

**Bài 1: Chương trình và Lập trình.**

**Cài đặt môi trường lập trình Code::Blocks.**

**Cấu trúc một chương trình C++**

**I. Khái niệm về chương trình và lập trình**

**1. Ngôn ngữ lập trình và chương trình dịch**

Máy tính điện tử hoạt động dựa trên sự ra lệnh của con người, tuy nhiên nó không hiểu được ngôn ngữ thông thường của con người sử dụng. Để diễn tả các thuật toán cho máy tính hiểu được, người ta sử dụng một tập các lệnh được chuẩn hóa theo một hệ thống quy tắc riêng, được gọi là ***ngôn ngữ lập trình***.

Có ba loại ngôn ngữ lập trình:

* ***Ngôn ngữ máy (Machine language):*** Các lệnh được mã hóa bằng các kí hiệu 0 - 1, là ngôn ngữ mà máy tính sử dụng. Chương trình được viết bằng ngôn ngữ máy có thể được nạp vào bộ nhớ và sử dụng luôn.
* ***Hợp ngữ (Assembly language):*** Còn gọi là ngôn ngữ lập trình bậc thấp, kết hợp giữa mã máy và các từ viết tắt tiếng Anh để diễn tả câu lệnh.
* ***Ngôn ngữ lập trình bậc cao (High-level programming language):*** Là ngôn ngữ sử dụng các từ khóa gần với ngôn ngữ tiếng Anh để mô tả thuật toán. Ngôn ngữ lập trình bậc cao rất dễ viết do gần với ngôn ngữ tự nhiên. Tùy vào mức độ trừu tượng hóa mà ngôn ngữ lập trình bậc cao được định nghĩa là cao tới cấp nào. Trong khóa học này, chúng ta sẽ tập trung nghiên cứu ngôn ngữ lập trình bậc cao trung cấp, cụ thể là ngôn ngữ C++ (tiền thân là C, nên người ta thường viết C/C++).

Để chuyển đổi chương trình được viết bằng ngôn ngữ lập trình bậc cao sang chương trình mà máy tính có thể thực thi được, người ta sử dụng một chương trình đặc biệt gọi là ***chương trình dịch***. Chương trình dịch nhận đầu vào là một chương trình viết bằng ngôn ngữ lập trình bậc cao, thực hiện chuyển đổi sang ngôn ngữ máy. Có hai loại chương trình dịch:

* ***Chương trình thông dịch (Interpreter):*** Chương trình dịch sẽ kiểm tra tính đúng đắn của câu lệnh tiếp theo trong chương trình nguồn, sau đó chuyển đổi câu lệnh đó sang ngôn ngữ máy và thực thi nó. Như vậy, quá trình chuyển đổi được diễn ra luân phiên, chương trình dịch sẽ dịch và thực hiện từng câu lệnh một.
* ***Chương trình biên dịch (Compiler):*** Chương trình dịch sẽ duyệt, phát hiện lỗi, kiểm tra tính đúng đắn của mọi câu lệnh trong chương trình nguồn, sau đó mới tiến hành dịch toàn bộ chương trình nguồn sang ngôn ngữ máy và có thể lưu trữ để sử dụng lại khi cần thiết.

Thông thường, các môi trường làm việc của các ngôn ngữ lập trình sẽ tích hợp sẵn chương trình dịch và các tính năng về soạn thảo, lưu trữ, theo dõi biến số,...Đối với C/C++, chúng ta có thể sử dụng các môi trường như: CodeBlocks, Sublime Text, Visual Studio Code,...đều được.

**2. Khái niệm về lập trình**

Mọi bài toán có thuật toán đều có thể giải được bằng máy tính điện tử. Các bước để giải một bài toán trên máy tính gồm có: Xác định bài toán, Xây dựng hoặc lựa chọn thuật toán khả thi, và sau cùng là lập trình.

Lập trình là việc sử dụng cấu trúc dữ liệu và tập các lệnh của ngôn ngữ lập trình cụ thể để mô tả dữ liệu và diễn đạt các thao tác của thuật toán.

**II. Giới thiệu ngôn ngữ lập trình C++**

**1. C++ là gì**

Ngôn ngữ lập trình C++, được phát triển bởi Bjarne Stroustrup vào năm 1979 là một ngôn ngữ lập trình ***bậc trung*** (kết hợp giữa các đặc điểm của ngôn ngữ lập trình bậc thấp và bậc cao), cung cấp khả năng lập trình hướng cấu trúc và hướng đối tượng. Thực ra, C++ là một bản nâng cấp của ngôn ngữ C, được phát triển dựa trên chính ngôn ngữ C nên nó vẫn mang phong cách lập trình hướng cấu trúc như C và hỗ trợ thêm phong cách lập trình hướng đối tượng. Vì thế, người ta thường gọi hai ngôn ngữ này bởi một cái tên chung là C/C++.

Bởi vì là ngôn ngữ được phát triển từ C, nên ngôn ngữ C++ vẫn có đầy đủ các câu lệnh và cú pháp của ngôn ngữ C. Trong khi lập trình C++, nhiều trường hợp chúng ta vẫn sử dụng các cú pháp của C và đặc biệt, trong lập trình thi đấu thì phong cách lập trình hướng cấu trúc của C (chia chương trình ra thành nhiều hàm nhỏ - mỗi hàm phụ trách một công việc cụ thể) vẫn được giữ nguyên vì tính đơn giản và phát huy được tốc độ trong quá trình giải các bài toán. Các file được viết bằng ngôn ngữ C++ đều có phần mở rộng là .cpp.

**2. Tại sao lại lựa chọn C++ trong lập trình thi đấu**

Trong những năm trở lại đây thì xu hướng sử dụng Pascal ngày càng giảm, ngược lại số lượng người sử dụng C++ ngày càng tăng lên. Có một vài nguyên nhân chính khiến cho C++ được ưa chuộng trong lập trình thi đấu nói riêng và trong lập trình nói chung:

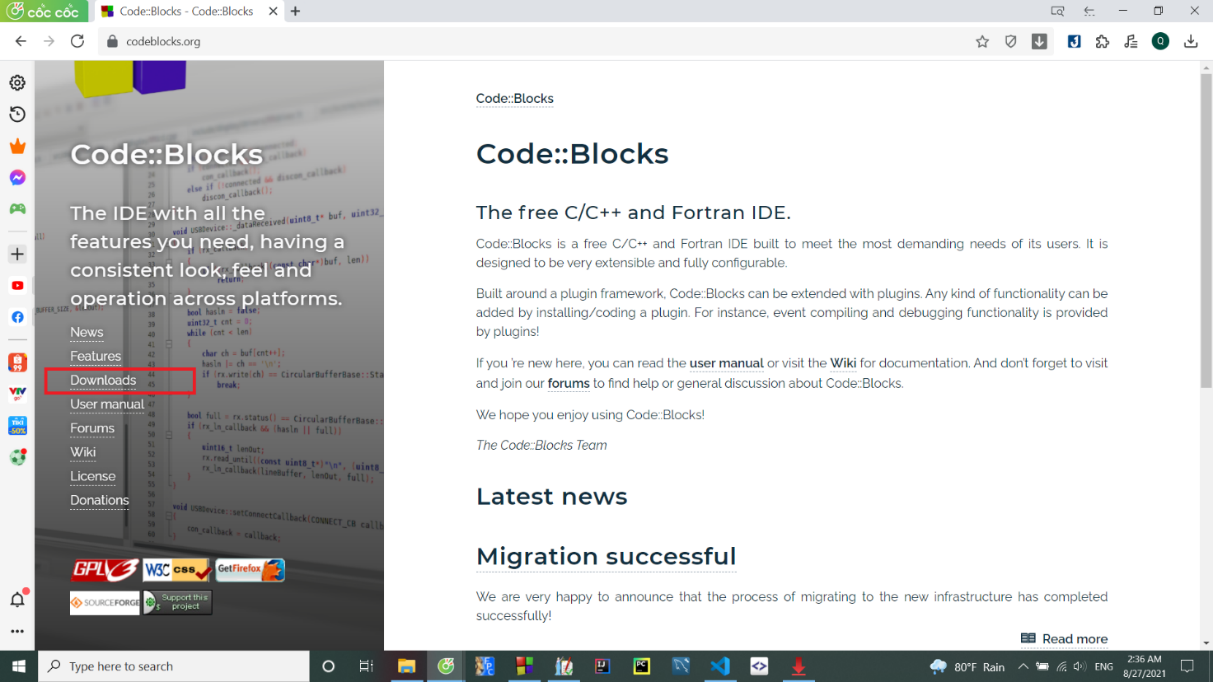
* ***Dễ tiếp cận:*** Cú pháp của C++ rất dễ để tiếp cận, gần với ngôn ngữ đời thường và rất tường minh. Học sinh - sinh viên khi mới học lập trình khi tiếp cận với C++ phần lớn sẽ cảm thấy đơn giản hơn khi sử dụng C++ làm phương tiện để tiếp nhận các kiến thức về thuật toán.
* ***Hiệu năng cực kỳ cao:*** Đây là ưu thế lớn của C++, đặc biệt là trong lập trình thi đấu khi các bài toán đều có ràng buộc về thời gian thực thi chương trình, yêu cầu có một ngôn ngữ hiệu quả về thời gian chạy. Đối chiếu với một người bạn của nó là Python, ta sẽ thấy ngay tốc độ chạy của hai ngôn ngữ này là khác biệt, mặc dù Python là một ngôn ngữ có cú pháp rất ngắn gọn nhưng tốc độ chạy lại lâu hơn C++.
* ***Thư viện hỗ trợ người dùng:*** Trong C++ có rất nhiều thư viện cung cấp các hàm dựng sẵn, hỗ trợ người dùng tối đa trong khi lập trình, đặc biệt là các thuật toán và cấu trúc dữ liệu. Việc sử dụng thành thạo các thư viện trong C++ sẽ "nối dài khả năng lập trình" của học sinh - sinh viên.

**III. Cài đặt môi trường lập trình Code::Blocks và tạo chương trình đầu tiên**

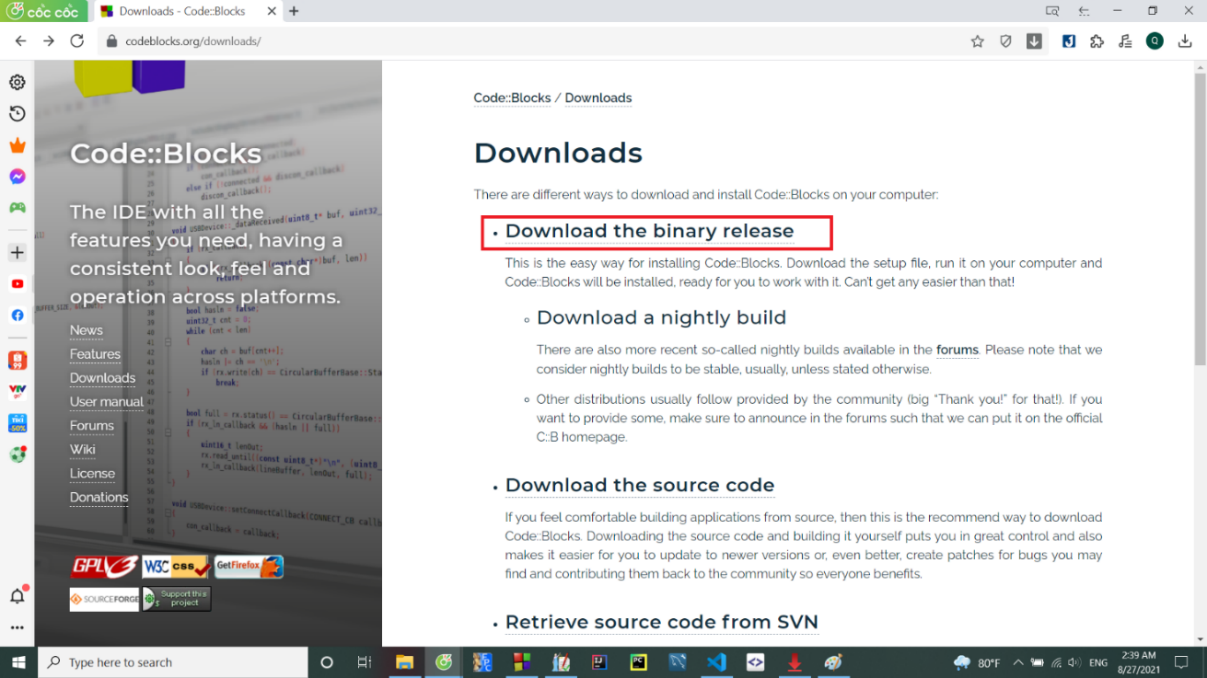
**1. Cài đặt Code::Blocks**

Để lập trình ngôn ngữ C++, chúng ta sẽ cần có một chương trình soạn thảo và chương trình biên dịch ngôn ngữ. Phần mềm Code::Blocks vừa tích hợp việc soạn thảo ngôn ngữ C++ và trình biên dịch, đồng thời có giao diện khá đẹp nên được rất nhiều học sinh - sinh viên lựa chọn sử dụng. Trong giáo án này, hướng dẫn các thao tác lập trình C++ bằng Code::Blocks.

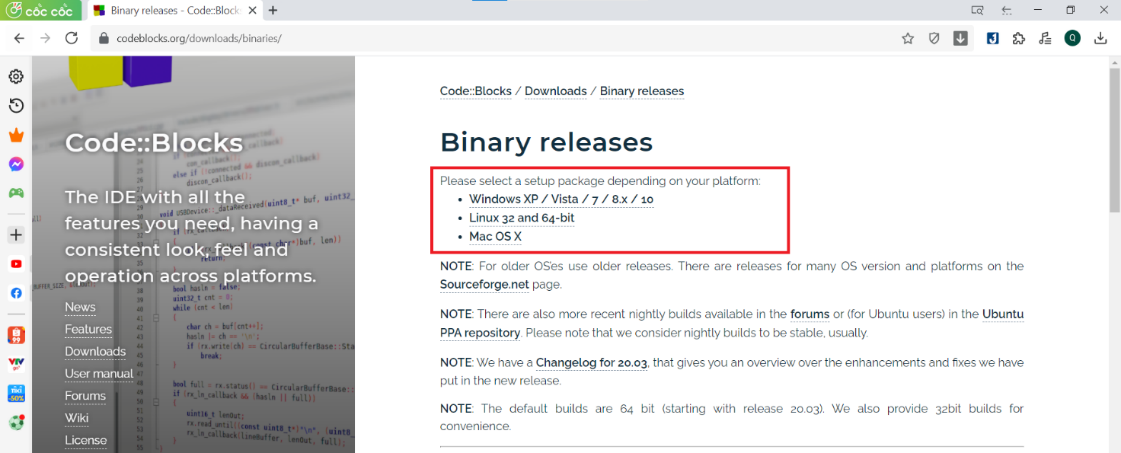
Đầu tiên, truy cập vào địa chỉ sau: <https://www.codeblocks.org/> - là trang chủ của phần mềm Code::Blocks. Giao diện của trang web sẽ hiện lên như hình bên dưới. Bấm vào mục **Dowloads** ở góc bên trái màn hình.



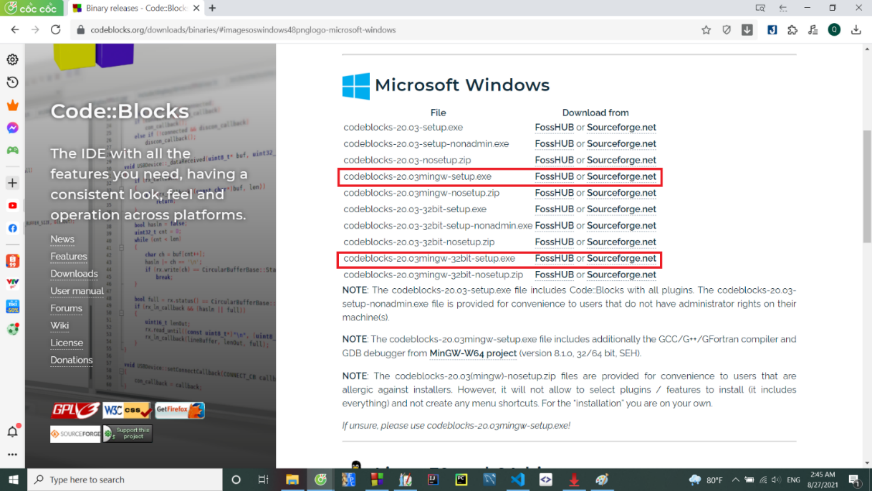
Chọn vào mục **Dowload the binary release**:



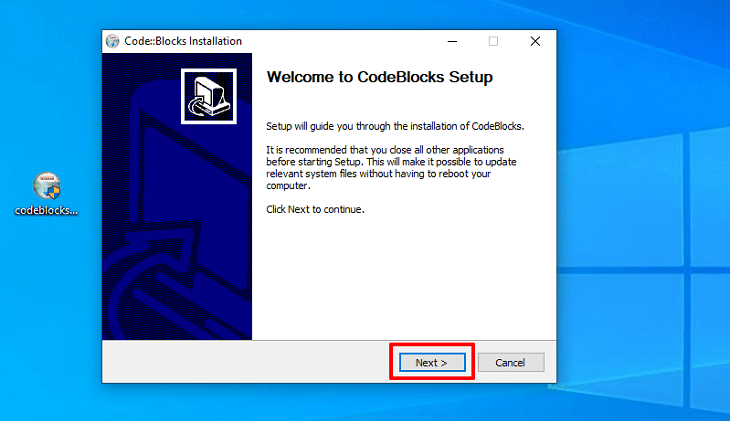
Lựa phiên bản phù hợp cho hệ điều hành máy tính của bạn. Trong giáo trình này sử dụng hệ điều hành Windows, đối với các hệ điều hành khác các bạn làm hoàn toàn tương tự:



Danh sách các bản cài đặt sẽ hiện ra sau khi bạn lựa chọn hệ điều hành. Hãy tải về file cài đặt có tích hợp trình biên dịch mingw của C++. Nếu hệ điều hành máy tính của bạn là 64 bit, lựa chọn đường link phía trên; nếu là 32 bit thì lựa chọn đường link phía dưới. Bạn có thể sử dụng một trong hai nguồn tải là **FossHub** hoặc [**Sourceforge.net**](http://sourceforge.net/) đều được. Nhấn vào link, chờ khoảng 5s là có thể tải về file cài đặt.



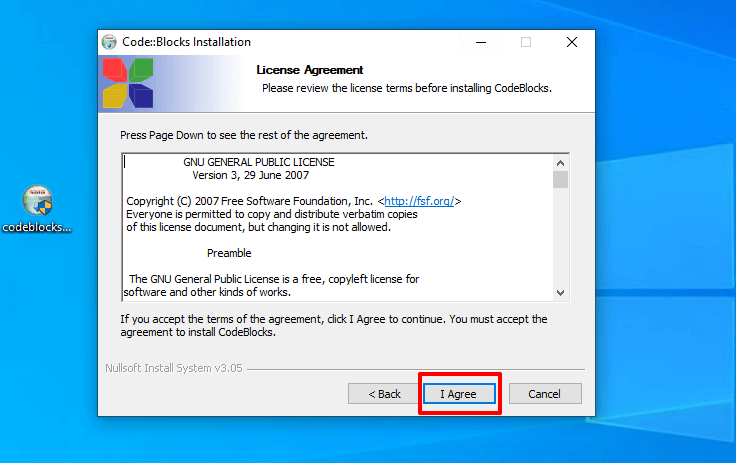
Sau khi tải về, các bạn khởi chạy file cài đặt Code::Blocks.



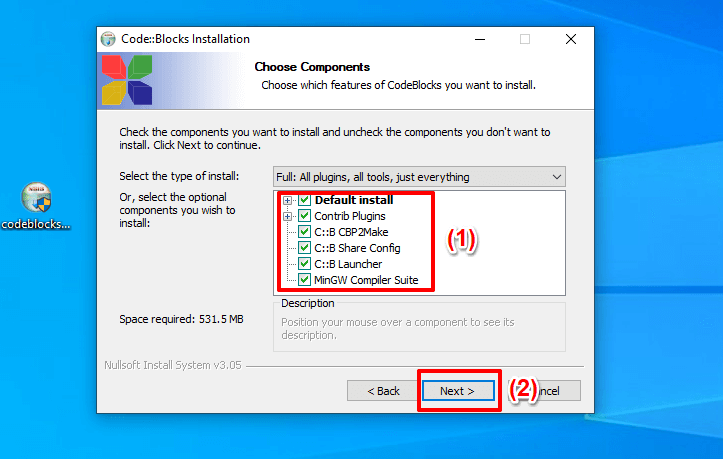
Bấm **Next** để tiếp tục.



Bước này bấm vào **I Agree** để chấp nhận điều khoản cài đặt và tiếp tục.

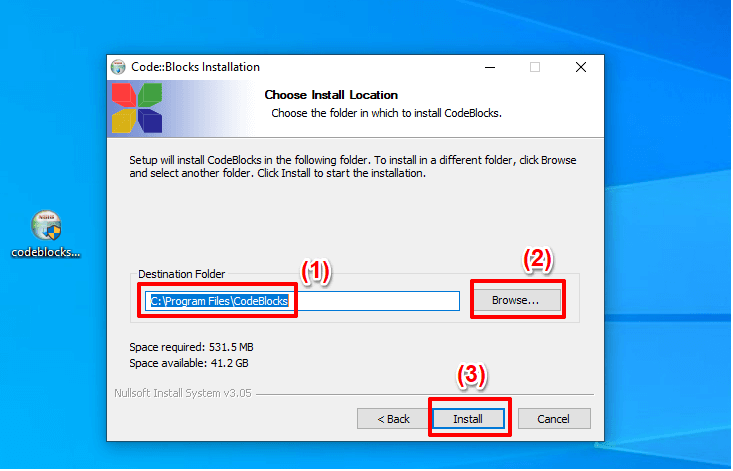


Bước này cần lựa chọn các option cài đặt. Các bạn cứ để mặc định rồi bấm vào **Next**.

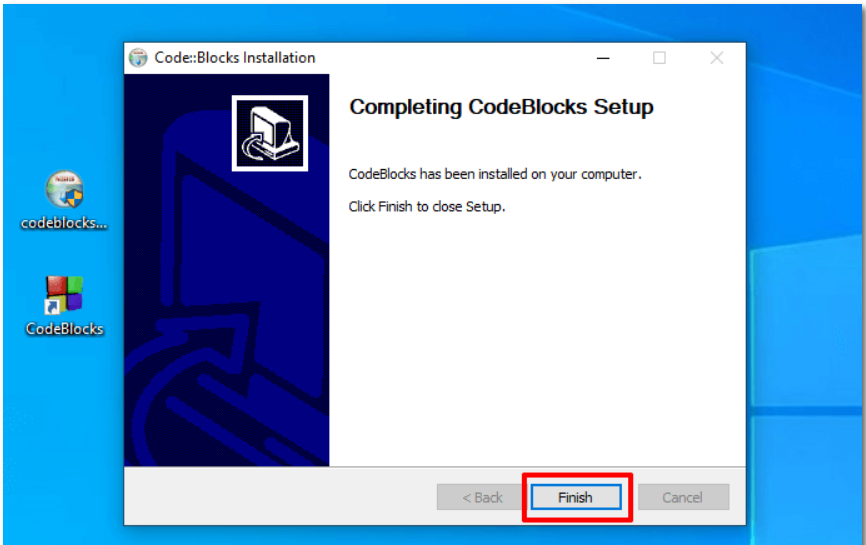


Bây giờ các bạn hãy lựa chọn thư mục mà bạn muốn cài đặt Code::Blocks vào:

* Thư mục (1) là vị trí cài đặt mặc định tại ổ C:\Program Files\CodeBlocks.
* Phần (2): Các bạn bấm vào *Browse*\* để thay đổi thư mục cài đặt.
* Cuối cùng bấm vào **Install** (3) để bắt đầu cài đặt.

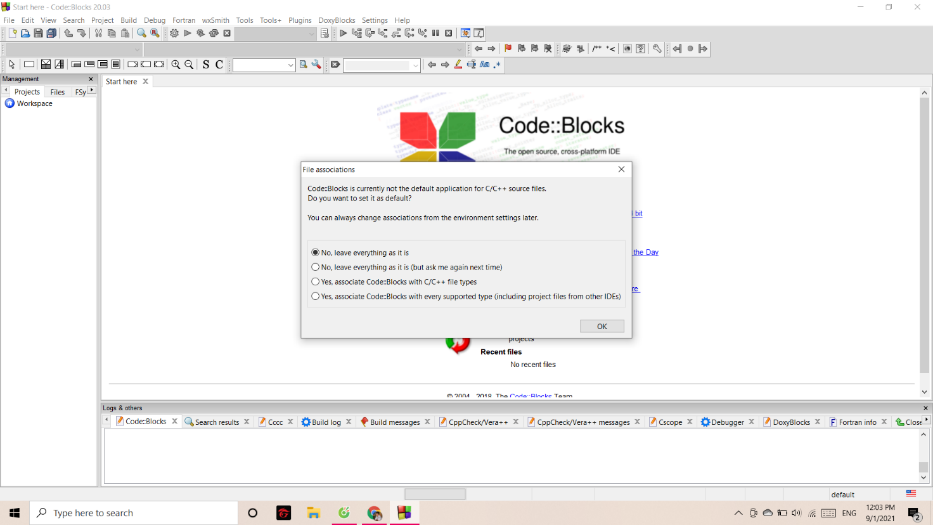


Quá trình cài đặt mất khoảng vài phút. Sau khi cài đặt xong, các bạn bấm vào **Finish** để kết thúc cài đặt.

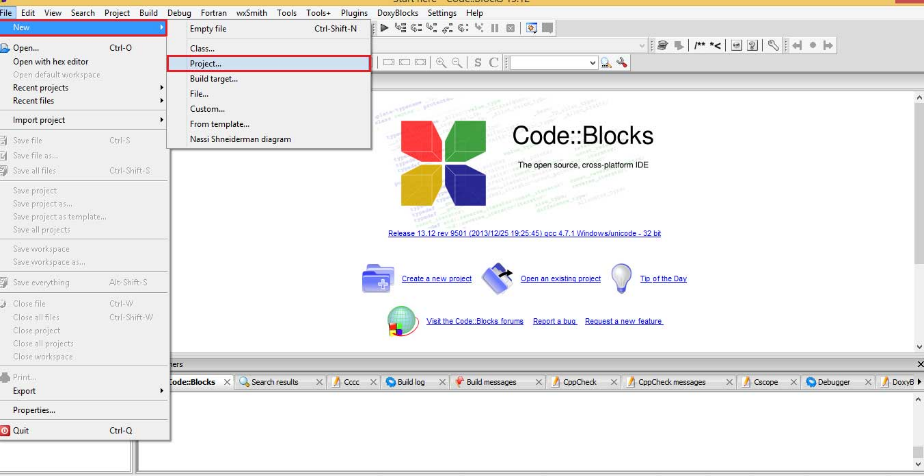


**2. Tạo project và chương trình C++ đầu tiên**

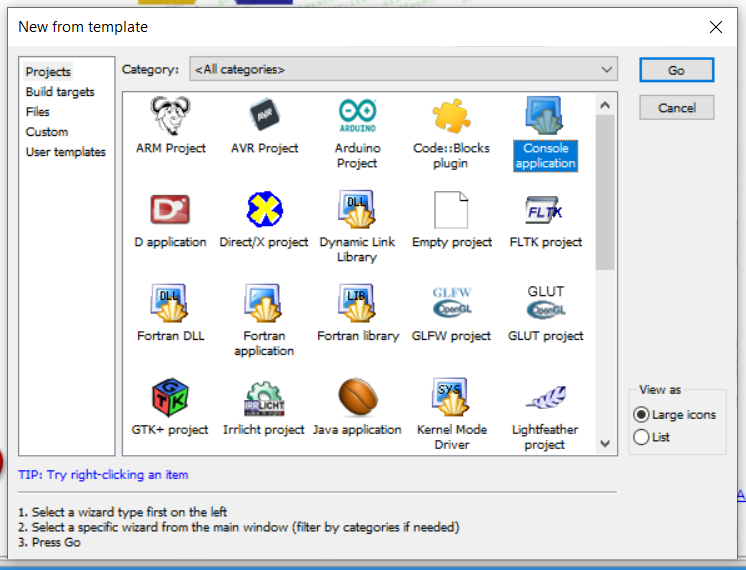
Sau khi cài đặt xong Code::Blocks, các bạn khởi chạy chương trình và sẽ thấy giao diện như hình bên dưới. Khi chạy Code::Blocks lần đầu, phần mềm sẽ hỏi các bạn có muốn đặt luôn nó làm phần mềm mặc định cho các file C++ hay không. Mình khuyên các bạn nên chọn **No** (dòng đầu tiên), vì biết đâu sau này chúng ta sẽ chuyển sang dùng một phần mềm soạn thảo C++ khác.



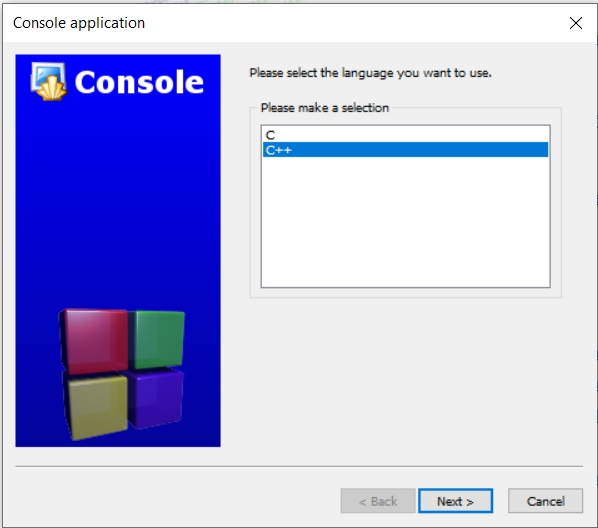
Kế đến, các bạn chọn **File** → **New** → **Project** để tạo một dự án trong CodeBlocks.



