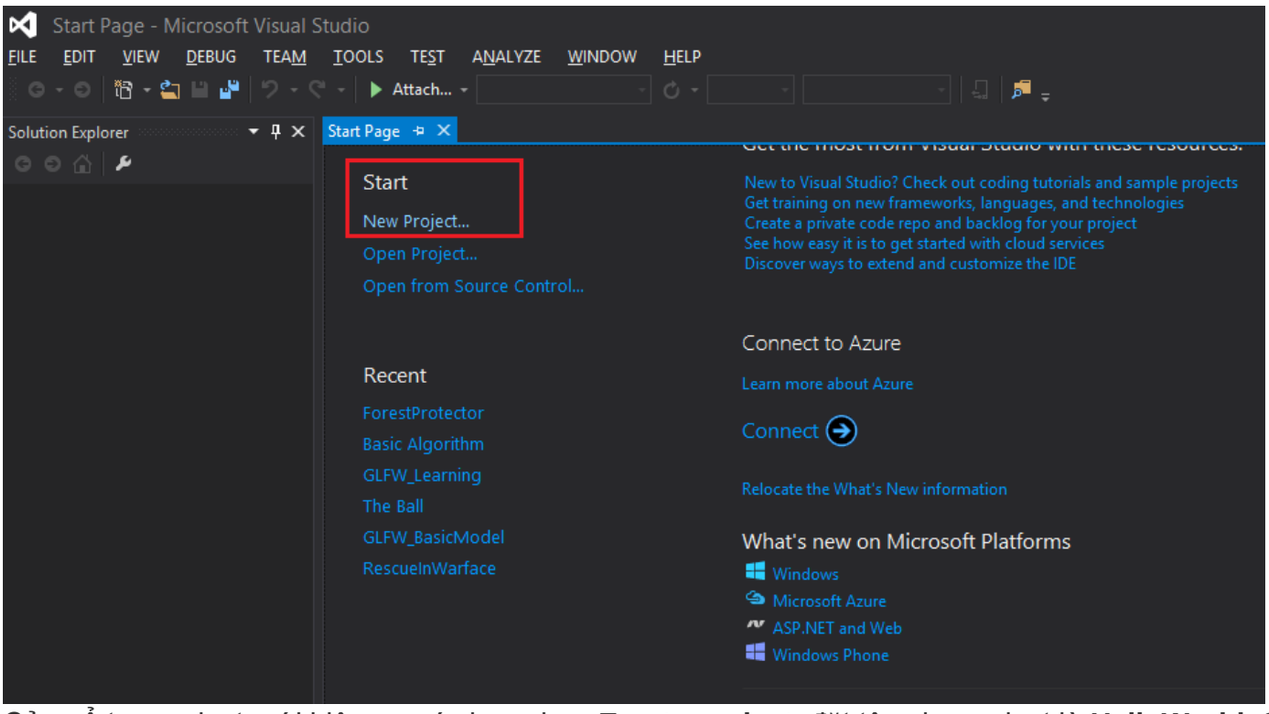
PHẦN 1: C++ CƠ BẢN

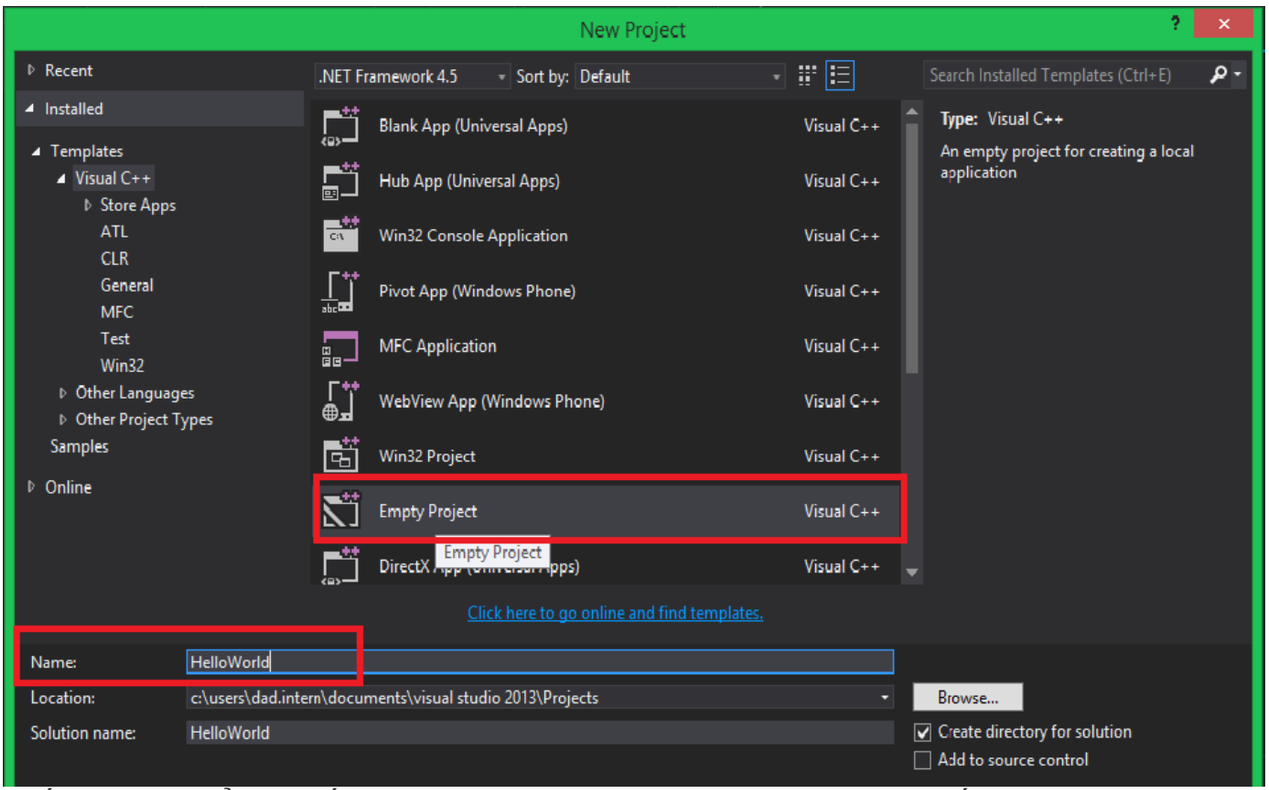
1.0 Viết chương trình đầu tiên

Ở các bài trước, chúng ta đã biết về quy trình làm việc để tạo ra một chương trình C++, những công cụ cần thiết và IDE mà chúng ta sẽ sử dụng để phát triển chương trình. Đến đây chắc các bạn cũng đang háo hức muốn bắt tay vào viết một cái gì đó. Trong bài này, chúng ta sẽ cùng viết một chương trình mà bất cứ lập trình viên C++ nào cũng từng trải qua. Một chương trình huyền thoại mang tên **"Hello World".**

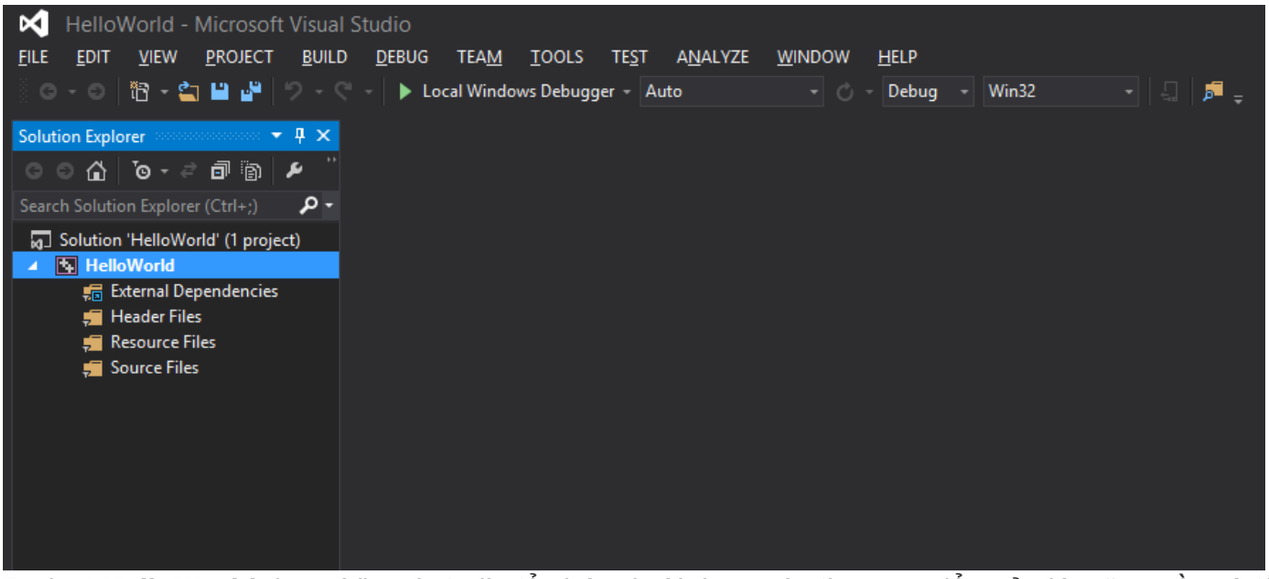
Để bắt đầu viết chương trình, chúng ta cùng mở IDE Visual studio 2015 lên và tạo một project. Tại giao diện **Start Page** của Visual studio, các bạn click chọn **New Project.**



Cửa sổ tạo project mới hiện ra, các bạn chọn **Empty project**, đặt tên cho project là **HelloWorld.** Sau đó, ở phần location các bạn có thể chọn đường dẫn thư mục để lưu project này vào.



Nhấn chọn OK để hoàn tất việc tạo project mới. Ngay khi Visual studio thiết lập project bạn vừa tạo. Bạn có thể nhìn vào cửa sổ **Solution Explorer** (mặc định là bên trái) để xem cấu trúc tổ chức của một project như thế nào.

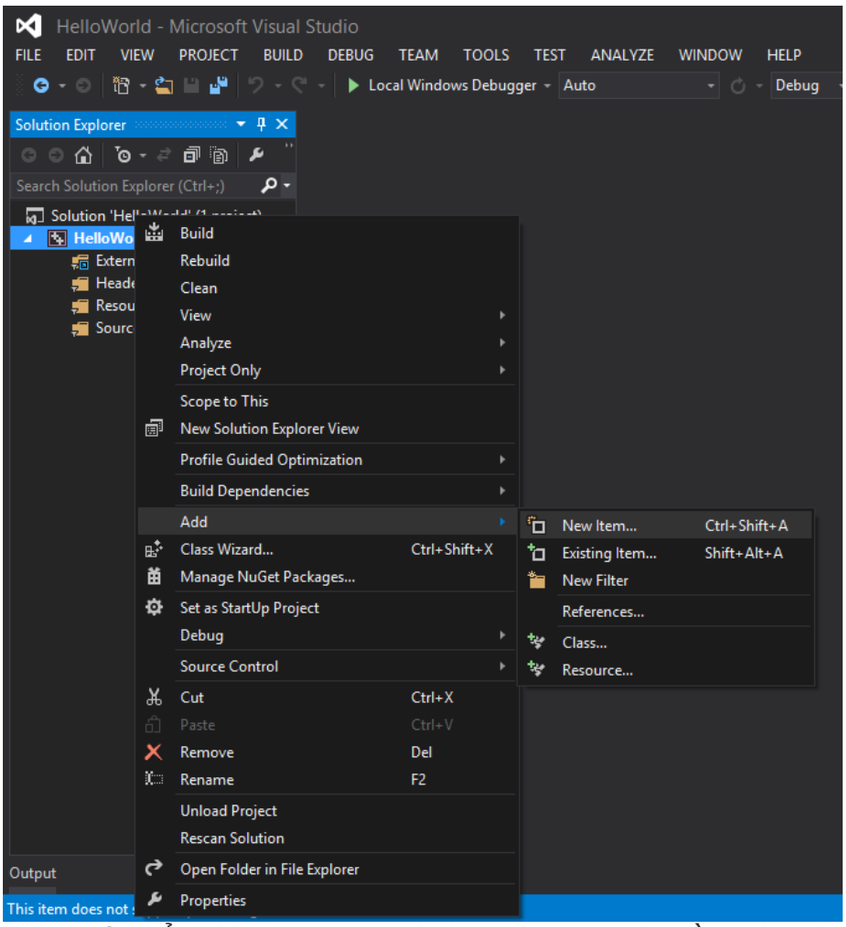


Project **HelloWorld** được Visual studio tổ chức dưới dạng cây thư mục để quản lý mã nguồn và tài nguyên.

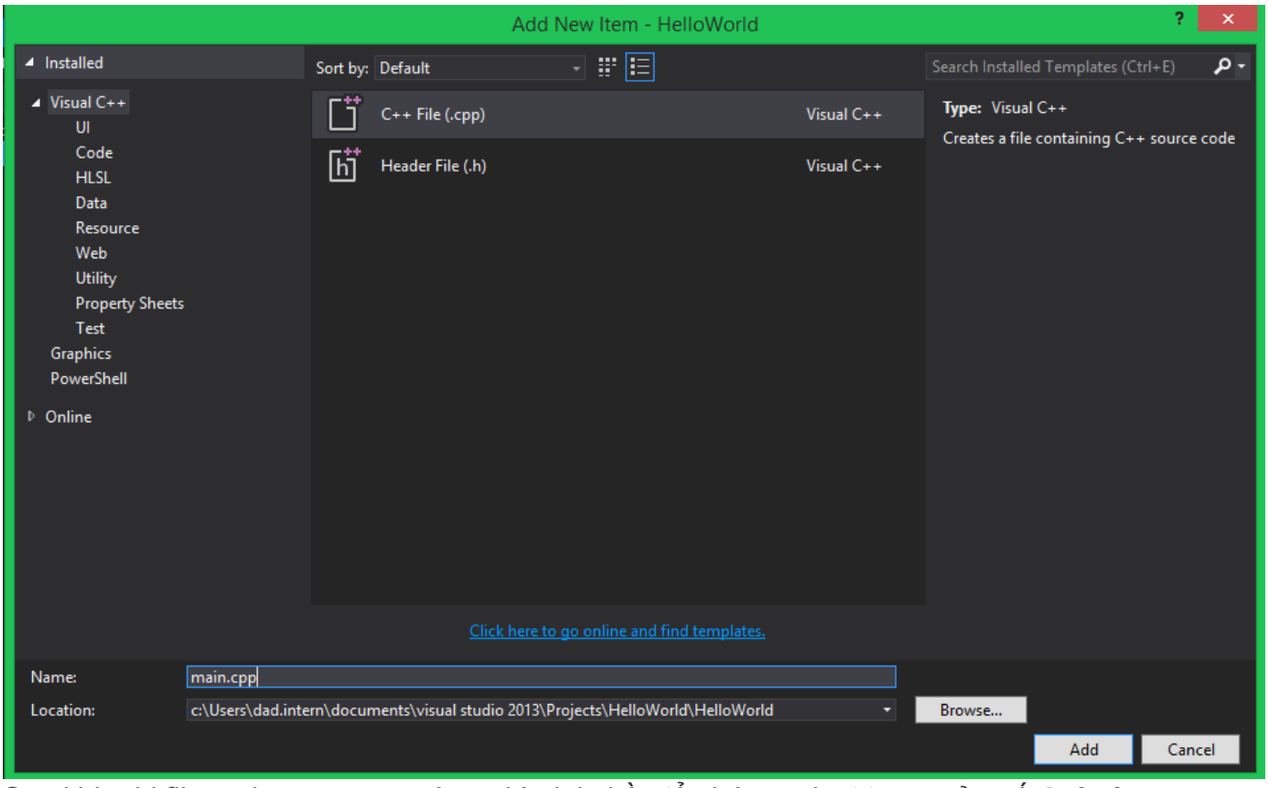
Trong project **HelloWorld**, hiện tại chúng ta quan tâm đến 2 phần chính:

- Header Files: dùng để chứa các phần khai báo class, khai báo hàm hoặc phần khai báo một số hằng số được sử dụng cho chương trình. Các file được chứa trong phần Header thường có phần đuôi mở rộng là .h, .hpp.
- Source Files: là nơi chứa các file định nghĩa các hàm, các class... Các file được đặt trong này thường có dạng .cpp.

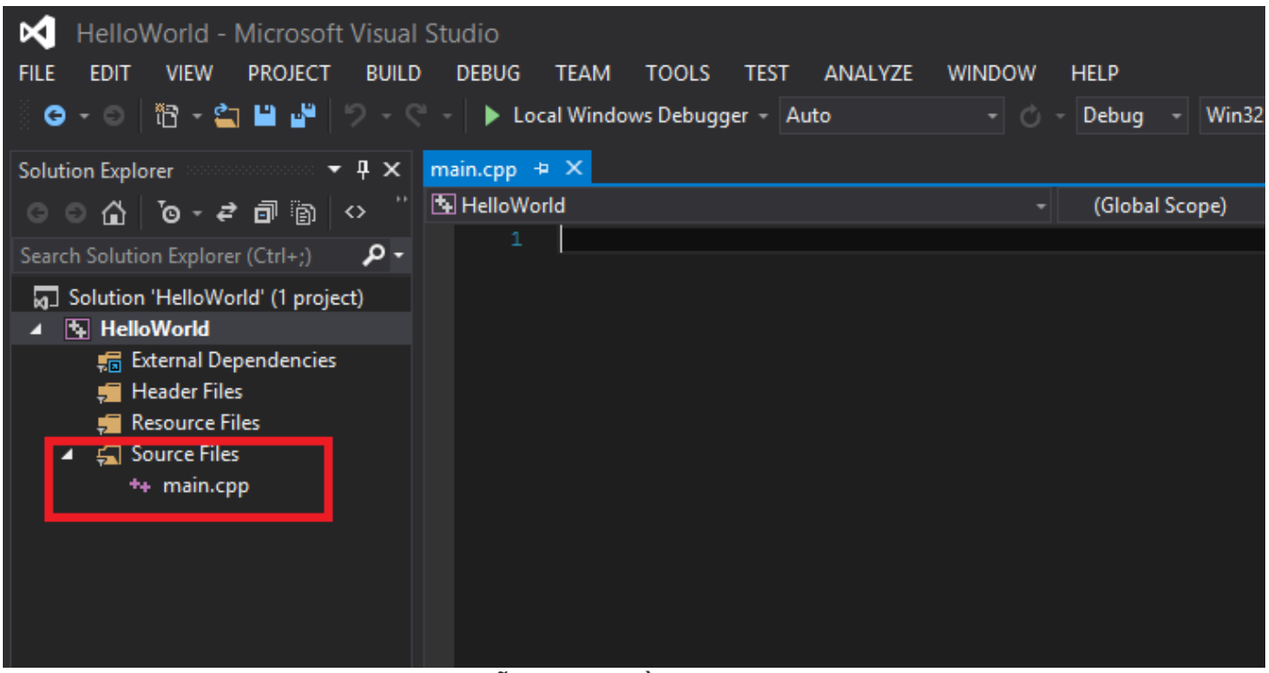
Bây giờ chúng ta cùng tạo file chương trình đầu tiên. Các bạn click chuột phải vào tên project ở trong khung **Solution Explorer**, chọn đến dòng **Add** và click chọn **New Item**...



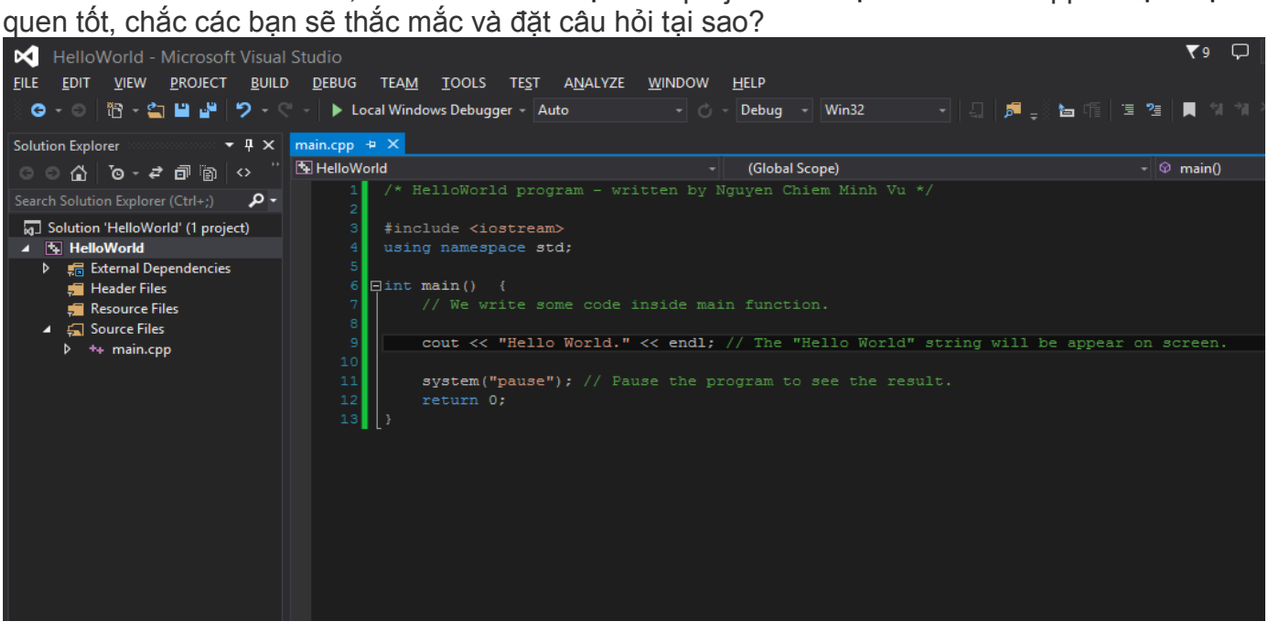
Trong cửa sổ **Add New Item**, các bạn chọn loại file cần thêm là C++ File (.cpp), đặt tên file ở textbox Name phía bên dưới. Để tạo một thói quen tốt, file này các bạn đặt tên là main.cpp sau đó click **Add**.



Sau khi add file main.cpp xong, cùng nhìn lại phần tổ chức project trong cửa số **Solution Explorer** xem điều gì đang xảy ra.

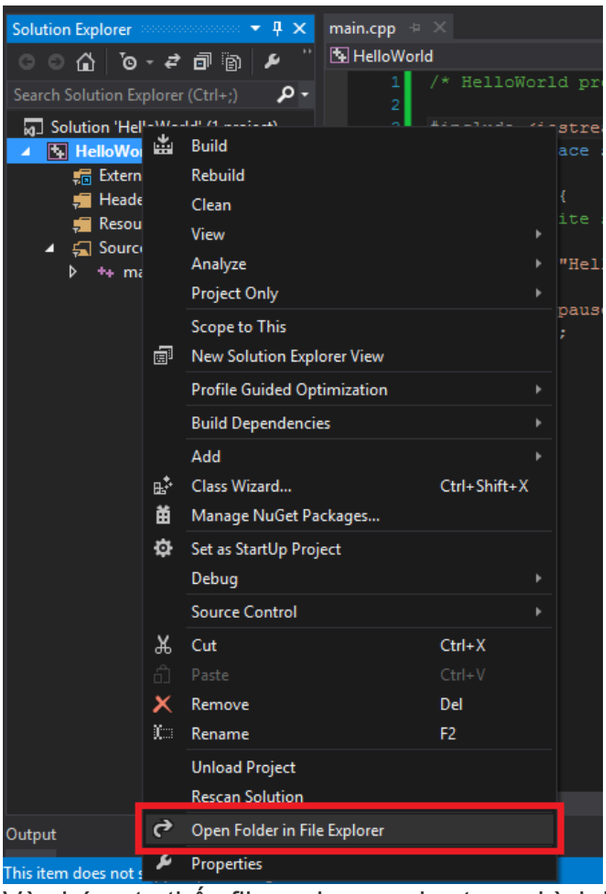


Chúng ta đã có thêm 1 file được đặt sẵn trong phần Source Files. Một file có đuôi mở rộng là .cpp luôn được đặt trong phần này. Phía bên phải là phần soạn thảo mã nguồn cho file main.cpp đã được mở sẵn. Như đã nói ở trên, file đầu tiên cần tạo cho project nên đặt tên là main.cpp để tạo một thói quen tốt, chắc các bạn sẽ thắc mắc và đặt câu hỏi tại sao?

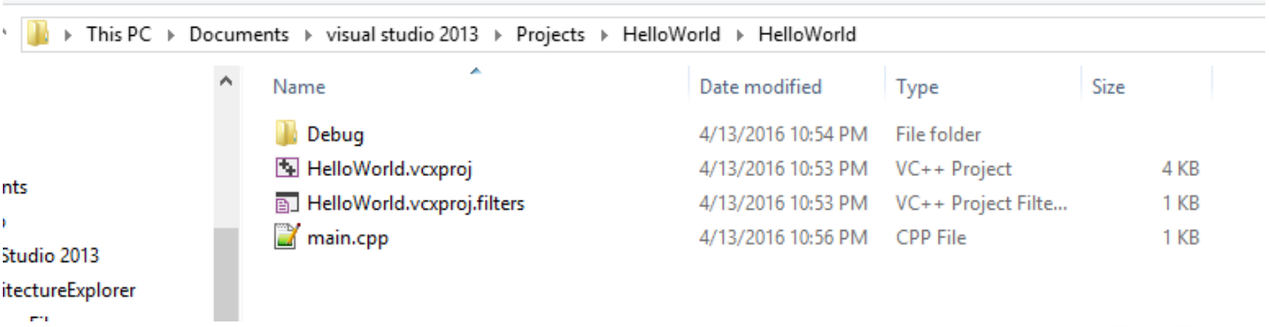


Trước hết, chúng ta cùng viết một ít mã lệnh cho **HelloWorld** program.

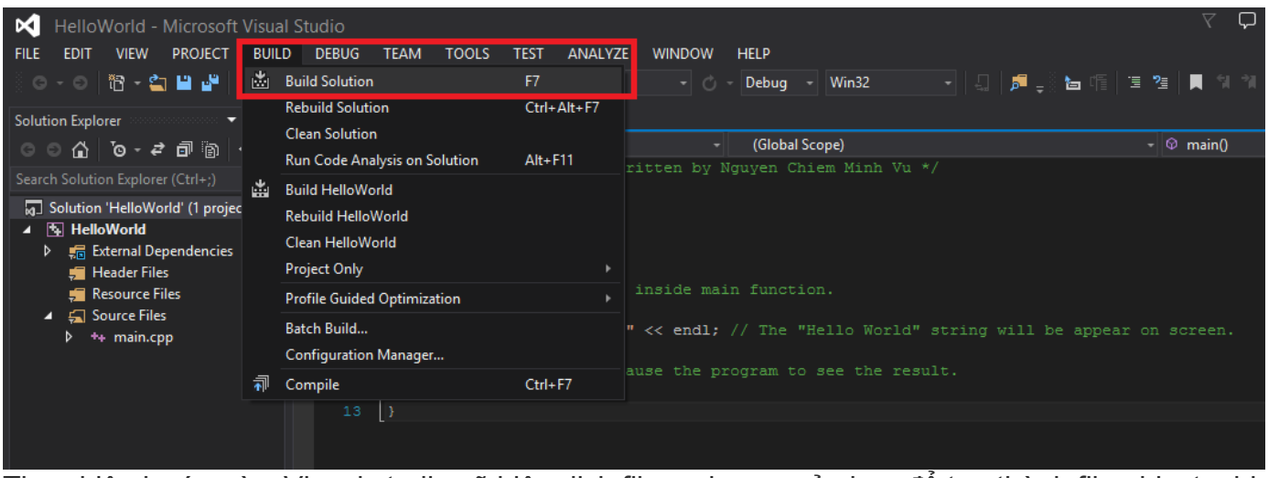
Nhấn tổ hợp phím Ctrl + S để lưu lại những gì bạn đã viết. Ở mức độ hiện tại, mình chỉ yêu cầu các bạn viết theo những gì mình đã viết, chưa yêu cầu các bạn phải hiểu được những dòng mã trên có ý nghĩa gì. Sau khi lưu file main.cpp lại, chúng ta đã có được file mã nguồn C++ đầu tiên. Các bạn có thể muốn xem thử file main.cpp vừa được lưu đang nằm chỗ nào. Để xem thư mục gốc của project, các bạn click chuột phải vào tên project HelloWorld trong cửa sổ **Solution Explorer** rồi chọn Open Folder in **File Explorer**.



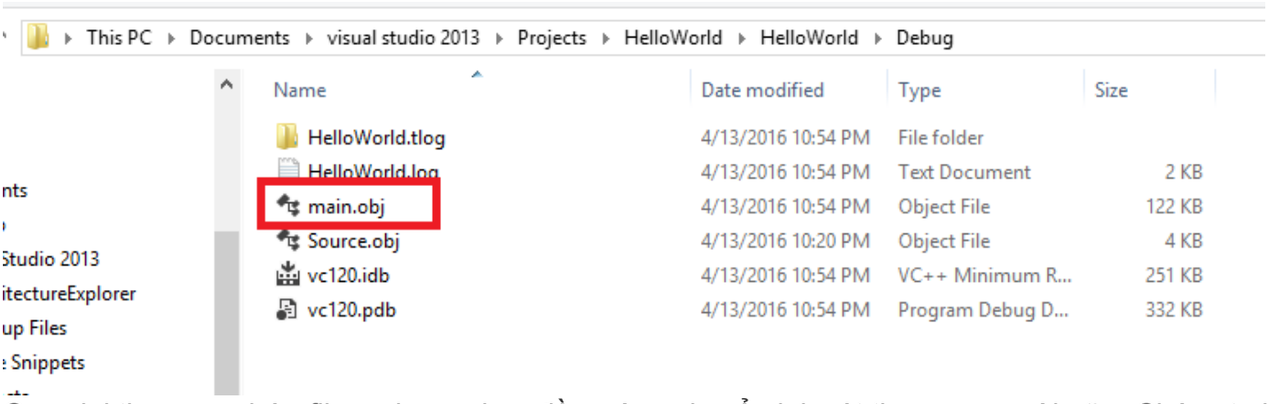
Và chúng ta thấy file main.cpp như trong hình bên dưới.



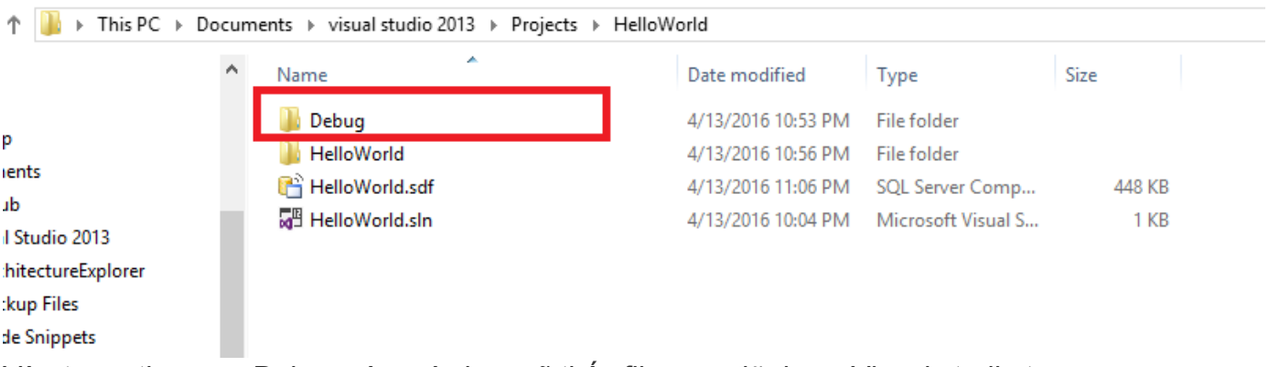
Quay lại với màn hình làm việc của Visual studio. Các bạn click chuột vào menu item BUILD trên Tool bar, sau đó chọn Build Solution (hoặc nhấn phím F7).



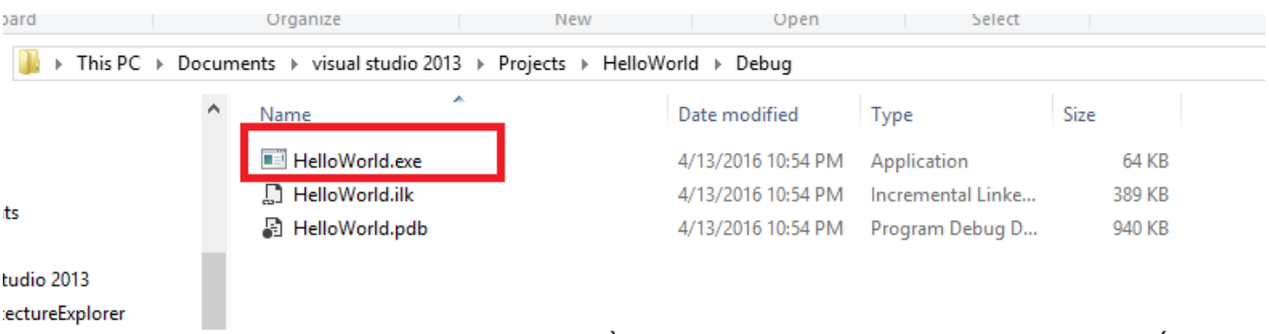
Thực hiện bước này, Visual studio sẽ biên dịch file main.cpp của bạn để tạo thành file object .obj, đồng thời liên kết file main.obj tạo thành file chương trình (có đuôi .exe). Chúng ta chuyển qua thư mục gốc của project chứa file main.cpp lúc nãy, double click vào thư mục Debug, chúng ta thấy file main.obj là kết quả của quá trình biên dịch mã nguồn.



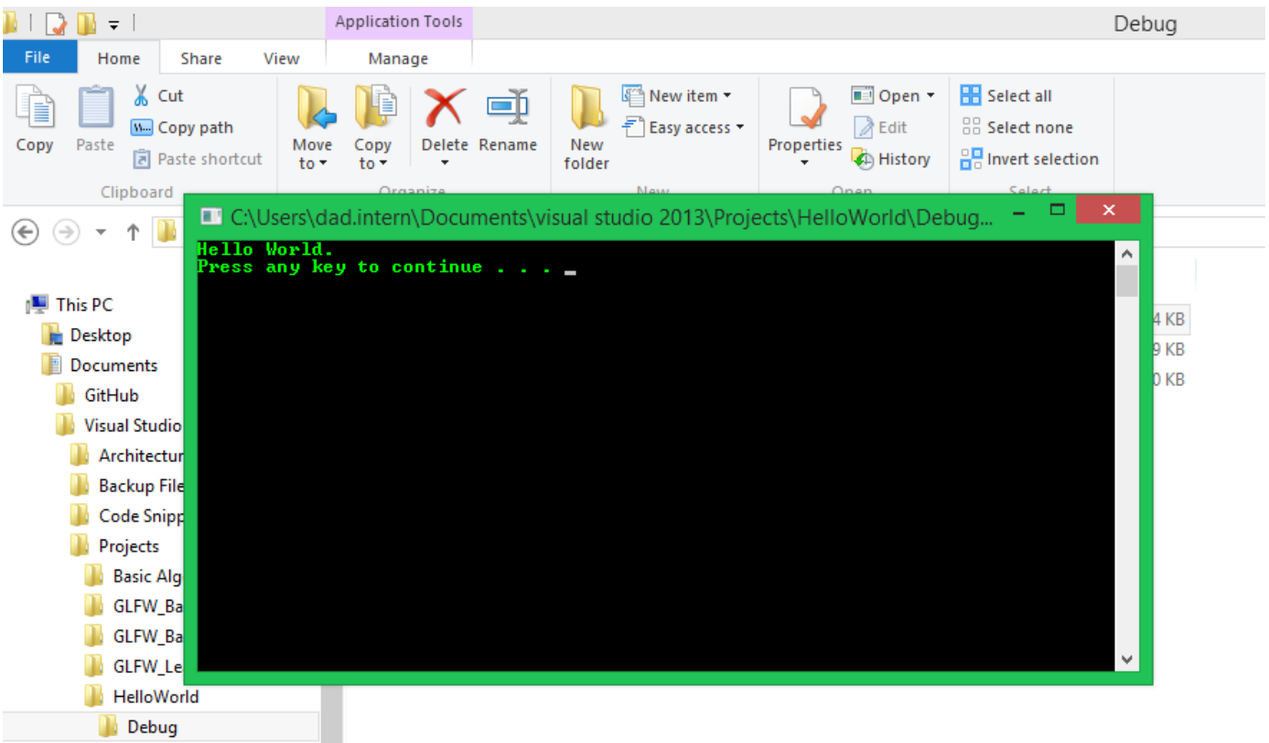
Quay lui thư mục chứa file main.cpp ban đầu, cùng chuyển lui một thư mục ngoài nữa. Chúng ta lại thấy một thư mục có tên là Debug khác.



Vào trong thư mục Debug này, các bạn sẽ thấy file .exe đã được Visual studio tạo ra.

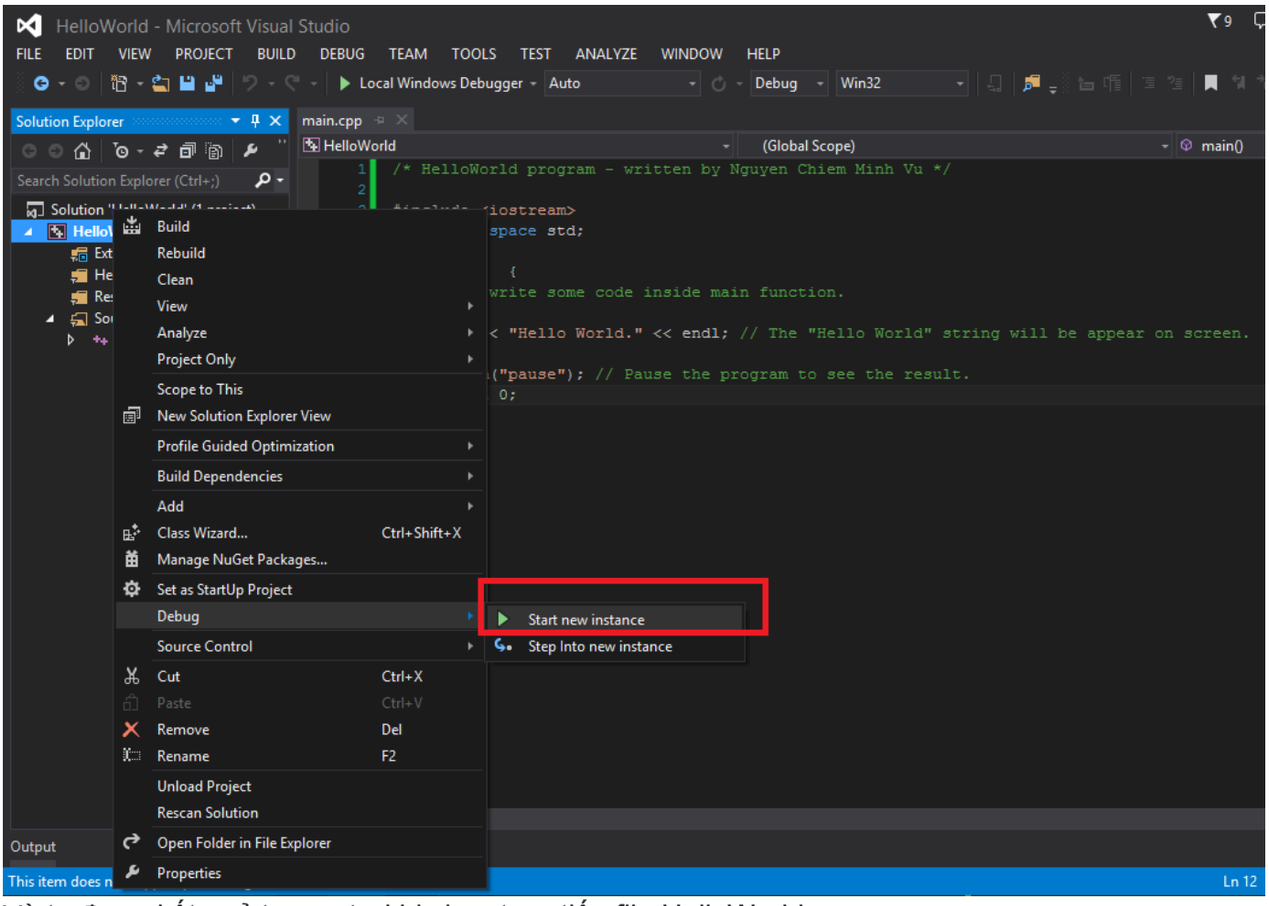


Bây giờ, các bạn mở file HelloWorld.exe này bằng cách double click vào nó. Cùng xem kết quả xuất hiện trên màn hình.

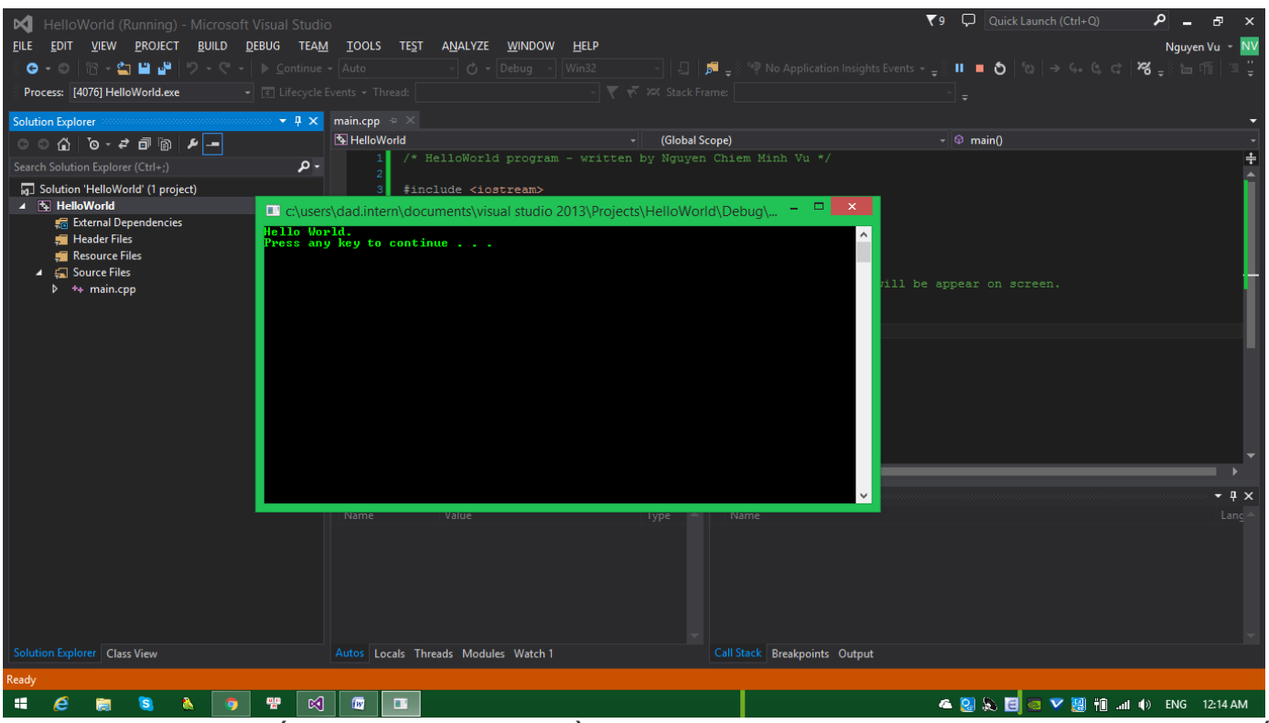


Như các bạn thấy, trong cửa sổ Console, chúng ta có một dòng chữ xuất hiện: "Hello World.", và một dòng gợi ý cho người dùng rằng: Hãy nhấn 1 phím bất kì để kết thúc chương trình.

Ngoài cách chạy trực tiếp file HelloWorld.exe trong thư mục Debug, các bạn còn có thể chạy chương trình ngay trên màn hình làm việc của Visual studio, bằng cách click chuột phải vào tên project trong cửa sổ Solution Explorer -> Debug -> Start new instance.

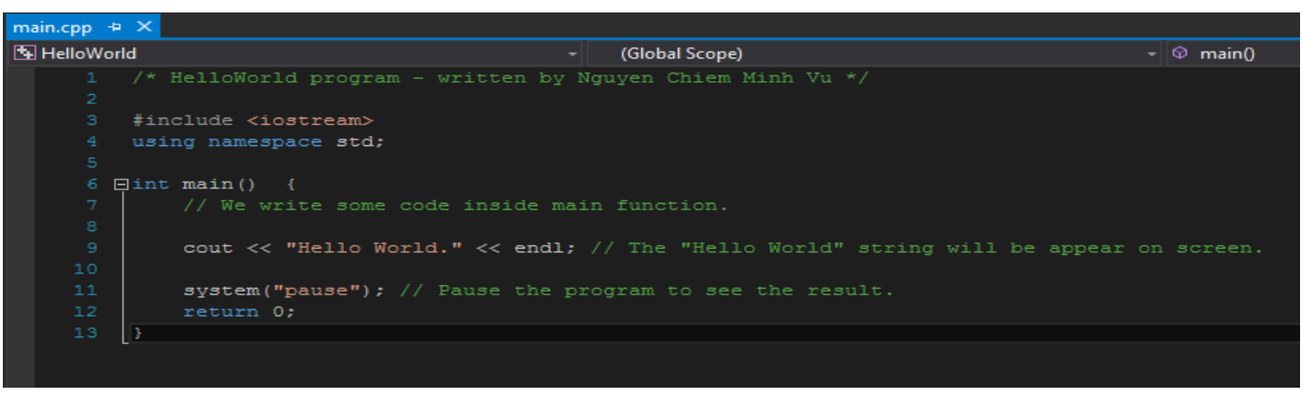


Và ta được kết quả tương tự khi chạy trực tiếp file HelloWorld.exe



Vậy là chúng ta đã viết xong chương trình đầu tiên của khóa học lập trình C++. Bây giờ mình muốn quay lại vấn đề mình đã nói ở trên, đó là tại sao chúng ta lại nên đặt tên file đầu tiên cho project là main.cpp?

Để giải thích vấn đề này, mình muốn các bạn nhìn lại mã nguồn của file main.cpp mà các bạn đã viết cùng mình để có cái nhìn tổng quan về cấu trúc của một chương trình C++ cơ bản.



Các bạn hãy chú ý đến dòng 6 trong chương trình trên. Chúng ta thấy

```
int main()
```

Đó là dòng bắt buộc phải có nếu muốn mã nguồn C++ có thể hoạt động được. Main trong tiếng Anh khi dịch ra có nghĩa là chính, quan trọng. Trong ngôn ngữ C++, main là điểm khởi đầu cho một chương trình. Trong một thời điểm, máy tính của chúng ta chỉ có thể thực hiện 1 dòng lệnh. Và ở thời điểm chương trình C++ bắt đầu chạy, nó sẽ tìm tới nơi có khai báo là main để thực hiện mã lệnh ở trong đó.
Mã lệnh mà chương trình thực hiện sẽ được đặt trong cặp ngoặc nhọn { và }.
Vì thế, cấu trúc chương trình C++ mà bạn cần nhớ sẽ như bên dưới.

```
int main()
{
    /* Đây sẽ là nơi mà chương trình C++ bắt đầu thực hiện các công việc */
}
```

```
Các bạn cần lưu ý, một chương trình C++ chỉ có duy nhất một hàm main.

Các bạn sẽ thấy nhiều hàm main có cách khai báo khác nhau. Nhưng với việc

bạn là người mới bắt đầu học C++, mình khuyến nghị các bạn nên sử dụng theo

cách trong hình trên.

int main()
{

}
```

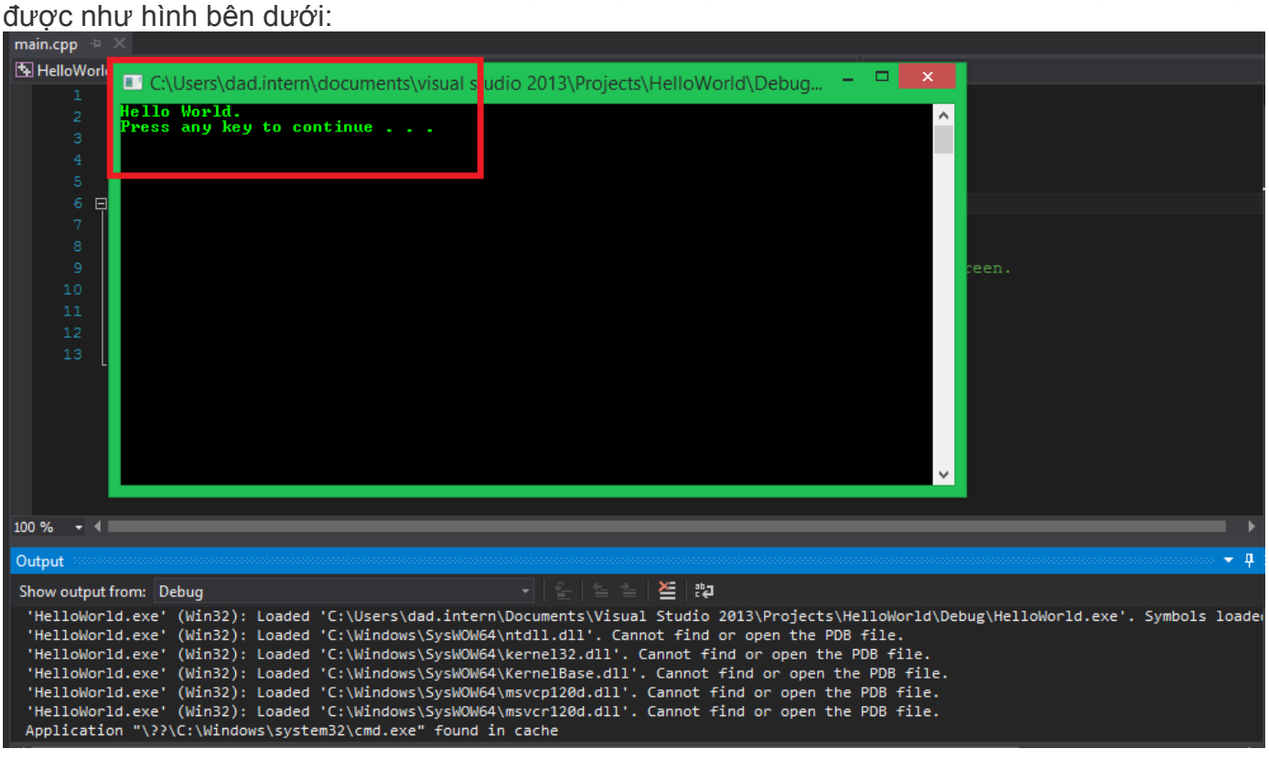
Chương trình của C++ sẽ thực hiện từng dòng lệnh trong cặp ngoặc nhọn { } ở phía sau hàm main một cách **có thứ tự**từ trên xuống dưới.

