy j từ 1 🡪 5 sau đó chạy i = 2 và j từ 1 🡪 5 và cứ thế cho tới i = 5 thì kết thúc.
  + Nhìn cấu trúc lặp lồng nhau có vẻ đơn giản nhưng không hề đơn giản một tí nào, nó tạo ra nhiều luồng hoạt động và phải thực sự hiểu cơ chế hoạt động mới hiểu về cấu trúc lặp lồng nhau.
  + Cấu trúc lặp lồng nhau hoàn có thể phụ thuộc vào nhau để tạo ra các bài vẽ hình từ cao tới thấp hoặc từ thấp tới cao.

# **NHỮNG LƯU Ý VỀ CẤU TRÚC LẶP LỒNG NHAU**

* + Cấu trúc lặp lồng nhau có thể phụ thuộc vào điều kiện bên ngoài để tạo ra một vòng lặp lồng nhau hoặc phụ thuộc vào nhau để tạo ra vòng lặp.
  + Cấu trúc lặp lồng nhau có thể lồng thêm các điều kiện rẽ nhánh để tạo thành cấu trúc lặp lồng nhau lồng rẽ nhánh.

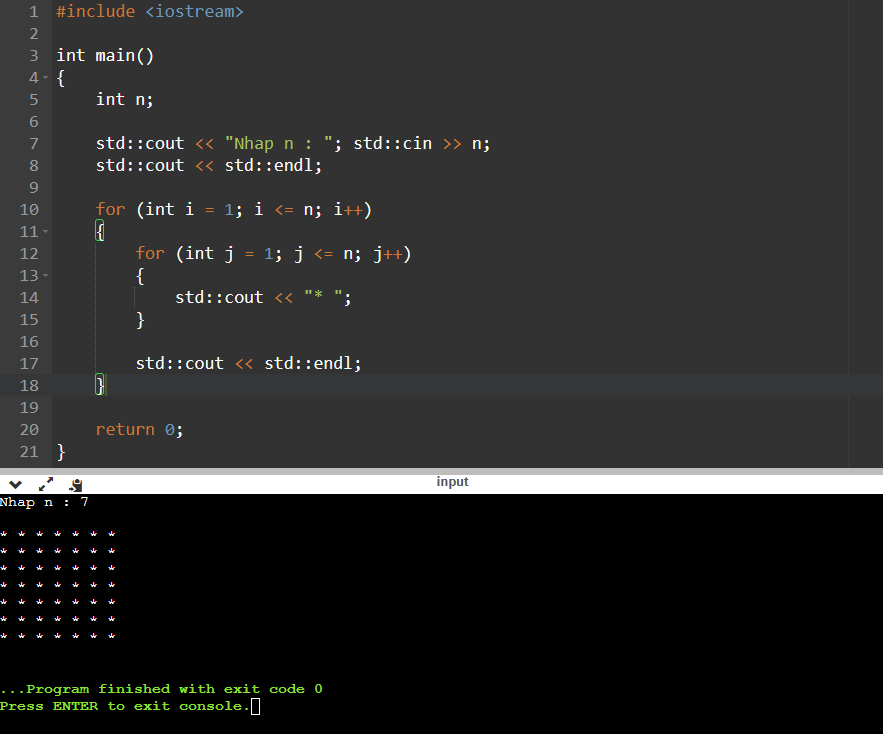
# **VÍ DỤ VỀ CẤU TRÚC LẶP LỒNG NHAU**

## **Vòng lặp in ra mục lục**



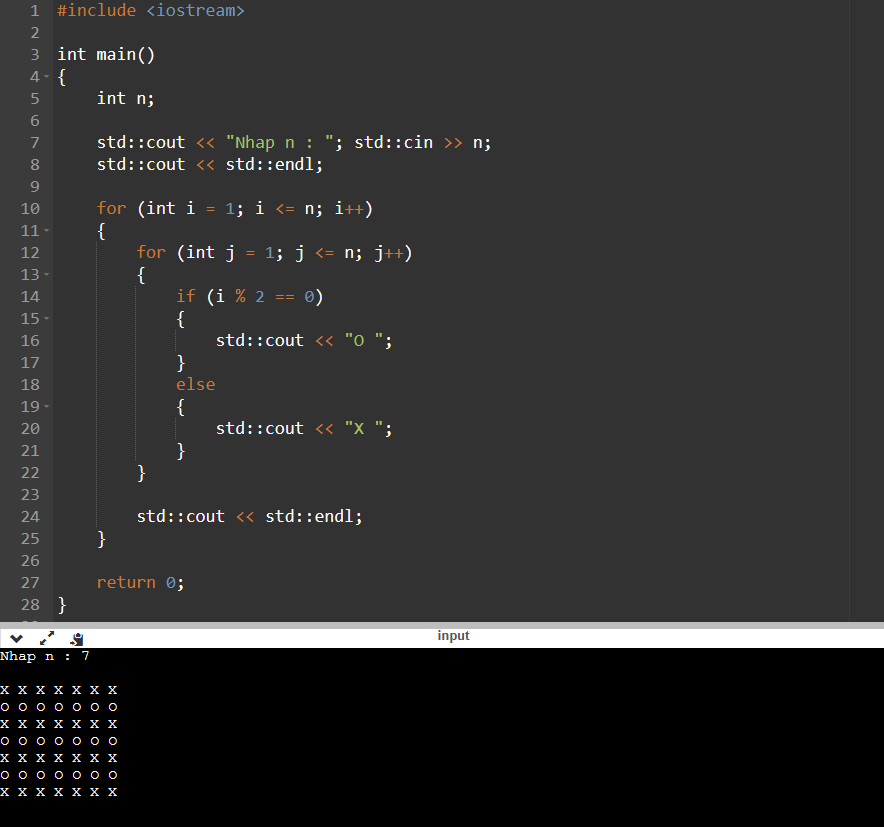
* + Ví dụ trên với số chương và số bài từ bàn phím, tạo ra mục lục bằng vòng lặp lồng nhau.
  + Ta tạo một vòng lặp cho i chạy từ i tới n\_chuong và tăng i++ sau mỗi lần chạy. Và bên trong mỗi một chương sẽ có số bài từ bài 1 tới n\_bai.
  + Ta sử dụng vòng lặp lồng nhau chạy i tới n\_chuong sau đó in ra chương rồi chạy j từ j tới n\_bai và trong một vòng lặp i chương bên ngoài sẽ tạo thành một mục lục như trên.
  + Sau khi kết thúc vòng lặp for bên trong thì thoát ra for i bên ngoài in ra chương tiếp theo rồi vào bên trong thực hiện xong rồi thoát ra vòng lặp bên ngoài tiếp tục cho tới khi thoả điều kiện.

## **Vòng lặp in ra hình vuông**



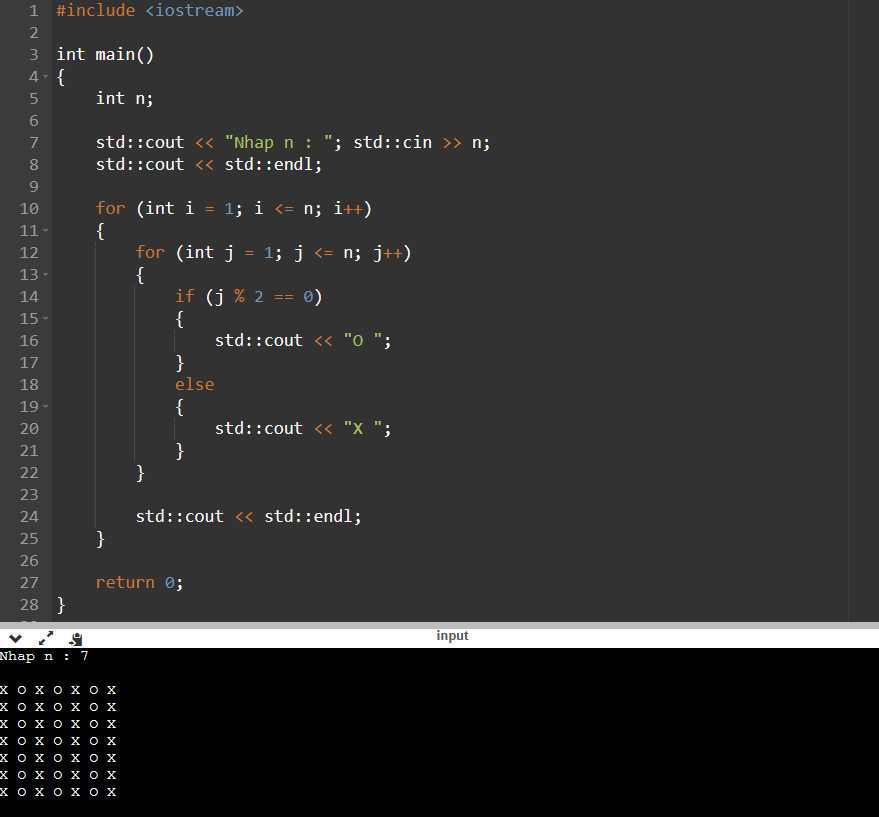
* + Ví dụ trên ta tạo một hình vuông với n được nhập từ bàn phím.
  + Ta sử dụng vòng lặp for bên ngoài với for i chạy từ i tới n (vòng lặp bên ngoài là vòng lặp cho hàng, tức là hàng 1 có n “\*” để tạo thành hình vuông).
  + Vòng lặp for bên trong với for j chạy từ j tới n (vòng lặp bên trong là vòng lặp cho hàng với n là số hàng được nhập từ bàn phím).
  + Và sau mỗi lần lặp xong 1 hàng thì xuống dòng 🡪 tạo thành hình vuông với n hàng và n cột.

## **Vòng lặp in ra hình vuông X O**



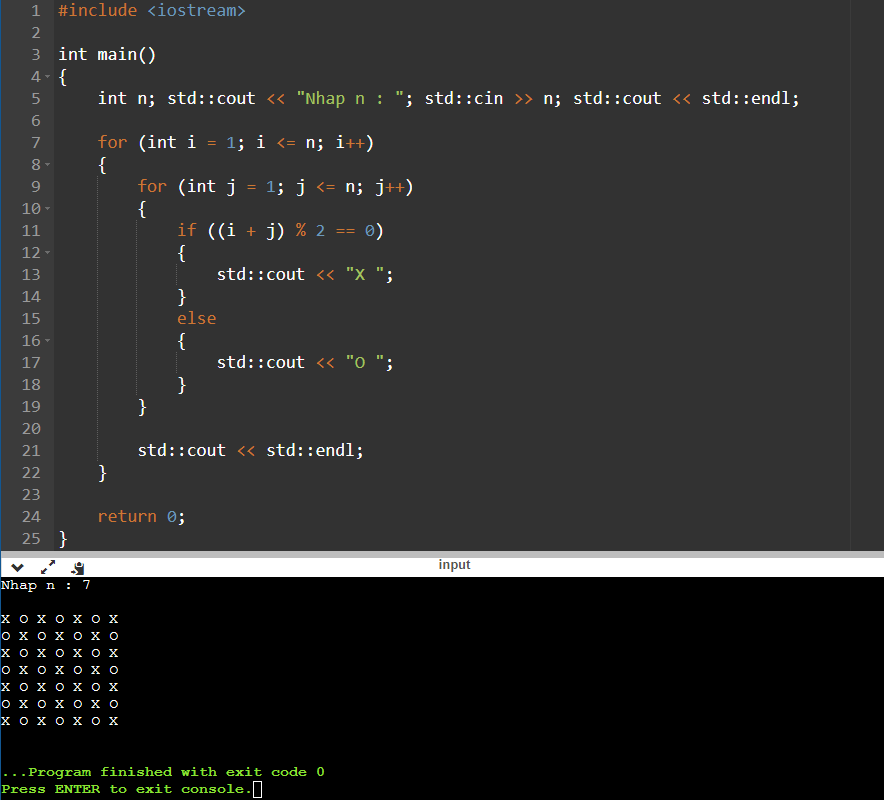
* + Ví dụ trên là vòng lặp lồng vòng lặp lồng rẽ nhánh.
  + Ta tạo vòng lặp i bên ngoài chạy từ i tới n (nhập từ bàn phím) và i được ví như hàng, với i chạy tới n thì là n hàng.
  + Vòng lặp j bên trong chạy từ j tới n (nhập từ bàn phím) và j được ví như cột, với j chạy tới n là n cột.
  + Cấu trúc rẽ nhánh nếu hàng chẵn thì cout ra O, hàng lẻ thì cout ra X.
  + Sau mỗi lần lặp thì xuống hàng endl;

## **Vòng lặp in ra hình vuông X O - 2**



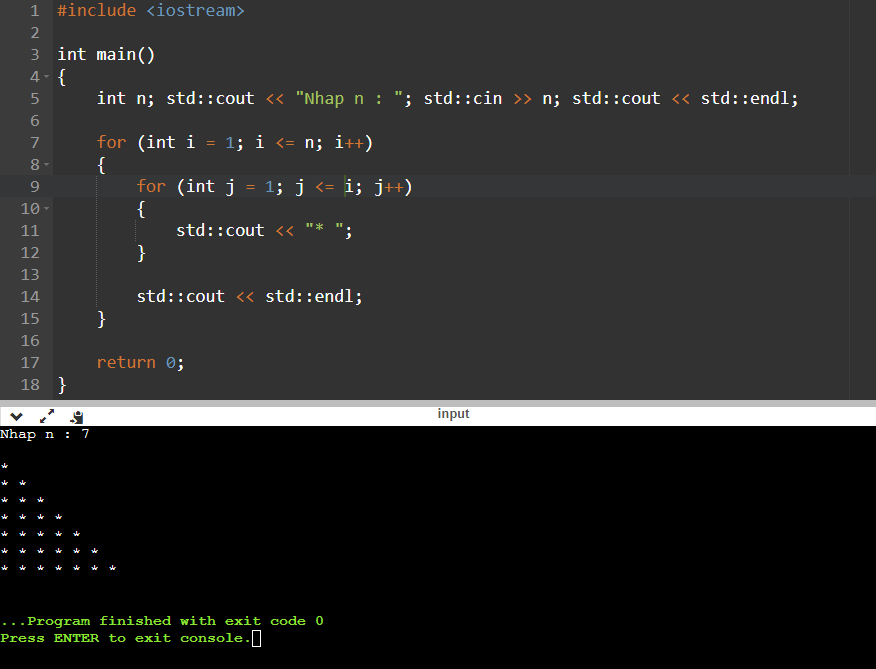
* + Ví dụ trên là vòng lặp lồng vòng lặp lồng rẽ nhánh.
  + Ta tạo vòng lặp i bên ngoài chạy từ i tới n (nhập từ bàn phím) và i được ví như hàng, với i chạy tới n thì là n hàng.
  + Vòng lặp j bên trong chạy từ j tới n (nhập từ bàn phím) và j được ví như cột, với j chạy tới n là n cột.
  + Cấu trúc rẽ nhánh nếu cột chẵn thì cout ra O, cột lẻ thì cout ra X.
  + Sau mỗi lần lặp thì xuống hàng endl;

## **Vòng lặp in ra hình vuông X O đan xen**



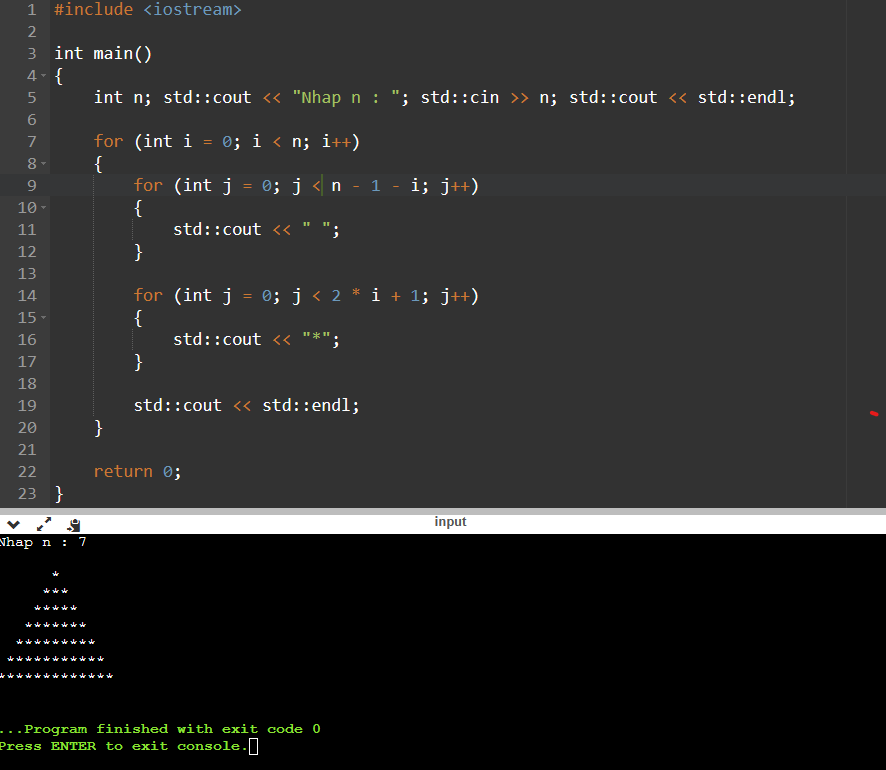
* + Trong ví dụ trên, ta có thể thấy được vòng lặp bên ngoài chạy từ i tới n với i hàng, vòng lặp bên trong chạy từ j tới n với j cột.
  + Ta có thể tạo thành ma trận X O đan xen lẫn nhau với điều kiện i + i % 2 == 0 thì X ngược lại là O.
    - Nếu tổng của chỉ số dòng i và chỉ số cột j là số chẵn, thì tại vị trí đó sẽ có giá trị là 'X'.
    - Ngược lại, nếu tổng của chỉ số dòng i và chỉ số cột j là số lẻ, thì tại vị trí đó sẽ có giá trị là 'O'.
  + Mỗi lần lặp như vậy thì xuống dòng endl;

## **Vòng lặp in ra nửa tam giác**



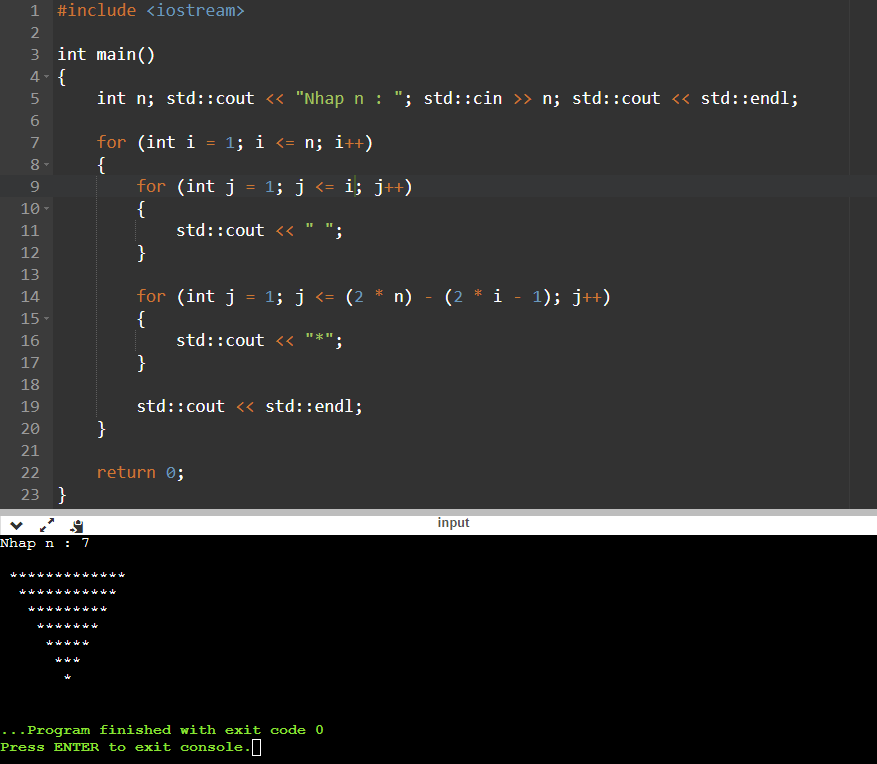
* + Trong ví dụ trên, ta sử dụng 2 vòng lặp để tạo ra nửa hình tam giác.
  + Vòng lặp i chạy từ i tới n với n hàng.
  + Trong mỗi vòng lặp i thì có vòng lặp j chạy từ j tới i, với số lượng tăng dần tới j = i = 7 thì dừng.
  + Mỗi lần lặp vòng lặp i hàng thì ta xuống dòng endl;
  + Vòng lặp i hàng tạo ra 7 hàng tương ứng với n nhập từ bàn phím.
  + Trong mỗi vòng lặp i có vòng lặp j với j cột tương ứng với hàng, ví dụ như hàng thứ 2 thì có 2 cột, hàng thứ 3 thì 3 cột, hàng thứ 4 thì 4 cột tương ứng như thế và tăng dần cho tới i <= n.