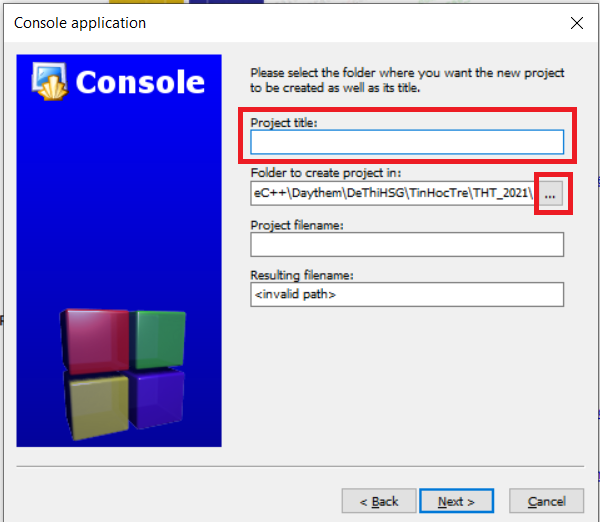
Một khung như hình bên dưới sẽ hiện ra. Các bạn cần lựa chọn loại dự án của mình. Trong Code::Blocks hỗ trợ lập trình rất nhiều dự án khác nhau. Đối với lập trình thi đấu, các bạn hãy lựa chọn **Console Application**.



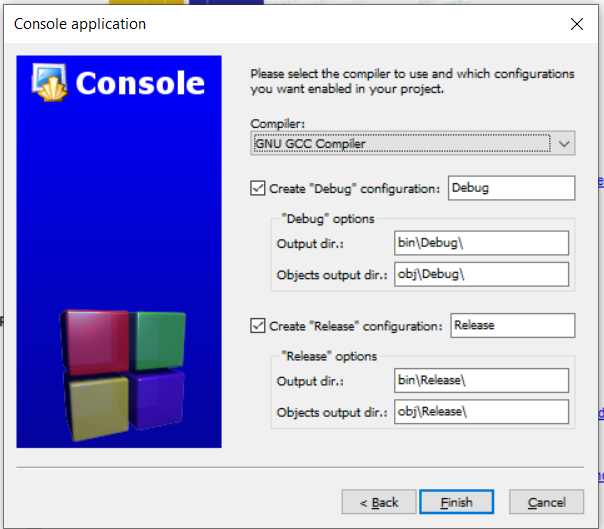
Lựa chọn ngôn ngữ là C++ rồi nhấn **Next**.



Điền tên của dự án vào mục **Project title**. Sau đó chọn nơi lưu trữ dự án bằng cách nhấn vào dấu ... ở bên cạnh phần **Folder to create project in**. Các bạn nên chọn một folder cố định làm nơi lưu trữ các dự án C++ của mình, sau đó sẽ tạo các dự án mới hoặc buổi học mới trong một folder cố định đó thôi, để tránh dữ liệu bị lưu trữ lộn xộn. Tiếp tục nhấn **Next**.

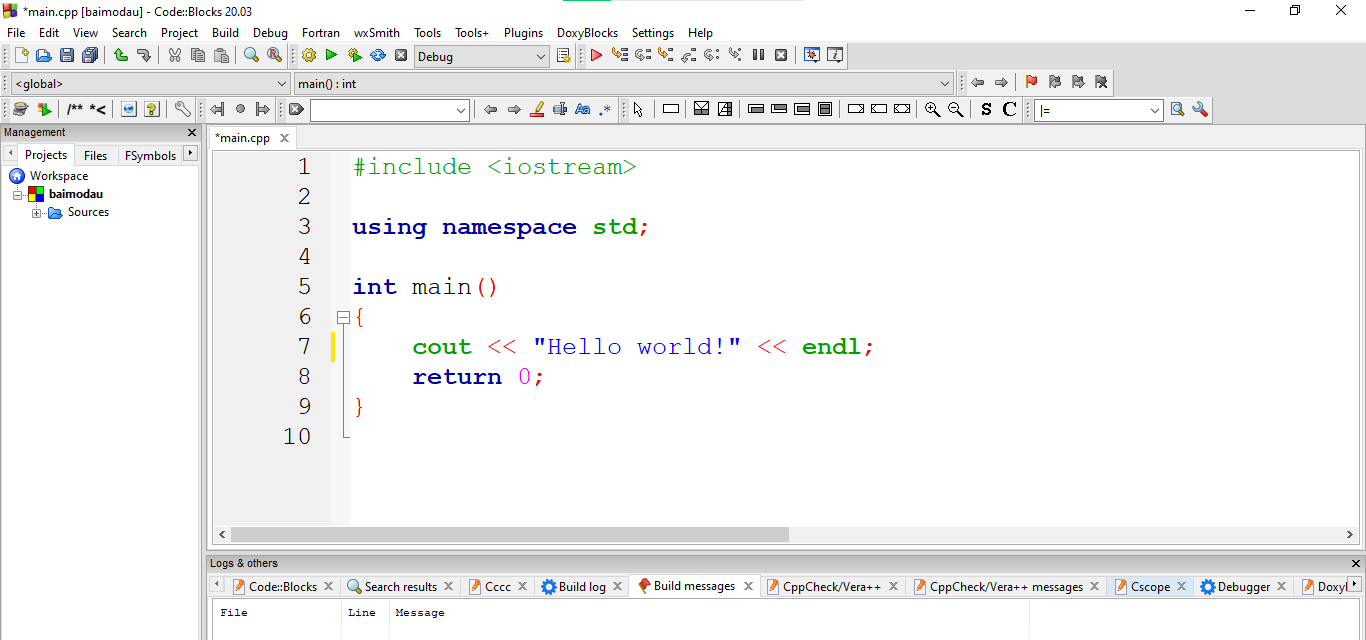


Nhấn vào finish để tiến hành tạo dự án.

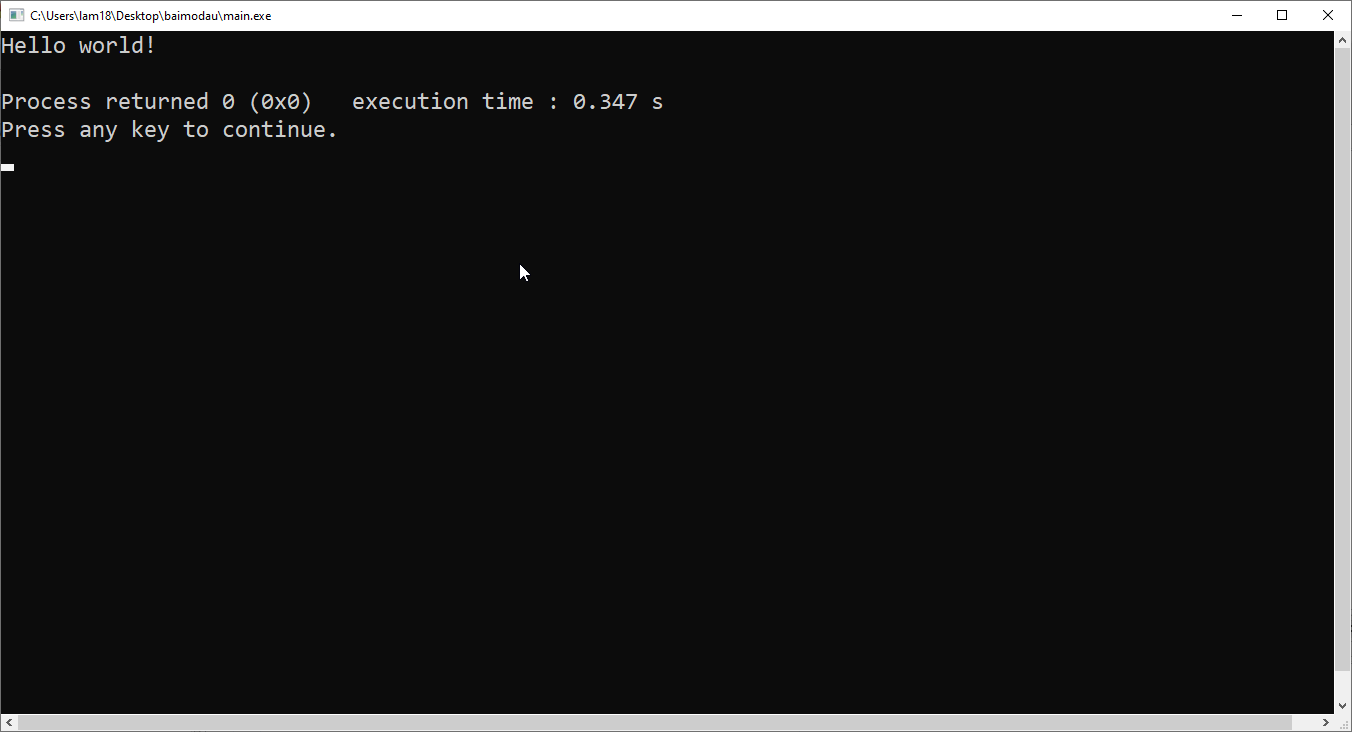


Giao diện chương trình sẽ hiện ra như hình bên dưới. Các bạn quan sát ở thanh công cụ bên trái:

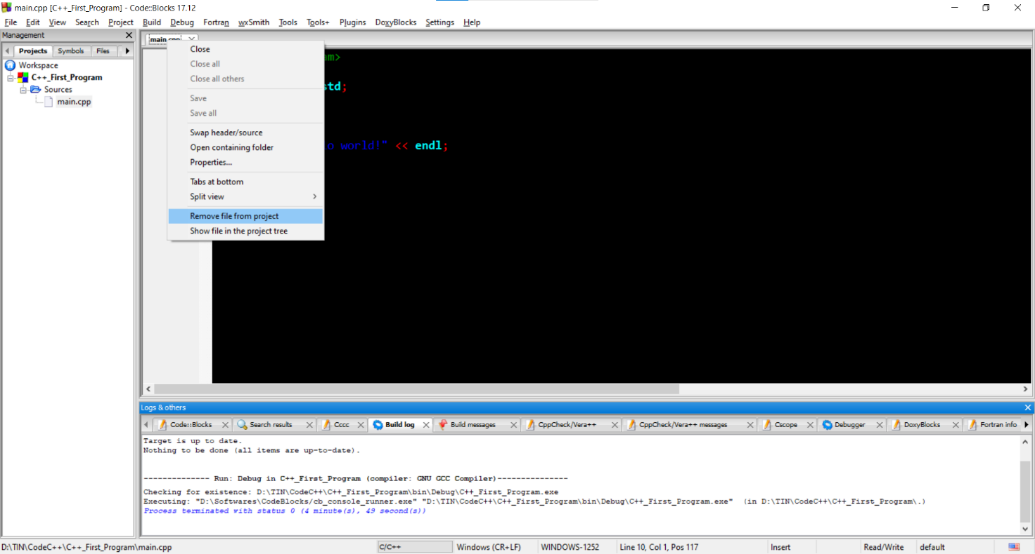
* Phần **C++\_First\_Program** chính là tên dự án mà mình vừa tạo ra.
* Mục **Sources** bên trong chứa các file chương trình của dự án. Mặc định sẽ có một filemain.cpp, các bạn bấm vào file này sẽ hiện ra một chương trình như bên dưới.



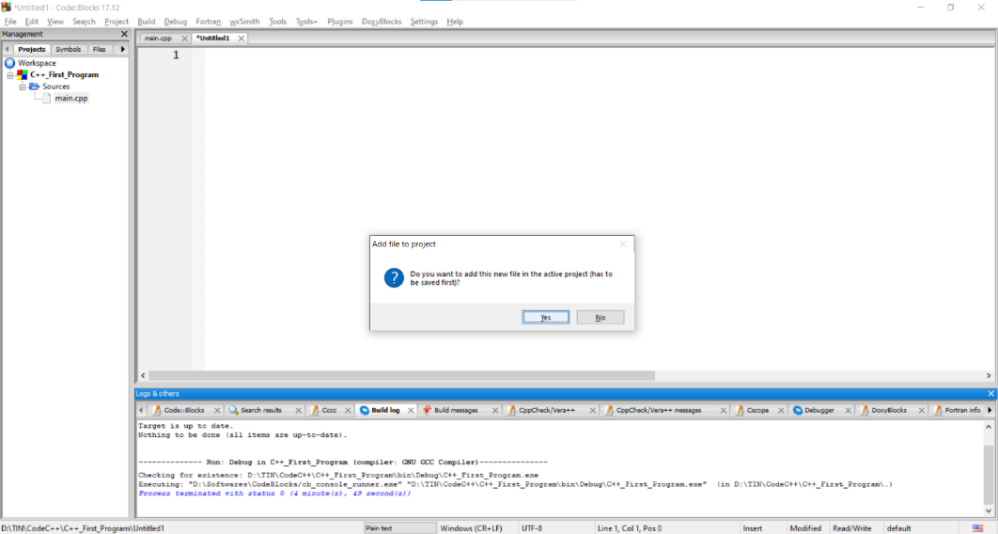
Để chạy chương trình, trước tiên các bạn nhấn tổ hợp phím **Ctrl - F9** để biên dịch chương trình, sau đó nhấn tổ hợp phím **Ctrl - F10** để chạy chương trình. Hoặc các bạn có thể chỉ nhấn phím **F9** để vừa biên dịch và chạy chương trình đồng thời. Giao diện phần chạy chương trình sẽ hiện lên như sau:



Các bạn cũng có thể loại bỏ file main.cpp này ra khỏi project và tạo ra những file chương trình mới của mình để làm những bài mới. Để làm điều đó, nhấn chuột phải vào tên file chương trình hiện tại, chọn **Remove file from project**.



Sau đó nhấn tổ hợp phím **Ctrl - Shift - N** để tạo ra một file chương trình mới. Chọn **Yes** để thêm file này vào project và nhớ lưu tên file với phần mở rộng là .cpp



**3. Vài lưu ý nhỏ khi lập trình C++ với Code::Blocks**

Để lập trình hiệu quả, khoa học với Code::Blocks:

* Tạo ra các project mới khi làm việc. Mỗi buổi học các bạn nên tạo ra một project mới, và lưu các file bài tập mình làm trong buổi học đó ở project tạo ra. Làm như vậy sẽ kiểm soát tốt bài tập của mỗi buổi và không cần tìm kiếm mỗi khi cần xem lại.
* Đừng để nhiều file chương trình trong thư mục **Sources** đôi khi sẽ gây ra những lỗi biên dịch không đáng có.
* Các bạn có thể chỉnh sửa giao diện của Code::Blocks theo ý mình. Hãy chọn mục **Settings** → **Editor** và điều chỉnh giao diện theo ý mình. Có rất nhiều thứ có thể điều chỉnh như: Font chữ, cỡ chữ, điều chỉnh màu chữ và màu cho các từ khóa của hệ thống,..Nhưng hãy để cỡ chữ to để dễ nhìn code, dễ dàng chỉnh sửa các lỗi sai.

**IV. Cấu trúc chương trình C++**

**1. Cấu trúc tổng thể của một chương trình**

Một chương trình được viết bằng ngôn ngữ lập trình bậc cao, nhìn chung sẽ gồm hai phần: ***phần khai báo*** và ***phần thân***:

* ***Phần khai báo:*** thư viện, các hằng, biến và chương trình con. Đối với ngôn ngữ C/C++, có một cải tiến rất thuận lợi đó là người dùng được quyền khai báo ở bất kỳ đâu trong chương trình, miễn là khai báo trước khi sử dụng một thứ gì đó.
* ***Phần thân:*** Bao gồm các dãy lệnh được đặt trong một cặp kí hiệu mở đầu “{“ và kết thúc “}”. Các câu lệnh phải được phân tách với nhau bằng dấu chấm phẩy.

**2. Các thành phần của một chương trình C++**

**2.1. Phần khai báo**

**Khai báo thư viện:** Một ngôn ngữ lập trình thường có sẵn một số thư viện cung cấp các chương trình đã được xây dựng sẵn. Để sử dụng chúng, cần khai báo thư viện chứa các chương trình đó. ***Cú pháp khai báo:***

#include <{Tên\_thư\_viện}>

***Ví dụ:***

#include <iostream>

Trong C++ có rất nhiều thư viện khác nhau, mỗi thư viện có tác dụng riêng. Tuy nhiên, đối với lập trình thi đấu chúng ta chỉ sử dụng một số thư viện. Một cách nhanh để khai báo tất cả các thư viện trong C++, đó là cú pháp #include <bits/stdc++.h>.

**Từ khóa define và typedef**

**Từ khóa define**

***Tác dụng:*** Sử dụng để định nghĩa một tên mới cho một kiểu dữ liệu, một câu lệnh, một hàm hoặc đơn giản là một giá trị nào đó, với mục đích giúp viết chương trình ngắn gọn, đẩy nhanh quá trình làm việc. Thông thường những khai báo bằng define sẽ được viết ngay sau khi khai báo thư viện. Kể từ sau khi khai báo, các tên mới có thể được sử dụng thay cho tên cũ, đồng thời tên cũ vẫn không bị mất đi. ***Cú pháp:***

#define {Tên\_mới} {Tên\_cũ}

***Ví dụ:***

#define infinity

Câu lệnh này định nghĩa cho tên infinity mang giá trị là , và từ sau khai báo này, tên infinity có thể sử dụng thay cho số .

**Từ khóa typedef**

* ***Tác dụng:*** Khác với define, từ khóa typedef có phạm vi sử dụng hẹp hơn. Nó chỉ được dùng để định nghĩa lại một tên mới cho những ***kiểu có sẵn*** hoặc những kiểu đã được người dùng định nghĩa. Chúng ta không thể sử dụng typedef để tạo ra một tên mới cho những giá trị tùy ý, mà phải là những từ khóa của C++.
* ***Cú pháp:***

typedef {Tên\_kiểu\_có\_sẵn} {Tên\_mới};

* ***Ví dụ:***

typedef long long LL;

**Khai báo không gian tên (namespace)**

***Định nghĩa namespace:*** Về bản chất, namespace định nghĩa một phạm vi sử dụng hàm. Giả sử trong khi lập trình, người dùng gọi ra một hàm F nào đó, nhưng có tới 22 thư viện có chung hàm tên là F (tất nhiên với chức năng khác nhau). Vậy làm sau để chương trình biết rằng người dùng cần sử dụng hàm F nào? Khái niệm namespace ra đời để phục vụ cho mục đích phân biệt ngữ cảnh sử dụng một hàm, một cái tên nào đó. C/C++ cung cấp một namespace tiêu chuẩn (std) đã được xây dựng sẵn để giúp người dùng sử dụng các hàm và lệnh trong thư viện chuẩn của C++.

***Cú pháp khai báo:***

using namespace {Tên\_namespace}

***Ví dụ:***

using namespace std;

**2.2. Phần thân**

Bao gồm các hàm của chương trình, trong đó bắt buộc có duy nhất một hàm main(), hay còn gọi là ***hàm thực thi***. Trong quá trình biên dịch, chương trình dịch sẽ đi vào dịch các câu lệnh trong hàm main() đầu tiên, vì vậy mọi hàm con của chương trình đều phải được gọi ra trong hàm main() nếu như muốn được thực thi.

Các ***câu lệnh*** trong phần thân chương trình luôn luôn kết thúc bằng dấu ;.

Trong các hàm sẽ bao gồm các ***lệnh đơn*** hoặc các ***khối lệnh***. Một ***khối lệnh*** là một cụm gồm các lệnh liên quan tới nhau, được đặt trong một cặp ngoặc {}. Thuật ngữ ***câu lệnh*** được sử dụng để chỉ chung cho câu lệnh đơn và khối lệnh.

Đôi khi chúng ta cần chú thích cho một đoạn chương trình nào đó để diễn giải cho người đọc, khi đó ta sử dụng ***comment***. Có hai dạng comment: Dạng khối - được đặt trong cặp dấu /\* và \*/, và dạng dòng đơn - bắt đầu bằng cặp dấu //. Khi biên dịch, những gì viết trong comment sẽ được trình biên dịch bỏ qua.

**3. Chương trình C++ đầu tiên**

Chương trình dưới đây là một chương trình C++ in ra dòng thông báo Hello World trên màn hình chương trình. Bạn đọc chưa cần hiểu ý nghĩa thực sự của các câu lệnh, chỉ cần đọc để biết cấu trúc một chương trình C++ trông sẽ như thế nào!

/\*

Đây là comment dạng khối.

Những gì viết trong này sẽ được bỏ qua khi biên dịch.

\*/

// Đây là comment dòng đơn.

#include <iostream> // Khai báo thư viện, iostream là thư viện nhập xuất.

#define program\_name "example" // Khai báo định nghĩa cho tên program\_name

using namespace std; // Khai báo không gian tên.

int main() // Hàm thực thi.

{

cout << "Hello World!"; // Một lệnh đơn.

return 0; // Trả ra giá trị cho hàm, sẽ học ở một bài khác.

}

Biên dịch và chạy chương trình trên sẽ cho ra kết quả như sau:

Hello World!

**Bài 2: Toán tử và Toán hạng**

**I. Toán tử và Toán hạng**

Toán tử là các biểu tượng, các kí hiệu mà chỉ cho chương trình dịch biết rằng nó cần thực hiện thao tác toán học hay logic nào. C++ cung cấp rất nhiều toán tử có sẵn, cụ thể được chia làm 6 loại:

* ***Toán tử số học***.
* ***Toán tử quan hệ***.
* ***Toán tử logic***.
* ***Toán tử gán***.
* ***Toán tử so sánh bit***.
* ***Toán tử hỗn hợp***.

**1. Toán tử số học và Toán hạng**

Toán tử số học là các toán tử thực hiện các tính toán số học, còn toán hạng đơn giản chỉ là đối tượng thực hiện tính toán số học. Kết quả của phép toán sẽ được tự động gán theo kiểu dữ liệu của các toán hạng. Giả sử ta có hai số kiểu thực a=5.0, b=10.0 (gọi là các toán hạng), các toán tử số học sẽ tính toán chúng theo bảng dưới đây:

|  |  |  |
| --- | --- | --- |
| Toán tử | Ý nghĩa | Ví dụ |
| + | Tính tổng hai toán hạng |  |
|  | Tính hiệu hai toán hạng |  |
| \* | Tính tích hai toán hạng |  |
| / | Tính thương hai toán hạng |  |
| % | Lấy phần dư của phép chia hai toán hạng |  |
| ++ | Tăng giá trị toán hạng thêm 1 đơn vị |  |
|  | Giảm giá trị toán hạng đi 1 đơn vị |  |

Lưu ý, đối với toán tử tăng ++ và toán tử giảm--, có sự khác biệt khi ta đặt chúng đằng trước hay đằng sau toán hạng:

* Đặt đằng trước: Việc tăng/giảm toán hạng sẽ được thực hiện trước câu lệnh.
* Đặt đằng sau: Việc tăng/giảm toán hạng sẽ được thực hiện sau câu lệnh.

Ví dụ dưới đây sẽ phân tích điểm khác nhau giữa dạng prefix và dạng postfix:

int main()

{

int a = 20;

cout << ++a << endl; // a tăng lên 21 trước rồi mới in ra.

cout << a++; // in ra a trước, sau đó tăng a lên 22.

cout << a; // a lúc này mới mang giá trị 22.

return 0;

}

Chạy chương trình trên sẽ cho ra kết quả:

21

21

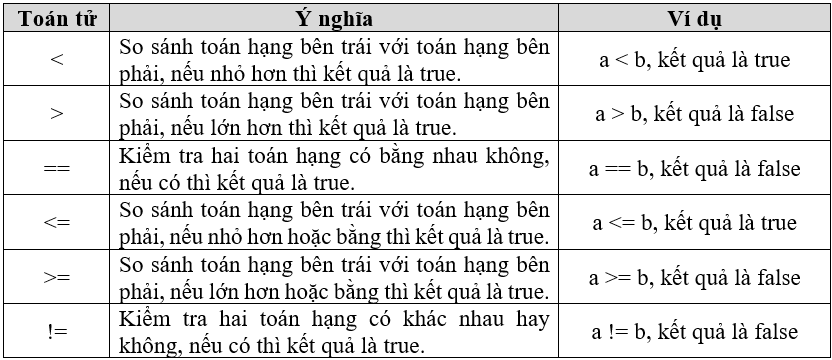
22

**2. Các loại toán tử khác**

Ngoài toán tử số học, trong C++ còn có nhiều loại toán tử khác. Tuy nhiên, do phạm vi kiến thức của bài học này nên mình sẽ chỉ giới thiệu sơ qua các loại toán tử cùng với một số ví dụ rất đơn giản. Cách sử dụng chi tiết của các loại toán tử này sẽ được đề cập đến trong những bài học khác chuyên sâu hơn.

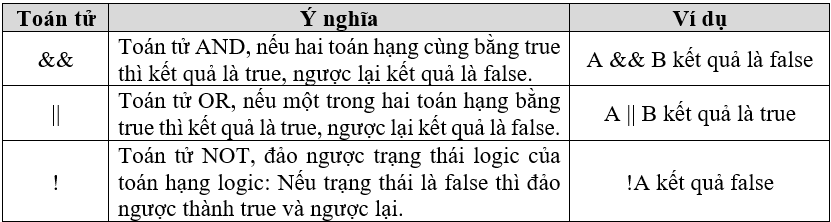
**2.1. Toán tử quan hệ**

Các loại toán tử quan hệ sẽ xác định quan hệ giữa các toán hạng, và trả ra kết quả là quan hệ đó đúng (true) hay không đúng (false). Bảng dưới đây thể hiện các quan hệ giữa hai toán hạng a=5, b=10:



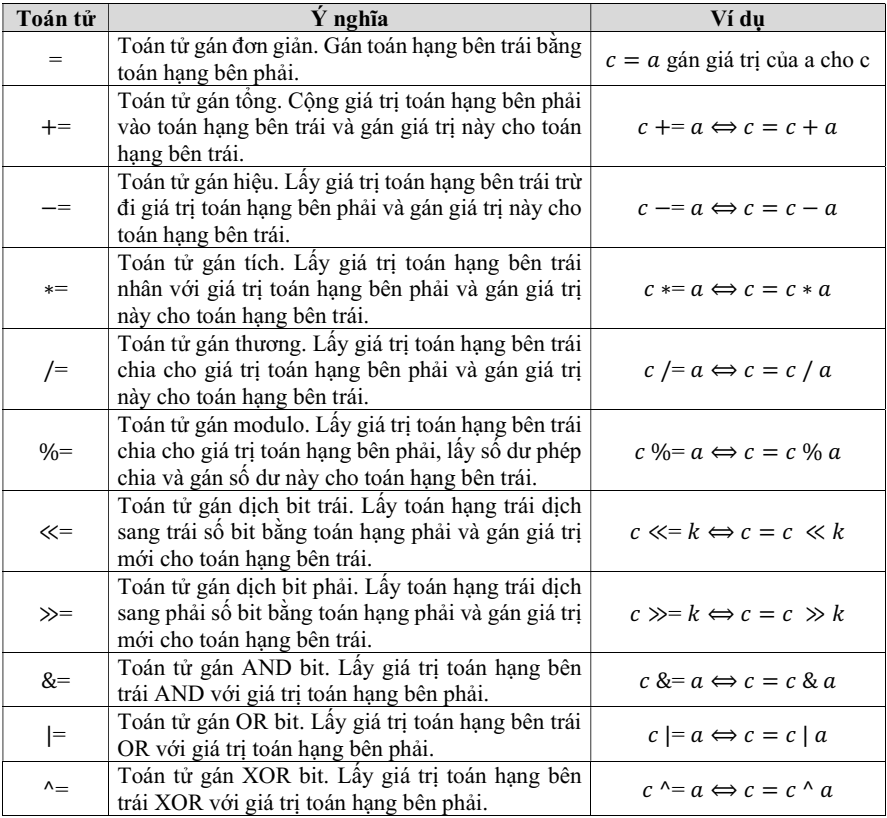
**2.2. Toán tử logic**

Các toán tử logic sẽ xác định quan hệ về mặt đúng/sai giữa các toán hạng logic. Giả sử ta có hai toán hạng logic A=true và B=false, bảng dưới đây thể hiện quan hệ logic giữa chúng:



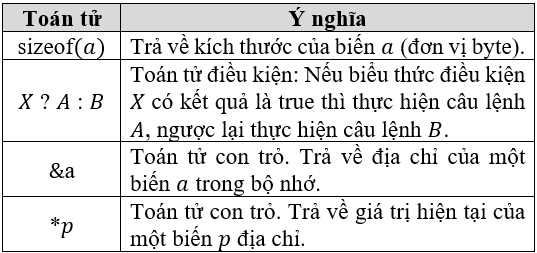
**2.3. Toán tử gán**

Toán tử gán cho phép gán một biến bằng một biểu thức nào đó. Các toán tử gán được thể hiện trong bảng dưới đây:



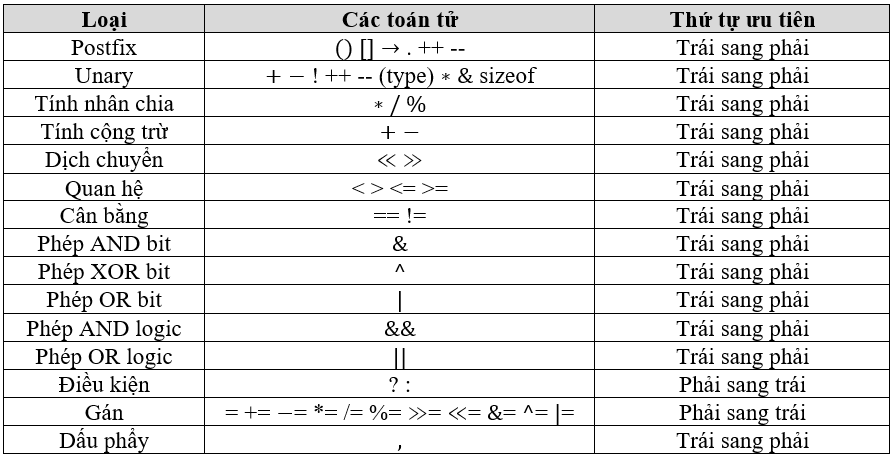
**2.4. Toán tử hỗn hợp**

Ngoài các loại toán tử kể trên, còn có một số toán tử hỗn hợp được hỗ trợ trong C++. Dưới đây là bảng liệt kê các toán tử hỗn hợp quan trọng, chi tiết về cách hoạt động của chúng sẽ được đề cập trong các bài học cụ thể:



**II. Thứ tự ưu tiên toán tử và trật tự kết hợp**

Thứ tự ưu tiên toán tử xác định cách mà biểu thức được tính toán. Những toán tử có độ ưu tiên cao hơn sẽ phải được thực hiện trước trong quá trình chạy chương trình. Bảng dưới đây liệt kê thứ tự ưu tiên của các toán tử trong cùng nhóm, và giữa các nhóm với nhau. Nhóm toán tử với quyền ưu tiên cao hơn sẽ xuất hiện ở phía trên của bảng:



**Bài 3: Kiểu dữ liệu, hằng và biến. Câu lệnh nhập - xuất cơ bản trong C++**

Trong bài viết này, chúng ta sẽ cùng nhau tìm hiểu về kiểu dữ liệu, hằng và biến - những khái niệm cơ sở của mọi ngôn ngữ lập trình.