Một chương trình C++ bắt buộc phải có 1 hàm main, thế nên mình khuyên các bạn nên đặt tên file đầu tiên trong chương trình main.cpp, và file này sẽ chứa mã nguồn C++ có hàm main ở trong đó, sau này các bạn làm việc với 1 dự án có nhiều file thì sẽ không bị nhầm lẫn.

1.1 Cấu trúc cơ bản của một chương trình C++

Chào mừng các bạn đến với bài học tiếp theo trong khóa học lập trình C++ hướng thực hành.

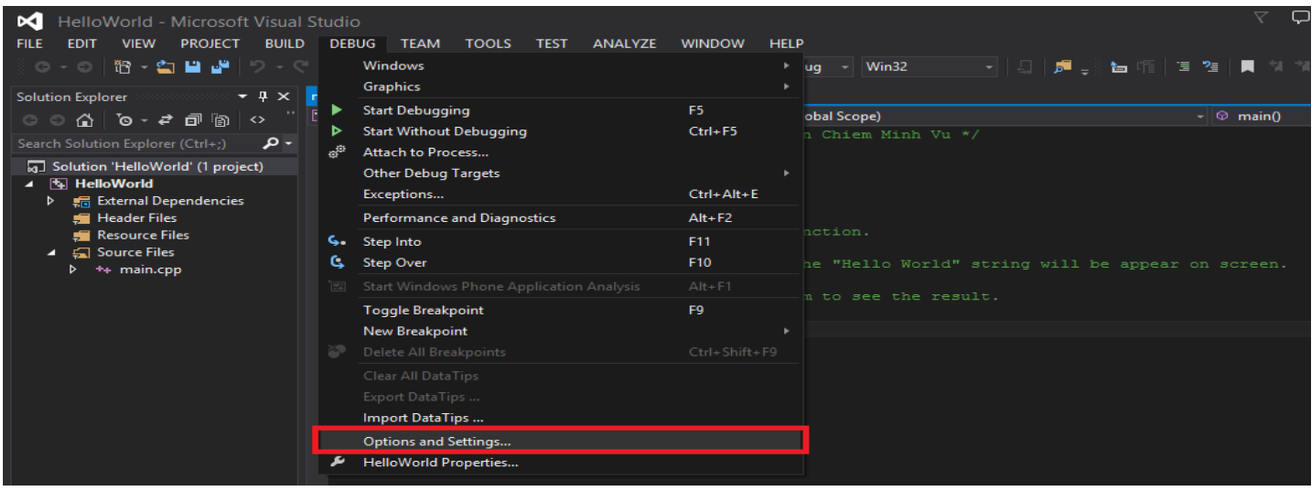
Trong bài trước, **Viết chương trình C++ đầu tiên**, chúng ta đã cùng nhau tạo 1 project có tên **HelloWorld**. Các bạn lưu ý rằng khi làm việc với Visual studio 2015 thì chúng ta làm việc trên 1 project chứ không làm việc với file mã nguồn đơn lẻ. Project **HelloWorld** hiện tại chỉ có một file có tên là **main.cpp**. Kết quả khi thực thi project này (bằng cách nhấn phím F5 để Debug) thì chúng ta được như hình bên dưới:



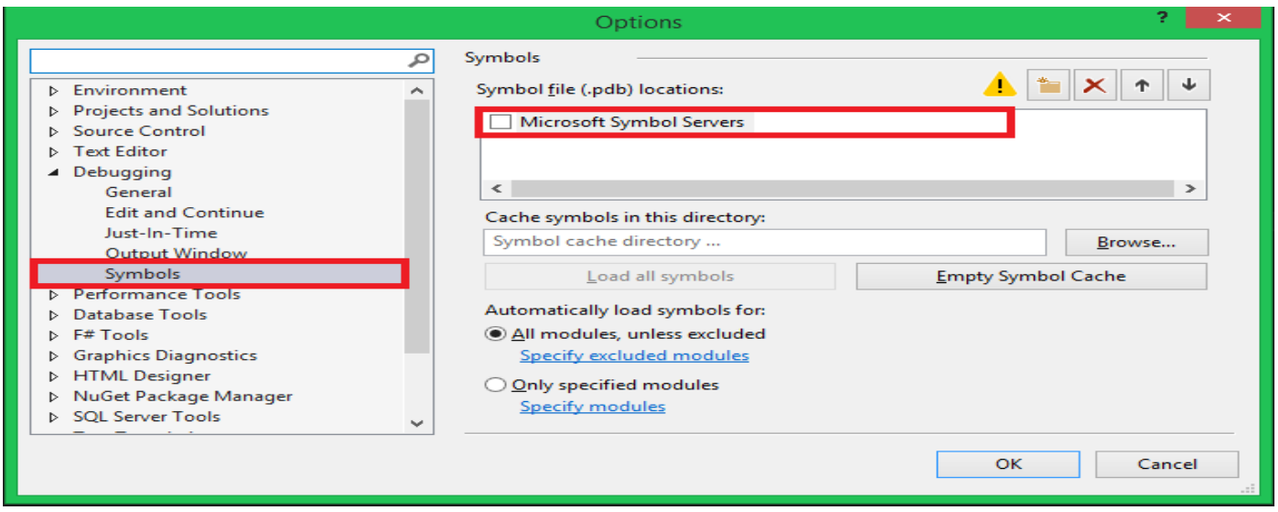
Kết quả là một dòng chữ **Hello World** xuất hiện trên console.

Khi các bạn Debug trên Visual studio 2015, có thể Visual studio sẽ download một số file PDB về làm tốn thời gian. Các bạn có thể tắt việc tự động download các file đó bằng cách làm theo các bước sau:

Đưa chuột vào phần DEBUG trên Menu bar -> chọn Options and Settings...



Chọn Symbols và bỏ dấu tick trong Symbol file (.pdb) locations đi



Bây giờ chúng ta nhìn lại mã nguồn trong file main.cpp và mình sẽ phân tích chức năng của từng dòng code.

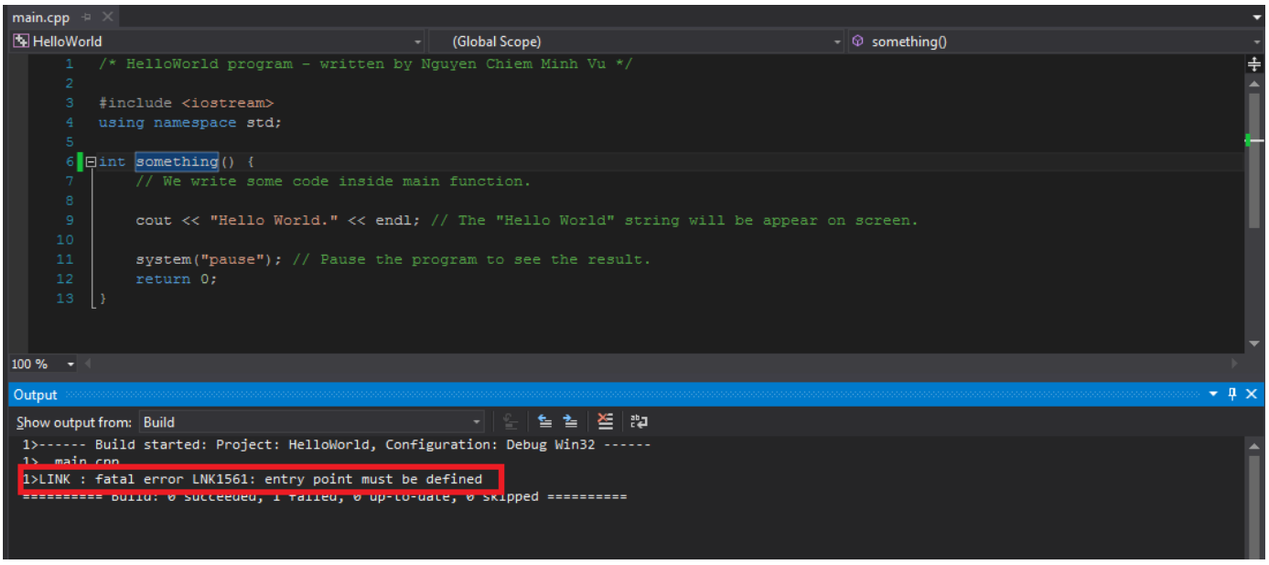
```
/* HelloWorld program - written by Nguyen Chiem Minh Vu */

#include <iostream>
using namespace std;

int main()      {
        // We write some code inside main function.

        cout << "Hello World." << endl; // The "Hello World" string will be appear on screen.

        system("pause"); // Pause the program to see the result.
        return 0;
}
```

- **Dòng 6**:

```
int main()
```

Như đã nói ở bài trước, **main là một hàm mà đi sau nó là một cặp dấu ngoặc nhọn { }**, một điểm xuất phát cho một project của ngôn ngữ C++. Không cần biết một project C++ của bạn có bao nhiêu file, một khi project đã được build và liên kết các file thành một file thực thi (.exe), hệ điều hành sẽ thực thi những dòng lệnh trong phạm vi dấu ngoặc nhọn nằm sau hàm main một cách **lần lượt từ trên xuống dưới**.

Với những bạn lần đầu viết code C++, các bạn có thể bỏ sót dòng này. Cùng xem thử Visual studio sẽ làm gì khi bạn không viết ra dòng int main() bằng cách đổi tên **main** thành một tên bất kỳ.



Visual studio thông báo một lỗi nghiêm trọng LNK1561, và nó còn đưa thêm hướng dẫn để lập trình viên có thể tự sửa lỗi có nghĩa là điểm khởi đầu cần được định nghĩa.

Chúng ta quay lại với đoạn mã nguồn có thể chạy được như lúc đầu bằng cách sửa lại tên hàm là `int main()`.

- **Dòng 7**:

```
// We write some code inside main function.
```

Đây không phải là một dòng lệnh. Đây là một dòng comment, mục đích của comment trong code là để ghi chú lại những gì mình đang làm. Việc ghi chú này cần được thực hiện thường xuyên đối với những người mới học lập trình. Ghi chú giúp bạn ít bị rối và khó hiểu khi nhìn lại những đoạn code cũ và những người làm việc cùng nhóm với bạn cũng sẽ hiểu được bạn đang muốn làm gì.

Một dòng comment bắt đầu với 2 dấu gạch chéo //.

Bây giờ bạn thử tự viết cho mình vài dòng comment đi nào. Comment bạn có thể đặt ở bất kỳ vị trí nào trong mã nguồn (ngoại trừ chèn comment làm ảnh hưởng đến dòng lệnh) mà không bị báo lỗi. Vì khi biên dịch, compiler nhìn thấy dòng comment thì nó sẽ bỏ qua và không làm gì cả.

- **Dòng 9**:

```
cout << "Hello World." << endl; // The "Hello World" string will be appear on screen.
```

Đây là một dòng lệnh và đi kèm sau đó là một dòng comment. Dòng lệnh này chính là thứ đã viết lên console dòng **Hello World** mà bạn đã thấy trong kết quả của chương trình.

Một dòng lệnh phải được kết thúc bằng dấu chấm phẩy ";"

Lệnh **cout** có tác dụng viết lên console tất cả những gì nằm trong cặp dấu ngoặc kép **" "**. Như các bạn thấy, chúng ta đặt 2 từ **Hello** và **World** bên trong cặp ngoặc kép nên nó đã được in ra màn hình console.

- **Dòng 11**:

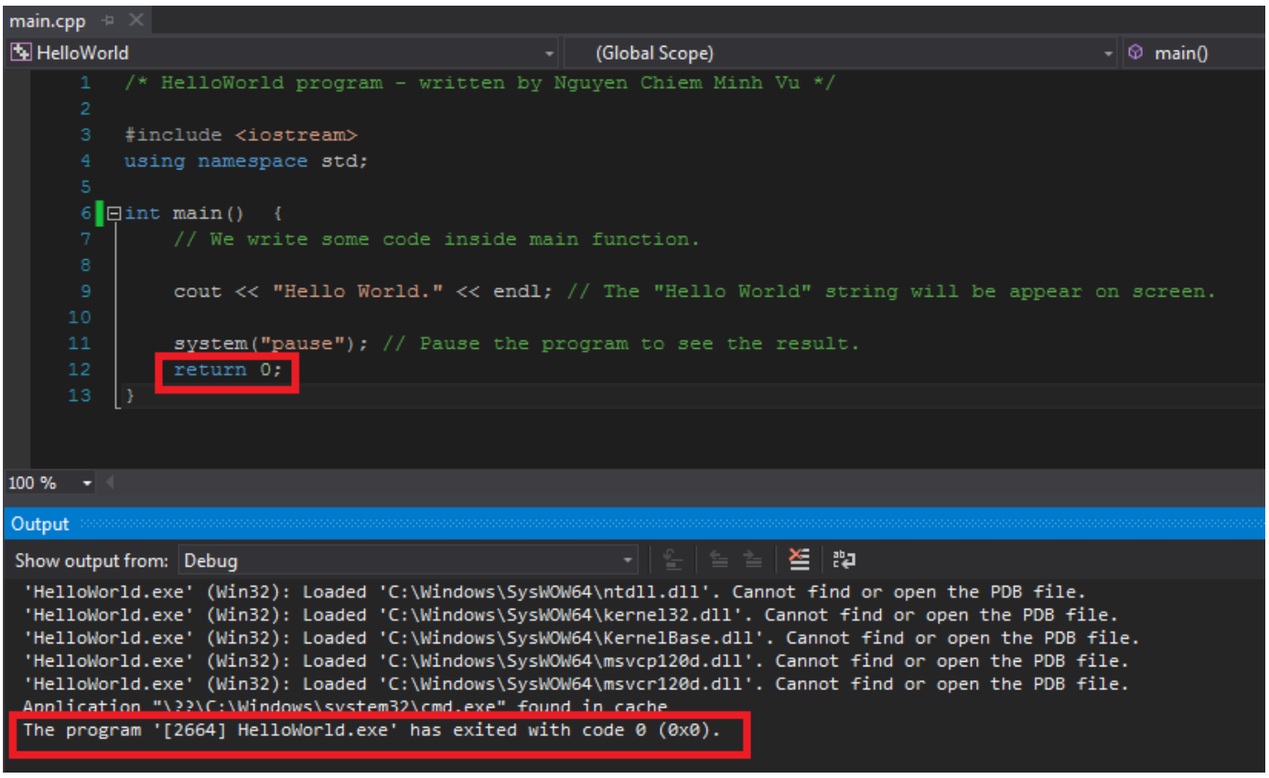
```
system("pause"); // Pause the program to see the result.
```

Tiếp tục là một dòng lệnh và đi kèm một dòng comment ở phía sau.
Mục đích của dòng lệnh này là để dừng chương trình và xem kết quả trên màn hình console. Các bạn có thể xóa dòng này đi và chạy lại chương trình bằng cách nhấn phím F5 để kiểm chứng kết quả. Lúc này màn hình console hiện lên và tắt ngay lập tức.

- **Dòng 12**:

```
return 0;
```

Là giá trị trả về của hàm main. Hàm main của chúng ta có từ khóa **int** đứng trước, có nghĩa là kiểu trả về của hàm main sẽ là một giá trị có kiểu **int** (integer - số nguyên). Giá trị trả về này do lập trình viên tự quy định. Kết quả hàm main sẽ hiển thị trong cửa sổ **Output** bên trong IDE sau khi bạn tắt chương trình HelloWorld đang chạy đi.



Thông thường, dòng này sẽ đặt cuối cùng trong phạm vi cặp ngoặc nhọn { } phía sau hàm main. Các bạn có thể thay bằng một con số bất kì sao cho bạn có thể hiểu được rằng, khi chương trình kết thúc, nếu **Output** xuất hiện con số mà bạn đã chọn, điều đó có nghĩa chương trình của bạn hoạt động một cách bình thường.

- **Dòng 3 và 4**:

```
#include <iostream>
using namespace std;
```

Đây là những dòng lệnh đặc biệt. Để có thể sử dụng dòng lệnh số 9 trong chương trình thì chúng ta cần có dòng lệnh số 3 và số 4 này. Mục đích của 2 dòng lệnh này là thêm thư viện có tên **iostream** và không gian tên **std** để tích hợp vào chương trình. Hay nói cách khác, vì lệnh **cout** được định nghĩa bên trong thư viện có tên **iostream** và bên trong không gian tên **std** nên chúng ta cần tích hợp 2 thứ đó vào chương trình. Đến đây các bạn sẽ thắc mắc là **"Làm thế nào biết được dòng lệnh nào đã được định nghĩa bên trong thư viện nào?"**

Qua quá trình thực hành trong khóa học này, mình sẽ cùng các bạn sử dụng một số chức năng bên trong một số thư viện chuẩn do ngôn ngữ C++ đã định nghĩa sẵn và các bạn sẽ quen với việc tìm và sử dụng chức năng nào trong thư viện nào.

Đây cũng là một đặc trưng của ngôn ngữ lập trình bậc cao. Chúng ta sử dụng lại những gì đã được định nghĩa sẵn giúp công việc lập trình của chúng ta dễ dàng hơn.

- **Dòng 1**:

```
/* HelloWorld program - written by Nguyen Chiem Minh Vu */
```

Đây cũng là một đoạn comment. Đoạn comment khác với dòng comment. Đoạn comment được đặt giữa cặp dấu /* và */ trong khi dòng comment đứng sau 2 dấu gạch chéo //. Chúng ta có thể có nhiều dòng comment trong 1 đoạn comment. Ví dụ:

```
/*
        Đây là một dòng comment.
```

```
        Đây là một dòng comment khác.
        Các bạn thích viết bao nhiêu dòng comment giữa này cũng được.
*/
```

Mình đã giải thích xong chức năng và cách hoạt động của mã nguồn file **main.cpp** trong project **HelloWorld**. Có thể các bạn chưa thể hiểu hết được, nhưng đừng lo lắng về điều đó, chúng ta sẽ quen với việc sử dụng ngôn ngữ C++ khi thực hành nhiều và nếu cần thiết các bạn sẽ được những người làm khóa học này hỗ trợ trực tiếp.

Bây giờ là lúc để hình dung về cấu trúc của chương trình C++ cơ bản mà chúng ta đã làm cùng nhau.

Cấu trúc cơ bản của chương trình C++

Đầu tiên, chúng ta có hàm **main**

```
int main()
```

Sau đó, chúng ta có phần thân của hàm main là cặp dấu ngoặc nhọn đứng sau từ khóa **main**, cuối thân hàm **main** là giá trị trả về của hàm **main**.

```
int main()
{

        return 0;
}
```

Tiếp đến, chúng ta có những dòng lệnh đặt bên trong thân hàm **main**

```
int main()
{
        cout << "This is a command" << endl;

        return 0;
}
```

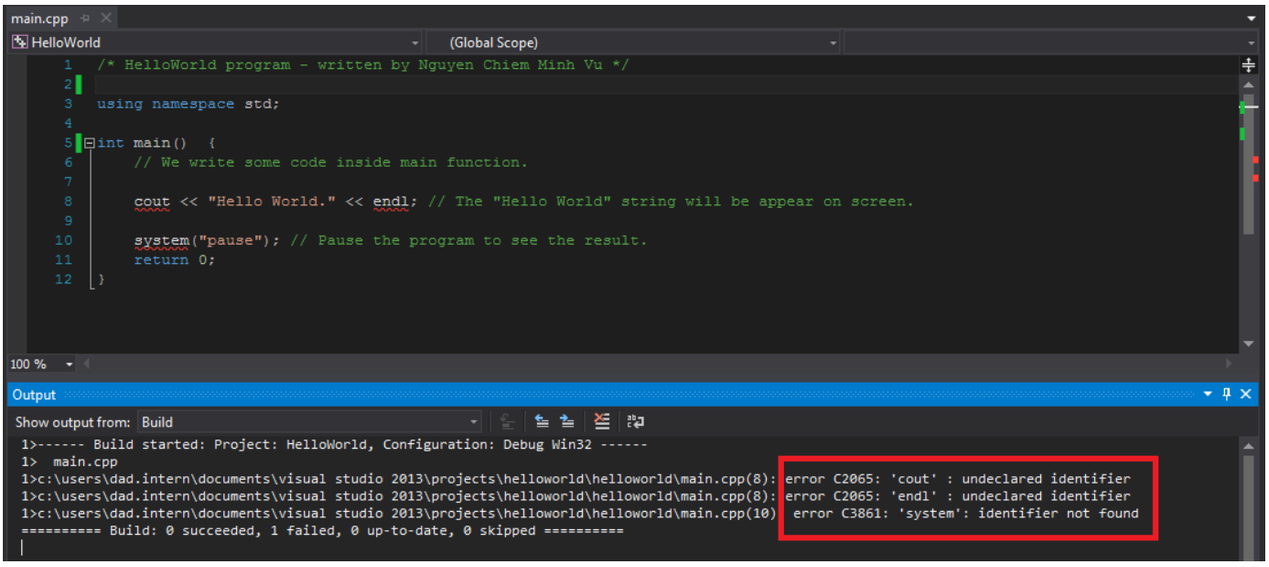
Bên cạnh những dòng lệnh, chúng ta còn có những dòng **comment**

```
// This comment is located outside main function

/* We can put comment everywhere in a C++ file */

int main()
{
        // We are coding inside main function
        cout << "This is a command" << endl;

        return 0;
}
```

Và cuối cùng là những thư viện cần thiết để compiler có thể hiểu được những lệnh đã được định nghĩa sẵn trong ngôn ngữ lập trình C++

```
#include <iostream>
#include <cmath>
#include <string>

using namespace std;

// This comment is located outside main function

/* We can put comment everywhere in a C++ file */

int main()
{
        // We are coding inside main function
        cout << "This is a command" << endl;

        return 0;
}
```

Các bạn lưu ý là không nên include cả đống thư viện chưa cần dùng đến nhé. Cần dùng lệnh gì đã được định nghĩa sẵn mới cần include vào. Tất nhiên khi thêm nhiều thư viện vào chương trình thì IDE sẽ không báo lỗi vì compiler biết thư viện nào được dùng, ~~nhưng chương trình của chúng ta sau khi build ra sẽ nặng hơn~~. Nếu các bạn sử dụng những lệnh được định nghĩa trong thư viện mà không include nó vào thì IDE sẽ báo lỗi ngay.
Ví dụ mình bỏ dòng `#include <iostream>` đi, IDE sẽ báo lỗi như hình bên dưới:



Như các bạn thấy, cả lệnh **cout**, **system("")**, và **endl** đều được định nghĩa bên trong thư viện **iostream** nên khi xóa thư viện đó đi, chương trình gạch chân màu đỏ các từ đó, đồng thời thông báo lỗi trong cửa sổ **Output**.

Tổng kết

Trong bài học hôm nay, chúng ta đã biết thêm một số điểm đáng chú ý khi làm việc với chương trình C++:

- Một dòng comment sẽ đứng sau 2 dấu gạch chéo //.
- Một đoạn comment sẽ nằm giữa cặp /* và */.
- Một dòng lệnh phải được kết thúc bằng dấu chấm phẩy **";"**
- Cấu trúc của một chương trình C++ cơ bản:
    - Hàm main: (bắt buộc phải có)
        - Kiểu trả về của hàm main (int).
        - Tên của hàm main (cũng là main luôn).
        - Thân của hàm main (cặp dấu ngoặc nhọn { và }).
        - Giá trị trả về của hàm main (return 0; //hoặc giá trị bao nhiêu cũng đc).
    - Những dòng lệnh bên trong thân hàm main. (Có thể có hoặc không)
    - Những dòng comment. (Có thể có hoặc không)
    - Tích hợp thư viện và không gian tên. (Phụ thuộc vào các lệnh mà bạn sử dụng).

Bài tập cơ bản

1. Trong chương trình C++ HelloWorld đầu tiên, ở dòng 1 là đoạn comment chứa thông tin về tên Project và tên người viết chương trình. Các bạn đã viết cùng mình mà hình như vẫn chưa có tên của các bạn đấy. Sửa lại comment đó đi nào!
2. Thay vì in ra dòng chữ Hello World, thay vì in ra dòng **Hello World**, hãy thử in ra cái gì đó thú vị hơn xem nào! (Tên của bạn thì sao?)
3. Hãy comment theo cách của bạn!

1.2 Lệnh, khối lệnh, từ khóa

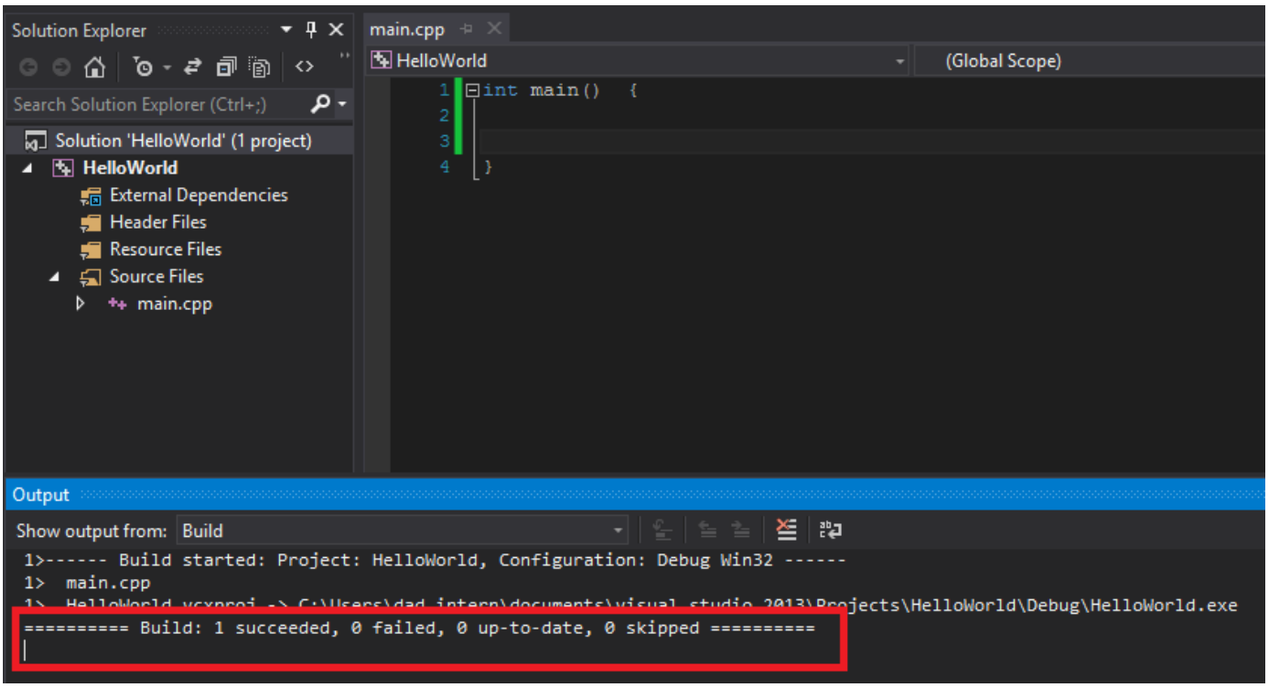
Rất vui khi được gặp lại các bạn trong bài học tiếp theo trong khóa học lập trình C++ cho người mới bắt đầu. Hôm trước, chúng ta đã có cái nhìn đầu tiên về các thành phần cơ bản hình thành nên một chương trình C++.

Mình sẽ nhắc lại một chút trong bài học trước.

Cấu trúc của một chương trình C++ cơ bản:

```
+ Hàm main: (bắt buộc phải có)

        + Kiểu trả về của hàm main (int).
        + Tên của hàm main (cũng là main luôn).
        + Thân của hàm main (cặp dấu ngoặc nhọn { và }).
        + Giá trị trả về của hàm main (return 0; //hoặc giá trị bao nhiêu cũng đc).


+ Những dòng lệnh bên trong thân hàm main. (Có thể có hoặc không)

+ Những dòng comment. (Có thể có hoặc không)

+ Tích hợp thư viện và không gian tên. (Phụ thuộc vào các lệnh mà bạn sử dụng).
```

Chúng ta đã biết hàm main là thứ quan trọng nhất cần phải có của một chương trình C++. Chúng ta hoàn toàn có thể khai báo hàm main xong và chạy chương trình ngay mà không bị báo lỗi.



Như các bạn thấy, chương trình vẫn được build thành file .exe và hoàn toàn có thể chạy được (Các bạn nhấn thử phím F5). Nhưng khi cửa sổ console vừa hiện lên thì chương trình đóng lại ngay lập tức. Vì bạn không yêu cầu máy tính thực hiện công việc gì cả. Một chương trình máy tính được tạo ra để không làm gì cả thì thật là vô ích.

Để giải quyết vấn đề mà chúng ta đặt ra trên máy tính, chúng ta cần ra lệnh cho máy tính thực hiện các công việc cụ thể. Chúng ta ra lệnh cho máy tính bằng các lệnh trong ngôn ngữ lập trình. Để máy tính thực hiện công việc và cho ra kết quả, nó cần nhận lệnh từ người lập trình. Vì thế, yếu tố quan trọng chỉ đứng sau hàm **main** chính là **những câu lệnh**.

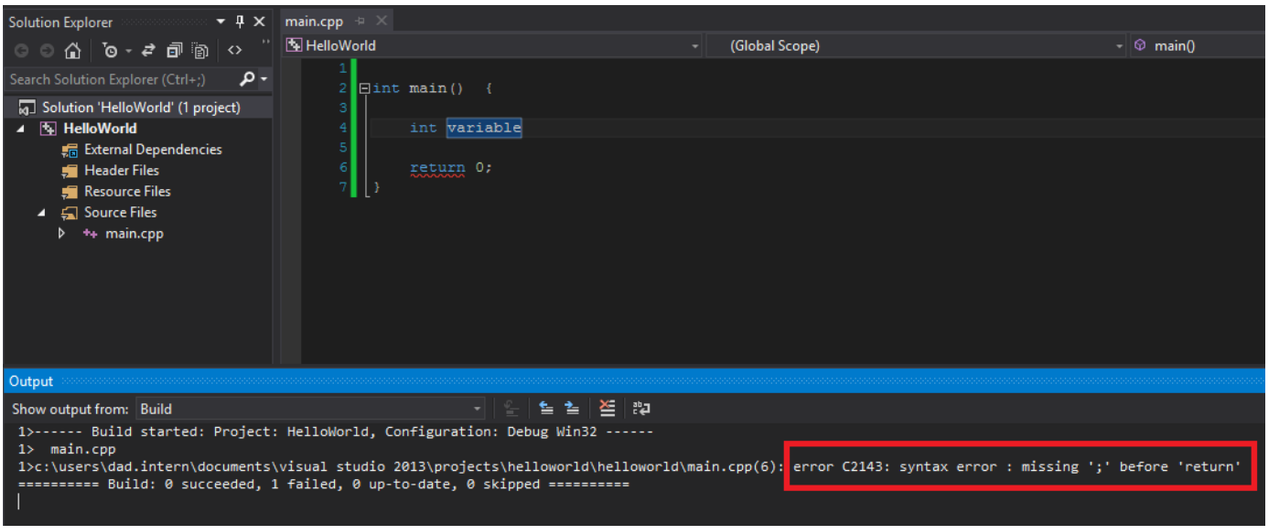
Lệnh trong ngôn ngữ lập trình C++

Trong C++, một lệnh là một chỉ thị riêng biệt của một chương trình.
Ví dụ:

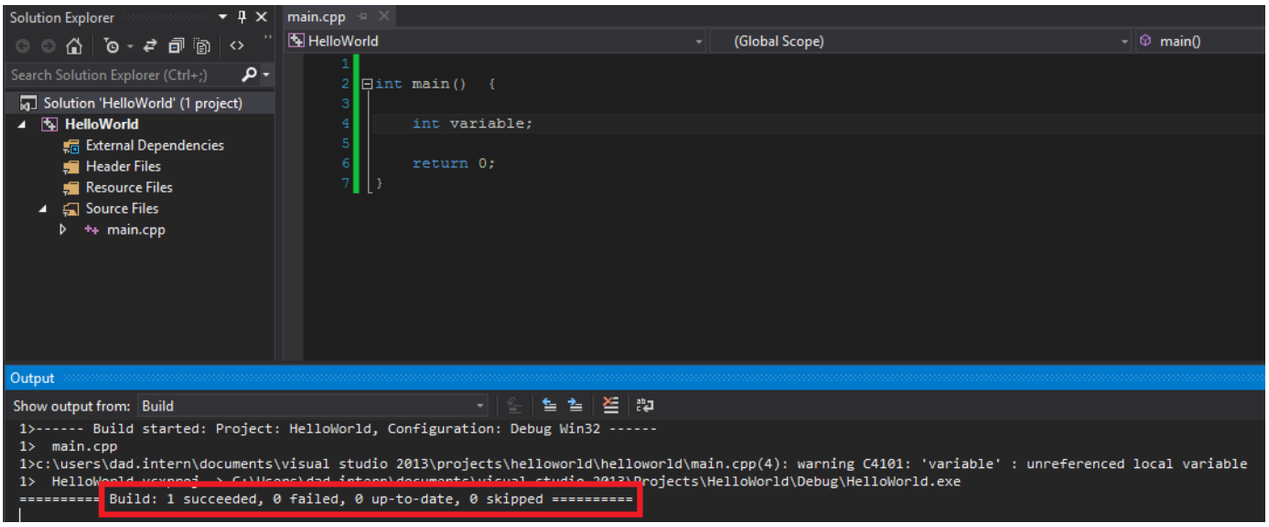
```
int variable = 0;

cout << "Print something";

variable = variable + 10;
```

Ở trên đây, chúng là có 3 dòng lệnh. 3 dòng lệnh này chỉ dẫn máy tính thực hiện 3 công việc khác nhau, nhưng trên phương diện cú pháp, nó có một đặc điểm chung rất quan trọng cần phải nhớ: **dòng lệnh kết thúc bằng dấu chấm phẩy ";"**

Chỉ cần một dòng lệnh bị bỏ sót dấu chấm phẩy, IDE sẽ thông báo với người lập trình lỗi về cú pháp.

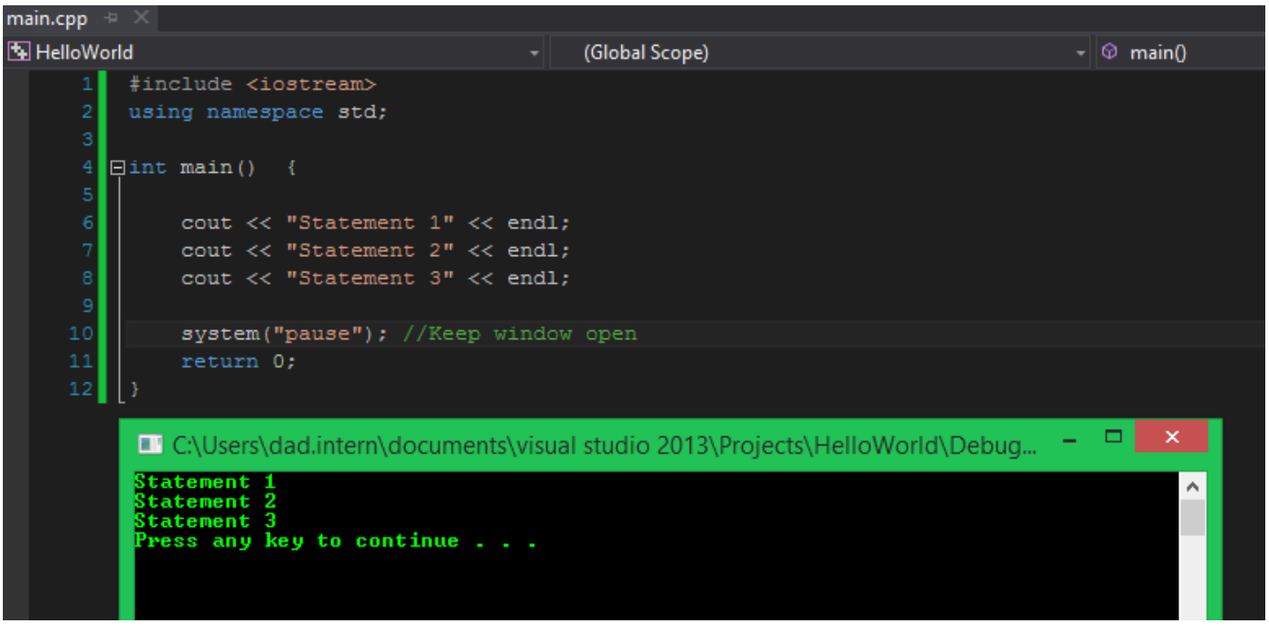


Thông báo lỗi này có nghĩa bạn viết thiếu dấu chấm phẩy ";" ở trước dòng thứ 6 trong chương trình.



Sau khi thêm dấu chấm phẩy vào dòng lệnh trước đó, chương trình được build bình thường.

Một điều cần lưu ý nữa là: **Tại một thời điểm, chương trình chỉ có thể thực hiện được 1 dòng lệnh. Các dòng lệnh được thực hiện tuần tự từ trên xuống dưới.**



Như kết quả trong hình, ta có dòng chữ **"Statement 1"** được in ra trước dòng **"Statement 2"** và **"Statement 3"** vì dòng lệnh dùng để in ra **"Statement 1"** nằm phía trên các lệnh còn lại.

Ở ví dụ trên chỉ bao gồm các câu lệnh đơn giản. Chúng ta sẽ được học những lệnh có cấu trúc đặc biệt như **câu lệnh có cấu trúc rẽ nhánh**, **câu lệnh có cấu trúc lặp**... trong những bài sau.

Khối lệnh

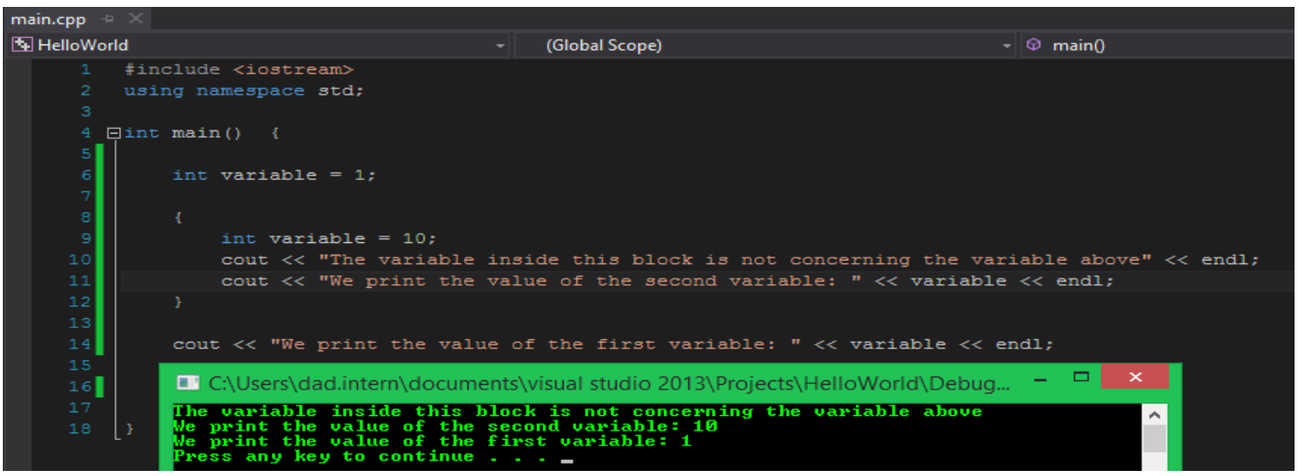
Trong C++, một khối lệnh là tập hợp những câu lệnh được đặt trong cặp dấu ngoặc nhọn **{** và **}**.

Một khối lệnh có thể chứa nhiều dòng lệnh, có thể chứa một dòng lệnh hoặc không chứa dòng lệnh nào. Một khối lệnh còn có thể chứa một hoặc nhiều khối lệnh khác.

```
#include <iostream>
using namespace std;

int main()	{

	int variable = 1;

	{
		int variable = 10;
		cout << "The variable inside this block is not concerning the variable above" << endl;
		cout << "We print the value of the second variable: " << variable << endl;
	}

	cout << "We print the value of the first variable: " << variable << endl;

	return 0;
}
```

Dễ thấy nhất là khối lệnh đóng vai trò **thân của hàm main**. Trong khối lệnh thân hàm main, chúng ta có một khối lệnh khác có chức năng in ra giá trị của biến variable thứ hai (bên trong khối lệnh con). Khối lệnh con này không gây ảnh hưởng gì với các câu lệnh bên ngoài nó.

Chúng ta cùng xem kết quả đoạn chương trình trên:



Chúng ta thấy giá trị của biến variable bên trong khối lệnh con hoàn toàn khác so với giá trị của biến variable bên ngoài. (**Các bạn sẽ hiểu được biến là gì qua những bài học tiếp theo**)

Từ khóa trong C++

Trong ngôn ngữ lập trình C++ hay bất kỳ ngôn ngữ lập trình nào khác, chúng ta đều có sẵn một số các **từ khóa** do người tạo ra ngôn ngữ đó định nghĩa sẵn. Mỗi từ khóa có một ý nghĩa riêng, khi chúng ta kết hợp các từ khóa và một số cú pháp đi kèm, chúng ta sẽ có được câu lệnh.

Ví dụ:

```
int var;
```

Lệnh trên sử dụng từ khóa **int** để định nghĩa một biến tên **var** có kiểu số nguyên (**integer**).

```
const float f_number = 1.0f;
```

Câu lệnh trên sử dụng 2 từ khóa: **const** và **float** kết hợp với một số yếu tố khác tạo nên một câu lệnh có chức năng khai báo một hằng số kiểu số thực và gắn cho nó giá trị cố định là 1.

Những từ khóa đã được định nghĩa và luôn sẵn sàng để sử dụng. Vì thế, chúng ta không cần **include** các thư viện ngoài vào để sử dụng chúng.

Dưới đây là bảng các từ khóa phổ biến dùng trong ngôn ngữ C++ 11

alignas (since C++11)	else	requires (concepts TS)
alignof (since C++11)	enum	return
and	explicit	short
and_eq	export(1)	signed
asm	extern	sizeof
auto(1)	false	static
bitand	float	static_assert (since C++11)
bitor	for	static_cast
bool	friend	struct
break	goto	switch
case	if	template
catch	inline	this
char	int	thread_local (since C++11)
char16_t (since C++11)	long	throw
char32_t (since C++11)	mutable	true
class	namespace	try
compl	new	typedef
concept (concepts TS)	noexcept (since C++11)	typeid
const	not	typename
constexpr (since C++11)	not_eq	union
const_cast	nullptr (since C++11)	unsigned
continue	operator	using(1)
decltype (since C++11)	or	virtual
default(1)	or_eq	void
delete(1)	private	volatile
do	protected	wchar_t
double	public	while
dynamic_cast	register	xor
	reinterpret_cast	xor_eq

Các bạn sẽ được cùng mình sử dụng những từ khóa thông dụng trong quá trình học và làm bài tập của khóa học này.

Trong bảng từ khóa trên có một vài từ khóa được bổ sung thông qua chuẩn C++ 11.

Tổng kết

Qua bài học ngày hôm nay, chúng ta hiểu rõ hơn khái niệm **Lệnh và khối lệnh**, để sau này khi mình thường xuyên dùng đến từ này thì các bạn sẽ tránh khỏi một số thắc mắc không cần thiết.

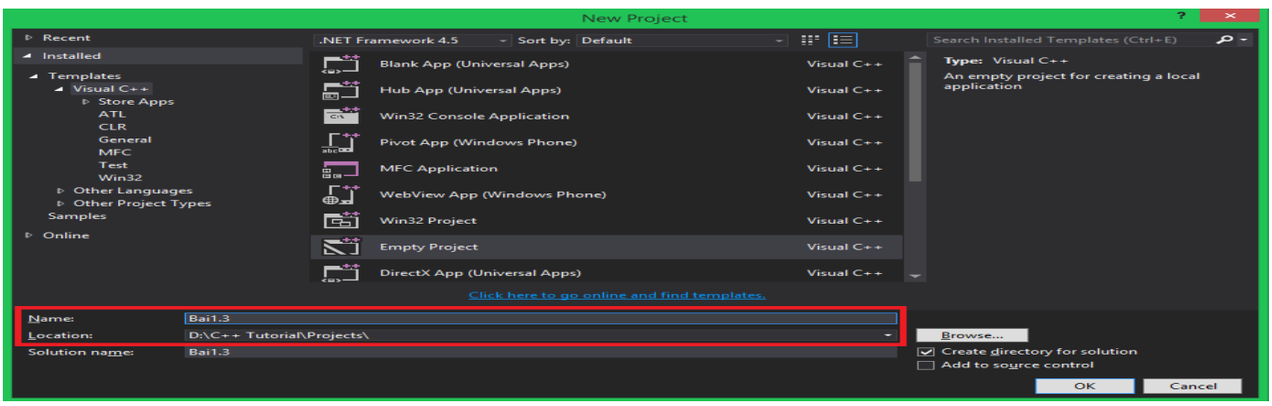
Ngoài ra, chúng ta còn được xem qua bảng các từ khóa đã được định nghĩa sẵn trong ngôn ngữ lập trình C++.

1.3 Sử dụng các lệnh liên quan đến xuất dữ liệu

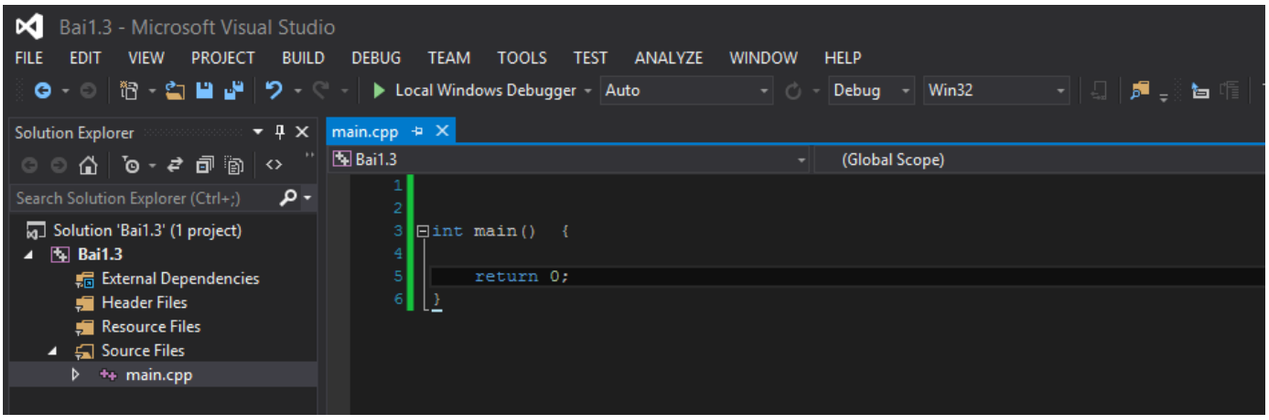
Hôm nay, chúng ta tiếp tục với khóa học lập trình C++ cho người mới bắt đầu.

Trong bài học này, chúng ta cùng nhau học cách sử dụng một số **lệnh** để in dữ liệu ra màn hình **console**, ngoài ra chúng ta còn tập cách định dạng chúng cho phù hợp, và một số thứ khác liên quan đến luồng dữ liệu output (**ostream**)...