## **Vòng lặp in ra tam giác thường**



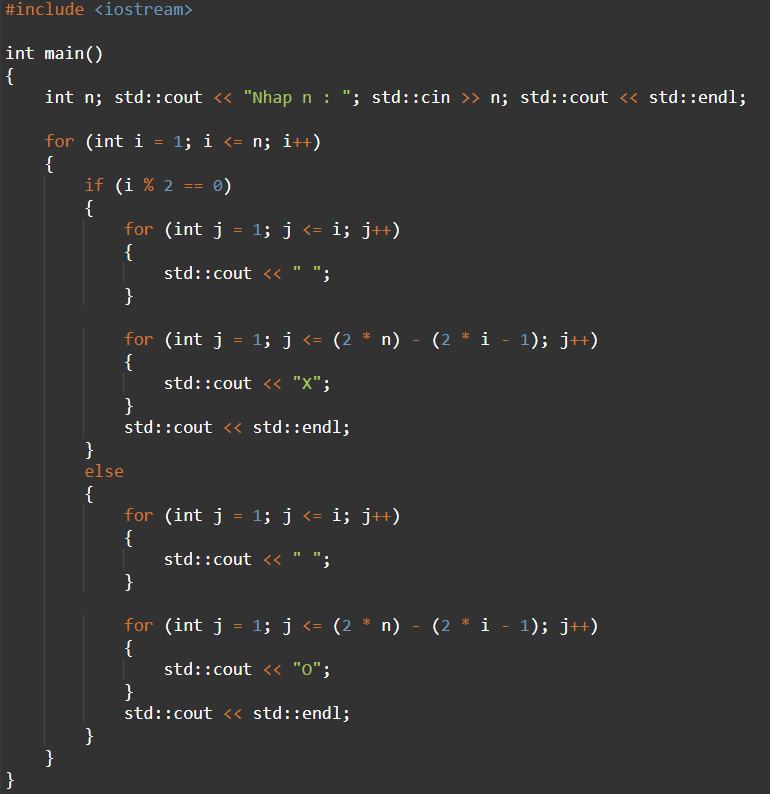
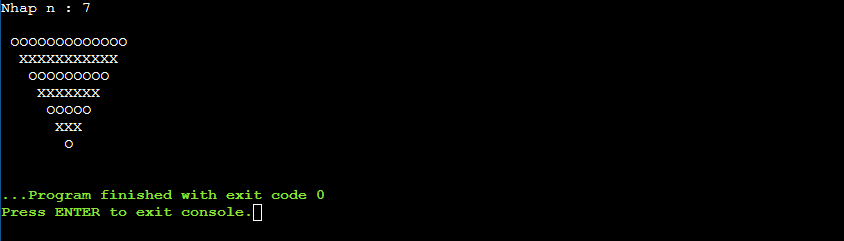
* + Trong ví dụ trên, ta tạo 3 vòng lặp lồng nhau.
  + Vòng lặp 1 chạy từ 0 tới n (với n nhập từ bàn phím) với i hàng. Ví dụ nhập 7 thì 0 🡪 7 = 7 hàng.
  + Vòng lặp j bên trong chạy từ 0 tới n – 1 – i với j cột. Ví dụ n = 7 thì sẽ giảm dần từ 6 5 4 3 2 1 với j cột.
  + Vòng lặp j bên trong in ra dấu \* chạy từ 0 tới 2 \* i + 1. Ví dụ n = 7 thì 2 \* 0 + 1 = 1 🡪 1 \* và tăng dần lên tới khi vòng lặp i thoả điều kiện i < n thì kết thúc vòng lặp.
  + Mỗi lần lặp là xuống dòng endl;

## **Vòng lặp in ra tam giác ngược**



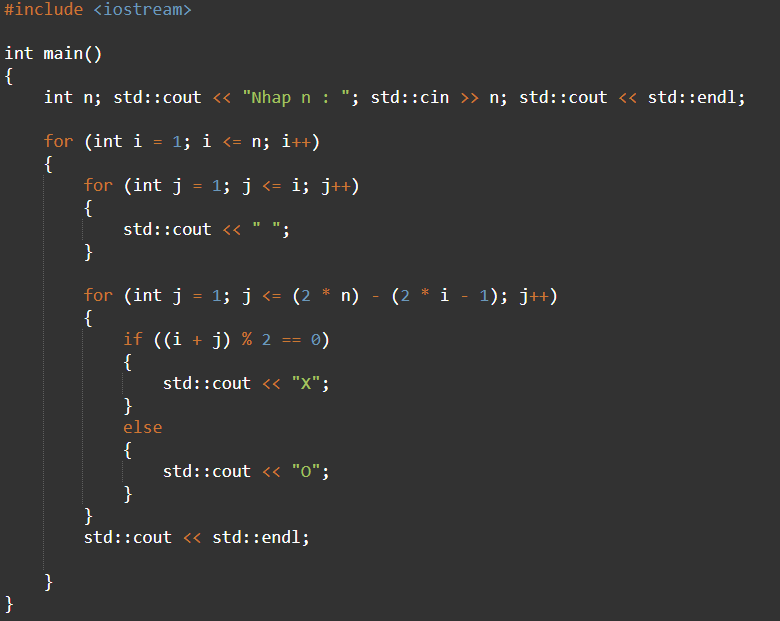
* + Với ví dụ trên ta có 3 vòng lặp lồng nhau.
  + Vòng lặp 1 chạy từ i tới n với i hàng. Ví dụ n = 7 thì chạy từ 1 <= 7 với 7 hàng.
  + Vòng lặp j\_1 chạy từ j <= i, với j cột. Ví dụ n = 7 thì chạy từ j <= i = 1 2 3 4 5 6 7. (vòng lặp in khoảng trắng thụt vào đúng với khoảng cách cần tạo tam giác).
  + Vòng lặp j\_2 chạy từ j <= 2 \* n – 2 \* i – 1 (tức là 14 -1) (14-3) Giảm dần xuống với j là số lẻ (vòng lặp in ra ngôi sao để tạo thành hình tam giác).
  + Với mỗi lần lặp xong thì xuống dòng endl;

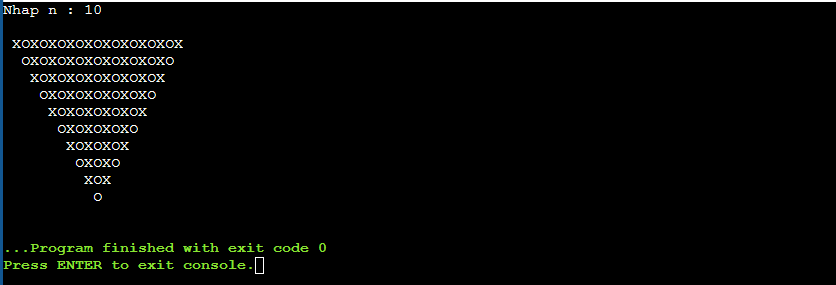
## **Vòng lặp in ra tam giác ngược X O**

* + Ví dụ trên với hàng chẵn thì là X và hàng lẻ thi là O.

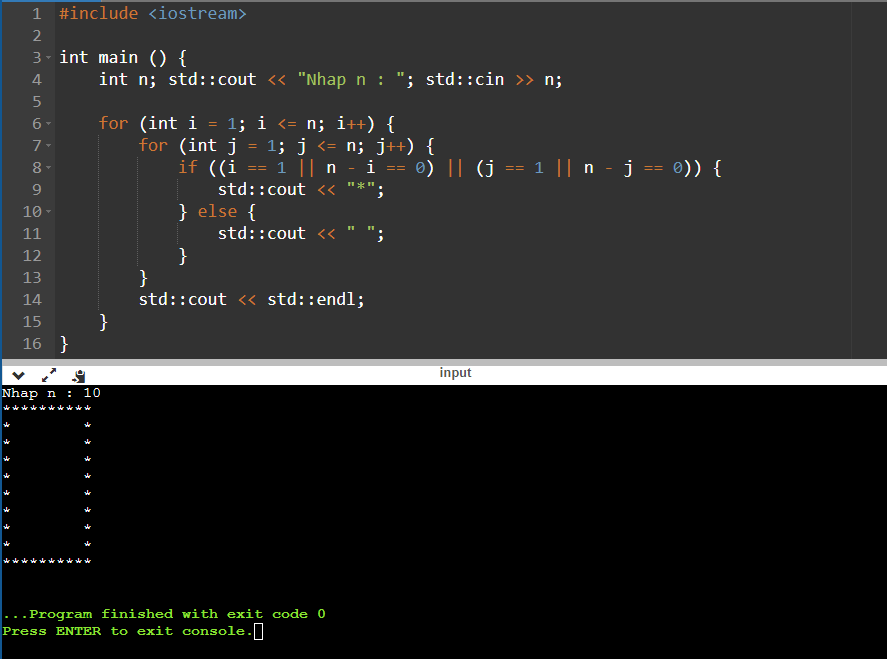
## **Vòng lặp in ra tam giác ngược X O đan xen**





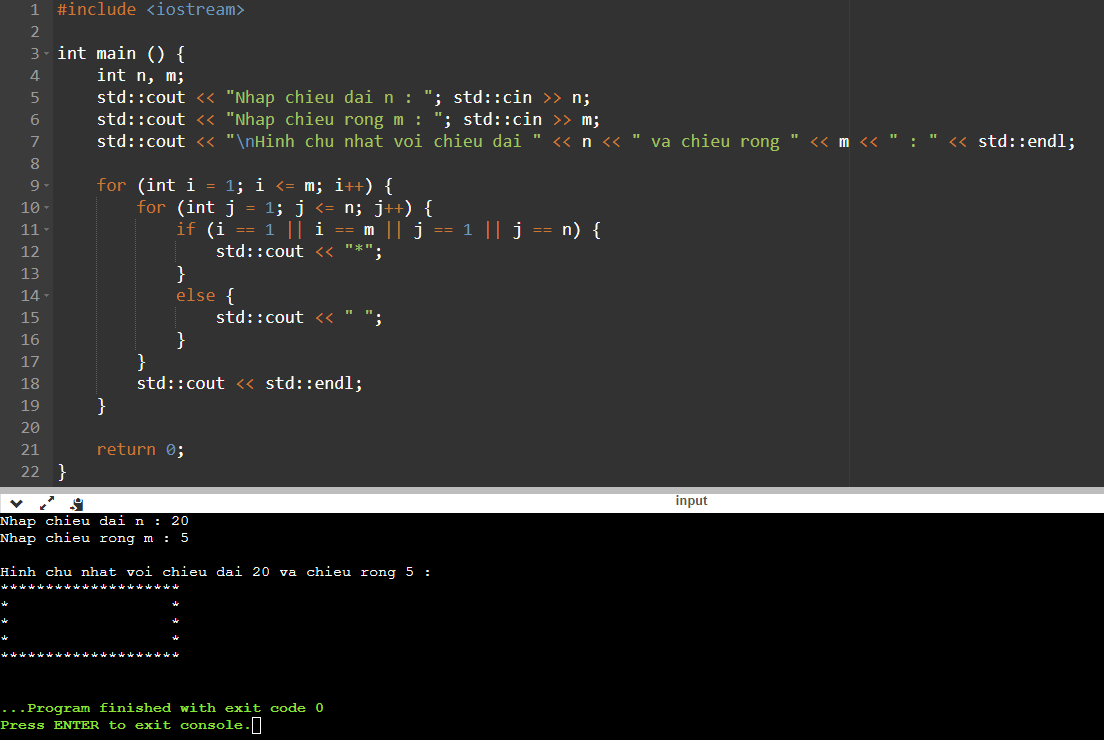
* + Ví dụ trên với X O đan xen vào nhau.

## **Vòng lặp in hình vuông rỗng**



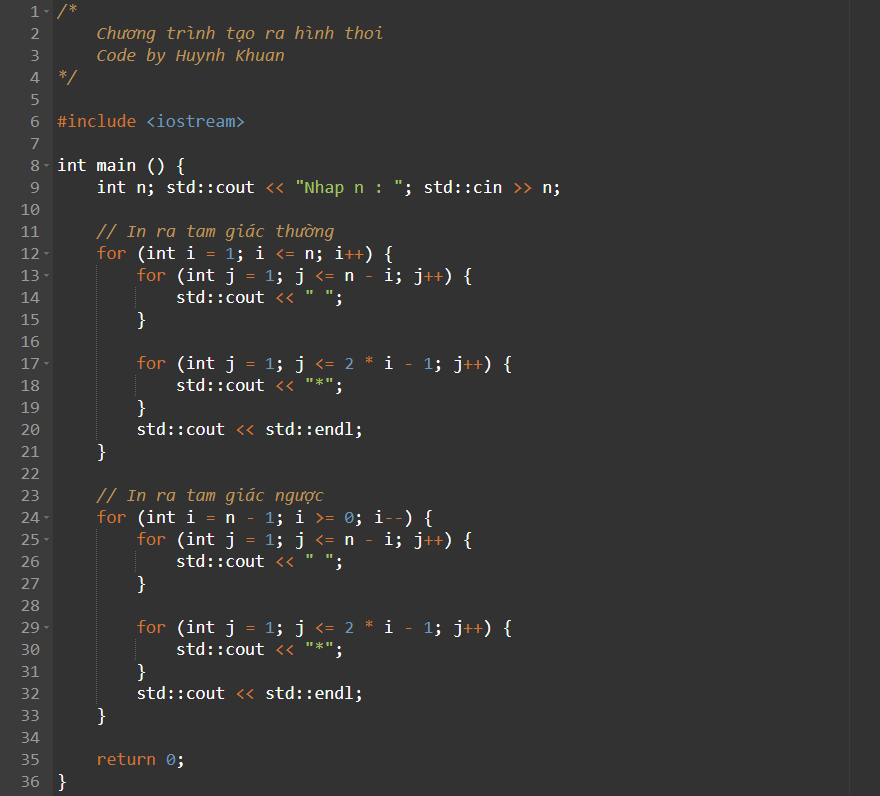
* + Ví dụ trên in ra hình vuông rỗng
  + Sử dụng 2 vòng lặp, vòng lặp i là i hàng, vòng lặp j là j hàng
  + Lồng thêm cấu trúc rẽ nhánh if (nếu là hàng 1 và hàng cuối thì in đầy đủ “\*” và nếu là cột 1 và cột cuối thì in đầy đủ “\*” còn lại là khoảng trắng.
  + Hình trên là hình vuông.

## **Vòng lặp in hình chữ nhật rỗng**



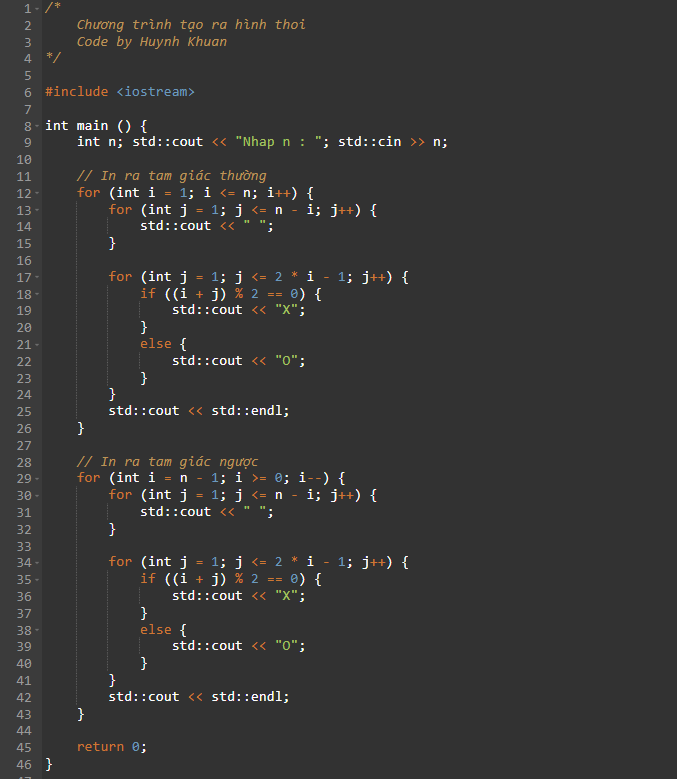
* + Ví dụ về hình chữ nhật
  + Với 2 vòng lặp, với n là chiều dài, m là chiều rồng.
  + Cho vòng lặp i hàng là chiều rộng chạy từ i tới chiều rộng m.
  + Vòng lặp i cột chạy từ j tới chiều dài n.
  + Lồng thêm cấu trúc rẽ nhánh (nếu là hàng 1 thì in ra \*, hàng = chiều rộng nhập thì in ra \*. Nếu cột = 1 thì in ra \*, nếu cột = chiều dài đã nhập thì in ra \*.
  + Mỗi lần lặp xong thì xuống dòng endl;

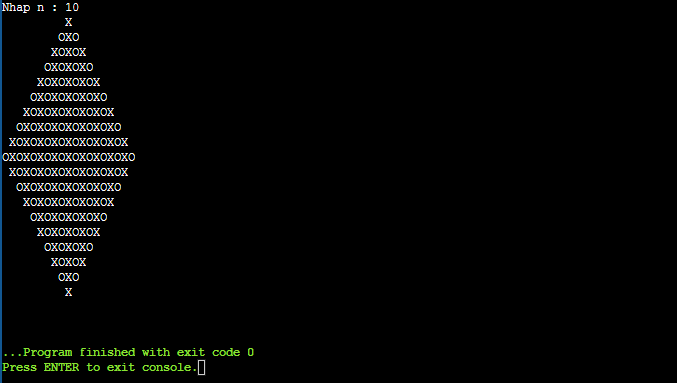
## **Vòng lặp in hình thoi**





## **Vòng lặp in hình thoi với X O đan xen**





* + Hình thoi X O đan xen nhau.

## **Vòng lặp in ra hình thoi rỗng**



* + Hình thoi rỗng
  + Nhập vào số n là kích thước của hình thoi
  + Đối với mỗi hàng i từ 1 đến 2n-1:
  + Đối với mỗi cột j từ 1 đến 2n-1:
  + Nếu i+j = n+1 hoặc i+j = 3n-1 hoặc i-j = n-1 hoặc j-i = n-1:
  + In ra ký tự "\*"
  + Ngược lại:
  + In ra khoảng trắng
  + In ra dòng mới
  + Kết thúc chương trình

**Tóm lại, cấu trúc lặp lồng nhau sử dụng cho các bài toán với độ khó cao như vẽ ra các hình, các điều kiện phải tương ứng với các hình cần tạo. Cấu trúc lặp lồng nhau mang một tầm quan trọng khá lớn, vì nắm được cấu trúc lặp lồng nhau thì gần như nắm được 50% của lập trình C++ cơ bản.**

CẤU TRÚC VÒNG LẶP – LỆNH CONTINUE VÀ BREAK

# **LỆNH CONTINUE VÀ BREAK LÀ GÌ**

## **Continue :**

* + - Trong C++, lệnh **continue** được sử dụng để bỏ qua các lần lặp hiện tại trong vòng lặp và tiếp tục vòng lặp tiếp theo.
    - Khi lệnh **continue** được gọi trong một vòng lặp **(for, while hoặc do-while)**, nó sẽ kết thúc lần lặp hiện tại và bỏ qua tất cả các câu lệnh còn lại trong lần lặp này, chuyển đến lần lặp tiếp theo. Ví dụ, trong vòng lặp for, lệnh **continue** sẽ bỏ qua toàn bộ phần thân của vòng lặp từ lệnh **continue** đến dấu đóng ngoặc nhọn của vòng lặp.
    - Tóm lại, lệnh Continue bỏ qua tất cả các câu lệnh phía dưới nó và tiếp tục vòng lặp kế tiếp.

## **Break :**

* + - Trong C++, lệnh "**break**" được sử dụng để thoát khỏi một vòng lặp (loop) hoặc một switch case statement. Khi lệnh "break" được thực thi trong một vòng lặp hoặc switch case statement, nó sẽ kết thúc vòng lặp đó hoặc thoát khỏi switch case statement đó và tiếp tục thực hiện các lệnh bên ngoài vòng lặp hoặc switch case statement đó.
    - Tóm lại lệnh **break** dùng để thoát khỏi vòng lặp và thực hiện các lệnh kế tiếp của chương trình

## **Lưu ý của lệnh continue và break :**

* + - Lệnh **continue** và **break** chỉ có **tác dụng với vòng lặp gần nó nhất** thôi. Ví dụ ta có 2 vòng lặp gần nhau thì lệnh break và continue chỉ có tác dụng với vòng lặp gần nhất.