**I. Kiểu dữ liệu**

**1. Kiểu dữ liệu nguyên thủy (Primitive Type)**

Các bài toán trong thực tế thường sẽ có dữ liệu đầu vào và kết quả ra thuộc những kiểu dữ liệu quen thuộc như số nguyên, số thực,... Việc sử dụng các kiểu dữ liệu có liên quan mật thiết đến các phép toán có thể thao tác trên dữ liệu và bộ nhớ cấp phát cho dữ liệu đó. Mỗi ngôn ngữ lập trình sẽ cung cấp cho người dùng một số kiểu dữ liệu và cho biết phạm vi lưu trữ cũng như các phép toán có thể tác động lên dữ liệu kiểu đó. Đối với ngôn ngữ C++, người dùng được cung cấp sẵn 7 kiểu dữ liệu nguyên thủy (Primitive Type) dưới đây:

|  |  |
| --- | --- |
| Kiểu dữ liệu | Từ khóa |
| Boolean | Bool |
| Ký tự | Char |
| Số nguyên | Int |
| Số thực | Float |
| Số thực dạng double | Double |
| Kiểu không có giá trị | Void |
| Kiểu wide character | Wchar\_t |

**2. Phạm vi của các kiểu dữ liệu**

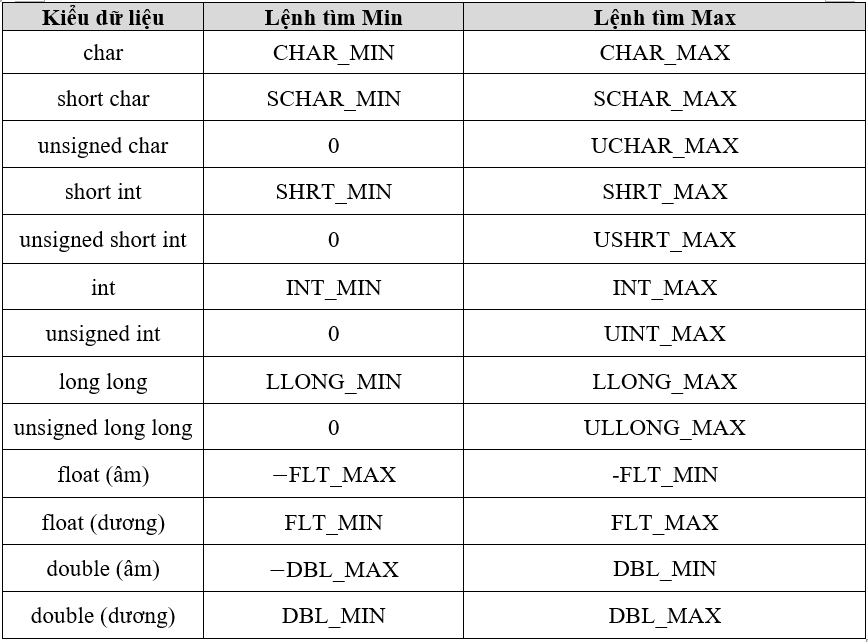
Mỗi kiểu dữ liệu đều có một phạm vi lưu trữ nhất định. Ngoài ra, người dùng có thể thêm vào một số modifier ở phía trước dữ liệu để thay đổi khoảng giá trị của kiểu. Có 4 loại modifier:

* signed: Kiểu có dấu.
* unsigned: Kiểu không dấu.
* long: Kiểu nhiều chữ số.
* short: Kiểu ít chữ số.

Bảng dưới đây thống kê phạm vi lưu trữ cũng như bộ nhớ tiêu tốn của các kiểu dữ liệu dạng số trong C++ kèm theo modifier của chúng:



Ngoài ra, C++ cung cấp một số từ khóa để đưa ra giá trị nhỏ nhất và lớn nhất lưu trữ được của một kiểu dữ liệu, thể hiện trong bảng dưới đây:



Riêng với kiểu dữ liệu char, kiểu này có thể lưu được cả số lẫn kí tự. Đối với kí tự, kiểu char lưu trữ được toàn bộ 256 kí tự thuộc bảng mã **ASCII** - là bảng kí tự và mã kí tự sử dụng trong tin học và một số lĩnh vực khác.

**II. Khai báo hằng và biến**

**1. Khai báo và sử dụng hằng số**

***Định nghĩa:*** Hằng là giá trị cố định, không thể thay đổi trong suốt chương trình sau khi đã khai báo. Khai báo hằng thường được sử dụng cho các giá trị xuất hiện nhiều lần trong chương trình.

***Cú pháp khai báo:***

const {Kiểu\_dữ\_liệu} {Tên\_hằng} = {Giá\_trị};

***Ví dụ:***

const double pi = 3.14; // Khai báo hằng.

***Sử dụng:*** Sau khi được khai báo, hằng số có thể được sử dụng trong các câu lệnh, đi kèm với các toán tử trong C++. Thông thường, khai báo hằng số được đặt ngay sau phần khai báo thư viện và không gian tên của chương trình. Dưới đây là ví dụ đưa ra diện tích của một hình tròn có bán kính bằng 2.5 sử dụng hằng số pi = 3.14:

#include <iostream>

using namespace std;

const double pi = 3.14;

int main()

{

double r = 2.5;

cout << r \* r \* pi;

return 0;

}

**2. Khai báo và sử dụng biến**

***Định nghĩa:*** Biến là giá trị sử dụng trong chương trình, có thể thay đổi tùy ý người sử dụng. Các biến trong chương trình cũng đều phải được đặt tên và khai báo cho chương trình dịch biết.

***Cú pháp khai báo:***

{Kiểu\_dữ\_liệu} {Danh\_sách\_biến} = {Giá\_trị\_ban\_đầu\_nếu\_có};

***Ví dụ:***

int x, y; // Khai báo biến, mặc định giá trị ban đầu là 0.

int index = 1; // Khai báo biến và khởi tạo giá trị ban đầu.

***Sử dụng:*** Sau khi được khai báo, biến có thể được sử dụng trong các câu lệnh, đi kèm với các toán tử trong C++. Dưới đây là ví dụ đưa ra tổng của hai số:

#include <iostream>

using namespace std;

int main()

{

int x = 5, y = 10;

cout << x + y;

return 0;

}

**3. Lưu ý khi khai báo và sử dụng hằng, biến**

Các tên biến, tên hằng, tên của các hàm,...đều được gọi chung là các ***định danh (identifier).*** Các định danh do người dùng đặt ra cần tuân theo một số quy tắc chung sau đây:

* Các định danh không được phép trùng với các ***từ khóa*** của C++. Các ***từ khóa*** là các từ tiếng Anh đã được quy định sẵn ý nghĩa và tác dụng, ví dụ như: cin, cout, enum, struct,...; và chúng sẽ được hiển thị với màu khác với các định danh của người dùng.
* Các định danh chỉ được phép chứa các kí tự chữ, số và dấu gạch dưới \_, ngoài ra không được phép chứa dấu cách, các kí tự đặc biệt và không được phép bắt đầu bằng chữ số.
* Các định danh có phân biệt giữa chữ hoa và chữ thường. Ví dụ, hai tên biến Your\_Name và your\_name là hoàn toàn khác nhau.
* Tên của các định danh nên đặt theo chuẩn mực lập trình C++ (C++ coding convention): chỉ gồm các chữ cái latin in thường và số, nếu tên biến gồm nhiều từ thì phân tách chúng bằng dấu gạch dưới \_ (quy tắc snake case). Ví dụ: school\_name, school\_id,...
* Tên của các định danh nên đặt theo ý nghĩa thực tế của nó trong quá trình sử dụng, tránh đặt tên bừa bãi như a, b, c,... vì nó sẽ gây khó khăn trong quá trình kiểm tra và tìm lỗi chương trình.

**III. Biến tự động (automatic variables)**

***Định nghĩa:*** Biến tự động là một phát triển rất tiện lợi của phiên bản C++11, nó cho phép người dùng sử dụng từ khóa auto để khai báo một biến mà không cần biết trước kiểu dữ liệu của biến đó. Từ khóa auto sẽ tự động chọn kiểu phù hợp cho biến. Điều này có thể không cần thiết đối với những kiểu dữ liệu nguyên thủy, tuy nhiên sẽ rất thuận lợi khi cần dùng các kiểu dữ liệu phức tạp sau này, hay thậm chí là tự động chọn kiểu trả về cho một hàm.

Ngoài ra bạn có thể kiểm tra kiểu trả về của biến bằng cú pháp:

typeid({Tên\_biến}).name();

***Ví dụ:***

int main()

{

auto x = 6.5F; // x tự động nhận kiểu float.

auto y = 7.1; // y tự động nhận kiểu double.

auto k = 10; // k tự động nhận kiểu int.

cout << x << endl;

cout << "Kiểu của x: " << typeid(x).name() << endl;

cout << y << endl;

cout << "Kiểu của y: " << typeid(y).name() << endl;

cout << k << endl;

cout << "Kiểu của k: " << typeid(k).name();

return 0;

}

***Kết quả:***

6.5

Kiểu của x: f // Đây là kiểu float.

7.1

Kiểu của y: d // Đây là kiểu double.

10

Kiểu của k: i // Đây là kiểu int

**III. Câu lệnh nhập - xuất cơ bản trong C++**

C++ cung cấp hai câu lệnh nhập xuất dữ liệu nằm trong thư viện <iostream> là cin và cout, sử dụng để nhận dữ liệu từ bàn phím hoặc file và trả kết quả tính ra ra màn hình hoặc file. Cách sử dụng như sau:

cin >> {Danh\_sách\_biến};

cout << {Các\_biểu\_thức\_hoặc\_biến};

Đối với câu lệnh cin, bạn chỉ được phép sử dụng nó để nhập vào giá trị cho các biến. Ví dụ: các câu lệnh cin >> a; cin >> x >> y;...là các câu lệnh hợp lệ, ngược lại cin >> 4; cin >> int x; cin >> "number";...là không hợp lệ và sẽ báo lỗi.

Đối với câu lệnh cout, bạn chỉ được phép sử dụng nó để in ra giá trị của các *biến - biểu thức có giá trị*, hoặc các *hằng số*, hoặc các *kí tự* và *chuỗi kí tự*. Khi đưa ra một kí tự thì đặt kí tự đó trong cặp dấu ‘’, còn khi đưa ra một chuỗi thì đặt chuỗi đó trong cặp dấu "". Ví dụ, cout << 5; cout << a + b; cout << "Hello World";...là các câu lệnh hợp lệ, ngược lại các câu lệnh cout << a = 5; cout << int g = a + b;...là không hợp lệ và sẽ báo lỗi.

Các lệnh cin và cout cũng có thể được áp dụng với nhiều biến, biểu thức hoặc hằng phía sau. Khi cần nhập hay xuất nhiều đối tượng liên tiếp, chỉ cần phân tách chúng bằng các dấu >> hoặc << tương ứng. Đối với lệnh cin, khi sử dụng để nhập giá trị cho nhiều biến thì phân tách chúng trong khi nhập bằng dấu cách hoặc dấu xuống dòng. Chẳng hạn: cin >> a >> b; hay cout << "Tổng các số từ 1 tới " << N << " là: " << S;.

***Ví dụ:*** Chương trình dưới đây cho phép nhập vào hai số nguyên a và b từ bàn phím sau đó đưa ra tổng của hai số đó:

int main()

{

int a, b;

cin >> a >> b;

cout << a + b;

}

**IV. Ép kiểu cho biến và hằng**

**1. Khi nào cần ép kiểu?**

Trong khi tính toán, có nhiều trường hợp kết quả của biểu thức cần tính sẽ có kiểu dữ liệu khác với các toán hạng trong biểu thức. Đơn cử một trường hợp như khi phải tính giá trị thập phân của một số hữu tỉ , với . Thông thường, viết chương trình như thế này:

int main()

{

int a, b;

cin >> a >> b;

cout << "Giá trị thập phân là: " << a / b;

}

Nếu chạy chương trình này với a = 5, b = 4, điều gì sẽ xảy ra? Có phải kết quả là 1.25 hay không? Không may là không! Kết quả của chương trình sẽ đưa ra như sau:

Giá trị thập phân là: 1

Lí giải cho điều này, đó là do a và b đều có kiểu dữ liệu số nguyên, mà kết quả của toán tử / trong C++ sẽ phụ thuộc vào kiểu của hai toán hạng. Vì thế, kết quả của phép tính a / b sẽ tự động chọn kiểu là số nguyên, mà số nguyên thì chỉ có thể lấy giá trị phần nguyên của số 1.25 thôi. Những trường hợp kiểu như vậy xuất hiện khá nhiều trong các phép tính, khi đó chúng ta cần sử dụng ép kiểu. Có 3 trường hợp cần sử dụng tới ép kiểu:

* Khi cần đưa dữ liệu về định dạng mình mong muốn, phục vụ cho tính toán.
* Khi cần khởi tạo hoặc gán một biến với giá trị của một biến (hoặc biểu thức) có kiểu khác với kiểu của biến gán (biến bên trái của toán tử gán =).
* Khi trả về kết quả cho một hàm nhưng kiểu dữ liệu của biến kết quả lại khác với kiểu dữ liệu của hàm.

**2. Cách ép kiểu cho biến và hằng**

Để ép kiểu cho biến trong C++, ta sử dụng cú pháp:

({Kiểu\_dữ\_liệu\_mới}) {Tên\_biến}

Toàn bộ cụm biến đã được ép kiểu có thể được sử dụng vào các phép tính như bình thường, nhưng sẽ có kiểu dữ liệu khác so với ban đầu khai báo (chỉ trong phép tính đó). Việc ép kiểu có thể làm thay đổi khoảng giá trị biểu diễn của một biến, ví dụ như từ số thực trở thành số nguyên, hoặc từ số nguyên sang số thực, từ kí tự sang mã của kí tự,...Điều này có khá nhiều tác dụng trong lập trình thi đấu nói riêng và lập trình nói chung, bởi trong nhiều thời điểm chúng ta sẽ cần những giá trị ở kiểu khác nhau của biến.

***Ví dụ 1:*** Đưa ra bình phương của số thực N và giá trị làm tròn của số thực N:

int main()

{

double N = 7.5;

cout << N \* N << endl; // Bình phương của N.

cout << (int)N; // Làm tròn của N.

}

***Kết quả 1:***

56.25

7

***Ví dụ 2:*** Đưa ra vị trí của kí tự x trong bảng kí tự **ASCII**:

int main()

{

char x = 'A';

cout << (int)x;

}

***Kết quả 2:***

65

***Lưu ý:***

* Trong các biểu thức, chỉ cần có một toán hạng có kiểu số thực thì kết quả của cả biểu thức sẽ có kiểu số thực.
* Khi tính toán các biểu thức, hãy cố gắng đặt kiểu dữ liệu của các toán hạng giống với kiểu dữ liệu của kết quả biểu thức đó. Ví dụ, phép nhân giữa hai số kiểu int là  và  sẽ cho ra kết quả là thuộc phạm vi của kiểu long long. Vậy hãy khai báo a và b ở kiểu long long, hoặc ép kiểu cho a và b trong phép tính. Cách làm này sẽ giúp cho dữ liệu luôn luôn chuẩn xác trong khi tính toán, tránh trường hợp tràn số gây kết quả sai.
* Đối với các hằng số, khi sử dụng chương trình sẽ tự động chọn một kiểu dữ liệu phù hợp cho hằng đó (trừ khi đã khai báo hằng bằng từ khóa const). Đây gọi là ép kiểu ngầm định của chương trình. Nếu muốn ép kiểu cụ thể cho hằng, ta cũng có thể làm như phương pháp bên trên. Ngoài ra, có thể ép kiểu cho các hằng số bằng những kí hiệu thêm vào phía sau, giả sử nếu cần ép kiểu long long cho số 5 (mặc định là int), ta viết: 5LL, hoặc ép kiểu float cho số 5.5 (mặc định là double), ta viết 5.5F,...

**V. Biểu diễn số thực trong C++**

**1. Kí hiệu khoa học của số thực**

Trong toán học, để biểu diễn các số thực rất nhỏ hoặc rất lớn, thông thường người ta sử dụng kí hiệu khoa học, đó là biểu diễn các số dựa trên lũy thừa cơ số 10. Ví dụ, khối lượng của một electron là , hay mol - một đơn vị đo lường trong hóa học có giá trị là ,...

Trong C++, ta sử dụng kí tự e để thay cho giá trị (×10) khi biểu diễn số thực. Ví dụ:

* 24327 = .
* 0.00069 = .
* ......

**2. Độ chính xác của kiểu số thực**

Như ta đã biết, có hai kiểu dữ liệu hỗ trợ biểu diễn số thực trong C++ ở dạng *dấu chấm động* là float và double. Mặc định một hằng số thực trong C++ luôn luôn có kiểu là double, còn nếu muốn sử dụng kiểu float thì ta thêm hậu tố f vào sau hằng số thực (chẳng hạn 3.5f).

Các số thực có dạng hữu hạn và dạng vô hạn. Đối với dạng vô hạn, sẽ có vô số chữ số phía sau dấu chấm thập phân, tuy nhiên bộ nhớ máy tính và kích thước kiểu dữ liệu thì lại hữu hạn. Các biến kiểu thực chỉ có thể lưu được một số lượng chữ số nhất định phía sau dấu chấm thập phân, và phần còn lại sẽ bị bỏ đi. Trong C++, câu lệnh cout mặc định sẽ có độ chính xác là 6 chữ số đối với số thực kể cả phần thực lẫn phần phân. Những số ngoài phạm vi sẽ bị cắt bỏ và làm tròn lên 1 đơn vị nếu số bị cắt sau nó lớn hơn 5, hoặc số đó có thể được chuyển sang ký hiệu khoa học trong vài trường hợp tùy vào từng trình biên dịch. Cùng xem ví dụ sau:

#include <iostream>

using namespace std;

int main()

{

double d = 9.87654321;

cout << d << endl;

d = 987.654321;

cout << d << endl;

d = 987654.321;

cout << d << endl;

d = 9876543.21;

cout << d << endl;

d = 0.0000987654321;

cout << d << endl;

d = 1.23456789;

cout << d << endl;

return 0;

}

Biên dịch và chạy chương trình trên với CodeBlocks, ta thu được kết quả:

9.87654

987.654

987654

9.87654e+006

9.87654e-005

1.23457

Đối với kiểu float, độ chính xác sẽ là 7 chữ số phần phân, còn với kiểu double, độ chính xác lên tới 16 chữ số phần phân. Do đó, khi cần sử dụng tới số thực với độ chính xác cao, kiểu double nên được ưu tiên để tránh sai số ở mức tối đa.

**3. Làm tròn số thực**

Bạn có thể làm tròn được các số thực tới số lượng chữ số mà mình mong muốn bằng cách sử dụng hàm setprecision trong thư viện <iomanip> của C++. Cú pháp như sau:

cout << fixed << setprecision({Số\_chữ\_số}) << {Hằng\_hoặc\_biến\_số\_thực};

Sử dụng hàm này, chúng ta có thể điều chỉnh số chữ số sẽ xuất hiện sau dấu chấm động của một hằng hoặc biến kiểu thực, đồng thời chương trình cũng sẽ tự làm tròn lên 1 đơn vị cho chữ số cuối cùng nếu như chữ số đầu tiên trong phần bị cắt đi lớn hơn 5. Chẳng hạn, khi viết ra số thực 9.87654321 với 3 chữ số phần phân, ta viết như sau:

#include <iostream>

#include <iomanip>

int main()

{

cout << fixed << setprecision(3) << 9.87654321;

}

Kết quả thu được sẽ là:

9.877

Tuy nhiên, như đã nói ở trên, kiểu dữ liệu số thực trong C++ chỉ cung cấp độ chính xác tối đa tới 16 chữ số, nên nếu như bạn cố viết ra các số thực với độ chính xác nhiều hơn 16 chữ số, sẽ dẫn đến kết quả bị sai số nhất định. Điều này hầu như sẽ không xảy ra trong các bài toán lập trình thi đấu.

**4. So sánh bằng nhau giữa hai số thực**

Trong C++, khi so sánh giữa hai số thực, nếu như chúng đều là các số thực hữu hạn thì không có gì đáng lưu tâm cả. Tuy nhiên, nếu so sánh những số thực vô hạn, thì sẽ không bao giờ xảy ra trường hợp chúng bằng nhau hoàn toàn được cả, vì số chữ số chính xác của kiểu dữ liệu là hữu hạn. Lấy ví dụ, số 1.0 khi chia cho 3 bằng 0.333333..., nhưng nếu đem 3 giá trị 0.333333... cộng với nhau thì chỉ thu được kết quả là 0.999999..., rõ ràng khác với 1.0. Cùng xem một ví dụ sau để hiểu kĩ hơn:

#include <iostream>

#include <iomanip>

using namespace std;

int main()

{

double d = 0.1;

cout << d << endl; // Độ chính xác mặc định 6 chữ số.

cout << std::setprecision(20); // Làm tròn 20 chữ số.

cout << d << endl;

return 0;

}

Kết quả của đoạn chương trình trên sẽ là:

0.1

0.1000000000000000555

Mặc dù cùng là số d = 0.1, nhưng khi làm tròn tới 20 chữ số thì ta lại thu được một giá trị lớn hơn 0.1. Lí do là vì, dữ liệu kiểu double chỉ có thể cho độ chính xác tới 16 chữ số, nếu người dùng muốn làm tròn ngoài khoảng đó thì sẽ xảy ra ***lỗi làm tròn dấu chấm động***. Bởi thế, khi so sánh giữa hai số thực trong C++, ta chỉ có thể sử dụng các phép toán <, >, <= và >= thôi, còn riêng với phép toán ==, thì cần sử dụng đến một khái niệm là ***độ chính xác epsilon***. Cụ thể, với hai số thực a và b, muốn biết chúng có bằng nhau hay không, người ta sẽ lựa chọn một giá trị thật nhỏ (thông thường là ), đặt nó làm giá trị epsilon. Sau đó, xét hiệu |a – b|, nếu hiệu này nhỏ hơn hoặc bằng epsilon thì sẽ coi rằng a và b bằng nhau. Ngược lại thì a và b sẽ khác nhau.

**Bài 4: Cấu trúc rẽ nhánh**

**I. Biểu thức logic trong C++**

**1. Kiểu dữ liệu bool**

Trong mọi ngôn ngữ lập trình đều tồn tại hai trạng thái đúng - sai, tương đương với hai giá trị 1 và 0. Để biểu diễn hai trạng thái này, trong C++ sử dụng kiểu dữ liệu nguyên thủy bool. Để khai báo các biến kiểu bool, ta sử dụng cú pháp:

bool {Danh\_sách\_biến};

Các biến kiểu bool còn được gọi là các ***biến logic***, chúng chỉ được phép nhận hai giá trị là true hoặc false. Tuy nhiên, như đã nói ở trên, trong C++, true tương ứng với 1, còn false tương ứng với 0. Vì thế, nếu như các biến kiểu bool được gán giá trị 0 hoặc 1 thì chương trình biên dịch vẫn sẽ hiểu được.

**2. Toán tử quan hệ và toán tử logic**

Phần này mình đã đề cập tới ở bài học đầu tiên, nhưng ở đây sẽ nhắc lại để bạn đọc dễ hình dung hơn về tác dụng của hai loại toán tử này trong biểu thức logic.

**2.1. Toán tử quan hệ**

Các loại toán tử quan hệ sẽ xác định quan hệ giữa các toán hạng, và trả ra kết quả là quan hệ đó đúng (true) hay không đúng (false). Bảng dưới đây thể hiện các quan hệ giữa hai toán hạng a=5, b=10:

|  |  |  |
| --- | --- | --- |
| Toán tử | Ý nghĩa | Ví dụ |
| < | So sánh toán hạng bên trái nếu nhỏ hơn toán hạng bên phải thì kết quả là true. | a < b, kết quả là true |
| > | So sánh toán hạng bên trái nếu lớn hơn toán hạng bên phải thì kết quả là true. | a > b, kết quả là false |
| = = | So sánh toán hạng bên trái nếu bằng toán hạng bên phải thì kết quả là true. | a = = b, kết quả là false |
| <= | So sánh toán hạng bên trái nếu nhỏ hơn hoặc bằng toán hạng bên phải thì kết quả là true. | a < b, kết quả là true |
| >= | So sánh toán hạng bên trái nếu lớn hơn hoặc bằng toán hạng bên phải thì kết quả là true. | a < b, kết quả là false |
| != | So sánh toán hạng bên trái nếu khác toán hạng bên phải thì kết quả là true. | a < b, kết quả là false |

**2.2. Toán tử logic**

Các toán tử logic sẽ xác định quan hệ về mặt đúng/sai giữa các toán hạng logic. Giả sử ta có hai toán hạng logic A=true và B=false, bảng dưới đây thể hiện quan hệ logic giữa chúng:

|  |  |  |
| --- | --- | --- |
| Toán tử | Ý nghĩa | Ví dụ |
| && | Toán tử AND, kết quả là true nếu tất cả các toán hạng cùng là true, còn lại là false. | A && B kết quả là false |
| || | Toán tử OR, kết quả là false nếu tất cả các toán hạng cùng là false, còn lại là true. | A || B kết quả là true |
| ! | Toán tử NOT, phủ định trạng thái logic của toán hạng. | !A kết quả là false |

**3. Biểu thức logic**

Biểu thức logic là những biểu thức thể hiện quan hệ giữa các biểu thức với nhau. Chúng sử dụng các toán tử quan hệ hoặc logic để kết nối các biểu thức thành phần, và luôn luôn trả về một trong hai giá trị true hoặc false. Để xây dựng một biểu thức quan hệ, ta viết theo cú pháp:

{Biểu\_thức\_A} {Phép\_toán} {Biểu\_thức\_B}

Trong đó, ***{Biểu\_thức\_A}, {Biểu\_thức\_B}*** có thể là các biến, hằng, hàm trả về giá trị hay các biểu thức toán học. Còn ***{Phép\_toán}*** là một trong những toán tử quan hệ hoặc toán tử logic thuộc tập sau: {>, <, >=, <=, ==, !=, &&, ||}. Ví dụ dưới đây là một vài biểu thức logic:

5 > 8

a != b && b != c

delta == 0

5 != 8 || 4 == 2

Những biểu thức này sẽ được tính toán theo đúng thứ tự ưu tiên của toán tử. Lấy ví dụ, trong biểu thức a != b && b != c, do toán tử != có độ ưu tiên cao hơn toán tử && nên chương trình sẽ tính toán hai biểu thức a != b và b != c trước, rồi mới lấy kết quả của chúng để thực hiện phép toán &&.

**3. Toán tử ! trong biểu thức logic**

Trong C++, toán tử quan hệ ! là một toán tử khá đặc biệt. Khác với các toán tử && hay ||, toán tử ! không dùng để kết nối các biểu thức logic, mà dùng để ***đảo ngược*** kết quả của một biểu thức logic. Cú pháp sử dụng toán tử này như sau:

!({Biểu\_thức\_logic})

Khi đó, nếu ***{Biểu\_thức\_logic}*** có giá trị là true thì cả biểu thức sẽ có kết quả ngược lại là false, tương tự với trường hợp ***{Biểu\_thức\_logic}*** có giá trị false. Chẳng hạn, biểu thức 5 > 4 có giá trị là true, nhưng biểu thức !(5 > 4) sẽ có giá trị là false.