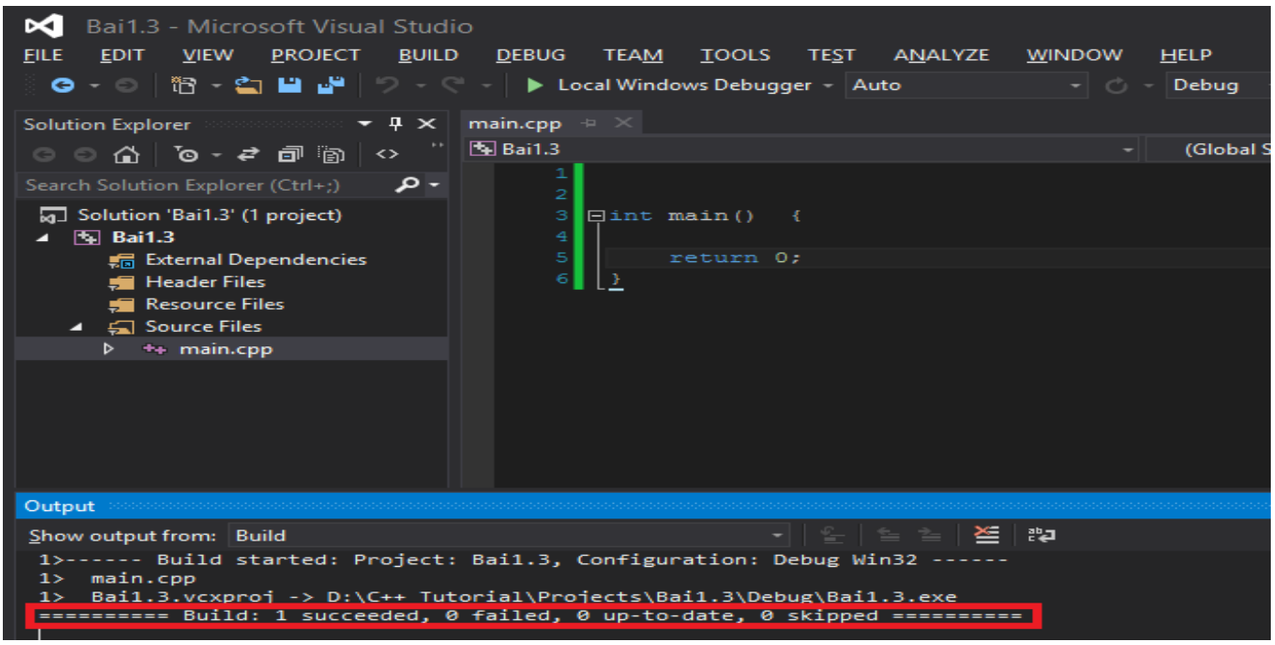
Trước khi bắt đầu, chúng ta tạo một project C++ mới có tên là Bai1.3 (các bạn nhớ chọn thư mục lưu sao cho phù hợp)



Sau đó tạo file **main.cpp** và viết sẵn cấu trúc cơ bản một chương trình C++ trong file main.cpp

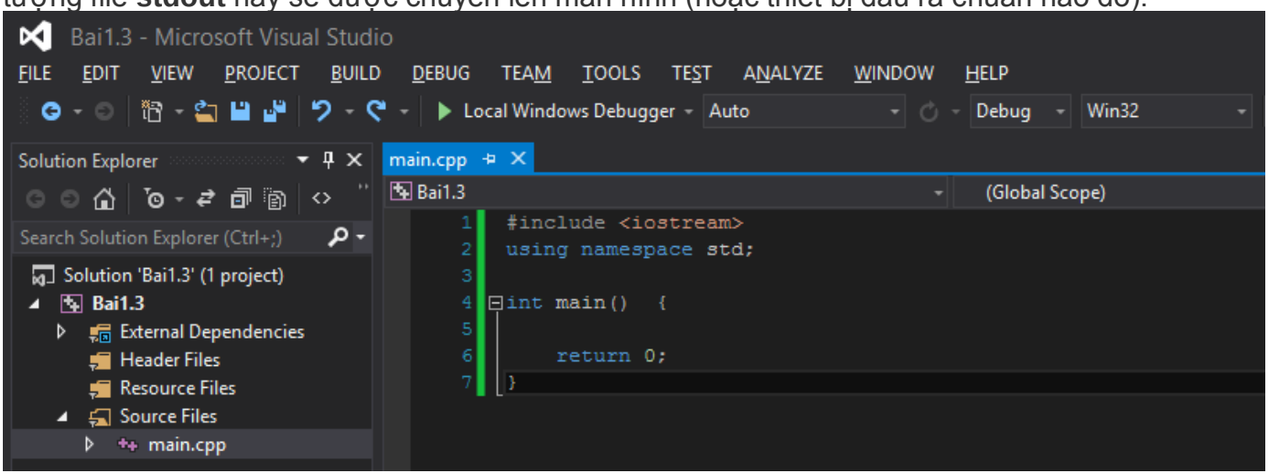


Nhấn tổ hợp phím **Ctrl + Shift + B** để thực hiện quá trình build project (làm thế để đảm bảo mọi thứ hoạt động bình thường trước khi bắt đầu viết code)



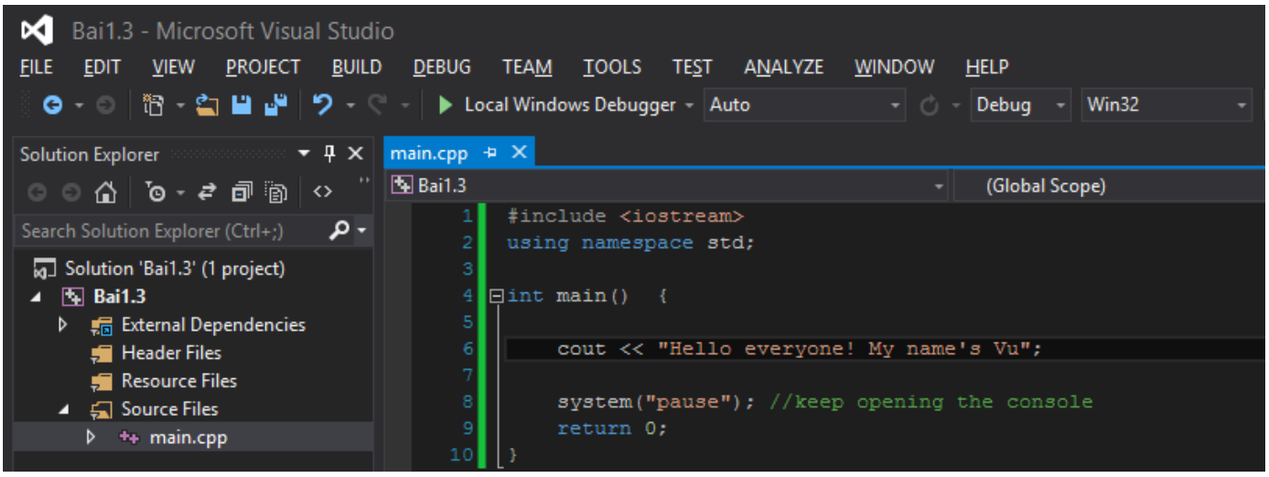
Mình bắt đầu với một ví dụ đơn giản, đó là in tên của mình ra màn hình console. Để làm được điều này, chúng ta sử dụng lệnh **cout** và chúng ta từng dùng trong project **HelloWorld**.

cout là một đối tượng được định nghĩa trong thư viện **iostream**, nó điều khiển một luồng dữ liệu đầu ra của chương trình, mặc định kết nối với output stream có tên là **stdout**. Dữ liệu được đưa vào đối tượng file **stdout** này sẽ được chuyển lên màn hình (hoặc thiết bị đầu ra chuẩn nào đó).



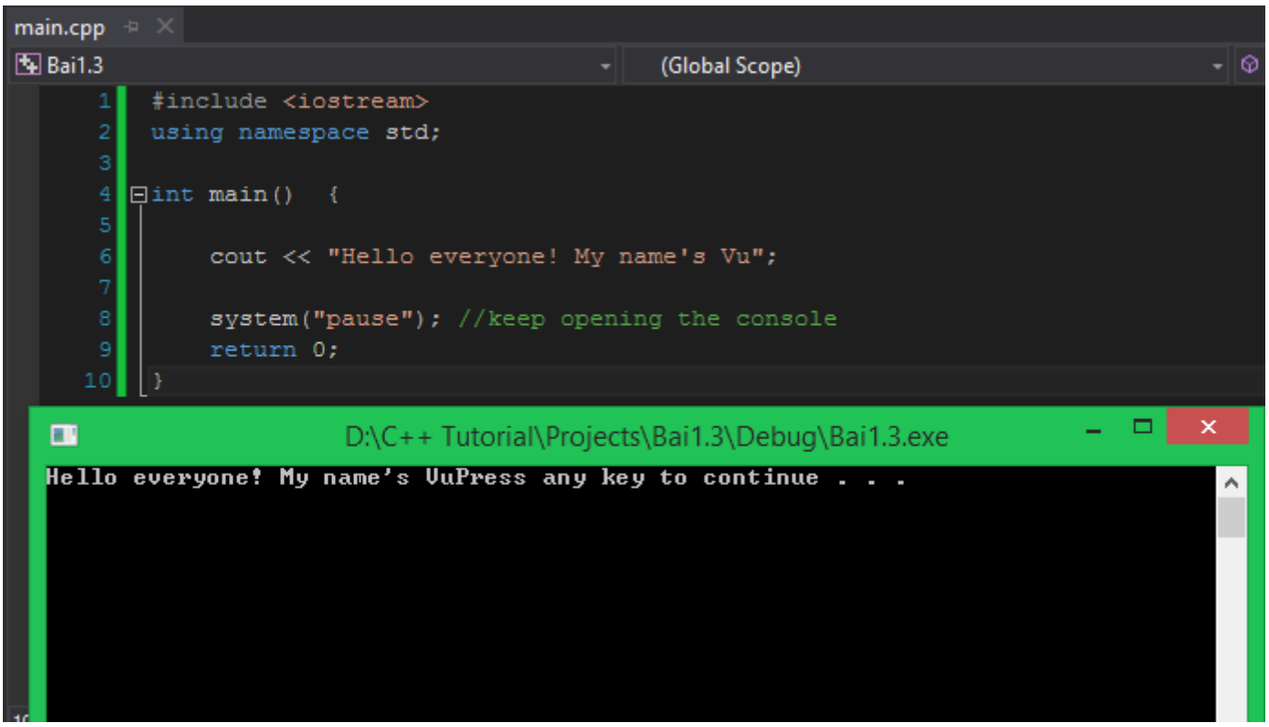
Vì thế, muốn sử dụng lệnh **cout** chúng ta cần include thư viện **iostream** vào trước (sử dụng luôn dòng **using namespace std** nhé).

Để đưa một chuỗi kí tự lên màn hình, chúng ta cần đặt chuỗi kí tự đó giữa cặp dấu ngoặc kép **"** và **"**

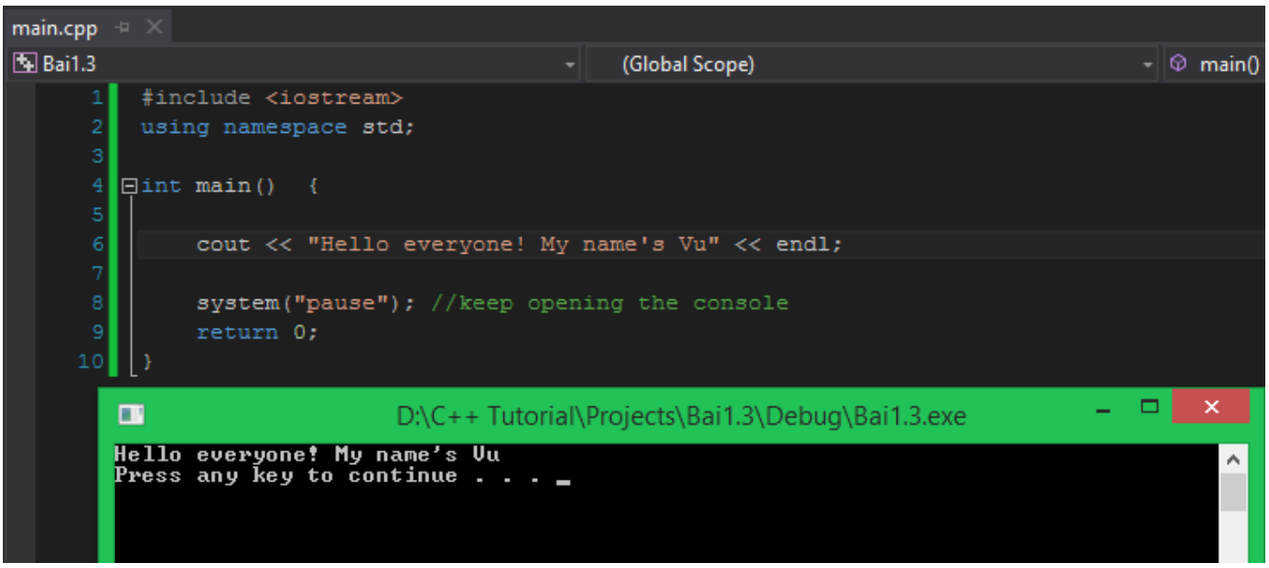


Có một toán tử đi kèm với lệnh **cout** là **<<**. Về mặt cú pháp, chúng ta đặt toán tử **<<** giữa lệnh **cout** và cái mà chúng ta muốn đưa lên màn hình (có thể là một chuỗi kí tự, một con số, một biến số...)

Chạy thử chương trình bằng cách nhấn phím **F5**

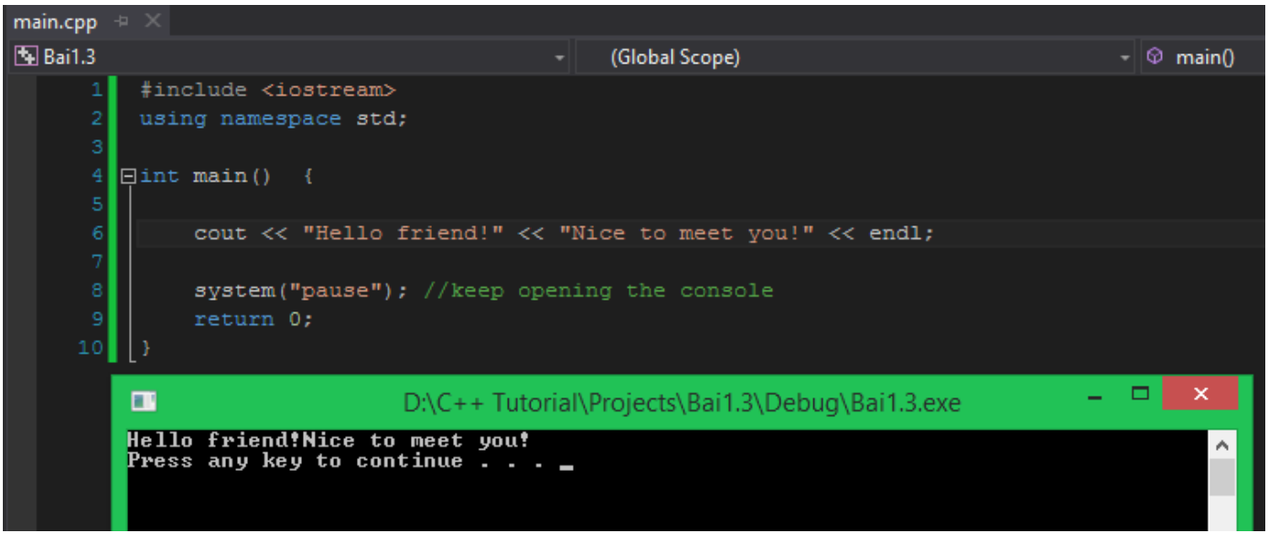


Nhìn vào kết quả chương trình, chúng ta thấy rất khó đọc vì dòng `cout` của mình bị dính với dòng chữ `Press any key to continue ...`. Để giải quyết vấn đề này, chúng ta cần làm cách nào đó để tách dòng chữ `Press any key to continue ...`. C++ đã hỗ trợ cho chúng ta một đối tượng khác cũng thuộc thư viện **iostream**, đó là **endl**.
Các bạn sử dụng **endl** như trong hình bên dưới.



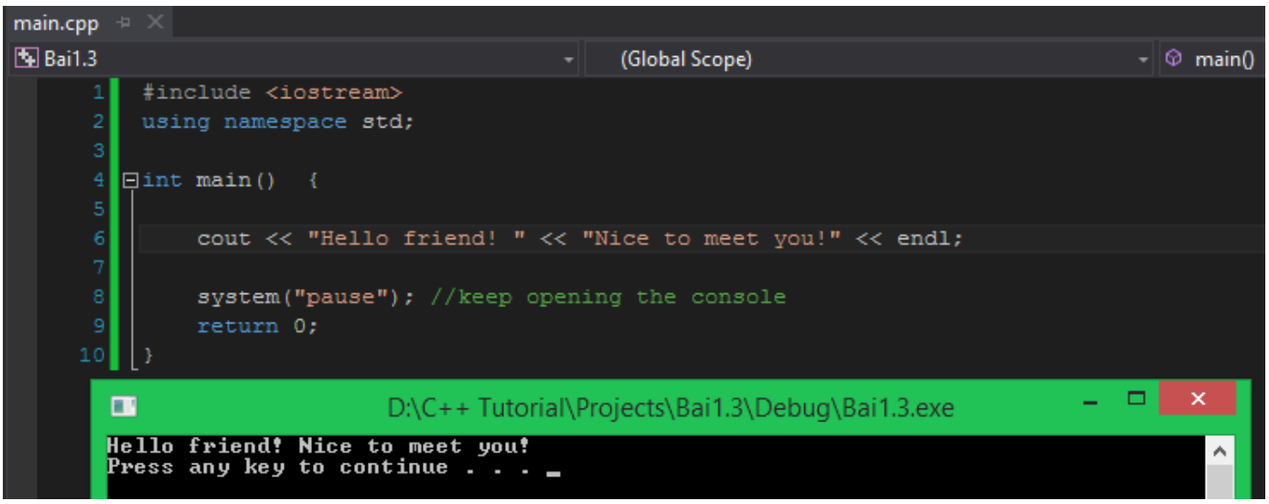
Nhìn vào kết quả, chúng ta thấy đã có sự khác biệt so với ban đầu.

Chúng ta còn có thể nối nhiều đoạn kí tự để in ra màn hình cùng lúc chỉ với 1 lần sử dụng lệnh **cout** bằng cách sử dụng nhiều lần toán tử **<<**



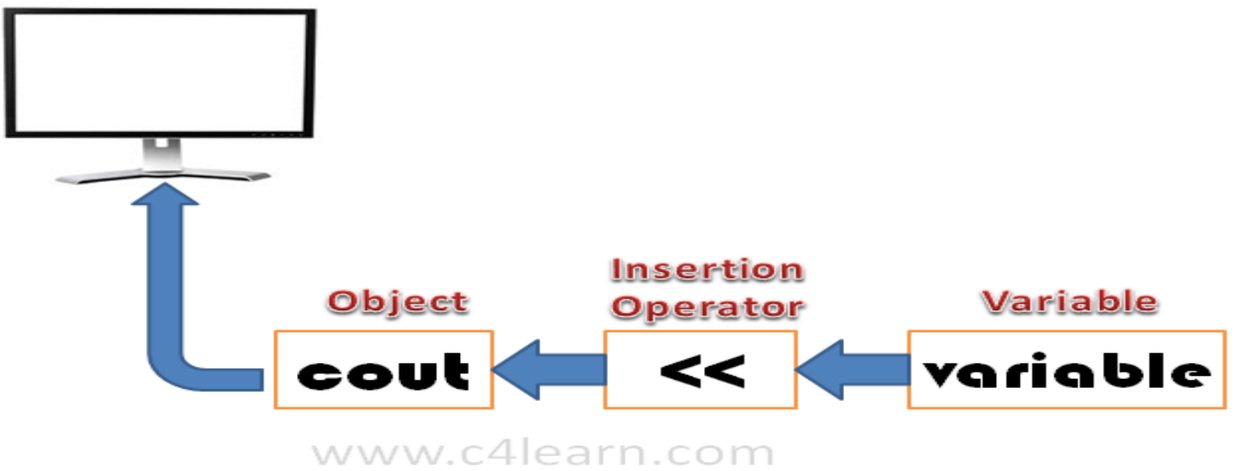
Dòng dữ liệu chúng ta in ra vẫn chưa được đẹp mắt lắm. Hai câu "Hello friend!" và "Nice to meet you!" được truyền lần lượt theo thứ tự vào đối tượng file **stdout** thông qua lệnh **cout**, nhưng khi sử dụng nhiều lần toán tử **<<**, nó không tự động sinh ra khoảng trắng giữa các chuỗi riêng biệt, mà nó cứ nối vào nhau cho đến khi đến giới hạn số kí tự cho phép trên 1 dòng của **console** thì mới xuống dòng.

Vì thế, chúng ta cần điều chỉnh lại một chút. (Thêm 1 kí tự trắng sau câu đầu tiên)



Hai câu chúng ta in ra giờ đã rõ đẹp hơn phải không nào?

Qua ví dụ trên, chắc các bạn cũng phần nào hình dung được cách thức hoạt động của lệnh **cout**.



(Nguồn: http://www.c4learn.com)

Ngoài việc sử dụng đối tượng **cout** để in các chuỗi kí tự lên màn hình, bạn còn có thể in những con số cụ thể.

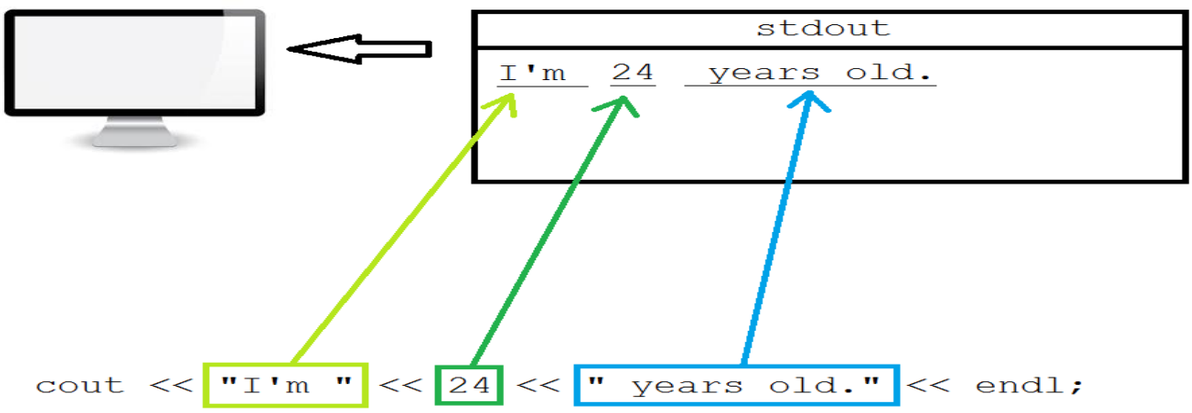
```
main.cpp
Bai1.3                    (Global Scope)                    main()
1   #include <iostream>
2   using namespace std;
3
4   int main()  {
5
6       cout << "Hello friend! " << "Nice to meet you!" << endl;
7       cout << "I'm " << 24 << " years old." << endl;
8
9       system("pause"); //keep opening the console
10      return 0;
11  }
```

```
D:\C++ Tutorial\Projects\Bai1.3\Debug\Bai1.3.exe
Hello friend! Nice to meet you!
I'm 24 years old.
Press any key to continue . . .
```

Mình vừa thêm dòng bên dưới vào chương trình.

```
cout << "I'm " << 24 << " years old." << endl;
```

Chúng ta cùng xem dòng này hoạt động như thế nào.



Khi bạn muốn in một giá trị lên màn hình, bạn có thể làm nhiều cách khác nhau. Bạn có thể đưa số đó vào trong cặp dấu ngoặc kép để biến nó thành chuỗi kí tự.

```
cout << "I'm 24 years old." << endl;
```

Bạn có thể đưa nó ra ngoài cặp dấu ngoặc kép (nhớ sử dụng thêm toán tử **<<** nữa, vì kiểu chuỗi kí tự và kiểu số là hai loại kiểu dữ liệu khác nhau, nên cần tách chúng ra bằng toán tử **<<** để lệnh cout có thể hiểu được).

```
cout << "I'm " << 24 << " years old." << endl;
```

Hoặc có một cách khác mà chúng ta sẽ dùng thường xuyên hơn trong các bài học sau, đó là đưa giá trị vào một biến số.

```
int myAge = 24;
cout << "I'm " << myAge << " years old." << endl;
```

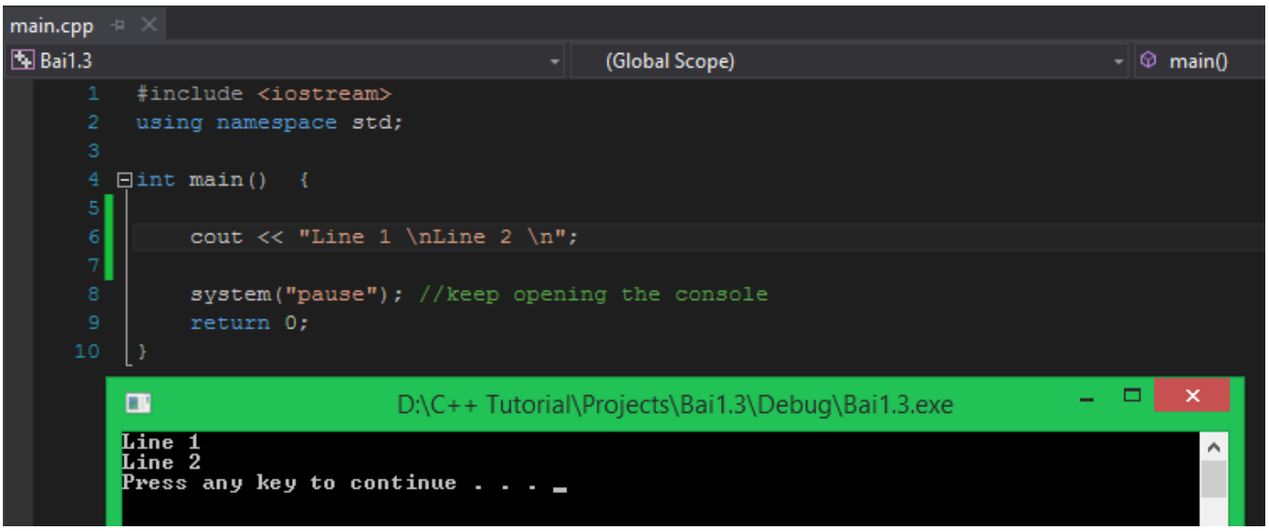
Cả 3 cách trên đều cho ra kết quả giống nhau.

Trong ngôn ngữ lập trình C++, có một số kí tự trên bàn phím chúng ta không thể đưa trực tiếp vào cặp dấu ngoặc kép để in ra màn hình trong lệnh **cout** được. Chúng ta cần định dạng chúng lại một chút. Sau đây là bảng một số kí tự đặc biệt và cách để in chúng ra màn hình:

Value	Escape sequence
newline	\n
horizontal tab	\t
vertical tab	\v
backspace	\b
carriage return	\r
form feed	\f
alert	\a
backslash	\\
question mark	? or \?
single quote	\'
double quote	\"
the null character	\0
octal	\ooo
hexadecimal	\xhhh
Unicode (UTF-8)	\uxxxx
Unicode (UTF-16)	\Uxxxxxxxx

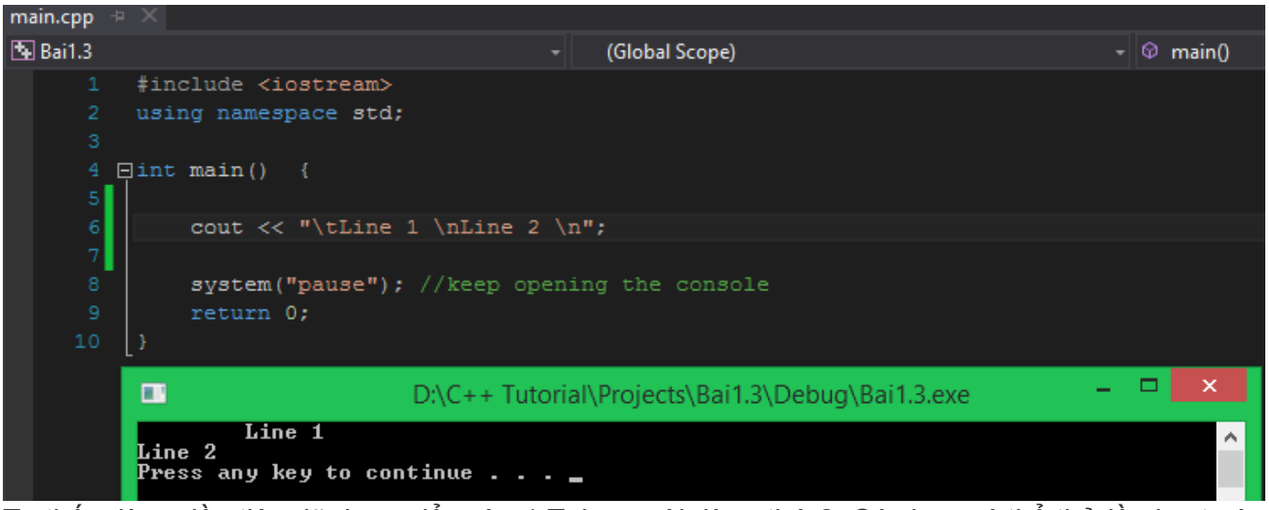
(Nguồn: https://msdn.microsoft.com)

Chúng ta cùng thử dùng một vài kí tự trong bảng trên và xem kết quả.
Đầu tiên là sử dụng kí tự xuống dòng:



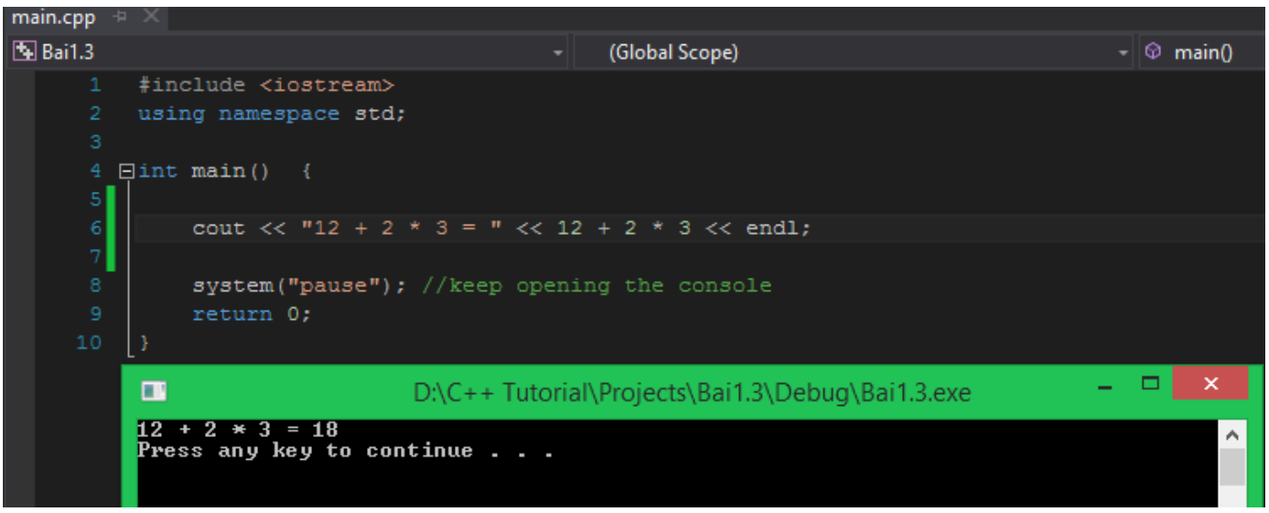
Như các bạn thấy, không còn sử dụng đối tượng **endl** nữa nhưng chuỗi kí tự trên vẫn được tách thành 2 dòng bằng cách sử dụng kí tự new line **"\n"**.

Tiếp theo, chúng ta thêm kí tự **Tab** và đầu chuỗi kí tự muốn in ra:



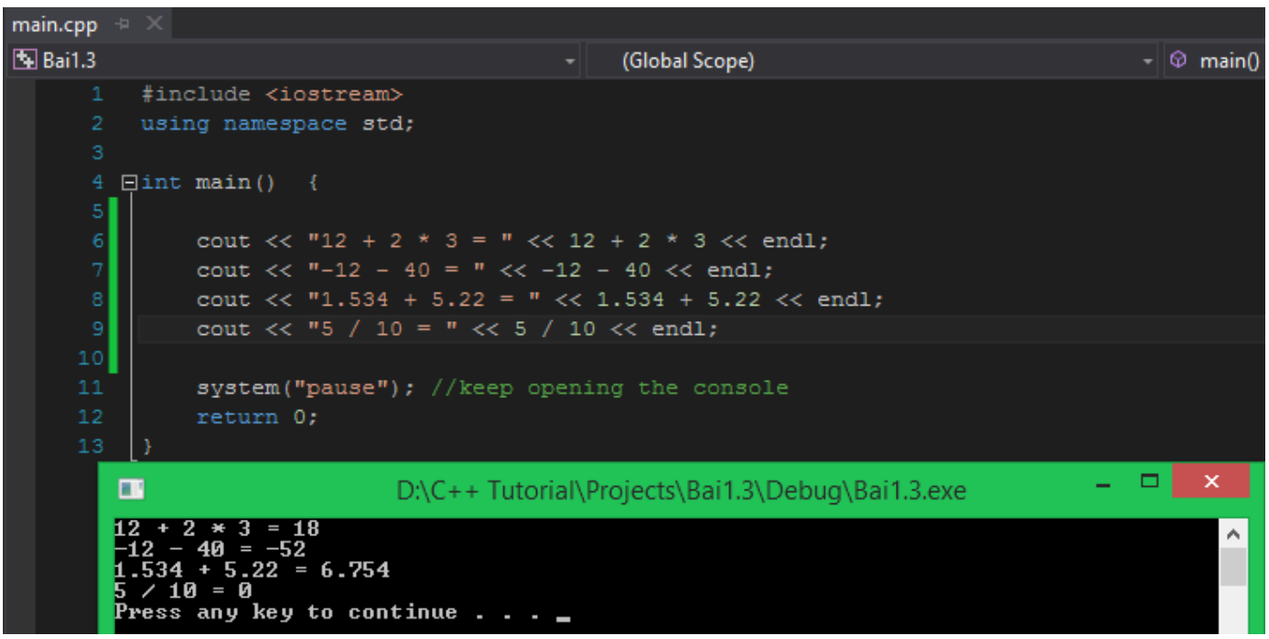
Ta thấy dòng đầu tiên đã được đẩy vào 1 Tab so với dòng thứ 2. Các bạn có thể thử lần lượt các kí tự đặc biệt trên nếu có thời gian. Tuy nhiên, cần lưu ý rằng kí tự new line "\n" và đối tượng **endl** đều đóng vai trò là kí tự xuống dòng nhưng nó hoàn toàn khác nhau, mình sẽ giải thích vấn đề này sau.

Ngoài ra, các bạn còn có thể sử dụng đối tượng **cout** để in ra kết quả của một biểu thức toán học:



Trong câu lệnh trên, biểu thức **12 + 2 * 3** được tính ra kết quả, kết quả biểu thức này được chương trình coi như một giá trị số, và nó hoàn toàn có thể đưa vào đối tượng file **stdout** bằng đối tượng **cout**.

Ngoài tính toán và cho ra kết quả số nguyên, một chương trình C++ còn có thể tính toán các biểu thức và in ra giá trị là số âm, số thực...



- Chương trình C++ không thể in ra giá trị là một phân số (trừ khi chúng ta tự định nghĩa lại), vì thế, ở biểu thức cuối cùng, máy tính không in ra được giá trị là **5/10** mà nó chỉ có thể in ra giá trị 0 (tại sao lại không phải là 0.5?), chúng ta sẽ tìm hiểu vấn đề này trong bài học **Biến - cách khai báo và sử dụng biến**.

- Compiler của Visual studio sẽ báo lỗi nếu nó bắt gặp biểu thức có dạng **x / 0**. Ví dụ:

```
cout << 5 / 0 << endl; //This command makes an error
```

Sử dụng thư viện iomanip

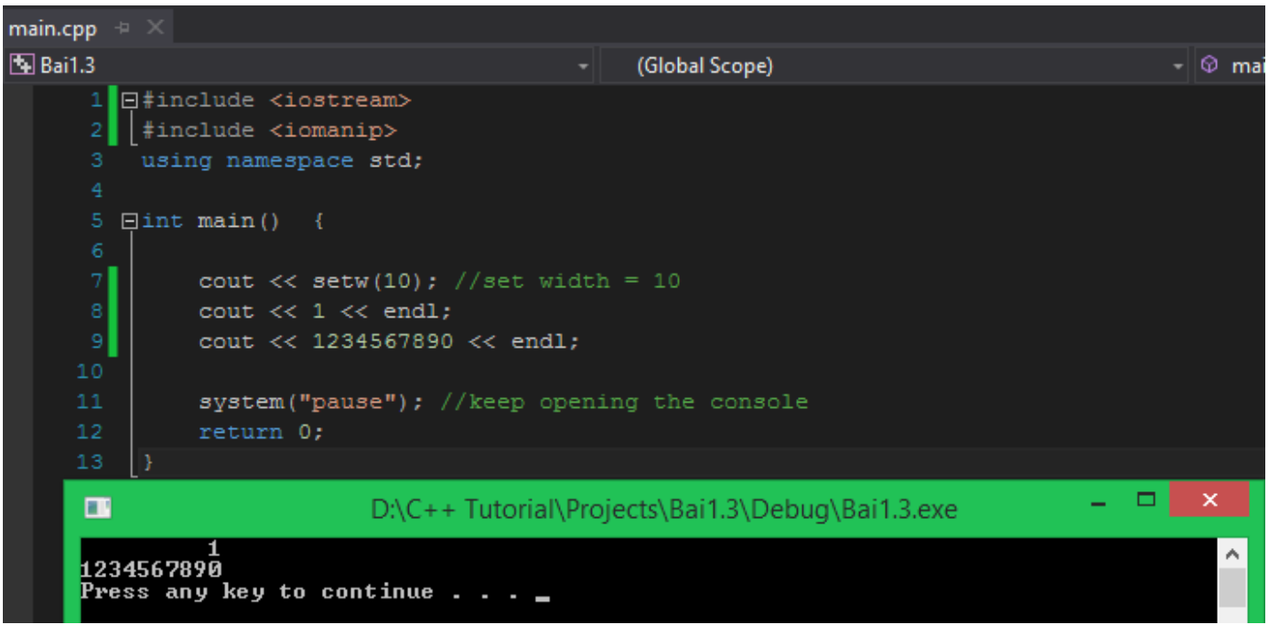
iomanip viết tắt của cụm từ **iostream manipulator** là một thư viện thuộc namespace **std**, nó định nghĩa một số hàm giúp lập trình viên có thể định dạng output.

Trong bài này, chúng ta chỉ làm quen với một số hàm đơn giản thường xuyên được sử dụng.

setw(int n)

setw là một hàm cho phép giới hạn độ rộng của một giá trị được xuất lên màn hình.

Cách sử dụng:

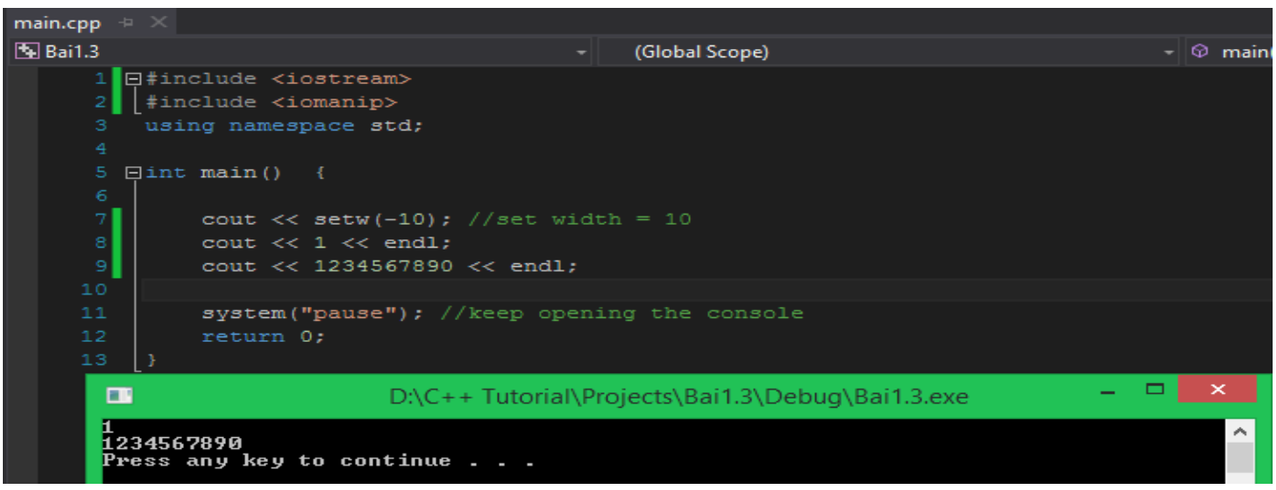


Cùng nhìn vào hình trên để xem cú pháp sử dụng và đánh giá kết quả.

- Đầu tiên, chúng ta include thêm thư viện **iomanip** vào chương trình.
- Tiếp theo, gọi hàm **setw(int n)** với n là một số nguyên (ví dụ: setw(8)) để định dạng độ rộng cho kiểu dữ liệu số, ngay lúc này, bất kì giá trị số nào được truyền vào luồng output stream thông qua **cout** đều bị đối tượng **cout**format có độ rộng là 10 ô.
- Cuối cùng thì thử truyền vài giá trị số nguyên vào để kiểm chứng thôi.

Trong hình trên, mình định dạng độ rộng của các số được đưa vào **cout** có độ rộng là 10, mình truyền thử 2 số nguyên mà số đầu tiên chỉ có 1 chữ số, số thứ 2 thì có 10 chữ số. Kết quả cho thấy có 9 khoảng trắng thừa đứng trước số 1 ở dòng đầu tiên.

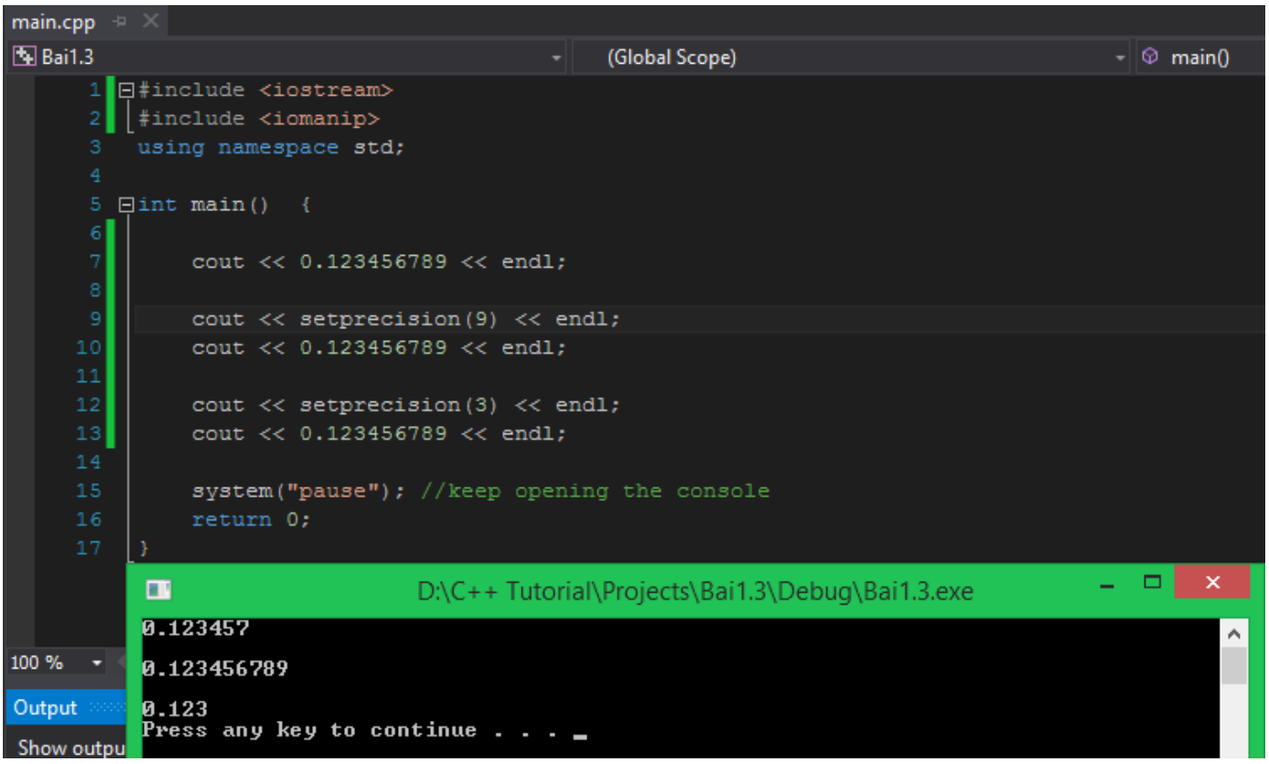
Nếu các bạn muốn định dạng khoảng trắng phía sau, chỉ cần đổi giá trị trong hàm **setw** thành số âm như hình bên dưới:



setprecision(int n)

Cũng tương tự như hàm **setw**, hàm **setprecision** cũng nhận vào một giá trị số nguyên, nhưng mục đích của hàm này là định dạng số lượng chữ số trong phần thập phân của kiểu số thực.

Cách sử dụng:



- Đảm bảo rằng thư viện **iomanip** đã được include vào chương trình.
- Đẩy định dạng độ chính xác **setpresicion(int n)** và đối tượng **cout** thông qua toán tử **<<**, sau thời điểm này, mọi số thực có phần thập phân sẽ được định dạng lại, với độ dài phần thập phân bằng với số nguyên mà bạn đặt trong hàm **setprecision**.

Nhìn vào kết quả của đoạn chương trình trên, ta thấy rằng mặc định phần thập phân của một số thực chỉ có 6 chữ số. Sau khi định dạng lại với hàm **setprecision(9)** thì độ chính xác đã lên đến 9 chữ số.

Tổng kết

Đến đây, chúng ta đã nắm được cú pháp và cách hoạt động của đối tượng **cout** thuộc thư viện **iostream** trong **namespace std**. Các bạn đã biết cách:

- In một dòng chữ lên màn hình console.
- In liên tiếp nhiều chuỗi kí tự trong một lần sử dụng đối tượng **cout**.
- In giá trị số nguyên, số thực.
- In các kí tự đặc biệt "\n", "\t", ... lên màn hình.
- In kết quả của một biểu thức.
- Một số định dạng cơ bản với số nguyên và số thực.

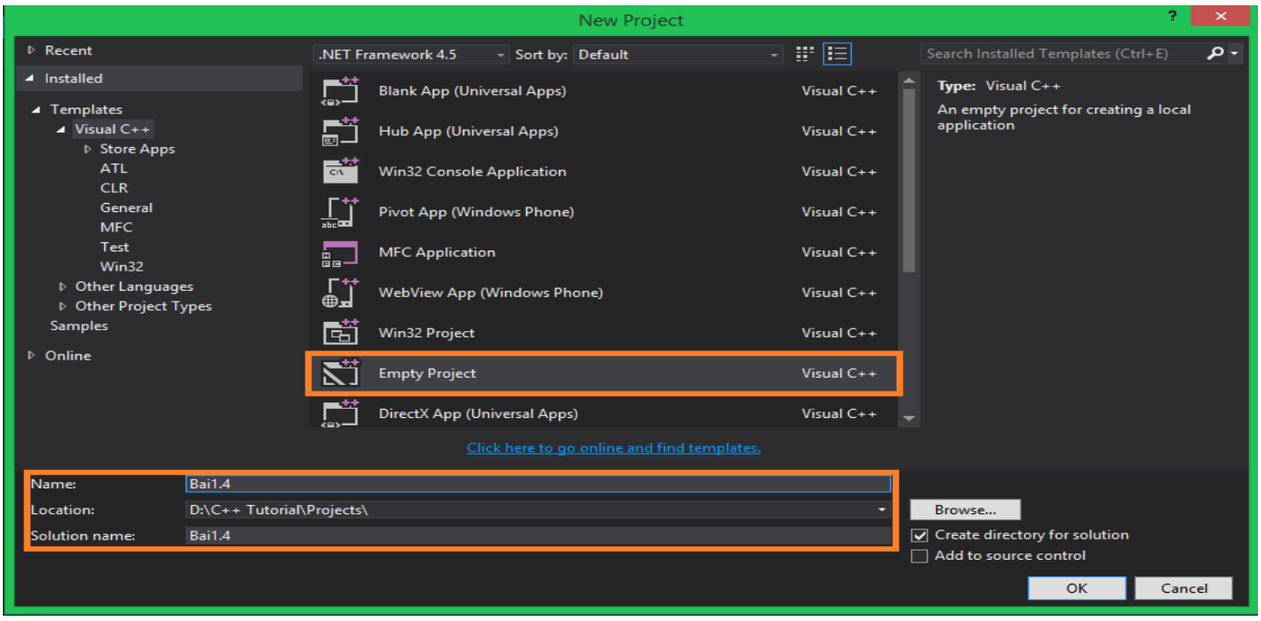
Bài tập rèn luyện

1. Viết chương trình in kết quả 4 phép tính +, -, *, / của 2 số.
2. Từ chương trình đã viết được ở câu 1, thêm vào 1 dòng lệnh khiến chương trình phát ra 1 âm báo.

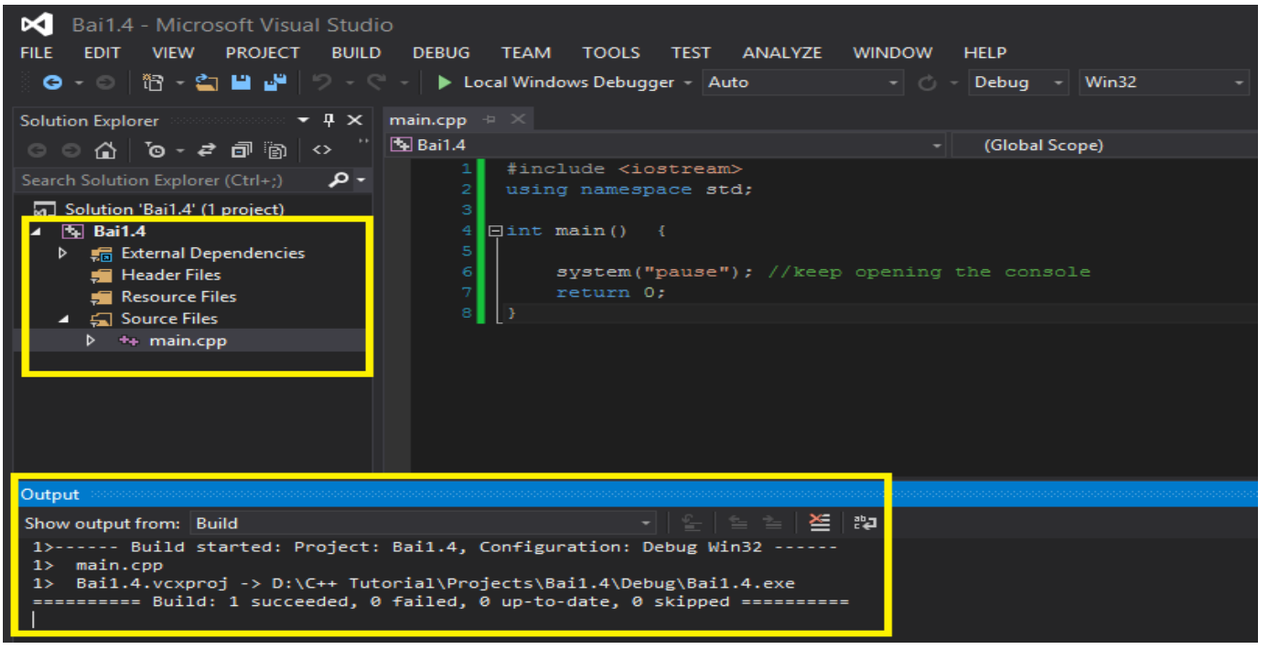
1.4 Biến và các kiểu dữ liệu trong C++

Trong bài học hôm nay, chúng ta sẽ tìm hiểu cơ bản về việc lưu trữ và sử dụng dữ liệu.