# **CÚ PHÁP CỦA LỆNH CONTINUE VÀ BREAK**

**for (giá trị khởi tạo; điều kiện; bước nhảy) {**

**break; // kết thúc vòng lặp luôn**

**}**

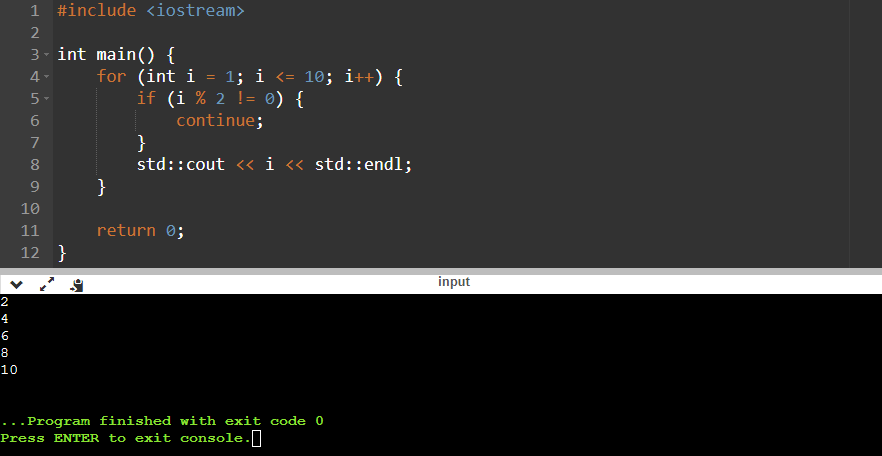
**for (giá trị khởi tạo; điều kiện; bước nhảy) {**

**continue; // Bỏ qua các lệnh dưới nó và tiếp tục vòng lặp**

**}**

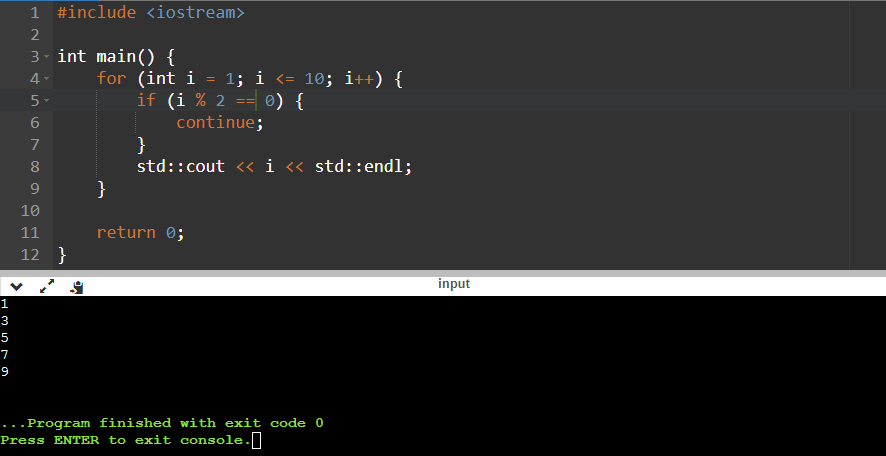
# **Ví dụ về continue và break**

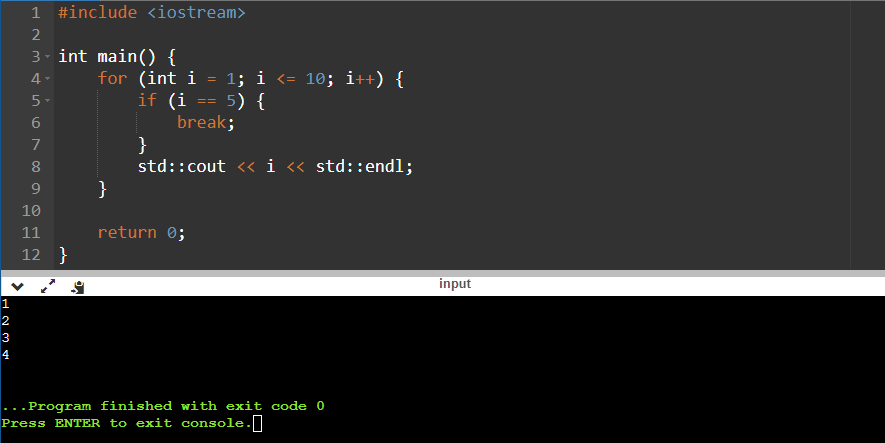
## **In ra các số chẵn**



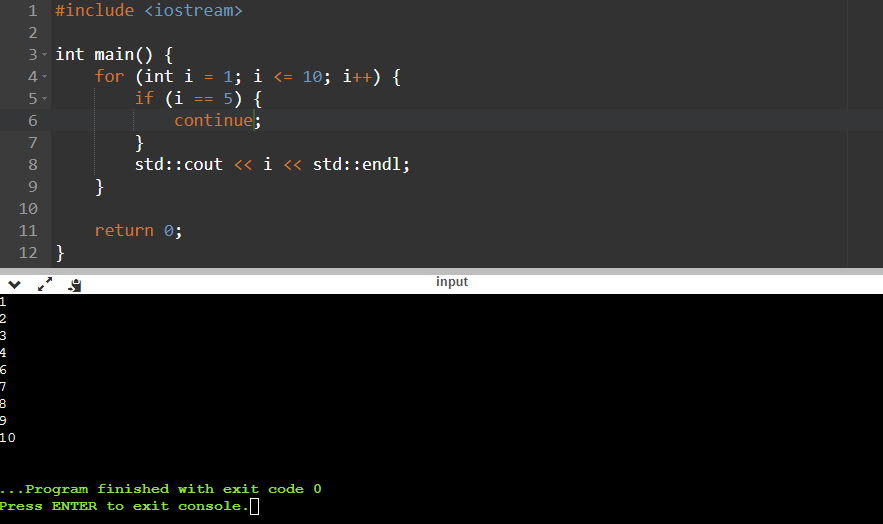
* + Với ví dụ ở trên, vòng lặp sẽ chạy từ i tới 10 và mỗi lần lặp như vậy thì tăng biến i lên 1 đơn vị.
  + Nếu i % 2 khác 0 thì tiếp tục vòng lặp và bỏ qua câu lệnh in ra i phía dưới.
  + Nếu i % 2 == 0 thì thì thực hiện câu lệnh in ra i phía dưới.

## **In ra các số lẻ**

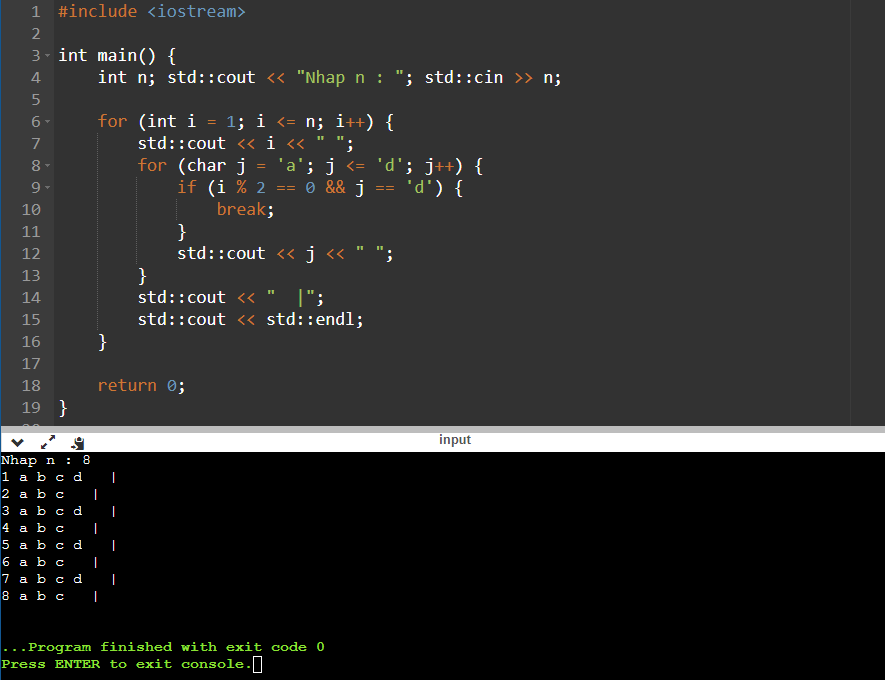




* + Nếu i == 5 thì kết thúc vòng lặp.



* + Nếu i == 5 thì tiếp tục vòng lặp mà ko in ra i, tức là bỏ qua 5.



Nếu i % 2 == 0 và j == ‘d’ thì thoát khỏi vòng lặp for j bên trong và tiếp tục in ra “ |” bên dưới 🡺 **Lệnh Continue và break có tác dụng với vòng lặp gần nó nhất.**

**Tóm lại, lệnh break và continue là 2 lệnh trong điều khiển luồng vòng lặp trong C++, chỉ có trong vòng lặp mới có 2 lệnh này, giúp cho công việc điều khiển luồng vòng lặp ngắn gọn hơn và đẹp hơn.**

CẤU TRÚC VÒNG LẶP – KĨ THUẬT GỘP / TÍCH LUỸ VỚI VÒNG LẶP.

# **KĨ THUẬT GỘP / TÍCH LUỸ TRONG VÒNG LẶP LÀ GÌ**

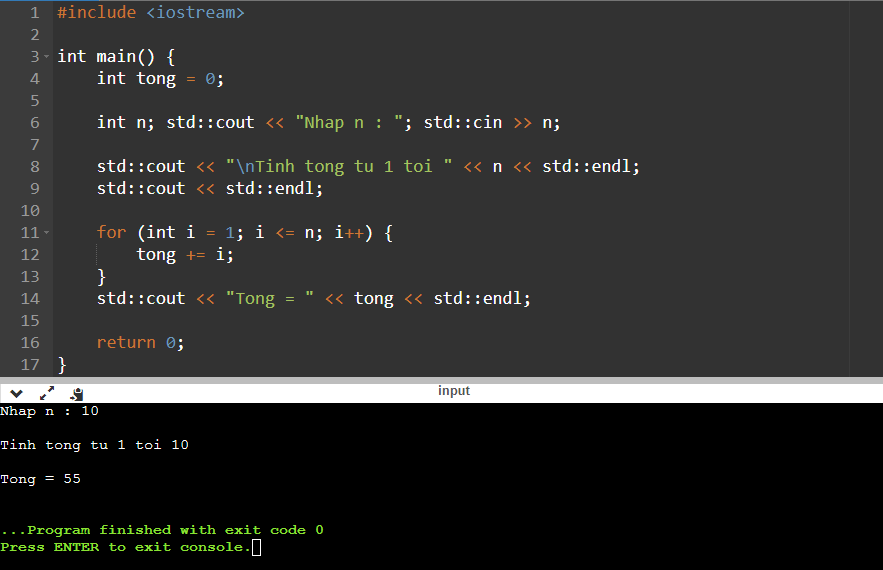
* + Kĩ thuật gộp / tích luỹ với vòng lặp trong C++ là một phương pháp tính toán tổng hoặc tích của một tập hợp các phần tử bằng cách sử dụng một vòng lặp để lặp qua các phần tử và tính tổng hoặc tích tại thời điểm lặp. Kĩ thuật này giúp tối ưu hóa việc tính toán tổng hoặc tích của một tập hợp các phần tử bằng cách tránh phải sử dụng một biến trung gian để tính tổng hoặc tích của tất cả các phần tử.
  + Tóm lại thì kĩ thuật gộp / tích luỹ với vòng lặp là tính tổng sau đó hoặc tính sau đó gán lại cho biến đó. Còn một cái khác đó chính là đếm các số trong dãy số của vòng lặp (ví dụ chạy từ i tới n có bao nhiêu số) (hoặc đếm số chẵn từ dãy số đó).
  + Kỹ thuật gộp/tích luỹ với vòng lặp là một phương pháp xử lý dữ liệu phổ biến trong lập trình và khoa học máy tính. Kỹ thuật này cho phép chúng ta tính tổng hoặc tích của một tập dữ liệu bằng cách lặp lại một phép tính nhất định trên các phần tử của tập dữ liệu.
  + **Các bước thực hiện kỹ thuật gộp/tích luỹ với vòng lặp bao gồm:**
    1. *Khởi tạo giá trị gộp/tích ban đầu.*
    2. *Lặp lại một phép tính trên từng phần tử của tập dữ liệu.*
    3. *Cập nhật giá trị gộp/tích với kết quả tính toán ở bước 2.*
    4. *Kết thúc vòng lặp và trả về giá trị gộp/tích cuối cùng.*
  + Các phép tính thường được sử dụng trong kỹ thuật này bao gồm tính tổng, tích, tối đa, tối thiểu và các phép tính logic như AND và OR.
  + Kỹ thuật gộp/tích luỹ với vòng lặp được sử dụng rộng rãi trong các thuật toán xử lý dữ liệu và tính toán số học, chẳng hạn như tính trung bình, phương sai, độ lệch chuẩn và tỷ lệ phần trăm. Nó cũng được sử dụng để tính toán các thống kê mô tả và phân tích dữ liệu.

# **MỘT SỐ LƯU Ý TRONG VIỆC SỬ DỤNG KĨ THUẬT GỘP / TÍCH LUỸ VỚI VÒNG LẶP**

* + Lưu ý trong việc sử dụng chính là việc khai báo một biến tổng hoặc tích hoặc biến đếm phải để ngoài vòng lặp (tính tự động thu hồi vùng nhớ của block trong bộ nhớ stack).
  + Các biến khi được gộp hoặc tích phải được gán lại cho biến đó hoặc sử dụng toán tử kết hợp (compound assignment).

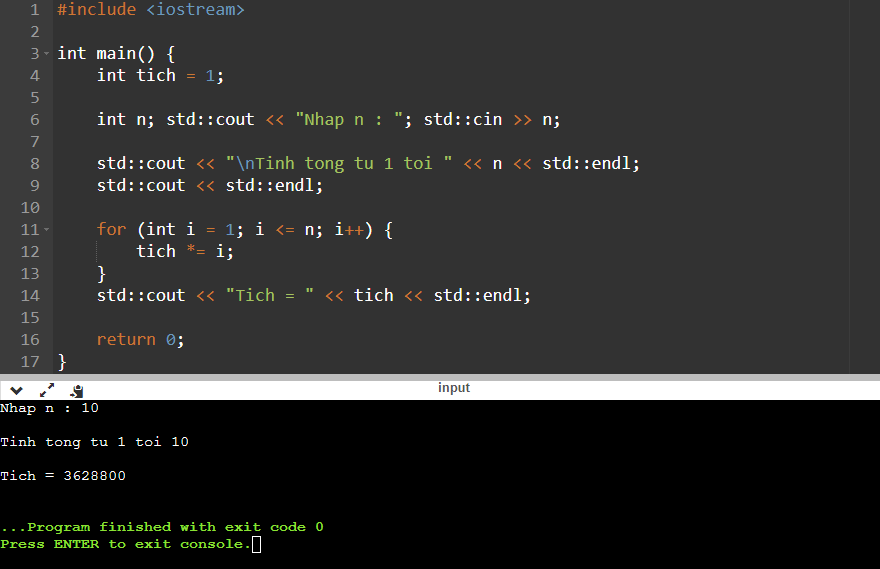
# **MỘT SỐ VÍ DỤ VỀ KĨ THUẬT GỘP / TÍCH LUỸ VỚI VÒNG LẶP**

## **VÍ DỤ VỀ KỸ THUẬT GỘP TỔNG**



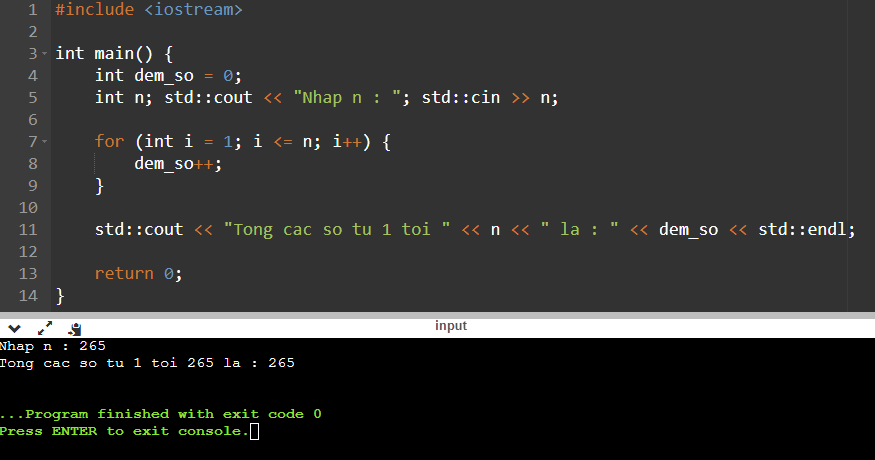
* + Tạo biến tong = 0 (để tính tổng từ 1 tới n).
  + Nhập n để thực hiện tính toán.
  + Tạo vòng lặp i = 1 lặp từ i <= n và mỗi lần lặp thì i++.
  + Với mỗi lần lặp thì ta lấy tong = tong + i (tong += i) 🡺 Để tính tổng từ 1 tới i.
  + In ra tổng.

## **Ví dụ về kỹ thuật gộp tích**



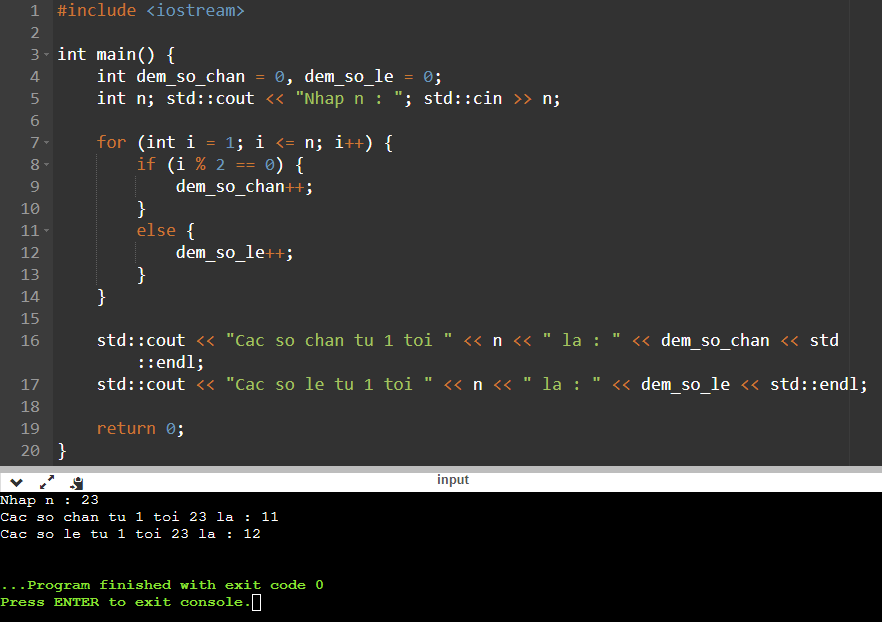
* + Tính tích từ 1 tới n.
  + Lưu ý rằng khi khai báo biến tích và khởi tạo = 0 thì số nào \* với 0 cũng bằng 0 cho nên sẽ gây ra lỗi chương trình khi khởi tạo biến tích = 0 cho nên sẽ phải khởi tạo = 1.
  + Vòng lặp chạy từ 1 <= n và mỗi lần chạy sẽ thực hiện tích luỹ biến tich = tich \* i (sau mỗi lần tich \* i thì gán lại giá trị sau khi tính toán cho biến tich) 🡪 gọi là kỹ thuật tích luỹ trong vòng lặp.
  + In ra biến tich.

## **Ví dụ về kỹ thuật đếm (gộp tổng)**



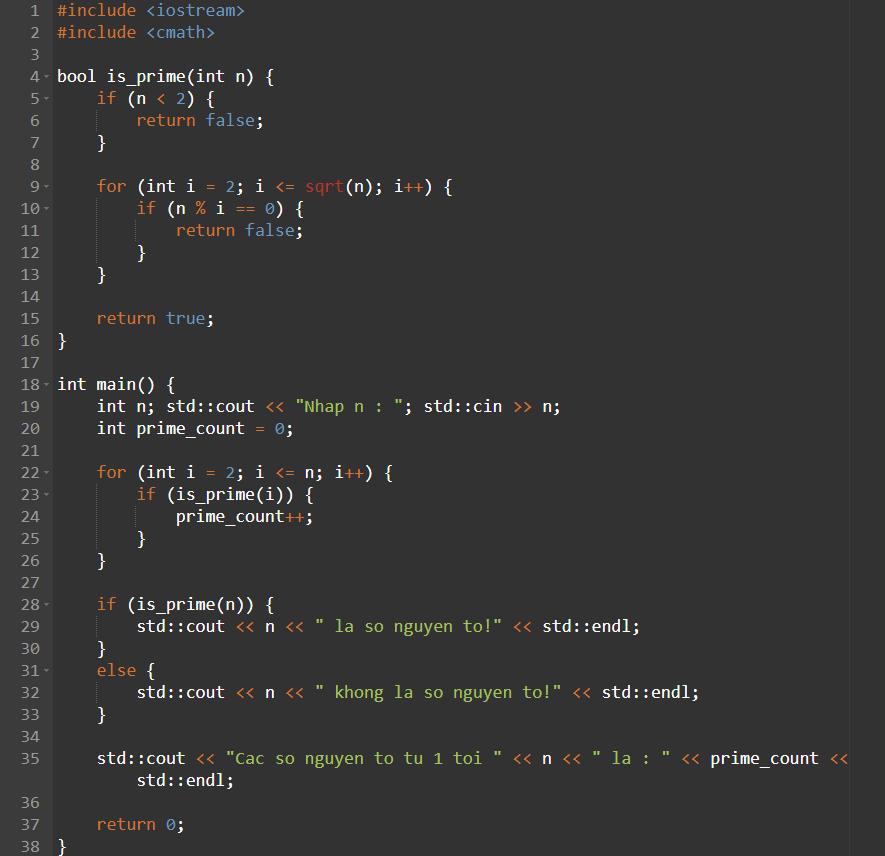
* + Vòng lặp đếm số từ 1 tới n
  + Tạo một biến dem\_so = 0 để đếm các số trong dãy số từ 1 tới n.
  + Nhập n và sử dụng vòng lặp duyệt từ i = 1 tới n và mỗi lần duyệt tăng biến i lên 1 đơn vị.
  + Trong vòng lặp cứ mỗi lần lặp và tăng i++ thì biến dem\_so++ (dem\_so = dem\_so + 1).
  + In ra có bao nhiêu số từ 1 tới n.
* Bài này khá đơn giản, vì thường không cần phải dùng tới vòng lặp, khi muốn đếm có tổng bao nhiêu số từ 1 tới n thì người ta lấy n + 1 (vì i <= n) – i.
* Ví dụ : Từ 1 tới 10 có bao nhiêu số.
  + (n + 1) – i = (10 + 1) – 1 = 11 – 1 = 10.
* Từ 1 tới 10 có 10 số.

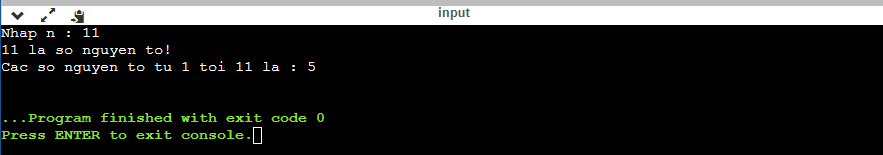
## **Ví dụ về đếm số chẵn – lẻ**



* + Đếm số chẵn và số lẻ
  + Tạo biến dem\_so\_chan và dem\_so\_le = 0 để đếm các số chẵn và số lẻ nếu thoả mãn điều kiện đã cho.
  + Nhập n để thực thực tính toán.
  + Tạo vòng lặp i = 1 lặp từ i <= n mỗi lần lặp thì tăng biến i++.
  + Trong vòng lặp với mỗi lần lặp thì ta tạo một cấu trúc rẽ nhánh với điều kiện :
    1. i % 2 == 0 (thì là các số chẵn mới % hết cho 2) 🡺 dem\_so\_chan++.
    2. Còn lại (thì là các số lẻ mới không % hết cho 2) 🡺 dem\_so\_le++.
  + In ra có bao nhiêu các số chẵn và các số lẻ.