**II. Mệnh đề "Nếu...thì" và "Nếu không...thì"**

Trong cuộc sống nói chung và trong Tin học nói riêng, có rất nhiều tình huống mà một công việc chỉ được thực hiện nếu như có một hoặc một số điều kiện được thỏa mãn. Lấy ví dụ, mẹ bạn Sơn nói với bạn Sơn rằng: "Nếu cuối tuần mẹ rảnh thì cả nhà sẽ đi chơi". Điều kiện cụ thể trong câu nói này là "nếu mẹ rảnh", và nếu điều kiện này thỏa mãn thì công việc sẽ được thực hiện là "cả nhà sẽ đi chơi". Cách diễn đạt như vậy là cách diễn đạt ở dạng thiếu:

{Nếu...thì...}

Nếu như mẹ bạn Sơn thay đổi câu nói thành: "Nếu cuối tuần mẹ rảnh thì cả nhà sẽ đi chơi, còn nếu mẹ không rảnh thì chúng ta sẽ ở nhà", thì cách diễn đạt trở thành dạng đủ:

{Nếu...thì..., Nếu không thì... (Ngược lại thì...)}

Trong mọi ngôn ngữ lập trình đều cung cấp cấu trúc để diễn tả các mệnh đề có dạng như trên, gọi là ***Cấu trúc rẽ nhánh***. Trong C++, có hai loại cấu trúc rẽ nhánh là if...else... và switch...case... Đây là một trong những cấu trúc căn bản của mọi ngôn ngữ lập trình, có tác dụng điều khiển các khối lệnh trong chương trình và tạo lập nên các thuật toán.

**III. Câu lệnh if và if...else...**

C++ sử dụng câu lệnh if và if...else để diễn tả hai mệnh đề điều kiện thiếu và đủ. Cú pháp của hai dạng như sau:

* Dạng thiếu:

if ({Biểu\_thức\_logic})

{Câu\_lệnh};

* Dạng đủ:

if ({Biểu\_thức\_logic})

{Câu\_lệnh\_1};

else

{Câu\_lệnh\_2};

Vế điều kiện trong các mệnh đề {Nếu...thì} hoặc {Nếu không...thì} chính là các ***{Biểu\_thức\_điều\_kiện}*** trong lập trình. Chẳng hạn, mệnh đề điều kiện **a chia hết cho 5** có thể viết thành biểu thức điều kiện a % 5 = = 0 trong C++.

Trong trường hợp có nhiều câu lệnh tạo thành một khối lệnh thì ta sẽ đặt tất cả các câu lệnh đó trong cặp dấu {} theo cú pháp dưới đây:

if ({Biểu\_thức\_logic})

{

{Khối\_lệnh\_1};

}

else

{

{Khối\_lệnh\_2};

}

***Các bước thực thi:***

* Ở dạng thiếu, ***{Biểu\_thức\_logic}*** sẽ được kiểm tra, nếu như điều kiện đó đúng thì ***{Câu\_lệnh}*** sẽ được thực hiện, ngược lại thì ***{Câu\_lệnh}*** bị bỏ qua.
* Ở dạng đủ, ***{Biểu\_thức\_logic}*** sẽ được kiểm tra, nếu như điều kiện đó đúng thì thực hiện ***{Câu\_lệnh\_1},*** ngược lại thì thực hiện ***{Câu\_lệnh\_2}.***

***Ví dụ 1:*** Để thực hiện tìm giá trị nhỏ nhất trong hai số nguyên a và b, ta viết như sau:

#include <iostream>

using namespace std;

int main()

{

int a, b;

cin >> a >> b;

int min\_value = 0;

if (a < b)

min\_value = a;

else

min\_value = b;

}

***Ví dụ 2:*** Xác định một số nguyên a có chia hết cho 3 hay không:

#include <bits/stdc++.h>

using namespace std;

int main()

{

int a;

cin >> a;

if (a % 3 == 0)

cout << "a chia hết cho 3";

else

cout << "a không chia hết cho 3";

}

**IV. Câu lệnh if lồng nhau và biểu thức logic gộp**

**1. Câu lệnh if lồng nhau**

Trong quá trình lập trình, đôi khi chúng ta phải kiểm tra cùng một lúc nhiều biểu thức logic rồi mới quyết định thực hiện một công việc nào đó. Chẳng hạn, để kết luận một bộ ba (a, b, c) có phải là ba cạnh của một tam giác hay không, chúng ta cần kiểm tra tới vài điều kiện khác nhau:

* (a > 0), (b > 0) và (c > 0).
* (a + b > c), (a + c > b) và (b + c > a).

Khi đó, ta có thể viết nhiều câu lệnh if gộp vào nhau để kiểm tra các điều kiện cùng lúc:

if (a > 0)

if (b > 0)

if (c > 0)

if (a + b > c)

if (a + c > b)

if (b + c > a)

...

Cú pháp tổng quát của câu lệnh if lồng nhau sẽ là:

if ({Biểu\_thức\_logic\_1})

if ({Biểu\_thức\_logic\_2})

if ({Biểu\_thức\_logic\_3})

...

Nếu như biểu thức ở dạng đầy đủ, tức là bao gồm cả else, thì chúng ta cũng có thể gộp tương tự:

if ({Biểu\_thức\_logic\_1})

{

if ({Biểu\_thức\_logic\_2})

{

if ({Biểu\_thức\_logic\_3})

...

else

{Câu\_lệnh\_3}

}

else

{Câu\_lệnh\_2};

}

else

{Câu\_lệnh\_1};

Khi viết ở dạng đầy đủ, các câu lệnh thuộc câu lệnh if bên trong nên được đặt trong một cặp dấu {} để phân tách với vế else của câu lệnh if bên ngoài. Chương trình sẽ tuần tự kiểm tra từng biểu thức logic từ ngoài vào trong, nếu tất cả các biểu thức đều có giá trị true cho tới biểu thức trong cùng thì câu lệnh trong cùng sẽ được thực hiện. Ngược lại thì chương trình sẽ thực hiện tới vế else của câu lệnh if đầu tiên có biểu thức logic của nó trả ra giá trị false.

**2. Biểu thức logic gộp**

Dễ dàng nhận thấy, nếu như với mỗi điều kiện mà ta lại sử dụng một câu lệnh if, thì chương trình sẽ rất dài và khó kiểm soát. Những trường hợp như vậy, thay vì viết nhiều câu lệnh if, thì ta sử dụng tới ***biểu thức logic gộp***. Hiểu một cách đơn giản, ***biểu thức logic gộp*** là nhiều biểu thức logic kết hợp lại với nhau bằng các toán tử logic && hoặc ||. Khi kết hợp các biểu thức logic lại với nhau, nếu như bạn muốn biểu thức nào đó được ưu tiên tính toán trước thì phải đặt chúng trong cặp ngoặc (). Cùng xem vài ví dụ:

(a > 0 && b > 0 && c > 0 && a + b > c && a + c > b && b + c > a)

(a + b == 0 || (a + b != 0 && a != b))

(5 \* 3 <= 15 && (5 == 5 || 5 == 6))

Nguyên tắc tính toán của các biểu thức logic sẽ được tính toán theo thứ tự ưu tiên của toán tử: Trong ngoặc trước, ngoài ngoặc sau, rồi tới các toán tử khác. Kết quả của cả biểu thức sẽ phụ thuộc vào sự kết hợp giữa các biểu thức logic thành phần là && hay ||. Nếu quan hệ giữa chúng là && - **đồng thời**, thì tất cả các biểu thức thành phần đều phải có giá trị true mới khiến cho biểu thức ban đầu mang giá trị true. Ngược lại, nếu quan hệ giữa chúng là || - **hoặc**, thì chỉ cần một trong số các biểu thức thành phần mang giá trị true, cả biểu thức sẽ mang giá trị true. Chẳng hạn, với biểu thức (5 \* 3 <= 15 && (5 == 5 || 5 == 6)), chương trình sẽ tính nó như sau:

* Đầu tiên xét biểu thức trong ngoặc: (5 == 5 || 5 == 6). Biểu thức này có kết quả là true, do 5 == 5 là true còn 5 == 6 là false, đồng thời quan hệ giữa chúng chỉ là || - **hoặc**.
* Kế đến xét biểu thức 5 \* 3 <= 15. Biểu thức này tất nhiên có giá trị là true.
* Cuối cùng gộp kết quả của (5 \* 3 <= 15) và (5 \* 3 <= 15 && (5 == 5 || 5 == 6)) bằng toán tử &&. Do cả hai biểu thức thành phần đều có giá trị true nên cả biểu thức ban đầu sẽ có giá trị true.

Nói chung, để đặt được đúng các biểu thức logic theo thứ tự mong muốn, các bạn cần có kiến thức vững về độ ưu tiên toán tử, nghĩa là biết cái gì sẽ được tính trước, cái gì sẽ được tính sau để đặt các biểu thức vào trong ngoặc cho đúng.

**V. Sử dụng toán tử điều kiện thay cho câu lệnh if...else**

Ở bài đầu tiên, có một loại toán tử đặc biệt trong C++, đó là toán tử hỗn hợp. Trong số đó, có một loại với cú pháp như sau:

(X) ? A : B

trong đó (X) là một biểu thức logic, A và B là các biểu thức hoặc câu lệnh. Thực ra, đây chính là một cách viết khác của câu lệnh if...else. Thay vì viết:

if (X)

{

A;

}

else

{

B;

}

thì ta sẽ viết như trên, cách viết đó sẽ ngắn gọn hơn rất nhiều. Chẳng hạn, muốn kiểm tra một số nguyên N có chia hết cho 2 hay không, thay vì viết:

if (N % 2 == 0)

cout << "N chia hết cho 2";

else

cout << "N không chia hết cho 2";

thì ta viết:

(N % 2 == 0) ? cout << "N chia hết cho 2" : cout << "N không chia hết cho 2";

Ngoài ra, khi sử dụng toán tử này, nếu như cần thực hiện nhiều câu lệnh với một điều kiện, thì ta có thể phân tách chúng bằng dấu , mà không cần phải tạo khối lệnh bằng cặp dấu {} như câu lệnh if..else truyền thống. Cú pháp viết như sau:

(X) ? A1, A2,..., AN : B1, B2,..., BN;

Tuy nhiên, có một lưu ý rằng, bản chất của cặp toán tử (X) ? A : B thực ra là ***trả về*** giá trị A nếu như biểu thức logic (X) có giá trị true, ngược lại thì ***trả về*** giá trị B. Do đó, chúng ta không được sử dụng câu lệnh ở vế A hoặc vế B khác kiểu nhau. Chẳng hạn, câu lệnh sau sẽ là không hợp lệ:

int b = 0;

(N % 2 == 0 && N % 3 == 0) ? cout << "N chia hết cho 6" : b = 1;

Lí do vì câu lệnh này có vế A là một câu lệnh in ra, còn vế B lại là biểu thức số học. Hơi khó để phân biệt, nhưng nếu làm nhiều bài tập các bạn sẽ biết rằng những biểu thức nào có thể kết hợp với nhau. Tuy nhiên, cách viết này không được ưu tiên, bởi vì nó có thể rất rối mắt trong những trường hợp câu lệnh dài và nhiều. Chúng ta chỉ nên sử dụng toán tử này khi cần thực hiện những câu lệnh rất ngắn.

**VI. Cấu trúc lựa chọn switch...case...**

**1. Lựa chọn nhiều trường hợp**

Đôi khi, người ta cần kiểm tra rất nhiều trường hợp xảy ra của một đối tượng, và ứng với mỗi trường hợp ta sẽ có cách xử lý khác nhau. Ví dụ, một cửa hàng quần áo đặt biển quảng cáo như sau:

* Nếu mua 1 chiếc áo, giá tiền là 50.000 VNĐ/chiếc.
* Nếu mua 2 chiếc áo, giá tiền là 45.000 VNĐ/chiếc.
* Nếu mua 3 chiếc áo, giá tiền là 40.000 VNĐ/chiếc.
* Nếu mua nhiều hơn 3 chiếc áo, giá tiền là 35.000 VNĐ/chiếc và được tặng thêm một đôi tất.

Ở đây, đối tượng đang được xem xét là ***số lượng chiếc áo được mua***, và ứng với mỗi giá trị 1, 2, 3 hoặc nhiều hơn 3 sẽ có một đơn giá khác nhau cho mỗi chiếc áo. Những cách diễn đạt như vậy được gọi là ***mệnh đề lựa chọn***. C++ cung cấp một cấu trúc để biểu diễn các mệnh đề dạng lựa chọn là switch...case....

**2. Cấu trúc lựa chọn switch...case...**

Cấu trúc switch...case... dùng để lựa chọn trường hợp giá trị cho một đối tượng xác định, và quyết định sẽ làm công việc gì tương ứng với mỗi trường hợp xảy ra.

***Cú pháp:***

switch ({Biểu\_thức})

{

case {Giá\_trị\_1}:

{Câu\_lệnh\_1};

break;

case {Giá\_trị\_2}:

{Câu\_lệnh\_2};

break;

case {Giá\_trị\_3}:

{Câu\_lệnh\_3};

break;

...

default:

{Câu\_lệnh\_mặc\_định};

}

Trong đó, ***{Biểu\_thức}*** là một biến/biểu thức, hoặc hàm có giá trị, thường là kiểu số nguyên hoặc kiểu ***chuỗi - kí tự***. Các giá trị 1, 2, 3... phải là các hằng số. Nếu như giá trị của đối tượng không trùng với bất kỳ giá trị nào trong danh sách thì câu lệnh ở default sẽ được thực hiện.

Trong trường hợp nhiều giá trị có chung công việc thực hiện, ta có thể viết theo cách sau:

case {Giá\_trị\_1} : case {Giá\_trị\_2} : ... : case {Giá\_trị\_n}:

{Câu\_lệnh};

break;

***Ví dụ 1:*** Nhập vào một tháng bất kỳ trong năm, in ra màn hình số ngày của tháng đó.

int month;

cin >> month;

switch (month)

{

case 1 : case 3 : case 5 : case 7 : case 8 : case 10 : case 12:

cout << "Có 31 ngày";

break;

case 2:

cout << "Có 28 hoặc 29 ngày";

break;

default:

cout << "Có 30 ngày";

break;

}

***Ví dụ 2:*** Nhập vào điểm môn học dạng chữ của một sinh viên (A, B, C, D, F),(*A*,*B*,*C*,*D*,*F*), hãy đưa ra điểm tích lũy tương ứng của môn học đó (4, 3, 2, 1, 0)?(4,3,2,1,0)?

char point;

cin >> point;

int CPA = 0; // Điểm tích lũy môn học.

switch (point)

{

case 'A':

CPA = 4;

break;

case 'B':

CPA = 3;

break;

case 'C':

CPA = 2;

break;

case 'D':

CPA = 1;

break;

default:

CPA = 0;

break;

}

**Bài 5: Cấu trúc lặp**

**I. Lặp với số lần biết trước và lặp với số lần không biết trước**

Cùng xem xét hai bài toán sau đây:

* Bài toán 1: Tính giá trị tổng S, biết:
* Bài toán 2: Tính giá trị tổng S, biết:

cho tới khi S ≥ 106.

Đối với bài toán 1, S có thể được tính bằng cách sau:

* Đầu tiên, S được gán giá trị 1.
* Tiếp theo, cộng vào tổng S một giá trị  với Việc cộng này được lặp lại n - 1 lần tới khi x = n thì thu được tổng S ban đầu.

Đối với bài toán 2, ta có một cách tính tương tự:

* Đầu tiên, S được gán giá trị 1.
* Tiếp theo, cộng vào tổng S một số x với x = 2, 3, 4,... Số lần lặp lại công việc này chưa được biết trước nhưng sẽ dừng lại khi đạt được tổng S ≥ 106.

Nhìn chung, trong mọi thuật toán đều sẽ có những thao tác phải lặp đi lặp lại nhiều lần. Có hai dạng lặp là ***lặp với số lần biết trước*** và ***lặp với số lần không biết trước***. C++ cung cấp cả hai cấu trúc lặp để biểu diễn hai thao tác lặp này.

**II. Câu lệnh for - Lặp với số lần biết trước**

Câu lệnh for là một cấu trúc điều khiển rất hiệu quả để lặp đi lặp lại một công việc trong một số lần nhất định.

***Cú pháp:***

for ({Khởi\_tạo\_biến\_đếm}; {Điều\_kiện\_dừng}; {Tăng\_giảm\_biến\_đếm})

{

{Khối\_lệnh};

}

***Quy trình thực hiện của vòng lặp:***

* Bước 1 - ***{Khởi\_tạo\_biến\_đếm}:*** Một lệnh khởi tạo ra một hoặc nhiều biến đếm bất kỳ trong vòng lặp, hoặc không khởi tạo biến nào cả (tuy nhiên vẫn phải có dấu chấm phẩy).
* Bước 2 - ***{Điều\_kiện\_dừng}:*** Là một biểu thức logic xác định điều kiện khi nào vòng lặp sẽ dừng lại. Trước khi thực hiện ***{Khối\_lệnh},*** biểu thức sẽ kiểm tra kết quả, nếu là true thì ***|Khối lệnh|*** sẽ được thực hiện tiếp, ngược lại thì vòng lặp sẽ dừng.
* Bước 3 - ***{Khối\_lệnh}:*** Sau khi kiểm tra ***{Điều\_kiện\_dừng}*** và thấy kết quả là true, khối lệnh sẽ được thực hiện.
* Bước 4 - ***{Tăng\_giảm\_biến\_đếm}:*** Cập nhật giá trị biến đếm theo bất kỳ cách nào người dùng mong muốn (có thể không cập nhật gì cả nhưng vẫn phải có dấu chấm phẩy). Sau khi cập nhật xong, quá trình quay trở lại từ bước 22 cho tới khi ***{Điều\_kiện\_dừng}*** nhận giá trị false.

***Ví dụ:*** Cùng xét chương trình sau:

#include <iostream>

using namespace std;

int main()

{

int N;

cin >> N;

int S = 0;

for (int i = 1; i <= N; ++i)

S = S + i;

cout << "Tổng từ 1 tới N là: " << S;

return 0;

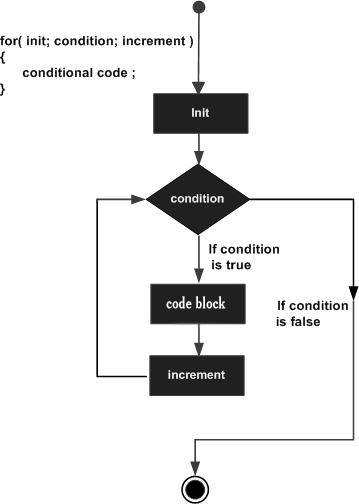
}

Vòng lặp sẽ khởi tạo biến đếm i = 1, kế đến kiểm tra điều kiện i <= N, nếu điều kiện này đúng thì thực hiện câu lệnh S = S + i, sau đó tăng biến i lên 1 đơn vị, rồi lại quay trở về kiểm tra điều kiện i <= N,...Quá trình sẽ lặp lại cho tới khi biến i nhận giá trị N + 1, vòng lặp dừng lại.

Giả sử nhập vào N bằng 10, kết quả in ra là:

Tổng từ 1 tới N là 55

***Sơ đồ quá trình thực hiện:***



**III. Câu lệnh while - lặp với số lần không biết trước**

Đôi khi chúng ta cần lặp lại một công việc nào đó, nhưng không biết trước số lần sẽ lặp mà chỉ biết được điều kiện khi nào công việc đó sẽ dừng lại, khi đó vòng lặp while sẽ trở nên rất hữu dụng. Có hai dạng vòng lặp **while** trong C++: while và do...while.

**1. Vòng lặp while**

***Cú pháp:***

while ({Điều\_kiện\_dừng})

{

{Khối\_lệnh};

}

***Quy trình thực hiện vòng lặp:***

* Bước 1 - ***{Điều\_kiện\_dừng}:*** Là biểu thức logic xác định điều kiện dừng của vòng lặp. Trước khi thực thi ***{Khối\_lệnh},*** chương trình sẽ kiểm tra ***{Điều\_kiện\_dừng},*** nếu kết quả là true thì tiếp tục thực thi ***{Khối\_lệnh},*** ngược lại vòng lặp kết thúc.
* Bước 2 - ***{Khối\_lệnh}:*** Sau khi kiểm tra ***{Điều\_kiện\_dừng}*** và thấy kết quả là true, khối lệnh sẽ được thực hiện. Sau đó, quy trình tiếp tục quay lại bước 11 tới khi vòng lặp dừng.

***Ví dụ:***

#include <iostream>

using namespace std;

int main()

{

int S = 0, i = 1;

while (S <= 100)

{

S = S + i;

i = i + 5;

}

cout << "Tổng S là: " << S;

return 0;

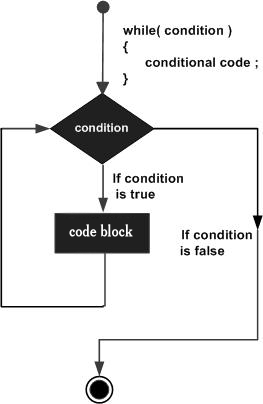
}

Đầu tiên chương trình sẽ khởi tạo biến hai biến S = 0 và i = 1. Sau đó khi vào vòng lặp, kiểm tra điều kiện S <= 100, nếu điều kiện này đúng thì thực hiện câu lệnh S = S + i, sau đó tăng biến i lên 5 đơn vị, rồi lại quay trở về kiểm tra điều kiện i <= N,...Quá trình sẽ lặp lại cho tới khi biến S nhận giá trị lớn hơn 100 thì vòng lặp dừng lại.

Kết quả in ra khi chạy chương trình là:

Tổng S là 112

***Sơ đồ quá trình thực hiện:***



**2. Vòng lặp do...while**

Về cơ bản, vòng lặp do...while giống với vòng lặp while, nhưng điểm khác biệt là vòng lặp do...while sẽ thực hiện khối lệnh trước rồi mới tiến hành kiểm tra điều kiện dừng. Có nghĩa là, khối lệnh trong vòng lặp do...while luôn luôn được thực thi ít nhất một lần, trong khi vòng lặp while thì có thể sẽ không thực hiện lần nào cả.

***Cú pháp:***

do

{

{Khối\_lệnh};

}

while ({Điều\_kiện\_dừng});

***Quy trình thực hiện vòng lặp:***

* Bước 1 - ***{Khối\_lệnh}:*** ***{Khối\_lệnh}|*** sẽ được thực thi trước tiên.
* Bước 2 - ***{Điều\_kiện\_dừng}:*** Kiểm tra ***{Điều\_kiện\_dừng},*** nếu biểu thức logic trả ra kết quả true thì tiếp tục quay trở về bước 1, ngược lại thì dừng vòng lặp.

***Ví dụ:***

#include <iostream>

using namespace std;

int main()

{

int S = 0, i = 1;

do

{

S = S + i;

i = i + 5;

}

while (S <= 100);

cout << "Tổng S là: " << S;

return 0;

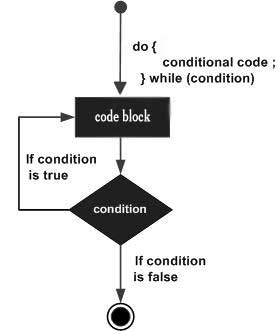
}

Đầu tiên chương trình sẽ khởi tạo biến hai biến S = 0 và i = 1. Khi vào vòng lặp, đầu tiên chương trình sẽ tính S = S + i, kế đến gán i = i + 5 rồi mới kiểm tra điều kiện S <= 100. Nếu điều kiện vẫn đang có kết quả là true thì tiếp tục thực hiện lại khối lệnh, lặp lại cho tới khi biến S nhận giá trị lớn hơn 100 thì vòng lặp dừng lại.

Kết quả in ra khi chạy chương trình là:

Tổng S là 112

***Sơ đồ quá trình thực hiện:***



**IV. Vòng lặp lồng nhau**

Cú pháp C++ cho phép người dùng có thể lồng nhiều vòng lặp vào nhau. Khi đó, ứng với mỗi lần lặp của vòng lặp bên ngoài thì vòng lặp bên trong sẽ được thực hiện đủ số lần lặp của nó. Dưới đây là một số ví dụ:

**1. Lồng vòng lặp for**

***Cú pháp:***

for ({Khởi\_tạo\_biến\_đếm\_1}; {Điều\_kiện\_dừng\_1}; {Tăng\_giảm\_biến\_đếm\_1})

{

for ({Khởi\_tạo\_biến\_đếm\_2}; {Điều\_kiện\_dừng\_2}; {Tăng\_giảm\_biến\_đếm\_2})

{

{Khối\_lệnh\_2};

...

}

{Khối\_lệnh\_1};

...

}

***Ví dụ:*** Chương trình dưới đây sử dụng hai vòng lặp for lồng nhau để đếm số ước nguyên dương của mỗi số nguyên dương từ 1 tới 10:

#include <iostream>

#include <cmath>

using namespace std;

int main()

{

for (int i = 1; i <= 10; ++i) // Duyệt các số i từ 1 tới 10.

{

int cnt\_divisors = 0; // Số ước của i.

for (int j = 1; j <= i; ++j) // Duyệt tìm các ước của i.

if (i % j == 0)

++cnt\_divisors;

cout << "Số ước của " << i << " là: " << cnt\_divisors << endl;

}

return 0;

}

Biên dịch và chạy chương trình trên sẽ cho ra kết quả:

Số ước của 1 là 1

Số ước của 2 là 2

Số ước của 3 là 2

Số ước của 4 là 3

Số ước của 5 là 2

Số ước của 6 là 4

Số ước của 7 là 1

Số ước của 8 là 4

Số ước của 9 là 3

Số ước của 10 là 4

**2. Lồng vòng lặp while**

***Cú pháp:***

while ({Điều\_kiện\_1})

{

while ({Điều\_kiện\_2})

{

{Khối\_lệnh\_2};

...

}

{Khối\_lệnh\_1};

...

}

***Ví dụ:*** Đặt F(N) là tổng các số từ 1 tới N, chương trình dưới đây sử dụng hai vòng lặp while lồng nhau để in ra các giá trị F(1), F(2),.., F(5):

#include <iostream>

#include <cmath>

using namespace std;

int main()

{

int i = 1;

while (i <= 5)

{

int j = 1, f = 0;

while (j <= i)

{

f += j;

++j;

}

cout << "F(" << i << " = " << f << endl;

++i; // Tăng i lên để tính F(i) tiếp theo.

}

return 0;

}

Biên dịch và chạy chương trình, ta thu được kết quả là:

F(1) = 1

F(2) = 3

F(3) = 6

F(4) = 10

F(5) = 15

**3. Lồng vòng lặp do...while**

***Cú pháp:***

do

{

{Khối\_lệnh\_1};

do

{

{Khối\_lệnh\_2};

...

}

while ({Điều\_kiện\_2});

...

}

while ({Điều\_kiện\_1});

***Ví dụ:*** Đặt F(N) là tổng các số từ 1 tới N, chương trình dưới đây sử dụng hai vòng lặp do...while lồng nhau để in ra các giá trị F(1), F(2),.., F(5):

#include <iostream>

#include <cmath>

using namespace std;

int main()

{

int i = 1;

do

{

int j = 1, f = 0;

do

{

f += j;

++j;

}

while (j <= i);

cout << "F(" << i << " = " << f << endl;

++i; // Tăng i lên để tính F(i) tiếp theo.

}

while (i <= 5);

return 0;

}

Biên dịch và chạy chương trình, ta thu được kết quả là:

F(1) = 1

F(2) = 3

F(3) = 6

F(4) = 10

F(5) = 15

**4. Lồng vòng lặp hỗn hợp**

***Cú pháp:***

for ({Khởi\_tạo\_biến\_đếm\_1}; {Điều\_kiện\_dừng\_1}; {Tăng\_giảm\_biến\_đếm\_1})

{

while ({Điều\_kiện\_2})

{

{Khối\_lệnh\_2};

do

{

{Khối\_lệnh\_3};

....

}

while ({Điều\_kiện\_3});

...

}

{Khối\_lệnh\_1};

...

}

***Ví dụ:*** Chương trình dưới đây lồng hai vòng lặp for và while để tính tổng 1! + 2! + … + 10!:

#include <iostream>

#include <cmath>

using namespace std;

int main()

{

int sum = 0; // Biến lưu tổng 1! + 2! + ... + 10!

for (int i = 1; i <= 10; ++i)

{

int j = 1, factorial = 1;

while (j <= i)

{

factorial \*= j;

++j;

}

sum += factorial;

}

cout << sum;

}

Biên dịch và chạy chương trình, ta thu được kết quả là:

4037913

**V. Các lệnh break và continue**

**1. Lệnh break**

***Tác dụng:*** Sử dụng để ngắt vòng lặp. Khi gặp lệnh break, vòng lặp đang chứa lệnh đó sẽ ngay lập tức bị ngắt kể cả khi ***{Điều\_kiện\_dừng}*** vẫn đang nhận giá trị true.

***Cú pháp:***

break;

***Ví dụ:***

#include <iostream>

using namespace std;

int main()

{

int S = 0;

for (int i = 1; i <= 15; ++i)

{

S = S + i;

cout << i << endl;

if (S > 10) // Nếu S > 10 thì dừng vòng lặp luôn.

break;

}

cout << "Tổng S là: " << S;

return 0;

}

Biên dịch và chạy chương trình sẽ cho kết quả:

1

2

3

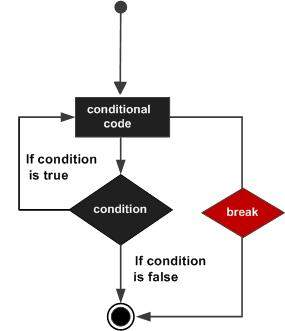
4

5

Tổng S là: 15

Ta thấy mặc dù vòng lặp có điều kiện dừng là i <= 15, nhưng khi đến i = 5 thì vòng lặp đã bị dừng lại vì điều kiện S = 10 đã nhận giá trị true và lệnh break sẽ phát huy tác dụng.