Như thường lệ, việc đầu tiên chúng ta làm là tạo một project C++ mới (Mình đặt tên project là Bai1.4 để tiện theo dõi, còn các bạn thích đặt tên project là gì cũng được).



Sau khi Visual studio thiết lập cấu hình cho project, ta tạo file main.cpp và viết một số dòng lệnh tạo nên cấu trúc cơ bản của chương trình C++.

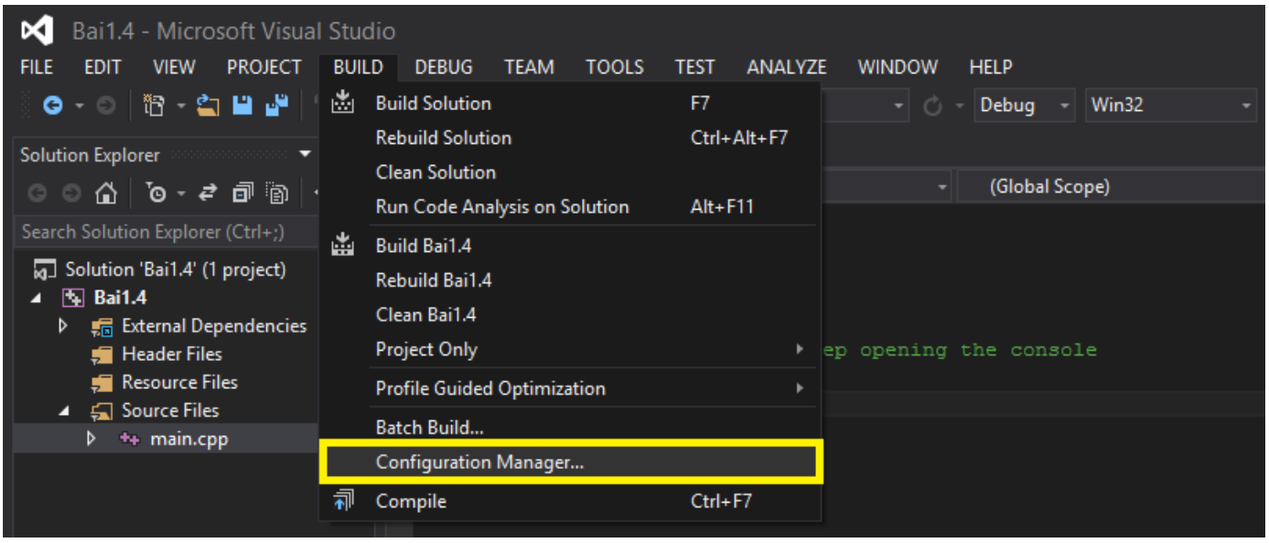


Sau đó nhấn tổ hợp phím **Ctrl + Shift + B** để thực hiện quá trình biên dịch file main.cpp thành file main.obj, và tạo thành file execute. Việc build chương trình trong giai đoạn đầu này nhằm đảm bảo mọi thứ hoạt động ổn định, và tiết kiệm thời gian cho những lần build sau.

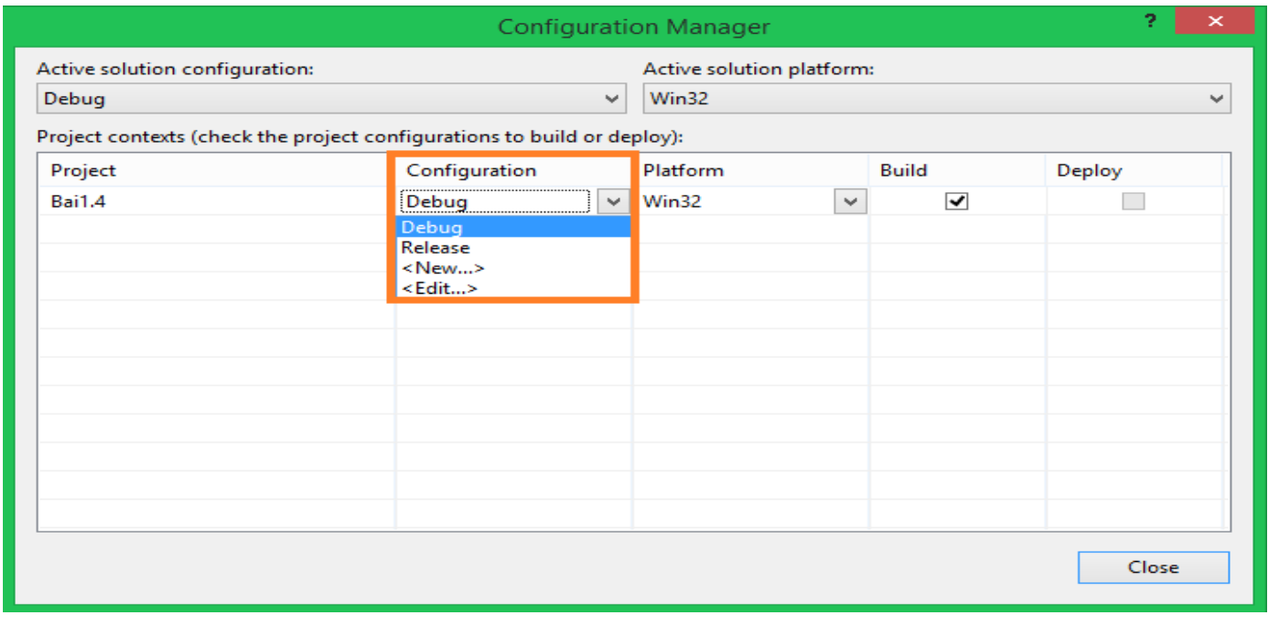
Mình xin phép dành thêm một ít thời gian để nói về cấu hình build ứng dụng của Visual studio 2015.

Khi thực hiện build project, Visual studio cung cấp cho chúng ta hai lựa chọn: **Debug** và **Release**.

Để chọn cấu hình build ứng dụng, các bạn vào **BUILD** trên thanh Menu Bar, chọn **Configuration Manager**.



Cửa sổ chọn cấu hình xuất hiện như bên dưới:



Hai lựa chọn này khác nhau như thế nào? Có thể hiểu build chương trình dưới cấu hình **Debug** thì sản phẩm của chúng chưa được hoàn thiện, cần thời gian để tìm lỗi, sửa lỗi... Và một khi bạn cho rằng sản phẩm của mình đã được viết hoàn tất, bạn chuyển sang cấu hình **Release** để build sản phẩm. Sản phẩm sau khi **Release** thường sẽ có dung lượng nhẹ hơn khi **Debug**, vì khi trong chế độ **Debug**, Visual studio sẽ tích hợp một số thư viện nhằm phục vụ cho quá trình tìm kiếm và sửa lỗi. Đây chỉ là phần ngoài lề của bài học, bây giờ chúng ta quay lại với nội dung chính.

Biến (Variable)

Khái niệm **biến** (variable) ra đời đã giải quyết được rất nhiều mặt hạn chế trong các chương trình mà các bạn đã viết trong các bài học trước. Ví dụ với chương trình tính kết quả biểu thức chứa 2 số nguyên, các bạn phải viết một vài dòng lệnh, biên dịch chương trình, chạy chương trình để in ra kết quả. Sau đó thì sao? Khi bạn muốn tính kết quả của phép tính trên với 2 giá trị khác, các bạn lại phải viết lại một vài dòng lệnh, và thực hiện quá trình trên lặp đi lặp lại.

Đây không phải là lập trình! Người viết chương trình phải đảm bảo chương trình sau khi viết ra phải có tính tổng quát, nghĩa là chương trình đó phải giải được một bài toán nào đó với nhiều giá trị đầu vào khác nhau mà không phải thay đổi bất kỳ đoạn code nào bên trong.

Để tiết kiệm thời gian cho việc này, chúng ta sẽ sử dụng **biến** (variable) để lưu trữ giá trị cần xử lý.

Ví dụ:

```
int var1 = 3;
int var2 = 5;
cout << "var1 + var2 = " << var1 + var2 << endl;
```

Trong đoạn code trên, var1 là một biến và var2 cũng là một biến. Hai biến này hiện đang lưu trữ hai giá trị khác nhau (cũng có thể sẽ trùng nhau). Và dòng lệnh **cout** bên dưới sẽ in ra kết quả phép cộng của hai biến này.

Và khi chúng ta muốn thay đổi giá trị của hai số cần tính, chúng ta đơn giản truyền vào cho hai biến này hai giá trị khác.

```
var1 = 10;
var2 = 20;
cout << "The new result: " << var1 * var2 << endl;
```

Như đoạn code trên, chúng ta sử dụng lại hai biến var1 và var2 mà không cần biên dịch lại chương trình hay viết lại bất kỳ đoạn code nào. Việc tái sử dụng tiết kiệm thời gian và công sức cho chúng ta rất nhiều.

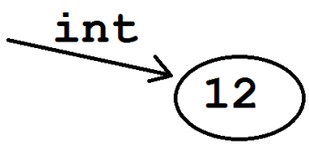
Đến đây sẽ có nhiều bạn vẫn chưa hình dung được **biến** là cái gì. Các bạn có thể hiểu như thế này:

Biến là một ô nhớ đơn lẻ hoặc một vùng nhớ được hệ điều hành cấp phát cho chương trình C++ nhằm để lưu trữ giá trị vào bên trong vùng nhớ đó. Chúng ta sẽ hỏi xin hệ điều hành những vùng nhớ thông qua các câu lệnh khai báo biến như bên dưới:

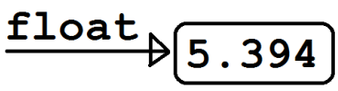
```
int valueInteger = 12; // (1) Chúng ta có một vùng nhớ để chứa 1 số nguyên

float valueFloat = 5.394; // (2) Xin hệ điều hành thêm một vùng nhớ để chứa 1 số thực

string myName = "Minh Vu"; // (3) Biến này chứa được một dãy các kí tự
```

Với 3 cách khai báo trên, chúng ta sẽ có 3 vùng nhớ nằm ở 3 vị trí khác nhau trên RAM của bạn.

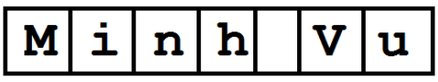
Với cách khai báo 1:



Với cách khai báo 2:



Với cách khai báo 3:



Sau đây là cú pháp cơ bản để chúng ta khai báo 1 biến:

```
<Kiểu dữ liệu> <Tên biến> = [Giá trị khởi tạo của biến];
```

Trong đó, tên biến giúp ngôn ngữ C++ xác định vùng nhớ mà chúng ta đã cấp phát, mỗi lần sử dụng biến, compiler sẽ tìm đến vùng nhớ mà chúng ta đã đặt tên cho nó và lấy giá trị ra để sử dụng. Kiểu dữ liệu sẽ phân loại giá trị của biến (kí tự, số nguyên, số thực, ...), chúng ta sẽ làm rõ hơn ở phần dưới của bài học này.

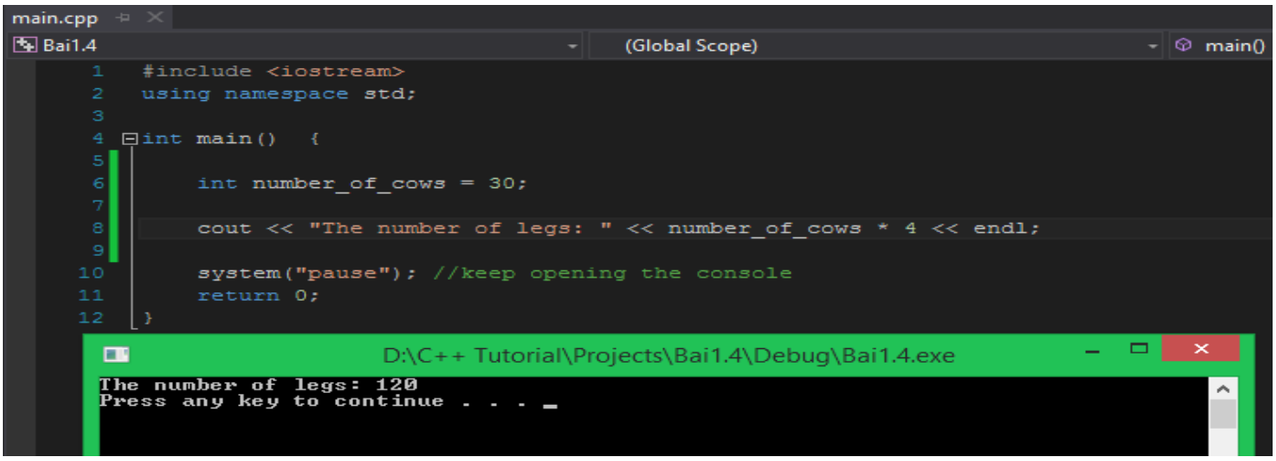
Khi khai báo biến, chúng ta có thể gắn 1 giá trị ban đầu cho biến hoặc không, nhưng chúng bản chất là một câu lệnh, nên chúng ta phải kết thúc bằng **dấu chấm phẩy**.

Lý thuyết nhiều rồi, bây giờ chúng ta sẽ tận dụng sức mạnh lưu trữ dữ liệu của biến để giải một bài toán cơ bản.

Bài toán: Nhà mình có nuôi 1 đàn bò, mình đã biết số lượng bò ở thời điểm hiện tại. Nhưng vì mình tính toán chậm nên gặp khó khăn trong việc tính số chân bò của đàn bò đang nuôi. Mình muốn đưa máy tính tính giúp mình, các bạn có thể giúp mình viết chương trình tính tổng số chân bò của cả đàn được không? (Các bạn có thể nghĩ ra một con số đại diện cho số bò mà mình đang nuôi, sao cho nó là số nguyên dương là được)

Sau khi viết xong đề của bài toán thì mình đã nghĩ ra giải pháp giải quyết bài toán này rồi. Chúng ta biết 1 con bò thì có 4 chân, vậy là mình chỉ cần 1 biến để chứa số bò hiện tại, và lấy giá trị của biến đó nhân với 4 là ra kết quả.

Đây là chương trình do mình viết, mình khuyên các bạn nên tự nghĩ cách viết trước khi tham khảo chương trình của mình.

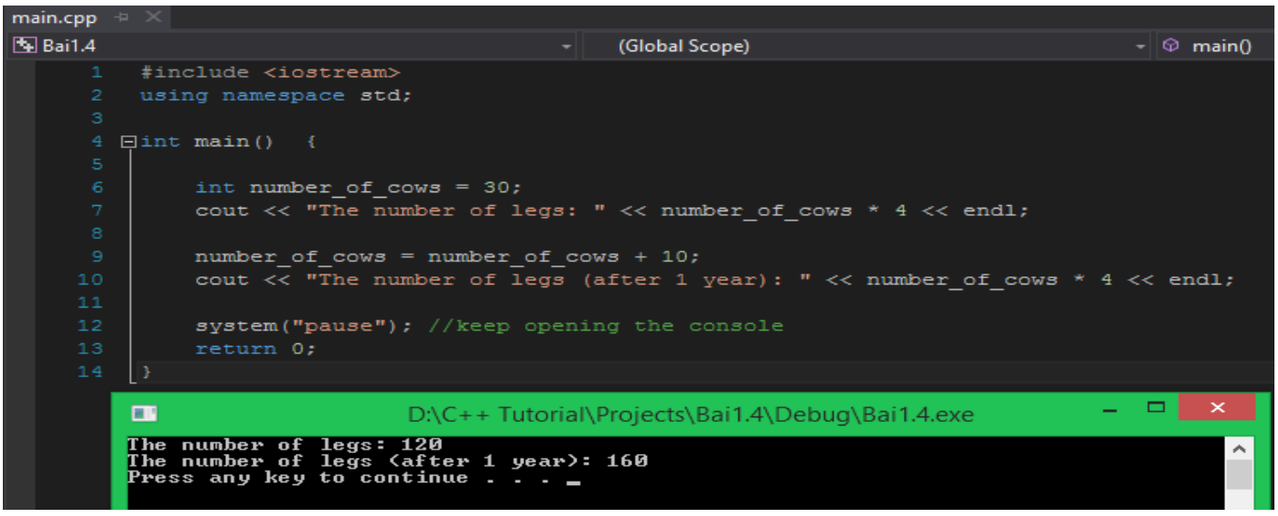


Hi vọng chương trình của bạn các viết cũng cho kết quả đúng như mong đợi.

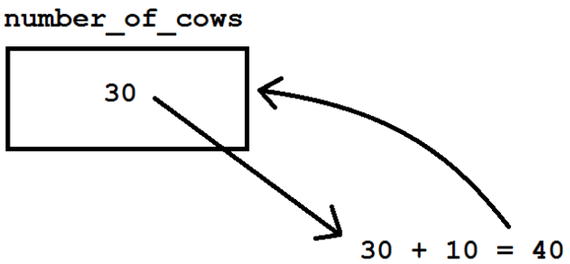
Bây giờ có một vấn đề phát sinh, đàn bò của mình sau một năm đã sinh thêm 10 con bò con. Bây giờ làm sao để mình tính tổng số chân của đàn bò mới?

Hướng giải quyết: Lấy số lượng bò cũ cộng thêm 10 con bò mới sinh, và lấy số lượng bò mới này nhân 4.

Chương trình của mình sẽ được viết lại như sau:



Ở đoạn code trên, dòng `number_of_cows = number_of_cows + 10;` có nghĩa là:



Chúng ta dùng toán tử **"="** để đưa một giá trị vào trong biến number_of_cows, giá trị mà chúng ta đưa vào sẽ bằng giá trị của biến `number_of_cows` hiện tại (đang là 30) cộng thêm 10. Sau khi thực hiện dòng lệnh này, giá trị mới trong biến `number_of_cows` sẽ là 40.

Chúng ta đã sử dụng lại biến `number_of_cows` để tính tiếp số chân bò sau 1 năm mà không cần phải viết lại chương trình tính số chân bò nữa.

Kiểu dữ liệu

Kiểu dữ liệu là một thành phần bắt buộc phải có khi muốn khai báo biến, nó giúp chương trình xác định kích cỡ của vùng nhớ mà bạn muốn xin hệ điều hành cấp phát trên RAM, đồng thời giúp chương trình xác định giới hạn giá trị mà biến đó có thể lưu trữ.

Dưới đây là bảng mô tả một số kiểu dữ liệu cơ bản:

Category	Type	Minimum Size	Note
boolean	bool	1 byte	
character	char	1 byte	May be signed or unsigned
	wchar_t	1 byte	
	char16_t	2 bytes	C++11 type
	char32_t	4 bytes	C++11 type
integer	short	2 bytes	
	int	2 bytes	
	long	4 bytes	
	long long	8 bytes	C99/C++11 type
floating point	float	4 bytes	
	double	8 bytes	
	long double	8 bytes	

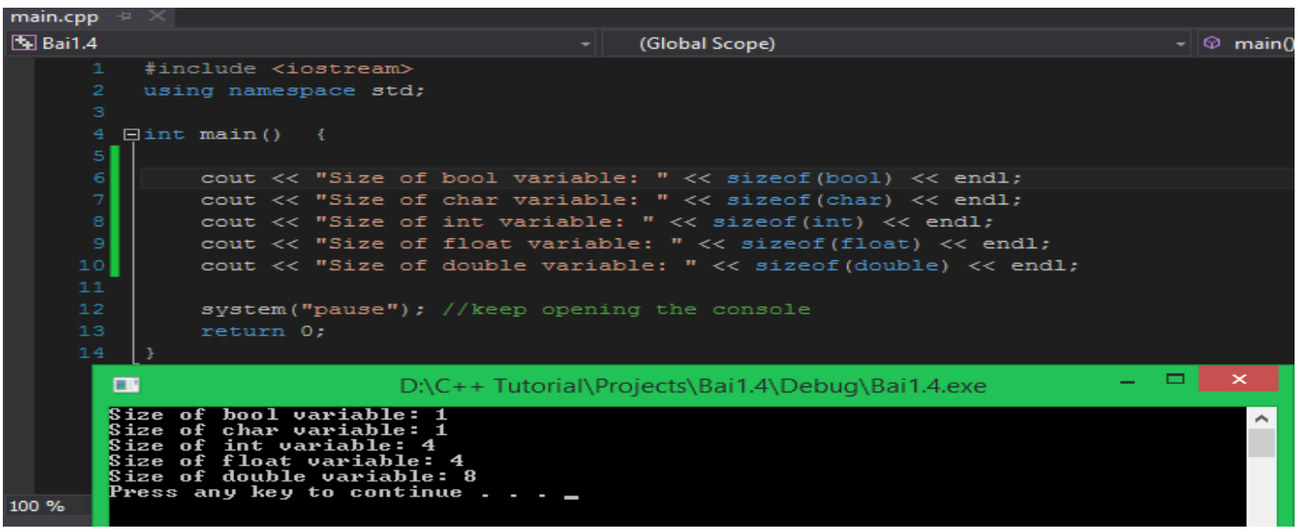
(Nguồn: www.learncpp.com)

Trong cột Category, người ta chia các kiểu dữ liệu cơ bản thành 4 loại:

- Kiểu logic (boolean): **bool**.
- Kiểu kí tự (character): **char**, wchar_t, char16_t, char32_t. Hiện tại chúng ta chỉ cần quan tâm đến kiểu **char**.
- Kiểu số nguyên (integer): short, **int**, long, long long. Thường dùng nhất là kiểu **int**.
- Kiểu số thực (floating): **float**, **double**, long double. Tùy vào độ chính xác mà chúng ta đòi hỏi ở phần thập phân mà chúng ta chọn kiểu dữ liệu **float** hoặc **double**. Kiểu double có kích thước vùng nhớ lớn hơn, nên sẽ lưu giá trị có độ chính xác cao hơn **float**.

Kích thước vùng nhớ của các kiểu dữ liệu này được tính bằng đơn vị **byte**. Đối với các bạn mới học ngôn ngữ lập trình, chúng ta chưa cần quan tâm nhiều đến khái niệm này.

Để xem kích thước vùng nhớ sẽ được cấp phát cho biến trong chương trình. Chúng ta sử dụng toán tử **sizeof()** như sau:



Theo kết quả của chương trình, kiểu dữ liệu số nguyên **int** có kích cỡ 4 **bytes**, đó là do IDE Visual studio 2015 định nghĩa kiểu **int** như vậy. Mỗi compiler sẽ có một chuẩn kiểu dữ liệu riêng, nên các bạn không cần lo lắng về sự khác biệt giữa kích bảng dữ liệu ở trên với kết quả thực tế.

Các bạn hiện tại chỉ cần hiểu với những kiểu dữ liệu có kích thước càng lớn thì phạm vi giá trị có thể lưu trữ cho biến càng lớn.

Dưới đây là bảng giới hạn giá trị cho từng kiểu dữ liệu mà chúng ta thường xuyên sử dụng trong Visual studio:

Type name	Bytes	Range of values
bool	1	false or true
char	1	-128 to 127
int	4	-2147483648 to 2147483647
unsigned int	4	0 to 4294967295
__int16	2	-32768 to 32767
__int32	4	-2147483648 to 2147483647
__int64	8	-9223372036854775808 to 9223372036854775807
float	4	3.4E +/- 38 (7 digits)
double	8	1.7E +/- 308 (15 digits)

Các bạn không cần nhớ chính xác những giới hạn trên, chỉ cần ước chừng phạm vi của mỗi kiểu dữ liệu để chọn kiểu dữ liệu phù hợp cho biến là được.

Tổng kết

Trong bài này, các bạn chỉ cần hiểu được một số khái niệm về biến và kiểu dữ liệu:

- Biến (variable) là một đối tượng chiếm giữ một vùng nhớ xác định.
- Biến (variable) dùng để lưu trữ giá trị (kí tự, số nguyên, số thực ... hoặc cũng có thể là một dãy các con số).
- Cú pháp khai báo biến:

  ```
  <kiểu dữ liệu> <tên biến> [= giá trị khởi tạo];
  ```
- Kiểu dữ liệu đứng trước tên biến nhằm xác định giới hạn giá trị mà biến có thể lưu trữ được.
- Đặt tên biến như thế nào cho phù hợp?
  - Tên biến không được bắt đầu bằng kí tự số.
  - Trong một khối lệnh { } không được có hai biến cùng tên.
  - Tên biến trong C++ phân biệt chữ hoa và chữ thường. Ví dụ: `int var1;` và `int Var1;` là hai biến phân biệt.
  - Các bạn có thể sử dụng kí tự gạch chân khi đặt tên biến.
  - Nên đặt tên biến sao cho thể hiện được ý nghĩa của biến. Ví dụ:

    ```
    int myAge;
    string myName; // các bạn sẽ được học về kiểu string trong các bài học sau
    bool isRunning = true;
    int current_score;
    ```
- Khi sử dụng biến trong các biểu thức toán học, các bạn nên dùng biến có cùng kiểu dữ liệu với nhau. (Tùy trường hợp mà có thể dùng kết hợp nhiều kiểu dữ liệu khác nhau)

1.5 Nhập và xuất dữ liệu

Sau khi học xong bài **Biến và các kiểu dữ liệu**, các bạn đã có thể tự mình giải một số bài toán cơ bản trên máy tính. Lấy một ví dụ cơ bản như sau:

Viết chương trình tính tổng giá trị hai số nguyên.

Mình tin rằng tất cả chúng ta ai cũng viết được một chương trình như thế này:

```cpp
#include <iostream>
using namespace std;

int main()  {

    int number1 = 4, number2 = 5;

    cout << number1 << " + " << number2 << " = " << number1 + number2 << endl;

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

```
D:\C++ Tutorial\Projects\Bai1.5\Debug\Bai1.5.exe
4 + 5 = 9
Press any key to continue . . .
```

Các bạn có thể bắt gặp một cách khai báo biến hơi lạ.

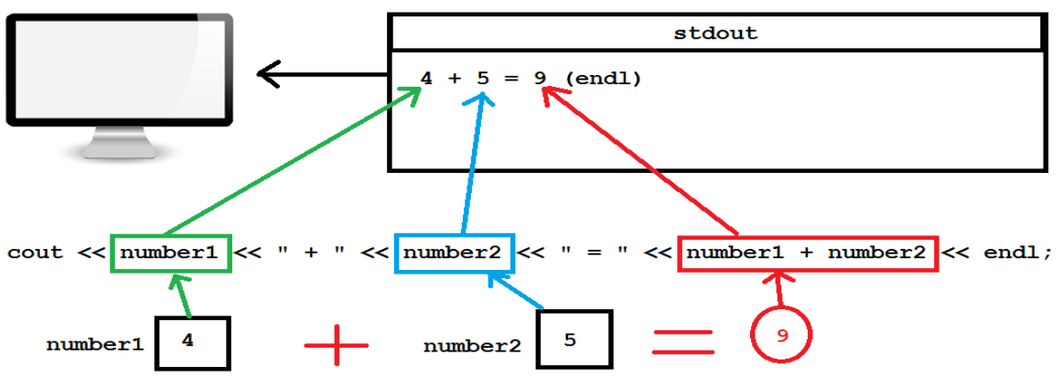
```cpp
int number1 = 4, number2 = 5;
```

Đây là cách khai báo các biến có cùng kiểu dữ liệu trên cùng một dòng đồng thời khởi tạo giá trị ban đầu cho mỗi biến. **Khi khai báo nhiều biến trên cùng 1 dòng, mỗi biến được khai báo sẽ cách nhau bằng 1 dấu phẩy.** Điều này hoàn toàn được cho phép trong C++.

Sau đó, chúng ta có sử dụng một lần đối tượng **cout** để in kết quả phép cộng ra màn hình:

```cpp
cout << number1 << " + " << number2 << " = " << number1 + number2 << endl;
```

Để các bạn khỏi bị rối khi nhìn thấy dòng lệnh **cout** phức tạp, các bạn theo dõi hình bên dưới để rõ hơn:



Sau khi chạy chương trình, chúng ta đã có được kết quả phép tính cộng ở trên màn hình. Bây giờ mình đặt ra trường hợp, mình muốn tính tổng của hai số nguyên có giá trị khác. Các bạn sẽ làm gì để giúp mình giải quyết vấn đề này? Có phải các bạn đang nghĩ tới việc tắt chương trình đang chạy đi, vào file **main.cpp**, thay hai số 4 và 5 thành hai con số khác, biên dịch và chạy lại chương trình một lần nữa?

Đó cũng là một cách giải quyết vấn đề mà vẫn cho kết quả đúng, nhưng có lẽ mình tự tính nhẩm trong đầu còn nhanh hơn.

Vì thế, ngôn ngữ C++ đã hỗ trợ cho chúng ta một cách để đưa giá trị vào biến trực tiếp ngay khi chương trình đang chạy. Điều này giúp chúng ta linh động hơn khi cần thay đổi giá trị tính toán mà không cần phải build lại chương trình sau khi đã cho ra **file execute**.

Việc **Input data** như trên có thể thực hiện bằng nhiều cách khác nhau:

- Lấy dữ liệu từ **File**.
- Sinh dữ liệu ngẫu nhiên.
- Nhận dữ liệu từ thiết bị khác gửi đến.
- Nhập dữ liệu từ thiết bị nhập chuẩn. (Trong C++, thiết bị nhập dữ liệu chuẩn là bàn phím)

Ở mức cơ bản này, chúng ta sẽ làm quen với cách nhập dữ liệu thông qua thiết bị nhập chuẩn (Standard Input) của C++.

Standard Input

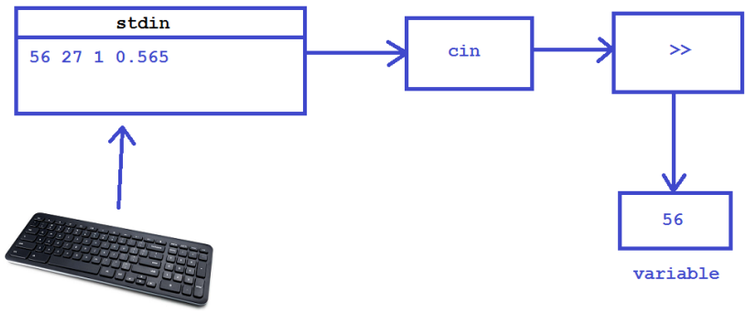
Để đưa dữ liệu vào biến, chúng ta sử dụng đối tượng **cin** được định nghĩa bên trong thư viện **iostream** thuộc namespace **std**.

Cú pháp sử dụng đối tượng **cin** như sau:

```
cin >> <tên biến>;
```
Các bạn lưu ý là tên biến phải được khai báo trước khi sử dụng đối tượng **cin** để đưa giá trị vào biến đó. Ta sử dụng toán tử **>>** ngược chiều với toán tử được sử dụng cho từ khóa **cout**.

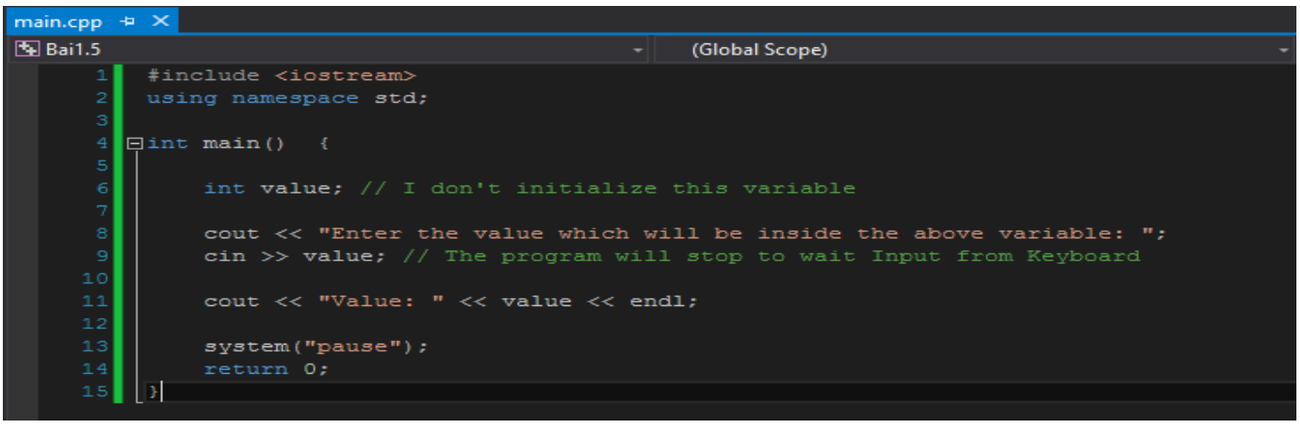
Cách hoạt động đối tượng cin

Ngược lại với đối tượng **cout** (**cout** đưa dữ liệu vào file **stdout** để xuất giá trị ra thiết bị xuất chuẩn), đối tượng **cin** lấy dữ liệu từ đối tượng file **stdin** để đẩy vào vùng nhớ của biến (variable).

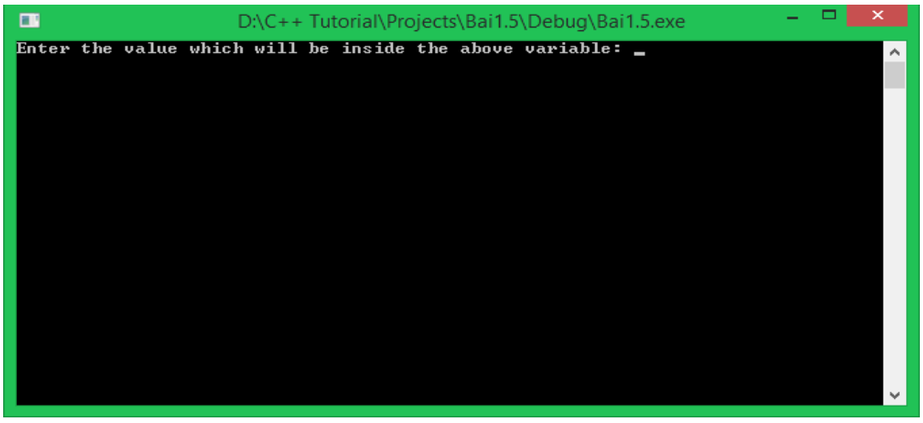


Khi biên dịch chương trình, nếu compiler bắt gặp dòng lệnh có sử dụng đối tượng **cin**, chương trình sẽ dừng lại để đợi người dùng nhập dữ liệu từ bàn phím (đến khi người dùng nhấn phím Enter), sau đó, dữ liệu vừa được nhập sẽ chuyển vào file **stdin**, đối tượng **cin** sẽ lấy giá trị đầu tiên **phù hợp với kiểu dữ liệu** để đưa vào biến thông qua toán tử **>>**.

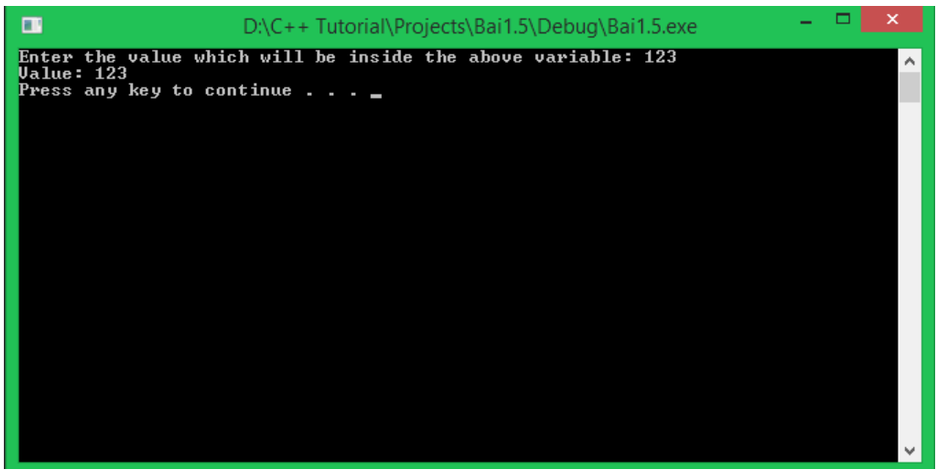
Các bạn cùng mình chạy thử đoạn chương trình bên dưới để xem kết quả thực tế:



Chúng ta chạy thử chương trình trên, đầu tiên chúng ta thấy:

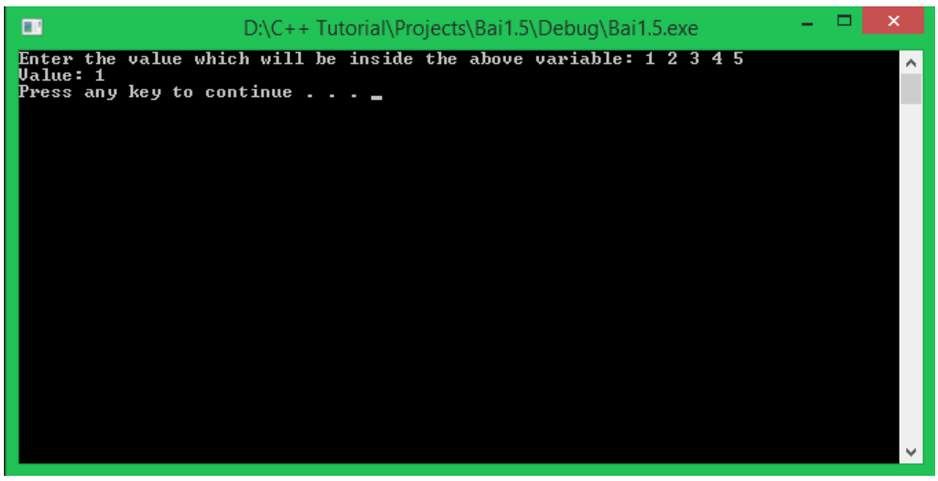


Sau khi chương trình thực thi dòng lệnh có sử dụng đối tượng **cout** đầu tiên, chương trình bắt gặp dòng lệnh có sử dụng đối tượng **cin**, ngay lúc này, chương trình dừng lại và đợi bạn nhập giá trị vào từ bàn phím.



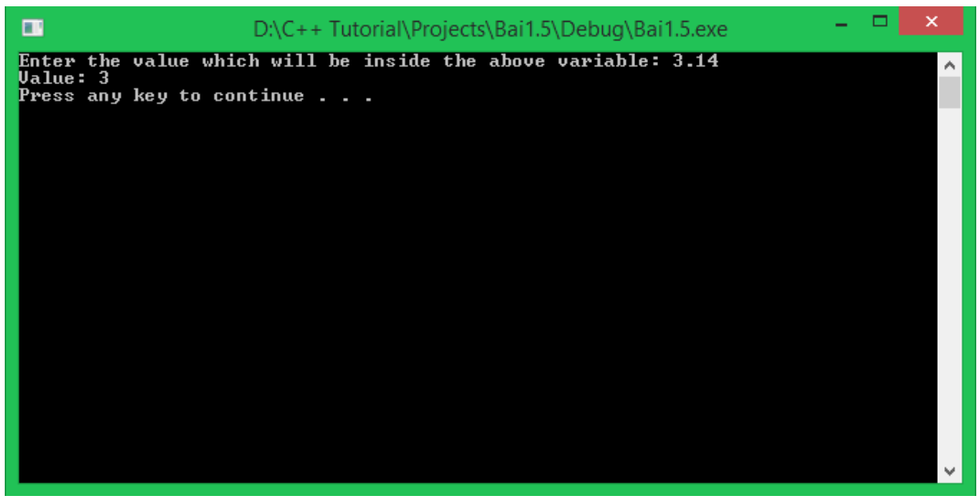
Tiếp theo mình thử nhập vào giá trị **123**, giá trị này cũng nằm trong giới hạn của kiểu số nguyên (**int**) nên hoàn toàn phù hợp với biến **value**. Ngay sau khi mình nhấn phím Enter, biến **value** nhận giá trị **123** và in ra trên màn hình.

Bây giờ, khi chúng ta muốn thay đổi giá trị khác cho biến **value**, chúng ta không cần phải gán lại giá trị mới trong mã nguồn nữa, chúng ta chỉ cần chạy lại chương trình và nhập giá trị mới từ bàn phím.



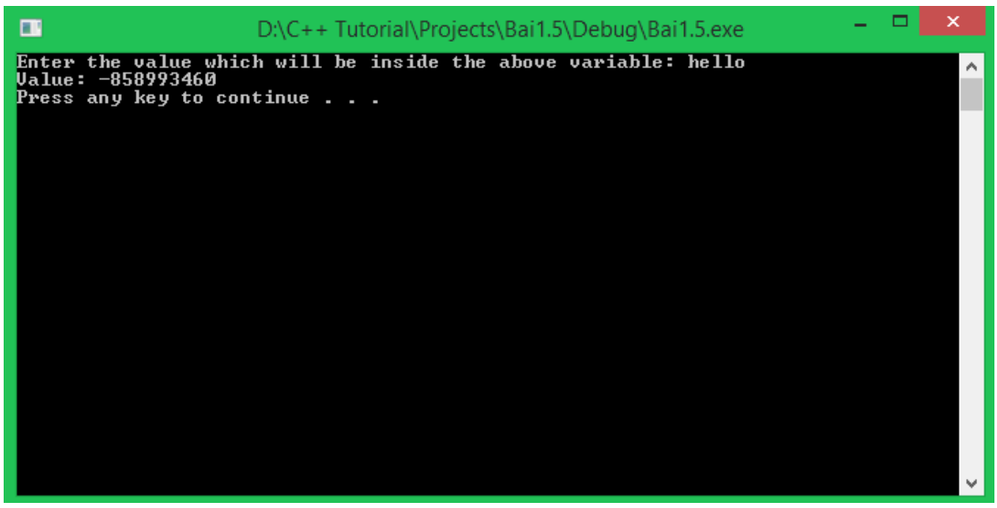
Mình vừa cố tình nhập một lúc 5 giá trị, mỗi giá trị cách nhau bởi một kí tự khoảng trắng. Và kết quả cho chúng ta thấy chỉ có giá trị đầu tiên mà chúng ta đưa vào được đẩy vào bên trong biến **value**.

Mình sẽ thử nhập một giá trị khác kiểu dữ liệu so với kiểu dữ liệu mà biến **value** được khai báo để xem kết quả:



Giá trị được nhập từ bàn phím là **3.14** là một giá trị thuộc kiểu số thực, nhưng kiểu dữ liệu chúng ta khai báo biến **value** là **int**, nên biến **value** chỉ chứa được phần nguyên của giá trị nhập vào, phần thập phân đã bị loại bỏ.

Cuối cùng, mình thử nhập giá trị không phải là kiểu số như bên dưới:

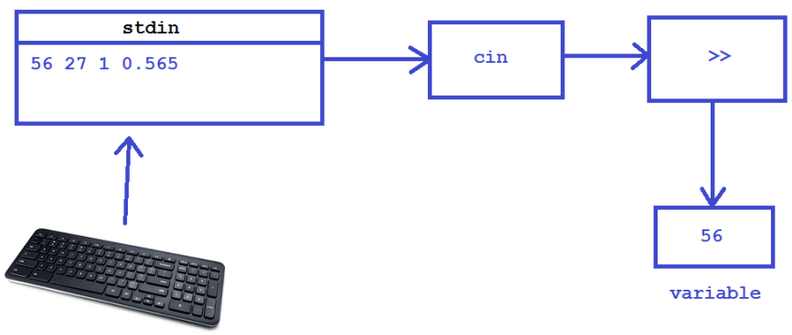


Chuỗi kí tự **hello** không phù hợp với kiểu số nguyên, nên biến **value** đã nhận giá trị sai.

Vì thế, các bạn cần nhập dữ liệu tương ứng với kiểu dữ liệu mà bạn đã khai báo cho biến.

Nhập giá trị cho nhiều biến

Chúng ta cùng xem lại hình ảnh về cách hoạt động của đối tượng **cin**:



Chúng ta có thể nhập một lúc nhiều giá trị khác nhau để đưa vào file **stdin**. Ngay khi một giá trị được đưa vào biến thông qua đối tượng **cin**, giá trị đó sẽ bị xóa ra khỏi file **stdin** lần lượt từ trái sang phải.

Chúng ta có thể tận dụng đặc điểm này để nhập dữ liệu cùng một lúc cho nhiều biến mà không cần viết nhiều dòng lệnh sử dụng đối tượng **cin**.

main.cpp

Bai1.5 (Global Scope)

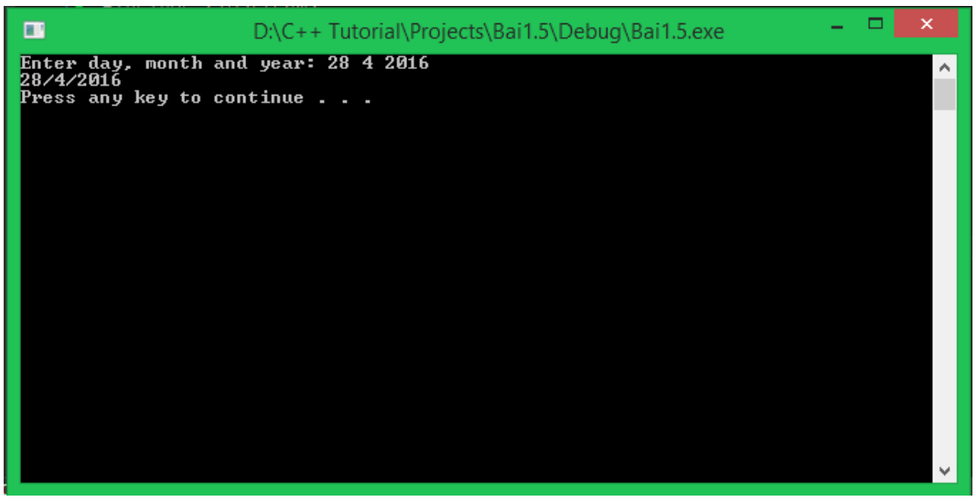
```cpp
#include <iostream>
using namespace std;

int main()  {

    int day, month, year;

    cout << "Enter day, month and year: ";
    cin >> day >> month >> year;

    cout << day << "/" << month << "/" << year << endl;

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

Trong chương trình trên, mình khai báo 3 biến có cùng kiểu dữ liệu số nguyên là **day**, **month** và **year** để lưu trữ ngày, tháng, năm hiện tại. Và mình chỉ sử dụng 1 dòng lệnh để nhập giá trị cho cả 3 biến trên:

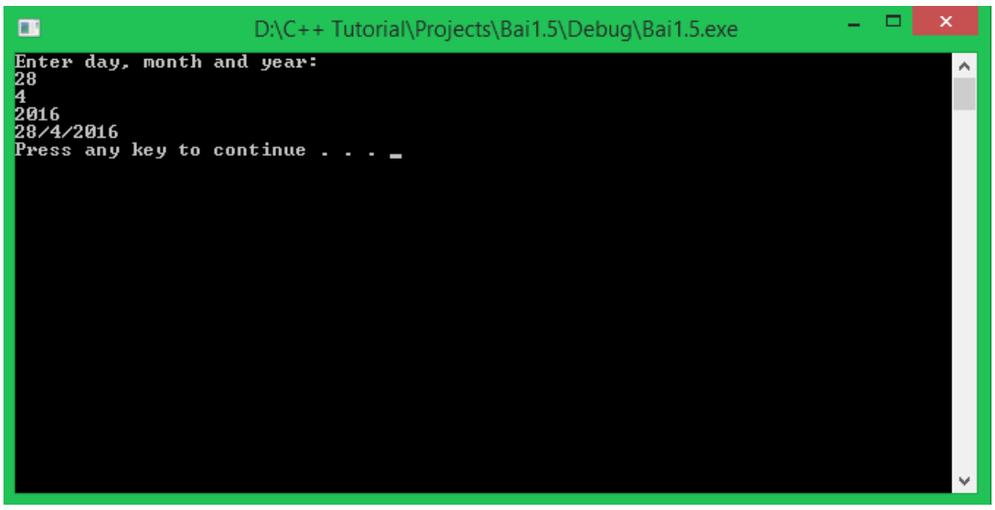
```cpp
cin >> day >> month >> year;
```

Đối tượng **cin** trên sẽ truy xuất đến file **stdin** và lấy giá trị đã được nhập vào từ trái sang phải để đưa vào theo tứ tự **day** đến **month** và cuối cùng là **year**.

Chúng ta cùng chạy chương trình để xem kết quả:



Các bạn hoàn toàn có thể nhập 3 giá trị trên 3 dòng khác nhau, lệnh **cin** vẫn khiến chương trình dừng cho đến khi nhận đủ 3 giá trị cho 3 biến day, month và year.

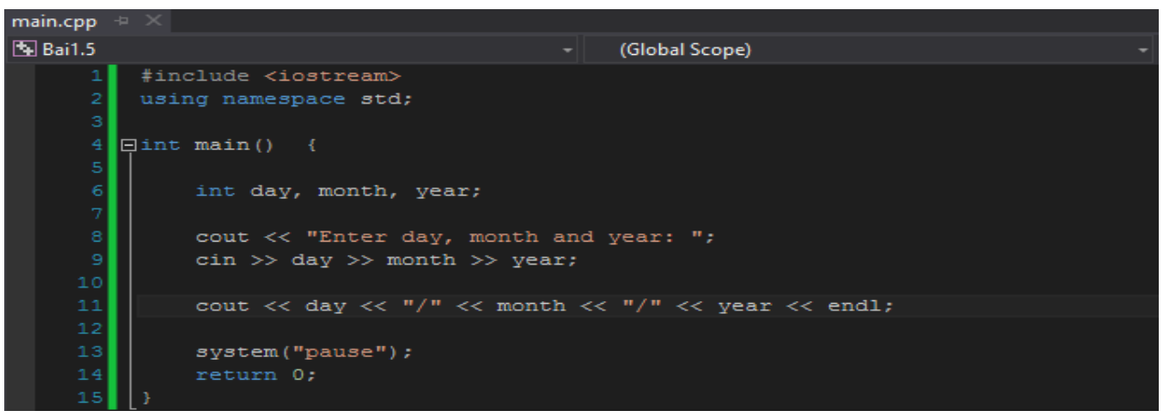


Standard Output

Các bạn đã được học và sử dụng **stardard output** của C++ trong bài **Sử dụng các lệnh liên quan đến xuất dữ liệu**. Trong C++, chúng ta sử dụng đối tượng **cout** được định nghĩa trong thư viện **iostream** thuộc **namespace std** để đưa dữ liệu ra thiết bị đầu ra (mặc định là màn hình).