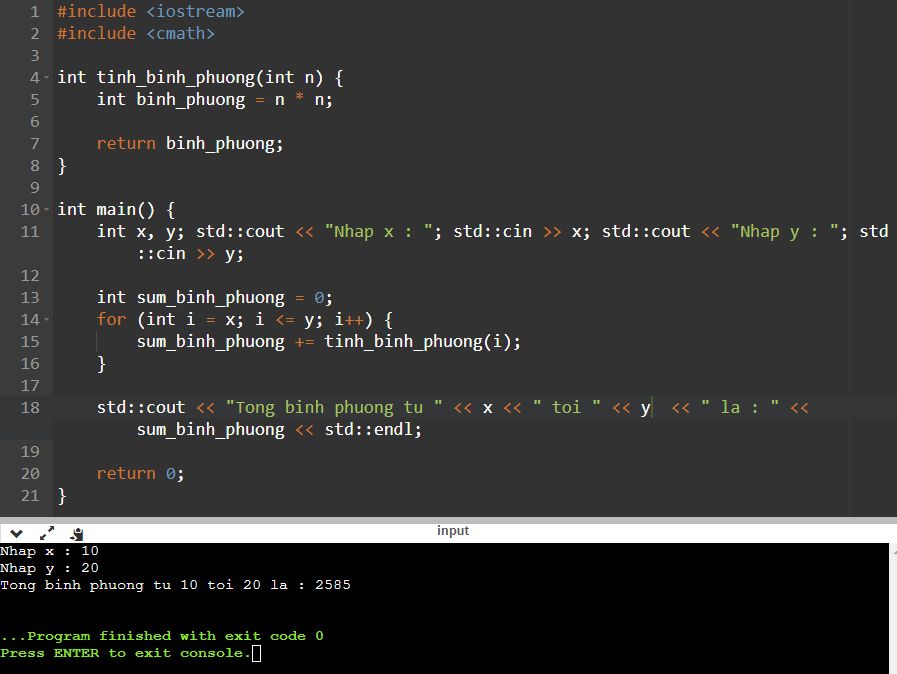
## **Ví dụ về đếm các số nguyên tố từ 1 tới n**





* + Chương trình đếm số nguyên tố từ 1 tới n

## **Ví dụ về tình tổng bình phương từ x tới y (x <= y)**



* + Chương trình tính tổng bình phương từ x tới y.
  + Tạo một hàm (function) để tính bình phương với giá trị truyền vào là int n. Giá trị trả về là binh\_phuong.
  + Nhập x, y và tạo vòng lặp i = x (lặp từ x) chạy từ i <= y (tức là x <= y); i++.
  + Mỗi lần lặp thì sum\_binh\_phuong = sum\_binh\_phuong + tinh\_binh\_phuong(i) 🡪 Với i (x) tăng dần thì tình bình phương từ x tới y sau đó cộng dồn vào sum\_binh\_phuong.
  + In ra tổng bình phương.

**Tóm lại, kỹ thuật gộp/tích luỹ với vòng lặp là một phương pháp xử lý dữ liệu phổ biến trong lập trình và khoa học máy tính. Kỹ thuật này cho phép chúng ta tính tổng hoặc tích của một tập dữ liệu bằng cách lặp lại một phép tính nhất định trên các phần tử của tập dữ liệu.**

CẤU TRÚC VÒNG LẶP

KỸ THUẬT CẮM CỜ HIỆU

# **KỸ THUẬT CẮM CỜ HIỆU LÀ GÌ?**

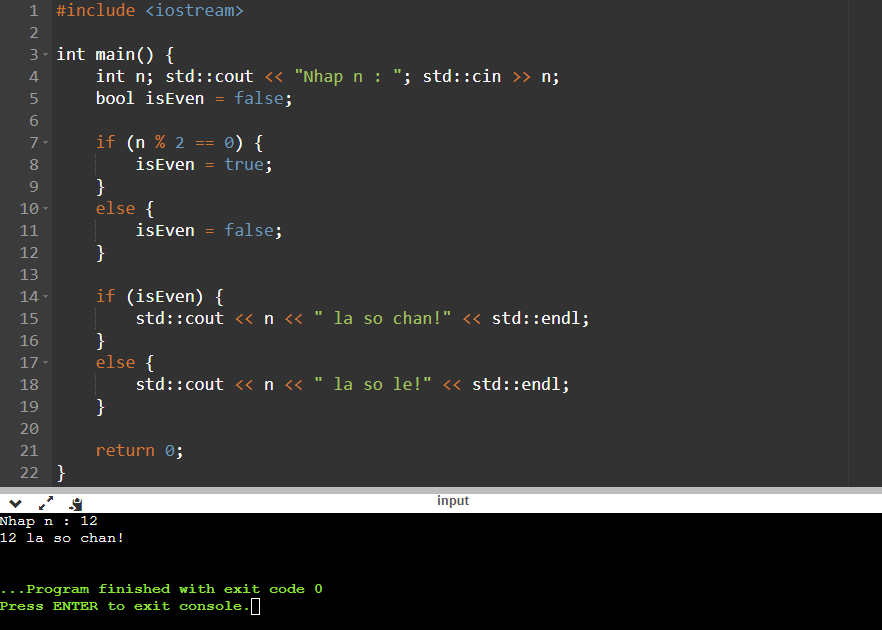
* + **KỸ THUẬT CẮM CỜ HIỆU** (**FLAG**) trong **C++** là một kỹ thuật được sử dụng để đánh dấu một **trạng thái** hoặc một **điều kiện** nào đó trong quá trình thực thi chương trình. Khi chạy chương trình, chúng ta có thể thiết lập hoặc đặt một cờ hiệu thành **TRUE** hoặc **FALSE** để đánh dấu một điều kiện nào đó. Sau đó, chúng ta có thể kiểm tra giá trị của cờ hiệu để thực hiện một số hành động cụ thể.
  + **Ví dụ**, **trong C++**, chúng ta có thể sử dụng **cờ hiệu** để kiểm tra xem một số nguyên có phải là **SỐ NGUYÊN TỐ** hay không. Trong quá trình kiểm tra, chúng ta có thể đặt cờ hiệu thành **TRUE** nếu số đó là số nguyên tố và **FALSE** nếu không phải. Sau đó, chúng ta có thể sử dụng giá trị của cờ hiệu để thực hiện các hành động khác nhau, ví dụ như **IN RA THÔNG BÁO CHO NGƯỜI DÙNG**.
  + Cách thực hiện **kỹ thuật cắm cờ hiệu** trong **C++** phụ thuộc vào cách triển khai của chương trình cụ thể. Tuy nhiên, thường thì chúng ta sẽ sử dụng các biến **bool** để lưu trữ giá trị của **cờ hiệu**. Để **đặt cờ hiệu**, chúng ta **sử dụng toán tử gán (=)** để gán giá trị **TRUE** hoặc **FALSE** cho biến **bool** tương ứng. Để kiểm tra giá trị của cờ hiệu, chúng ta có thể sử dụng các **câu lệnh điều kiện** **IF** hoặc **SWITCH** trong chương trình.

# **NHỮNG LƯU Ý TRONG VIỆC SỬ DỤNG KỸ THUẬT CẮM CỜ HIỆU TRONG C++**

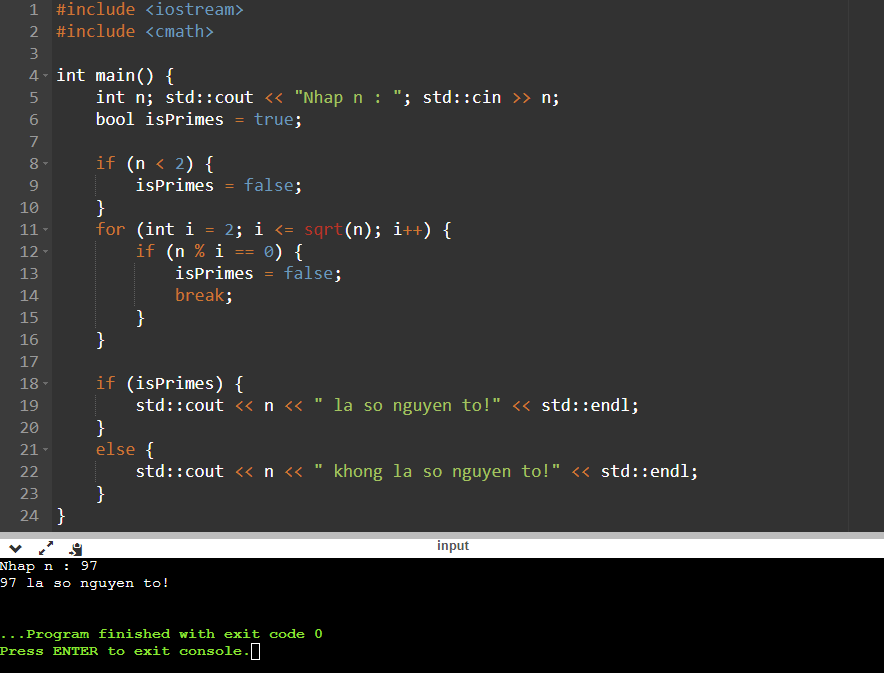
* 1. **ĐẶT TÊN BIẾN CỜ HIỆU RÕ RÀNG**: Để dễ hiểu và dễ bảo trì, hãy đặt tên biến cờ hiệu sao cho rõ ràng và miêu tả được ý nghĩa của cờ hiệu đó.
  2. **KHỞI TẠO GIÁ TRỊ BAN ĐẦU CHO CỜ HIỆU**: Trong một số trường hợp, bạn cần khởi tạo giá trị ban đầu cho cờ hiệu. Giá trị ban đầu này phải phù hợp với kiểu dữ liệu của cờ hiệu, ví dụ như false cho bool.
  3. **ĐẶT CỜ HIỆU THÀNH TRUE HOẶC FALSE**: Khi đặt cờ hiệu, hãy đảm bảo rằng bạn đặt giá trị phù hợp với mục đích sử dụng của cờ hiệu.
  4. **KIỂM TRA GIÁ TRỊ CỦA CỜ HIỆU**: Khi kiểm tra giá trị của cờ hiệu, hãy đảm bảo rằng bạn sử dụng câu lệnh điều kiện đúng cách. Nếu giá trị của cờ hiệu là true, chương trình sẽ thực hiện hành động trong khối if. Ngược lại, nếu giá trị của cờ hiệu là false, chương trình sẽ thực hiện hành động trong khối else (nếu có).
  5. **ĐẢM BẢO TÍNH CHÍNH XÁC CỦA CỜ HIỆU**: Trong một số trường hợp, bạn cần đảm bảo tính chính xác của cờ hiệu bằng cách đặt lại giá trị của nó trong quá trình thực thi chương trình.
  6. **TRÁNH SỬ DỤNG QUÁ NHIỀU CỜ HIỆU**: Nếu sử dụng quá nhiều cờ hiệu, chương trình của bạn có thể trở nên khó hiểu và khó bảo trì. Hãy đảm bảo rằng bạn chỉ sử dụng các cờ hiệu cần thiết cho mục đích của mình.

# **Các ví dụ về kỹ thuật cắm cờ hiệu trong C++**

## **Kiểm tra n có phải số chẵn hay không?**

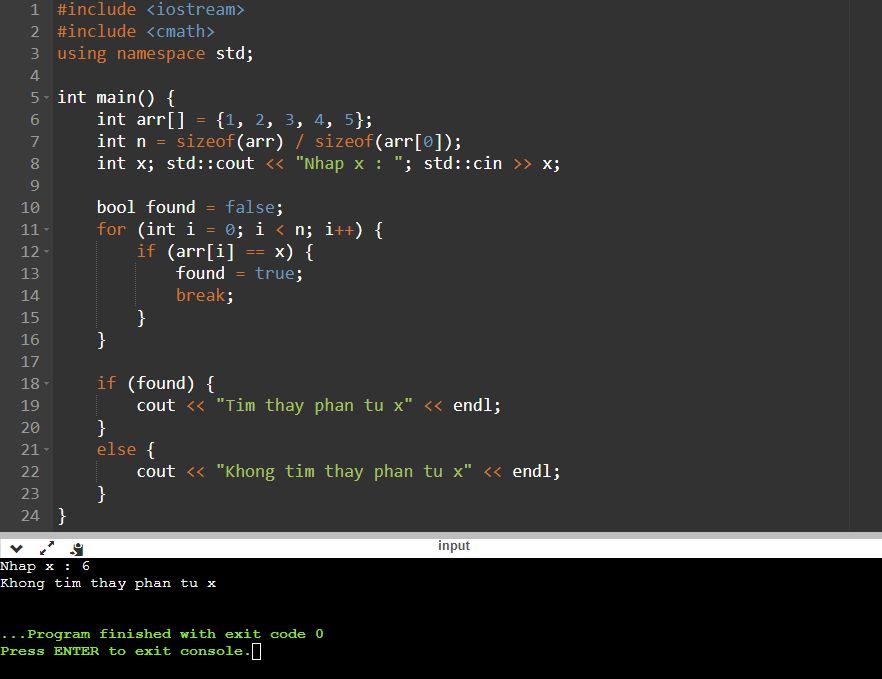


## **Kiểm tra n có phải số nguyên tố hay không?**

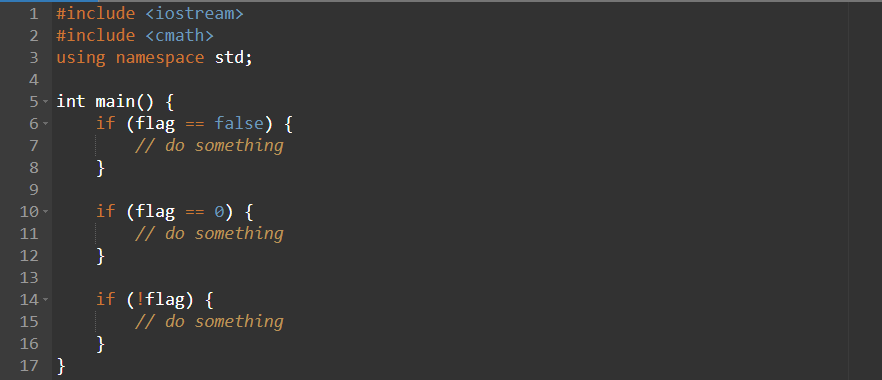


* + Khai báo n để kiểm tra n phải số **nguyên tố hay không**
  + Khai báo biến **bool** (**kiểu dữ liệu logic**) **isPrimes** = **true** (sử dụng kỹ thuật cắm cờ hiệu và khởi tạo giá trị là 1 (**true**).
  + Nếu **n < 2** 🡪 **isPrimes** = **false**.
  + Tạo vòng lặp **chạy từ 2 tới căn bậc 2 của n** (vì các **số nhỏ hơn 2 không là số nguyên tố**), mỗi lần lặp thì tăng biến **i++.**
  + Trong vòng lặp sử dụng cấu trúc rẽ nhánh với điều kiện n % i (số từ 2 tới căn bậc 2 của n) { Giải thích dễ hiểu hơn là n chia được cho số nào ngoài 1 và chính nó } thì thay đổi **isPrimes** = **false** và **break** thoát khỏi vòng lặp, không cần lặp nữa.
  + Sử dụng cấu trúc rẽ nhánh tiếp và kiểm tra nếu là số nguyên tố (**isPrimes** = **true**) thì in ra thông báo là số nguyên tố, ngược lại (**isPrimes** = **false**) không là số nguyên tố.

## **Kiểm tra phần tử đã được tìm thấy hay chưa ?**



* + Khởi tạo một mảng với 5 giá trị. Tính n = toán tử sizeof.
  + Nhập x để tiến hành kiểm tra x có nằm trong mảng không.
  + Tạo một cờ (**flag**) **bool** **found** = **false**.
  + **Tạo vòng lặp chạy từ 0 < n; i++.**
  + Nếu **arr**[**i**] == **x** (mà người dùng nhập, tức là sẽ lập duyệt mảng và nếu người dùng nhập một số có trong mảng thì **found** = **true** và thoát khỏi vòng lặp.
  + Sử dụng cấu trúc rẽ nhánh **if** (**found** == **true**) thì in ra tìm thấy phần từ **x**, ngược lại (**found** **==** **false**) thì in ra không tìm thấy phần tử **x**.



* + 3 cách viết cờ hiệu trong C++, cách cuối cùng nhìn rất chuyên nghiệp phải không?

**Cờ hiệu là một kỹ thuật quan trọng trong lập trình C++, giúp chúng ta lưu trữ thông tin về một trạng thái hoặc xử lý các điều kiện phức tạp. Khi sử dụng cờ hiệu, chúng ta tạo ra một biến boolean và đặt giá trị của nó tùy thuộc vào trạng thái hoặc điều kiện hiện tại của chương trình. Cờ hiệu có thể được sử dụng để thực hiện các nhiệm vụ như kiểm tra tính đối xứng của một chuỗi, kiểm tra tính nguyên tố của một số, kiểm tra tính đủ điều kiện để thực hiện một hành động hoặc kết thúc một vòng lặp. Sử dụng cờ hiệu trong C++ giúp chúng ta giải quyết các bài toán phức tạp một cách dễ dàng và hiệu quả.**

ĐỊNH NGHĨA FUNCTION (HÀM)

# **FUNCTION (HÀM) LÀ GÌ**

* + Trong C++, hàm (function) là một khối mã được đặt tên, có thể được gọi từ các phần khác của chương trình. Hàm thường được sử dụng để thực hiện một tác vụ cụ thể hoặc tính toán một giá trị và trả về kết quả.
  + Hàm (Function) được khai báo như một biến toàn cục (Global) và được gọi từ bất kì đâu trong chương trình.
  + Hàm (Function) là một chương trình con được định nghĩa để giảm bớt các đoạn code và nhìn đỡ rối trong hàm main(). Ta có thể chia các chương trình con nhỏ để có thể sử dụng và gọi đến bất cứ lúc nào.
  + **Hàm (Function) bao gồm các quá trình : Input 🡺 Processing 🡺 Output.**
    - Giá trị truyền vào cho hàm là gì, kiểu trả về của hàm.
    - Hàm sử lý các nội dung gì.
    - Giá trị trả về của hàm là gì (return).
  + **Định nghĩa của một hàm bao gồm các thành phần sau:**
    - Kiểu trả về (nếu hàm trả về giá trị).
    - Tên hàm.
    - Tham số đầu vào (nếu có).
    - Thân hàm, chứa các câu lệnh để thực hiện tác vụ hoặc tính toán kết quả.
    - Lệnh trả về (nếu hàm trả về giá trị).

# **VÍ DỤ VỀ FUNCTION (HÀM)**

**int sum(int a, int b) {**

**int result = a + b;**

**return result;**

**}**

* + **Trong đó:**
    - int là kiểu trả về của hàm, tức là hàm sẽ trả về một giá trị kiểu số nguyên.
    - sum là tên của hàm.
    - a và b là hai tham số đầu vào của hàm.
    - int result = a + b; là thân hàm, thực hiện tính toán tổng của hai tham số đầu vào.
    - return result; là lệnh trả về, trả về giá trị tính được từ thân hàm.

# **TÁC DỤNG CỦA FUNCTION (HÀM)**

* + Hàm (function) là một thành phần quan trọng trong C++, nó cung cấp nhiều tác dụng cho việc lập trình như sau:
    1. Tái sử dụng: Hàm cho phép tái sử dụng mã, giúp giảm thiểu sự lặp lại trong chương trình và tiết kiệm thời gian lập trình.
    2. Tách biệt chức năng: Hàm cho phép tách biệt chức năng của chương trình thành các phần nhỏ hơn, giúp dễ dàng quản lý và phát triển chương trình.
    3. Quản lý tham số: Hàm cho phép truyền các giá trị hoặc đối tượng vào hàm thông qua tham số đầu vào, giúp quản lý các giá trị và đối tượng trong chương trình một cách hiệu quả.
    4. Tính đóng gói: Hàm cho phép đóng gói mã, ẩn các chi tiết thực hiện của hàm và chỉ cung cấp các giao diện để sử dụng hàm, giúp bảo vệ mã và tránh sự xâm nhập từ bên ngoài.
    5. Tái sử dụng thư viện: Hàm cũng cho phép sử dụng các thư viện đã được định nghĩa trước đó, giúp tiết kiệm thời gian lập trình và sử dụng các tính năng đã được kiểm tra.
    6. Sắp xếp chương trình: Hàm giúp sắp xếp các phần của chương trình và làm cho chương trình trở nên dễ đọc hơn, dễ hiểu hơn và dễ bảo trì hơn.
  + Tóm lại, hàm là một công cụ quan trọng trong C++ giúp cho việc lập trình trở nên dễ dàng, hiệu quả và tiết kiệm thời gian.