**Sơ đồ hoạt động:**



**2. Lệnh continue**

***Tác dụng:*** Sử dụng để bỏ qua một lần lặp. Khi gặp lệnh continue, toàn bộ phần lệnh bên dưới lệnh này (thuộc cùng khối lệnh) sẽ bị bỏ qua và chuyển tới lần lặp tiếp theo trong vòng lặp đó. Đối với vòng lặp for, lệnh continue sẽ điều khiển tới phần ***{Tăng giảm biến đếm};*** còn đối với lệnh while và do...while thì lệnh continue sẽ điều khiển tới phần ***{Điều\_kiện\_dừng}.***

***Cú pháp:***

continue;

***Ví dụ:***

#include <iostream>

using namespace std;

int main ()

{

int a = 10;

do

{

if (a == 15) // Nếu a = 15 thì bỏ qua toàn bộ đoạn lệnh bên dưới.

{

a = a + 1;

continue;

}

cout << "Gia tri cua a la: " << a << endl;

a = a + 1;

}

while(a < 20);

return 0;

}

Biên dịch và chạy chương trình trên sẽ cho ra kết quả là:

Giá trị của a là 10

Giá trị của a là 11

Giá trị của a là 12

Giá trị của a là 13

Giá trị của a là 14

Giá trị của a là 16

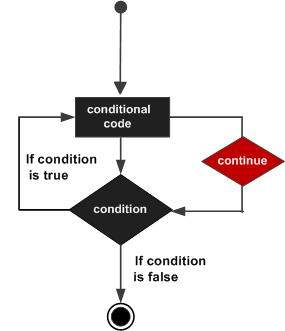
Giá trị của a là 17

Giá trị của a là 18

Giá trị của a là 19

Ta thấy giá trị a = 15 đã bị bỏ qua mất do lệnh continue.

***Sơ đồ hoạt động:***



**VI. Vòng lặp vô hạn**

Đôi khi trong chương trình, chúng ta sẽ gặp phải những trường hợp vòng lặp bị lặp vô hạn lần, dẫn đến chương trình không thể trả ra kết quả. Điều này là do thuật toán của chúng ta đã bị sai sót ở đâu đó khiến cho điều kiện dừng của vòng lặp không bao giờ có thể đạt tới giá trị false. Cùng xem xét ví dụ sau:

for (int i = 1; i <= 10; ++i)

{

S = S + i;

i = 1;

}

cout << S;

Biên dịch đoạn code này sẽ không trả ra được kết quả, nguyên do là vì ở mỗi lần lặp, biến i lại được gán lại bằng 1 trong khối lệnh, dẫn đến i không bao giờ đạt được giá trị lớn hơn 10 cả. Vì vậy cần hết sức chú ý vấn đề này trong khi thiết kế giải thuật.

Vòng lặp vô hạn đôi khi cũng có ích, ví dụ trong các trường hợp như lập trình viên không thể xác định được số lần lặp để xử lý một logic nào đó (như trong quá trình nhập dữ liệu từ file mà không biết trước số dòng chẳng hạn). Lúc này cần sử dụng đến vòng lặp vô hạn, và các dạng thường được dùng sẽ là: while (true), for( ; ;), while (1),...Khi điều kiện của vòng lặp for bị bỏ trống, nó sẽ mặc định nhận giá trị true.

**Bài 6: Hàm trong C++**

**I. Phân chia chương trình thành từng hàm**

**1. Kĩ thuật lập trình hướng cấu trúc**

Cùng xem xét một ví dụ sau: *Cho ba số nguyên dương a, b, c, đều không nhỏ hơn 2, hãy kiếm tra xem những số nào là số nguyên tố và đưa ra thông báo số đó là số nguyên tố?*.

#include <iostream>

using namespace std;

int main()

{

int a, b, c;

cin >> a >> b >> c;

bool kt = true; // Biến logic kt dùng để kiểm tra a có phải số nguyên tố không.

for (int i = 2; i < a; ++i)

if (a % i == 0)

{

kt = false;

break; // Vì a đã không phải số nguyên tố nên có thể dừng kiểm tra luôn.

}

if (kt == true)

cout << a << " là số nguyên tố";

else

cout << a << " không là số nguyên tố";

// Thực hiện hoàn toàn tương tự đối với b và c.

kt = true;

for (int i = 2; i < b; ++i)

if (b % i == 0)

{

kt = false;

break; // Vì a đã không phải số nguyên tố nên có thể dừng kiểm tra luôn.

}

if (kt == true)

cout << b << " là số nguyên tố";

else

cout << b << " không là số nguyên tố";

kt = true;

for (int i = 2; i < c; ++i)

if (c % i == 0)

{

kt = false;

break; // Vì a đã không phải số nguyên tố nên có thể dừng kiểm tra luôn.

}

if (kt == true)

cout << c << " là số nguyên tố";

else

cout << c << " không là số nguyên tố";

return 0;

}

Biên dịch và chạy chương trình trên với a=2, b=5, c = 9 sẽ cho ra kết quả:

2 là số nguyên tố

5 là số nguyên tố

9 không là số nguyên tố

Vấn đề chúng ta nhận thấy ở đây là, quá trình kiểm tra số nguyên tố đối với từng số là quá trình tương tự nhau, chỉ thay đổi giá trị số thôi, nhưng chúng ta phải viết lại tới ba lần khiến cho chương trình rất dài. Điều này sẽ còn tệ hơn nữa nếu như cần kiểm tra 4, 5,... thậm chí là 1000 số!

Trên thực tế, lập trình chính là xây dựng một quá trình giải bài toán gồm nhiều bước khác nhau, mỗi bước sẽ giải quyết một công việc cụ thể. Rất nhiều trường hợp ta sẽ gặp phải các công việc tương tự nhau, để tránh phải viết đi viết lại một đoạn code gây lãng phí thời gian, ta sẽ chia chương trình thành các hàm con, mỗi hàm phụ trách một công việc nhất định và tất cả sẽ được gọi ra trong một hàm chính. Cách lập trình như vậy gọi là ***lập trình hướng cấu trúc*** hay ***lập trình module hóa***. Ưu điểm của phương pháp này là:

* Tư duy giải thuật rõ ràng.
* Chương trình đơn giản và dễ hiểu.
* Có thể tái sử dụng lại các đoạn code tại nhiều vị trí trong chương trình mà không phải viết nhiều lần.
* Dễ dàng theo dõi và kiểm tra, chỉnh sửa giải thuật.

Cách lập trình này rất phù hợp trong lập trình thi đấu, vì mỗi giải thuật thường chỉ áp dụng cho những bài toán nhất định. Tất nhiên, nó cũng có những nhược điểm, nhưng chúng ta sẽ không bàn đến trong bài học này.

**2. Định nghĩa về hàm. Hàm tự định nghĩa và hàm dựng sẵn**

C++ là một ngôn ngữ lập trình có khả năng hướng cấu trúc, bao gồm một hàm main() và các hàm con khác trong chương trình. Hàm là một khối các lệnh dùng để xử lý một phần công việc nhỏ nào đó trong chương trình.

Có hai loại hàm trong C++ là ***hàm tự định nghĩa*** và ***hàm dựng sẵn*** được cung cấp bởi các thư viện của C++. Sử dụng linh hoạt các hàm sẽ hỗ trợ rất tốt cho lập trình viên trong quá trình xây dựng chương trình.

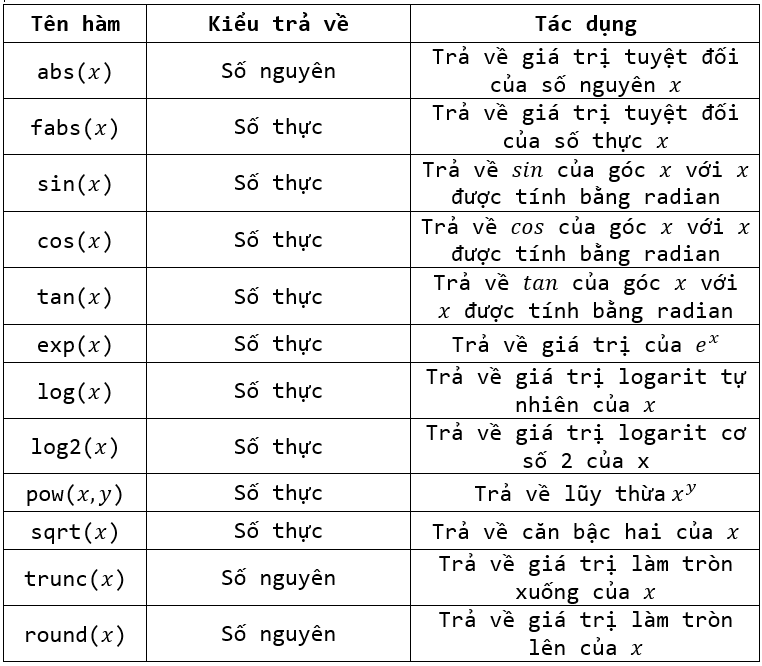
**II. Các hàm toán học dựng sẵn trong C++**

Trong thư viện chuẩn của C++ cung cấp khá nhiều hàm dựng sẵn giúp ích cho người lập trình, và một trong số đó là các hàm toán học. Để sử dụng các hàm toán học trong C++, đầu tiên ta cần khai báo thư viện và không gian tên chứa chúng:

#include <cmath>

using namespace std;

Bảng dưới đây là một số hàm thường sử dụng trong quá trình làm việc với C++:



**III. Các bước định nghĩa và sử dụng một hàm**

**1. Khai báo hàm**

Giống như các biến và hằng, một hàm phải được khai báo và định nghĩa trước khi sử dụng. Một khai báo hàm có thể đặt ở bất kỳ đâu trong chương trình. Cú pháp khai báo một hàm như sau:

{Kiểu\_trả\_về} {Tên\_hàm}({Danh\_sách\_tham số});

Trong đó ta có:

* ***{Kiểu\_trả\_về}:*** Mỗi hàm đều phải trả về một giá trị nào đó. ***{Kiểu\_trả\_về}*** là kiểu dữ liệu của giá trị mà hàm đó sẽ trả về. Ngoài ra, C++ cho phép tạo ra những hàm chỉ thực hiện công việc mà không trả ra giá trị nào cả, khi đó ***{Kiểu\_trả\_về}*** sẽ là void.
* ***{Tên\_hàm}:*** Là tên mà người dùng đặt cho hàm, quy ước đặt tên theo đúng convention của C++ là ***snake\_case*** và không nên trùng với các từ khóa của hệ thống.
* ***{Danh\_sách\_tham số}:*** Khi sử dụng một hàm để thực hiện công việc nào đó, ta cần cung cấp dữ liệu đầu vào cho hàm. Các tham số chính là các biến lưu trữ những dữ liệu đó để hàm sử dụng trong quá trình tính toán. Danh sách tham số không bắt buộc phải luôn luôn có.

Ví dụ, dưới đây là một vài khai báo hàm hợp lệ:

int sum(int a, int b); // Hàm tính tổng hai số nguyên a và b.

double get\_circle\_area(double d) // Hàm tính diện tích hình tròn có đường kính là d.

long long multiply(long long a, long long b, long long c) // Hàm tính tích ba số nguyên a, b, c.

**2. Định nghĩa hàm**

Phần định nghĩa hàm là phần quan trọng nhất của một hàm, nó quyết định hàm đó sẽ làm công việc gì và trả ra kết quả là gì. Cú pháp định nghĩa hàm như sau:

{Khai\_báo\_hàm}

{

{Thân\_hàm};

}

Trong đó, phần ***{Thân\_hàm}*** chứa các câu lệnh của hàm. Đối với hàm có ***{Kiểu\_trả\_về}*** là một ***kiểu dữ liệu*** nào đó thì bắt buộc phải có ít nhất một dòng lệnh return {Giá\_trị\_trả\_về}; trong thân hàm. Còn đối với hàm có ***{Kiểu\_trả\_về}*** là void thì không cần phải (và cũng không thể) return một giá trị nào cả. Phần định nghĩa hàm có thể đặt ngay bên dưới khai báo hàm, hoặc sau khi đã khai báo hàm rồi mới định nghĩa ở một vị trí nào đó trong chương trình.

***Ví dụ:*** Hàm dưới đây trả về giá trị lớn nhất giữa hai số nguyên a và b:

int max\_value(int a, int b)

{

if (a > b)

return a;

else

return b;

}

**3. Lời gọi hàm**

Một hàm sau khi được định nghĩa, nếu muốn hoạt động thì phải được gọi ra trong hàm main(), hoặc được gọi trong một hàm khác và hàm đó được gọi trong main().

Việc hàm được gọi ra như thế nào cũng tùy thuộc vào ***{Kiểu\_trả\_về}*** của hàm. Nếu như ***{Kiểu\_trả\_về}*** là một ***kiểu dữ liệu*** - nghĩa là hàm có giá trị trả về - thì lời gọi hàm đó được phép sử dụng kết hợp trong các câu lệnh gán, biểu thức tính toán hoặc logic. Ngược lại nếu ***{Kiểu\_trả\_về}*** là void thì hàm đó chỉ được phép đứng đơn lẻ khi gọi ra.

***Ví dụ 1:*** Hàm int max\_value(int a, int b) được gọi ra trong chương trình chính để tìm giá trị lớn nhất của 3 cặp số khác nhau:

#include <iostream>

using namespace std;

int max\_value(int a, int b)

{

if (a > b)

return a;

else

return b;

}

int main()

{

int x1, y1, x2, y2, x3, y3;

cin >> x1 >> y1 >> x2 >> y2 >> x3 >> y3;

// Lời gọi hàm.

cout << "Max giữa x1 và y1 là: " << max\_value(x1, y1) << endl;

cout << "Max giữa x2 và y2 là: " << max\_value(x2, y2) << endl;

cout << "Max giữa x3 và y3 là: " << max\_value(x3, y3);

return 0;

}

Kết quả chạy chương trình với x1 = 1, y1 = 2, x2 = 5, y2 = 4, x3 = 10, y3 = 10 là:

Max giữa x1 và y1 là: 2

Max giữa x2 và y2 là: 5

Max giữa x3 và y3 là: 10

Ta thấy ở ví dụ trên, cần tìm giá trị lớn nhất giữa ba cặp số (x, y) khác nhau. Nếu không sử dụng hàm, ta sẽ phải viết lại quy trình tìm giá trị lớn nhất đủ 3 lần, nhưng khi dùng hàm thì việc kiểm tra chỉ cần viết một lần trong hàm và gọi hàm đó ra sử dụng ba lần với tham số truyền vào lần lượt là các cặp số cần kiểm tra thôi. Đây chính là lợi thế của việc viết chương trình thành các hàm.

***Ví dụ 2:*** Dùng hàm void sum(int a, int b) để tính tổng hai số a và b, sau đó lưu vào biến s. Vì kiểu của hàm là void nên chỉ có thể gọi hàm bằng lời gọi đơn lẻ:

#include <iostream>

using namespace std;

int s = 0;

void sum(int a, int b)

{

s = a + b;

}

int main()

{

int a = 5, b = 6;

sum(a, b);

cout << s;

return 0;

}

Kết quả chạy chương trình:

11

**IV. Tham số của hàm**

Tham số thực chất là các biến đại diện cho dữ liệu truyền vào hàm khi có một lời gọi hàm nào đó. Khi dữ liệu truyền vào hàm, nó sẽ được lưu vào tham số và hàm sẽ tính toán bằng dữ liệu trên tham số đó. Tham số được phân làm hai loại: Tham số thực và tham số hình thức.

**1. Tham số thực**

Chính là các biến, hằng ở bên ngoài truyền vào trong một hàm. Khi truyền các giá trị này vào hàm, các giá trị đó sẽ được sử dụng dưới tên của tham số hình thức. Lấy ví dụ:

int sum(int a, int b)

{

return a + b;

}

int main()

{

int a = 10, b = 5;

cout << sum(a, b);

}

Trong hàm main(), ta thấy hai biến a*a* và b*b* được truyền vào hàm int sum(int a, int b);. Ở đây, a*a* và b*b* chính là các tham số thực sự, vì chúng chứa dữ liệu thực tế cần thao tác.

**2. Tham số hình thức**

Là danh sách tham số đứng phía sau khai báo của một hàm. Những tham số này sẽ đại diện cho dữ liệu truyền vào hàm, nhờ vào tham số mà chúng ta có thể tái sử dụng hàm với rất nhiều các bộ dữ liệu khác nhau. Có hai cách truyền tham số thực sự từ bên ngoài vào hàm:

**2.1. Truyền tham trị**

Với cách truyền tham số này, hàm sẽ tạo một bản sao của dữ liệu được truyền vào, sau đó thực hiện các tính toán trong hàm với dữ liệu đó mà không làm thay đổi dữ liệu bên ngoài truyền vào. Ví dụ:

#include <iostream>

using namespace std;

void increase(int x) // Tăng giá trị biến x dùng tham trị.

{

x = x + 2;

}

int main()

{

int a = 5;

cout << "Giá trị a ban đầu: " << a << endl;

increase(a);

cout << "Giá trị a mới: " << a;

}

Chạy chương trình này sẽ thu được kết quả:

Giá trị a ban đầu: 5

Giá trị a mới: 5

Ta thấy hai giá trị a lúc đầu và lúc sau vẫn giữ nguyên. Nguyên do là vì, giá trị của biến a ở hàm main() khi thực hiện lời gọi increase(a) hoàn toàn không bị ảnh hưởng, chương trình đã tạo ra một bản sao của biến a này để truyền vào hàm increase(int x), do đó biến a trong hàm main() và tham số x trong hàm increase(int x) là độc lập.

***Khi nào sử dụng truyền tham trị:*** Nên sử dụng cách truyền tham trị trong những trường hợp mà dữ liệu truyền vào chỉ dùng để tính toán trung gian cho những kết quả khác.

**2.2. Truyền tham chiếu**

Với tham chiếu, biến truyền vào sẽ được truyền thẳng địa chỉ của nó vào trong hàm nhưng với một cái tên khác, và dữ liệu bị thay đổi trong hàm sẽ được cập nhật sang dữ liệu gốc đã truyền vào. Muốn truyền dữ liệu bằng tham chiếu, chỉ cần thêm toán tử & ở phía trước tham số hình thức trong khai báo hàm. Ví dụ:

#include <iostream>

using namespace std;

void increase(int &x) // Tăng giá trị biến x dùng tham trị.

{

x = x + 2;

}

int main()

{

int a = 5;

cout << "Giá trị a ban đầu: " << a << endl;

increase(a);

cout << "Giá trị a mới: " << a;

}

Chạy chương trình này sẽ thu được kết quả:

Giá trị a ban đầu: 5

Giá trị a mới: 7

Giá trị a lúc sau so với ban đầu đã thay đổi. Lí do là vì khi sử dụng tham chiếu để truyền giá trị a vào thì chính địa chỉ của biến a*a* được truyền vào hàm, nhưng dưới cái tên là x, từ đó mọi thay đổi của tham số x sẽ được cập nhật thẳng lên a.

***Khi nào sử dụng truyền tham chiếu:*** Khi cần chuyển dữ liệu đã xử lý ra ngoài hàm để sử dụng trong các hàm khác.

Trong trường hợp muốn sử dụng tham chiếu nhưng không muốn dữ liệu bị cập nhật thay đổi lên biến gốc, ta sử dụng cú pháp khai báo như sau:

{Kiểu\_trả\_về} {Tên\_hàm} (const {Kiểu\_dữ\_liệu} &{Tên\_tham\_chiếu})

***Ví dụ:***

void increase(const int &a);

int get\_sum(const int &a, const int &b, int &sum);

**V. Biến toàn cục và Biến cục bộ**

Tùy vào vị trí khai báo biến mà chúng ta có thể chia các biến trong C++ ra làm hai loại: Biến toàn cục (global variables) và Biến cục bộ (local variables). Sử dụng thành thạo hai loại biến này có ý nghĩa rất quan trọng trong lập trình nói chung và lập trình thi đấu nói riêng.

**1. Biến toàn cục (global variables)**

***Định nghĩa:*** Là các biến được khai báo ở ngoài hàm, có giá trị kể từ vị trí nó được khai báo cho tới hết chương trình. Mọi hàm kể từ vị trí khai báo đều có thể sử dụng biến toàn cục. Khi khai báo biến toàn cục, các biến sẽ nhận giá trị mặc định là 0 đối với kiểu số, và giá trị rỗng đối với kiểu chuỗi hoặc kí tự.

***Ví dụ:***

#include <iostream>

using namespace std;

int x, y; // Biến toàn cục x và y.

void func1() // Hàm thứ nhất.

{

x = x + 1;

y = y + 1;

cout << "Giá trị lần 1: " << x << ' ' << y << endl;

}

void func2() // Hàm thứ hai.

{

x = x \* 2;

y = y \* 2;

cout << "Giá trị lần 2: " << x << ' ' << y;

}

int main()

{

// Gọi hai hàm trong main để thay đổi giá trị x, y.

func1();

func2();

return 0;

}

***Kết quả chạy chương trình:***

Giá trị lần 1: 1 1

GIá trị lần 2: 2 2

Ta thấy hai biến x và y sẽ bị thay đổi ở cả hai hàm func1() và func2(), do x và y là hai biến toàn cục.

**2. Biến cục bộ (local variables)**

***Định nghĩa:*** Là các biến được khai báo ở trong một hàm, chỉ có tác dụng cho tới hết khối lệnh mà nó thuộc vào và sẽ biến mất khi khối lệnh kết thúc. Khi khai báo các biến cục bộ, chúng sẽ nhận giá trị mặc định là những giá trị tùy ý sinh ra bởi chương trình. Do đó, chúng ta cần lưu ý luôn luôn khởi tạo giá trị ban đầu cho các biến cục bộ trước khi thực hiện tính toán liên quan tới biến đó (trừ khi ta sẽ nhập giá trị cho biến đó sau khi khai báo).

***Ví dụ:***

#include <iostream>

using namespace std;

int get\_sum1()

{

int a = 5, b = 6; // Hai biến cục bộ.

return a + b; // Trả về tổng a và b;

}

int get\_sum2()

{

int a = 4, b = 8; // Hai biến cục bộ.

return a + b;

}

int main()

{

cout << "Giá trị lần 1: " << get\_sum1() << endl;

cout << "Giá trị lần 2: " << get\_sum2();

}

***Kết quả:***

Giá trị lần 1: 11

Giá trị lần 2: 12

Ta thấy hai biến a và b ở hàm get\_sum1() là hoàn toàn độc lập so với hai biến a và b ở hàm get\_sum2(), mặc dù chúng có cùng tên. Lí do là vì a và b ở mỗi hàm đều là những biến cục bộ, khi kết thúc hàm thì chúng sẽ bị hủy đi, nên các hàm khác sẽ không nhầm lẫn những biến đó với nhau.

**VI. Nạp chồng hàm và nạp chồng toán tử**

**1. Nạp chồng hàm**

Ngôn ngữ C++ cho phép người dùng định nghĩa ra các hàm giống tên và chức năng với nhau, nhưng khác nhau về kiểu trả về hoặc khác nhau về tham số. Khi gọi hàm, chương trình sẽ dựa vào đặc điểm của tham số trong lời gọi để quyết định hàm nào sẽ được gọi. Các hàm trùng tên, tính năng tương tự nhưng khác về tham số như vậy được gọi là các ***hàm nạp chồng***. Hàm nạp chồng giúp lập trình viên tiết kiệm thời gian hơn khi không phải mất công nghĩ ra các tên hàm khác nhau cho nhiều biến thể của cùng một công việc.

***Ví dụ:*** Dưới đây định nghĩa hàm tong() nhưng với ba kiểu tham số khác nhau:

#include <bits/stdc++.h>

using namespace std;

int tong(int a, int b) // Tính tổng hai số nguyên.

{

return a + b;

}

int tong(int a, int b, int c) // Tính tổng ba số nguyên.

{

return a + b + c;

}

double tong(double a, double b) // Tính tổng hai số thực.

{

return a + b;

}

int main()

{

int a = 1, b = 2, c = 3;

cout << tong(a, b) << endl; // Sử dụng hàm thứ nhất.

cout << tong(a, b, c) << endl; // Sử dụng hàm thứ hai.

double x = 1.5, y = 2.0;

cout << tong(x, y); // Sử dụng hàm thứ ba.

return 0;

}

Biên dịch và chạy chương trình trên sẽ cho ra kết quả:

3

6

3.5

***Lưu ý:*** Các hàm nạp chồng buộc phải khác nhau hoàn toàn về danh sách tham số, chứ không thể chỉ khác nhau về kiểu trả về.

Việc sử dụng hàm nạp chồng rất hữu ích trong lập trình, bởi vì có rất nhiều kiểu dữ liệu sẽ phát sinh trong quá trình làm việc. Thay vì sử dụng ép kiểu khiến cho chương trình trở nên dài dòng và rối mắt, ta nên tạo ra các hàm nạp chồng cho mọi kiểu dữ liệu cần dùng. Rất nhiều hàm có sẵn của C++ được xây dựng bằng phương pháp nạp chồng hàm này.

**2. Nạp chồng toán tử**

Trên thực tế, các toán tử trong C++ chính là các hàm được viết sẵn. Khi người dùng sử dụng một toán tử nào đó, trình biên dịch sẽ gọi ra phiên bản tương ứng của toán tử đó dựa vào kiểu dữ liệu của toán hạng. Lấy ví dụ:

#include <iostream>

using namespace std;

int main()

{

int a = 3, b = 5;

cout << a + b << endl;

double x = 3.0, b = 5.0;

cout << x + y;

return 0;

}

Trong trình biên dịch đã chứa sẵn các phiên bản khác nhau của toán tử +. Đối với câu lệnh cout << a + b << endl;, ta có thể liên tưởng tới một lời gọi hàm operator + (a, b), với operator + là tên hàm và trả ra kết quả tương ứng là tổng của hai số nguyên.

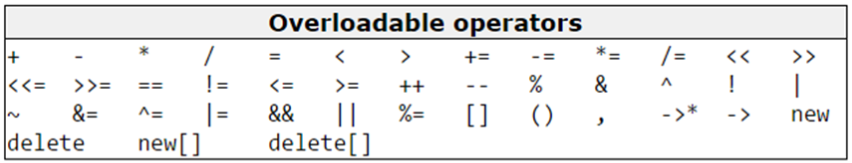
Đối với câu lệnh thứ hai: cout << x + y;, lời gọi hàm operator + (x, y) vẫn được thực hiện, nhưng lúc này tham số truyền vào lại là hai số thực. Nạp chồng hàm sẽ chỉ định trình biên dịch cần gọi ra phiên bản toán tử + mà nhận vào tham số là số thực và trả ra tổng của hai số thực.

Lập trình viên có thể nạp chồng hầu hết các toán tử trong C++. Cú pháp để nạp chồng toán tử hoàn toàn giống như nạp chồng hàm:

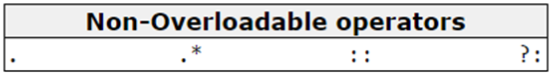
{Kiểu\_trả\_về} operator {Toán\_tử}({Danh\_sách\_tham\_số})

Trong đó, ***{Kiểu\_trả\_về}*** là một kiểu có sẵn hoặc một kiểu do người dùng tự định nghĩa. ***{Toán\_tử}*** là một toán tử sẵn có của C++. ***{Danh\_sách\_tham\_số}*** là các tham số với kiểu do người dùng tự định nghĩa (không được sử dụng kiểu có sẵn cho các tham số).

Dưới đây là danh sách các toán tử có thể nạp chồng trong C++:



Các toán tử không thể nạp chồng bao gồm:



**VII. Khuôn mẫu hàm (Function Template) trong C++**

Khuôn mẫu hàm là một phương thức dùng để tạo ra các hàm tổng quát. Đặt trường hợp chúng ta có một công việc nhưng phải xử lý trên rất nhiều kiểu dữ liệu khác nhau. Nếu như định nghĩa quá nhiều hàm nạp chồng cũng sẽ gây dài dòng và khó kiểm tra. Khuôn mẫu hàm sẽ giúp cho lập trình viên giải quyết việc đó, bằng cách tạo ra 1 hàm tổng quát và trình biên dịch sẽ tự chọn kiểu dữ liệu phù hợp cho nó trong quá trình biên dịch mã nguồn.

***Cú pháp:***

template <typename {Kiểu\_dữ\_liệu\_tổng\_quát}>

{Kiểu\_trả\_về} {Tên\_hàm}({Danh\_sách\_tham\_số})

{

{Thân\_hàm};

}

Ở đây, ***{Kiểu\_dữ\_liệu\_tổng\_quát}*** là một định danh bất kỳ do người dùng đặt ra để thay thế cho tên kiểu cụ thể. Từ khóa typename có thể thay thế bằng từ khóa class hoàn toàn không ảnh hưởng.

***Ví dụ:*** Dưới đây là một function template trả về giá trị lớn nhất trong hai giá trị ở các kiểu: số nguyên, số thực và chuỗi kí tự.