Ngoài các cách xuất dữ liệu mà các bạn đã được học, chúng ta còn có thể đưa vào đối tượng **cout** một biến, và giá trị mà biến đó đang chứa sẽ được đối tượng **cout** đưa ra màn hình.

Cùng nhìn lại chương trình nhập vào ngày, tháng, năm từ bàn phím và in ngày, tháng, năm vừa nhập ra màn hình:



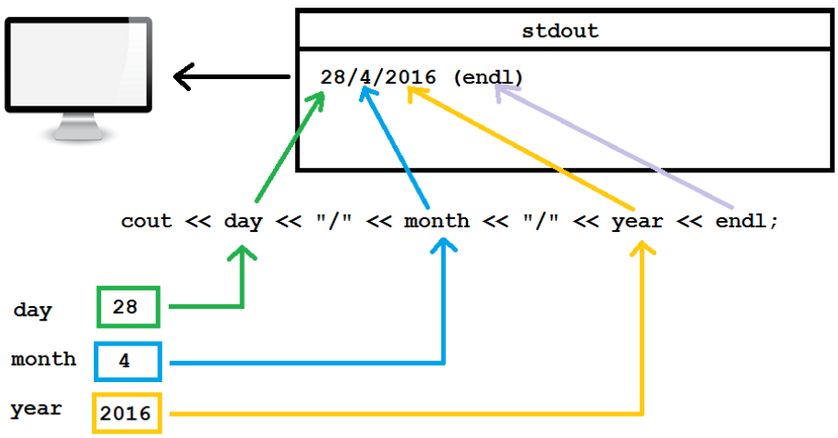
Sau dòng lệnh

```
cin >> day >> month >> year;
```

chúng ta có dòng lệnh

```
cout << day << "/" << month << "/" << year << endl;
```

Các bạn cùng nhìn vào hình bên dưới để xem cách mà đối tượng **cout** trong dòng lệnh trên hoạt động:



Đối tượng **cout** sẽ tìm đến ô nhớ mà tên biến đang nắm giữ, lấy giá trị bên trong biến đó ra và đẩy giá trị đó vào file **stdout**.

Đối tượng **cout** có thể nhận giá trị thuộc mọi kiểu dữ liệu được định nghĩa sẵn trong ngôn ngữ C++. Chúng ta không những sử dụng những dữ liệu được định nghĩa sẵn trong C++ mà còn tự định nghĩa những kiểu dữ liệu mới cho riêng mình. Các bạn sẽ được học phần này trong các bài học kế tiếp trong khóa học này.

Tổng kết

Trong bài học này, các bạn đã được học về:

- Cách đưa giá trị vào một biến trong khi chương trình đang chạy bằng **Stardard Input** trong C++.
- Cú pháp và một số cách hoạt động của đối tượng **cin**.
- Ôn lại một chút về **Stardard output** trong C++.

Bài tập cơ bản

1. Viết chương trình nhập vào điểm trung bình của ba môn học Toán, Lý và Hóa của bạn. In ra màn hình trung bình cộng điểm của ba môn học trên.

2. Các bạn chắc đã biết về phương trình bậc nhất 1 ẩn số: ax + b = 0;

Các bạn hãy viết chương trình cho phép nhập từ bàn phím 2 giá trị a và b, tính nghiệm x của phương trình bậc nhất 1 ẩn số. Thử dự đoán vấn đề gặp phải với chương trình mà bạn vừa viết.

1.6 Hằng số

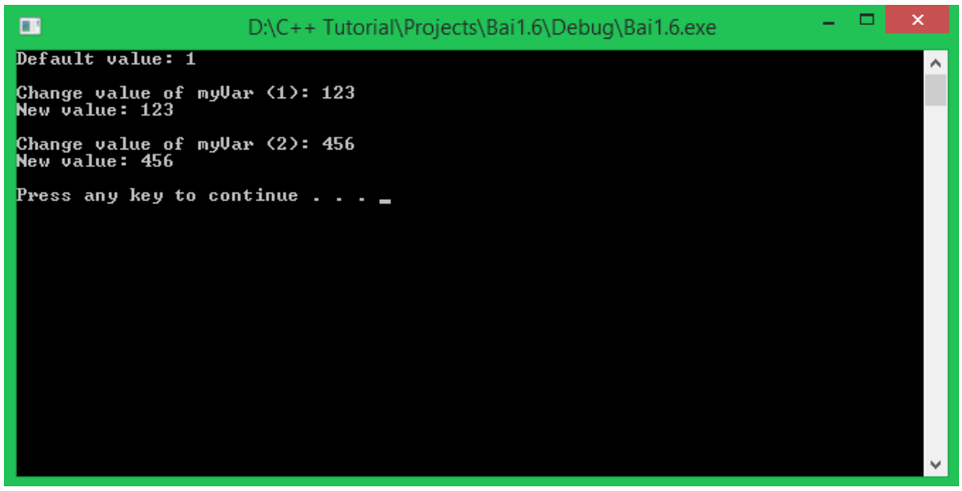
Trong bài học trước, chúng ta đã biết cách sử dụng **Starndard Input** trong C++ để nhập giá trị từ bàn phím và đưa vào vùng nhớ mà tên biến đang quản lý. Mỗi lần sử dụng đối tượng **cin** để nhập dữ liệu vào biến, giá trị trong vùng nhớ của biến đó sẽ bị thay đổi 1 lần. Đối với một số biến có cách khai báo thông thường, **compiler** cho phép ta thực hiện thay đổi giá trị của biến không giới hạn số lần.

main.cpp

Bai1.6 (Global Scope) main()

```cpp
#include <iostream>
using namespace std;

int main()  {

    int myVar = 1;
    cout << "Default value: " << myVar << endl;

    cout << "Change value of myVar (1): ";
    cin >> myVar;
    cout << "New value: " << myVar << endl;

    cout << "Change value of myVar (2): ";
    cin >> myVar;
    cout << "New value: " << myVar << endl;

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

Trong đoạn chương trình trên, biến **myVar** được khởi tạo giá trị ban đầu là 1. Và mình đã sử dụng 2 lần đối tượng **cin** để nhập giá trị mới cho biến **myVar**



Trong một số bài toán, giá trị của biến cần được thay đổi nhiều lần. Bên cạnh đó, có một số giá trị chúng ta muốn khởi tạo một lần và giữ nguyên giá trị đó trong suốt thời gian chương trình hoạt động. Ví dụ:

```
PI = 3.14;
gravity_on_earth = 9.8;
```

Những giá trị này được gọi là hằng số.

Việc định nghĩa một biến trong C++ như một hằng số sẽ giúp bạn đảm bảo giá trị của biến đó không bị thay đổi ngoài ý muốn.

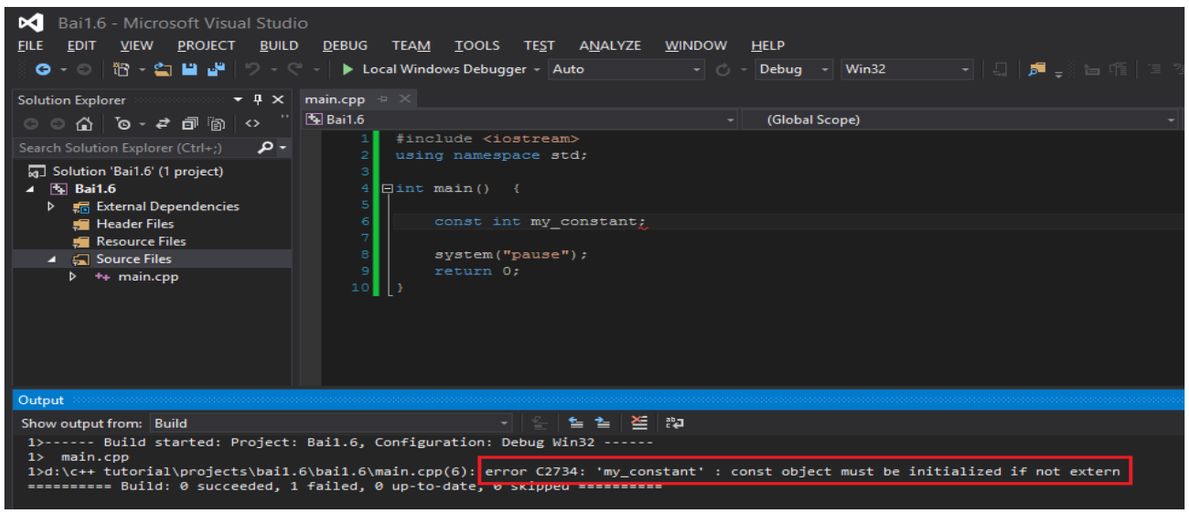
Khai báo hằng số

Để khiến một biến trong C++ trở thành một hằng số, bạn chỉ cần đặt *từ khóa* **const** trước hoặc sau kiểu dữ liệu của biến. Ví dụ:

```
const float gravity = 9.8;
int const my_constant = 123;
```

Lưu ý: Bạn phải khởi tạo giá trị cho biến hằng số mỗi khi định nghĩa chúng.

Định nghĩa một hằng số không có giá trị khởi tạo sẽ phát sinh lỗi khi biên dịch chương trình.



Các bạn có thể dùng giá trị của một biến không phải là hằng số để khởi tạo giá trị cho một biến hằng số.

```
int non_const_variable = 10;
const int const_variable = non_const_variable;
```

Một khi từ khóa **const** đã được sử dụng cho một biến, mọi hành vi khiến giá trị biến đó bị thay đổi đều bị **compiler** báo lỗi. Ngoài ra, bạn có thể sử dụng biến hằng số để tính toán, in giá trị của biến hằng số ra màn hình, ... sử dụng như một biến thông thường.

main.cpp

Bai1.6 (Global Scope) main()

```cpp
#include <iostream>
using namespace std;

int main()  {

    const int year_of_birth = 1992;

    year_of_birth = 1995;
    cin >> year_of_birth;

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

Một số cách để khởi tạo giá trị cho biến

Như mình đã nói ở trên:

> Một biến hằng số phải được khởi tạo giá trị sau khi định nghĩa.

Việc khởi tạo giá trị có thể được viết bằng nhiều cách khác nhau. Ví dụ mình có biến **year_of_birth** có kiểu **int**, mình có thể khởi tạo biến này như sau:

```
int year_of_birth = 1992;
int year_of_birth(1992);
int year_of_birth { 1992 };
```

Nhưng theo ý kiến cá nhân của mình, sử dụng toán tử bằng "=" để khởi tạo giá trị khiến chương trình dễ hiểu hơn.

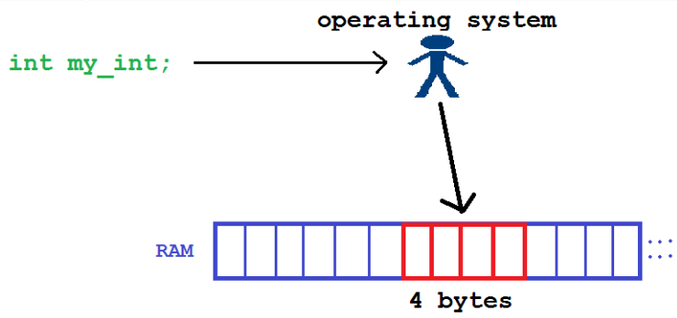
Tổng kết

Trong bài học này, các bạn đã được biết thêm khái niệm **hằng số**, cách khai báo, định nghĩa và sử dụng hằng số trong ngôn ngữ C++. Ngoài ra, các bạn còn biết thêm một số cách khởi tạo giá trị thông dụng cho biế

1.7 Phạm vi của biến

Trong các bài học trước, chúng ta đã cùng nhau tìm hiểu cách sử dụng biến (**variable**) gồm có cách khai báo, khởi tạo, nhập giá trị từ bàn phím và đưa vào biến, tính toán giá trị của biến và đưa giá trị của biến lên màn hình...

> Khi một biến được khai báo, hệ điều hành sẽ cấp phát cho chương trình một vùng nhớ có độ lớn tương ứng với độ lớn kiểu dữ liệu của biến.



Vấn đề là không phải chỉ có một mình chương trình mà các bạn đang viết sử dụng các vùng nhớ trên RAM, mà còn nhiều chương trình khác đang chạy ngầm nữa.

Trong khi đó, bộ nhớ RAM của chúng ta chỉ có giới hạn. Vì thế, một khi biến (**variable**) không còn giá trị sử dụng nữa, chúng phải được tiêu hủy để trả lại vùng nhớ mà nó đang giữ, để cấp phát cho những ứng dụng khác cần sử dụng bộ nhớ.

Khi bạn kiểm soát được việc lúc nào cần khai báo biến, khi nào cần tiêu hủy biến sẽ giúp bạn quản lý tài nguyên máy tính tốt hơn. Điều này cần kĩ năng tổ chức và thiết kế chương trình, một kĩ năng quan trọng cần có thời gian để rèn luyện.

Trong bài học này, chúng ta sẽ tìm hiểu hai khái niệm luôn luôn gắn liền với biến (**variable**):

- Phạm vi của biến.
- Thời gian tồn tại của biến.

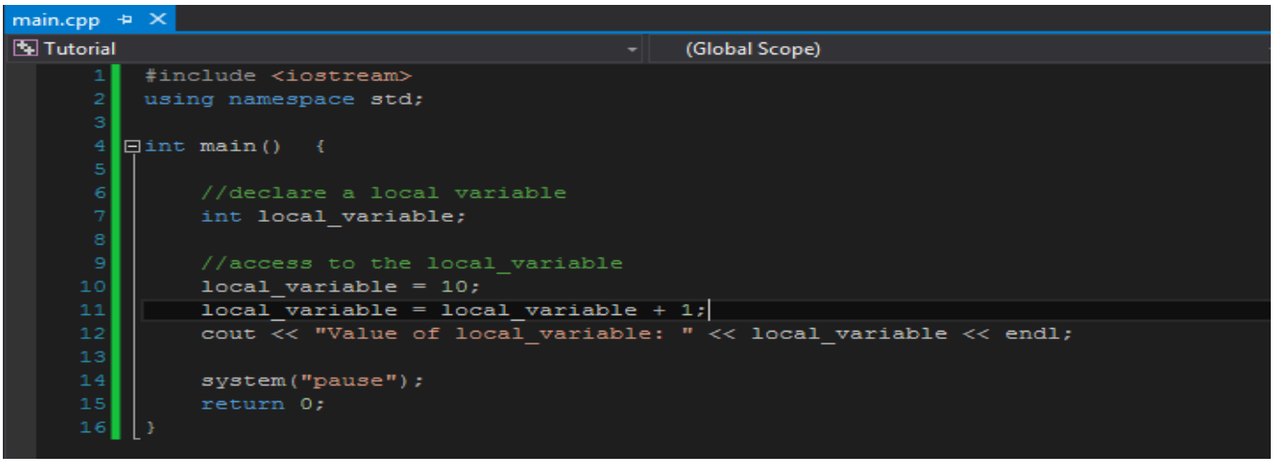
Hai khái niệm này thường có liên kết chặt chẽ với nhau.

Phạm vi của biến

Phạm vi của biến xác định nơi chúng ta có thể truy cập vào biến.

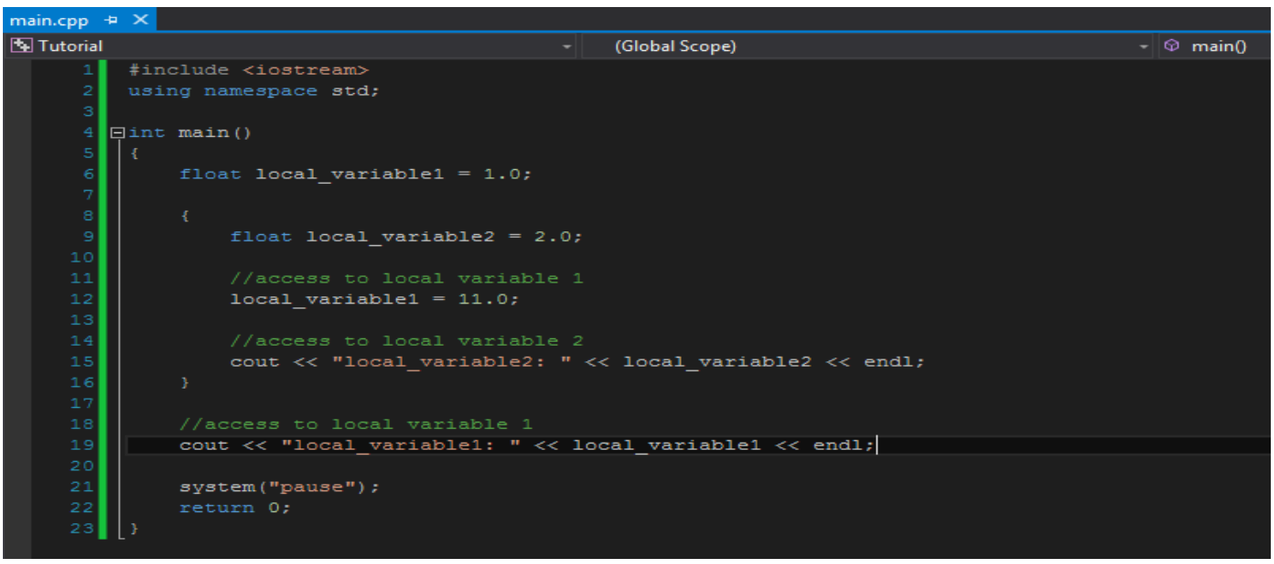
- Biến được khai báo bên trong **khối lệnh** (block) được gọi là **biến cục bộ** (local variable).

Chương trình bên dưới minh họa cho việc khai báo biến cục bộ, truy cập và truy xuất giá trị của biến cục bộ.



Biến **local variable** được khai báo bên trong khối lệnh của hàm **main**, nên các câu lệnh truy xuất đến biến **local variable** hoàn toàn hợp lệ.

Một khối lệnh có thể chứa nhiều khối lệnh con khác nhau. Ví dụ:



trong đoạn chương trình trên, chúng ta có thêm một khối lệnh nằm bên trong khối lệnh của hàm **main**, và xuất hiện một biến có tên **local variable 2** được khai báo bên trong nó. Ở trong khối lệnh con này (khối lệnh nằm trong khối lệnh của hàm **main**) chúng ta có thể truy xuất giá trị của biến **local variable 2** như mình đã làm thông qua dòng lệnh

```
cout << "local_variable2: " << local_variable2 << endl;
```

để in giá trị của biến **local variable 2** lên màn hình. Ngoài ra, mình còn sử dụng phép gán (với toán tử "**=**") để sửa đổi giá trị cho biến **local variable 1** vốn được định nghĩa bên ngoài khối lệnh con.

Điều này có nghĩa là chúng ta có thể truy cập đến một biến đã được khai báo trong những khối lệnh con bên dưới biến đó nếu những khối lệnh con này cũng được đặt trong khối lệnh chứa biến được khai báo.

Khi các bạn sử dụng các khối lệnh con, bạn có thể đặt tên biến trùng với biến được khai báo trong khối lệnh bên ngoài mà nó chứa khối lệnh con đó. Các bạn nhìn vào chương trình bên dưới để thấy rõ hơn:

```cpp
#include <iostream>
using namespace std;

int main()
{
    //we have a local variable, it's located inside main's block
    int number_of_employees = 30;

    {
        //we define another local variable with the same name
        int number_of_employees = 10;

        cout << "Number of employees of the inner block: " << number_of_employees << endl;
    }

    cout << "Number of employees of the outer block: " << number_of_employees << endl;

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

Chương trình trên không hề vi phạm quy tắc đặt tên biến mà mình đã nói ở những bài trước.

Trong cùng một khối lệnh không được phép có hai biến trùng tên.

Trong chương trình trên, hai biến **number of employees** hoàn toàn được khai báo trong hai khối lệnh khác nhau. Bây giờ chúng ta chạy thử chương trình xem kết quả in ra trên màn hình như thế nào.

```
D:\C++ Tutorial\Projects\Tutorial\Debug\Tutorial.exe
Number of employees of the inner block: 10
Number of employees of the outer block: 30
Press any key to continue . . .
```

Như các bạn cũng đã thấy, khi mình thực hiện truy xuất giá trị của biến **number of employees** bên trong khối lệnh con thì chỉ lấy được giá trị của biến được khai báo bên trong khối lệnh con đó. Tương tự, khi mình thực hiện truy xuất giá trị của biến **number of employees** của khối lệnh sau hàm **main** thì chỉ lấy được giá trị của biến được khai báo trong khối lệnh sau hàm **main**.

Việc đặt tên biến trùng nhau trong nhiều khối lệnh lồng nhau được compiler của Visual studio cho phép, nhưng mình khuyên các bạn nên nghĩ ra một tên biến khác phù hợp hơn để tránh việc nhầm lẫn khi thiết kế một chương trình có quy mô lớn.

- Biến được khai báo bên ngoài **khối lệnh** được gọi là **biến toàn cục** (global variable).

Các bạn cùng nhìn vào đoạn chương trình mẫu bên dưới để xem cách mình khai báo một biến toàn cục như thế nào.

```cpp
#include <iostream>
using namespace std;

int global_variable;

int main()
{
    cout << "Enter a value to global variable: ";
    cin >> global_variable;

    cout << "Value of global variable: " << global_variable << endl;

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

Như các bạn thấy, mình không đặt dòng khai báo biến bên trong khối lệnh của hàm **main** nữa mà mình đặt nó bên ngoài và nằm trên khối lệnh của hàm **main**.

Trong bài học đầu tiên, mình có nói về việc khối lệnh của hàm main sẽ là nơi mà chương trình bắt đầu thực thi, ngoại trừ một số câu lệnh đặc biệt có thể đặt ngoài khối lệnh (khai báo biến, include thư viện, gọi namespace, định nghĩa các class, ...).

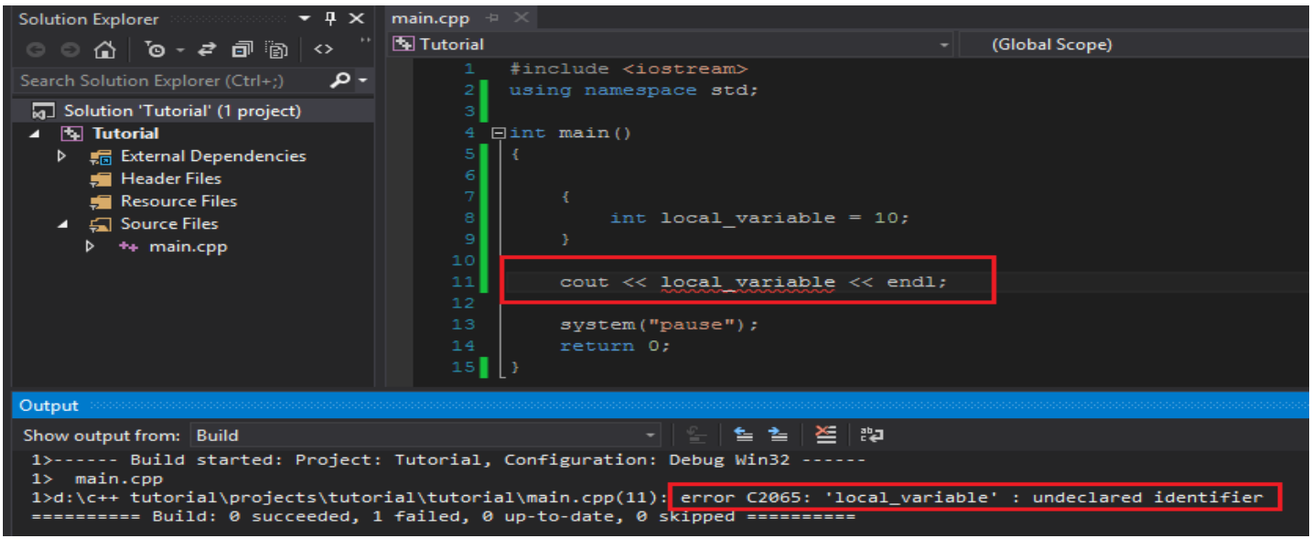
Khi có một biến khác nằm trong phạm vi của khối lệnh hàm main được khai báo cùng tên với biến toàn cục bên ngoài khối lệnh hàm main, mỗi câu lệnh truy xuất đến biến đó đều được ưu tiên tìm đến biến cục bộ bên trong hàm main trước. **Vậy có cách nào để ta truy xuất được biến toàn cục bên ngoài hàm main không?**

Câu trả lời là có! Chúng ta sử dụng toán tử phạm vi (**::**) như sau:

```
#include <iostream>
using namespace std;

int value = 1;

int main()	{

	int value = 10;

	cout << "local value: " << value << endl;
	cout << "global value: " << ::value << endl;

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

Các bạn thử chạy đoạn code trên xem chương trình thông báo kết quả như thế nào nhé.

Một khi biến toàn cục đã được khai báo, chúng có thể được truy cập tại mọi khối lệnh nằm bên dưới nó. Trong khi đó, biến cục bộ chỉ được phép truy cập khi dòng lệnh còn đặt bên trong khối lệnh chứa nó. Ví dụ:



Chương trình báo lỗi biến **local_variable** không được khai báo trước đó trong khi mình đã khai báo bên trong khối lệnh con của khối lệnh hàm **main**.

Nguyên nhân là do biến **local_variable** đã bị tiêu hủy trước khi mình kịp truy xuất đến nó.

Thời gian tồn tại của biến

Đối với những biến cục bộ (local variable) có kiểu dữ liệu thông thường như các bạn đã học trong những bài trước, vùng nhớ của biến sẽ tự động giải phóng khi ra khỏi khối lệnh chứa nó.

Việc cố gắng truy cập đến một biến đã bị hủy sẽ gây nên lỗi. Thời gian tồn tại của biến cục bộ phụ thuộc vào của khối lệnh chứa nó.

Đối với biến toàn cục (được khai báo bên ngoài khối lệnh của hàm **main**), nó sẽ tồn tại cho đến khi chương trình kết thúc hoặc bị kết thúc bởi người dùng.

Vì thế, các bạn chỉ nên sử dụng biến toàn cục khi cần thiết, để tránh việc vùng nhớ của biến toàn cục được cấp phát nhưng bị chiếm giữ quá lâu gây ảnh hưởng đến việc cấp phát bộ nhớ cho những chương trình khác.

Tổng kết

- Biến cục bộ được khai báo bên trong khối lệnh. Những biến này chỉ được phép truy cập ở bên trong khối lệnh đó. Biến cục bộ sẽ bị hủy tại thời điểm kết thúc khối lệnh.
- Biến toàn cục được khai báo bên ngoài khối lệnh. Những biến này được phép truy cập trong mọi khối lệnh nằm bên dưới nó. Biến toàn cục chỉ bị hủy khi chương trình kết thúc.

1.8 Các phép toán cơ bản

Trong bài học hôm nay, chúng ta sẽ học cách sử dụng các phép toán cơ bản như phép cộng, trừ, nhân, chia, chia lấy phần dư, căn bậc 2, lũy thừa, giá trị tuyệt đối, ... áp dụng trên các kiểu dữ liệu số cơ bản (int, float, double ...).

Ngôn ngữ C++ đã định nghĩa sẵn một số toán tử toán học cơ bản cho các phép tính thông dụng (+, -, *, /, ...), một số phép toán phức tạp hơn như căn bậc 2, lũy thừa, ... chưa có toán tử được định nghĩa, vì thế chúng ta sẽ sử dụng thêm thư viện **cmath** để tính kết quả các phép toán trên.

Các toán tử toán học đã được định nghĩa trong C++

Các toán tử toán học được chia thành hai loại: Toán tử một ngôi (**unary operators**) và toán tử hai ngôi (**binary operators**).

- Toán tử một ngôi (unary operators) là toán tử chỉ đi cùng với một toán hạng để tạo thành biểu thức có nghĩa.
- Toán tử hai ngôi (binary operators) là toán tử thường dùng kèm với hai toán hạng để tạo thành một biểu thức có nghĩa.

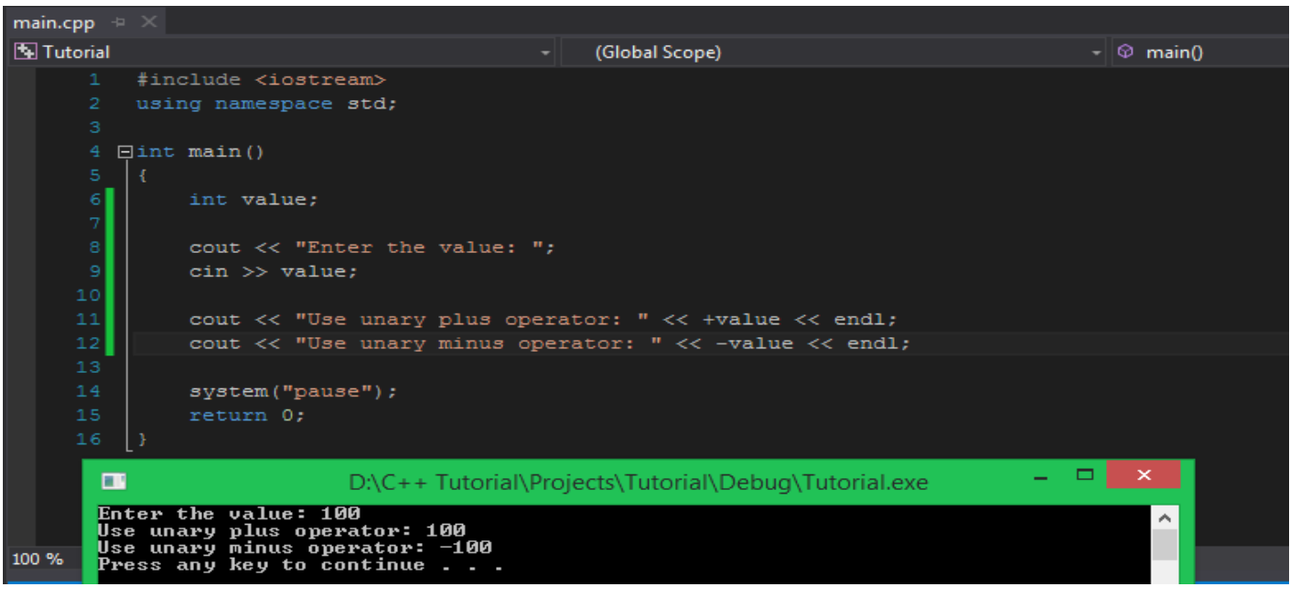
Trong ngôn ngữ lập trình C++, một toán hạng có thể là một giá trị hoặc một biến (**variable**).

Toán tử một ngôi

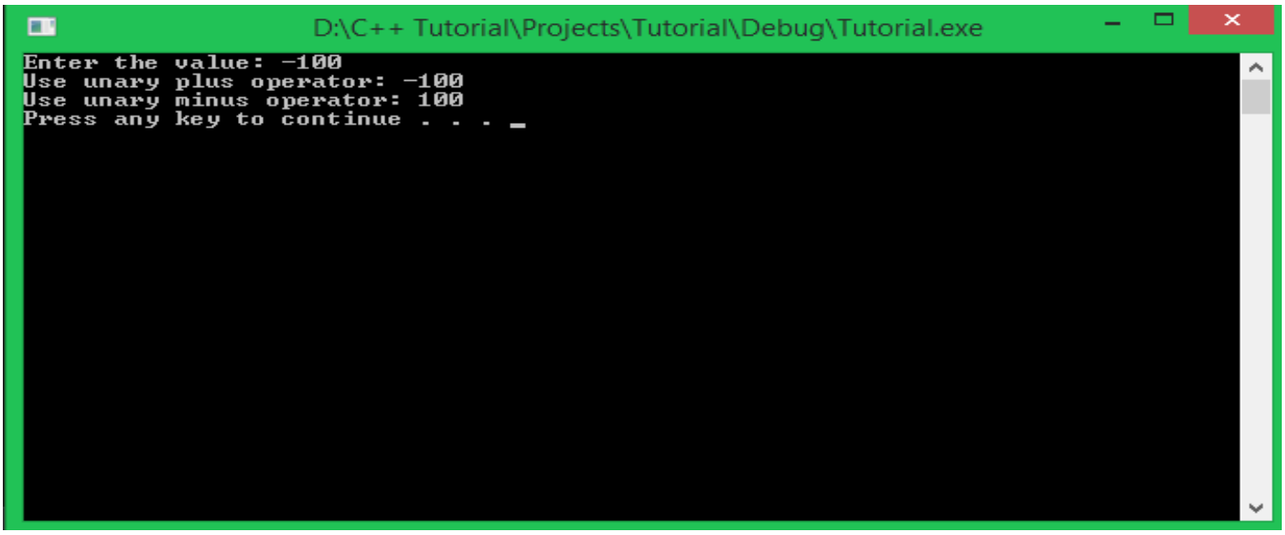
Có hai toán tử một ngôi trong C++:

Operator	Symbol	Form	Operation
Unary plus	+	+x	Value of x
Unary minus	-	-x	Negation of x

Sử dụng toán tử cộng một ngôi trước một giá trị thì kết quả trả về giá trị dương, ngược lại, ta nhận được giá trị âm. Ví dụ:



Chạy lại chương trình trên và nhập từ bàn phím vào một giá trị âm, ta được kết quả:



Giá trị ban đầu nhập vào là -100. Khi sử dụng toán tử một ngôi, ta viết lại như sau:

+(-100) = -100

-(-100) = 100

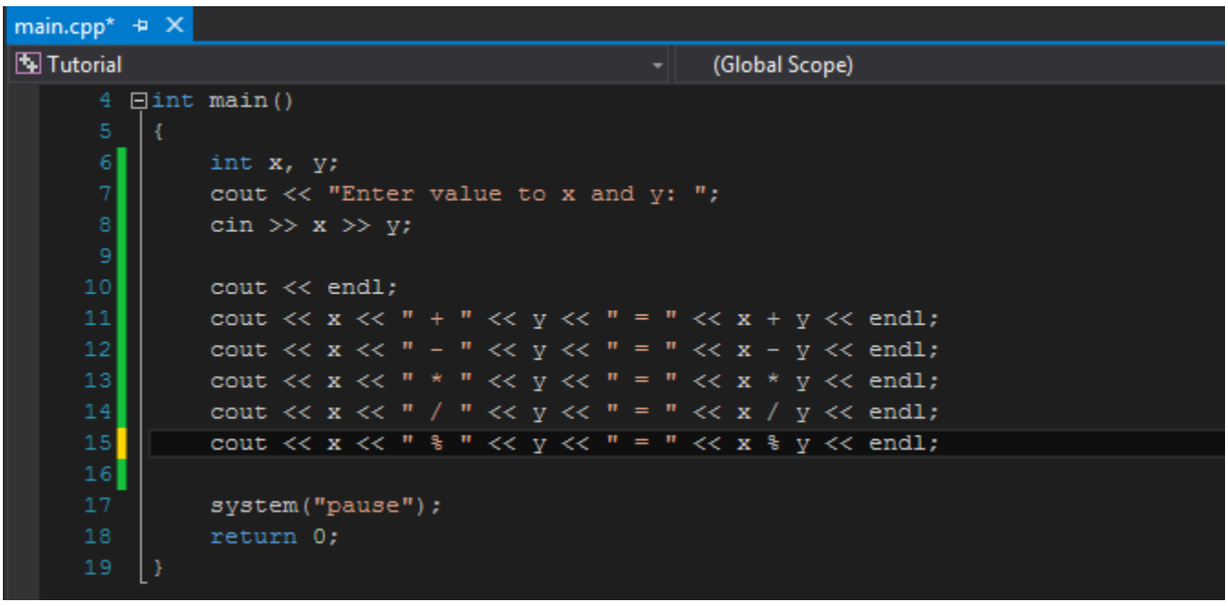
Toán tử hai ngôi

Ngôn ngữ C++ định nghĩa cho chúng ta 5 toán tử toán học hai ngôi như bảng bên dưới:

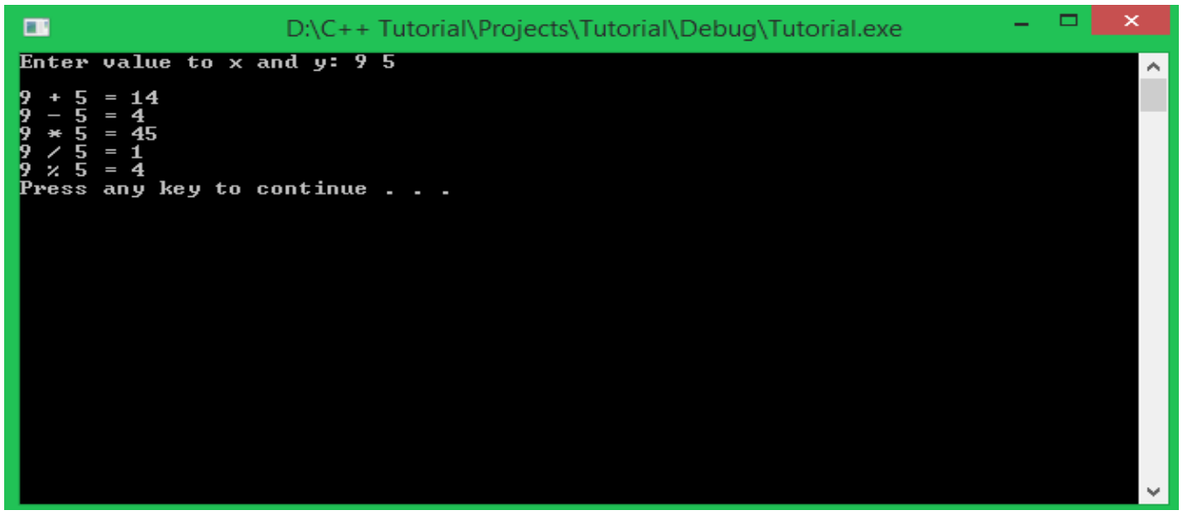
Operator	Symbol	Form	Operation
Addition	+	x + y	x plus y
Subtraction	-	x - y	x minus y
Multiplication	*	x * y	x multiplied by y
Division	/	x / y	x divided by y
Modulus	%	x % y	The remainder of x divided by y

Phép toán Modulus (%) có nghĩa là thực hiện phép chia hai số nhưng chỉ lấy phần dư. **Phép toán Modulus (%) chỉ cho phép thực hiện với hai giá trị số nguyên.**

Chúng ta cùng viết một chương trình in ra kết quả của các phép toán sử dụng toán tử hai ngôi trong C++:



Chạy chương trình trên, nhập vào giá trị cho x là 9, nhập giá trị cho y là 5 và xem kết quả.



Chương trình cho kết quả của các biểu thức như mong đợi, ngoại trừ kết quả của phép chia (*/*).

Khi thực hiện tính giá trị biểu thức **9 / 5** trong toán học, chúng ta được kết quả là **1.8**, nhưng vì kiểu dữ liệu của hai biến chúng ta sử dụng là **int** (kiểu số nguyên) nên kết quả cũng trả về một giá trị số nguyên (bị mất phần thập phân).

Để giải quyết vấn đề này chúng ta có hai cách:

- Sử dụng kiểu dữ liệu số thực (float, double, ...) cho biến.
- Ép kiểu.

Sử dụng static_cast<> để thực hiện phép chia hai số nguyên

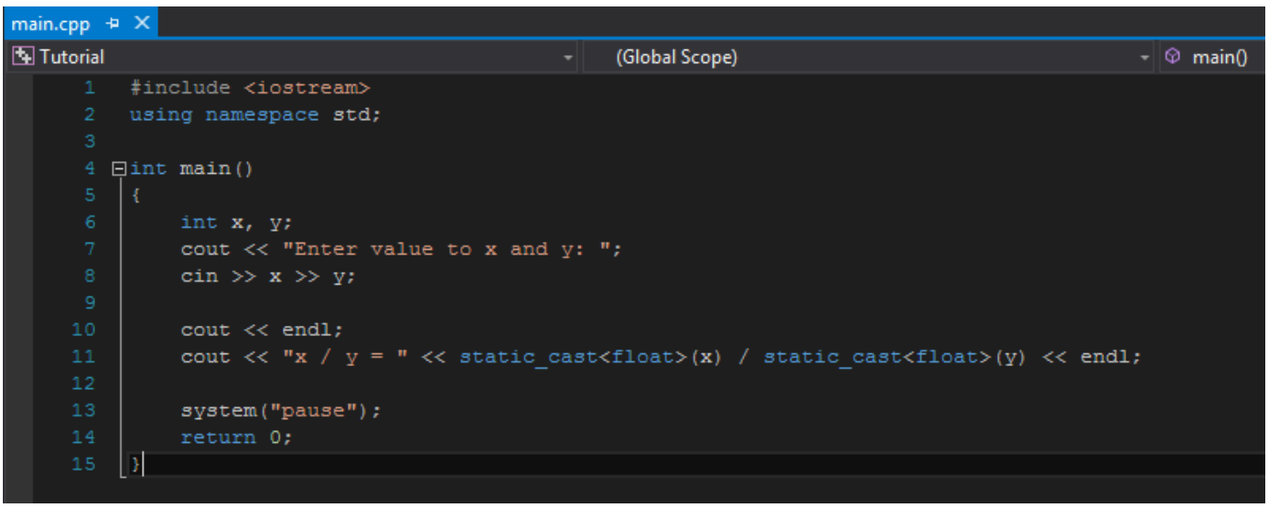
Sử dụng **static_cast<>** là một cách để ép kiểu dữ liệu trong C++. Ép kiểu sẽ tạo ra một giá trị từ một giá trị có kiểu dữ liệu khác.

Cú pháp sử dụng **static_cast<>**:

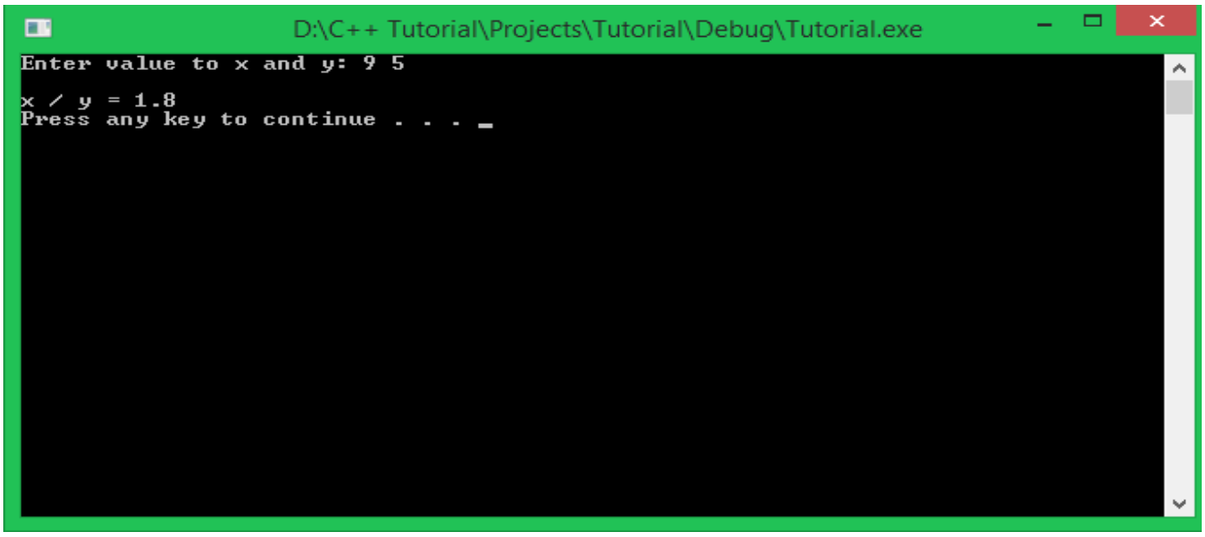
```
static_cast<new_type>(expression)
```

static_cast có thể nhận một biểu thức làm đầu vào, chuyển nó thành bất cứ kiểu dữ liệu cơ bản gì mà **new_type** mô tả.

Các bạn cùng xem ví dụ bên dưới để rõ hơn về cách sử dụng **static_cast**



Để lấy giá trị kiểu float của biến x, chúng ta viết `static_cast<float>(x)`. Trong chương trình trên, chỉ cần ép kiểu cho một biến x là đủ để thực hiện phép chia trả về số thực.
Cùng xem kết quả chương trình:



Chúng ta đã nhận được kết quả đúng.

Có một lưu ý khi thực hiện phép chia hai số nguyên có chứa giá trị âm trong C++. Trước phiên bản C++11, compiler tự ý làm tròn lên hoặc xuống. Ví dụ **-5 / 2** sẽ được kết quả là -3 hoặc -2 tùy vào cách mà compiler làm tròn số.

Toán tử gán (assignment operator)

Phép gán cũng là một trong những toán tử toán học được C++ định nghĩa. Phép gán có tác dụng đưa giá trị của một con số, một biểu thức hoặc lấy giá trị của một biến khác để đưa vào biến được gán.

Cú pháp sử dụng toán tử gán như sau:

```
<variable> = <expression>;
```

Biến được gán giá trị luôn luôn nằm bên trái toán tử "=".

Toán tử gán có thể dùng ngay khi khai báo biến để vừa khai báo vừa khởi tạo giá trị cho biến, hoặc có thể tách riêng thành một dòng lệnh.

```
int variable = 5;
variable = 10;
variable = 5 * 3 + 2;

int another_variable = 3;
variable = another_variable * 2;

variable = variable + 1; //tăng giá trị biến variable lên 1.
variable = variable - 1; //giảm giá trị biến variable đi 1.
variable = variable * 2; //nhân giá trị biến variable lên 2 lần.
variable = variable / 2; //chia giá trị biến variable đi 2 lần.
variable = variable % 3; //lấy phần dư của biến variable khi chia 3.
```

Những cách sử dụng toán tử gán như trên hoàn toàn hợp lệ.

Riêng với 5 dòng lệnh gán cuối cùng, chúng ta có một cách viết tắt khác ngắn gọn hơn.

```
variable += 1;
variable -= 1;
variable *= 2;
variable /= 2;
variable %= 3;
```

Cách dùng này có ý nghĩa hoàn toàn giống với cách viết ở trên.

Ý nghĩa của các toán tử này các bạn có thể tra ở bảng bên dưới:

Operator	Symbol	Form	Operation
Assignment	=	x = y	Assign value y to x
Addition assignment	+=	x += y	Add y to x
Subtraction assignment	-=	x -= y	Subtract y from x
Multiplication assignment	*=	x *= y	Multiply x by y
Division assignment	/=	x /= y	Divide x by y
Modulus assignment	%=	x %= y	Put the remainer of x / y in x

Sử dụng thư viện cmath

Thư viện **cmath** định nghĩa cho chúng ta một số hàm tính toán và chuyển đổi toán học cơ bản. Để sử dụng thư viện này, các bạn chỉ cần thêm dòng

```
#include <cmath>
```
tại phần khai báo thư viện trong chương trình.

Một số hàm tính lũy thừa, số mũ:

- **Pow**:

```
double pow (double base, double exponent);
float pow (float base, float exponent);
long double pow (long double base, long double exponent);
```
Các bạn chưa cần phải hiểu về cách khai báo hàm pow như trên. Về mặt ý nghĩa, giá trị thứ nhất (base) được đưa vào hàm pow là cơ số, giá trị thứ hai (exponent) là số mũ, giá trị trả về là lũy thừa cơ số base mũ exponent.

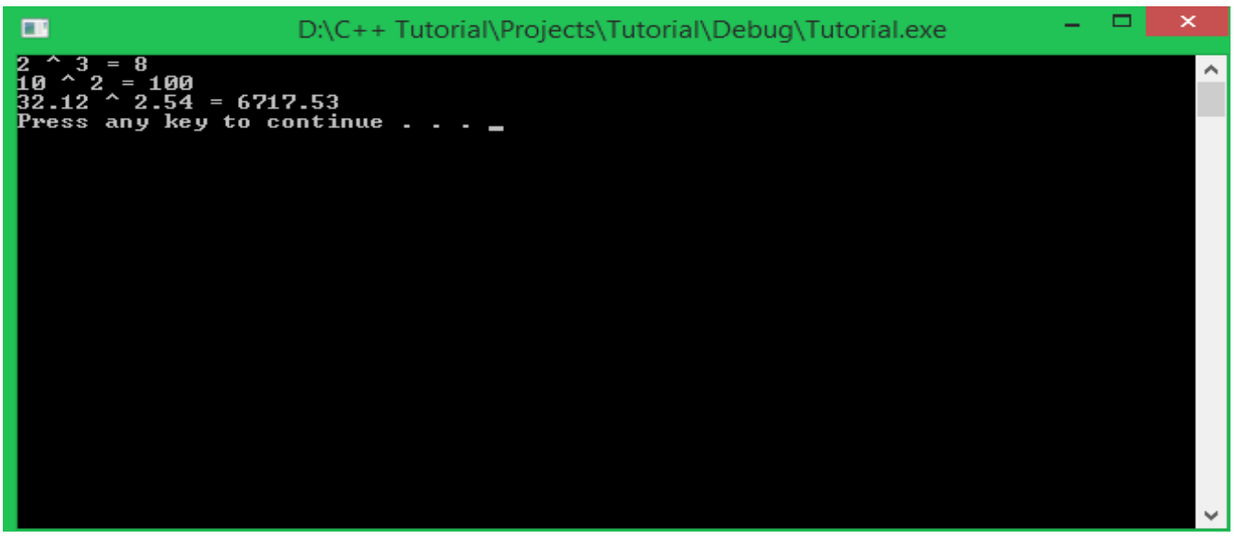
Ví dụ:

main.cpp

Tutorial (Global Scope)

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    cout << "2 ^ 3 = " <<  pow(2, 3) << endl;
    cout << "10 ^ 2 = " << pow(10, 2) << endl;
    cout << "32.12 ^ 2.54 = " << pow(32.12, 2.54) << endl;

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

Các bạn cùng viết ví dụ trên vào Visual studio và chạy chương trình để xem kết quả mà hàm pow trả về.



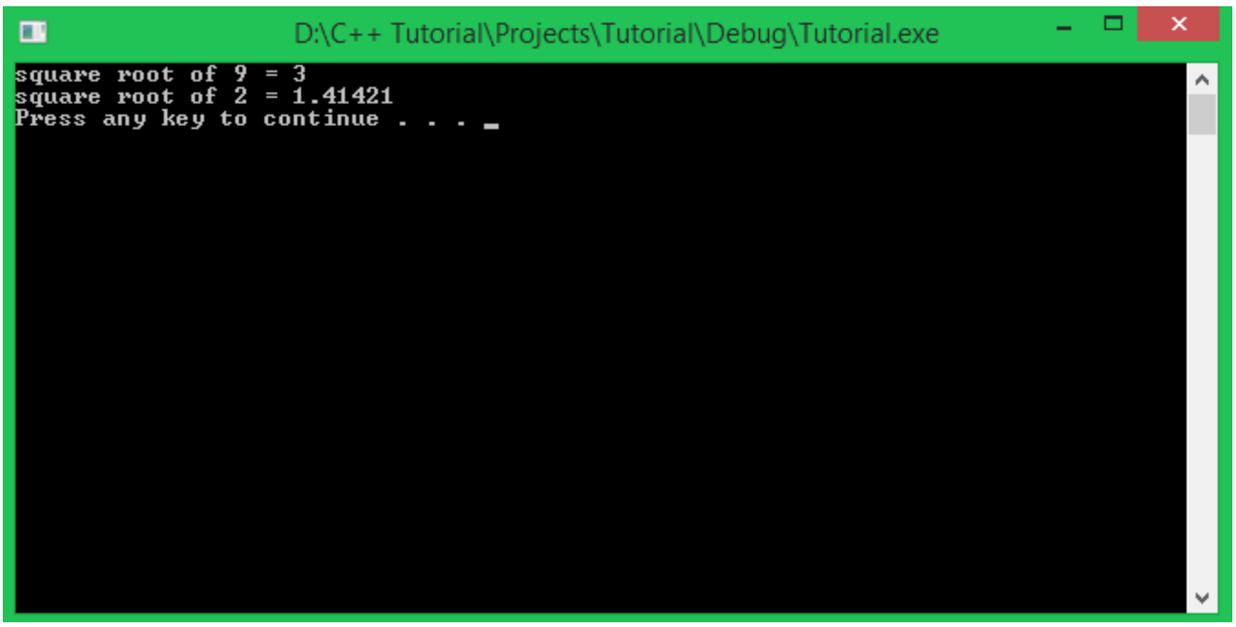
- **Sqrt**:

```
double sqrt (double x);
float sqrt (float x);
long double sqrt (long double x);
```
Phía trên là phần khai báo hàm **sqrt** trong thư viện **cmath**, hàm này nhận vào một giá trị số thực (float, double, long double) và trả về giá trị là căn bậc 2 của giá trị mà bạn đưa vào.

Sau đây là ví dụ mẫu về cách sử dụng hàm sqrt để tính căn bậc 2:

```cpp
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    cout << "square root of 9 = " << sqrt(9) << endl;
    cout << "square root of 2 = " << sqrt(2) << endl;

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

Kết quả chúng ta thu được như sau:



Một số hàm lượng giác

- **Cos**:

```
double cos (double angle);
float cos (float angle);
long double cos (long double angle);
```

Hàm **cos** nhận vào một giá trị số thực angle (đơn vị **radian**) đại diện cho góc mà bạn muốn tính đường cosine, và trả về giá trị là cosine của góc angle đó.