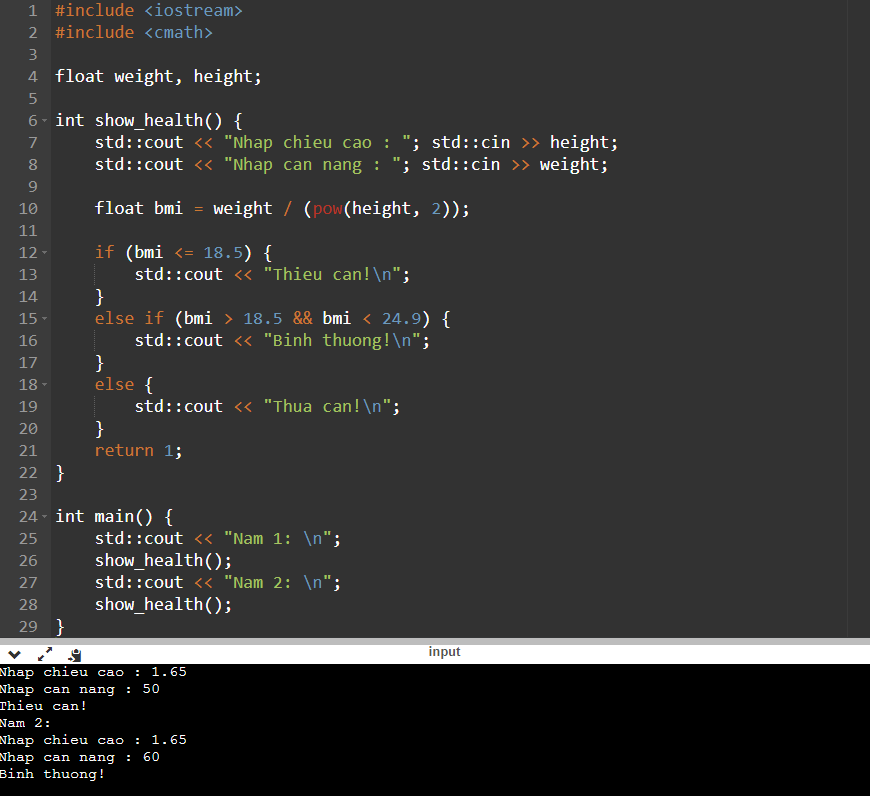
# **NHỮNG LƯU Ý VỀ FUNCTION (HÀM)**

* + **Khi sử dụng hàm (function) trong C++, có một số lưu ý sau cần lưu ý:**
    1. Định nghĩa hàm phải nằm trong phạm vi (scope) của chương trình, tức là định nghĩa hàm phải nằm trong cùng một file hoặc được khai báo trước khi được sử dụng.
    2. Tên hàm phải độc đáo và không được trùng với tên của các biến hoặc hàm khác trong chương trình.
    3. Kiểu trả về của hàm phải được khai báo đúng và phù hợp với giá trị mà hàm trả về.
    4. Tham số đầu vào của hàm phải được khai báo đúng kiểu dữ liệu và phải có số lượng tham số đúng với định nghĩa hàm.
    5. Các biến được khai báo trong hàm chỉ tồn tại trong phạm vi của hàm và không thể được truy cập từ bên ngoài hàm.
    6. Lệnh return trong hàm sẽ trả về giá trị và thoát khỏi hàm.
    7. Nếu hàm không trả về giá trị, kiểu trả về của hàm được khai báo là void.
    8. Hàm có thể được gọi nhiều lần trong chương trình.
    9. Thứ tự khai báo hàm trong chương trình quyết định thứ tự các hàm được thực thi.
    10. Hàm có thể được chuyển dưới dạng tham số cho các hàm khác.
  + Những lưu ý trên giúp cho việc sử dụng hàm trong C++ trở nên chính xác và hiệu quả, đồng thời giúp giảm thiểu lỗi trong quá trình lập trình.

# **VÍ DỤ MINH HOẠ VỀ FUNCTION (HÀM)**



* + Khai báo / định nghĩa hàm (function) với tên hàm là show\_health(); 🡺 Ở đây không có tham số truyền vào, vì đây là bài định nghĩa về hàm, tham số truyền vào sẽ được trình bày ở bài sau.
  + Nhập chiều cao / cân nặng và tính BMI của cơ thể mỗi người
  + Nếu BMI <= 18.5 thì thiếu cân, 18.5 < BMI < 24.9 thì bình thường, BMI > 25 thì thừa cân!
  + Trong hàm main() chỉ cần gọi tên hàm (function) để thực hiện, và có thể tái sử dụng nhiều lần và nhìn trong gọn hơn.
  + ***Và lưu ý rằng :*** *khi hàm main() gọi tên chương trình con thì trong hàm main() sẽ thực hiện tuần tự từ trên xuống dưới và khi chúng ta gọi hàm (function call / invoke) thì luồng chương trình sẽ chạy đến tên chương trình đó và thực hiện trả về một kết quả rồi tiếp tục thực hiện phía dưới của hàm main() theo một trình tự tuần tự.*



* + Lý do tại sao nên sử dụng Function (hàm) vì sẽ rút gọn trong hàm main() nhìn một cách thẩm mỹ cũng như gọn gàng hơn và tái sử dụng một cách linh hoạt hơn, lúc cần chỉ cần gọi hàm xuất hiện.



* + Trình tự thực hiện của hàm (function)
  + Khi ta khai báo hàm 1 và sau đó khai báo hàm 2 và gọi hàm 1 trong hàm 2.
  + Trong hàm main() ta gọi hàm 2 thì chương trình sẽ chuyển luồng lên hàm 2 và trong hàm 2 gọi hàm 1 ở dòng đầu tiên thì chương trình sẽ chuyển ta lên hàm 1 và thực hiện các khối lệnh trong hàm 1 và trả về kết quả sau đó tiếp tục thực hiện các câu lệnh sau trong hàm 2 tới khi kết thúc hàm.

**Trong C++, hàm (function) là một khối mã có thể được tái sử dụng để thực hiện một công việc cụ thể. Các hàm có thể được định nghĩa bởi người dùng hoặc được xây dựng sẵn trong thư viện tiêu chuẩn của C++.**

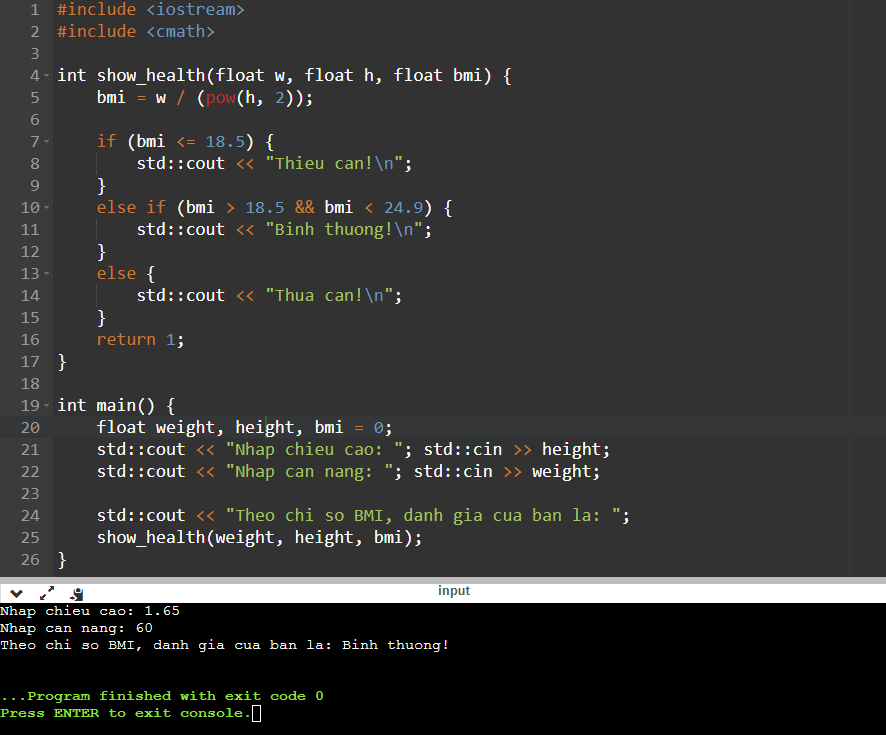
**Các hàm trong C++ có thể có một hoặc nhiều tham số đầu vào, có thể trả về một giá trị hoặc không trả về giá trị nào. Các hàm có thể được gọi bằng cách sử dụng tên hàm và các đối số tương ứng.**

PARAMETER (THAM SỐ)

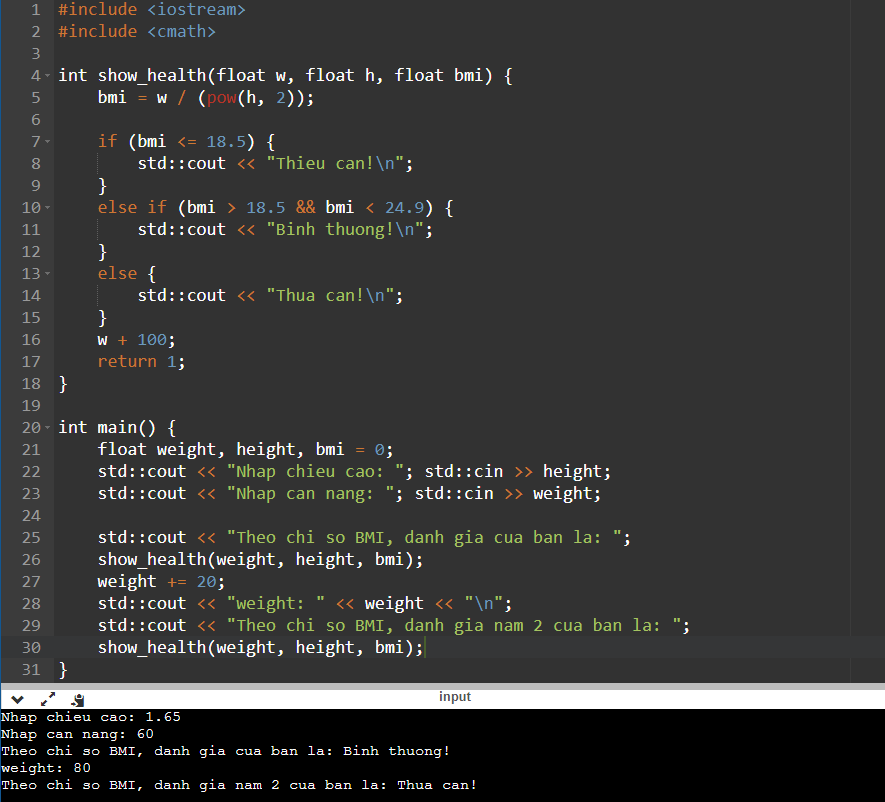
# **PARAMETER (THAM SỐ) LÀ GÌ**

* + Trong ngôn ngữ lập trình C++, tham số (parameter) là các giá trị truyền vào cho một hàm trong quá trình gọi hàm. Các tham số này được định nghĩa trong khai báo của hàm và có thể được sử dụng bên trong hàm để thực hiện các tác vụ cụ thể.
  + Một hàm có thể có 0 hoặc nhiều tham số và các tham số này có thể có kiểu dữ liệu khác nhau. Việc sử dụng tham số giúp cho hàm có thể được tái sử dụng với các giá trị đầu vào khác nhau và tăng tính linh hoạt của chương trình.
  + Việc ta muốn sử dụng một tham số (parameter) mà không cần khai báo trong hàm (function) đó hoặc khai báo một biến toàn cục (global variable) thì việc truyền tham số (parameter) là giải pháp phù hợp nhất.
  + Việc truyền vào các tham số (parameter) trong hàm (function) thì việc gọi hàm (function) trong hàm main() ta phải truyền đúng các đối số (arugment) cho hàm đó.
  + Việc truyền vào các tham số (parameter) và gọi nó trong hàm main() và truyền vào các đối số hoặc tham số phải đúng vị trí và đúng kiểu dữ liệu, nếu không có thì có thể chương trình sẽ lỗi.
  + Truyền tham số cho hàm gọi nó trong hàm main() và truyền cho nó đối số cũng như là một phép gán vậy, nó sẽ copy các biến, hoặc giá trị mà ta truyền đối số cho hàm main() và nó sẽ tạo ra một bản sao và cách truyền tham số (parameter) được gọi là pass by value (truyền theo giá trị).
  + Các thay đổi của biến tham số được truyền vào khi ta thực hiện thay đổi trên hàm (function) đó thì khi thoát khỏi chương trình con (function) đó thì giá trị đó sẽ không đổi hoặc có thể nói là không được cập nhật.

# **VÍ DỤ VỀ PARAMETER (THAM SỐ)**



* + Theo ví dụ trên, ta truyền vào 3 tham số (parameter) và ta có thể thực hiện các tính toán trên tham số đó, và việc mà ta gọi hàm và truyền vào đối số (Argument) thì chương trình sẽ thực hiện copy các đối số (argument) đó cho các tham số (parameter) trong hàm được gọi là pass by value.
  + Các biến hay tham số (parameter) là các biến cục bộ (local variable) thì khi ta gọi hàm và truyền đối số (argument) thì chương trình con đó sẽ chạy lên trên thực hiện các đoạn mã trong function (hàm) đó và tạo ra một block {}. Các biến hay các tham số chỉ có tác dụng trong block đó thôi và khi thực hiện xong hàm (function) được gọi thì block đó sẽ được bộ nhớ Stack thu hồi và các biến cục bộ (local variable) sẽ bị thu hồi.
  + Và theo đó ta thực hiện thay đổi các giá trị weight và height trong chương trình con (function) thì khi thoát khỏi function thì các giá trị đó không thay đổi. Việc truyền tham số cho hàm cũng như là gán một giá trị bản sao đó cho tham số (parameter) được truyền vào và cách này được gọi là pass by value. Thì khi đó ta thực hiện các thay đổi trong hàm đó khi thoát khỏi chương trình con thì giá trị bên ngoài hàm main() sẽ không được thay đổi.



* + Như ví dụ trên, ta thực hiện thay đổi giá trị của w (weight) trong hàm (function) con thì khi thoát khỏi chương trình con đó thì giá trị weight không hề thay đổi và được + thêm 20 ở hàm main mà không + thêm 100 trong chương trình con, theo đó ta thấy được việc truyền tham số (parameter) như việc gán weight ở hàm main() cho float w trong hàm (function) con, hai biến là độc lập nhau và không liên kết với nhau.
  + Nếu muốn một biến thay đổi trong hàm (function) con thì sử dụng tham chiếu (reference) hoặc pointer (con trỏ) để truyền các giá trị vào thì khi đó 2 biến có sự liên kết với nhau, nếu giá trị của biến trong chương trình con thay đổi thì giá trị trong hàm main cũng thay đổi theo.

# **NHỮNG LƯU Ý VỀ PARAMETER (THAM SỐ)**

* + Dưới đây là một số lưu ý về việc sử dụng tham số trong ngôn ngữ lập trình C++:
    1. Tên tham số trong khai báo hàm không bắt buộc phải giống với tên biến được truyền vào. Tuy nhiên, nên đặt tên tham số sao cho có ý nghĩa và dễ hiểu.
    2. Kiểu dữ liệu của tham số phải được khai báo rõ ràng trong khai báo hàm. Kiểu dữ liệu này phải tương ứng với kiểu dữ liệu của biến được truyền vào khi gọi hàm.
    3. Tham số được truyền vào hàm có thể là một biến, một hằng số hoặc một biểu thức.
    4. Trong hàm, tham số được sử dụng như các biến cục bộ và có giá trị được khởi tạo bằng giá trị của biến được truyền vào khi gọi hàm.
    5. Trong khai báo hàm, số lượng tham số không bắt buộc phải được khai báo. Nếu không khai báo số lượng tham số, thì hàm có thể nhận bất kỳ số lượng tham số nào khi được gọi.
    6. Trong C++, có thể định nghĩa các hàm có cùng tên nhưng số lượng hoặc kiểu dữ liệu của các tham số khác nhau (được gọi là hàm đa hình).
    7. Trong C++, các tham số được truyền vào hàm theo kiểu truyền tham chiếu (pass by reference) hoặc truyền tham trị (pass by value). Khi truyền tham chiếu, giá trị của biến được truyền vào có thể được thay đổi bên trong hàm, trong khi truyền tham trị thì giá trị của biến không thay đổi.
    8. Có thể sử dụng mặc định tham số trong C++ để định nghĩa giá trị mặc định cho một hoặc nhiều tham số. Giá trị mặc định này sẽ được sử dụng nếu giá trị tham số không được truyền vào khi gọi hàm.
    9. Trong C++, các tham số của hàm có thể được sắp xếp lại theo thứ tự bất kỳ khi định nghĩa và khi gọi hàm. Tuy nhiên, khi gọi hàm, thứ tự các tham số phải tương ứng với thứ tự khai báo của chúng trong định nghĩa hàm.

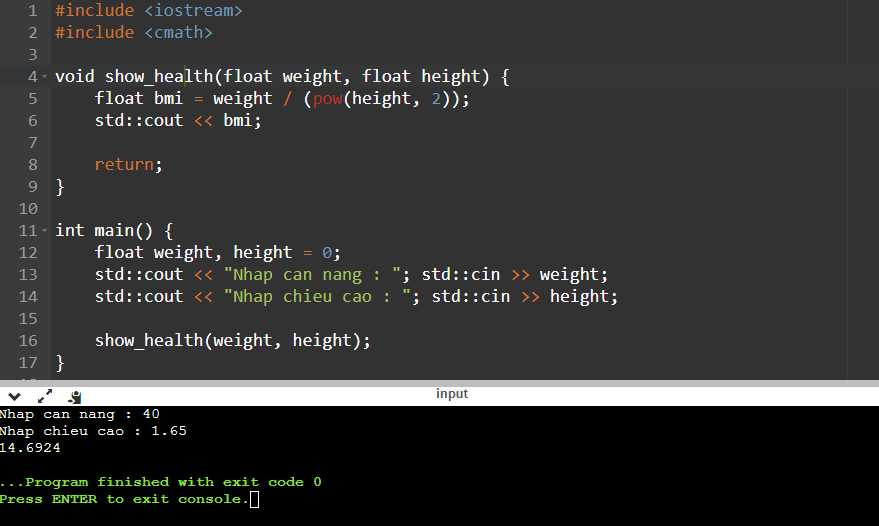
LỆNH RETURN

# **LỆNH RETURN (QUAY LẠI) LÀ GÌ**

* + Lệnh return trong C++ được sử dụng để kết thúc một hàm và trả về một giá trị cho hàm gọi.
  + **Ý nghĩa của lệnh Return :**
    1. **Lệnh return (quay lại)** : khi ta gọi hàm trong hàm main() thì luồng chương trình sẽ chạy tới chương trình con đó và thực hiện các câu lệnh trong chương trình con đó hay còn gọi là function. Và khi thực hiện xong thì quay lại luồng chương trình chính để tiếp tục các câu lệnh tiếp theo của luồng chương trình chính.
    2. **Lệnh return trả về giá trị** : lệnh return trả về giá trị khi hàm đó được khai báo có giá trị trả về thì lệnh return chính là trả về các giá trị trong hàm đó được định nghĩa, tuỳ vào từng kiểu dữ liệu mà lệnh return trả về các giá trị khác nhau.
    3. **Lệnh return dùng để kết thúc một chương trình con (function)** : lệnh return dùng để kết thúc chương trình con, khi ta gọi lệnh return thì nó sẽ trả về giá trị hoặc dữ liệu nào đó và quay trở lại luồng chương trình chính để thực hiện các câu lệnh tiếp theo. Do đó khi ta muốn kết thúc một chương trình thì sử dụng lệnh return. Lưu ý rằng lệnh return phải nằm ở sau cùng, và lệnh return nếu đặt ở trước các câu lệnh thì các câu lệnh sau return sẽ không được thực hiện.
  + Hàm có giá trị trả về thì bắt buộc phải có return để trả về một kiểu dữ liệu nào đó, ngược lại nếu là hàm void (hàm không có giá trị trả về thì không cần lệnh return hoặc cần nếu muốn kết thúc chương trình).

# **VÍ DỤ VỀ LỆNH RETURN**





# **NHỮNG LƯU Ý VỀ LỆNH RETURN**

* + **Dưới đây là một số lưu ý quan trọng về lệnh return trong C++:**
    1. Lệnh return được sử dụng để trả về giá trị từ một hàm. Giá trị được trả về có thể là bất kỳ kiểu dữ liệu nào, bao gồm cả kiểu void.
    2. Khi gặp lệnh return trong một hàm, hàm sẽ kết thúc ngay lập tức và giá trị được trả về sẽ được truyền lại cho hàm gọi.
    3. Nếu một hàm không có lệnh return hoặc lệnh return không trả về giá trị, thì hàm đó được coi là trả về giá trị kiểu void.
    4. Trong một hàm có nhiều lệnh return, chỉ có duy nhất một lệnh return được thực thi và kết thúc hàm. Lệnh return được thực thi phụ thuộc vào điều kiện được chỉ định trong từng lệnh return.
    5. Trong một số trường hợp, lệnh return có thể được sử dụng để thoát khỏi một vòng lặp hoặc một đoạn mã.
    6. Lệnh return không chỉ dừng việc thực thi của hàm hiện tại, mà còn có thể dừng toàn bộ chương trình nếu được sử dụng trong hàm main và giá trị trả về là 0.
    7. Lệnh return không được sử dụng bên trong một hàm kiểu void.

Lệnh return trong C++ được sử dụng để trả về giá trị từ một hàm. Giá trị được trả về có thể là bất kỳ kiểu dữ liệu nào, bao gồm cả kiểu void. Khi gặp lệnh return trong một hàm, hàm sẽ kết thúc ngay lập tức và giá trị được trả về sẽ được truyền lại cho hàm gọi.

Lệnh return là một trong những cấu trúc điều khiển cơ bản trong lập trình C++ và được sử dụng rộng rãi trong các hàm để trả về giá trị và điều khiển luồng thực thi của chương trình.