#include <iostream>

#include <string>

using namespace std;

template <typename cherry>

cherry const& Max (cherry const& a, cherry const& b)

{

return a < b ? b : a;

}

int main ()

{

int i = 15;

int j = 26;

cout << "Giá trị lớn nhất của (i, j) là: " << Max(i, j) << endl;

double f1 = 4.5;

double f2 = 14.2;

cout << "Giá trị lớn nhất của (f1, f2) là: " << Max(f1, f2) << endl;

string s1 = "Học lập trình";

string s2 = "Tại nhà";

cout << "Giá trị lớn nhất của (s1, s2) là: " << Max(s1, s2) << endl;

return 0;

}

Biên dịch và chạy chương trình trên sẽ cho ra kết quả:

Giá trị lớn nhất của (i, j) là: 26

Giá trị lớn nhất của (f1, f2) là: 14.2

Giá trị lớn nhất của (s1, s2) là: Tại nhà

**Bài 7 (phần 1): Mảng một chiều**

**I. Khái niệm về mảng**

Trong lập trình, đôi khi ta gặp tập dữ liệu gồm rất nhiều đối tượng có kiểu giống nhau. Lấy ví dụ:

* Danh sách điểm tổng kết của các học sinh trong lớp (một danh sách các số thực).
* Danh sách tên của một phòng thi (một danh sách các chuỗi kí tự). \dots…

Mọi ngôn ngữ lập trình đều cung cấp những kiểu dữ liệu có cấu trúc để lưu trữ các dạng dữ liệu như mô tả bên trên. Đối với C++, đó là ***mảng***, ***mảng động*** và ***danh sách liên kết***. Tuy nhiên với mục tiêu hướng tới lập trình thi đấu, chúng ta sẽ tập trung nghiên cứu ***mảng*** và ***mảng động***.

**II. Khai báo và khởi tạo mảng một chiều**

**1. Khai báo mảng một chiều**

Để khai báo một mảng trong C++, ta sử dụng cú pháp:

{Kiểu\_phần\_tử} {Tên\_mảng}[{Kích\_thước\_mảng}];

Trong đó, ***{Kiểu\_phần\_tử}*** là một kiểu dữ liệu nguyên thủy hoặc kiểu do người dùng tự định nghĩa - thể hiện kiểu dữ liệu của các phần tử trong mảng; ***{Tên\_mảng}*** là một định danh do người dùng đặt ra và không được trùng với từ khóa của hệ thống; ***{Kích\_thước\_mảng}*** là một số nguyên thể hiện kích thước tạo ra cho mảng. Giả sử kích thước được khởi tạo là N, thì hệ thống sẽ tạo ra một dãy gồm N ô nhớ liền nhau trong bộ nhớ để biểu thị cho mảng.

***Ví dụ:*** Khai báo một mảng số nguyên gồm 10 phần tử:

int a[10];

***Lưu ý:*** Nên tránh việc khai báo mảng là biến cục bộ, vì có thể gây ra tràn bộ nhớ (đây là kinh nghiệm thi cử). Với các mảng có kích thước nhỏ (dưới 1000) thì có thể khai báo cục bộ không sao, nhưng với các mảng kích thước lớn, kiểu dữ liệu lớn (như long long hay double) thì nên khai báo mảng là biến toàn cục sẽ an toàn hơn.

**2. Khởi tạo mảng một chiều**

Cũng giống như biến, mảng có thể được khởi tạo trước các giá trị khi khai báo. Số lượng phần tử khởi tạo không được phép vượt quá kích thước mảng đã khai báo. Nếu như kích thước mảng được để trống thì hệ thống sẽ tự tạo ra số ô nhớ vừa đủ để chứa các phần tử khởi tạo. Cú pháp khởi tạo mảng là:

{Kiểu\_phần\_tử| {Tên\_mảng}[{Kích\_thước\_mảng}] = {{Danh\_sách\_phần\_tử\_khởi\_tạo}};

Có nhiều cách khác nhau để khởi tạo mảng

* Khởi tạo một mảng với kích thước cố định:

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

* Khởi tạo một mảng có số phần tử khởi tạo ít hơn kích thước đã khai báo:

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

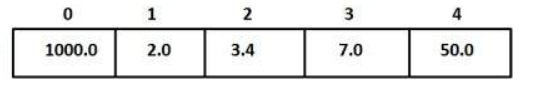
Trong trường hợp này, các phần tử chưa được khởi tạo sẽ nhận một giá trị bất kỳ nào đó. Nếu mảng được khai báo là biến toàn cục thì các phần tử trống sẽ nhận giá trị 0, nhưng nếu mảng là biến cục bộ thì các phần tử trống sẽ nhận những giá trị tùy ý. Thông thường khi khai báo mảng chúng ta nên khai báo biến toàn cục thay vì biến cục bộ để tránh những nhầm lẫn không đáng có trong khi tính toán vì những phần tử mang giá trị tùy ý.

* Khởi tạo mảng không có kích thước:

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

Đối với trường hợp này, mảng sẽ có số vị trí bằng đúng với số phần tử được khởi tạo.

* Minh họa mảng một chiều bằng hình vẽ:



**III. Các thao tác cơ bản trên mảng một chiều**

**1. Truy cập các phần tử trong mảng**

Các phần tử trong mảng đều được đánh số, bắt đầu từ 0 tới N-1 (N là kích thước mảng). Để truy cập và sử dụng một phần tử trong mảng, ta sử dụng toán tử [] với cú pháp:

{Tên\_mảng}[{Chỉ\_số\_phần\_tử}];

Mỗi phần tử của mảng khi được truy cập sẽ trở thành giống như một biến đơn, có thể sử dụng để tính toán, kết hợp cùng các câu lệnh và toán tử.

***Ví dụ:***

* Gán giá trị cho một phần tử của mảng:
* a[50] = 10;
* Lấy giá trị của mảng gán cho một biến:
* int value = a[50];

**2. Duyệt các phần tử của mảng**

Để duyệt qua tất cả các phần tử trong một mảng, ta có thể sử dụng vòng lặp kết hợp với toán tử [] để truy cập vào từng phần tử trong mảng. Cú pháp tổng quát như sau:

for ({Biến\_đếm} = {Chỉ\_số\_đầu}; {Biến\_đếm} <= {Chỉ\_số\_cuối}; {Tăng\_giảm\_biến\_đếm})

{Các\_thao\_tác\_truy\_cập\_phần\_tử};

Chẳng hạn, nếu cần duyệt qua và in ra các phần tử trong một mảng A gồm 4 phần tử, ta có thể viết:

for (int i = 0; i < 4; ++i)

cout << a[i] << endl;

Để duyệt ngược mảng hay duyệt một đoạn phần tử trên mảng, chúng ta chỉ cần biến đổi vòng lặp đi một chút là được. Ngoài ra, vòng lặp while cũng có thể được sử dụng để duyệt qua mảng.

**3. Nhập dữ liệu vào mảng**

Trong trường hợp cần yêu cầu nhập vào giá trị cho một mảng gồm N phần tử, ta có thể làm như sau:

int a[N];

for (int i = 0; i < N; ++i)

cin >> a[i];

Hoặc nếu muốn cho mảng bắt đầu từ vị trí 1 thì khai báo tăng kích thước thêm 1 đơn vị:

int a[N + 1];

for (int i = 1; i <= N; ++i)

cin >> a[i];

***Ví dụ cụ thể:*** Dưới đây minh họa một chương trình nhập vào một mảng gồm N số nguyên sau đó in ra mảng theo thứ tự ngược lại.

#include <iostream>

using namespace std;

int a[100]; // Khai báo mảng là biến toàn cục.

int main()

{

int N;

cin >> N;

for (int i = 1; i <= N; ++i)

cin >> a[i];

cout << "Mảng in ngược lại: "

for (int i = N; i >= 1; --i)

cout << a[i] << ' ';

}

Giả sử nhập vào N = 4 và mảng a={1, 2, 3, 4}, kết quả chạy chương trình sẽ đưa ra như sau:

Mảng in ngược lại: 4 3 2 1

**4. Thêm giá trị vào cuối mảng**

Đôi khi chúng ta cần thêm các giá trị mới vào mảng trong quá trình tính toán, thường gặp nhất là thêm giá trị vào cuối mảng. Khi đó, ta sẽ sử dụng một biến đếm {cnt} để lưu số lượng phần tử hiện có trong mảng, sau đó tăng biến {cnt} lên và gán vị trí {cnt} trong mảng bằng phần tử cần thêm vào. Lúc này kích thước của mảng sẽ chính bằng {cnt}:

int a[100], cnt = 0;

void insert\_element(int x)

{

++cnt;

a[cnt] = x; // Có thể viết gọn là a[++cnt] = x;

}

Trong trường hợp cần thêm phần tử vào đầu hoặc giữa mảng, việc xử lý sẽ trở nên hơi khó khăn. Tất nhiên, sử dụng mảng ta vẫn có thể chèn được phần tử vào giữa.

**IV. Vài bài toán cơ bản trên mảng một chiều**

**1. Bài toán tìm kiếm tuần tự**

**Đề bài**

Cho một mảng số nguyên gồm n số nguyên a1, a2,..., an​ và một số nguyên x, hãy đếm số lần xuất hiện của x trong mảng?

***Input:***

* Dòng đầu tiên chứa số nguyên dương n - số lượng phần tử trong mảng ().
* Dòng thứ hai chứa n số nguyên a1, a2,..., an​ phân tách nhau bởi dấu cách ().

***Output:***

* Số nguyên duy nhất là số lượng phần tử bằng với giá trị x trong mảng.

***Sample Input:***

5 10

10 10 2 1 4

***Sample Output:***

2

**Ý tưởng**

Đây có thể nói là bài toán cơ bản nhất với mảng một chiều. Ta có thể giải rất đơn giản bằng một vòng lặp từ đầu tới cuối mảng, nếu phần tử nào của mảng có giá trị bằng x thì tăng một biến đếm lên 1 đơn vị.

**Cài đặt**

#include <bits/stdc++.h>

using namespace std;

int main()

{

int n, x;

cin >> n >> x;

int a[n];

for (int i = 0; i < n; ++i)

cin >> a[i];

int res = 0; // Biến đếm số phần tử X trong mảng.

for (int i = 0; i < n; ++i)

if (a[i] == x)

++res;

cout << res;

}

**2. Bài toán tính tổng mảng**

**Đề bài**

Cho một mảng số nguyên gồm n số nguyên a1, a2,..., an​, hãy tính tổng tất cả các phần tử trong mảng?

***Input:***

* Dòng đầu tiên chứa số nguyên dương n - số lượng phần tử trong mảng ().
* Dòng thứ hai chứa n số nguyên a1, a2,..., an​ phân tách nhau bởi dấu cách ().

***Output:***

* In ra số nguyên duy nhất là tổng các số trong mảng.

***Sample Input:***

5

1 2 3 4 5

***Sample Output:***

15

**Ý tưởng**

Sử dụng một biến {array\_sum} để lưu tổng các phần tử trong mảng. Dùng một vòng lặp từ đầu tới cuối mảng và cộng giá trị các phần tử vào biến đó.

Tuy nhiên lưu ý rằng tổng các phần tử có thể vượt quá phạm vi kiểu dữ liệu int, vì thế cần đặt kiểu dữ liệu cho biến {array\_sum} là long long.

**Cài đặt**

#include <bits/stdc++.h>

using namespace std;

int main()

{

int n;

cin >> n;

int a[n];

long long array\_sum = 0;

for (int i = 0; i < n; ++i)

{

cin >> a[i];

array\_sum += a[i];

}

cout << array\_sum;

}

**3. Bài toán tìm giá trị lớn nhất - giá trị nhỏ nhất trong mảng**

**Đề bài**

Cho một mảng số nguyên gồm n số nguyên a1, a2,..., an, hãy tìm số lớn nhất và số nhỏ nhất trong mảng?

***Input:***

* Dòng đầu tiên chứa số nguyên dương n - số lượng phần tử trong mảng ().
* Dòng thứ hai chứa n số nguyên a1, a2,..., an​ phân tách nhau bởi dấu cách ().

***Output:***

* Đưa ra hai số nguyên lần lượt là giá trị nhỏ nhất và giá trị lớn nhất trong mảng.

***Sample Input:***

5

-1 5 3 -10 8

***Sample Output:***

-10 8

**Ý tưởng**

Áp dụng một kĩ thuật gọi là ***kĩ thuật đặt cờ***. Ta gọi hai biến {min\_value} và {max value} lần lượt là giá trị nhỏ nhất và giá trị lớn nhất của mảng, ban đầu gán cả hai bằng số đầu tiên trong mảng. Sau đó duyệt một vòng lặp từ phần tử thứ hai tới cuối mảng và cập nhật giá trị min - max vào hai biến với mỗi phần tử duyệt đến.

**Cài đặt**

#include <bits/stdc++.h>

using namespace std;

int main()

{

int n;

cin >> n;

int a[n];

for (int i = 0; i < n; ++i)

cin >> a[i];

int min\_value = a[0], max\_value = a[0];

for (int i = 1; i < n; ++i)

{

if (a[i] < min\_value) // Tìm thấy phần tử khác nhỏ hơn min\_value.

min\_value = a[i];

if (a[i] > max\_value) // Tìm thấy phần tử khác lớn hơn min\_value;

max\_value = a[i];

}

cout << min\_value << ' ' << max\_value;

}

**Bài 7 (Phần 2): Mảng hai chiều trong C++ và truyền mảng vào hàm.**

**I. Mảng hai chiều trong C++**

**1. Khai báo và truy xuất**

Ngoài kiểu dữ liệu mảng một chiều, C++ hỗ trợ kiểu dữ liệu mảng từ hai chiều tới nhiều chiều. Mảng hai chiều là ví dụ rất trực quan và dễ tưởng tượng, ta có thể xem nó như một bảng hình chữ nhật gồm có M hàng và N cột. Cú pháp khai báo rất đơn giản:

{Kiểu\_phần\_tử} {Tên\_mảng}[{Số\_hàng}][{Số\_cột}];

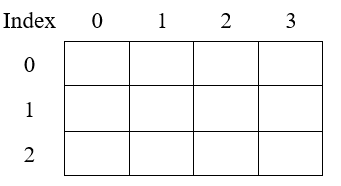
Khi đó, tổng số phần tử của mảng sẽ là {Số\_hàng} × {Số\_cột}. Chẳng hạn, khai báo một mảng hai chiều gồm 10 hàng và 12 cột chứa toàn số nguyên, ta viết như sau:

int a[10][12];

Các hàng và cột của mảng hai chiều đều sẽ được đánh số từ 0. Cách truy cập phần tử tương tự như ở mảng một chiều, chỉ cần dùng toán tử [] ở từng chiều để đưa ra một phần tử nào đó. Ví dụ, muốn truy cập phần tử ở hàng 3, cột 4, ta chỉ cần viết:

a[3][4];

Để tiện cho việc đánh số và biểu diễn trên hình, thường người ta sẽ quy ước đánh số các hàng từ trên xuống dưới và các cột từ trái qua phải:



Tuy nhiên, có một lưu ý nho nhỏ, đó là khi khai báo mảng hai chiều các bạn không nên khai báo bằng biến cục bộ. Lí do là vì, khi khai báo biến cục bộ thì bộ nhớ cấp phát cho biến sẽ lưu trong stack của máy tính, và đối với một số trình biên dịch có thể gây ra lỗi không đáng có!

**2. Khởi tạo mảng hai chiều**

Giống như mảng một chiều, mảng hai chiều cũng có thể khởi tạo trước giá trị. Cú pháp như sau:

{Kiểu\_phần\_tử} {Tên\_mảng}[{Số\_hàng}][{Số\_cột}] =

{

{{Danh\_sách\_phần\_tử\_của\_hàng\_0}};

{{Danh\_sách\_phần\_tử\_của\_hàng\_1}};

...

{{Danh\_sách\_phần\_tử\_của\_hàng\_cuối}};

};

***Ví dụ:*** Khởi tạo mảng hai chiều kích thước 3×4 gồm 12 số nguyên:

int a[3][4] =

{

{1, 2, 3, 4};

{5, 6, 7, 8};

{9, 10, 11, 12};

};

Ngoài cách khởi tạo mảng với số phần tử cố định, trên mảng hai chiều cũng có thể khởi tạo với các cách không khai báo số lượng hàng, cột hoặc không khởi tạo hết các phần tử giống như mảng một chiều. Bạn đọc có thể tự mình cài đặt các cách khởi tạo khác nhau để kiểm chứng. Trong C++ không chỉ có mảng hai chiều, mà còn có mảng nhiều chiều, nhưng sẽ khá khó tưởng tượng và cũng không thường xuyên sử dụng.

**3. Nhập xuất dữ liệu trên mảng hai chiều**

Ví dụ dưới đây sẽ minh hoạt một chương trình yêu cầu nhập vào một mảng hai chiều kích thước MxN và in ra toàn bộ mảng đó theo thứ tự hàng cột. Bạn đọc có thể áp dụng đúng phương pháp này cho việc nhập và truy xuất dữ liệu trên các mảng 3 chiều, 4 chiều,...:

#include <iostream>

using namespace std;

int main()

{

int M, N;

cin >> M >> N;

for (int i = 0; i < M; ++i)

for (int j = 0; j < N; ++j)

cin >> a[i][j];

cout << "Mảng đã nhập vào là: " << endl;

for (int i = 0; i < M; ++i)

{

for (int j = 0; j < N; ++j)

cout << a[i][j] << ' ';

cout << endl;

}

}

Giả sử nhập vào mảng kích thước 3×4 với các giá trị từ 1 tới 12, chạy chương trình sẽ thu được kết quả sau:

Mảng đã nhập vào là:

1 2 3 4

5 6 7 8

9 10 11 12

**II. Một vài bài toán với mảng hai chiều**

**1. Tìm giá trị lớn nhất trong mảng hai chiều**

**Đề bài**

Cho mảng hai chiều A gồm m hàng n cột, các hàng được đánh số từ 1 tới m từ trên xuống dưới, các cột được đánh số từ 1 tới n từ trái qua phải. Ô nằm trên giao của hàng i, cột j gọi là ô (i, j) và có chứa số nguyên ai,j​.

Hãy xác định giá trị lớn nhất trong mảng A?

***Input:***

* Dòng đầu tiên chứa hai số nguyên dương m, n (1 ≤ m, n ≤ 1000).
* m dòng tiếp theo, mỗi dòng chứa n số nguyên ai,j​ (ai,j ​≤ 109).

***Output:***

* In ra giá trị lớn nhất trong mảng A.

***Sample Input:***

4 5

1 2 3 4 5

-1 -2 0 3 5

10 4 -5 -10 6

4 4 4 4 4

***Sample Output:***

10

**Ý tưởng**

Sử dụng kĩ thuật đặt cờ, gán một biến res = a1,1​ để coi như phần tử lớn nhất trong mảng là a1,1. Sau đó duyệt qua tất cả các giá trị trong bảng, nếu phần tử nào lớn hơn res thì cập nhật lại res bằng phần tử đó.

Kết quả cuối cùng chính là res.

**Cài đặt**

#include <bits/stdc++.h>

using namespace std;

int a[1001][1001];

main()

{

int m, n;

cin >> m >> n;

for (int i = 1; i <= m; ++i)

for (int j = 1; j <= n; ++j)

cin >> a[i][j];

int res = a[1][1];

for (int i = 1; i <= m; ++i)

for (int j = 1; j <= n; ++j)

if (a[i][j] > res)

res = a[i][j];

cout << res;

}

**2. Tính tổng các phần tử trong mảng**

**Đề bài**

Cho mảng hai chiều A gồm m hàng n cột, các hàng được đánh số từ 1 tới m từ trên xuống dưới, các cột được đánh số từ 1 tới n từ trái qua phải. Ô nằm trên giao của hàng i, cột j gọi là ô (i, j) và có chứa số nguyên ai,j​.

Hãy tính tổng các phần tử trong mảng?

***Input:***

* Dòng đầu tiên chứa hai số nguyên dương m, n (1 ≤ *m*, *n* ≤ 1000).
* m dòng tiếp theo, mỗi dòng chứa n số nguyên ai,j​ (ai,j​ ≤ 109).

***Output:***

* Số nguyên duy nhất là tổng các phần tử trong mảng.

***Sample Input:***

3 3

1 2 3

4 5 6

7 8 9

***Sample Output:***

45

**Ý tưởng**

Giống như mảng một chiều, chúng ta chỉ cần sử dụng một biến sum*sum* để lưu tổng các phần tử trong mảng, rồi duyệt qua toàn bộ các phần tử và tính tổng của chúng.

**Cài đặt**

#include <bits/stdc++.h>

using namespace std;

int a[1001][1001];

main()

{

int m, n;

cin >> m >> n;

int sum = 0;

for (int i = 1; i <= m; ++i)

for (int j = 1; j <= n; ++j)

{

cin >> a[i][j];

sum += a[i][j];

}

cout << sum;

}

**3. Tổng đường chéo**

**Đề bài**

Cho mảng hai chiều dạng vuông A gồm m hàng m cột, các hàng được đánh số từ 1 tới m từ trên xuống dưới, các cột được đánh số từ 1 tới n từ trái qua phải. Ô nằm trên giao của hàng i, cột j gọi là ô (i, j) và có chứa số nguyên ai,j​.

Đường chéo chính của ma trận là đường chéo nối ô (1, 1) với ô (*m*, *m*). Đường chéo phụ của ma trận là đường chéo nối ô (1, *m*) với ô (*m*, 1).

Hãy tính tổng các số trên đường chéo chính và đường chéo phụ của ma trận vuông?

***Input:***

* Dòng đầu tiên chứa số nguyên dương m (1 ≤ *m* ≤ 1000).
* m dòng tiếp theo, mỗi dòng chứa m số nguyên cách nhau bởi dấu cách (1 ≤ ai,j​≤ 109).

***Output:***

* Hai số nguyên lần lượt là tổng đường chéo chính và đường chéo phụ của ma trận.

***Sample Input:***

3

1 2 1

3 1 8

2 5 4

***Sample Output:***

6 4

**Ý tưởng**

Một ô (i, j) sẽ thuộc đường chéo chính của ma trận nếu như i = j. Còn nếu như i = m - i + 1, thì ô đó sẽ thuộc đường chéo phụ của ma trận.

Ta duyệt qua các phần tử của ma trận và kết hợp lệnh if để tính tổng hai đường chéo.

**Cài đặt**

#include <bits/stdc++.h>

using namespace std;

int a[1001][1001];

int main()

{

int m;

cin >> m;

for (int i = 1; i <= m; ++i)

for (int j = 1; j <= m; ++j)

cin >> a[i][j];

long long main\_diagonal = 0, secondary\_diagonal = 0;

for (int i = 1; i <= N; ++i)

for (int j = 1; j <= N; ++j)

{

if (i == j)

main\_diagonal += a[i][j];

if (j == m - i + 1)

secondary\_diagonal += a[i][j];

}

cout << main\_diagonal << ' ' << secondary\_diagonal;

}

**III. Truyền mảng vào hàm như một tham số**

**1. Truyền mảng một chiều**

Mảng cũng có thể được truyền vào hàm giống như một tham số để tính toán. Có 3 cách để truyền mảng vào hàm: ***Sử dụng tham số mảng có kích cỡ***, ***sử dụng tham số mảng không có kích cỡ*** và ***sử dụng con trỏ***. Trong bài học này chúng ta sẽ tập trung vào hai cách đầu tiên.

***Cú pháp:***

* Truyền mảng có kích cỡ:

{Kiểu\_trả\_về} {Tên\_hàm}({Kiểu\_phần\_tử} {Tên\_mảng}[{Kích\_cỡ}])

***Ví dụ:***

int function1(int a[10])

{

{Thân\_hàm};

}

* Truyền mảng không kích cỡ:

{Kiểu\_trả\_về} {Tên\_hàm}({Kiểu\_phần\_tử} {Tên\_mảng}[])

Ví dụ:

int function2(int a[])

{

{Thân\_hàm};

}

Khác với việc truyền biến vào hàm, việc truyền mảng vào hàm không phân biệt ra tham trị hay tham chiếu. Khi truyền mảng, hệ thống luôn luôn truyền trực tiếp ***địa chỉ của phần tử đầu tiên*** của mảng ban đầu vào hàm, dẫn đến mọi thay đổi trên tham số đại diện trong hàm sẽ tác động đến mảng gốc bên ngoài, dù là theo cách nào trong số các cách trên. Vì vậy cần hết sức chú ý đến dữ liệu khi thực hiện truyền mảng vào hàm. Dưới đây là ví dụ cụ thể:

#include <bits/stdc++.h>

using namespace std;

void increase(int b[])

{

for (int i = 0; i <= 4; ++i)

++b[i];

}

int main()

{

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

increase(a);

cout << "Mảng sau khi tăng lên: ";

for (int i = 0; i <= 4; ++i)

cout << a[i] << ' ';

return 0;

}

Khi biên dịch và chạy chương trình này, ta sẽ thu được kết quả:

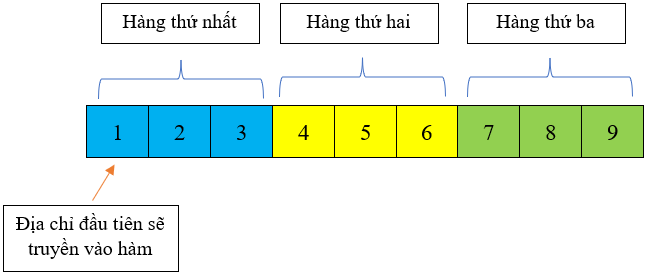
Mảng sau khi tăng lên: 2 3 4 5 6

Trong phần tham số của hàm void increase(int b[]), nếu ta thay int b[] bằng int b[5] thì cũng vẫn trả ra kết quả tương tự, do hệ thống sẽ không tạo ra một bản sao của mảng mà lấy trực tiếp địa chỉ của mảng gốc rồi truyền vào hàm.

Một lưu ý khác là C++ không cho phép trả về trực tiếp một mảng như là kết quả của hàm. Để trả ra kết quả cho hàm là một mảng, cần phải sử dụng ***con trỏ***, nhưng việc đó khá phức tạp. Để tránh việc phải trả ra kết quả là một mảng, ta nên khai báo các mảng là ***biến toàn cục*** hoặc sử dụng kiểu mảng động <vector>.

**2. Truyền mảng hai chiều**

Về bản chất, mảng hai chiều thực ra là một "mảng chứa các mảng", nghĩa là nó giống như một mảng một chiều nhưng mỗi phần tử lại là một mảng một chiều khác. Do đó, trên bộ nhớ, toàn bộ các phần tử của mảng hai chiều thực ra được viết liền kề với nhau, chứ không phải tạo thành một bảng số như chúng ta vẫn tưởng tượng. Chính vì thế, khi truyền mảng hai chiều vào hàm thì thực chất chương trình vẫn sẽ truyền địa chỉ của phần tử đầu tiên trong mảng hai chiều vào hàm. Chẳng hạn, với một mảng hai chiều 3×3 thì địa chỉ của các phần tử trên thanh RAM sẽ trông như thế này:



Mảng hai chiều cũng có thể được truyền vào hàm như mảng một chiều, tuy nhiên có một số lưu ý khi khai báo tham số hình thức. Có ba cách để truyền mảng hai chiều vào hàm:

* Cách 1: Khai báo cụ thể số hàng, số cột:

{Kiểu\_trả\_về} {Tên\_hàm}({Kiểu\_phần\_tử} {Tên\_mảng}[{Số\_hàng}][{Số\_cột}])

* Cách 2: Khai báo cụ thể số cột, số hàng bỏ trống:

{Kiểu\_trả\_về} {Tên\_hàm}({Kiểu\_phần\_tử} {Tên\_mảng}[][{Số\_cột}])

* Cách 3: Sử dụng từ khóa typedef để định nghĩa trước kiểu mảng:

typedef {Kiểu\_dữ\_liệu} {Tên\_thay\_thế}[{Số\_hàng}][{Số\_cột}];

// Khi truyền vào hàm.

{Kiểu\_trả\_về} {Tên\_hàm}({Tên\_thay\_thế} {Tên\_mảng});

Ví dụ, nếu truyền mảng 3×3 vào hàm void function() thì ta có thể khai báo tham số hình thức theo các cách sau đều được:

typedef int arr[100][100];

void function(int a[3][3]); // Cách 1.

void function(int a[][3]); // Cách 2.

void function(arr a); // Cách 3.

Thông thường, cách thứ 2 và 3 sẽ được ưu tiên sử dụng hơn. Lưu ý nhỏ, nếu truyền mảng từ hai chiều trở lên thì chỉ có chiều đầu tiên được phép để trống kích thước, còn các chiều sau đó đều phải xác định kích thước. Ví dụ, cách khai báo tham số int a[][] sẽ là không hợp lệ và bị báo lỗi khi biên dịch.

**Bài 8: Xâu kí tự (Phần 1) - Giới thiệu xâu kí tự trong C++**

**I. Dữ liệu dạng văn bản**

**1. Bảng kí tự ASCII**

***ASCII*** - tên đầy đủ là ***American Standard Code for Information Interchange*** - là một bảng mã chuẩn trao đổi thông tin Hoa Kỳ, bao gồm các kí tự và mã của chúng dựa trên bảng chữ cái Latin được dùng trong tiếng Anh hiện đại. Bảng mã này bao gồm 256 kí tự được đánh số hiệu thập phân từ 0 tới 255, thường được dùng để hiển thị văn bản trong máy tính và các thiết bị thông tin khác. Các kí tự mà chúng ta sử dụng trong lập trình đều nằm trong bảng mã này. Để làm việc dễ dàng trên máy tính, mỗi kí tự đều được mã hóa bởi những bit nhị phân 0 – 1 và quy đổi ra giá trị thập phân tương ứng để người dùng dễ thao tác hơn. Ví dụ, kí tự A có số hiệu thập phân là 65, kí tự z có số hiệu là 122,...