Ví dụ như sau:

```cpp
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    const double PI = 3.14159265;

    double angle = 60.0; //60 degree
    cout << "sine of 60 degree: " << cos(angle * PI / 180) << endl;

    angle = 0.0; //0 degree
    cout << "sine of 0 degree: " << cos(angle * PI / 180) << endl;

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

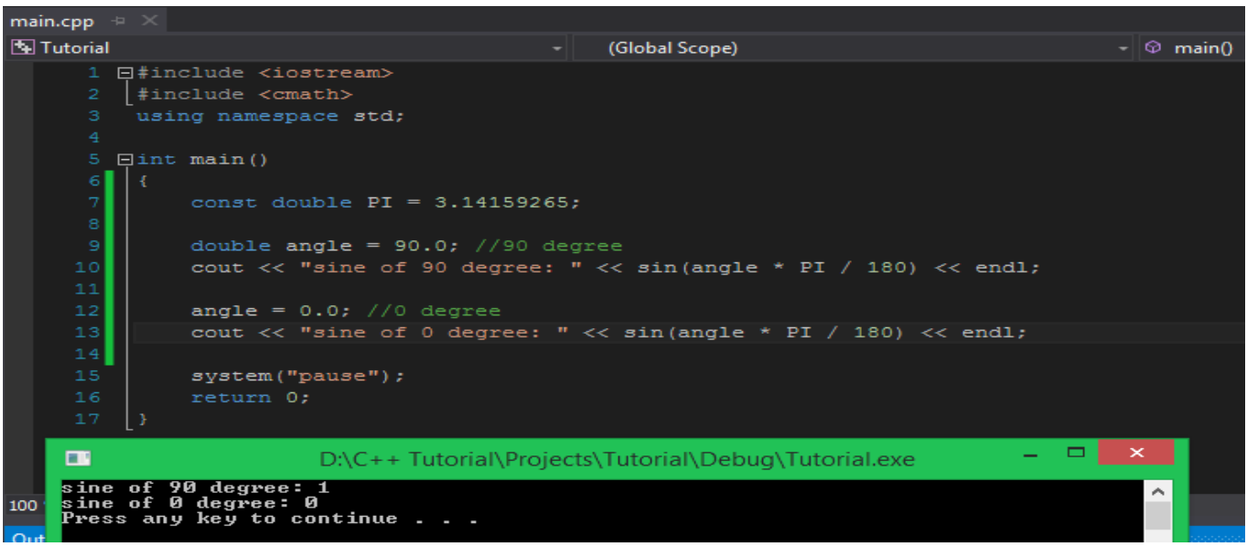
```
D:\C++ Tutorial\Projects\Tutorial\Debug\Tutorial.exe
sine of 60 degree: 0.5
sine of 0 degree: 1
Press any key to continue . . .
```

- **Sin**:

```
double sin (double x);
float sin (float x);
long double sin (long double x);
```

Hàm **sin** nhận vào một giá trị số thực angle (đơn vị **radian**) đại diện cho góc mà bạn muốn tính đường sine, và trả về giá trị trên đường sine của góc angle đó.

Ví dụ mẫu:



Ngoài ra, chúng ta còn có nhiều hàm khác như **tan**, **atan**, ... đã được định nghĩa bên trong thư viện **cmath**.

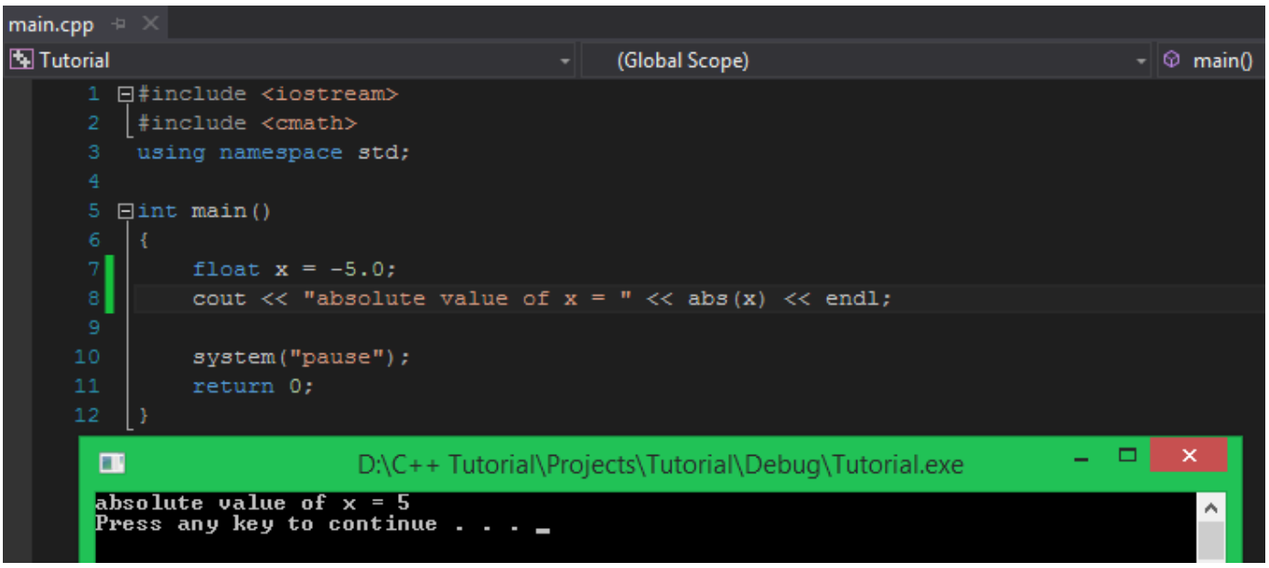
Một số hàm khác

- **Abs**:

```
double abs (double x);
float abs (float x);
long double abs (long double x);
```

Hàm **abs** sẽ nhận vào một giá trị số thực **x** (kiểu float, double hoặc long double) và trả về giá trị tuyệt đối của **x**.

Các bạn cùng thử làm theo ví dụ mẫu để làm quen với cách sử dụng hàm **abs**.



Giá trị ban đầu được khởi tạo cho biến x là -5.0, giá trị tuyệt đối được trả về thông qua hàm **abs** là 5.0.

Do số lượng các hàm toán học được định nghĩa rất nhiều, nên mình xin dẫn đường link hướng dẫn sử dụng các hàm trong thư viện **cmath** để các bạn có thể tiện tham khảo khi cần thiết.

http://www.cplusplus.com/reference/cmath/

1.9 Toán tử tăng giảm

Chào các bạn! Tiếp tục với khóa học lập trình C++ trực tuyến, trong bài học hôm nay, chúng ta tìm hiểu thêm 2 toán tử rất quan trọng thường xuyên được sử dụng trong ngôn ngữ lập trình C++.

Toán tử tăng (increment operator)

Toán tử tăng (kí hiệu: **++**) có thể đứng trước hoặc sau một biến (variable). Ví dụ:

```
int value = 5;
++value;
value++;
```

Cả hai vị trí đứng của toán tử tăng đều có chung một mục đích: **Tăng giá trị của biến lên 1 đơn vị**.

Nhưng chúng hoàn toàn khác nhau về mặt ngữ nghĩa.

Operator	Symbol	Form	Operation
Prefix increment	++	++x	Increment x, then evaluate x
Postfix increment	++	x++	Evaluate x, then increment x

Toán tử tăng khi làm tiền tố cho một biến rất dễ hiểu. Giá trị của x sẽ được tăng lên 1 đơn vị, sau đó giá trị mới sẽ được định cho biến x.

Ví dụ:

```
int x = 5;
int y = ++x; //giá trị của x bây giờ là 6, giá trị 6 sẽ được gán vào biến y
```

Toán tử tăng làm hậu tố của một biến có vẻ khó hiểu hơn một chút. Compiler sẽ tạo ra một bản sao của biến x hiện tại, tăng giá trị của biến x ban đầu lên 1 đơn vị, và sau đó định giá trị bằng bản sao của biến x. Ngay sau đó, bản sao của biến x sẽ bị loại bỏ.

Ví dụ:

```
int x = 5;
int y = x++; //giá trị của x bây giờ là 6, giá trị 5 ban đầu của x sẽ được gán cho y
```

Mình trình bày lại cách hoạt động của đoạn code trên thêm 1 lần nữa.

Đầu tiên, biến x được khai báo và khởi tạo giá trị ban đầu là 5. Tiếp theo, biến y được khai báo và gán cho giá trị là x++. Khi gặp toán tử tăng (++) làm hậu tố của biến x, compiler tạo ra một bản sao của biến x mang theo giá trị 5 ban đầu. Bây giờ, biến x gốc được tăng giá trị lên 1 đơn vị (x sẽ bằng 6), nhưng giá trị được gán cho biến y không phải là biến x ban đầu mà là bản sao của biến x (bản sao mang giá trị 5). Sau khi gán giá trị xong, bản sao của biến x bị xóa bỏ.

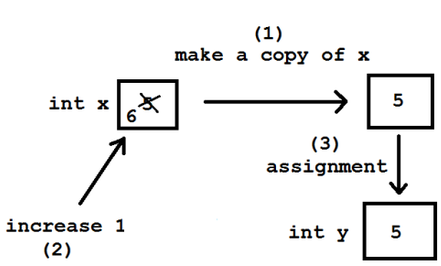
Operator	Symbol	Form	Operation
Prefix decrement	--	--x	Decrement x, then evaluate x
Postfix decrement	--	x--	Evaluate x, then decrement x

Toán tử giảm

Toán tử giảm (kí hiệu: **--**) có thể làm tiền tố (đứng trước) hoặc hậu tố (đứng sau) một biến (variable). Ví dụ:

```
int value = 10;
--value;
value--;
```

Mục đích khi sử dụng toán tử này là để **giảm giá trị của biến đi 1 đơn vị**, nhưng có sự khác nhau giữa cách sử dụng tiền tố và hậu tố (tương tự toán tử tăng).



Cũng tương tự như mình đã trình bày cho toán tử tăng (++), toán tử giảm (--) khi làm tiền tố cho biến cũng khá đơn giản. Giá trị biến x sẽ bị trừ đi 1 đơn vị, mọi thao tác với biến x sẽ có hiệu lực trên giá trị mới ngay lập tức.

Ví dụ:

```
int x = 5;
int y = --x; //Giá trị của x bây giờ là 4, giá trị 4 được gán cho biến y
```

Đối với toán tử giảm (--) làm hậu tố cho biến, compiler cũng tạo ra một bản sao của biến x, giảm giá trị biến x đi 1 đơn vị, gán giá trị của bản sao biến x vào biến y, sau đó loại bỏ bản sao của biến x ra khỏi chương trình.

```
int x = 5;
int y = 5--; //Giá trị của biến x bây giờ là 4, giá trị 5 được gán cho biến y
```

Tổng kết

Toán tử tăng, giảm là những toán tử được sử dụng khá thường xuyên trong thực tế. Các bạn cần nắm rõ bài học hôm nay trước khi đi tiếp những bài học tiếp theo.

1.10 Độ ưu tiên của các toán tử

Thế nào là độ ưu tiên của toán tử?

Để đánh giá đúng một biểu thức chứa nhiều toán tử, ví dụ `5 + 2 * 4 / 2`, chúng ta phải biết mỗi toán tử trong biểu thức đó thực hiện công việc gì, và thứ tự mà chúng thực hiện. Thứ tự thực hiện các phép tính của một biểu thức kết hợp nhiều toán tử gọi là **độ ưu tiên của toán tử** (operator precedence).
Áp dụng độ ưu tiên của các toán tử toán học vào biểu thức `5 + 2 * 4 / 2`, ta có thể đánh giá lại biểu thức này dưới dạng `5 + ((2 * 4) / 2)` và kết quả là **9**.
Khi có 2 toán tử có cùng độ ưu tiên được đặt cạnh nhau trong 1 biểu thức, chúng ta sử dụng nguyên tắc kết hợp (**associativity rules**) để biết được toán tử nào sẽ được thực hiện trước.

Ví dụ các toán tử toán học có được đánh giá từ trái qua phải, nên khi gặp biểu thức `3 * 2 * 6`, chúng ta hiểu được rằng biểu thức sẽ được tính là `(3 * 2) * 6`.
Để biết được ngôn ngữ C++ định nghĩa độ ưu tiên các toán tử (operators precedence) như thế nào, chúng ta không có cách nào khác ngoài việc tra trong bảng độ ưu tiên toán tử bên dưới.

Trong bảng này, các toán tử được chia thành nhiều nhóm khác nhau, độ ưu tiên được sắp xếp giảm dần từ trên xuống dưới.

C++ Operator Precedence and Associativity

- **Group 1 (no associativity)**:

Operator Description	Operator
Scope resolution	::

- **Group 2 (left to right associativity)**:

Operator Description	Operator
Member selection (object or pointer)	. or ->
Array subscript	[ ]
Function call	( )
Postfix increment	++
Postfix decrement	--
Type name	typeid( )
Constant type conversion	const_cast
Dynamic type conversion	dynamic_cast
Reinterpreted type conversion	reinterpret_cast
Static type conversion	static_cast

- **Group 3 (right to left associativity)**:

Operator Description	Operator
Size of object or type	sizeof
Prefix increment	++
Prefix decrement	--
One's complement	~
Logical not	!
Unary negation	-
Unary plus	+
Address-of	&
Indirection	*
Create object	new
Destroy object	delete
Cast	Cast: ()

- **Group 4 (left to right associativity)**:

Operator Description	Operator
Pointer-to-member (objects or pointers)	.* or ->*

- **Group 5 (left to right associativity)**:

Operator Description	Operator
Multiplication	*
Division	/
Modulus	%

- **Group 6 (left to right associativity)**:

Operator Description	Operator
Addition	+
Subtraction	-

- **Group 7 (left to right associativity)**:

Operator Description	Operator
Left shift	<<
Right shift	>>

- **Group 8 (left to right associativity)**:

Operator Description	Operator
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=

- **Group 9 (left to right associativity)**:

Operator Description	Operator
Equality	==
Inequality	!=

- **Group 10 (left to right associativity)**:

Operator Description	Operator
Bitwise AND	&

- **Group 11 (left to right associativity)**:

Operator Description	Operator
Bitwise exclusive OR	^

- **Group 12 (left to right associativity)**:

Operator Description	Operator
Bitwise inclusive OR	|

- **Group 13 (left to right associativity)**:

Operator Description	Operator
Logical AND	&&

- **Group 14 (left to right associativity)**:

Operator Description	Operator
Logical OR	||

- **Group 15 (right to left associativity)**:

Operator Description	Operator
Conditional	? :

Group 16 (right to left associativity):

Operator Description	Operator
Assignment	=
Multiplication assignment	*=
Division assignment	/=
Modulus assignment	%=
Addition assignment	+=
Subtraction assignment	-=
Left-shift assignment	<<=
Right-shift assignment	>>=
Bitwise AND assignment	&=
Bitwise inclusive OR assignment	|=
Bitwise exclusive OR assignment	^=

- **Group 17 (right to left associativity)**:

Operator Description	Operator
throw expression	throw

- **Group 18 (left to right associativity)**:

Operator Description	Operator
Comma	,

PHẦN 2: CẤU TRÚC RẼ NHÁNH

2.0 Boolean

Trong bài học này, chúng ta cùng tìm hiểu về một kiểu dữ liệu được dùng rất thường xuyên trong ngôn ngữ lập trình C++. Đó là kiểu dữ liệu **bool**.

Nhìn lại bảng các kiểu dữ liệu cơ bản trong các bài trước, chúng ta thấy kiểu **bool** có kích thước nhỏ nhất.

Category	Type	Minimum Size	Note
boolean	bool	1 byte	
character	char	1 byte	May be signed or unsigned
	wchar_t	1 byte	
	char16_t	2 bytes	C++11 type
	char32_t	4 bytes	C++11 type
integer	short	2 bytes	
	int	2 bytes	
	long	4 bytes	
	long long	8 bytes	C99/C++11 type
floating point	float	4 bytes	
	double	8 bytes	
	long double	8 bytes	

Kiểu dữ liệu **bool** được dùng để lưu trữ kết quả của một mệnh đề toán học trong máy tính.

Mệnh đề toán học là gì?

Mệnh đề toán học (hay còn gọi là mệnh đề logic) là một phát biểu mà nó chỉ có thể xảy ra một trong hai trường hợp: **đúng hoặc sai**.

Ví dụ:

- Mệnh đề A = "Chúng ta đang học lập trình C++". Mình có thể nói A là một mệnh đề đúng.
- Mệnh đề B = "5 là số chẵn". Đây hiển nhiên là một mệnh đề sai.

Vậy thì, kết quả đúng hoặc sai của một mệnh đề là một sự hiển nhiên, có thể thấy được ngay.

Tính đúng sai của một mệnh đề cũng có thể thay đổi theo thời gian.

Ví dụ:

- Mệnh đề C = "Hôm nay là thứ ba". Mệnh đề này có thể đúng hoặc sai tùy vào thời điểm mình phát biểu nó.

Những câu mệnh lệnh, cảm thán hay câu hỏi ... đều không thể đóng vai trò là một mệnh đề vì chúng không phản ánh được sự đúng hoặc sai.

Mệnh đề khẳng định và mệnh đề phủ định

Thử xét lại mệnh đề B ở ví dụ trên.

B = "5 là số chẵn". Đây chính là một mệnh đề khẳng định, nó khẳng định rằng 5 là số chẵn. Và nó cho chúng ta kết quả sai.

Vậy thì nếu chúng ta phủ định lại mệnh đề B, chúng ta sẽ được một mệnh đề có kết quả đúng.

X = "5 không phải là số chẵn".

Để phủ định một mệnh đề, chúng ta thường thêm vào từ **không** hoặc **không phải**. Nếu chúng ta phủ định mệnh đề B 2 lần, chúng ta được mệnh đề B ban đầu. Đây được gọi là quy luật phủ định của phủ định.

Mệnh đề trong ngôn ngữ lập trình C++

Ngôn ngữ C++ có hỗ trợ cho chúng ta việc biểu diễn các mệnh đề toán học. Mình lấy một số ví dụ như sau:

```
1 < 2; //đúng
5 > 10; //sai
1 + 1 == 2; //đúng

int a = 2, b = 4;
a * 3 != b; //đúng
```

Trên đây là một vài ví dụ về cách biểu diễn mệnh đề trong ngôn ngữ C++. Như các bạn thấy, máy tính không thể hiểu được các phát biểu bằng lời như "Đây là ngôn ngữ C++" hay là "Học lập trình không khó", chúng chỉ có thể hiểu được các mệnh đề dưới dạng các con số, các biểu thức so sánh...

Khai báo và khởi tạo biến kiểu bool

Kiểu **bool** là kiểu dữ liệu chỉ nhận một trong hai giá trị **true** (đúng) hoặc **false** (sai) tương ứng với kết quả của mệnh đề toán học trong C++.

Chúng ta khai báo (và khởi tạo) biến kiểu bool tương tự như cách khai báo biến có các kiểu dữ liệu mà các bạn đã được làm quen.

```
bool b;
```

Trong đó, **bool** là kiểu dữ liệu và **b** là tên biến.

Chúng ta có thể gán trực tiếp giá trị **true** hoặc **false** cho biến kiểu **bool**.

```
bool b1 = true;
bool b2(false);
bool b3 { true };
```

Giá trị của biến kiểu **bool** có thể bị đảo từ **true** sang **false** hoặc ngược lại nếu sử dụng toán tử **not (!)**.

```
bool b1 = !true; //not true => false
bool b2(!false); //not false => true
```

Khi biểu diễn giá trị của biến kiểu bool trên máy tính, nó hoàn toàn không phải là **true** hoặc **false** mà được định dạng kiểu **integer**. Giá trị **true** ứng với số **1**, giá trị **false** ứng với số **0**. Cùng thử chạy đoạn code mẫu dưới đây để kiểm chứng:

```
#include <iostream>
using namespace std;

int main()
{
    bool b(true);
    cout << b << endl; //1
    cout << !b << endl; //0

    bool b2(false);
    cout << b2 << endl; //0
    cout << !b2 << endl; //1

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

Sau khi chạy đoạn chương trình trên, kết quả chúng ta nhận được là:

```
1 0 0 1
```

Nếu các bạn muốn đối tượng **cout** in ra giá trị **true** hoặc **false** thay vì chỉ in ra các giá trị **0** hoặc **1**, các bạn có thể sử dụng **std::boolalpha**.

```
#include <iostream>
using namespace std;

int main()
{
        cout << true << endl;
        cout << false << endl;

        cout << boolalpha << endl;
        cout << true << endl;
        cout << false << endl;

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

Kết quả:

```
1
0
true
false
```

Kiểu **bool** chỉ có thể lưu trữ một trong hai giá trị **true** hoặc **false** tương ứng với giá trị 1 và 0 trong số nguyên, điều gì xảy ra nếu chúng ta gán cho biến kiểu bool những giá trị khác? Cùng thử chạy đoạn chương trình bên dưới để tìm kết quả:

```
#include <iostream>
using namespace std;

int main()
{
        bool b;
        cout << boolalpha;

        b = 0; cout << b << endl;
        b = 1; cout << b << endl;
        b = 100; cout << b << endl;
        b = -999; cout << b << endl;

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

Kết quả chúng ta được:

```
false
true
true
true
```

Khi gán những giá trị số nguyên cho biến kiểu **bool**, ngoài giá trị 0 ra, những giá trị khác đều được quy đổi về giá trị **true**.

Gán các mệnh đề toán học cho biến kiểu bool

Mình sẽ lấy lại một số ví dụ về các biểu thức biểu diễn mệnh đề toán học trong ngôn ngữ C++ như bên dưới.

```
1 < 2; //đúng
5 > 10; //sai
1 + 1 == 2; //đúng

int a = 2, b = 4;
a * 3 != b; //đúng
```

Những biểu thức này sẽ cho ra kết quả là giá trị đúng hoặc sai. Do đó, chúng ta có thể gán các biểu thức này cho biến kiểu **bool**. Ví dụ:

```
bool b1 = 1 < 2;
bool b2 = 5 > 10;
bool b3 = (1 + 1 == 2);

int a = 2, b = 4;
bool b4 = (a * 3 != b);

cout << b1 << " " << b2 << " " << b3 << " " << b4 << endl;
```

Kết quả đoạn lệnh trên sẽ cho ra kết quả

```
1 0 1 1
```

b1 có giá trị đúng vì mệnh đề **(1 < 2)** là đúng. **b2** có giá trị sai vì **(5 > 10)** là sai. Tương tự cho **b3** và **b4**.

Các mệnh đề toán học trong C++ được tạo nên từ những biểu thức chứa những toán tử quan hệ (relational operators). Các phép so sánh sẽ trả về giá trị **đúng** hoặc **sai**.

Các toán tử quan hệ (Comparisons)

Ngôn ngữ C++ đã định nghĩa 6 toán tử quan hệ dùng để so sánh các kiểu dữ liệu cơ bản.

Operator	Symbol	Form	Operation
Greater than	>	x > y	true if x is greater than y, false otherwise
Less than	<	x < y	true if x is less than y, false otherwise
Greater than or equals	>=	x >= y	true if x is greater than or equal to y, false otherwise
Less than or equals	<=	x <= y	true if x is less than or equal to y, false otherwise
Equality	==	x == y	true if x equals y, false otherwise
Inequality	!=	x != y	true if x does not equal y, false otherwise

Các bạn lưu ý phân biệt toán tử gán (=) và toán tử so sánh tương đương (==). Khi muốn thực hiện phép so sánh bằng, chúng ta sử dụng 2 dấu bằng liên tiếp nhau. Ngược lại với toán tử so sánh tương đương (==) là toán tử so sánh không tương đương (!=), toán tử này trả về giá trị đúng nếu 2 giá trị không bằng nhau.

Chúng ta lấy ví dụ sau để hiểu rõ hơn cách hoạt động của các toán tử quan hệ:

Tuổi của A là 15, tuổi của B là 20. Sử dụng các toán tử quan hệ cho tuổi của 2 người này, ta được bảng kết quả như sau:

Toán tử	Cách vận hành	Kết quả
==	So sánh tuổi A có bằng tuổi B hay không?	FALSE
!=	So sánh tuổi A có khác tuổi B hay không?	TRUE
>	So sánh tuổi A có lớn hơn tuổi B hay không?	FALSE
<	So sánh tuổi A có nhỏ hơn tuổi B hay không?	TRUE
>=	So sánh tuổi A có lớn hơn hoặc bằng tuổi B hay không?	FALSE
<=	So sánh tuổi A có lớn hơn hoặc bằng tuổi B hay không?	TRUE

So sánh số thực

Sử dụng các toán tử so sánh để thực hiện so sánh số thực có thể cho ra kết quả không mong muốn. Ví dụ:

```
#include <iostream>

int main()
{
    double d1(100 - 99.99); // should equal 0.01
    double d2(10 - 9.99); // should equal 0.01

        bool b1 = (d1 == d2);
        bool b2 = (d1 > d2);
        bool b3 = (d1 < d2);

        cout << b1 << endl;
        cout << b2 << endl;
        cout << b3 << endl;

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

Đoạn chương trình trên cho ra kết quả là

```
0
1
0
```

Có nghĩa là biểu thức so sánh **(d1 > d2)** là đúng. Trong chương trình trên, d1 = 0.0100000000000005116 và d2 = 0.0099999999999997868. Cả 2 giá trị này đều gần bằng 0.1, nhưng d1 lớn hơn d2 nên đã cho ra kết quả sai. Do đó, chúng nên tránh thực hiện so sánh số thực nếu không cần thiết.

Toán tử logic (logical operators)

Chúng ta sử dụng các toán tử quan hệ (relational operators) để kiểm tra một biểu thức mệnh đề cụ thể đúng hay sai, nhưng chúng chỉ có thể kiểm tra 1 mệnh đề tại 1 thời điểm. Đôi khi chúng ta cần kiểm tra cùng lúc nhiều mệnh đề trong cùng thời điểm.

Ví dụ: Khi chúng ta muốn kiểm tra thử có trúng vé số hay không, chúng ta cần so khớp nhiều chữ số khác nhau. Nếu tờ vé số có 5 chữ số, chúng ta cần 5 lần so sánh. Điều kiện trúng giải là tất cả các cặp chữ số đều phải khớp với nhau.

Một trường hợp khác, chúng ta cần kiểm tra rằng có ít nhất một mệnh đề trong số các mệnh đề đưa ra là đúng hay không.

Ví dụ: Nếu chúng ta muốn nghỉ làm việc trong hôm nay, phải có ít nhất 1 trong 2 mệnh đề sau đây là đúng. Thứ nhất là "chúng ta bị ốm", thứ hai là "chúng ta đã hoàn thành công việc". Hoặc mệnh đề "chúng ta bị ốm" đúng, hoặc mệnh đề "chúng ta đã hoàn thành công việc" đúng thì chúng ta có thể nghỉ làm việc hôm nay. Nếu chỉ sử dụng các toán tử so sánh, chúng ta phải thực hiện so sánh 2 lần.

Toán tử logic (logical operators) hỗ trợ cho chúng ta kiểm tra nhiều mệnh đề cùng một lúc.

Ngôn ngữ C++ cung cấp cho chúng ta 3 toán tử logic:

Operator	Symbol	Form	Operation
Logical NOT	!	!x	true if x is false, or false if x is true
Logical AND	&&	x && y	true if both x and y are true, false otherwise
Logical OR	||	x || y	true if either x or y are true, false otherwise

Toán tử NOT

Toán tử NOT kí hiệu là (!) là toán tử một ngôi có chức năng đảo ngược giá trị của biến kiểu **bool**. Khi sử dụng, chúng ta đặt toán tử NOT đứng trước giá trị kiểu **bool** hoặc biến kiểu **bool**.

Ví dụ:

```
!true;
!false;
bool b = false;
bool b1 = !b;
```

Dưới đây là bảng chân trị của toán tử NOT:

Logical NOT (operator !)	
Right operand	Result
TRUE	FALSE
FALSE	TRUE

Nếu toán tử NOT tác động đến giá trị True, nó sẽ chuyển thành giá trị False và ngược lại.

Toán tử OR

Toán tử OR là một toán tử hai ngôi dùng để kiểm tra một trong hai mệnh đề có đúng hay không. Ví dụ: "Tôi thích chơi game" OR "Tôi thích học lập trình C++". Nếu mệnh đề "Tôi thích chơi game" đúng, hoặc mệnh đề "Tôi thích học lập trình C++" đúng thì toán tử OR trả về kết quả đúng.

Logical OR (operator ||)		
Left operand	Right operand	Result
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

Ví dụ:

```
int value = 1;
value == 0 || value == 1; // true
value == 0 || value == 2; // false
```

Toán tử AND

Toán tử AND là một toán tử hai ngôi dùng để kiểm tra cả hai mệnh đề có đều đúng hay không. Dưới đây là bảng chân trị của toán tử AND:

Logical AND (operator &&)		
Left operand	Right operand	Result
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

Ví dụ:

```
int value = 1;
value != 0 && value != 2; //true
value == 1 && value == 2; //false
```

2.1 Giới thiệu một số cấu trúc điều khiển

Như chúng ta đã tìm hiểu, khi chạy một chương trình C++, CPU bắt đầu thực thi các câu lệnh tại điểm trên cùng của hàm **main**, thực hiện lần lượt các câu lệnh từ trên xuống dưới, và kết thúc tại điểm dưới cùng của hàm **main**. Chuỗi các câu lệnh được CPU thực thi gọi là **program's path**. Phần lớn các chương trình mà bạn từng thấy được thực thi theo dạng **straight-line** (tuần tự từ trên xuống dưới). Tuy nhiên, trong một số trường hợp, đây không phải là điều chúng ta muốn.

Ví dụ nếu chúng ta yêu cầu người dùng đưa ra một lựa chọn, và người dùng nhập vào lựa chọn không phù hợp, chúng ta nên yêu cầu người dùng đưa ra một lựa chọn khác. Với cấu trúc chương trình dạng **straight-line**, điều này là bất khả thi.

Một trường hợp khác, chúng ta muốn chương trình thực hiện lặp đi lặp lại một công việc nào đó với số lần thực hiện chưa biết trước. Ví dụ chúng ta muốn in ra điểm số của một trò chơi trên màn hình cho đến khi trò chơi kết thúc, chúng ta không thể biết chính xác thời điểm kết thúc trò chơi là khi nào.

Do đó, ngôn ngữ C++ cung cấp các cấu trúc điều khiển **(control flow statements)** nó cho phép lập trình viên thay đổi hướng đi của chương trình. Có một số dạng cấu trúc điều khiển khác nhau và mình sẽ giới thiệu sơ lược để các bạn có sự hình dung ban đầu.

Halt

Cấu trúc điều khiển dừng (**halt**) là một cấu trúc thường gặp, nó yêu cầu chương trình ngừng làm việc ngay lập tức. Trong C++, cấu trúc **Halt** có thể được thực hiện thông qua hàm **exit()** trong thư viện **cstdlib**. Hàm exit nhận vào một giá trị số nguyên và nó sẽ được trả về cho hệ điều hành như một mã kết thúc chương trình, tương tự như giá trị trả về của hàm **main**.Đây là đoạn chương trình mẫu cho việc thực hiện cấu trúc điều khiển **Halt**:

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
	cout << "This line is printed out." << endl;
	exit(-1); //Terminate and return -1 to operating system.
	cout << "This line will never be printed out." << endl;

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

Jumps

Cấu trúc điều khiển tiếp theo mình muốn đề cập đến là **Jump**. Cấu trúc Jump không điều kiện khiến CPU nhảy đến thực thi một số các câu lệnh khác. **goto, break, continue** là các từ khóa được sử dụng trong cấu trúc Jump, chúng có kiểu Jump khác nhau, chúng ta sẽ được tìm hiểu chi tiết trong các bài học sắp tới.

Cấu trúc rẽ nhánh có điều kiện

Cấu trúc rẽ nhánh có điều kiện khiến chương trình thay đổi hướng thực thi dựa trên giá trị của biểu thức điều kiện (hoặc các mệnh đề). Tiêu biểu cho cấu trúc rẽ nhánh là **câu lệnh if**.

```
int main()
{
	//do A

	if(expression)
		//do B;
	else
		//do C;

	//do D

	return 0;}
```

Chương trình này có 2 hướng có thể đi. Nếu biểu thức expression cho kết quả đúng (**true**), chương trình sẽ thực thi A rồi đến B và đến D. Nếu biểu thức expression cho kết quả sai (**false**), chương trình sẽ đi theo hướng A đến C rồi đến D. Cấu trúc này không còn dạng **straight-line** nữa mà là dạng cấu trúc rẽ nhánh (**conditional branches**).

Cấu trúc vòng lặp (Loops)

Một cấu trúc vòng lặp khiến chương trình thực hiện lặp đi lặp lại một chuỗi các câu lệnh cho đến khi không còn thõa mãn một điều kiện nào đó.

```
int main()
{
        //do A
        //do B 0 or more times
        //do C
}
```

Chương trình này có thể thực hiện theo hướng ABC, ABBC, ABBBC, ABBB...BBBC, hoặc cũng có thể là AC. Như các bạn thấy, một lần nữa đây không phải là **straight-line program**, hướng thực thi các câu lệnh phụ thuộc vào số lần các câu lệnh trong vòng lặp được thực thi.

while, do...while, for là 3 cấu trúc vòng lặp mà ngôn ngữ C/C++ cung cấp. Chuẩn C++11 còn cung cấp cho chúng ta thêm cấu trúc vòng lặp tên là **for each**.

Exceptions

Cuối cùng, **exceptions** là một cơ chế xử lý lỗi xảy ra bên trong hàm. Nếu một lỗi xảy ra bên trong hàm mà hàm không thể xử lý, hàm đó ném ra một ngoại lệ (exception). Điều này khiến chương trình nhảy đến khối lệnh chuyên dùng để xử lý ngoại lệ có kiểu tương ứng với ngoại lệ được hàm ném ra.

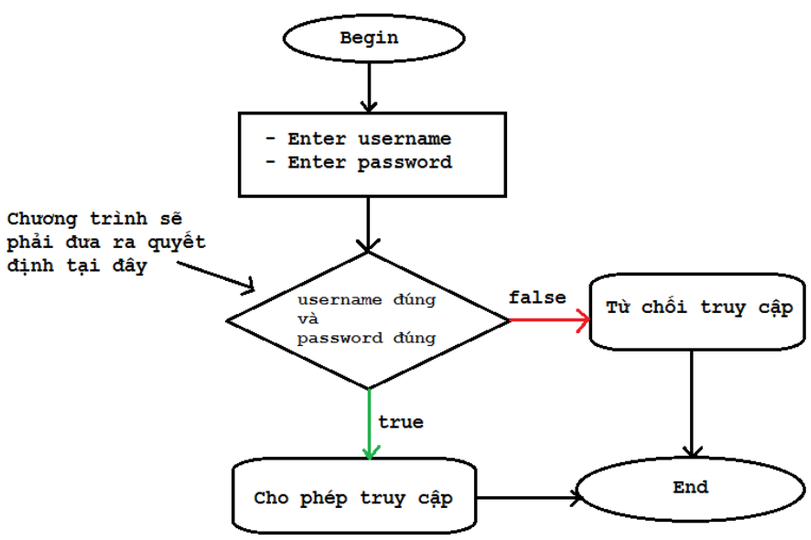
Xử lý ngoại lệ là một đặc trưng khá mới được hổ trợ trong ngôn ngữ C++.

2.2 if statements

Trong bài học trước, mình đã giới thiệu đến các bạn một số cấu trúc điều khiển thường thấy trong chương trình C++, các cấu trúc điều khiển có khả năng quyết định phần code nào sẽ được thực thi tại thời điểm chương trình đang chạy. Và cấu trúc đầu tiên mình muốn giúp các bạn tìm hiểu là **cấu trúc rẽ nhánh có điều kiện (conditional branch)** với **câu lệnh if (if statement)**.

If statement

If statement cho phép chúng ta điều khiển chương trình thực hiện một đoạn lệnh nào đó dựa trên biểu thức điều kiện có giá trị **true** hoặc **false**. Quan trọng hơn là **if statement** cho phép chúng ta làm điều này dựa trên input của người dùng. Ví dụ: sử dụng if statement để kiểm tra username và password, và chương trình sẽ quyết định người dùng có được phép truy cập vào hệ thống hay không.



Về mặt ngữ nghĩa, if statement cũng giống như luật nhân - quả. Ví dụ: Nếu tôi hết tiền trong tài khoản thì tôi không thể rút tiền từ máy ATM. Vậy biểu thức điều kiện ở đây là số tiền trong tài khoản vẫn còn, nếu đúng thì máy ATM cho phép rút tiền và ngược lại.

Cấu trúc cơ bản của if statement

Dưới đây là cấu trúc cơ bản nhất của một **if statement**:

```
if (expression)
        execute the next statement;
```

Câu lệnh tiếp theo thuộc **if statement** chỉ được thực hiện nếu biểu thức điều kiện **expression** có giá trị **true**.

Biểu thức điều kiện được tạo ra từ một hoặc nhiều mệnh đề toán học.
Ví dụ:

```
int main()
{
        if (5 < 10)
                cout << "(5 < 10) is true" << endl;

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

Trong đoạn chương trình trên, chúng ta đánh giá mệnh đề (5 < 10) để xem nó có đúng hay không. Điều này hiển nhiên đúng, do đó, câu lệnh đứng ngay sau **if statement** sẽ được thực hiện.

If with multiple statments

Để thực hiện nhiều hơn 1 câu lệnh khi biểu thức điều kiện trong if statement đúng, chúng ta đặt thêm cặp dấu ngoặc nhọn phía sau để tạo thành một khối lệnh. Ví dụ:

```
int main()
{
        if (5 < 10)
        {
                cout << "(5 < 10) is true" << endl;
                cout << "This line will be printed" << endl;
                cout << "Because they are inside the block of if statement" << endl;
        }

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

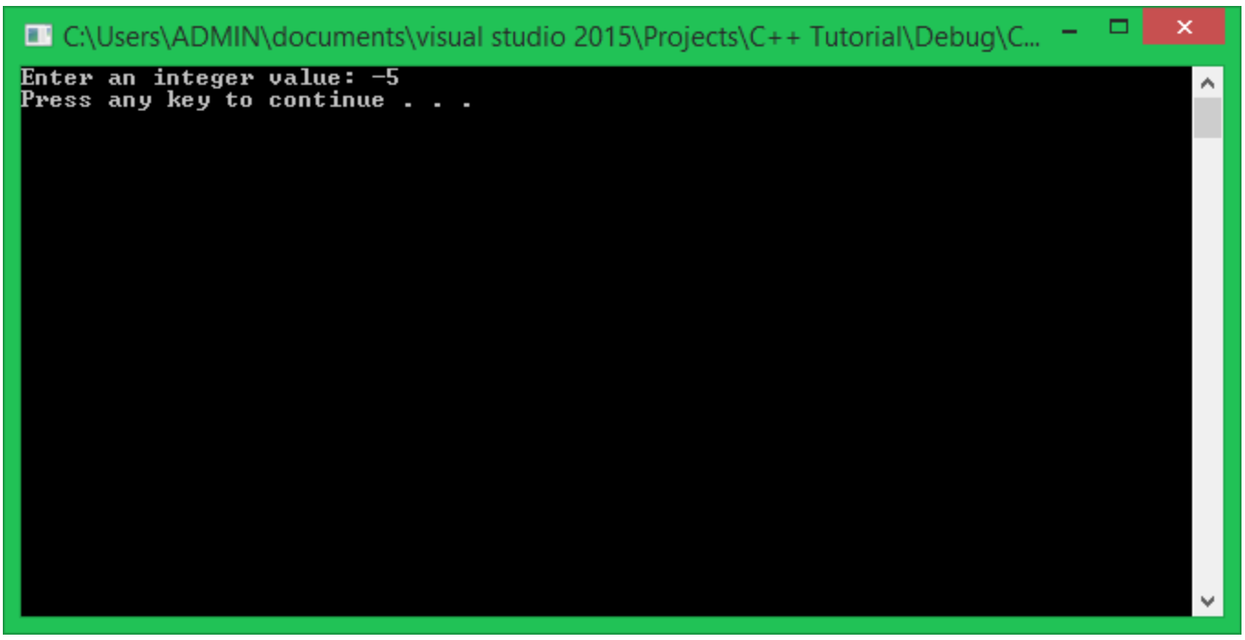
Mình khuyến nghị các bạn nên có thói quen sử dụng cặp dấu ngoặc nhọn đứng sau **if statement** cho dù phía sau nó chỉ có 1 lệnh cần xử lý.

Else

Chúng ta thử lấy thêm 1 ví dụ về **if statement** cơ bản:

```
int value;
cout << "Enter an integer value: "; cin >> value;

if(value >= 0)
{
        cout << "Positive number" << endl;
}
```

Với đoạn code trên, điều gì sẽ xảy ra nếu chúng ta nhập giá trị -5 cho biến value?

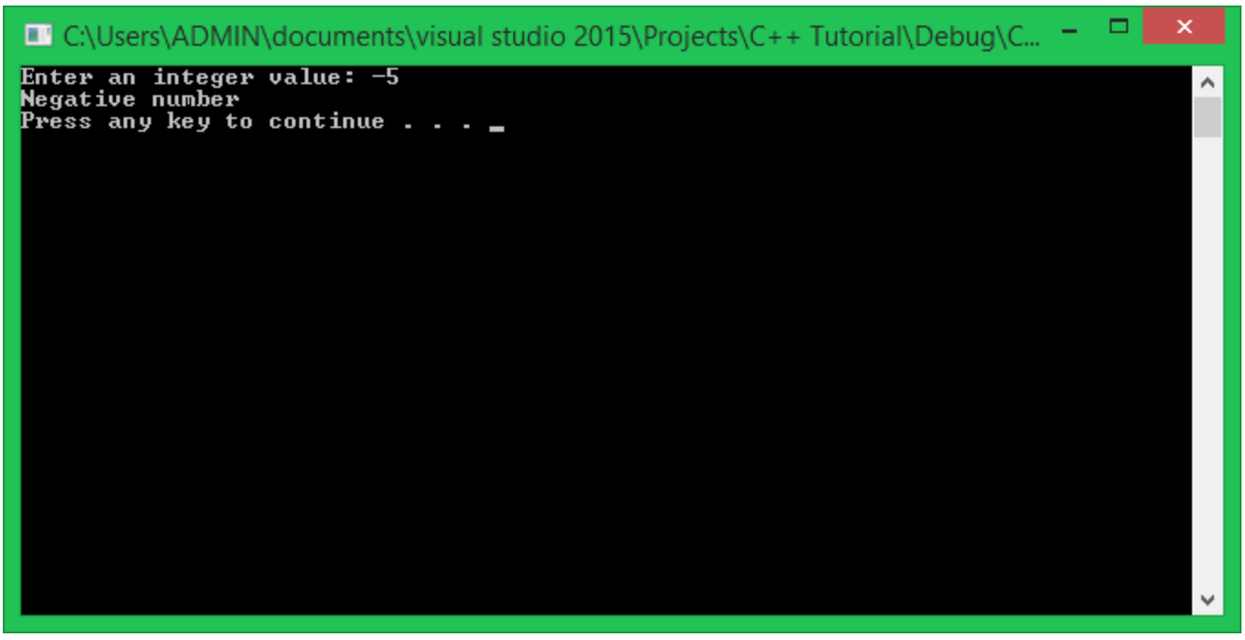


Kết quả sau khi nhập giá trị xong thì chương trình không làm gì cả. Vì giá trị -5 làm sai biểu thức điều kiện **(value >= 0)**, do đó, chương trình bỏ qua dòng lệnh **cout** bên dưới.

Trong một số trường hợp, chúng ta muốn thực thi một số lệnh khi biểu thức điều kiện sai, thay thế cho trường hợp biểu thức điều kiện đúng. Lúc này, từ khóa "**else**" sẽ giúp chúng ta làm điều đó.