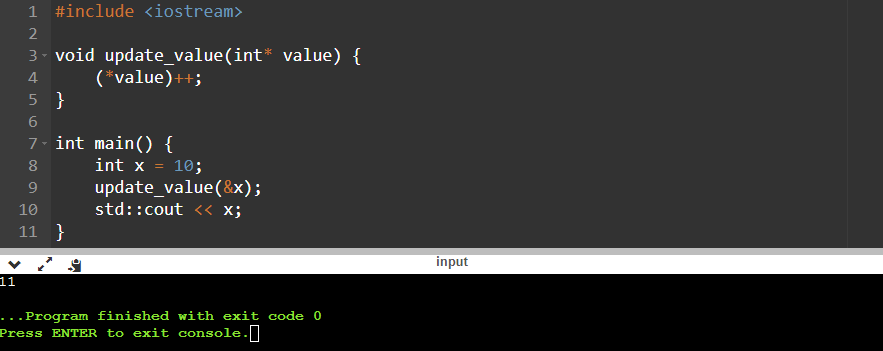
TRUYỀN THAM SỐ (PARAMETER) LÀ POINTER (ĐỊA CHỈ) VÀ REFERENCE (THAM CHIẾU)

# **TRUYỀN THAM SỐ LÀ POINTER VÀ REFERENCE LÀ GÌ**

* + Trong C++, các tham số của một hàm có thể được truyền bằng giá trị, con trỏ (pointer) hoặc tham chiếu (reference).
    1. **Con trỏ (Pointer)** là một biến đặc biệt chứa địa chỉ của một biến khác trong bộ nhớ. Khi truyền một tham số bằng con trỏ, ta truyền địa chỉ của biến đó. Trong hàm, ta có thể sử dụng con trỏ để truy cập đến giá trị của biến được truyền vào, hoặc thay đổi giá trị của biến đó. Để truyền tham số bằng con trỏ, ta sử dụng dấu \* trước tên tham số trong khai báo hàm.
    2. **Tham chiếu (Reference)** cũng là một biến đặc biệt, tuy nhiên nó không chứa địa chỉ của biến khác, mà chỉ là một cái tên khác cho biến đó. Khi truyền một tham số bằng tham chiếu, ta truyền biến đó trực tiếp. Trong hàm, ta có thể sử dụng tham chiếu để truy cập đến giá trị của biến được truyền vào, hoặc thay đổi giá trị của biến đó. Để truyền tham số bằng tham chiếu, ta sử dụng dấu & trước tên tham số trong khai báo hàm.
  + **Tuy nhiên, một số điểm khác biệt giữa con trỏ và tham chiếu là:**

1. Con trỏ có thể được khởi tạo và trỏ đến một địa chỉ bất kỳ trong bộ nhớ, trong khi tham chiếu phải được khởi tạo ngay lập tức khi khai báo, và nó phải tham chiếu đến một biến đã tồn tại.
2. Con trỏ có thể được gán lại để trỏ đến một vùng nhớ khác, trong khi tham chiếu không thể thay đổi đối tượng mà nó tham chiếu.
3. Con trỏ có thể có giá trị null, trong khi tham chiếu không bao giờ được khởi tạo bằng giá trị null.
   * **Trong Function, ta có thể truyền theo giá trị (pass by value) truyền theo địa chỉ (pass by pointer) và truyền theo tham chiếu (pass by reference).**
   * Khi ta truyền theo giá trị (pass by value) thì chương trình sẽ thực hiện copy (assigned) đối số (argument) được truyền vào một hàm, tức là thực hiện gán đối số (argument) cho tham số của hàm và biến tham số và biến trong hàm main() được truyền vào hàm là 2 biến ĐỘC LẬP nhau, không liên quan tới nhau, cho nên khi ta thực hiện các phép toán với tham số trong hàm thì biến ngoài hàm main() không bị ảnh hưởng.
   * Đặt vấn đề là ta muốn giá trị trong hàm (function) thay đổi thì giá trị bên ngoài hàm main() cũng thay đổi theo, tức là có sự liên kết với nhau, thì giải pháp cho việc này là sử dụng truyền theo địa chỉ (pass by pointer) và truyền theo tham chiếu (pass by reference).
   * **NHẮC LẠI MỘT CHÚT VỀ KIẾN THỨC POINTER VÀ REFERENCE :**
     1. **POINTER (ĐỊA CHỈ - CON TRỎ) :**
        1. Khi ta khởi tạo một biến trong hàm main() thì trong bộ nhớ Stack (Stack Memory) sẽ tạo ra một vùng nhớ chứa biến và dữ liệu đó. Theo từng vùng nhớ sẽ có từng địa chỉ riêng và là địa chỉ cho chính vùng nhớ đó, khi ta thao tác trên chính địa chỉ của biến hay vùng nhớ đó thì các giá trị trong địa chỉ đó ta có thể truy cập và thay đổi các giá trị đó.
        2. Việc lấy ra địa chỉ của một biến con trỏ (int\* ptr) phải sử dụng toán tử “&”variable. Thì ta thực hiện gán địa chỉ của biến đó cho biến con trỏ, theo đó ta thực hiện truy cập vào biến con trỏ đó thì nó cũng chính là truy cập vào chính địa chỉ của biến được lấy địa chỉ, ta thao tác trên chính địa chỉ đó cũng chính như việc ta thao tác trên biến đó, truy cập và thay đổi giá trị của địa chỉ đó cũng như việc ta thay đổi giá trị của chính biến đó.
        3. **Lưu ý** : Pointer (con trỏ) khá khó hiểu và việc phải lấy địa chỉ và truy cập vào địa chỉ của biến đó khiến cho việc lập trình khá nhập nhằng cho nên trong C++ có thêm một Reference (tham chiếu) với tác dụng y chang Pointer nhưng việc sử dụng đơn giản hơn.
     2. **REFERENCE (THAM CHIẾU) :**
        1. Reference khác với Pointer là tạo ra một biến con trỏ và gán địa chỉ của các biến khác cho biến con trỏ đó, Reference tạo ra một biến Reference và gán biến khác cho Reference, khi đó 2 biến &r và biến x (ví dụ) thì biến reference đó và biến x được liên kết chung một vùng nhớ, tức là &r và x là chung một ô nhớ trong vùng nhớ Stack và cùng trỏ tới một giá trị chung. Đó là sự khác nhau giữa biến con trỏ và Reference.
        2. Theo đó với Reference thì ta có thể thay đổi các dữ liệu, trên biến reference thì giá trị trên biến gốc giá trị cũng được thay đổi theo.

# **VÍ DỤ VỀ TRUYỀN POINTER VÀ REFERENCE VÀO PARAMETER**





# **NẾU GIÁ TRỊ OUTPUR (RETURN) LÀ POINTER HOẶC REFERENCE THÌ SAO**

* + Việc output (return) giá trị trả về là một Pointer và Reference thì tuỳ vào từng chương trình dịch để trả ra một kết quả khác nhau.
  + Ví dụ về chương trình dịch MSVC của Microsoft, thì khi ta khởi tạo một hàm xong rồi trả về một giá trị là Pointer thì MSVC có thể trả về một địa chỉ của một biến LOCAL trong hàm Function đó. Còn chương trình dịch của GCC thì không thể trả về một địa chỉ hay một giá trị nào cả. GCC cảnh báo rằng không thể trả về một LOCAL VARIABLE là Pointer, Reference tương tư như vậy.
  + **LÍ DO :** Liên quan tới Variable Scope, thì khi ta trả về địa chỉ của một giá trị output (return) thì khi đó biến nằm trong hàm function con thì địa chỉ của biến trong hàm đó đã bị thu hồi rồi thì khi đó return là một Pointer thì con trỏ (pointer) đó không trỏ tới một vùng địa chỉ nào cả cho nên sẽ là 0, reference thì tương tư như vậy biến mà reference đó liên kết với nhau đã bị thu hồi cho nên Reference không liên kết tới biến nào cả.
  + **KẾT LUẬN :** Ta không nên return Pointer và Reference đối với biến cục bộ (Local Variable) còn đối với biến toàn cục (Global Variable) thì có thể trả về một Pointer hoặc Reference tương tự.

# **NHỮNG LƯU Ý KHI TRUYỀN THAM SỐ LÀ POINTER VÀ REFERENCE**

* + Khi truyền tham số là pointer hay reference trong C++, ta cần lưu ý những điểm sau đây:
    1. Kiểm tra trước khi sử dụng: Trong hàm, ta cần kiểm tra xem con trỏ hoặc tham chiếu có trỏ đến một vùng nhớ hợp lệ hay không trước khi sử dụng nó. Nếu không, ta có thể gây ra lỗi "segmentation fault" hoặc "undefined behavior".
    2. Cẩn thận với tham số const: Nếu một tham số được khai báo là const, ta không thể thay đổi giá trị của nó trong hàm. Trong trường hợp này, ta chỉ có thể truy cập đến giá trị của nó bằng con trỏ const hoặc tham chiếu const. Nếu ta cố gắng thay đổi giá trị của tham số const, chương trình sẽ báo lỗi.
    3. Tính đúng đắn của kiểu dữ liệu: Khi truyền một tham số bằng pointer hoặc reference, ta cần đảm bảo rằng kiểu dữ liệu của tham số phải khớp với kiểu dữ liệu của biến được truyền vào. Nếu không, ta có thể gây ra lỗi "type mismatch".
    4. Tham số pointer có thể có giá trị null: Khi truyền một tham số bằng pointer, ta cần kiểm tra xem con trỏ có giá trị null hay không trước khi sử dụng nó trong hàm. Nếu ta không kiểm tra, chương trình có thể gây ra lỗi.
    5. Tham số reference không thể có giá trị null: Khi truyền một tham số bằng reference, ta không cần kiểm tra giá trị null, bởi vì tham chiếu luôn phải tham chiếu đến một biến đã tồn tại. Tuy nhiên, ta cần chú ý không được tham chiếu đến biến đã bị giải phóng khỏi bộ nhớ.
    6. Tham số pointer có thể trỏ đến vùng nhớ khác sau khi được truyền vào: Trong hàm, ta có thể thay đổi địa chỉ mà con trỏ đang trỏ đến. Nếu ta thay đổi địa chỉ mà con trỏ đang trỏ đến, nó sẽ ảnh hưởng đến giá trị của biến được truyền vào từ bên ngoài hàm.
    7. Tham số reference không thể trỏ đến biến khác sau khi được truyền vào: Trong hàm, ta không thể thay đổi đối tượng mà tham chiếu đang tham chiếu đến.

Truyền tham số là pointer và reference trong C++ cho phép chúng ta truyền biến vào trong hàm mà không cần sao chép giá trị của biến đó.

Trong trường hợp sử dụng pointer, ta truyền địa chỉ của biến vào hàm và thực hiện các thao tác trên biến đó thông qua con trỏ.

Trong trường hợp sử dụng reference, ta truyền biến trực tiếp vào hàm và sử dụng tham chiếu để truy cập đến giá trị của biến đó.

Cả hai cách truyền tham số này đều có lợi thế giúp tiết kiệm bộ nhớ và tăng hiệu suất khi thực hiện các thao tác trên biến trong hàm.

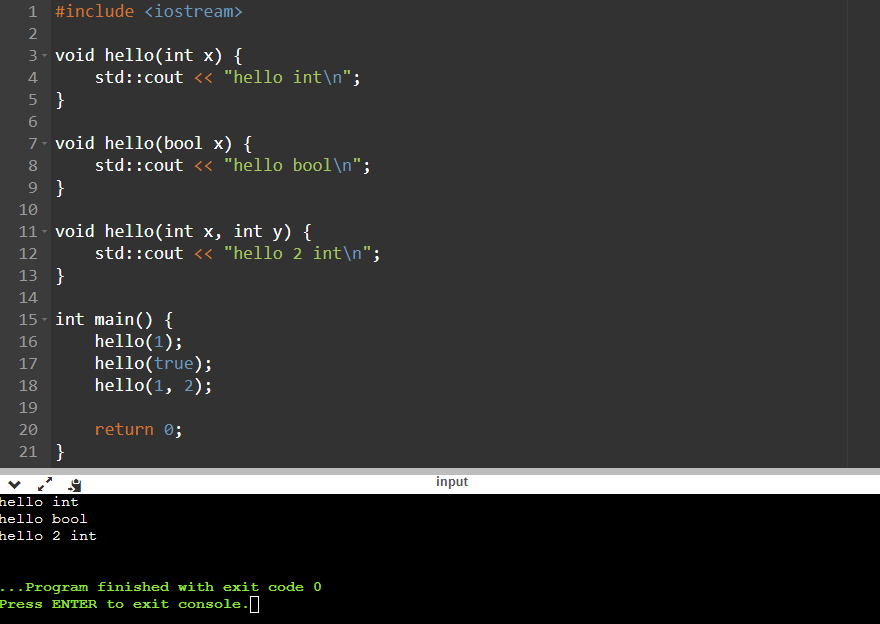
Tuy nhiên, khi sử dụng pointer hoặc reference, ta cần lưu ý những vấn đề về tính đúng đắn của kiểu dữ liệu, tránh gây ra lỗi segmentation fault hoặc undefined behavior và đảm bảo an toàn cho các biến được truyền vào trong hàm.

CƠ CHẾ FUNCTION OVERLOADING

# **CƠ CHẾ FUNCTION OVERLOADING LÀ GÌ**

* + Function Overloading trong C++ là tính năng cho phép một lớp có thể có nhiều hơn một hàm với cùng một tên, nhưng với các tham số khác nhau hoặc với kiểu dữ liệu đầu vào khác nhau. Có thể hiểu đơn giản là một hàm có thể có nhiều hơn một định nghĩa.
  + Cơ chế này cho phép các hàm được gọi dựa trên kiểu và số lượng các tham số được truyền vào. Khi một hàm được gọi, trình biên dịch sẽ kiểm tra các tham số được truyền vào và tìm kiếm phương thức phù hợp nhất để gọi.
  + Tính năng này giúp cho mã nguồn trở nên dễ đọc và dễ hiểu hơn bởi vì chúng ta có thể sử dụng các tên hàm ngắn gọn mà không phải lo lắng về việc xung đột với các tên hàm khác.
  + Tóm lại, cơ chế Function Overloading là cơ chế mà chương trình dịch cho phép đặt các Function (hàm) cùng một tên (trùng tên) với nhau mà các tham số (Parameter) truyền vào khác nhau về kiểu dữ liệu và khác nhau về tham số (nhiều hơn 1 tham số).
  + Lưu ý rằng cơ chế Function Overloading khác nhau về Parameter (input) chứ không phụ thuộc vào return giá trị nào (output).

# **VÍ DỤ VỀ FUNCTION OVERLOADING**



# **NHỮNG LƯU Ý VỀ FUNCTION OVERLOADING**

* + Dưới đây là những lưu ý cần lưu ý về cơ chế Function Overloading trong C++:
    1. Hàm được định nghĩa bằng cách khai báo nhiều lần với các tham số khác nhau hoặc kiểu dữ liệu khác nhau.
    2. Tên hàm không thể bị trùng lặp nhau trong cùng phạm vi và cùng kiểu trả về.
    3. Tham số truyền vào phải khác nhau về số lượng hoặc kiểu dữ liệu.
    4. Phạm vi truyền vào không ảnh hưởng đến cơ chế function overloading.
    5. Hàm overloaded có thể trả về các kiểu dữ liệu khác nhau hoặc không có giá trị trả về (void).
    6. Hàm overloaded có thể khác nhau về kiểu trả về, nhưng không thể khác nhau chỉ về kiểu con trỏ hoặc tham chiếu.
    7. Khi gọi hàm, trình biên dịch sẽ tìm kiếm hàm phù hợp dựa trên kiểu và số lượng tham số được truyền vào.
  + Việc sử dụng cơ chế function overloading trong C++ giúp tăng tính linh hoạt của chương trình, cho phép sử dụng nhiều hàm có cùng tên để giải quyết các vấn đề khác nhau một cách dễ dàng và hiệu quả. Tuy nhiên, để tránh gây nhầm lẫn, tên hàm nên được đặt một cách thận trọng và dễ hiểu.

Cơ chế Function Overloading trong C++ cho phép một lớp có thể có nhiều hơn một hàm với cùng một tên, nhưng với các tham số khác nhau hoặc với kiểu dữ liệu đầu vào khác nhau. Khi gọi hàm, trình biên dịch sẽ tìm kiếm hàm phù hợp dựa trên kiểu và số lượng tham số được truyền vào. Việc sử dụng cơ chế Function Overloading trong C++ giúp tăng tính linh hoạt của chương trình và giúp mã nguồn trở nên dễ đọc và dễ hiểu hơn. Tuy nhiên, tên hàm nên được đặt một cách thận trọng và dễ hiểu để tránh gây nhầm lẫn.

SỰ KHÁC NHAU GIỮA FUNCTION VÀ OPERATOR

# **SỰ KHÁC NHAU GIỮA FUNCTION VÀ OPERATOR**

* + Trong C++, Function (hàm) và Operator (toán tử) đều là các phương thức được sử dụng để thực hiện các thao tác trên dữ liệu. Tuy nhiên, chúng có một số sự khác nhau cơ bản như sau:
    - Function là một phương thức được sử dụng để thực hiện một hành động hoặc tính toán trên các giá trị được đưa vào. Operator là một phương thức được sử dụng để thực hiện một phép toán trên các giá trị được đưa vào.
    - Function có thể có bất kỳ số lượng tham số đầu vào nào, trong khi Operator thường có hai hoặc một số hạn chế về số lượng các toán hạng mà nó có thể thao tác.
    - Function có thể được gọi theo tên của nó và thông qua tham số đầu vào, trong khi Operator được sử dụng thông qua ký hiệu toán học được định nghĩa sẵn trong C++.
    - Function có thể trả về một giá trị, trong khi Operator thường trả về một giá trị hoặc một đối tượng mới.
    - Function có thể được định nghĩa bởi người dùng, trong khi Operator có một số ký hiệu được định nghĩa sẵn và hành vi của chúng không thể được thay đổi.
  + Tóm lại, Function và Operator đều là các phương thức quan trọng trong C++, và chúng có những sự khác biệt cơ bản về tính chất và cách sử dụng.

# **SỰ GIỐNG NHAU GIỮA FUNCTION VÀ OPERATOR**

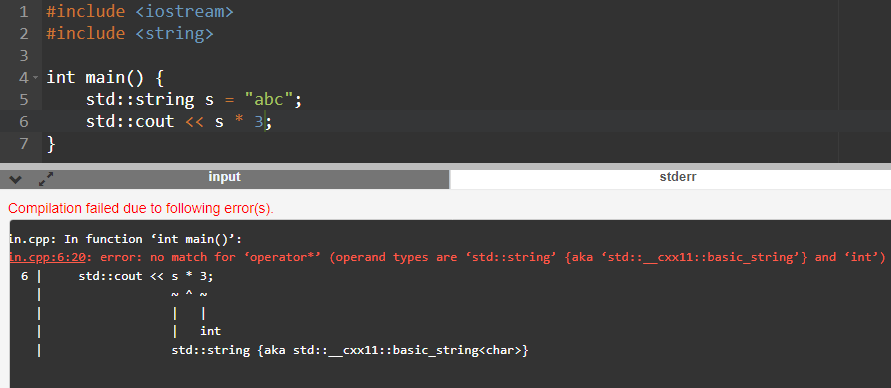
* + Mặc dù Function (hàm) và Operator (toán tử) có những sự khác biệt cơ bản, tuy nhiên chúng cũng có một số điểm chung như sau:
    - Cả Function và Operator đều là các phương thức được sử dụng để thực hiện các thao tác trên dữ liệu.
    - Cả Function và Operator đều có thể được sử dụng để thực hiện các tính toán và hành động phức tạp trên các giá trị.
    - Cả Function và Operator đều có thể trả về một giá trị.
    - Cả Function và Operator đều có thể được nạp chồng (overload) để cho phép sử dụng với các kiểu dữ liệu khác nhau.
    - Cả Function và Operator đều có thể được sử dụng để thực hiện các tính năng phức tạp hơn, như toán tử số học trên đối tượng hoặc thao tác với danh sách liên kết.
  + Tóm lại, mặc dù Function và Operator có những sự khác biệt, chúng cũng có nhiều điểm chung trong việc sử dụng để thực hiện các tính toán và hành động phức tạp trên dữ liệu.

# **LƯU Ý VỀ SỰ KHÁC NHAU GIỮA FUNCTION VÀ OPERATOR**

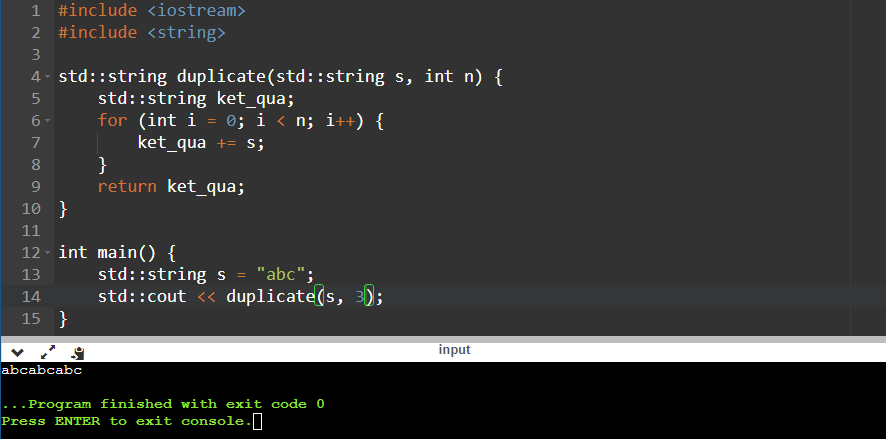
* + Sự giống nhau giữa Function và Operator là đều trả về một giá trị hay một biểu thức cụ thể.
  + **Sự khác nhau giữa Function và Operator**
    - Function (hàm) cần giá trị input và output để trả về giá trị cụ thể của một Function.
    - Operator (toán tử) thì không cần khởi tạo hay thư viện gì cả, chương trình dịch sẽ hiểu luôn các Operator là một phép toán cụ thể nào.
    - Function (hàm) đóng vai trò như một phép toán mở rộng và có thể truyền các tham số vào hàm và có thể thực hiện các biểu thức hay các phép toán phức tạp hơn nhiều.
    - Operator (toán tử) như một phép toán đơn giản và thường để xử lý các phép toán cơ bản.