Trong bảng mã ***ASCII*** có những kí tự in được và không in được. Trong chương này, chúng ta sẽ làm việc với các kí tự in được, nhiều nhất là các kí tự chữ cái và chữ số. Nếu muốn tìm hiểu kĩ hơn về bảng mã ***ASCII*** đầy đủ, truy cập vào đường link sau:  
<https://vi.wikipedia.org/wiki/ASCII>

**2. Kí tự và chuỗi kí tự**

Trong máy tính, thông tin được biểu diễn ở dạng số và dạng phi số. Chúng ta đã quá quen thuộc với thông tin dạng số, vậy còn dạng phi số? Đó là văn bản, hình ảnh, âm thanh,... Đối với lập trình thi đấu, thông tin dạng văn bản cũng xuất hiện thường xuyên không kém dạng số, và các ngôn ngữ lập trình đều cung cấp những kiểu dữ liệu để lưu trữ thông tin dạng văn bản. Có hai loại dữ liệu dạng văn bản thường gặp nhất là kiểu kí tự và kiểu chuỗi kí tự (nhiều kí tự ghép lại với nhau).

Đối với kiểu kí tự, chúng ta có kiểu dữ liệu char để biểu diễn, còn chuỗi kí tự thì có hai cách khác nhau:

* Sử dụng một mảng gồm nhiều phần tử kiểu char.
* Sử dụng lớp chuỗi <string> đã được xây dựng sẵn trong C++. Cách này được ưa chuộng hơn vì thao tác dữ liệu tốt hơn.

**II. Lớp <string> trong C++**

Thư viện chuẩn của C++ cung cấp cho chúng ta một lớp <string> hỗ trợ việc lưu trữ các ***chuỗi kí tự (xâu kí tự)*** và rất nhiều các phương thức xử lý đi kèm. Khái niệm về ***lớp*** sẽ hiểu về nó khi học tới bài ***Cấu trúc và Lớp***. Còn bây giờ, có thể coi <string> như một kiểu dữ liệu nhưng hỗ trợ người dùng xử lý dữ liệu luôn.

**1. Khai báo và truy cập các phần tử chuỗi**

Để khai báo một chuỗi sử dụng lớp <string>, đầu tiên ta cần khai báo thư viện và không gian tên chứa nó bằng cú pháp:

#include <string>

using namespace std;

Sau đó, khai báo một chuỗi bằng cú pháp:

string {Tên\_chuỗi};

Vẫn như thường lệ, ***{Tên\_chuỗi}*** là một định danh do người dùng đặt ra, miễn là không trùng với từ khóa của hệ thống. Ta không cần khai báo độ dài của chuỗi, mà mỗi khi thêm một kí tự vào thì string sẽ tự động điều chỉnh độ dài của chuỗi cho vừa khớp với số lượng kí tự. Khi khai báo chuỗi, mặc định chuỗi đó sẽ là ***chuỗi rỗng*** (không có kí tự nào).

Các kí tự trong chuỗi sẽ được đánh số từ 0. Để truy cập một vị trí trong chuỗi (với điều kiện vị trí đó hiện đang có kí tự hoặc đã được khởi tạo), ta dùng cú pháp:

{Tên\_chuỗi}[{Vị\_trí}]

Khá giống với mảng, sau khi truy cập, mỗi vị trí trong chuỗi có thể được thao tác giống như một kí tự và kết hợp với các câu lệnh cũng như toán tử. Ví dụ, gán một biến c bằng kí tự ở vị trí số 2 của chuỗi s bằng cú pháp:

char c = s[2];

**2. Cách nhập xuất một chuỗi**

**2.1. Nhập xuất các biến kiểu <string>**

Khi nhập xuất một biến chuỗi kí tự, ta có thể coi chuỗi đó như một biến đơn và sử dụng hai câu lệnh cin và cout để nhập xuất. Cú pháp như sau:

cin >> {Tên\_biến\_chuỗi}; // Nhập vào một biến chuỗi.

cout << {Tên\_biến\_chuỗi}; // Viết ra một biến chuỗi.

Tuy nhiên, có một lưu ý khi sử dụng lệnh cin đối với string, đó là nếu như trong quá trình nhập liệu gặp phải dấu cách, thì việc nhập sẽ bị ngắt tại đó, cho dù người dùng có nhập thêm bao nhiêu kí tự đi chăng nữa thì chuỗi cũng sẽ chỉ lưu trữ phần ở phía trước dấu cách mà thôi. Cùng xem ví dụ dưới đây:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string name;

cin >> name;

cout << "Tên vừa nhập là: " << name;

}

Nếu người dùng nhập vào một tên là Vũ Quế Lâm, thì khi chạy chương trình ta sẽ thu được kết quả này:

Tên vừa nhập là: Vũ

Do đó, trong trường hợp cần đọc vào một chuỗi có cả dấu cách, ta sẽ sử dụng kết họp hai cú pháp:

getline(cin, {Tên\_chuỗi});

Lệnh getline() sẽ thu nhận cả dòng dữ liệu nhập vào, bao gồm cả những dấu cách. Nó sẽ dừng việc đọc lại khi gặp kí tự \n - tức là kí tự xuống dòng:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string name;

getline(cin, name);

cout << "Tên vừa nhập là: " << name;

}

Lúc này, với tên nhập vào là Vũ Quế Lâm, chạy chương trình sẽ thu được kết quả chính xác:

Tên vừa nhập là: Vũ Quế Lâm

**2.2. Nhập nhiều chuỗi kí tự hoặc chuyển đổi từ nhập số sang nhập chuỗi bằng getline()**

Nếu như chúng ta chỉ nhập một chuỗi kí tự duy nhất, thì sẽ không có điều gì đáng lưu tâm cả, bạn chỉ cần lựa chọn giữa cin và getline(cin) tùy vào việc chuỗi bạn nhập vào có dấu cách hay không. Tuy nhiên, khi dữ liệu đầu vào có nhiều chuỗi kí tự khác nhau, hoặc khi dữ liệu đầu vào bao gồm cả số và chuỗi, thì câu chuyện sẽ khác đi. Về bản chất, khi các bạn nhập bất cứ thứ gì vào từ bàn phím, chúng sẽ được đẩy vào bộ nhớ đệm, rồi hàm cin sẽ "đọc" dữ liệu ra từ bộ nhớ đệm rồi nạp vào biến.

Chẳng hạn, nếu như các bạn nhập vào một số là 123, rồi ấn phím Enter (chính là kí tự xuống dòng), thì 123 và kí tự \n sẽ được đẩy vào bộ nhớ đệm trước, sau đó hàm cin mới "quét qua" dữ liệu trong bộ nhớ đệm và đưa nó vào biến phía sau. Trong trường hợp biến phía sau là một biến kiểu số, thì chỉ có các chữ số mới được ghi nhận vào biến, còn những kí tự như \n sẽ bị bỏ qua, nhờ đó nên trong trường hợp đọc nhiều số khác nhau, chương trình vẫn sẽ phân tách đúng các số, dù các bạn dùng dấu cách hay dấu xuống dòng. Tuy nhiên, nếu như theo sau số nhập vào là một kí tự, hoặc một chuỗi kí tự, thì nó sẽ đọc được cả kí tự \n (vì kí tự \n chỉ bị bỏ qua chứ nó vẫn còn tồn tại trong bộ nhớ đệm), dẫn đến chuỗi kí tự sẽ bị đọc sai. Cùng xem một ví dụ dưới đây:

int main()

{

int id;

cin >> id; // Nhập mã số sinh viên.

string name;

getline(cin, name);

cout << id << endl << name;

}

Nếu như nhập vào dữ liệu là id = 1 và name = Nguyen Van A, thì khi chạy chương trình các bạn sẽ thấy kết quả in ra chỉ có như sau:

1

Nguyên do là vì, khi nhập xong số 1, chúng ta sẽ nhấn Enter hoặc dấu cách. Biến id chỉ có thể đọc được số 1, còn kí tự dấu cách hoặc xuống dòng vẫn nằm lại trong bộ nhớ đệm. Hàm getline(cin, name) tiếp theo sẽ đọc luôn cả những kí tự đó, dẫn đến hai trường hợp sau:

* Nếu kí tự còn lại là dấu cách, thì kí tự đó sẽ bị thêm vào đầu của chuỗi kí tự name.
* Nếu kí tự còn lại là dấu xuống dòng, thì hàm getline() sẽ dừng lại luôn và chuỗi kí tự name sẽ bị mất giá trị.

Tựu chung lại, dữ liệu của chúng ta sẽ bị sai! Vậy giải pháp là gì? Chúng ta cần xóa bộ nhớ đệm trước khi nhập một chuỗi (nên làm như vậy dù nhập một chuỗi hay nhiều chuỗi). Hàm cin trong C++ cung cấp phương thức cin.ignore() để làm điều đó. Cú pháp như sau:

cin.ignore(n, c);

Phương thức cin.ignore() sẽ xóa đi n kí tự trong bộ nhớ đệm cho tới khi gặp kí tự c thì dừng lại, và luồng nhập dữ liệu sẽ bắt đầu từ kí tự phía sau kí tự c. Nếu như ta để trống tham số thì chương trính sẽ tự động hiểu là chỉ xóa 1 kí tự trong bộ nhớ đệm. Điều này khá hữu ích khi nhập chuỗi ngay sau một số, vì chúng ta sẽ có thói quen dùng một dấu cách hoặc một dấu xuống dòng sau khi nhập số. Như vậy, đoạn code phía trên có thể sửa lại như sau:

int main()

{

int id;

cin >> id; // Nhập mã số sinh viên.

string name;

cin.ignore();

getline(cin, name);

cout << id << endl << name;

}

Lúc này, kết quả sẽ trở nên chính xác với bộ dữ liệu nhập vào:

1

Nguyen Van A

**2.3. Xuất ra các hằng chuỗi hoặc hằng kí tự**

Trong trường hợp cần viết các ***hằng chuỗi*** hoặc ***hằng kí tự*** trong C++, ta có quy tắc như sau:

* Nếu viết ra một kí tự, thì đặt kí tự đó trong cặp ngoặc '' hoặc "". Ví dụ, viết ra kí tự a thì có thể viết cout << 'a'; hoặc cout << "a"; đều được.
* Nếu viết ra một chỗi có nhiều hơn 1 kí tự, thì cần đặt chuỗi trong cặp dấu "". Ví dụ, khi muốn viết ra thông báo Bạn đã đăng nhập thành công thì phải viết là cout << "Bạn đã đăng nhập thành công";.

**III. Duyệt và tìm kiếm tuần tự trên chuỗi**

**1. Duyệt chuỗi**

Để duyệt qua các phần tử trên chuỗi, ta sử dụng một vòng lặp từ vị trí đầu tiên tới vị trí cuối cùng của chuỗi. Lớp <string> cung cấp cú pháp {Tên\_chuỗi}.size() để lấy độ dài của chuỗi, mà ta biết rằng các phần tử trong chuỗi được đánh số từ vị trí 0, nên cú pháp duyệt chuỗi như sau:

for ({Biến\_đếm} = {Vị\_trí\_đầu}; {Biến\_đếm} < {Tên\_chuỗi}.size(); {Tăng\_giảm\_biến\_đếm})

{

{Các\_câu\_lệnh};

}

Chẳng hạn, để duyệt các phần tử của một chuỗi s từ đầu tới cuối chuỗi, ta viết như sau:

for (int i = 0; i < s.size(); ++i)

Nếu muốn duyệt một đoạn nhỏ trên chuỗi, hoặc duyệt ngược từ cuối về đầu chuỗi chẳng hạn, chỉ cần biến đổi vòng lặp một chút xíu. Bạn đọc hãy thử tự suy nghĩ về vấn đề này nhé.

Ngoài ra, chúng ta còn có thể duyệt qua tất cả các phần tử trong chuỗi theo cú pháp duyệt trực tiếp phần tử như sau:

for (char {Tên\_biến\_kí\_tự}: {Tên\_chuỗi})

{

{Các\_câu\_lệnh};

}

Ví dụ, muốn duyệt qua mọi phần tử của chuỗi s bất kỳ theo cách này, ta viết:

for (char c: s)

Tuy nhiên, cách duyệt này chỉ có thể duyệt được mọi phần tử của chuỗi theo thứ tự từ trái qua phải, và buộc phải duyệt qua tất cả chuỗi. Vì thế nó không được ưu tiên như cách thứ nhất.

**2. Tìm kiếm tuần tự trên chuỗi**

Bài toán đặt ra rất đơn giản: Cho chuỗi kí tự s chỉ gồm toàn chữ cái latin in thường và một kí tự chữ cái c bất kỳ, hãy đếm số lượng kí tự c trong chuỗi s?

Bằng cách duyệt qua các phần tử trên chuỗi và áp dụng các toán tử, ta có thể giải quyết bài toán này như sau:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string s;

char c;

cin >> s;

cin >> c;

int cnt = 0; // Biến đếm số kí tự khác c trong chuỗi.

for (int i = 0; i < s.size(); ++i)

if (s[i] == c) // Nếu kí tự ở vị trí i khác c thì tăng biến đếm lên.

++cnt;

cout << cnt;

return 0;

}

**Bài 8 (phần 2): Xâu kí tự - Các thao tác xử lý trên xâu**

**IV. Các thao tác xử lý chuỗi kí tự**

**1. Phép so sánh**

Như đã đề cập ở phần I, máy tính sử dụng một bảng chữ cái gồm 256 kí tự được đánh số từ 0 tới 255, mỗi kí tự đều được mã hóa bằng những bit nhị phân, gọi là bảng mã **ASCII**. Hai chuỗi kí tự được so sánh với nhau dựa trên bảng mã này. Quy trình so sánh hai chuỗi kí tự X và Y trong C++ diễn ra như sau:

* Các kí tự được đánh số từ 0 ở mỗi chuỗi, sau đó tìm vị trí i đầu tiên sao cho Xi ≠ Yi. Khi đó, nếu Xi nằm sau Yi​ trong bảng mã ASCII thì chuỗi X sẽ lớn hơn chuỗi Y, ngược lại chuỗi Y lớn hơn chuỗi X.
* Trong trường hợp không tìm được vị trí i thỏa mãn thì chuỗi nào dài hơn sẽ là chuỗi lớn hơn.
* Nếu cả hai trường hợp trên không xảy ra thì kết luận chuỗi X bằng chuỗi Y.

Các toán tử >, <, <=, >=, ==, != có thể được sử dụng trực tiếp để so sánh hai kí tự hoặc hai chuỗi trong C++, tất nhiên là theo quy tắc nêu trên vì hệ thống đã có sẵn các toán tử so sánh nạp chồng cho kiểu chuỗi.

***Ví dụ 1:*** Chương trình dưới đây sẽ so sánh hai xâu kí tự và đưa ra xâu lớn hơn

#include <iostream>

#include <string>

using namespace std;

int main()

{

string s1 = "Tôi đi học";

string s2 = "Tôi đi ngủ";

cout << "Chuỗi lớn hơn là: "

if (s1 > s2)

cout << s1;

else

cout << s2;

}

Kết quả chạy chương trình sẽ là:

Chuỗi lớn hơn là: Tôi đi ngủ

Chuỗi Tôi đi học nhỏ hơn chuỗi Tôi đi ngủ vì kí tự n lớn hơn kí tự h trong bảng mã **ASCII**. Một điều rất thú vị trong so sánh chuỗi, đó là mặc dù số 100 lớn hơn số 90, nhưng chuỗi 100 sẽ nhỏ hơn chuỗi 90, vì kí tự 1 đứng trước kí tự 9 trong bảng mã **ASCII**. Do đó, khi so sánh các số ở dạng chuỗi cần hết sức chú ý (vấn đề này sẽ được nhắc tới trong chủ đề ***Xử lý số nguyên lớn*** ở phần lập trình thi đấu).

***Ví dụ 2:*** In ra các kí tự chữ cái latin in thường trên một dòng (không có dấu cách):

#include <iostream>

using namespace std;

int main()

{

for (char c = 'a'; c <= 'z'; ++c)

cout << c;

return 0;

}

Kết quả chạy chương trình:

abcdefghijklmnopqrstuvwxyz

**2. Phép nối chuỗi**

Khác với phép cộng ở kiểu số, toán tử + khi được kết hợp với hai chuỗi có ý nghĩa là nối hai chuỗi đó với nhau. Ví dụ dưới đây có thể làm rõ:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string s1 = "Ngày mai ";

string s2 = "tôi đi học.";

cout << s1 + s2;

}

Kết quả chạy chương trình sẽ là:

Ngày mai tôi đi học.

***Lưu ý:***

* Đặc điểm của phép cộng chuỗi là chuỗi đứng sau toán tử + sẽ được nối vào phía sau của chuỗi đứng trước toán tử +. Ví dụ, nếu như ta viết cout << s2 + s1; ở chương trình trên, thì kết quả sẽ trả ra là tôi đi học.Ngày mai thay vì Ngày mai tôi đi học.
* Phép nối chuỗi bản chất là tạo ra một bản sao của chuỗi ban đầu, nối bản sao đó với chuỗi cần nối rồi gán ngược trở lại chuỗi ban đầu. Vì vậy, phép nối bằng toán tử + sẽ có thời gian chạy khá lâu trong các trường hợp chuỗi dài, cần hết sức lưu ý khi sử dụng.

**3. Lấy số hiệu trong bảng mã ASCII của một kí tự**

Bằng kĩ thuật ép kiểu, ta có thể xác định được số thứ tự trong bảng mã ASCII của một kí tự c bất kỳ, với c là một biến kí tự hoặc hằng kí tự. Cú pháp như sau:

(int)(c);

Nếu c là một hằng kí tự thì ta cần đặt nó trong cặp dấu '', còn nếu là biến kí tự thì không cần. Ví dụ, muốn biết số thứ tự của kí tự a, ta có câu lệnh(int)('a') sẽ trả ra kết quả 97, còn nếu muốn biết số thứ tự của một biến kí tự c, thì chỉ cần viết (int)(c) thôi. Số hiệu của kí tự phải được sử dụng trong các câu lệnh chứ không được đặt nó đứng đơn lẻ.

Hoàn toàn tương tự, ta có thể xác định được kí tự ứng với số hiệu x trong bảng mã **ASCII** bằng cú pháp ép kiểu char ngược lại:

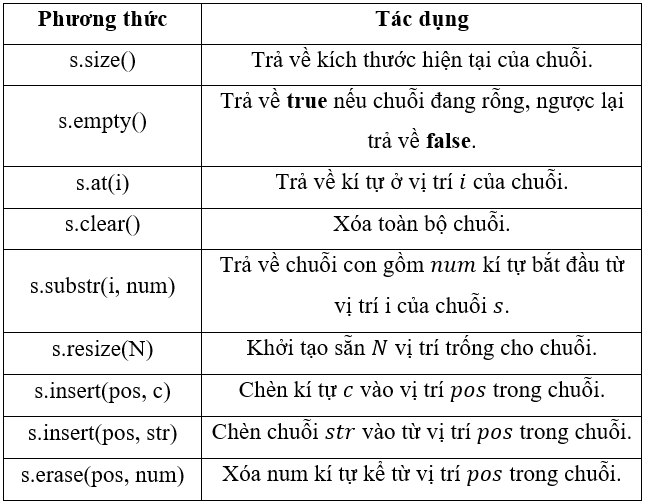
(char)(x);

Chẳng hạn, câu lệnh cout << (char)(48); sẽ trả ra kết quả là kí tự chữ số 0.

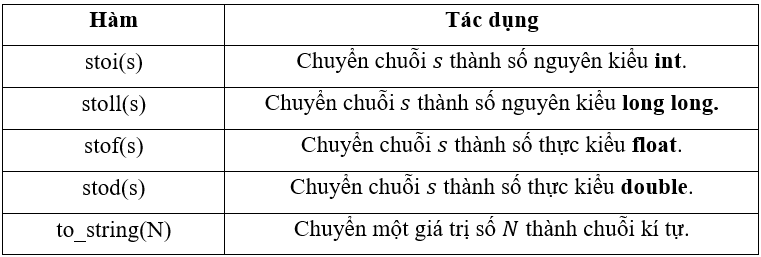
Có rất nhiều bài tập ứng dụng phần lấy số hiệu kí tự này, chẳng hạn như đổi từ kí tự số sang số đếm được, hay đổi chữ in hoa thành in thường và ngược lại,...

**5. Các hàm xử lý chuỗi có sẵn trong thư viện của C++**

Giả sử ta khai báo một chuỗi kí tự s bằng cú pháp: string s;. Bảng dưới đây liệt kê những phương thức xử lý dữ liệu thường dùng nhất, được hỗ trợ sẵn trong lớp <string> dành cho chuỗi s:



Thư viện <string> cũng cung cấp các hàm liên quan tới chuyển đổi giữa chuỗi - số ở bảng dưới đây:



Ngoài ra còn rất nhiều phương thức khác được xây dựng sẵn để hỗ trợ người dùng, ta có thể tra cứu ở địa chỉ: [**Lớp <string> trong C++**](https://www.cplusplus.com/reference/string/).

**6. Xóa các kí tự trong chuỗi:**

Như đã thấy ở mục 5, khi cần xóa một kí tự hoặc một chuỗi con trong chuỗi ban đầu, ta có thể sử dụng hàm erase() của lớp <string>. Tuy nhiên, khi xóa các kí tự trong chuỗi, thì sẽ xảy ra tình huống là các kí tự phía sau đoạn được xóa sẽ đẩy lên phía trên và nối liền với đoạn phía trước, dẫn đến số thứ tự của các kí tự trong chuỗi được đánh số lại. Nếu không cẩn thận khi xử lý sẽ rất dễ đưa ra kết quả sai. Cùng xem một ví dụ dưới đây, ta sẽ xóa các dấu cách trong một chuỗi và đưa ra chuỗi đó sau khi xóa:

#include <bits/stdc++.h>

using namespace std;

int main()

{

string s;

getline(cin, s);

for (int i = 0; i < s.size(); ++i)

if (s[i] == ' ')

s.erase(i, 1);

cout << s;

}

Nếu chạy đoạn chương trình trên với s*s* bằng ab c css e ad, kết quả trả về sẽ như sau:

abc cssead

Ta thấy kết quả hoàn toàn sai, điều này là do các kí tự bị đánh số lại, chiều dài chuỗi cũng thay đổi mỗi khi xóa dẫn đến vị trí của các dấu cách cũng thay đổi theo. Để khắc phục điểm này, khi xóa các kí tự hoặc chuỗi con trong một chuỗi, hãy xóa từ phải qua trái, và luôn đảm bảo rằng phần chuỗi phía sau đoạn bị xóa đi ở mỗi lần xóa sẽ không còn cần sử dụng đến nữa!

#include <bits/stdc++.h>

using namespace std;

int main()

{

string s;

getline(cin, s);

for (int i = s.size() - 1; i >= 0; --i)

if (s[i] == ' ')

s.erase(i, 1);

cout << s;

}

Với đoạn code mới này, kết quả sẽ trả ra hoàn toàn chính xác:

abccssead

**V. Chuỗi kí tự theo phong cách ngôn ngữ C (đọc thêm)**

Vì C++ có nền tảng là ngôn ngữ C, nên cũng hỗ trợ xử lý chuỗi kí tự theo phong cách ngôn ngữ C. Trong C, chuỗi kí tự được biểu diễn dưới dạng một mảng chứa các kí tự. Cú pháp để khai báo chuỗi phong cách C là:

char {Tên\_chuỗi}[{Kích\_thước\_chuỗi}];

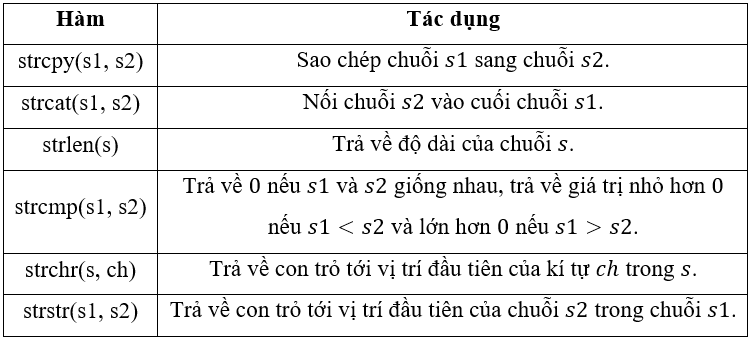
Các kí tự trong chuỗi kiểu C vẫn được đánh số từ 00. Vì nó là mảng nên cách sử dụng cũng giống như mảng thông thường. Ví dụ khai báo một chuỗi Hello theo phong cách C, ta có thể viết theo quy tắc khởi tạo mảng như sau:

char test\_str[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

hoặc viết theo quy tắc khởi tạo chuỗi, thì kích thước chuỗi sẽ tự động điều chỉnh cho khớp với số lượng kí tự:

char test\_str[] = "Hello";

Các hàm xử lý với chuỗi theo phong cách C được hỗ trợ không nhiều, được liệt kê ở bảng dưới đây. Nói chung ta nên ưu tiên sử dụng lớp <string> vì nó hỗ trợ xử lý chuỗi cực kỳ tốt, đặc biệt trong các bài toán lập trình thi đấu cần yêu cầu tốc độ lập trình nhanh.



**VI. Một số bài toán quen thuộc về xâu kí tự**

**1. Xâu đối xứng**

Một xâu kí tự được gọi là đối xứng nếu như khi viết ngược nó lại, ta vẫn thu được một xâu mới giống xâu ban đầu. Chẳng hạn, aba, aabbaa là các xâu đối xứng.

Cho trước một xâu kí tự s, hãy xác định xâu đó có phải đối xứng hay không?

***Input:***

* Một dòng duy nhất chứa xâu kí tự s chỉ bao gồm các chữ cái latin in thường.

***Ràng buộc:***

* Độ dài xâu s không vượt quá 106.

***Output:***

* In ra YES nếu như xâu s là xâu đối xứng, ngược lại in ra NO.

***Sample Input:***

aabbaa

***Sample Output:***

YES

**Ý tưởng**

Cách làm dễ nhất là sử dụng một xâu s\_1, lưu các kí tự của xâu s theo chiều ngược lại, rồi so sánh hai xâu. Cách làm này không phải một cách hay, vì phép cộng xâu trong C++ sẽ có độ phức tạp bằng độ dài của xâu mới, ngoài ra phép so sánh hai xâu cuối cùng cũng sẽ có độ phức tạp bằng đúng độ dài xâu. Vì thế, cách này chưa tối ưu.

Gọi n là độ dài của xâu kí tự và đánh số các kí tự trong xâu từ vị trí 0 tới vị trí n - 1. Ta nhận xét thấy, nếu một xâu là đối xứng, thì cặp kí tự thứ i và n - i - 1 sẽ giống nhau. Vì thế, chỉ cần sử dụng một vòng lặp duyệt i từ 0 tới  rồi kiểm tra cặp kí tự i và n - i - 1 có giống nhau hay không, nếu tồn tại một cặp khác nhau thì kết luận luôn xâu không phải đối xứng.

Bằng cách này, chúng ta giảm được số lần lặp xuống chỉ còn tối đa  lần.

**Cài đặt**

#include <bits/stdc++.h>

using namespace std;

bool is\_palindrome(string &s)

{

int n = s.size();

for (int i = 0; i < n; ++i)

if (s[i] != s[n - i - 1])

return false;

return true;

}

main()

{

string s;

cin >> s;

if (is\_palindrome(s))

cout << "YES";

else

cout << "NO";

}

**2. Chuẩn hóa xâu**

Cho một xâu kí tự s chỉ gồm các chữ cái latin in thường và in hoa cùng các dấu cách. Một xâu được gọi là chuẩn hóa nếu như nó thỏa mãn các điều kiện sau:

* Không bao gồm các dấu cách thừa ở đầu và cuối xâu.
* Gồm nhiều từ, mỗi từ bắt đầu bằng một chữ cái in hoa và theo sau là các chữ cái in thường.
* Giữa các từ phân tách nhau bằng đúng một dấu cách.

Hãy chuẩn hóa xâu s và đưa ra kết quả?

***Input:***

* Một dòng duy nhất chứa xâu kí tự s*s* chỉ bao gồm các chữ cái latin và dấu cách.

***Ràng buộc:***

* Độ dài xâu s không vượt quá 1000.

***Output:***

* In ra xâu s sau khi đã chuẩn hóa.

***Sample Input:***

Toi ten lA nhaT MINH

***Sample Output:***

Toi Ten La Nhat Minh

**Ý tưởng**

Cách làm hoàn toàn đơn giản như sau:

* Đầu tiên xóa các dấu cách thừa ở đầu và cuối xâu, sử dụng hàm erase() của thư viện <string>.
* Sau đó duyệt qua các kí tự của xâu s, lần lượt xử lý các trường hợp kí tự đó là chữ thường, chữ hoa hay dấu cách.
  + Nếu là chữ thường mà đứng ở đầu một từ thì phải viết hoa nó lên, còn là chữ hoa mà không đứng đầu một từ thì viết thường nó.
  + Nếu là dấu cách thì ta bỏ qua không quan tâm tới nó.
  + Nếu kí tự đó là kí tự đầu tiên của mỗi từ thì in thêm ra một dấu cách.
  + Cuối cùng in ra kí tự vừa xử lý.

Tuy nhiên, cần lưu ý trong bài này, xâu nhập vào có dấu cách, vì thế ta cần sử dụng getline() để nhập xâu.