```
int value;
cout << "Enter an integer value: "; cin >> value;

if (value >= 0)
{
	cout << "Positive number" << endl;
}
else
{
	cout << "Negative number" << endl;
}
```

Bây giờ khi nhập lại giá trị -5, chúng ta được kết quả như sau:



Từ khóa "**else**" có nghĩa là trường hợp ngược lại với biểu thức điều kiện trong **if statement**. Khi biểu thức điều kiện trong **if statement** có giá trị **false**, khối lệnh đứng sau **else statement** sẽ được thực thi.

Chaining if statements

Với cách dùng cấu trúc **if ... else** như trên, chúng ta chỉ có thể kiểm tra 2 trường hợp phủ định của nhau. Một cách dùng khác của **if statement** là khi cần kiểm tra nhiều trường hợp khác nhau có thể xảy ra. Chúng ta có thể dùng "**else if**" statement để đưa vào những trường hợp cần kiểm tra khác.

Cấu trúc khi sử dụng **"else if" statement**:

```
if (expression)
{
        // do A
}
else if (another_expression)
{
        // do B
}
else if (one_more_expression)
{
        // do C
}
else
{
        // do D
}
```

Bằng cách nối chuỗi các **if statement**, chúng sẽ được lần lượt kiểm tra các điều kiện từ trên xuống dưới, nếu tìm thấy biểu thức điều kiện có kết quả **true**, khối lệnh tương ứng sẽ được thực thi, những khối lệnh còn lại sẽ bị bỏ qua.

Ví dụ:

```
int score;
cout << "Enter your average score: "; cin >> score;

if (score <= 5)
        cout << "You need to try more" << endl;
else if (score <= 7)
        cout << "Not bad" << endl;
else if (score <= 9)
        cout << "Good job" << endl;
else
        cout << "Incredible" << endl;
```

Với đoạn chương trình như trên, ban đầu biến score được nhập giá trị từ bàn phím và bắt đầu kiểm tra với mệnh đề **(score <= 5)**, nếu sai, **if statement** sẽ đi đến mệnh đề tiếp theo và cứ thế cho đến khi nào tìm được mệnh đề có giá trị đúng. Nếu toàn bộ các mệnh đề đưa ra đều sai, chương trình không thực hiện **if statement**.

Nesting if statements

Ngôn ngữ C++ cho phép chúng ta viết những **if statements** lồng vào nhau, ví dụ:

```
float a, b;
cin >> a >> b;

if (b != 0)
{
        if (a <= 0)
                cout << "a = 0 is not accepted" << endl;
        else
                cout << "a / b = " << a / b << endl;
}
else
{
        cout << "Divided by zero" << endl;
}
```

Đối với các trường hợp sử dụng if statements lồng vào nhau, chúng ta nên sử dụng thêm các cặp dấu ngoặc nhọn để giúp chương trình rõ ràng hơn.

Với đoạn chương trình trên, phép toán **a / b** chỉ được thực hiện khi cả điều kiện **b != 0** đúng và **a <= 0** sai.

Sử dụng các toán tử logic với if statement

Sử dụng các toán tử logic (AND, OR, NOT, ...) có thể giúp chúng ta kiểm tra nhiều trường hợp khác nhau chỉ cần dùng 1 mệnh đề ghép. Lấy ví dụ trong phần **Nesting if statements**, chúng ta sẽ thực hiện phép toán **a / b** khi cùng lúc thỏa mãn cả 2 điều kiện **(b != 0) AND (a > 0)**, chúng ta có thể sửa lại cấu trúc đoạn chương trình trên thành như sau:

```
if (b != 0 && a > 0)
{
        cout << "a / b = " << a / b << endl;
}
else
```

```
{
        if (b == 0)
                cout << "Divided by zero" << endl;
        if (a <= 0)
                cout << "a <= 0 is not accepted" << endl;
}
```
Các bạn cần dựa vào bảng chân trị của từng toán tử logic để sử dụng vào mệnh đề cho hợp lý.

Lệnh if ngắn gọn

Ta có thể sử dụng lệnh if ngắn gọn với cấu trúc như sau

```
expression ? do A : do B;
```
Lệnh này có thể hiểu như sau:

```
Điều kiện ? nếu đúng thực hiện A : nếu sai thực hiện B;
```

Tổng kết

Trên đây mình vừa giới thiệu cho các bạn về một trong số các cấu trúc rẽ nhánh có điều kiện được sử dụng phổ biến, và một số cách sử dụng **if statement** khác nhau tùy vào tình huống cụ thể.

Bài tập cơ bản

1/ Viết chương trình sinh ra một số ngẫu nhiên trong khoảng từ 0 đến 100 nhưng không in ra màn hình. Yêu cầu người dùng đoán xem số ngẫu nhiên vừa sinh ra lớn hơn 50 hay bé hơn 50, nếu chọn trường hợp bé hơn 50 nhập giá trị 0, ngược lại nhập giá trị 1. In kết quả thông báo người dùng đã đoán đúng hay sai ra màn hình.

Ví dụ:

```
A random number was generated
Guess: 1

Wrong! The random number is 26 (26 < 50)
```
2/ Viết chương trình giải phương trình bậc 2: ax2 + bx + c = 0 với a, b và c nhập từ bàn phím.

Hướng dẫn:

- Nhập vào 3 biến a, b, c.
- Tính delta = b*b - 4*a*c
- Nếu delta < 0 thì kết luận phương trình vô nghiệm.
- Ngược lại, nếu delta = 0 thì kết luận nghiệm x1 = x2 = -b/(2*a)
- Ngược lại, nếu delta > 0 thì kết luận x1 = (-b + sqrt(delta)) / (2*a) và x2 = (-b - sqrt(delta)) / (2*a).

2.3 switch case statements

Trong bài học này, chúng ta cùng nhau tìm hiểu câu lệnh được xây dựng sẵn trong ngôn ngữ lập trình C++ cũng được đưa vào dạng cấu trúc rẽ nhánh có điều kiện. Đó là câu lệnh được tạo nên bởi từ khóa **switch** và **case**, còn gọi là **switch case statement**.

switch case statement thường được dùng để thay thế cho **if statement** trong trường hợp số lượng trường hợp cần so sánh quá dài. Mặc dù chúng ta có thể sử dụng kỹ thuật **chaining if statement** để nối các trường hợp cần kiểm tra lại với nhau, nhưng nó khiến chương trình khó đọc. Ví dụ:

```
int main()
{
        cout << "1: BLACK\n2: BLUE\n3: GREEN\n4: YELLOW\n5: WHITE" << endl;
        cout << "Enter your favorite color: ";
        int color;
        cin >> color;

        if (color == 1)
                cout << "You like BLACK color" << endl;
        else if (color == 2)
                cout << "You like BLUE color" << endl;
        else if (color == 3)
```

```
                cout << "You like GREEN color" << endl;
        else if (color == 4)
                cout << "You like YELLOW color" << endl;
        else if (color == 5)
                cout << "You like WHITE color" << endl;
        else
                cout << "Unknown" << endl;

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

Cũng với ví dụ mẫu này, mình sẽ chuyển nó về cấu trúc switch ... case để các bạn có cái nhìn đầu tiên về **switch case statement**:

```
int main()
{
        cout << "1: BLACK\n2: BLUE\n3: GREEN\n4: YELLOW\n5: WHITE" << endl;
        cout << "Enter your favorite color: ";
        int color;
        cin >> color;

        switch (color)
        {
                case 1:
                        cout << "You like BLACK color" << endl;
                        break;
                case 2:
                        cout << "You like BLUE color" << endl;
                        break;
                case 3:
                        cout << "You like GREEN color" << endl;
                        break;
                case 4:
                        cout << "You like YELLOW color" << endl;
                        break;
                case 5:
                        cout << "You like WHITE color" << endl;
                        break;
                default:
                        cout << "Unknown" << endl;
                        break;
        }

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

Các bạn có thể hiểu cấu trúc rẽ nhánh với **switch case statement** như sau: **switch** nhận vào 1 biến hoặc 1 biểu thức có giá trị kiểu số nguyên, mỗi nhãn **case** sẽ gắn kèm với 1 số nguyên cụ thể và chúng sẽ được lần lượt so sánh bằng (equality) với giá trị của biến (hoặc biểu thức) trong **switch**. Nếu nhãn case nào có giá trị tương xứng với biến hoặc biểu thức trong **switch**, những câu lệnh đứng sau nhãn đó sẽ được thực thi. Nếu không có nhãn nào có giá trị tương xứng, các câu lệnh đứng sau nhãn **default** sẽ được thực thi.

Như vậy, chúng ta có thể đưa ra khuôn dạng của **switch case statement** như sau:

```
switch ( <variable> )
{
case this-value:
        //Code to execute if <variable> == this-value
        break;
case that-value:
        Code to execute if <variable> == that-value
        break;
//.....................
default:
        //Code to execute if <variable> does not equal the value following any of the cases
        break;
}
```

- **switch statement** được bắt đầu bởi từ khóa **switch**, theo sau đó là một giá trị số nguyên (thường là một biến đơn), có thể là một biểu thức **(ví dụ 3 * 2 + 5)**. Một hạn chế của switch statement là biểu thức điều kiện chỉ có thể thuộc 1 trong số các kiểu số nguyên **(char, short, int, long, int32_t, enum, ...).**

- **case label** được định nghĩa thông qua từ khóa **case**, theo sau từ khóa case là một hằng số, một giá trị cụ thể. **Lưu ý: Các nhãn case khác nhau phải được theo sau bởi các giá trị khác nhau.** Ví dụ:

```
switch (x)
{
  case 1:
  case 1: //illegal
}
```

- **default label** được định nghĩa thông qua từ khóa **default**. Những câu lệnh đứng sau nhãn này chỉ được thực thi nếu không có nhãn **case** nào có giá trị tương ứng với biểu thức điều kiện của **switch**. Lưu ý: cấu trúc switch ... case có thể không cần sử dụng nhãn **default**.

- **break** là một từ khóa trong ngôn ngữ C/C++, khi sử dụng trong **switch case statement** sẽ khiến chương trình thoát ra khỏi khối lệnh của cấu trúc **switch**. Chúng ta cần đặt từ khóa **break** tại cuối các câu lệnh của mỗi nhãn **case** để ngăn cách các **case** riêng biệt. Chúng ta cùng nhìn vào ví dụ sau khi không sử dụng từ khóa **break** thì điều gì sẽ xảy ra:

```
switch (2)
{
   case 1: // Does not match
       std::cout << 1 << '\n'; // skipped
   case 2: // Match!
       std::cout << 2 << '\n'; // Execution begins here
   case 3:
       std::cout << 3 << '\n'; // This is also executed
   case 4:
       std::cout << 4 << '\n'; // This is also executed
   default:
       std::cout << 5 << '\n'; // This is also executed
}
```

Với ví dụ này, kết quả in ra sẽ là: `2 3 4 5`

Đây là kết quả ngoài ý muốn, trường hợp này được gọi là **fall-through**. Để khắc phục trường hợp này, chúng ta cần sử dụng thêm từ khóa **break** đặt tại cuối mỗi nhãn **case**:

```
switch (2)
{
   case 1: // Does not match
       std::cout << 1 << '\n'; // skipped
     break;
   case 2: // Match!
       std::cout << 2 << '\n'; // Execution begins here
            break;
   case 3:
       std::cout << 3 << '\n'; // This is also executed
     break;
   case 4:
       std::cout << 4 << '\n'; // This is also executed
     break;
   default:
       std::cout << 5 << '\n'; // This is also executed
}
```

Khai báo và khởi tạo biến trong case statement

Chúng ta có thể khai báo biến bên trong mỗi case statement, nhưng chúng ta không thể khởi tạo giá trị cho chúng.

```
switch (x)

{
    case 1:
        int y; // declaration is allowed
        y = 4; // this is an assignment
        break;

    case 2:
        y = 5; // y was declared above, so we can use it here too
        break;

    case 3:
        int z = 4; // illegal, you can't initialize new variables in the case statements
        break;
```

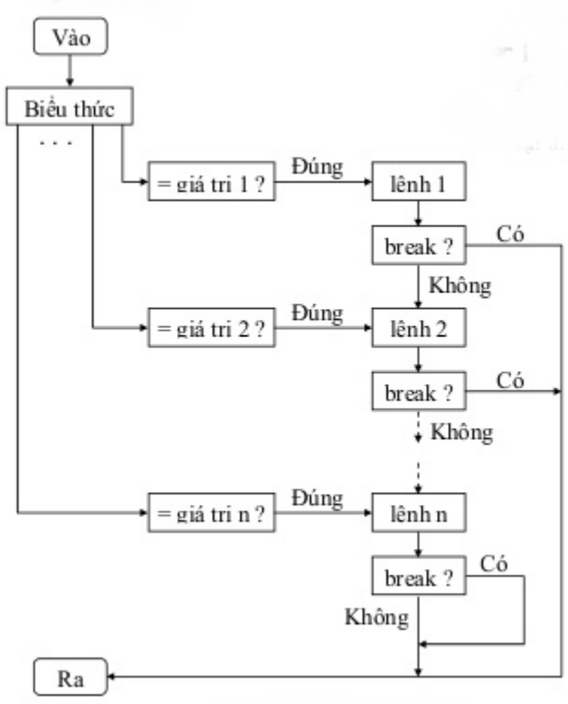
```
    default:
        std::cout << "default case" << std::endl;
        break;
}
```

Nhưng chúng ta nên tránh khai báo biến bên trong case statement, nó sẽ khiến chương trình chúng ta khó đọc hơn. Chúng ta có nhiều giải pháp thay thế dễ hiểu hơn, nếu có dịp mình sẽ hướng dẫn cho các bạn.

Tổng kết

Vậy là chúng ta đã làm quen thêm một dạng cấu trúc rẽ nhánh có điều kiện khác. If statement được sử dụng khi muốn kiểm tra tính đúng sai của một hoặc một số mệnh đề. Switch case statement được sử dụng khi muốn kiểm tra một giá trị số nguyên. Đối với trường hợp số lượng biểu thức điều kiện cần so sánh là quá nhiều, chúng ta ưu tiên sử dụng **switch case statement** hơn vì cú pháp rõ ràng hơn.

Chúng ta có thể biểu diễn cấu trúc của **switch case statement** dưới dạng sơ đồ khối như sau:



Bài tập cơ bản

Viết chương trình nhập vào tháng, in ra tháng đó có bao nhiêu ngày.

Gợi ý: Nhập vào tháng

- Nếu là tháng 1, 3, 5, 7, 8, 10, 12 thì có 31 ngày.
- Nếu là tháng 4, 6, 9, 11 thì có 30 ngày.
- Nếu là tháng 2 thì có 28 ngày (vì chúng ta chưa xét đến năm).

Lợi dụng trường hợp **fall-through** để tổ chức chương trình ngắn gọn hơn.

PHẦN 3: CẤU TRÚC VÒNG LẶP

3.0 Vòng lặp while

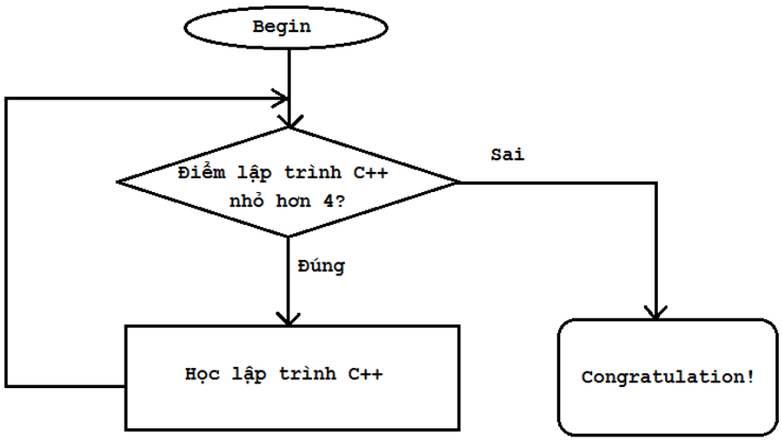
Trong chương trình, chúng ta sẽ cùng nhau tìm hiểu về cấu trúc vòng lặp **(Loops)** - một trong những cấu trúc điều khiển được sử dụng phổ biến trong ngôn ngữ C++.

Cấu trúc vòng lặp khiến chương trình thực hiện lặp đi lặp lại một chuỗi các câu lệnh cho đến khi không còn thõa mãn một điều kiện nào đó.
Ví dụ:

```
int main()
{
	//do A
	//do B 0 or more times
	//do C
}
```

Như vậy, chúng ta hiểu rằng có một cấu trúc vòng lặp được đặt tại B, nó có thể bắt buộc chương trình thực hiện công việc B 1 lần, hoặc cũng có thể thực hiện công việc B vô số lần mà không thể chuyển sang thực hiện công việc C sau đó. Số lần thực hiện công việc B nó sẽ phụ thuộc vào biểu thức điều kiện được đặt trong cấu trúc vòng lặp.

Mình lấy một ví dụ trong đời sống hằng ngày. *Sinh viên A đăng kí học môn lập trình C++ tại trường đại học, nếu sinh viên A không đủ điểm để qua môn học này thì sinh viên A sẽ phải học lại. Trong trường hợp sinh viên A phải học lại lần thứ 2, chúng ta lại nói rằng nếu sinh viên A không đủ điểm qua môn học này thì sinh viên A phải học lại... Vậy việc sinh viên A học lại là công việc sẽ được lặp đi lặp lại nhiều lần trong khi điều kiện sinh viên A đủ điểm để qua môn vẫn còn sai.*



While statements

While statement là cấu trúc vòng lặp đơn giản nhất trong số các kiểu vòng lặp cơ bản mà ngôn ngữ C++ cung cấp.

```
while (expression)
{
	statements;
}
```

Vòng lặp **while** được định nghĩa bởi từ khóa **while**. Một khi vòng lặp while được thực thi, biểu thức điều kiện trong **while** sẽ được đánh giá. Nếu biểu thức điều kiện cho giá trị đúng, các câu lệnh trong khối lệnh của vòng lặp **while** sẽ được thực thi.

Trong trường hợp chúng ta chỉ thực hiện 1 câu lệnh khi biểu thức điều kiện đúng, chúng ta có thể bỏ cặp dấu ngoặc nhọn đi.

```
while (expression)
	statement;
```

Nhưng mình không khuyến khích điều này.

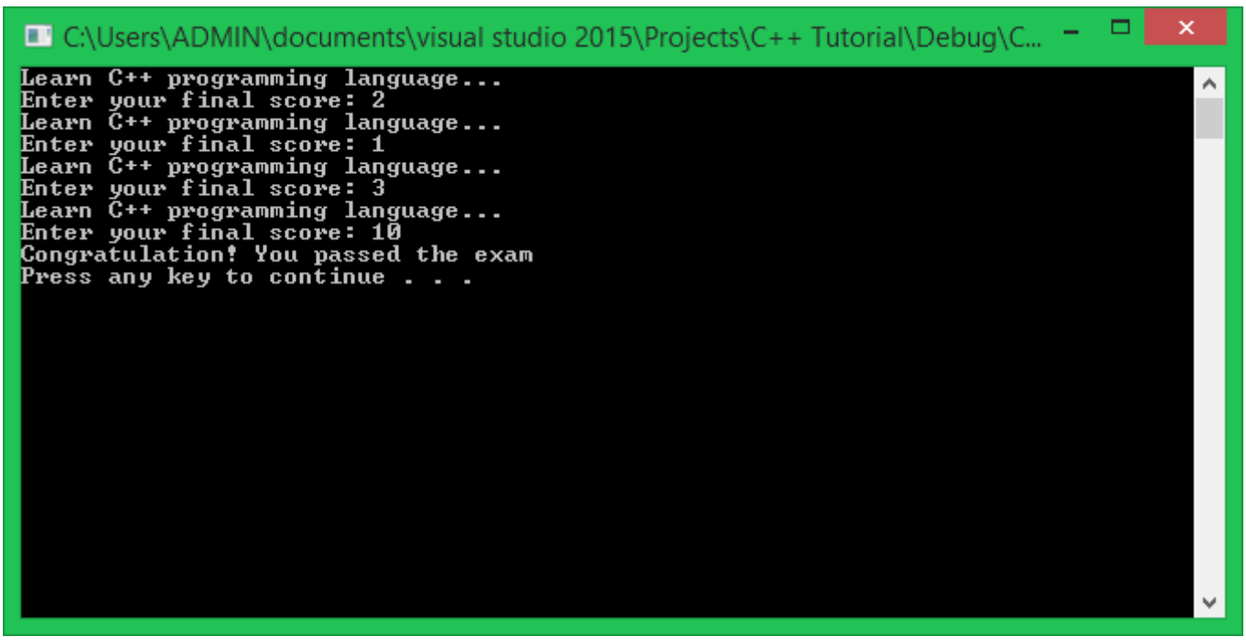
Khác với **if statement**, một khi kết thúc 1 lần lặp của vòng lặp **while**, chương trình sẽ quay lên lại vị trí bắt đầu vòng lặp **while** để đánh giá lại biểu thức điều kiện, nếu biểu thức điều kiện vẫn còn cho giá trị đúng, các câu lệnh trong khối lệnh của vòng lặp **while** được thực hiện lại.

Ví dụ:

```
int main()
{
	int score = 0;
	while (score < 4)
	{
		cout << "Learn C++ programming language..." << endl;
		cout << "Enter your final score: ";
		cin >> score; //new score
	}
```

```
        cout << "Congratulation! You passed the exam" << endl;

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

Trong ví dụ trên, đến khi nào biến score không còn thõa mãn điều kiện (score < 4) thì vòng lặp while mới kết thúc. Chúng ta cùng xem kết quả chương trình:



Khi số điểm được nhập vào là 10, ngay lập tức mệnh đề (score < 4) được đánh giá là false, vòng lặp ngừng thực thi ngay sau đó.

Với vòng lặp **while**, chúng ta không thể biết trước số lần lặp lại khối công việc. Chúng ta chỉ biết rằng, vòng lặp **while** sẽ ngừng thực thi khi nào biểu thức điều kiện cho giá trị **false**.
Chúng ta có thể sử dụng thêm một biến để đếm số lần thực hiện khối lệnh của vòng lặp while:

```
int score = 0;
int count = 0;

while (score < 4)
{
        count++;
        cout << "Learn C++ programming language..." << endl;
        cout << "Enter your final score: ";
        cin >> score; //new score
}

cout << "Congratulation! You passed the exam after " << count << " times" << endl;
```

Infinite loops

Vòng lặp vô tận xảy ra trong trường hợp không có sự tác động đến biểu thức điều kiện của vòng lặp **while** và nó luôn luôn đúng. Ví dụ:

```
int count = 0;
while (count < 10)
{
        cout << count << " ";
}
```

Biến count trong trường hợp này không bị ai tác động đến giá trị, nên nó vẫn là 0 và luôn bé hơn 10. Dó đó, điều kiện luôn luôn đúng và vòng lặp không thể kết thúc được.

Chúng ta có thể cố ý khai báo vòng lặp vô hạn bằng cách sau:

```
while (true)
{
        //This loop will be executed forever
}
```

Cách duy nhất để thoát ra khỏi vòng lặp vô tận là sử dụng từ khóa break, return, goto,...

Loop variables

Thông thường, chúng ta muốn vòng lặp thực hiện công việc trong một số lần có giới hạn. Để làm điều này, chúng ta thường sử dụng thêm các biến vòng lặp (loop variable), những biến này thường được sử dụng cho mục đích đếm số lần thực hiện khối lệnh của vòng lặp.

Mình lấy ví dụ chương trình đếm ngược từ 10 về 0 như sau:

```
int count = 10;
while (count >= 0)
{
	_sleep(1000); //stop 1000 miliseconds
	if (count == 0)
		cout << "Finished" << endl;
	else
		cout << count << " ";

	count--;
}
```

Kết quả:

```
10 9 8 7 6 5 4 3 2 1 Finished
```

Tại cuối vòng lặp, mình thực hiện trừ giá trị của biến count đi 1, điều này sẽ dẫn đến mệnh đề (count >= 0) sẽ sai trong tương lai. Như vậy, vòng lặp while này có điểm dừng.

Nest while loops

Cũng tương tự như **if statement** hay **switch case statement**, chúng ta có thể đặt vòng lặp while bên trong khối lệnh của vòng lặp while khác.

```
int outer = 1;
while (outer <= 5)
{
	int inner = 1;
	while (inner <= 5)
	{
		cout << inner << " ";
		inner++;
	}
	cout << endl; // print a newline at the end of each row

	outer++;
}
```

Cứ mỗi lần lặp của vòng lặp while ngoài, chương trình lại thực hiện toàn bộ vòng lặp while trong, sau đó thực hiện tăng biến outer lên 1 giá trị. Kết quả chương trình sẽ là:

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

Tổng kết

Tìm hiểu về vòng lặp while giúp chúng ta định hình tư duy về cấu trúc vòng lặp trong ngôn ngữ C/C++. Cấu trúc vòng lặp while khá ngắn gọn, dễ hiểu. Chúng ta thường sử dụng vòng lặp while cho các trường hợp số lần lặp lại công việc là chưa biết trước.

Bài tập cơ bản

1/ Viết chương trình tính tổng các số nguyên được nhập từ bàn phím cho đến khi nhập số 0 thì dừng.

2/ Viết chương trình in ra tất cả ký tự thuộc bảng mã ASCII từ 0 đến 127.

3/ Viết chương trình sử dụng vòng lặp while để chương trình in ra như sau:

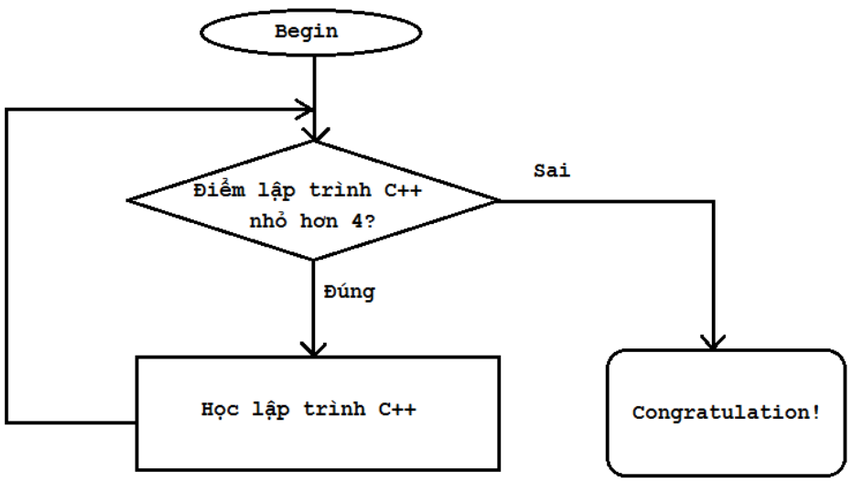
```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

3.1 Vòng lặp do-while

Trong bài học trước, chúng ta đã tìm hiểu về cấu trúc và cách hoạt động của vòng lặp **while**. Bây giờ mình sẽ giới thiệu đến các bạn vòng lặp **do-while**, và so sánh sự giống và khác nhau giữa 2 cấu trúc vòng lặp này để các bạn có thể chọn cấu trúc lặp cho phù hợp với những vấn đề khác nhau.

Mình lấy lại ví dụ trong bài trước: *Sinh viên A đăng kí học môn lập trình C++ tại trường đại học, nếu sinh viên A không đủ điểm để qua môn học này thì sinh viên A sẽ phải học lại. Trong trường hợp sinh viên A phải học lại lần thứ 2, chúng ta lại nói rằng nếu sinh viên A không đủ điểm qua môn học này thì sinh viên A phải học lại... Vậy việc sinh viên A học lại là công việc sẽ được lặp đi lặp lại nhiều lần trong khi điều kiện sinh viên A đủ điểm để qua môn vẫn còn sai.*

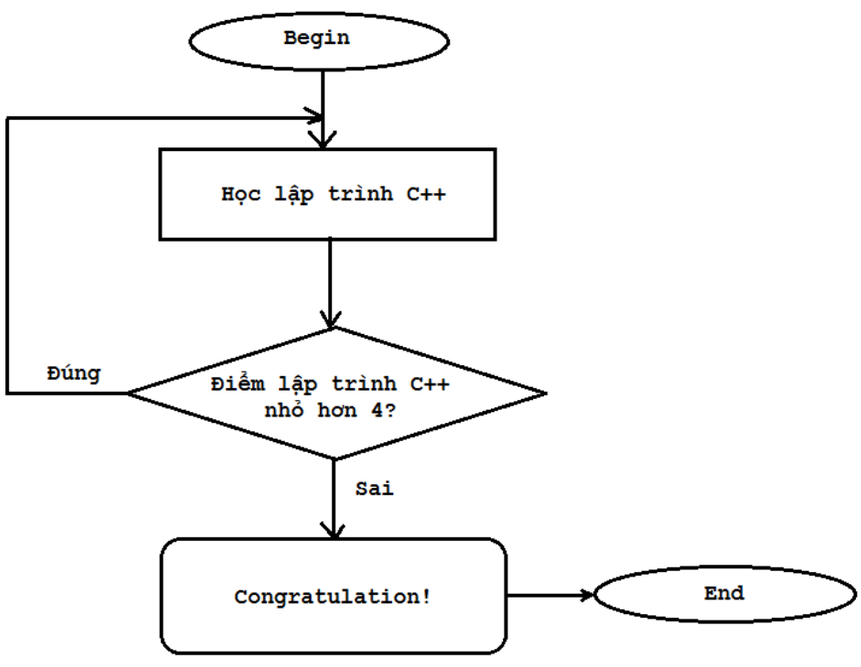
Khi sử dụng vòng lặp while để áp dụng cho trường hợp sinh viên A này, luồng thực hiện của chương trình sẽ diễn ra như sơ đồ khối sau:



Nhưng trong thực tế, có thể sinh viên A mới đăng kí học môn lập trình C++ lần đầu, lúc này sinh viên A chưa có điểm thi nhưng vẫn được đưa vào biểu thức điều kiện trong vòng lặp **while** để kiểm tra, như vậy vẫn giải quyết được bài toán nhưng chưa phù hợp lắm. Điều chúng ta mong muốn là sinh viên A phải thực hiện công việc "học lập trình C++" trước, sau đó chúng ta mới lấy điểm của sinh viên A để đánh giá và ra quyết định sinh viên A có phải học lại hay không.

Với cấu trúc thực hiện công việc trước và kiểm tra điều kiện sau, chúng ta nên sử dụng cấu trúc vòng lặp do-while.

Vậy sơ đồ khối của vòng lặp do-while dùng để biểu diễn bài toán của sinh viên A sẽ là:



Nhìn vào sơ đồ, chúng ta thấy sinh viên A phải học lập trình C++ ít nhất 1 lần, sau đó đưa điểm số của sinh viên A vào biểu thức điều kiện để đánh giá và quyết định sinh viên A có bị học lại hay không. Cấu trúc vòng lặp **do-while** áp dụng vào bài toán này phù hợp hơn cấu trúc vòng lặp **while**.

---

do-while statements

do-while statement là cấu trúc vòng lặp thứ 2 mình muốn giới thiệu đến các bạn:

```
do
{
        statements;
} while (expression);
```

Các câu lệnh bên trong khối lệnh của cấu trúc **do-while** sẽ được thực thi ít nhất 1 lần. Sau khi thực thi các câu lệnh, vòng lặp **do-while** sẽ đánh giá biểu thức điều kiện. Nếu biểu thức điều kiện đúng, chương trình quay trở lại thực hiện khối công việc của vòng lặp **do-while**, ngược lại, nếu biểu thức điều kiện sai, chương trình thoát khỏi vòng lặp **do-while**.

Lưu ý: vòng lặp do-while kết thúc bằng dấu chấm phẩy.

Ví dụ về vòng lặp **do-while**:

```
int main()
{
        int selection;
        do
        {
                cout << "_____Please make a selection_____" << endl;
                cout << "1/ Addition" << endl;
                cout << "2/ Subtraction" << endl;
                cout << "3/ Multiplication" << endl;
                cout << "4/ Division" << endl;

                cout << "Your selection: ";
                cin >> selection;

                cout << "Do something with your selection here" << endl;

        }while (selection >= 1 && selection <= 4);

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

Có một điều đáng chú ý trong vòng lặp **do-while** là biến vòng lặp dùng cho biểu thức điều kiện cần được khai báo trước vòng lặp **do-while**. Vì từ khóa while được đặt bên ngoài khối lệnh của vòng lặp nên những biến khai báo bên trong khối lệnh sẽ bị hủy trước khi đến biểu thức điều kiện.

Trong chương trình trên, vòng lặp sẽ dừng lại khi các bạn lựa chọn giá trị không nằm trong khoảng **[1, 4]**. Lựa chọn giá trị nằm ngoài khoảng **[1, 4]** sẽ khiến biểu thức điều kiện sai.

Tổng kết

Khi sử dụng vòng lặp do-while các bạn chỉ cần lưu ý rằng các câu lệnh bên trong vòng lặp này sẽ được thực hiện trước khi kiểm tra biểu thức điều kiện, còn lại nó hoạt động hoàn toàn tương tự vòng lặp while mà mình đã trình bày ở bài học trước.

Bài tập cơ bản

Giả sử userID và password của chương trình được định nghĩa như bên dưới

```
#include <iostream>
using namespace std;

const int ID = 123;
const int password = 123456;

int main()
{
        //.................}
```

Viết tiếp chương trình trên sử dụng vòng lặp do-while để kiểm tra userID và password được nhập từ bàn phím. Chương trình chỉ thực hiện tiếp khi người dùng nhập đúng userID và password. Nếu nhập sai, chương trình sẽ yêu cầu người dùng nhập lại.

3.2 Vòng lặp for

Trong các bài học trước, chúng ta đã cùng nhau tìm hiểu các cấu trúc điều khiển chương trình, trong đó có 2 bài học mình đề cập đến cấu trúc vòng lặp **while** và **do-while**. Hai cấu trúc lặp này tuy có khác nhau, nhưng chúng đều được sử dụng khi chưa biết được số lần lặp lại công việc tại thời điểm **run-time**.

Trong bài học này, mình sẽ giới thiệu đến các bạn vòng lặp **for** (**for loops**), cũng là vòng lặp cơ bản cuối cùng trong ngôn ngữ lập trình C++.

Một số đặc điểm của vòng lặp for:

- Vòng lặp for có cú pháp phức tạp hơn, nhưng ngắn gọn hơn các vòng lặp **while** hay **do-while** khi sử dụng.
- Vòng lặp **for** hoàn toàn có thể thay thế vòng lặp **while**.
- Vòng lặp for thường được sử dụng cho các trường hợp biết trước số lần lặp lại khối công việc.

Cú pháp vòng lặp for

```
for (variable_initialization; condition; variable_update)
{
        statements;
}
```

Mình lấy 1 ví dụ trước khi giải thích các thành phần của vòng lặp **for**:

```
for (int count = 1; count <= 10; count++)
{
        cout << "count = " << count << endl;
}
```

Vòng lặp for được định nghĩa bởi từ khóa **for** và được chia làm 3 phần chính, mỗi phần được ngăn cách bởi dấu chấm phẩy:

- **Variable initialization (phần khởi tạo biến)**

  Khác với vòng lặp while và do-while, biến vòng lặp có thể được khai báo và khởi tạo giá trị ngay bên trong phần khởi tạo của vòng lặp **for**. Như ở ví dụ trên, biến count được khai báo và khởi tạo với giá trị 1.

  Phần khởi tạo biến được thực thi đầu tiên và chỉ thực thi 1 lần duy nhất trong vòng lặp **for**.

- **Condition (biểu thức điều kiện)**

  Phần này tương tự như vòng lặp **while**, khối lệnh của vòng lặp for sẽ được thực hiện nếu biểu thức điều kiện cho giá trị đúng. Vòng lặp **for** kiểm tra biểu thức điều kiện trước khi thực hiện khối lệnh.

- **Variable update (cập nhật biến vòng lặp)**

  Phần này sẽ được thực thi cuối mỗi lần lặp, sau khi khối lệnh của vòng lặp **for** được thực thi. Phần này thường chịu trách nhiệm thay đổi giá trị biến vòng lặp được sử dụng trong biểu thức điều kiện (nhằm tránh tình trạng lặp vô hạn). Sau khi thực thi xong phần cập nhật biến vòng lặp, chương trình quay trở lại đánh giá biểu thức điều kiện của vòng lặp **for** và cứ như thế.

Vậy chúng ta rút ra được các bước thực hiện vòng lặp **for** như sau:

```
initialize loop variables --> check condition expression --> execute statements --> update loop variables.
for (int count = 1; count <= 10; count++)
{
        cout << count << " ";
}
```

Ví dụ trên có thể được chuyển về dưới dạng vòng lặp while như sau:

```
int count = 1; //variable initialization
```

```
while (count <= 10) //condition
{
        cout << count << " "; //statements

        count++; //variable update
}
```

Nhìn có vẻ dài dòng hơn vòng lặp for, nhưng vẫn có đủ 3 thành phần: **variable initialization, condition và variable update.**

Những lập trình viên mới học đến vòng lặp **for** sẽ cảm thấy khó đọc hơn vòng lặp **while**. Tuy nhiên, khi sử dụng thành thạo, vòng lặp **for** có nhiều tiện ích hơn.

Một số ví dụ về vòng lặp for

Dưới đây là một ví dụ sử dụng vòng lặp for để in ra tất cả các số chẵn từ 0 đến 10. Chúng ta đã biết trước rằng biến vòng lặp sẽ đi từ 0 đến 10, nên việc sử dụng vòng lặp for là phù hợp.

```
for (int i = 0; i <= 10; i++)
{
        if (i % 2 == 0)
                cout << i << " ";
}
```

Mình vừa sử dụng vòng lặp **for** để cho biến i tăng giá trị từ 1 đến 10, cứ mỗi lần lặp, mình kiểm tra giá trị hiện tại của biến i, nếu giá trị hiện tại của i chẵn, mình thực hiện in biến i ra màn hình.

Vòng lặp này có thể được rút gọn lại như sau:

```
for (int i = 0; i <= 10; i += 2)
{
        cout << i << " ";
}
```

Chúng ta biết rằng số chẵn tiếp theo sẽ cách số chẵn trước đó 2 đơn vị, do đó, mình thực hiện cộng thêm 2 đơn vị cho biến i tại phần **variable update**. Nhờ đó, mình không cần thực hiện kiểm tra giá trị của biến i trong vòng lặp nữa.

Bây giờ, thay vì chúng ta thực hiện lặp từ 0 đến 10, chúng ta có thể đi ngược lại từ 10 về 0 như sau:

```
for (int i = 10; i >= 0; i -= 2)
{
        cout << i << " ";
}
```

Kết quả in ra sẽ là:

```
10 8 6 4 2 0
```

Multiple declarations

Trong một số trường hợp, vòng lặp của chúng ta cần sử dụng đồng thời nhiều biến khác nhau. Ngôn ngữ C++ hỗ trợ cho chúng ta khai báo và khởi tạo nhiều biến bên trong phần **variable initialization** của vòng lặp **for**.

```
for (int hh = 0, mm = 0, ss = 0 ; true; ss++)
{
        if (ss >= 60)
        {
                ss = 0;
                mm++;
                if (mm >= 60)
                {
                        hh++;
                        if (hh >= 24)
                        {
                                hh = 0;
                        }
                }
        }

        cout << hh << ":" << mm << ":" << ss << endl;

        _sleep(1000);
        system("cls");
}
```

Nếu các bạn đặt khai báo biến vòng lặp tại phần **variable initialization** của vòng lặp **for**, những biến này phải có cùng kiểu dữ liệu.

Lược bỏ một số thành phần trong vòng lặp for

Một đặc điểm nổi bật hơn so với các vòng lặp khác là vòng lặp for cho phép lập trình viên lược bỏ các thành phần nếu không cần sử dụng. Ví dụ:

```
int loop = 0
for ( ; loop <= 10; )
{
        cout << loop++ << " ";
}
```

Trong ví dụ trên, mình không cần sử dụng tới thành phần khởi tạo biến, cũng như thành phần cập nhật giá trị biến. Lúc này, vòng lặp này hoàn toàn giống với vòng lặp **while**.

Chúng ta có thể lược bỏ luôn cả 3 thành phần cơ bản của vòng lặp for:

```
for ( ; ; )
{
        // do something
}
```

Khi biểu thức điều kiện trong for được bỏ trống, nó đồng nghĩa với việc biểu thức điều kiện luôn luôn đúng. Vòng lặp for này tương đương:

```
while (true)
{
        // do something
}
```

Nesting for loops

Tương tự như vòng lặp **while** hay **do-while**, vòng lặp **for** có thể chứa nhiều vòng lặp khác trong khối lệnh của nó.

```
for (int i = 0; i < 5; i++)
{
        for(int j = 0; j < 10; j++)
        {
                cout << "* ";
        }
        cout << endl;
}
```

Kết quả của đoạn code này là:

```
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
```

Tổng kết

Vòng lặp **for** được sử dụng phần lớn trong các cấu trúc lặp trong ngôn ngữ C++ mà mình đã giới thiệu đến các bạn. Vòng lặp **for** phù hợp cho cả trường hợp biết trước số lần lặp lẫn không biết trước số lần lặp.

Bài tập cơ bản

1/ Viết chương trình tính giai thừa của một số nguyên n nhập từ bàn phím.

2/ Viết chương trình tính dân số của một thành phố sau 10 năm nữa, biết rằng dân số hiện tại là 500.000 người, và tỉ lệ tăng dân số hằng năm của thành phố này là 1.6%.

3/ Viết chương trình in ra bảng cửu chương.

3.3 từ khóa break và continue

Trong bài học này, chúng ta sẽ cùng nhau tìm hiểu về cấu trúc điều khiển **Jump** mà mình đã giới thiệu sơ lược đến các bạn. Sau bài học này, các bạn sẽ biết cách sử dụng từ khóa **break, continue** trong cấu trúc vòng lặp hoặc cấu trúc rẽ nhánh với **switch case statement**.

Break

Từ khóa **break** được dùng để kết thúc vòng lặp, hoặc cấu trúc **switch**.

Break a switch

Khi sử dụng trong switch case statement, từ khóa break thường được đặt tại cuối mỗi khối lệnh mỗi nhãn case.

```
switch (character)
{
        case '+':
                cout << "addition" << endl;
                break;
        case '-':
                cout << "subtraction" << endl;
                break;
        case '*':
                cout << "multiplication" << endl;
                break;
        case '/':
                cout << "division" << endl;
                break;
       default:
               break;
}
```

Break a loop

Khi sử dụng trong các dạng vòng lặp khác nhau, từ khóa break đều có cùng mục đích là kết thúc sớm quá trình thực thi của vòng lặp.

```
for (int i = 10; i >= 0; i--)
{
        cout << "Count down: " << i << endl;
        if (i <= 5)
                break;
}
```

Đoạn chương trình này thực hiện đếm ngược từ 10 về 0, nhưng khi đếm về 5 thì biểu thức điều kiện của lệnh **if** bên trong vòng lặp **for** đúng, nên chương trình sẽ **break** vòng lặp for.

Từ khóa break thường được dùng để dừng vòng lặp vô hạn:

```
bool running = true;
while (true)
{
        // do something on running variable

        if(!running)
                break;
}
```

Continue

Từ khóa **continue** thường được sử dụng trong vòng lặp **for** để chuyển đến bước cuối cùng trong 1 lần lặp (**update variable**). Cùng nhìn lại 4 bước thực thi cơ bản của 1 lần lặp của vòng lặp **for**:

(1) Initialize loop variables

(2) Check condition expression

(3) Execute the statements

(4) Update variables

Mỗi khi bắt gặp từ khóa continue trong bước (3), chương trình sẽ bỏ qua phần còn lại của bước (3) để chuyển đến thực hiện bước (4), và bắt đầu 1 lần lặp mới từ bước (2).

```
for (int i = 0; i <= 20; i++)
{
        if (i % 5 == 0)
                continue;

        cout << i << " ";
}
```

Đoạn chương trình này sẽ in ra tất cả các số nguyên từ 0 đến 20, ngoại trừ các số chia hết cho 5 như 0, 5, 10, 15.

Từ khóa **continue** ít được sử dụng trong vòng lặp **while** hoặc **do-while** bởi một số hạn chế của nó.

```
int i = 1; // (1)
while (i <= 20) // (2)
{
        if (i % 5 == 0) // (3)
                continue;

        cout << i << " ";
        i++; // (4)
}
```

Cũng với ví dụ mẫu trên, nhưng được chuyển sang sử dụng cấu trúc của vòng lặp **while**. Như các bạn thấy, vòng lặp này vẫn thực hiện đủ 4 bước như vòng lặp **for** ở trên. Nhưng kết quả chỉ in ra được:

```
1 2 3 4
```

Và sau đó, vòng lặp này lặp vô hạn vì biến i sẽ không bao giờ được tăng lên nữa sau khi câu lệnh if được thực thi, vì bước (4) đặt trong vòng lặp **while** được coi như thuộc bước (3) của vòng lặp **for**. Do đó, khi gặp từ khóa **continue**, bước 4 của vòng lặp **while** cũng bị bỏ qua luôn, mà dòng lệnh `i++` sẽ không bao giờ được thực thi nữa.
Vì thế mà chúng ta thường xuyên sử dụng vòng lặp **for** hơn, và cũng thường đặt từ khóa **continue** trong vòng lặp **for** hơn.

Tổng kết

break và **continue** là 2 từ khóa tiêu biểu cho cấu trúc điều khiển Jump mà ngôn ngữ C++ cung cấp. Các bạn nên tránh lạm dụng 2 từ khóa này vì nó dễ gây sai sót cho chương trình.

PHẦN 4: NÂNG CAO VỀ BIẾN & KIỂU DỮ LIỆU

4.0 Sử dụng thư viện cstdint

Chào các bạn! Trong chương này, mình muốn giới thiệu đến các bạn một số kiểu dữ liệu khác với những gì các bạn đã được học và sử dụng trong các bài học trước, ngoài ra còn có thêm một số từ khóa về kiểu dữ liệu thường dùng trong chuẩn **C++11**.

Trong bài học này, chúng ta sẽ tìm hiểu cách sử dụng thư viện **cstdint**, một thư viện được chuẩn **C++99** định nghĩa và chúng có thể được sử dụng trên nhiều nền tảng. Nhưng trước hết, chúng ta cùng nhìn lại những kiểu dữ liệu số nguyên mà chúng ta từng biết.

```
signed short            _short;         //2 bytes: -32768 to 32767
unsigned short          _u_short;               //2 bytes: 0 to 65535
signed int              _int;                   //4 bytes: -2147483648 to 2147483647
unsigned int            _u_int;         //4 bytes: 0 to 4294967295
signed long long        _long_long;     //8 bytes: -9223372036854775808 to 9223372036854775807
unsigned long long      _u_long_long;   //8 bytes: 0 to 18446744073709551615
```

*Lưu ý: Khi khai báo biến không có từ khóa **unsigned** đứng trước, compiler sẽ mặc định đặt từ khóa **signed** trước kiểu dữ liệu. (kiểu **signed** có nhận giá trị âm, kiểu **unsigned** chỉ nhận giá trị từ 0 trở lên)*

Khi làm việc với một số hệ thống cũ, kiểu dữ liệu **int** và **unsigned int** chỉ có kích thước 2 bytes chứ không phải 4 bytes như Visual studio 2015 đã định nghĩa.

Các bạn đã nhận ra những rắc rối mà chúng ta sẽ gặp phải khi làm việc với kiểu số nguyên rồi phải không? Trong khi chúng ta cần phải nhớ rất nhiều thứ trong ngôn ngữ lập trình C++, chúng ta bây giờ còn phải nhớ cả kích thước của từng kiểu dữ liệu số nguyên khi chúng được định nghĩa với những cái tên hoàn toàn khác nhau (**short**, **int**, **long**, ...).

Đó chính là lý do thư viện **cstdint** xuất hiện, nó đảm bảo rằng biến (variable) kiểu số nguyên mà bạn chọn chiếm cùng kích thước vùng nhớ trên mọi kiến trúc hệ thống.

Fixed-width integers

Trong thư viện này, chúng ta hiện tại chỉ quan tâm đến một số kiểu dữ liệu số nguyên thông dụng sau:

```
//Name           //Type                          //Range
int8_t           1 byte signed                   -128 to 127
uint8_t          1 byte unsigned                 0 to 255
int16_t          2 bytes signed                  -32768 to 32767
uint16_t 2 bytes unsigned 0 to 65535
int32_t          4 bytes signed                  -2147483648 to 2147483647
uint32_t 4 bytes unsigned 0 to 4294967295
int64_t          8 bytes signed                  -9223372036854775808 to 9223372036854775807
uint64_t 8 bytes unsigned 0 to 18446744073709551615
```

int8_t là kiểu dữ liệu số nguyên có dấu với độ lớn 8 bits (bit là đơn vị lưu trữ nhỏ nhất trong máy tính), và 1 byte độ lớn vùng nhớ máy tính tương đương với 8 bits. Tương tự cho các kiểu dữ liệu khác như **int64_t** là kiểu số nguyên 8 bytes (64 bits bằng 8 bytes).

Khi sử dụng cách khai báo kiểu dữ liệu này, chúng ta biết chính xác kích thước vùng nhớ mà chúng ta sử dụng là bao nhiêu, từ đó dễ dàng suy luận ra phạm vi giá trị mà biến có thể chứa hơn.

Để sử dụng những cách khai báo kiểu dữ liệu này, các bạn chỉ cần đơn giản include thư viện **cstdint** tại phần khai báo thư viện sử dụng. Ví dụ:

```
#include <iostream>
#include <cstdint>
using namespace std;

int main()	{

	int32_t var(5);
	int another_integer(var);

	if(var == another_integer)	{
		cout << "(var == another_variable) is true" << endl;
	}

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

Như các bạn thấy ở ví dụ trên, hai biến **var** và **another_integer** tuy khác cách khai báo nhưng chúng có cùng kích thước bộ nhớ và cùng lưu trữ giá trị số nguyên, nên mình hoàn toàn có thể sử dụng hai biến này với mục đích giống nhau. Điều khác biệt duy nhất là khi nhìn vào biến **var**, ta dễ dàng nhận biết kích thước vùng nhớ mà biến đó đang chiếm giữ hơn.

Macros

Thư viện **cstdint** định nghĩa cho chúng ta một số **macro** để chúng ta dễ dàng lấy giá trị cực đại (max value) hoặc cực tiểu (min value) của mỗi kiểu dữ liệu số nguyên chúng ta vừa học ở trên.

Macro là gì? Nó là một cái tên mà lập trình viên đề ra. Bất cứ khi nào cái tên đó được sử dụng trong chương trình, nó thay thế bằng nội dung mà **macro** đó định nghĩa. Để định nghĩa một macro, các bạn sử dụng chỉ thị `#define` như sau:

```
#define MY_VALUE 100
```

Sau đó, khi compiler bắt gặp các bạn sử dụng cái tên MY_VALUE, nó sẽ được thay thế bằng giá trị 100.

```
uint16_t value = MY_VALUE; // value = 100
cout << MY_VALUE + pow(MY_VALUE, 2) << endl; // 100 + pow(100, 2)
```

Chúng ta còn nhiều điều để nói về **Macro** trong C++, nhưng trong bài này mình chỉ nói cách định nghĩa **Macro** tương tự việc khai báo một hằng số (**const**) như trên để phục vụ cho bài học này.

Trong thư viện **cstdint** chúng ta có thể sử dụng các **Macro** sau:

Signed integers: minimum value

```
INT8_MIN - Giá trị cực tiểu (minimum value) của kiểu int8_t
INT16_MIN - Giá trị cực tiểu (minimum value) của kiểu int16_t
INT32_MIN - Giá trị cực tiểu (minimum value) của kiểu int32_t
INT64_MIN - Giá trị cực tiểu (minimum value) của kiểu int64_t
```

Các bạn hãy thử chạy những dòng lệnh này trên Visual studio 2015 và xem lại những giá trị **MIN** trong bảng ở phần trên.

```
cout << "Minimum value of uint8_t = " <<  INT8_MIN << endl;
cout << "Minimum value of uint16_t = " <<  INT16_MIN << endl;
cout << "Minimum value of uint32_t = " <<  INT32_MIN << endl;
cout << "Minimum value of uint64_t = " <<  INT64_MIN << endl;
```

Signed integers: maximum value

```
INT8_MAX - Giá trị cực đại (maximum value) của kiểu int8_t
INT16_MAX - Giá trị cực đại (maximum value) của kiểu int16_t
INT32_MAX - Giá trị cực đại (maximum value) của kiểu int32_t
INT64_MAX - Giá trị cực đại (maximum value) của kiểu int64_t
```

Cùng thử in ra giá trị của những Macro này xem thế nào.

```
cout << "Maximum value of uint8_t = " <<  INT8_MAX << endl;
cout << "Maximum value of uint16_t = " <<  INT16_MAX << endl;
cout << "Maximum value of uint32_t = " <<  INT32_MAX << endl;
cout << "Maximum value of uint64_t = " <<  INT64_MAX << endl;
```

Unsigned integers: maximum value

Vì kiểu **unsigned integer** có giá trị nhỏ nhất là 0 nên người ta không cần phải định nghĩa những **Macro** cho minimum value mà chỉ cần quan tâm đến maximum value.