# **ĐỊNH NGHĨA LẠI OPERATOR (OPERATOR OVERLOADING)**

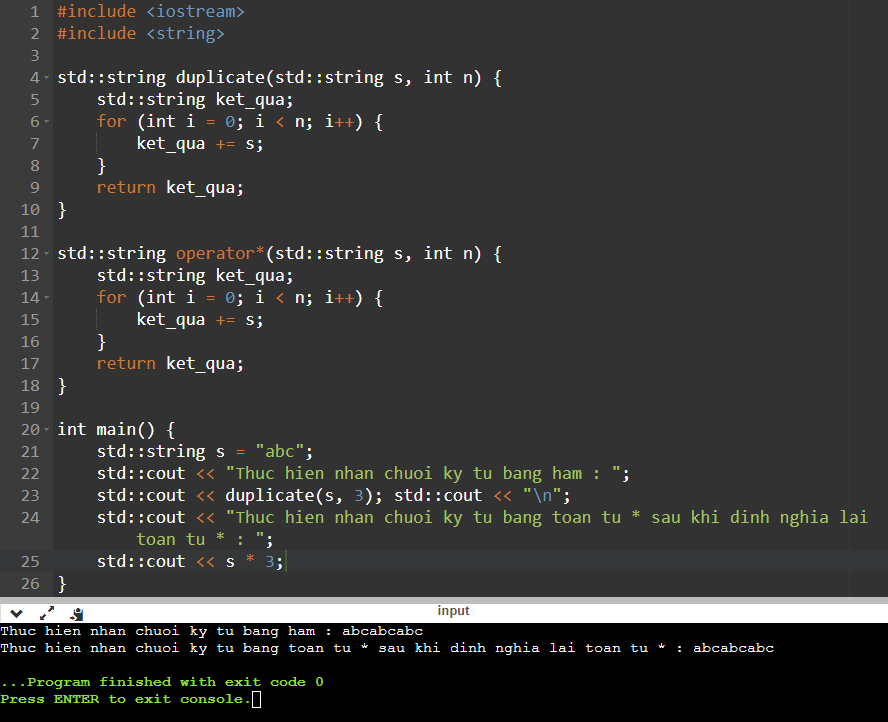
* + Operator Overloading là một tính năng trong ngôn ngữ lập trình C++ cho phép chúng ta định nghĩa lại cách thức hoạt động của các toán tử được xây dựng sẵn trong ngôn ngữ, như toán tử +, -, \*, /, %, =, >, <, >=, <=, !=, ==, ++, -- và nhiều hơn nữa.
  + Khi ta định nghĩa lại cách hoạt động của các toán tử này, ta có thể sử dụng chúng cho các lớp đối tượng do người dùng tự định nghĩa. Ví dụ, ta có thể sử dụng toán tử "+" để cộng hai đối tượng của một lớp được định nghĩa bởi người dùng, hoặc toán tử "<<" để hiển thị nội dung của một đối tượng đó.
  + Với tính năng này, C++ cho phép người lập trình tạo ra các kiểu dữ liệu tùy chỉnh và thực hiện các phép toán trên chúng giống như các kiểu dữ liệu nguyên thủy. Tuy nhiên, việc sử dụng quá mức hoặc sai cách của Operator Overloading có thể dẫn đến những lỗi khó khắc phục, vì vậy ta cần phải sử dụng tính năng này một cách thận trọng và cẩn thận.
  + Operator Overloading sử dụng trong trường hợp nào ?
    - Như ta đã biết trong các ngôn ngữ lập trình bậc cao khi ta thực hiện phép nhân với chuỗi kí tự thì hoàn toàn có thể, còn trông ngôn ngữ lập trình C++ thì ta không thể thực hiện phép nhân với các chuỗi ký tự sting ngoại trừ phép cộng.
    - Đặt trường hợp là ta có một chuỗi ký tự là “abc” và ta muốn “abc” \* 4 thì trong ngôn ngữ C++ không thể thực hiện phép nhân với chuỗi ký tự. Do đó ta phải định nghĩa lại phép nhân bằng cơ chế Operator Overloading.



* + Theo đó ta có thể thấy được việc nhân một chuỗi ký tự với một số nguyên là không thể, do đó ta phải định nghĩa lại phép nhân để có thể nhân chuỗi ký tự.



* + Ta có thể viết hàm để thực hiện nhân chuỗi ký tự.
  + Nhưng theo vài trường hợp ta không muốn thực hiện gọi một hàm dài như thế thì ta có thể gọi một ký hiệu hoặc toán tử \*, theo đó ta phải định nghĩa lại phép \* trong C++.



* + Như ta đã thấy, việc thực hiện định nghĩa lại toán tử như một hàm thì toán tử đó được thực hiện như hàm đó, do ta đã thực hiện định nghĩa lại toán tử “\*” như trong hàm ta thực hiện, đó là cơ chế Operator Overloading.
  + Tuy nhiên, cơ chế Operator Overloading nằm ở phần nâng cao của C++ cho nên chúng ta phải thật sự cẩn trọng và nghiêm túc nếu tìm hiểu nó.

**MẢNG (ARRAY)**

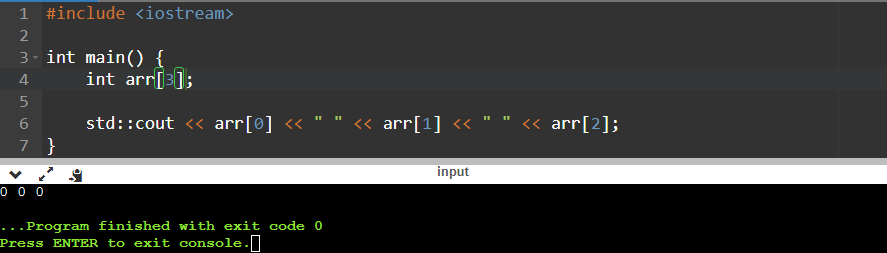
# **MẢNG (ARRAY) LÀ GÌ**

* + Mảng (array) trong C++ là một cấu trúc dữ liệu được sử dụng để lưu trữ một tập hợp các giá trị có cùng kiểu dữ liệu. Mảng được định nghĩa bằng cách sử dụng từ khóa "array" hoặc "[]" sau đó là kiểu dữ liệu của mảng và kích thước của mảng. Kích thước của mảng phải được xác định trước khi chương trình được thực thi và không thể thay đổi sau đó.
  + Mảng (array) trong C++ là một cấu trúc dữ liệu được sử dụng để lưu trữ một tập hợp các giá trị có cùng kiểu dữ liệu. Mảng là một tập hợp các biến có kiểu dữ liệu giống nhau được xếp chồng lên nhau trong bộ nhớ và được truy cập thông qua một chỉ mục (index) duy nhất.
  + Mỗi phần tử trong mảng có thể được truy cập thông qua chỉ số của nó. Chỉ số đầu tiên của mảng là 0 và chỉ số cuối cùng của mảng là kích thước của mảng trừ 1. Kích thước của mảng phải được xác định trước khi chương trình được thực thi và không thể thay đổi sau đó.
  + Một mảng trong C++ được định nghĩa bằng cách sử dụng từ khóa "array" hoặc dấu ngoặc vuông "[]" sau đó là kiểu dữ liệu của mảng và kích thước của mảng.
  + Tóm lại, mảng (array) trong C++ là một kiểu dữ liệu (hay còn gọi là cấu trúc) dữ liệu để lưu trữ các giá trị cùng một kiểu dữ liệu và được nằm cạnh nhau trong bộ nhớ Stack (xếp chồng lên nhau).

# **VÍ DỤ VỀ MẢNG (ARRAY)**



* + Khai báo mảng với 3 phần tử (element / item). Lưu ý rằng việc cặp ngoặc [] để khai báo có bao nhiêu phần tử có thể để khuyết nếu ta có Initializer list (danh sách phần tử khởi tạo). Do đó ta có thể để khuyết việc khai báo mảng có bao nhiêu phần tử. Nhưng không được phép để khuyết Initializer list khi khuyết việc khai báo phần tử cho mảng. Theo việc đó, ta không biết mảng cần có bao nhiêu phần tử thì ta có thể để khuyết số lượng element trong mảng.



* + Nếu ta để khuyết Initializer list (danh sách khởi tạo) thì giá trị mặc định của các phần tử trong mảng là một giá trị rác (một giá trị bất định) và việc này) tuỳ vào từng chương trình biên dịch khác nhau mà cho ra kết quả khác nhau.



* + Việc truy cập vào các index (chỉ mục) của mảng ta có thể sử dụng [] để truy cập vào từng phần tử trong mảng và theo việc đó ta có thể thay đổi từng phần tử trong mảng bằng cách truy cập vào từng vị trí (index) trong mảng.
  + Lưu ý : Mảng (array) sẽ bắt đầu từ 0 và kết thúc là n – 1 (ví dụ : nếu n = 5, thì mảng sẽ bắt đầu từ 0 và kết thúc là 4).
  + Lưu ý : Ta không thể khai báo một mảng với kích thước sau đó; rồi xuống dòng mới khởi tạo giá trị cho mảng, việc này là không được và sẽ bị lỗi luôn.



* + Việc số lượng phần tử (length / size) của mảng phải là một giá trị CỐ ĐỊNH, tức là giá trị không đổi. Nếu ta truyền một biến (variable) vào thì biến đó phải là một HẰNG SỐ (constant). Cho nên ta phải khai báo một biến là HẰNG SỐ thì mới được khởi tạo số lượng phần tử cho mảng.
  + Nếu mảng ta khai báo kích thước / số lượng phần tử cho mảng là một số lớn, việc ngồi khai báo hết số phần tử đó đã là một thử thách lớn, cho nên ta có thể khai báo số lượng phần tử (element) lớn nhưng ta có thể khởi tạo danh sách phần tử là một giá trị {0} thì các phần tử sau của mảng sẽ được gán giá trị mặc định là 0.
  + Lưu ý : Khi ta khởi tạo một mảng với n phần tử thì ta nên khởi tạo một giá trị là {0} để tránh gây ra các lỗi không mong muốn trong quá trình thực hiện.

# **NHỮNG LƯU VỀ MẢNG (ARRAY)**

* + Mảng là một trong những cấu trúc dữ liệu cơ bản trong C++. Nó là một tập hợp các biến cùng kiểu dữ liệu và được đánh số thứ tự, được truy cập thông qua chỉ số của nó.
  + Dưới đây là một số lưu ý quan trọng về mảng trong C++:
  + Mảng trong C++ có kích thước cố định và được khai báo với từ khóa "array".
  + Chỉ số của mảng bắt đầu từ 0. Ví dụ, arr[0] là phần tử đầu tiên trong mảng.
  + Việc truy cập một phần tử của mảng nằm trong giới hạn chỉ số của nó. Nếu truy cập phần tử nằm ngoài giới hạn, sẽ xảy ra lỗi.
  + Khi khai báo một mảng, bạn có thể khởi tạo giá trị cho các phần tử của nó.
  + Kích thước của mảng có thể được tính bằng toán tử "sizeof".
  + Mảng có thể được truyền vào hàm dưới dạng tham số. Nếu mảng được truyền vào như một tham số, nó sẽ tự động chuyển thành một con trỏ trỏ tới phần tử đầu tiên của mảng.
  + Bạn có thể sử dụng vòng lặp để duyệt qua các phần tử của mảng và thực hiện các thao tác trên chúng.
  + Bạn có thể sử dụng toán tử [] để truy cập một phần tử của mảng, hoặc sử dụng con trỏ để truy cập vào phần tử đầu tiên của mảng và sử dụng toán tử \* để truy cập các phần tử khác.
  + Khi một mảng được khai báo bên trong một hàm, các phần tử của mảng sẽ không tồn tại khi hàm kết thúc, vì chúng được khai báo trên ngăn xếp.

**Mảng là một cấu trúc dữ liệu cơ bản trong C++, được sử dụng để lưu trữ một tập hợp các biến cùng kiểu dữ liệu và được đánh số thứ tự. Các phần tử trong mảng có thể được truy cập thông qua chỉ số của chúng, và chỉ số của mảng bắt đầu từ 0.**

**Mảng trong C++ có kích thước cố định và được khai báo với từ khóa "array". Kích thước của mảng có thể được tính bằng toán tử "sizeof".**

**Khi khai báo một mảng, bạn có thể khởi tạo giá trị cho các phần tử của nó. Mảng có thể được truyền vào hàm dưới dạng tham số và bạn có thể sử dụng vòng lặp để duyệt qua các phần tử của mảng và thực hiện các thao tác trên chúng.**

**Bạn có thể sử dụng toán tử [] để truy cập một phần tử của mảng, hoặc sử dụng con trỏ để truy cập vào phần tử đầu tiên của mảng và sử dụng toán tử \* để truy cập các phần tử khác.**

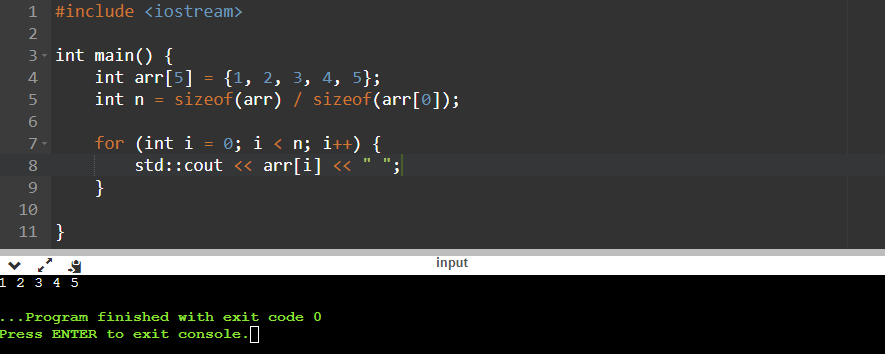
**Tuy nhiên, khi một mảng được khai báo bên trong một hàm, các phần tử của mảng sẽ không tồn tại khi hàm kết thúc, vì chúng được khai báo trên ngăn xếp.**

DUYỆT MẢNG (ARRAY) VỚI VÒNG LẶP FOR CỔ ĐIỂN

# **DUYỆT MẢNG (ARRAY) LÀ GÌ**

* + Mảng là một tập hợp các giá trị có cùng một kiểu dữ liệu và được nằm cạnh nhau trên vùng nhớ Stack.
  + Ví dụ nếu ta có một mảng gồm 1000 phần tử thì việc ngồi cout từng phần tử trong mảng đó là một việc bất khả thi, do đó ta có cơ chế duyệt mảng (iteral array) để duyệt từ phần tử đầu tiên tới phần tử cuối cùng.
  + Ở bài này, ta sẽ duyệt mảng (array) với For cổ điển.

# **VÍ DỤ VỀ DUYỆT MẢNG (ARRAY)**



Khai báo mảng với 5 phần tử và gán các phần tử lần lượt từ 0 tới n – 1 cho mảng.

Khai báo biến n là số lượng phần tử và tính n bằng lấy kích thước của mảng đó chia cho kích thước đầu tiên của mảng thì sẽ ra số lượng phần tử có trong mảng đó.

Sử dụng vòng lặp for cổ điển để duyệt các phần tử trong mảng và duyệt từ 0 tới n và in các phần tử ra môi trường console.

# **NHỮNG LƯU Ý VỀ VIỆC DUYỆT MẢNG (ARRAY) VỚI FOR CỔ ĐIỂN**

* + Khi duyệt mảng với vòng lặp for, có một số điểm cần lưu ý để đảm bảo việc duyệt mảng được chính xác và hiệu quả:
    - Sử dụng biến đếm phù hợp: Biến đếm trong vòng lặp for cần phải có kiểu dữ liệu phù hợp với kích thước của mảng và có giá trị khởi tạo phù hợp. Nếu giá trị khởi tạo không đúng, có thể dẫn đến lỗi logic hoặc tràn số.
    - Sử dụng toán tử so sánh phù hợp: Vòng lặp for cần sử dụng toán tử so sánh phù hợp để kiểm tra điều kiện dừng vòng lặp. Nếu sử dụng toán tử sai, có thể dẫn đến vòng lặp không dừng hoặc không duyệt đến hết các phần tử trong mảng.
    - Sử dụng lệnh break hoặc continue khi cần thiết: Trong một số trường hợp, bạn có thể muốn thoát khỏi vòng lặp for hoặc bỏ qua một số phần tử trong mảng. Để làm điều này, bạn có thể sử dụng lệnh break hoặc continue trong vòng lặp.
    - Cẩn thận với mảng có độ dài động: Nếu mảng có độ dài động (có thể thay đổi trong quá trình chạy), bạn cần đảm bảo rằng vòng lặp chỉ duyệt đến độ dài hiện tại của mảng, tránh truy cập vào các phần tử không hợp lệ.
    - Tối ưu vòng lặp khi cần thiết: Nếu bạn cần duyệt mảng nhiều lần hoặc với một mảng lớn, việc tối ưu vòng lặp có thể giúp tăng hiệu suất chương trình. Các kỹ thuật tối ưu có thể bao gồm sử dụng vòng lặp while hoặc for-each, sắp xếp mảng trước khi duyệt, hoặc sử dụng các thuật toán tối ưu hơn để xử lý dữ liệu.

DUYỆT ARRAY VỚI CẤU TRÚC RANGE-BASED FOR [C++ ONLY]

# **CẤU TRÚC RANGE – BASED FOR LÀ GÌ**

* + Cấu trúc range-based for loop là một tính năng được giới thiệu trong C++11 cho phép lặp lại qua các phần tử của một container một cách dễ dàng hơn.
  + Cấu trúc range-based for loop cho phép bạn duyệt qua một mảng hoặc một container bằng cách sử dụng vòng lặp for với cú pháp như sau:

**for (element\_type element : array\_or\_container) {**

**// Code xử lý element**

**}**

* + Ví dụ, để in tất cả các phần tử của một mảng số nguyên, bạn có thể sử dụng cấu trúc range-based for loop như sau:

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

**for (int x : arr) {**

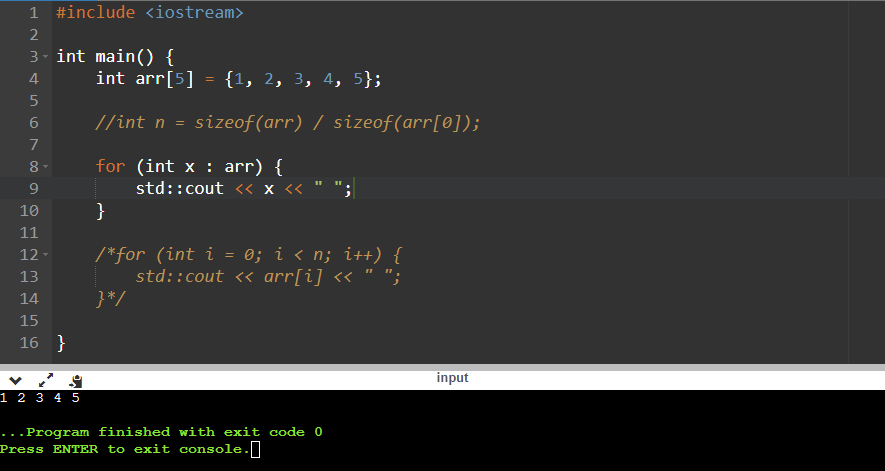
**std::cout << x << " ";**

**}**

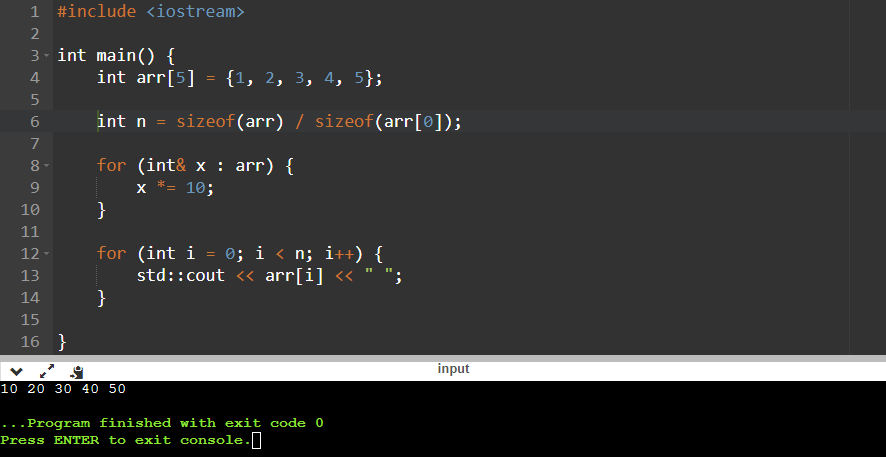
**Kết quả in ra sẽ là: "1 2 3 4 5".**

* + Cấu trúc range-based for loop cũng hoạt động với các container của STL (Standard Template Library) như vector, set, map và nhiều loại container khác.
  + Cấu trúc range – based FOR không dựa vào các chỉ mục (index) tức là các vị trí trong từng phần tử mà duyệt mảng, mà dựa vào chính mảng đó có bao nhiêu phần tử mà thực hiện duyệt mảng theo số lượng phần tự đó, nên được gọi là range – based.
  + **LƯU Ý** : Khi ta thực hiện duyệt mảng range – based FOR thì chỉ được sử dụng trên C++11 và ngôn ngữ C thuần sẽ không hỗ trợ cấu trúc duyệt mảng với range – based FOR.
  + **LƯU Ý** : Khi duyệt mảng với cấu trúc range – based FOR thì ta khai báo một biến để chứa các phần tử đó, tức là nó sẽ lặp từng phần tử và gán cho biến đó (biến đó như là một biến tạm thời) để lưu các phần tử trong mảng, do đó thì 2 biến chứa các phần tử và từng phần tử trong mảng sẽ độc lập nhau. Nếu ta muốn cập nhật các dữ liệu trong mảng đó thì ta có thể sử dụng tham chiếu & reference thì lúc đó biến lưu trữ tạm thời sẽ có sự liên kết với nhau thì lúc đó ta thay đổi biến tạm thời đó thì các phần tử trong mảng tương ứng với số lần duyệt sẽ được cập nhật giá trị.