**Cài đặt**

#include <bits/stdc++.h>

using namespace std;

void enter(string &s)

{

getline(cin, s);

}

void solution(string &s)

{

// Xóa các dấu cách thừa ở đầu chuỗi và cuối chuỗi.

while (s[0] == ' ')

s.erase(0, 1);

while (s.back() == ' ')

s.erase(s.size() - 1, 1);

for (int i = 0; i < (int)s.size(); ++i)

{

if (s[i] == ' ')

continue;

else if (s[i] == '.') // Kí tự dấu chấm kết thúc chuỗi.

cout << s[i];

else if (s[i - 1] == ' ' || i == 0) // Kí tự đầu tiên của mỗi từ.

{

if ('a' <= s[i] && s[i] <= 'z')

s[i] = (char)(s[i] - 32);

// Nếu không phải kí tự đầu tiên của chuỗi thì in ra

// một dấu cách để phân tách với từ phía trước.

if (i != 0)

cout << ' ';

cout << s[i];

}

// Nếu không phải kí tự đầu tiên của từ mà bị viết hoa thì viết thường nó lại.

else if (s[i - 1] != ' ')

{

if ('A' <= s[i] && s[i] <= 'Z')

s[i] = (char)(s[i] + 32);

cout << s[i];

}

}

}

int main()

{

string s;

enter(s);

solution(s);

return 0;

}

**Bài 10: Thư viện STL C++**

**I. Giới thiệu về thư viện chuẩn C++ (STL)**

**1. Lời mở đầu**

***Standard Template Library*** - thư viện Template chuẩn của C++ có lẽ là một trong những thứ mà các bạn học lập trình C++ được nghe tới rất nhiều. Nếu như bạn đọc còn nhớ, ở các bài trước mình đã giới thiệu về khái niệm ***Template - khuôn mẫu hàm***. STL chính là một thư viện chứa những template của cấu trúc dữ liệu cũng như thuật toán được xây dựng một cách tổng quát nhất, nhằm hỗ trợ cho người dùng trong quá trình lập trình. Có thể nói, điều làm nên sức mạnh của ngôn ngữ C++ chính là STL, thư viện này giúp việc lập trình có tính khái quát hóa rất cao, đồng thời giúp việc lập trình trở nên dễ dàng hơn nhiều.

Thư viện STL giúp người dùng thực hiện toàn bộ các công việc như vào ra dữ liệu, quản lý mảng động, xây dựng sẵn những cấu trúc dữ liệu cơ bản (ngăn xếp, hàng đợi, tập hợp,...) và bao gồm cả các giải thuật cơ bản như sắp xếp, tìm min - max, tính tổng, thậm chí là tìm ước chung lớn nhất chẳng hạn. Việc sử dụng STL là rất quan trọng đối với những bạn nào có định hướng tham gia những kỳ thi HSG Tin học hoặc nghiên cứu về thuật toán trên ngôn ngữ C++.

Ở đây, chỉ chia sẻ những thứ cơ bản nhất của STL kèm theo một vài template hữu dụng nhất của STL đối với các học sinh đang học lập trình C++ cơ bản. Những ứng dụng tiếp theo của STL sẽ được đề cập cụ thể trong từng chuyên đề của cấu trúc dữ liệu và giải thuật.

**2. Các thành phần của thư viện STL**

Thư viện STL vô cùng rộng lớn, gồm rất nhiều các template khác nhau. Nhưng ta có thể chia STL làm 4 phần chính:

* ***Containers Library:*** Thư viện chứa các cấu trúc dữ liệu mẫu như vector, stack, queue, deque, set, map, …
* ***Algorithm Library:*** Chứa các thuật toán viết sẵn để thao tác với dữ liệu.
* ***Iterator Library:*** Là các biến lặp, sử dụng để truy cập, duyệt các phần tử dữ liệu của các containers. Về cơ bản, nó giống như các biến chạy trên dữ liệu nhưng truy cập vào địa chỉ của dữ liệu. Dạng dễ hình dung nhất của iterator là con trỏ, nhưng chúng ta đã bỏ qua con trỏ nên iterator sẽ được đề cập trong từng containers.
* ***Numeric Library:*** Chứa các hàm toán học.

Từ những buổi học trước, các bạn cũng đã sử dụng nhiều thứ thuộc thư viện STL như các hàm nhập xuất của thư viện <iostream> là cin, cout hay thư viện <string> để xử lý chuỗi kí tự. Bài học này sẽ giới thiệu thêm một vài thư viện con hữu ích của STL, sẽ hỗ trợ rất tốt các bạn trong quá trình học lập trình C++.

Để sử dụng thư viện STL, các bạn cần khai báo không gian tên là using namespace std;, sau đó khai báo thư viện cần dùng bằng cú pháp #include <{Tên\_thư\_viện}>. Qua mỗi phiên bản của C++, lại có thêm những template mới được bổ sung vào STL, vì thế học sinh nên tham khảo thêm về thư viện này trên trang web [cplusplus.com](http://cplusplus.com/).

**II. Sử dụng thư viện STL**

**1. Thư viện <vector> (Mảng động)**

Thực tế, vector chỉ là một trong số các ***Containers Library (Thư viện lưu trữ)*** của STL, ngoài ra trong thư viện này còn có rất nhiều các containers khác nữa. Tuy nhiên đối với những học sinh đang học cơ bản, thì việc giới thiệu các cấu trúc dữ liệu khác trong thư viện này cũng không có tác dụng gì, sẽ mau chóng quên vì chưa sử dụng tới chúng. Do đó, trong này chỉ giới thiệu vector vì nó sẽ đi theo các bạn trong suốt quá trình học tập ngôn ngữ C++. Các containers khác sẽ được giới thiệu tới học sinh học chuyên môn tin khi học tới những bài học cụ thể liên quan tới chúng.

**1.1. Khai báo và truy cập phần tử**

Vector là kiểu dữ liệu mảng động - hỗ trợ người dùng lưu trữ các phần tử có cùng kiểu. Nhưng khác với mảng thông thường, vector rất linh hoạt và có nhiều phương thức để hỗ trợ người dùng. Nếu như mảng thông thường phải khai báo trước số lượng phần tử cố định (do đó còn gọi là ***mảng tĩnh***), thì vector sẽ tự động cập nhật các ô nhớ mới cho dữ liệu đưa vào, qua đó giảm thiểu tối đa sự lãng phí vùng nhớ. Cú pháp khai báo như sau:

#include <vector> // Khai báo thư viện chứa vector

using namespace std;

vector < {Kiểu\_phần\_tử} > {Tên\_vector};

***{Kiểu phần tử}*** là một kiểu dữ liệu bất kỳ, còn ***{Tên\_vector}*** là một định danh của người dùng. Ví dụ dưới đây khai báo một vector chứa số nguyên.

#include <vector>

using namespace std;

vector < int > integer\_list;

Khi mới khai báo, mặc định trong vector sẽ không có gì cả, trừ khi bạn khởi tạo trước giá trị cho nó. Ta có thể khởi tạo trước cho vector có bao nhiêu vị trí, thậm chí khởi tạo đồng loạt giá trị ban đầu cho tất cả các vị trí đó theo cách sau:

vector < {Kiểu\_phần\_tử} > {Tên\_vector}({Số\_vị\_trí}, {Giá\_trị});

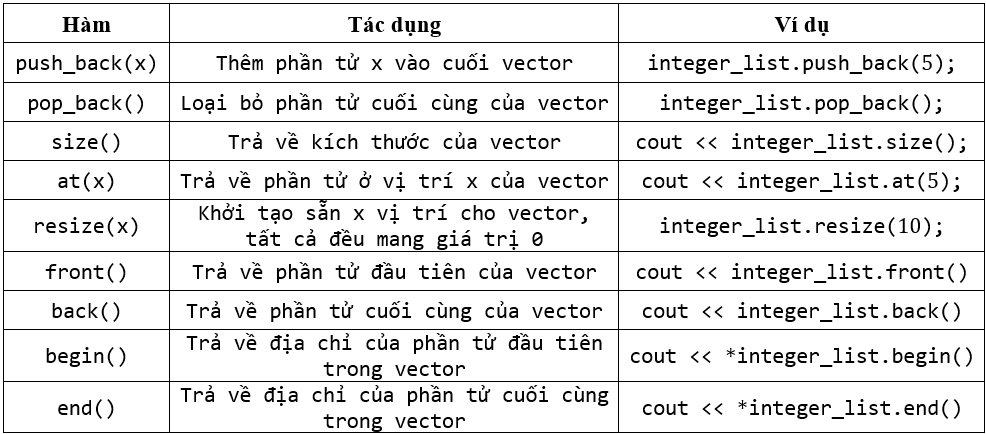
Các phần tử trong vector cũng sẽ mặc định đánh số từ vị trí 0. Ví dụ dưới đây khởi tạo một vector gồm 10 số 1:

vector < int > integer\_list(10, 1);

Để truy cập một phần tử trong vector, ta sử dụng toán tử [] giống như mảng (tất nhiên vị trí đó phải tồn tại trong vector rồi). Ví dụ, cú pháp integer\_list[5] sẽ truy cập tới phần tử ở vị trí thứ 5 của vector.

**1.2. Các hàm cung cấp sẵn của vector**

Thư viện vector đã được viết sẵn rất nhiều hàm hỗ trợ. Để sử dụng hàm có sẵn, ta sử dụng cú pháp {Tên\_vector}.{Tên hàm}. Có những hàm chỉ đứng riêng lẻ, và có những hàm lại chỉ đi cùng với các câu lệnh cụ thể. Bảng dưới đây liệt kê những hàm thường sử dụng của vector và tác dụng của chúng, kèm theo ví dụ về cách sử dụng chúng:



**1.3. Duyệt vector bằng chỉ số phần tử**

Muốn duyệt qua các phần tử của vector rất đơn giản, ta áp dụng vòng lặp giống như mảng. Giả sử vector a đã được khởi tạo với kích thước là N. Do vector được đánh số từ 0, nên các phần tử sẽ có số thứ tự lần lượt là 0, 1, 2,..., N-1. Dựa vào đó có thể duyệt các phần tử của vector bằng một vòng lặp qua các chỉ số như sau:

for (int i = {Chỉ\_số\_đầu}; i <= {Chỉ\_số\_cuối}; ++i)

{

{Các câu lệnh với a[i]};

}

Ví dụ, muốn in ra các phần tử của vector a = {1,2,3,4}, ta viết như sau:

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

cout << a[i] << ' ';

Kết quả chạy chương trình:

1 2 3 4

**1.4. Duyệt và truy cập vector bằng biến lặp (iterator)**

Mỗi containers trong STL đều hỗ trợ sẵn ***iterator (biến lặp)*** dùng để duyệt qua và truy cập các phần tử, đôi khi sẽ dùng để thao tác với các hàm thành viên của containers. Cú pháp khai báo như sau:

vector < |Kiểu phần tử| > :: iterator |Tên biến lặp|;

***Ví dụ:***

vector < int > :: iterator it\_1;

vector < double > :: iterator it\_2;

Khi được khai báo, các biến lặp sẽ chỉ duyệt qua được các phần tử có kiểu đúng với kiểu đã khai báo cho biến lặp. Đối với tất cả các containers khác trong STL, cách khai báo biến lặp hoàn toàn tương tự. Sau khi đã khai báo, để duyệt và in ra các phần tử bằng biến lặp, ta sử dụng vòng lặp như sau:

for ({Biến\_lặp} = {Địa\_chỉ\_đầu}; {Biến\_lặp} != {Địa chỉ cuối}; {Tăng\_giảm\_biến\_lặp})

cout << \*{Tên\_Biến\_Lặp};

Ngoài lệnh cout, biến lặp cũng có thể thao tác với các câu lệnh khác. Các biến lặp iterator có thể được sử dụng kèm với các phép toán như ++, --, !=, ==, = và +, - với các hằng số. Các phép toán này đã được nạp chồng sẵn trong thư viện STL.

***Ví dụ cụ thể:*** Dưới đây là chương trình sử dụng biến lặp để duyệt qua tất cả các phần tử của một vector và in ra các phần tử đó:

#include <iostream>

#include <vector>

using namespace std;

int main()

{

vector < int > integers;

integers.push\_back(1);

integers.push\_back(2);

integers.push\_back(3);

integers.push\_back(4); // v = {1, 2, 3, 4}.

vector < int > :: iterator it;

for (it = vector.begin(); it != vector.end(); ++it)

cout << \*it << ' ';

return 0;

}

Biên dịch và chạy chương trình trên sẽ cho ra kết quả:

1 2 3 4

**1.5. Duyệt vector bằng biến auto**

Từ phiên bản C++11 trở đi, sự xuất hiện của biến auto đã khiến cho việc duyệt phần tử trong vector trở nên đơn giản hơn rất nhiều. Ta có thể dùng một biến auto để duyệt toàn bộ các phần tử từ đầu vector tới cuối vector bằng cú pháp:

for (auto e: {Tên\_vector})

{

{Các\_câu\_lệnh};

}

Ví dụ:

vector < int > integers;

integers.push\_back(1);

integers.push\_back(2);

integers.push\_back(3);

integers.push\_back(4); // v = {1, 2, 3, 4}.

// Duyệt và in ra các phần tử trong vector.

for (auto e: integers)

cout << e << ' ';

Kết quả chạy chương trình:

1 2 3 4

Tuy nhiên, cách duyệt này có một nhược điểm là chỉ duyệt được các phần tử từ đầu tới cuối vector chứ không thể duyệt được một đoạn phần tử trên vector. Do đó nó ít khi được sử dụng.

**2. Kiểu pair**

Thư viện <utility> trong STL cung cấp một kiểu dữ liệu rất hữu dụng là pair, cho phép ghép hai biến hoặc hằng thành một biến gồm hai trường giá trị (có thể khác kiểu dữ liệu nhau). Để sử dụng, ta khai báo như sau:

#include <utility>

using namespace std;

pair <{Kiểu\_dữ\_liệu\_1}, {Kiểu\_dữ\_liệu\_2}> {Tên\_biến};

Ví dụ, có thể khai báo một biến kiểu pair lưu trữ hai thông tin về một sinh viên là mã số và tên bằng cách khai báo:

pair < int, string > student;

Sau khi khai báo, hai trường giá trị của biến pair có thể được truy cập thông qua hai từ khóa first và second. Cú pháp truy cập:

{Tên\_biến\_pair}.first

{Tên\_biến\_pair}.second

Sau khi truy cập vào các trường, ta có thể sử dụng kết hợp các câu lệnh với từng trường giống như một biến đơn.

pair < int, string > student;

cout << student.first << ' ' << student.second;

Ngoài ra, chúng ta cũng có thể sử dụng pair như một kiểu dữ liệu cho các phần tử trong mảng hay vector. Có rất nhiều bài toán mà pair sẽ thể hiện sự hữu ích của nó, ở phần bài tập chúng ta sẽ được biết thêm về cách sử dụng pair!

**3. Algorithm Library (Thư viện thuật toán)**

Thư viện thuật toán của STL chứa rất nhiều thuật toán viết sẵn từ đơn giản tới nâng cao. Để sử dụng thư viện này, trước tiên ta phải khai báo tên thư viện và tất nhiên là không gian tên std:

#include <algorithm>

using namespace std;

Dưới đây là vài thuật toán hữu ích và dễ sử dụng cho học sinh mới học C++.

**3.1. Hàm tìm min - max giữa hai số**

***Cú pháp:***

min(a, b); // Hàm tìm min giữa a và b.

max(a, b); // Hàm tìm max giữa a và b.

Hai hàm này vì có trả về giá trị nên buộc phải đi kèm với phép toán gán hoặc nằm trong một biểu thức - câu lệnh. Đặc biệt hai biến a và b phải có cùng kiểu dữ liệu thì mới sử dụng được hai hàm này.

***Ví dụ:***

#include <iostream>

#include <algorithm>

using namespace std;

int main()

{

int a = 5, b = 10;

cout << "Số nhỏ hơn là: " << min(a, b);

return 0;

}

Kết quả chạy chương trình:

Số nhỏ hơn là: 5

**3.2. Hàm hoán đổi giá trị**

Hàm swap() cho phép chúng ta hoán đổi giá trị của hai biến ***cùng kiểu*** trong C++ theo cú pháp:

swap({Biến\_thứ\_nhất}, {Biến\_thứ\_hai});

Ví dụ:

#include <iostream>

#include <algorithm>

using namespace std;

int main()

{

int a = 5, b = 10;

swap(a, b);

cout < "Hai số sau khi hoán đổi giá trị: " << a << ' ' << b;

return 0;

}

Kết quả chạy chương trình

Hai số sau khi hoán đổi giá trị: 10 5

**3.3. Truy cập địa chỉ của phần tử trong mảng và vector**

Các hàm thao tác trên đoạn trong thư viện <algorithm> đều sử dụng các tham số là các ***địa chỉ***. Mỗi biến được tạo ra trong khi lập trình đều có địa chỉ cụ thể trong bộ nhớ, và các ***biến địa chỉ (iterator)*** sẽ giúp chúng ta truy cập tới địa chỉ của biến đó. Đối với mảng và vector, ta có cách truy cập nhanh tới địa chỉ của các phần tử như sau:

* Đối với mảng:

{Tên\_mảng} + {Vị\_trí};

Chẳng hạn, a + 0 tức là địa chỉ của a\_0​, a + 1 là địa chỉ của a\_1,...

* Đối với vector:

{Tên\_vector}.begin() + {Vị\_trí};

Ví dụ, a.begin() là địa chỉ của phần tử đầu tiên trong vector, a.begin() + 1 là địa chỉ của phần tử thứ nhất trong vector,...Tuy nhiên, vector có một đặc điểm là luôn luôn tồn tại một vị trí cuối cùng là a.end() có tác dụng đánh dấu vector đã kết thúc (nhưng không mang giá trị gì cả), vì thế địa chỉ của phần tử này có thể hiểu là a.begin() + N, với N là kích thước của vector a.

Tiếp theo, chúng ta sẽ cùng tìm hiểu một số hàm dựng sẵn cho phép thao tác trên đoạn trong thư viện STL C++!

**3.4. Hàm tìm min - max trên một dãy số**

STL cung cấp một hàm để tìm min - max trên một dãy số, cụ thể là mảng hoặc vector. Cú pháp sử dụng như sau:

\*min\_element(l, r); // Tìm min đoạn.

\*max\_element(l, r); // Tìm max đoạn.

Trong đó, l và r là hai biến trỏ vào ***địa chỉ*** của phần tử đầu và phần tử cuối trong đoạn cần tìm min - max. Đối với mảng hay vector, cách truy cập tới ***địa chỉ*** của phần tử sẽ khác nhau. Hàm sẽ trả về giá trị min - max trong đoạn [l, r - 1].

***Ví dụ:***

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int main()

{

int a[5] = {1, 5, 2, 10, 14};

vector < int > v;

v.push\_back(-10);

v.push\_back(5);

v.push\_back(2);

v.push\_back(9); // v = {-10, 5, 2, 9}.

// In ra min của cả mảng a, vị trí cuối là a[4].

cout << \*min\_element(a, a + 4 + 1) << endl;

// In ra max của cả vector v.

cout << \*max\_element(v.begin(), v.end()) << endl;

// In ra min từ a[1] tới a[3].

cout << \*min\_element(a + 1, a + 3 + 1) << endl;

// In ra max từ v[0] tới v[2].

cout << \*max\_element(v.begin(), v.begin() + 2 + 1);

return 0;

}

Kết quả thu được là:

1

9

2

2

**3.5. Hàm tính tổng một dãy số**

STL cung cấp sẵn một hàm tính tổng các số trên một mảng hoặc vector. Các lưu ý chi tiết về địa chỉ đầu và cuối của đoạn cần tính tổng giống hệt với hai hàm tìm min - max trên đoạn vừa giới thiệu ở trên. Dưới đây chỉ cung cấp cú pháp tổng quát:

accumulate(l, r, x);

Trong đó, l và r là hai biến trỏ vào ***địa chỉ*** của phần tử đầu và phần tử cuối trong đoạn cần tính tổng, x là một hằng số hoặc biến kiểu số. Hàm sẽ tính tổng tất cả các phần tử thuộc đoạn [l, r - 1] rồi cộng vào x, sau đó trả ra tổng cuối. Tất nhiên, hàm này có trả về một kết quả nên luôn luôn phải đi kèm với toán tử hoặc nằm trong câu lệnh - biểu thức.

***Ví dụ:***

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int main()

{

int a[5] = {1, 5, 2, 10, 14};

vector < int > v;

v.push\_back(-10);

v.push\_back(5);

v.push\_back(2);

v.push\_back(9); // v = {-10, 5, 2, 9}.

// In ra tổng của cả mảng a, vị trí cuối là a[4].

cout << accumulate(a, a + 4 + 1, 0) << endl;

// In ra tổng của cả vector v.

cout << accumulate(v.begin(), v.end(), 0) << endl;

// In ra tổng từ a[1] tới a[3].

cout << accumulate(a + 1, a + 3 + 1, 0) << endl;

// In ra tổng từ v[0] tới v[2].

cout << accumulate(v.begin(), v.begin() + 2 + 1, 0);

return 0;

}

Kết quả thu được là:

32

6

17

-3

**3.6. Hàm sắp xếp**

**Hàm sắp xếp cơ bản**

Thư viện thuật toán cung cấp một hàm sắp xếp có thể sắp xếp các kiểu dữ liệu bao gồm số, kí tự, chuỗi kí tự và cả các kiểu dữ liệu tự định nghĩa của người dùng. Cú pháp như sau:

sort(l, r);

Trong đó, l và r là hai biến trỏ vào ***địa chỉ*** của phần tử đầu và phần tử cuối trong đoạn cần sắp xếp. Hàm sẽ sắp xếp toàn bộ các phần tử thuộc đoạn [l, r - 1]. Tuy nhiên hàm sort() sẽ đứng đơn lẻ chứ không đi kèm các câu lệnh khác. Mặc định hàm sort() sẽ sắp xếp các phần tử trong đoạn cần sắp xếp theo thứ tự tăng dần (số hoặc kí tự theo đúng quy tắc riêng của mỗi kiểu dữ liệu).

***Ví dụ:***

#include <bits/stdc++.h> // Khai báo luôn các thư viện cho ngắn gọn.

using namespace std;

int main()

{

int a[] = {5, 2, 10, 3, 1};

vector < int > v;

v.push\_back(-10);

v.push\_back(5);

v.push\_back(2);

v.push\_back(9); // v = {-10, 5, 2, 9}.

// Sắp xếp mảng và vector tăng dần.

sort(a, a + 4 + 1); // a = {1, 2, 3, 5, 10}.

sort(v.begin(), v.end()); // v = {-10, 2, 5, 9}.

// In ra kết quả sắp xếp.

cout << "Kết quả sắp xếp: " << endl;

for (int i = 0; i < 5; ++i)

cout << a[i] << ' ';

cout << endl;

for (int i = 0; i < 4; ++i)

cout << v[i] << ' ';

return 0;

}

Kết quả chạy chương trình trên như sau:

Kết quả sắp xếp:

1 2 3 5 10

-10 2 5 9

**Tùy biến việc sắp xếp theo ý thích**

Hàm sắp xếp thực tế còn có một tham số thứ ba, dùng để điều chỉnh việc sắp xếp theo ý muốn của người dùng. Cú pháp dạng này của hàm sắp xếp là:

sort(l, r, cmp);

Trong đó, cmp là một hàm kiểu boolean do người dùng tự định nghĩa, hoặc là một trong hai từ khóa thể hiện phép so sánh: less hoặc greater.

**Cách 1: Sử dụng 2 phép toán less và greater**

Hai từ khóa less và greater thể hiện cho hai phép toán sắp xếp tăng dần hoặc giảm dần (thực ra chính là thể hiện của các toán tử < và >), khi muốn điều chỉnh cách sắp xếp ta chỉ cần thêm hai phép toán này vào tham số thứ ba của hàm sắp xếp theo cú pháp:

sort(l, r, greater < {Kiểu\_phần\_tử} >());

sort(l, r, less < {Kiểu\_phần\_tử} >());

Trong đó, ***{Kiểu\_phần\_tử}*** là kiểu dữ liệu của các phần tử trong tập hợp cần sắp xếp.

***Ví dụ:***

#include <bits/stdc++.h> // Khai báo luôn các thư viện cho ngắn gọn.

using namespace std;

int main()

{

int a[] = {5, 2, 10, 3, 1};

sort(a, a + 4 + 1); // a = {1, 2, 3, 5, 10}.

// In ra kết quả sắp xếp.

cout << "Sắp xếp tăng dần: " << endl;

for (int i = 0; i < 5; ++i)

cout << a[i] << ' ';

cout << endl;

sort(a, a + 4 + 1, greater < int >()); // a = {10, 5, 3, 2, 1}.

cout << "Sắp xếp giảm dần: " << endl;

for (int i = 0; i < 4; ++i)

cout << a[i] << ' ';

return 0;

}

Kết quả chạy chương trình trên như sau:

Sắp xếp tăng dần:

1 2 3 5 10

Sắp xếp giảm dần:

10 5 3 2 1

***Lưu ý:***

* Phép so sánh mặc định của hàm sort là less, do đó nếu muốn sắp xếp tăng dần ta không cần thêm từ khóa less mà chỉ cần viết hàm sắp xếp và bỏ qua tham số thứ ba là được.
* Đối với tập hợp gồm các phần tử kiểu pair, hàm sắp xếp sẽ tự động sắp xếp ưu tiên theo trường giá trị first, nếu như hai phần tử trước sau có trường first bằng nhau thì mới xét tới trường second. Cụ thể, phép toán less sẽ ưu tiên sắp xếp các phần tử tăng dần theo trường giá trị first, nếu trường first bằng nhau thì sẽ sắp xếp tăng dần theo trường giá trị second; tương tự với phép toán greater.

**Cách 2: Sử dụng hàm sắp xếp tự định nghĩa**

Khi muốn sắp xếp theo những cách riêng, ví dụ như sắp xếp các số chẵn ra phía đầu, số lẻ ra phía cuối, hoặc khi kiểu dữ liệu của tập cần sắp xếp là những kiểu dữ liệu do người dùng tự định nghĩa, ta có thể tự viết ra một hàm cmp dùng làm tham số thứ ba cho hàm sort. Cú pháp như sau:

bool cmp({Tham\_số\_thứ\_nhất}, {Tham\_số\_thứ\_hai})

{

{Định\_nghĩa\_quan\_hệ\_so\_sánh\_giữa\_hai\_tham\_số};

}

Trong đó, ***{Tham\_số\_thứ\_nhất}*** đại diện cho phần tử đứng trước, ***{Tham\_số\_thứ\_hai}*** đại diện cho phần tử đứng sau trong dãy. Hàm sort() sẽ tự động sắp xếp lại các phần tử theo thứ tự bạn quy định giống như hai tham số này. Lấy ví dụ, nếu ta muốn sắp xếp một dãy số nguyên giảm dần, có thể viết như sau:

#include <bits/stdc++.h>

using namespace std;

bool cmp(int A, int B)

{

return A > B;

}

int main()

{

int a[] = {5, 2, 10, 3, 1};

// Sắp xếp mảng.

sort(a, a + 5, cmp); // a = {10, 5, 3, 2, 1}.

// In ra kết quả sắp xếp.

cout << "Kết quả sắp xếp: " << endl;

for (int i = 0; i < 5; ++i)

cout << a[i] << ' ';

return 0;

}

Biên dịch và chạy chương trình trên sẽ cho ra kết quả:

Kết quả sắp xếp:

10 5 3 2 1

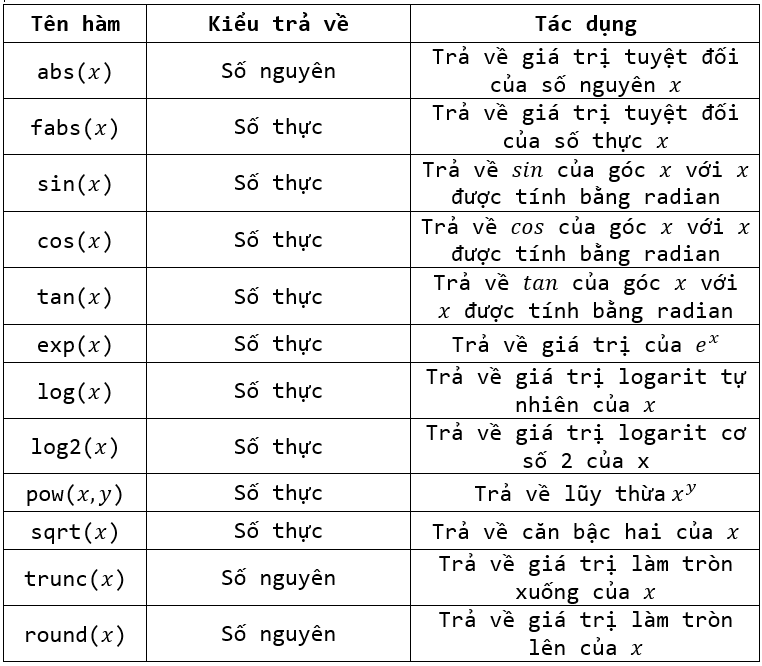
**4. Numeric Library (Thư viện số học):**

Thư viện số học trong STL được khai báo bằng cú pháp:

#include <cmath>

using namespace std;

Thư viện này cung cấp nhiều hàm toán học để xử lý số học trong quá trình tính toán. Bảng dưới đây là một số hàm thường sử dụng trong quá trình làm việc với C++:



---------- KẾT THÚC PHẦN CƠ BẢN ----------