# **VÍ DỤ VỀ DUYỆT ARRAY VỚI CẤU TRÚC RANGE-BASED FOR [C++ ONLY]**



* + Ta sử dụng vòng lặp for và khai báo biến x tạm thời trong vòng lặp for đó để lưu trữ các phần tử của mảng arr và vòng lặp sẽ phụ thuộc vào độ dài (length / size) của mảng để quyết định sẽ có bao nhiêu phần tử trong mảng đó và tiến hành duyệt mảng.
  + Lợi thế của việc sử dụng cấu trúc range – base FOR để duyệt mảng là nhìn gọn hơn, dễ nhìn hơn cũng như các giá trị khởi tạo và truyền vào ngắn hơn so với for cổ điển.
  + Tuy nhiên, duyệt mảng với cấu trúc range – based FOR sẽ không thể truy cập trực tiếp vào các phần tử trong mảng và thao tác trên chính phần tử đó, đó chính là cái hại của việc sử dụng cấu trúc duyệt nâng cao range – based FOR.
  + Việc duyệt mảng với cấu trúc này sẽ thực hiện gán từng phần tử trong mảng đó cho một biến tham thời được khai báo, việc đồng nghĩa với nó là các phần tử đó và biến int x sẽ độc lập nhau. Do đó khi ta muốn cập nhật các giá trị trong mảng bằng cấu trúc range – based FOR và sử dụng một vòng lặp for để duyệt mảng rồi in mảng ra thì phải sử dụng tham chiếu (reference) hoặc Pointer (con trỏ) trong C++.



Theo đó ta có thể thay việc bất tiện của cấu trúc range based FOR chính là không thể truy cập vào từng phần tử bằng toán tử truy cập “[]” và biến tạm thời đó cũng sẽ biến mất được cập nhật sau mỗi lần gán như vậy.

# **NHỮNG LƯU Ý KHI DUYỆT MẢNG VỚI CẤU TRÚC RANGE – BASED FOR [C++ ONLY]**

* + Khi sử dụng cấu trúc range-based for loop để duyệt một mảng trong C++, bạn cần lưu ý một số điểm sau đây:
    - Cấu trúc range-based for loop chỉ được hỗ trợ từ phiên bản C++11 trở đi.
    - Cấu trúc range-based for loop không cho phép truy cập trực tiếp vào chỉ số hoặc vị trí của các phần tử trong mảng. Nếu bạn cần truy cập chỉ số hoặc vị trí của các phần tử, bạn nên sử dụng vòng lặp for thông thường.
    - Khi sử dụng cấu trúc range-based for loop để duyệt một mảng, bạn không nên thay đổi giá trị của các phần tử trong mảng trong quá trình lặp lại, vì điều này có thể dẫn đến kết quả không chính xác hoặc lỗi thực thi. Nếu bạn cần thay đổi giá trị của các phần tử trong mảng, bạn nên sử dụng vòng lặp for thông thường.
    - Nếu bạn muốn sử dụng cấu trúc range-based for loop để duyệt qua một mảng và thực hiện một số thao tác trên các phần tử trong mảng mà không ảnh hưởng đến giá trị của các phần tử đó, bạn có thể sử dụng tham chiếu hằng (const reference) để tránh sao chép không cần thiết.
    - Nếu mảng được truyền vào cấu trúc range-based for loop không có kích thước cố định, ví dụ như một con trỏ tới mảng, bạn nên chắc chắn rằng mảng được kết thúc bằng một phần tử đặc biệt để xác định điểm kết thúc của mảng.
    - Nếu mảng được truyền vào cấu trúc range-based for loop là một mảng 2 chiều, bạn có thể sử dụng một vòng lặp range-based for lồng vào một vòng lặp range-based for để duyệt qua tất cả các phần tử của mảng 2 chiều.

Tóm lại, duyệt mảng với cấu trúc range – based FOR mang lại tiện lợi khi duyệt mảng ngắn và tiện lợi, không quá khó nhiều cũng như khó hiểu, nhưng song theo việc đó thì duyệt mảng với cấu trúc này sẽ không thể truy cập vào từng chỉ mục (index) hay là vị trí trong mảng và thao tác trên chính vị trí đó, việc mà là linh hồn của mảng (array) cho nên kết luận lại cấu trúc range – based FOR sẽ giúp ích khi duyệt mảng với một cấu trúc ngắn, dễ hiểu và viết như thế sẽ dễ nhìn hơn.

THAO TÁC CRUD VỚI MẢNG (ARRAY)

# **CRUD (CREATE / READ / UPDATE / DELETE) LÀ GÌ**

* + CRUD là viết tắt của Create (Tạo), Read (Đọc), Update (Cập nhật) và Delete (Xóa). Các thao tác CRUD thường được sử dụng trong các ứng dụng web hoặc cơ sở dữ liệu để thao tác với dữ liệu.
  + Tuy nhiên, trong C++, mảng (array) là một kiểu dữ liệu đơn giản, không có sẵn các thao tác CRUD như các cơ sở dữ liệu hay ứng dụng web. Để thực hiện các thao tác CRUD trên mảng trong C++, bạn cần tự viết các hàm tương ứng.
  + Ví dụ, để tạo (Create) một mảng trong C++, bạn có thể sử dụng cú pháp như sau:

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

* + Để đọc (Read) giá trị của một phần tử trong mảng, bạn có thể sử dụng chỉ số (index) của phần tử đó. Ví dụ:

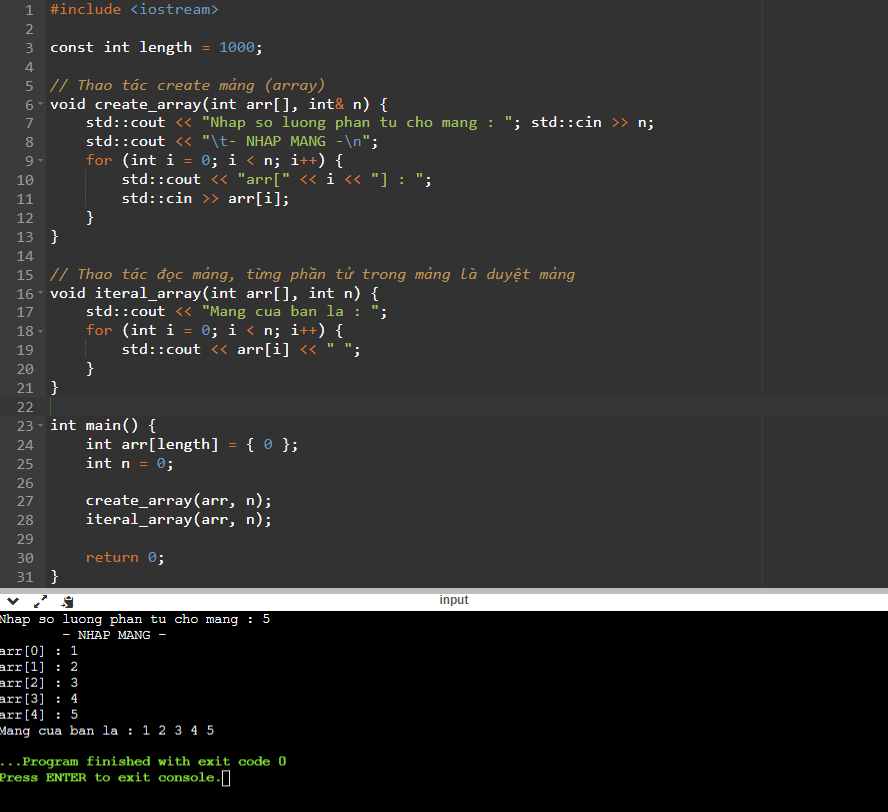
**int x = myArray[2]; // đọc giá trị của phần tử thứ 3 trong mảng**

* + Để cập nhật (Update) giá trị của một phần tử trong mảng, bạn cũng có thể sử dụng chỉ số của phần tử đó. Ví dụ:

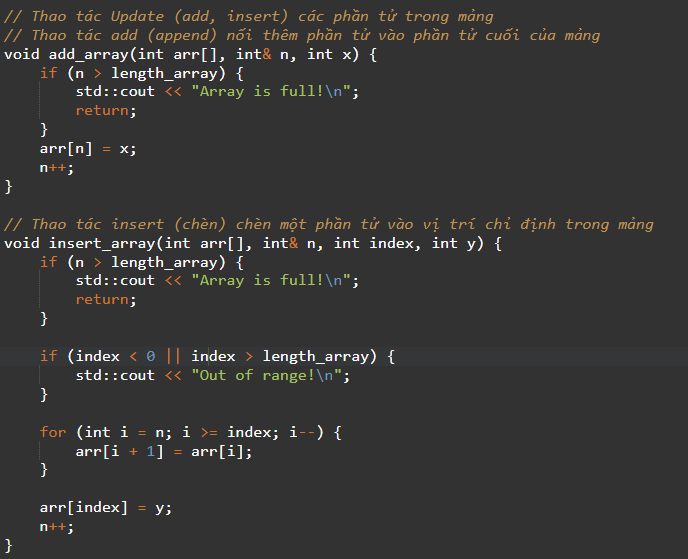
**myArray[2] = 10; // cập nhật giá trị của phần tử thứ 3 trong mảng thành 10**

* + Để xóa (Delete) một phần tử trong mảng, bạn có thể dịch chuyển các phần tử phía sau của phần tử cần xóa và giảm kích thước của mảng đi 1. Tuy nhiên, trong C++, kích thước của mảng là không thay đổi sau khi được khai báo, vì vậy việc xóa phần tử khá phức tạp.
  + CRUD là các thao tác cơ bản với các cấu trúc dữ liệu (data structure). C++ hỗ trợ nhiều kiểu cấu trúc dữ liệu dạng tập hợp như : Mảng (array) và các dữ liệu nâng cao như Vector, List, Map của thư viện STL.
  + **CRUD viết tắt của:** Create 🡺 Read 🡺 Update 🡺 Delete. Tương ứng với Việc tạo mảng 🡺 đọc mảng (duyệt mảng) 🡺 Update (thêm phần tử vào mảng) 🡺 Xoá phần tử chỉ định trong mảng. Đây được xem là các thao tác chính và cũng như cơ bản khi ta làm việc với cấu trúc dữ liệu dạng mảng (array).
  + **THAO TÁC DUYỆT MẢNG** : kiểm tra lại các phần tử có trong mảng, như trong quân đội (duyệt binh) chỉ huy sẽ kiểm tra lại các thành viên trong đoàn quân đội tương tư như thao tác duyệt mảng.

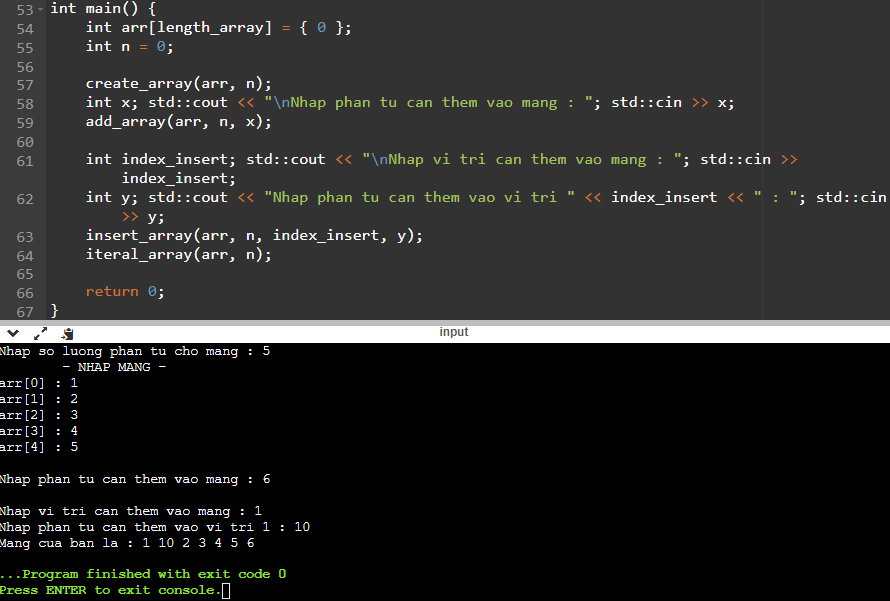
# **VÍ DỤ VỀ THAO TÁC CRUD TRONG MẢNG (ARRAY)**



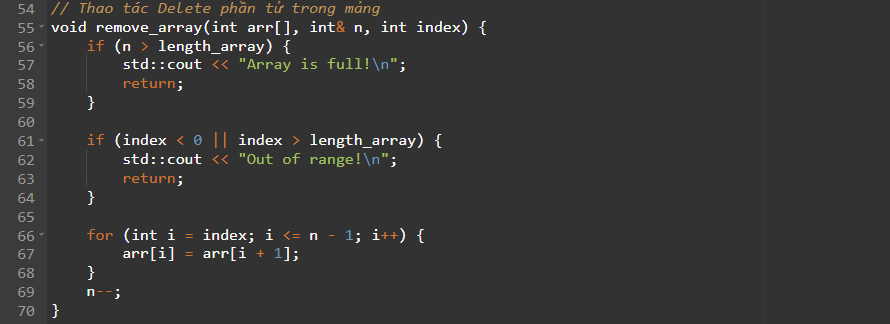
* + **THAO TÁC TẠO MẢNG (CREATE):**
    - Tạo một hàm void để thực hiện tạo mảng, các đối số truyền vào thì bao gồm một mảng và số lượng phần tử trong mảng và số lượng phần tử trong mảng “n” được truyền dưới dạng tham chiếu, tức là n trong hàm (function) đó thay đổi thì số lượng n ở hàm main() cũng sẽ thay đổi theo.
    - Sử dụng vòng lặp để duyệt hết mảng đó và gán từng giá trị khi người dùng nhập vào cho mảng bằng for cổ điển.
  + **THAO TÁC ĐỌC MẢNG (READ):**
    - Tạo hàm void và các tham số truyền vào gồm mảng và n phần tử của mảng.
    - Duyệt hết mảng đó từ phần tử 0 tới n – 1 và in từng phần tử ra bằng vòng lặp for cổ điển và toán tử truy cập “[]”.
  + Ở hàm main(), ta tạo một mảng với SZIE của mảng là 1000 được định nghĩa là một biến hằng const = 1000; để dọn sẵn các phần tử cho mảng sau này.
  + Ta khai báo biến n số lượng phần tử của mảng = 0;
  + Ta gọi 2 hàm và truyền các đối số vào cho hàm gồm mảng arr[] và số lượng phần tử n.



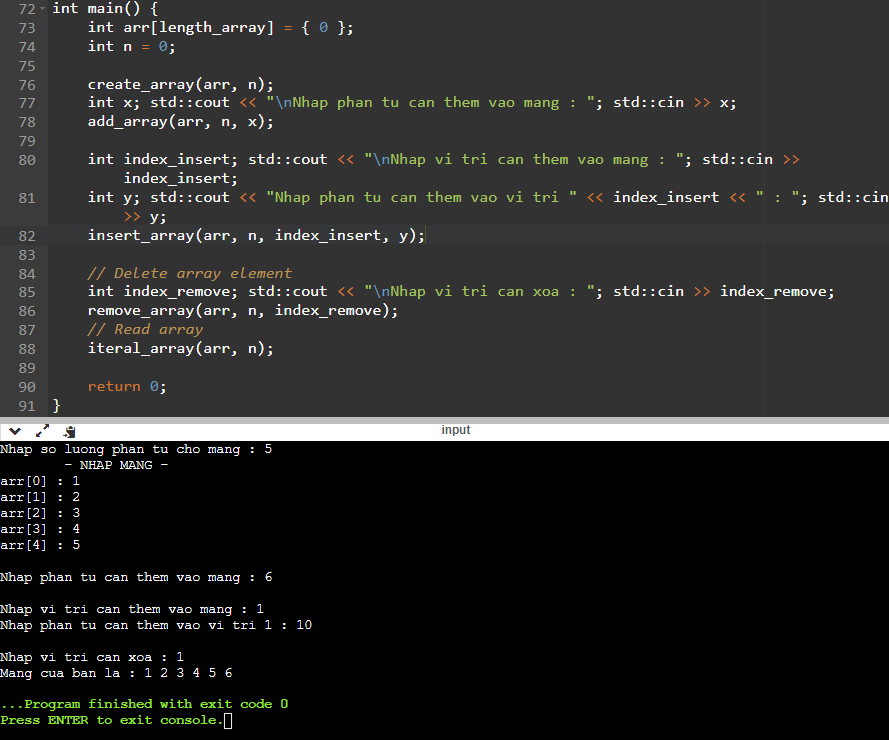
* + **VỚI THAO TÁC UPDATE (ADD, INSERT) PHẦN TỬ VÀO MẢNG:**
    - Tạo hàm void với chức năng add truyền các đối số là arr[] (lưu ý arr[] ở trong hàm được thay đổi giá trị thì biến arr[] của hàm main() cũng sẽ thay đổi theo, vì mảng (array) được truyền vào trong hàm là một Pointer, là cùng trỏ tới cùng một vùng dữ liệu với arr[] trong hàm main()) và truyền tham chiếu (pass by reference) n và truyền thêm int x để người dùng nhập một phần tử để thêm vào cuối mảng.
    - Sử dụng cấu trúc rẽ nhánh kiểm tra xem mảng đã full hay chưa.
    - Thực hiện gán x (người dùng nhập) vào phần tử cuối cùng của mảng arr[n].
    - Và n++ 🡺 Khi mảng tăng thêm một giá trị thì số lượng phần tử cũng tăng thêm 1 cho nên n++ và lưu ý rằng mảng sẽ chạy từ 0 tới n – 1.
  + **THAO TÁC INSERT TRONG MẢNG (ARRAY):**
    - Tạo hàm void và các đối số truyền vào bao gồm : arr[], &n, index, y.
    - Sử dụng 2 cấu trúc rẽ nhánh để kiểm tra các điều kiện của mảng.
    - Ý tưởng để chèn một phần tử vào vị trí được chọn trong mảng : Ta sẽ dịch các phần tử từ phần tử cuối cùng và chạy tới phần tử index (được người dùng nhập) mỗi lần dịch chuyển như vậy thì giảm i--. Tức là dịch chuyển các phần tử tính từ vị trí index (chỉ mục) về sau 1 vị trí thì theo đó ta sẽ dư ra vị trí được chỉ định để thêm một phần tử y vào do người dùng nhập.
    - Thực hiện gán arr[index] = y; n++;



* + Khai báo và gọi hàm để thực hiện Update các phần tử trong mảng (Update).



* + **THAO TÁC XOÁ PHẦN TỬ TRONG MẢNG (DELETE):**
    - Tạo hàm để thực hiện xoá mảng với tham số truyền vào tương tư như vậy.
    - Sử dụng cấu trúc rẽ nhánh để kiểm tra thông tin mảng.
    - Sử dụng vòng lặp for duyệt từ vị trí người dùng nhập và chạy tới n – 1 (phần tử cuối cùng) trong mảng và i++.
    - Mỗi lần thực hiện như vậy ta chỉ cần gán đè các phần tử sau cho phần tử trước và chạy tới phần tử cuối cùng và gán phần tử rác cho phần tử cuối, sau đó lấy n – 1 thì phần tử rác cuối đó sẽ biến mất (cụ thể trong trường hợp này thì phần tử rác đó sẽ là 0) và tuỳ vào chương trình dịch.
    - Với thao tác xoá phần tử mảng này đòi hỏi phải 1 xíu tư duy để thực hiện duyệt mảng từ vị trí nào và cách làm như thế nào. Cách đơn giản nhất là cách gán đè phần tử từ index tới n – 1 và thực hiện xong thì thực hiện n--;
    - Ví dụ : {1, 2, 3, 4, 5} 🡺 Nhập index = 1 🡺 {1, 3, 3, 4, 5} 🡺 {1, 3, 4, 4, 5} 🡺 {1, 3, 4, 5, 5} 🡺 {1, 3, 4, 5, 0} 🡺 n-- 🡺 {1, 3, 4, 5} 🡺 n = 4 🡺 Finish FOR.



* + Gọi hàm xoá phần tử trong mảng và truyền tham số vào và thực hiện duyệt mảng để xem kết quả.

**Trong C++, mảng (array) là một kiểu dữ liệu đơn giản và không có sẵn các thao tác CRUD như các cơ sở dữ liệu hay ứng dụng web. Tuy nhiên, bạn có thể tạo (Create) một mảng bằng cách sử dụng cú pháp khai báo mảng và khởi tạo giá trị cho các phần tử trong mảng.**

**Để đọc (Read) giá trị của một phần tử trong mảng, bạn có thể sử dụng chỉ số (index) của phần tử đó. Để cập nhật (Update) giá trị của một phần tử trong mảng, bạn cũng có thể sử dụng chỉ số của phần tử đó. Tuy nhiên, việc xóa (Delete) một phần tử trong mảng khá phức tạp, vì kích thước của mảng là không thay đổi sau khi được khai báo. Nếu muốn xóa phần tử, bạn cần dịch chuyển các phần tử phía sau của phần tử cần xóa và giảm kích thước của mảng đi 1. Tóm lại, trong C++, việc thực hiện các thao tác CRUD trên mảng đòi hỏi phải viết code tự tạo các hàm tương ứng.**