```
UINT8_MAX - Giá trị cực đại (maximum value) của kiểu uint8_t
UINT16_MAX - Giá trị cực đại (maximum value) của kiểu uint16_t
UINT32_MAX - Giá trị cực đại (maximum value) của kiểu uint32_t
UINT64_MAX - Giá trị cực đại (maximum value) của kiểu uint64_t
```

Tương tự như trên, các bạn thử in ra giá trị của những **Macro** này.

Sử dụng thư viện cstdint

Bây giờ mình sẽ sử dụng thư viện **cstdint** để giải một bài toán đơn giản.

Yêu cầu: Nhập 5 giá trị số nguyên từ bàn phím, in ra màn hình kết quả lớn nhất và nhỏ nhất của 5 giá trị đó.

Sau khi đọc yêu cầu bài toán, chúng ta cần đưa ra giải pháp trước khi bắt tay vào viết mã.

Trước hết, chúng ta cần 2 biến kiểu số nguyên, một biến để lưu giá trị lớn nhất trong 5 số vừa nhập, một biến để lưu giá trị nhỏ nhất của 5 số đó.

```
int32_t min_value;
int32_t max_value;
```

Bây giờ, chúng ta tạm thời bỏ qua việc tìm giá trị lớn nhất và nhỏ nhất của 5 số khác nhau, mà cùng giải quyết một bài toán đơn giản hơn, đó là tìm giá trị lớn nhất và nhỏ nhất của 2 số nguyên **a** và **b**.

- Tìm giá trị lớn nhất trong hai số **a** và **b**:

Chúng ta chỉ cần thực hiện 1 phép so sánh kiểm tra xem giá trị của a có lớn hơn b hay không? Nếu **(a lớn hơn b)** là **đúng**, giá trị lớn nhất là **a**, ngược lại, giá trị lớn nhất là **b**.

```
if(a > b)
        cout << a << endl;
else
        cout << b << endl;
```

- Tìm giá trị nhỏ nhất trong hai số **a** và **b**:

Chúng ta làm tương tự như trên, nhưng cần thay đổi một chút ở phần biểu thức so sánh. Nếu **(a bé hơn b)** là **đúng**, giá trị nhỏ nhất là **a**, ngược lại, giá trị nhỏ nhất là **b**.

Khá đơn giản phải không các bạn! Áp dụng lại cho bài toán tìm giá trị lớn nhất và nhỏ nhất từ 5 giá trị số nguyên mà người dùng nhập vào từ bàn phím:

- Tìm giá trị lớn nhất của 5 số khác nhau:

Như mình đã khai báo biến `max_value` ở trên, biến này sẽ lưu giá trị lớn nhất tại thời điểm ban đầu. Cứ mỗi lần so sánh với một giá trị trong 5 giá trị người dùng vừa nhập, nếu phát hiện giá trị nào lớn hơn giá trị max hiện tại, chúng ta gán lại giá trị `max_value` bằng giá trị của người dùng vừa nhập.

```
for(int8_t i = 1; i <= 5; i++)   {
        int32_t value;
        //nhập giá trị vào biến value tại đây

        //thực hiện so sánh giá trị max hiện tại với giá trị vừa nhập
        if(value > max_value) //nếu đúng thì thực hiện gán lại giá trị max mới
                max_value = value;
}

//in ra kết quả là giá trị lớn nhất của 5 số vừa nhập
```

- Tìm giá trị nhỏ nhất của 5 số khác nhau:

Đối với trường hợp này, chúng ta sử dụng biến `min_value` tương tự trường hợp tìm `max_value` nhưng đổi lại điều kiện so sánh một chút. Cứ mỗi lần so sánh với một giá trị trong 5 giá trị người dùng vừa nhập, nếu phát hiện giá trị nào nhỏ hơn giá trị `min_value` hiện tại, chúng ta gán lại giá trị `min_value` bằng giá trị người dùng vừa nhập

```
for(int8_t i = 1; i <= 5; i++)   {
        int32_t value;
        //nhập giá trị vào biến value tại đây

        //thực hiện so sánh giá trị max hiện tại với giá trị vừa nhập
        if(value < min_value) //nếu đúng thì thực hiện gán lại giá trị min mới
                min_value = value;
}

//in ra kết quả là giá trị nhỏ nhất của 5 số vừa nhập
```

Chúng ta còn bỏ sót một chi tiết vô cùng quan trọng! Giá trị ban đầu của `max_value` và `min_value` nên là bao nhiêu?
Đối với biến `max_value`, chúng ta cần một giá trị đảm bảo rằng người dùng sẽ không nhập vào số nguyên nào nhỏ hơn giá trị max ban đầu, và không thể vượt quá phạm vi lưu trữ giá trị của kiểu dữ liệu bạn chọn. Chúng ta không còn giá trị nào phù hợp hơn ngoài **INT32_MIN**.
Tương tự, giá trị ban đầu phù hợp nhất cho biến `min_value` là **INT32_MAX**.
(Vì mình chọn sử dụng biến kiểu *int32_t* để lưu trữ)

Cuối cùng, chúng ta có thể viết ra một chương trình tương đối hoàn thiện cho bài toán trên:

```
#include <iostream>
#include <cstdint>
using namespace std;

int main()
{
        int32_t min_value = INT32_MAX;
        int32_t max_value = INT32_MIN;
        const int8_t number_of_value = 5;

        for (int8_t i = 1; i <= 5; i++)  {
                int32_t current_value;
                cout << "Please enter an integer value: ";
                cin >> current_value;

                if (current_value < min_value)
                        min_value = current_value;

                if (current_value > max_value)
                        max_value = current_value;
        }

        cout << "Minimum value: " << min_value << endl;
        cout << "Maximum value: " << max_value << endl;
```

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

Tổng kết

- Sử dụng thư viện **cstdint** giúp các bạn kiểm soát tốt hơn kích thước vùng nhớ của kiểu dữ liệu số nguyên mà bạn khai báo cho biến, đồng thời cũng dễ dàng ước lượng phạm vi giá trị của biến cho phù hợp.
- Thư viện **cstdint** cũng được định nghĩa bên trong **namespace std**, vì thế khi sử dụng thư viện này, các bạn nên có thêm dòng khai báo `using namespace std;` để đảm bảo mọi thứ hoạt động bình thường.
- Visual studio 2015 hỗ trợ cho chúng ta một số cách biểu diễn kích thước kiểu integer mà không cần thêm vào thư viện **cstdint**:

```
__int8;
__int16;
__int32;
__int64;
```

Nếu các bạn sử dụng Visual studio thì nên sử dụng các cách khai báo kiểu **int** như trên.

4.1 Kiểu kí tự

Trong hầu hết tất cả các bài học trước đây, chúng ta chỉ làm việc cùng nhau trên kiểu dữ liệu số. Chúng ta sử dụng các biến lưu trữ giá trị số (số nguyên **int**, số thực như **float** hoặc **double**, ...) để phục vụ cho việc tính toán toán học, giải quyết các bài toán đơn giản là chủ yếu.

Trong bài học ngày hôm nay, chúng ta sẽ tìm hiểu một kiểu dữ liệu cũng là một trong những kiểu dữ liệu cơ bản trong ngôn ngữ C và C++, đó là **kiểu kí tự**.

Kiểu kí tự là gì?

Cũng tương tự như các kiểu dữ liệu số (**int32_t**, **float**, **uint64_t**, ...), kiểu kí tự là một kiểu dữ liệu có độ lớn **1 byte (8 bits)** dùng để lưu trữ 1 kí tự trong vùng nhớ máy tính. Kí tự có thể là các chữ cái đơn trong bảng chữ cái (a, b, c, ... x, y z), có thể là các kí hiệu toán học (+, -, *, /, ...), hay có thể là những con số (0, 1, 2, ..., 9)...

Một đặc điểm của kiểu kí tự là KHÔNG PHẢI MỌI KÝ TỰ đều có thể hiển thị được lên màn hình.

Trong C/C++, kiểu kí tự có thể lưu trữ **1 kí tự** trong bảng mã **ASCII**.

Đây là bảng mã kí tự **ASCII** đầy đủ:

Code	Symbol	Code	Symbol	Code	Symbol	Code	Symbol	Code	Symbol	Code	Symbol	Code	Symbol
0	NUL (null)	20	DC4 (data control 4)	40	(	60	<	80	P	100	d	120	x
1	SOH (start of header)	21	NAK (negative acknowledge)	41	)	61	=	81	Q	101	e	121	y
2	STX (start of text)	22	SYN (synchronous idle)	42	*	62	>	82	R	102	f	122	z
3	ETX (end of text)	23	ETB (end of transmission block)	43	+	63	?	83	S	103	g	123	{
4	EOT (end of transmission)	24	CAN (cancel)	44	,	64	@	84	T	104	h	124	|
5	ENQ (enquiry)	25	EM (end of medium)	45	-	65	A	85	U	105	i	125	}
6	ACK (acknowledge)	26	SUB (substitute)	46	.	66	B	86	V	106	j	126	~
7	BEL (bell)	27	ESC (escape)	47	/	67	C	87	W	107	k	127	DEL (delete)
8	BS (backspace)	28	FS (file separator)	48	0	68	D	88	X	108	l		
9	HT (horizontal tab)	29	GS (group separator)	49	1	69	E	89	Y	109	m		
10	LF (line feed/new line)	30	RS (record separator)	50	2	70	F	90	Z	110	n		
11	VT (vertical tab)	31	US (unit separator)	51	3	71	G	91	[	111	o		
12	FF (form feed / new page)	32	(space)	52	4	72	H	92	\	112	p		
13	CR (carriage return)	33	!	53	5	73	I	93	]	113	q		
14	SO (shift out)	34	"	54	6	74	J	94	^	114	r		
15	SI (shift in)	35	#	55	7	75	K	95	_	115	s		
16	DLE (data link escape)	36	$	56	8	76	L	96	`	116	t		
17	DC1 (data control 1)	37	%	57	9	77	M	97	a	117	u		
18	DC2 (data control 2)	38	&	58	:	78	N	98	b	118	v		
19	DC3 (data control 3)	39	'	59	;	79	O	99	c	119	w		

Bảng mã **ASCII** được chia làm 2 cột:

- Cột **Code** là số thứ tự của kí tự trong bảng mã **ASCII**.

- Cột **Symbol** là kí tự được chuyển đổi từ mã **Code** sang dạng có thể đọc được.

Khai báo biến kiểu kí tự như thế nào?

Để khai báo biến kiểu kí tự trong C/C++, ta dùng từ khóa **char** như sau:

```
char character; //declare a variable type char
char ch(65); //declare a variable type char and initialze with ASCII code
char a = 'a'; //declare a variable type char and initialize with a symbol of ASCII table
```

Cú pháp hoàn toàn giống việc thực hiện khai báo biến thông thường.

Biến kiểu kí tự (**char**) thực tế cũng là một kiểu số nguyên kích thước 1 byte (tương đương với **int8_t**), nó lưu trữ giá trị là mã **Code** của kí tự đó, nhưng khi hiển thị lên màn hình, nó cho ra kết quả là kí tự (**Symbol**) chứ không in ra mã ASCII của kí tự đó.

Vì thế, chúng ta có thể khởi tạo cho biến kiểu **char** bằng cách gán một kí tự đặt giữa cặp dấu nháy đơn. Ví dụ:

```
char ch = 'a';
ch = 'b';
```

hoặc cũng có thể gán trực tiếp mã Code của kí tự đó trong bảng mã ASCII. Ví dụ:

```
char ch = 97; //kí tự 'a' trong bảng mã ASCII có mã là 97
ch = 98; //kí tự 'b' trong bảng mã ASCII có mã là 98
```

In biến kiểu kí tự ra màn hình

Để in một kí tự ra màn hình, chúng ta có thể thực hiện bằng nhiều cách khác nhau:

- In trực tiếp một kí tự đặt trong cặp dấu nháy đơn:

```
cout << 'h' << 'e' << 'l' << 'l' << 'o' << endl;
```

Ở câu lệnh trên, mình sử dụng đối tượng **cout** để in ra một dãy nhiều kí tự đơn nối tiếp nhau. Sau khi chạy chương trình, dòng lệnh trên sẽ in ra màn hình dãy kí tự **hello**.

- Sử dụng biến kiểu **char** để lưu trữ một kí tự:

```
char h = 'h', e = 'e', l = 'l', o = 'o';
cout << h << e << l << l << o << endl;
```

Chúng ta có thể sử dụng lại nhiều lần 1 biến, nên mình chỉ cần khai báo 1 lần biến **l** để lưu trữ kí tự **'l'**. Câu lệnh trên cho kết quả hoàn toàn tương tự, dòng **hello** sẽ được in ra màn hình.

- In trực tiếp mã **Code** của kí tự trong bảng mã ASCII (nhưng ép về kiểu **char**):

```
cout << static_cast<char>(67) << static_cast<char>(43) << static_cast<char>(43) << endl;
```

Các bạn thử tra trong bảng mã **ASCII** xem thử hai số **67** và **43** đại diện cho 2 kí tự gì nhé, sau đó đoán xem kết quả in ra màn hình của dòng lệnh trên là gì.

- Chúng ta có thể in ra mã **Code** của 1 biến kí tự:

```
char ch = 'V';
cout << static_cast<int16_t>(ch) << endl;
```

Bằng cách ép kiểu của biến **ch** về kiểu số nguyên, chương trình sẽ in ra 1 con số là số thứ tự của kí tự đó trong bảng mã **ASCII**.

Như mình đã nói, kiểu kí tự (**char**) hoàn toàn là kiểu số nguyên (**int8_t**). Để in ra kí tự đại diện cho số nguyên đó, chúng ta cần định dạng cho nó là kiểu kí tự (**char**) thì compiler mới hiểu là chúng ta đang cần hiển thị kí tự chứ không phải con số.

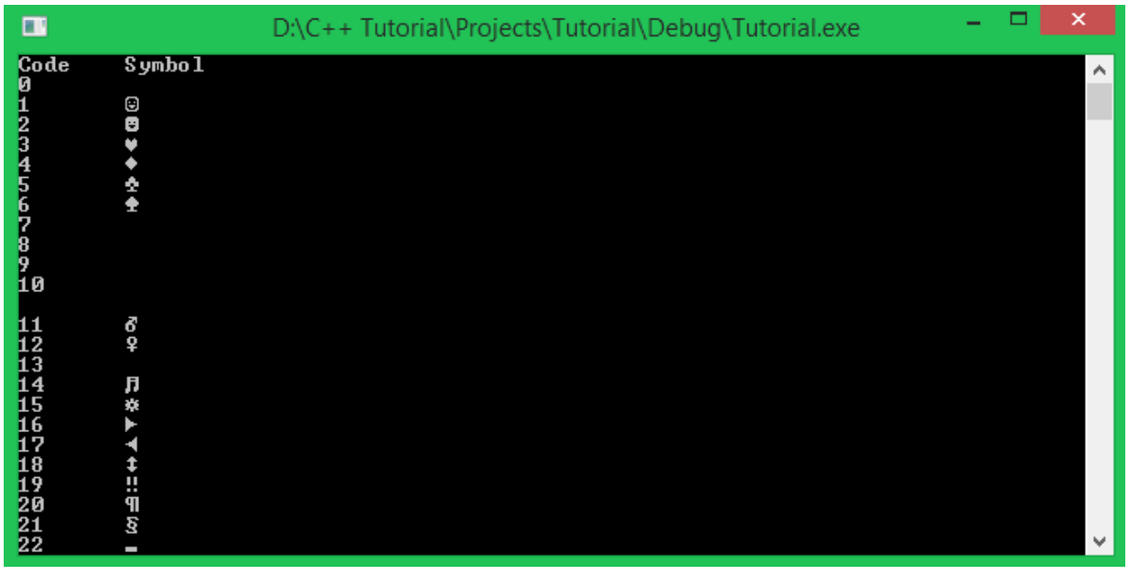
Bây giờ chúng ta thử in ra toàn bộ bảng mã *ASCII* trên màn hình dưới dạng

```
<Code>: <Symbol>
```

bằng cách sử dụng 1 vòng lặp **for** để in ra toàn bộ kí tự từ mã **0** đến mã **127**.

```
cout << "Code" << '\t' << "Symbol" << endl;
for(int16_t ascii_code = 0; ascii_code <= 127; ascii_code++)       {
        cout << ascii_code << '\t' << static_cast<char>(ascii_code) << endl;
}
```

Và kết quả in ra màn hình:



*Chắc các bạn còn nhớ kí tự đặc biệt **'\t'** tương đương với 1 Tab trên màn hình console. Nếu không nhớ thì cũng không sao, mình sẽ nhắc lại một chút ở phần bên dưới.*

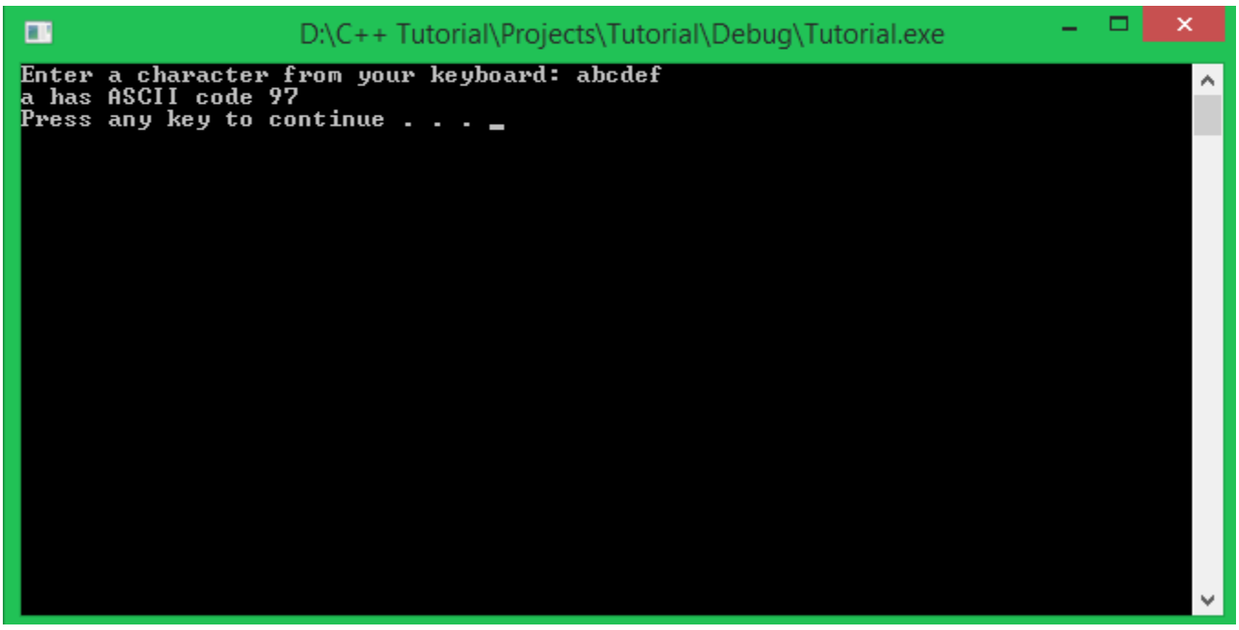
Có một số mã **Code** cho ra kí tự khoảng trắng vì đó là những kí tự đặc biệt, ví dụ mã **7** đại diện cho 1 tiếng **Beep**, nên nó không có kí tự để in ra.

Nhập giá trị cho kiểu kí tự (char) từ bàn phím

Nhập kí tự từ bàn phím cũng tương tự việc nhập giá trị số từ bàn phím để gán cho biến. Chúng ta có thể sử dụng đối tượng **cin** như cách chúng ta sử dụng với biến số nguyên, số thực ...

```
char ch;
cout << "Enter a character from your keyboard: ";
cin >> ch;
cout << ch << " has ASCII code " << static_cast<int16_t>(ch) << endl;
```

Các bạn cùng nhìn vào phần kết quả chương trình mình đã thực thi bên dưới:



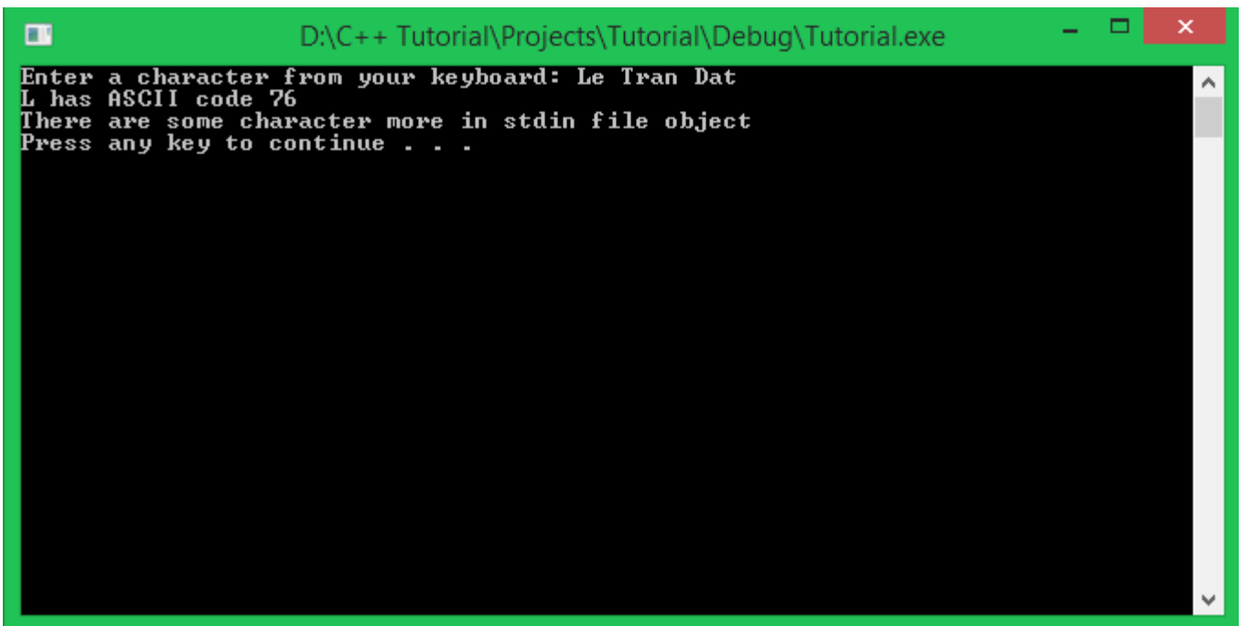
Mình không thực hiện nhập 1 kí tự từ bàn phím, thay vào đó, mình nhập nhiều kí tự liên tiếp nhau, và điều gì xảy ra? Biến **ch** (kiểu **char**) chỉ nhận vào 1 kí tự duy nhất là kí tự đầu tiên mà mình nhập vào.

Vậy thì những kí tự còn lại sẽ đi đâu? Nó vẫn còn được lưu trữ tạm thời bên trong đối tượng file **stdin**. Để kiểm chứng điều này, mình thêm một đoạn mã nhỏ sau khi in ra kí tự của biến **ch** trên màn hình:

```
char ch;
cout << "Enter a character from your keyboard: ";
cin >> ch;
cout << ch << " has ASCII code " << static_cast<int16_t>(ch) << endl;

//check if there is any character exist in stdin file object
if (!cin.eof())  {
        cout << "There are some character more in stdin file object" << endl;}
```

Nếu **cin.eof()** trả về giá trị là đúng, điều này có nghĩa chúng ta đã lấy hết kí tự trong đối tượng file **stdin** ra và đọc được kí tự kết thúc file (**EOF** = End of file). Vì thế, nếu điều này không xảy ra, tức là **!cin.eof()** là đúng, nghĩa là vẫn còn kí tự bên trong đối tượng file **stdin**.

Mình sẽ chạy lại chương trình với đoạn mã mà mình vừa thêm vào để các bạn cùng xem kết quả:

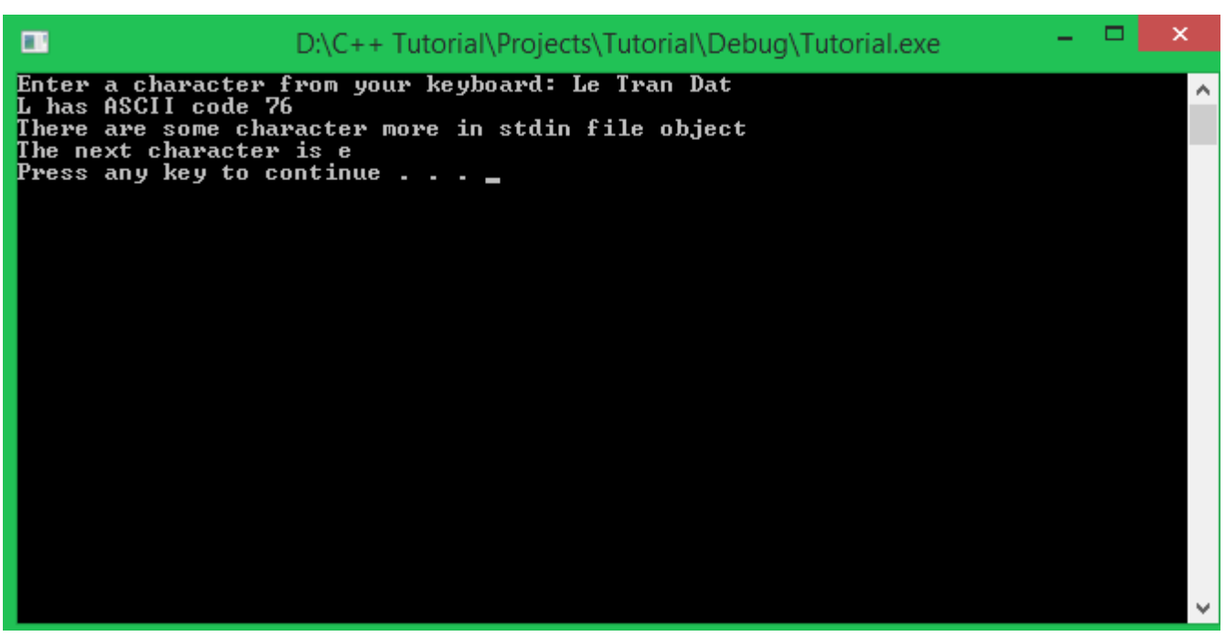


Mình nhập vào "**Le Tran Dat**" và biến ch (kiểu **char**) nhận vào kí tự đầu tiên (kí tự **'L'**), chương trình thông báo tiếp vẫn còn kí tự tồn tại bên trong đối tượng file **stdin**.

Vì thế, khi chúng ta tiếp tục thêm vào dòng lệnh nhập kí tự khác phía sau đoạn chương trình trên, nó sẽ không dừng lại chờ người dùng nhập kí tự nữa mà nó lấy luôn kí tự tiếp theo trong đối tượng file **stdin** để đưa vào biến. Các bạn cùng chạy đoạn mã lệnh sau để kiểm chứng kết quả:

```
char ch;
cout << "Enter a character from your keyboard: ";
cin >> ch;
cout << ch << " has ASCII code " << static_cast<int16_t>(ch) << endl;

//check if there is any character exist in stdin file object
if (!cin.eof())  {
        cout << "There are some character more in stdin file object" << endl;
}

//Continue reading a character from stdin file object
char next_ch;
cin >> next_ch;
cout << "The next character is " << next_ch << endl;
```

Và đây là những gì chương trình cho ra kết quả:



Biến **ch** nhận vào kí tự đầu tiên là **'L'**, biến **next_ch** lấy ngay kí tự **'e'** mà không đợi người dùng nhập thêm kí tự khác.

Sẽ có trường hợp chúng ta chỉ muốn nhận vào biến kí tự đầu tiên chúng ta nhập vào, những kí tự thừa phía sau có thể là do chúng ta nhấn nhầm phím nào đó, và chúng ta muốn loại bỏ những kí tự thừa đi để nhập lại kí tự khác cho biến char tiếp theo. Trong trường hợp này, có hai cách để thực hiện xóa toàn bộ dữ liệu đang tồn tại trong đối tượng file **stdin**:

- Sử dụng hàm `fflush(FILE *file)`:

Đây là một hàm được định nghĩa trong ngôn ngữ C, nhưng chúng ta hoàn toàn có thể sử dụng nó trong ngôn ngữ C++. Hàm `fflush` nhận vào một đối tượng file mà chúng ta muốn xóa dữ liệu bên trong nó (trong trường hợp này là đối tượng file **stdin**).

```
fflush(stdin); //Add this command line where you want to clear all data in stdin file object
```

- Sử dụng phương thức `ignore` của đối tượng **cin** để bỏ qua toàn bộ kí tự bên trong đối tượng file **stdin**:

Phương thức `ignore` này nhận vào 2 đối số là số kí tự nó sẽ bỏ qua, và kí tự khiến lệnh này dừng lại khi gặp phải trong đối tượng file **stdin**, ở đây mình sử dụng kí tự **'\n'** là kí tự được tạo ra khi nhấn phím **Enter**.

```
cin.ignore( INT64_MAX, '\n');
```

Sau khi sử dụng một trong hai cách trên, lần yêu cầu nhập dữ liệu từ bàn phím tiếp theo (thông qua đối tượng *cin*) sẽ phải thực hiện nhập lại từ đầu.

Một cách nhập dữ liệu khác cho kiểu kí tự (char)

Cũng sử dụng đối tượng **cin** thuộc thư viện **iostream**, nhưng không dùng toán tử **>>** mà sử dụng phương thức **get** của đối tượng **cin**. Có hai cách sử dụng phương thức **get** của đối tượng **cin** để nhập dữ liệu cho biến kiểu **char**, các bạn có thể nhớ 1 trong 2 cách mà bạn cảm thấy dễ hiểu nhất:

```
char ch;
ch = cin.get(); //get method return the character which you just entered
cin.get(ch); //put a char variable into the brackets
```

Cả 2 cách trên đều cho ra kết quả tương đương nhau.

Escape sequences

Trong C/C++ có một số kí tự có ý nghĩa đặc biệt, nó được gọi là **escape sequences**. Một **escape sequences** bắt đầu bằng một dấu **'\'** và theo sau là một kí tự hoặc con số.

Name	Symbol	Meaning
Alert	\a	Makes an alert, such as a beep
Backspace	\b	Moves the cursor back one space
Formfeed	\f	Moves the cursor to next logical page
Newline	\n	Moves cursor to next line
Carriage return	\r	Moves cursor to beginning of line
Horizontal tab	\t	Prints a horizontal tab
Vertical tab	\v	Prints a vertical tab
Single quote	\'	Prints a single quote
Double quote	\"	Prints a double quote
Backslash	\\	Prints a backslash
Question mark	\?	Prints a question mark
Octal number	\(number)	Translates into char represented by octal
Hex number	\x(number)	Translates into char represented by hex number

Ví dụ:

```
cout << "First line\nSecond line" << endl;
```

Dòng lệnh trên sẽ cho ra output là:

```
First line
Second line
```

Hay dòng lệnh dưới đây:

```
cout << "First part\tSecond part" << endl;
```

Sẽ cho chúng ta kết quả:

```
First part    Second part
```

Bạn chỉ có thể in ra kí tự nháy kép bằng cách thêm dấu backslash **'\'** trước kí tự nháy kép trong dãy kí tự bạn muốn in ra.

```
cout << "This is \"quote\" text" << endl;
```

Có một số bạn thắc mắc rằng, kí tự xuống dòng '\n' và đối tượng endl trong thư viện iostream khác nhau như thế nào?

Khi sử dụng **std::endl** (sử dụng toán tử phạm vi để truy cập vào đối tượng **endl** bên trong namespace **std**), output sẽ được đẩy vào vùng bộ nhớ đệm, đối tượng **cout** có thể không chuyển text trực tiếp đến thiết bị đầu ra ngay lập tức.

Cả kí tự '\n' và đối tượng **endl** đều chuyển con trỏ đến vị trí đầu dòng tiếp theo, thêm vào đó, đối tượng **endl** đảm bảo thứ tự trên thiết bị đầu ra đúng với lúc nhập dữ liệu từ đầu vào.

```
cout << endl;
```
tương đương với
```
cout << '\n' << std::fflush;
```
Sử dụng đối tượng **std::endl** sẽ làm sạch luôn stream, trong khi đó, sử dụng kí tự **'\n'** chỉ đơn giản là đưa kí tự xuống dòng lên màn hình.

Câu trả lời ngắn gọn cho việc khi nào sử dụng **std::endl** và **'\n'** là:

- Sử dụng **std::endl** khi bạn cần đảm bảo cho ra kết quả ngay lập tức, cụ thể khi làm việc trên các thiết bị đầu ra chậm.
- Sử dụng **'\n'** cho các trường hợp còn lại.

Sự khác nhau khi đặt kí tự bên trong cặp dấu nháy đơn và cặp dấu nháy kép là gì?

Như đã học trong bài này, một biến kí tự (**char**) chỉ được dùng để đặc tả 1 kí tự trong bảng mã **ASCII**, và chúng ta luôn đặt 1 kí tự đơn bên trong 1 cặp dấu nháy đơn.

```
char ch('65');
ch = 'a';
```
Những kí tự được đặt bên trong cặp dấu nháy kép được gọi là chuỗi kí tự (**string**). Một **string** là một tập hợp các kí tự nối tiếp nhau. Ví dụ:

```
cout << "Hello everyone!" << endl; //Hello everyone is a string
```
Tất nhiên làm việc với chuỗi kí tự (**string**) sẽ phức tạp hơn, nên các bạn sẽ được học nó trong các bài học sau.

Do stupid thing with char type

Trước khi kết thúc bài học này, mình sẽ hướng dẫn các bạn làm một cái gì đó với kiểu kí tự (char) mà các bạn đã được học.

Mình muốn thực hiện nhập họ và tên của mình (Viết không dấu do bảng mã ASCII bị giới hạn) từ bàn phím. Xóa màn hình console đi và in ra lại họ tên mà mình vừa nhập từ bàn phím, nhưng in ra lần lượt từng kí tự, mỗi lần in kí tự sẽ tạm dừng trong một khoảng thời gian ngắn.

Để thực hiện được yêu cầu này, mình sẽ cung cấp cho các bạn một số chức năng cần thiết:

- `system("cls");`

Hàm này gọi đến lệnh `cls`, thực hiện xóa dữ liệu đã in ra trên console.

- `Sleep(DWORD miliseconds);`

Hàm này sẽ tạm dừng mọi công việc thực hiện trên console trong một khoảng thời gian **miliseconds** mà bạn truyền vào. Để sử dụng hàm này cần thêm thư viện **windows.h** tại phần khai báo thư viện.

Các bạn chưa được học cách để lưu trữ biến là một chuỗi các kí tự liên tiếp nhau, nên việc lưu trữ dãy kí tự tên của bạn bên trong biến là rất khó khăn. Chúng ta chỉ mới biết đến cách lưu trữ 1 kí tự bên trong 1 biến kiểu **char**.

Nhưng thử nhớ lại những điều mình đã nói, khi thực hiện nhập kí tự từ bàn phím mà bạn nhập thừa kí tự thì điều gì xảy ra? Những kí tự thừa vẫn còn lưu trữ bên trong đối tượng file **stdin**, vì thế, chúng ta chỉ cần lấy những kí tự đó ra 1 lần nữa thông qua đối tượng **cin**.

Ban đầu, chúng ta yêu cầu người dùng nhập tên đầy đủ của mình vào:

```
char ch;
cout << "Enter your full name: ";
cin >> ch;
```

Kí tự đầu tiên mà bạn nhập sẽ lưu vào biến **ch**, những kí tự còn lại vẫn lưu trong đối tượng file **stdin**.

Tiếp theo, chúng ta thực hiện xóa màn hình console:

```
system("cls");
```

Công việc còn lại, chúng ta lấy lần lượt từng kí tự vẫn được lưu trong đối tượng file **stdin** cho đến khi gặp kí tự xuống dòng **'\n'** hoặc kí tự kết thúc file **EOF**. Các bạn nhớ phải in kí tự đã lưu trong biến **ch** ra trước rồi mới đọc tiếp vào nhé.

```
do
{
        Sleep(50); //Pause the program for 50 miliseconds
        cout << ch;
        ch = cin.get();
} while (ch != '\n' && ch != EOF);
```

Vòng lặp trên sẽ dừng khi biến **ch** nhận được kí tự xuống dòng **'\n'** (lúc bạn nhấn Enter để kết thúc nhập) hoặc kí tự kết thúc file **EOF**.

Kết hợp những phần trên thành một chương trình hoàn chỉnh:

```
#include <iostream>
#include <Windows.h>
using namespace std;

int main()
{
        char ch;
        cout << "Enter your full name: ";
        cin >> ch;

        system("cls");

        do
        {
                cout << ch;
                ch = cin.get();
                Sleep(50);
        } while (ch != '\n' && ch != EOF);
        cout << endl;

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

Các bạn thử chạy chương trình, nhập full-name của các bạn vào xem điều gì xảy ra nhé!

4.2 Ép kiểu dữ liệu

Các bạn đã làm việc với một số kiểu dữ liệu cơ bản trong ngôn ngữ C++ như kiểu số nguyên (**int**, **int32_t**, ...), kiểu số nguyên không dấu (**uint32_t**, **uint64_t**, ...), kiểu số thực (**float**, **double**, ...), hay kiểu kí tự (char).

Những kiểu dữ liệu này khi khai báo biến sẽ tạo ra những vùng nhớ trong máy tính để lưu trữ những giá trị có ý nghĩa. Những kiểu dữ liệu khác nhau có thể lưu trữ giá trị giống nhau, ví dụ giá trị **3** trong số nguyên cũng tương đương với giá trị **3.0** trong tập hợp số thực.

Trong một số trường hợp tính toán cụ thể hoặc cần biểu diễn giá trị dưới những định dạng khác nhau, chúng ta cần thực hiện **ép kiểu (casting)** để chuyển đổi qua lại những kiểu dữ liệu có khả năng lưu trữ giá trị giống nhau.

Trong ngôn ngữ C++, ép kiểu (**casting**) được chia làm 2 loại:

Ép kiểu ngầm định (implicit type conversion)

Ép kiểu ngầm định (**implicit type conversion**) được thực hiện bất cứ khi nào một kiểu dữ liệu cơ bản được sử dụng, nhưng giá trị được cung cấp thuộc kiểu dữ liệu cơ bản khác, và người dùng không nói cho **compiler** biết cách để thực hiện việc chuyển đổi này.

Chúng ta cùng xem qua ví dụ này:

```
float f_value = 3; //initialize float point value with integer 3
```

Trong trường hợp này, **compiler** không chỉ thực hiện copy giá trị 3 gán vào vùng nhớ mà biến **f_value** đang nắm giữ, mà còn thực hiện chuyển giá trị số nguyên 3 sang số thực 3.0f, sau đó, giá trị 3.0f mới được gán cho biến **f_value**.

Việc khởi tạo giá trị cho biến bằng giá trị của một biến có kiểu dữ liệu khác cũng đi kèm với việc thực hiện ép kiểu ngầm định.

```
float f_value = 10.0f;
double d_value(f_value);
```

Mặc dù **f_value** và **d_value** đều dùng để lưu trữ giá trị số thực, nhưng khi gặp 2 kiểu dữ liệu khác nhau, **compiler** vẫn thực hiện ép kiểu cho lệnh khởi tạo này.

Trong một số trường hợp cụ thể, **compiler** sẽ đưa ra cảnh báo về việc ép kiểu ngầm định có thể gây mất hoặc sai dữ liệu.

```
double d_value = DBL_MAX; // DBL_MAX is maximum value of double type
float f_value(d_value);
cout << f_value << endl;
```

Như các bạn đã biết, kiểu **double** có kích thước 8 bytes trong khi kiểu **float** chỉ có kích thước 4 bytes. Vì thế, giá trị mà biến kiểu **double** có thể vượt ngoài khả năng lưu trữ của biến kiểu **float**.

Điều này dẫn đến việc in giá trị biến **f_value** lên màn hình là không chính xác. Với việc gán giá trị thuộc kiểu dữ liệu có kích thước lớn hơn cho một biến có kích thước nhỏ hơn hoàn toàn có khả năng xảy ra mất dữ liệu nếu không kiểm soát được giá trị của nó.

Ép kiểu rõ ràng (explicit type conversion)

Ép kiểu rõ ràng (**explicit type conversion**) là việc chuyển đổi kiểu dữ liệu một cách rõ ràng bởi yêu cầu của lập trình viên.

Có 5 cách khác nhau trong việc ép kiểu rõ ràng:

- C-Style casts.
- Static casts.
- Const casts.
- Dynamic casts.
- Reinterpret casts.

Chúng ta cùng xét ví dụ sau:

```
int i_value1 = 10;
int i_value2 = 4;
float f_value = i_value1 / i_value2;
```

Biến **f_value** sẽ được gán giá trị là 2.0 vì phép chia hai số nguyên sẽ trả về kết quả là một giá trị số nguyên. Làm thế nào chúng ta có thể nói với **compiler** rằng chúng ta muốn có kết quả trả về là số thực?

C-style casts

Trong chuẩn ngôn ngữ C, **casting** được thực hiện thông qua toán tử **()** với tên của kiểu dữ liệu bạn muốn chuyển đổi về được đặt bên trong.

```
int i_value1 = 10;
int i_value2 = 4;
float f_value = (float)i_value1 / i_value2;
```

Trong đoạn chương trình trên, mình sử dụng **float cast** để nói với **compiler** đưa **i_value1** về kiểu **float**. Sau khi **i_value1** được ép về kiểu **float**, **i_value2** cũng được ép về kiểu **float** để thực hiện phép chia 2 số thực.

Ngôn ngữ C++ cho phép thực hiện **C-style cast** giống với cách gọi hàm:

```
f_value = float(i_value1) / i_value2;
```
Việc thực hiện **C-style cast** không được **compiler** kiểm tra tại thời điểm biên dịch chương trình, nên **compiler** sẽ không đưa ra những cảnh báo, điều này khiến các lập trình viên có thể làm những việc không có ý nghĩa.

Các bạn nên tránh sử dụng C-style cast.

Static casts

Ngôn ngữ C++ giới thiệu cho chúng ta 1 toán tử ép kiểu gọi là **static_cast**. Các bạn có thể đã gặp toán tử này (trong bài kiểu kí tự khi mình cần lấy mã **ASCII** của 1 kí tự).

```
char ch = 'A';
cout << static_cast<int16_t>(ch) << endl; //print 65, not 'A'
```
Ưu điểm của toán tử **static_cast** là nó yêu cầu **compiler** kiểm tra kiểu dữ liệu tại thời điểm biên dịch chương trình, hạn chế được những lỗi ngoài ý muốn.

```
int i_value1 = 10;
int i_value2 = 4;
float f_value = static_cast<float>(i_value1) / i_value2;
```

Tổng kết

Việc ép kiểu nên được hạn chế sử dụng, vì bất cứ khi nào thực hiện hành vi ép kiểu cũng tiềm ẩn khả năng xảy ra vấn đề với chương trình. Trong một số trường hợp cụ thể chúng ta bắt buộc phải sử dụng ép kiểu, nên sử dụng **static_cast** thay vì ép kiểu theo **C-Style**.

Trong phần hướng dẫn lập trình C++ cơ bản, mình chỉ hướng dẫn các bạn sử dụng toán tử **static_cast** để thực hiện chuyển đổi các kiểu dữ liệu cơ bản. Những cách ép kiểu rõ ràng khác (reinterpret_cast và dynamic_cast) mình sẽ giới thiệu đến các bạn trong những phần có liên quan về sau.

4.3 auto và decltype

Trong bài học hôm nay, chúng ta làm quen với một số khái niệm mới được cung cấp bởi chuẩn **C++11** khi làm việc về kiểu dữ liệu.

Nhưng trước hết, mình muốn giới thiệu với các bạn 1 thư viện trong Visual studio 2015, nó sẽ hỗ trợ cho chúng ta xem thông tin về kiểu dữ liệu của một đối tượng ta đang xem xét.

Thư viện typeinfo

Thư viện này định nghĩa 1 **class** (các bạn sẽ được học về class trong phần lập trình hướng đối tượng với C++) có tên là **type_info**, **class** này giữ thông tin về kiểu dữ liệu của đối tượng đang được xem xét.

Sử dụng thư viện **typeinfo** chúng ta có thể thực hiện phép so sánh **==** hoặc **!=** giữa hai đối tượng để kiểm tra chúng có cùng hay khác kiểu dữ liệu.

Chúng ta còn có thể lấy ra thông tin về kiểu dữ liệu của đối tượng thông qua toán tử **typeid**. Hoặc sử dụng phương thức **name** định nghĩa bên trong class **type_info** để lấy ra tên của kiểu dữ liệu của đối tượng.

```
namespace std    {
        ...
        class type_info;
        class bad_cast;
        class bad_typeid;
        ...
}
```
Vì class **type_info** được định nghĩa trong **namespace std** nên chúng ta cũng nên khai báo **using namespace std** trước khi sử dụng.

Một ví dụ về việc sử dụng thư viện **typeinfo**:

- Sử dụng toán tử **==** và **!=** trong class **type_info**:

```
int n;

// compare the type of n with type int32_t
if (typeid(int32_t) == typeid(n))
  cout << "n is an object of type int32_t" << endl;

// compare the type of n with type float
if (typeid(float) != typeid(n))
  cout << "n is not an object of type float" << endl;
```

Các bạn thử chạy lại đoạn code mẫu trên để tự mình xem kết quả.

- Lấy ra tên kiểu dữ liệu của một đối tượng cụ thể:

```
int64_t i_value;
float     f_value;

cout << "Type of i_value is " << typeid(i_value).name() << endl;
cout << "Type of f_value is " << typeid(f_value).name() << endl;
```

Toán tử **typeid** nhận vào một đối tượng (có thể là 1 biến), ví dụ `typeid(i_value)`, toán tử **typeid** khi sử dụng sẽ trả về một đối tượng kiểu **type_info**. Chúng ta sử dụng dấu chấm để gọi ra phương thức **name** được định nghĩa bên trong kiểu **type_info**, phương thức **name** trả về 1 chuỗi kí tự là tên kiểu dữ liệu của đối tượng chúng ta đưa vào.

```
typeid(i_value).name(); //Có thể sử dụng đối tượng cout để in tên của i_value lên màn hình
```

*Trong bài này, mình chỉ mới sử dụng thư viện **typeinfo** cho các kiểu dữ liệu cơ bản, một số thứ khác mình sẽ đề cập đến trong phần lập trình hướng đối tượng với C++.*

Từ khóa auto (auto keyword)

Các bạn cùng nhìn lại cách thông thường mà chúng ta khai báo biến.

```
<data_type> <name_of_variable> [= <original value>];
```

Dựa trên cú pháp khai báo biến này, lập trình viên phải xác định trước được kiểu dữ liệu cần sử dụng để lưu trữ giá trị.

Với chuẩn C++11 ra đời, **compiler** có thể thay bạn quyết định kiểu dữ liệu cho giá trị mà bạn muốn sử dụng bằng cách sử dụng từ khóa **auto**.

Cách sử dụng từ khóa **auto**:

```
auto <variable_name> = <expression>;
```

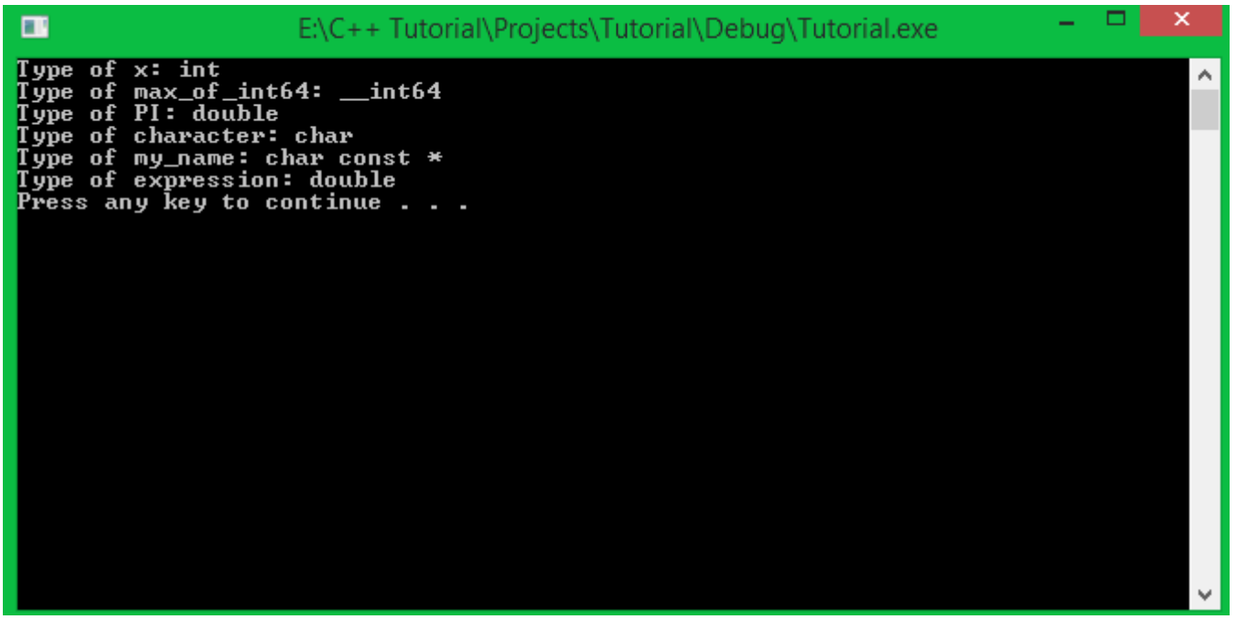
Giá trị khởi tạo là thành phần bắt buộc phải có khi sử dụng từ khóa **auto**, **compiler** sẽ dựa trên giá trị khởi tạo để quyết định kiểu dữ liệu nào phù hợp với biến (có thể là 1 con số, 1 kí tự, 1 chuỗi kí tự, hoặc 1 biểu thức toán học...).

```
auto x = 0;
auto max_of_int64 = INT64_MAX;
auto PI = 3.14;
auto character = 'V';
auto my_name = "Le Tran Dat";
```

Chúng ta cùng thử dùng thư viện **typeinfo** mà mình đã giới thiệu ở trên để xem từ khóa **auto** đã chọn kiểu dữ liệu gì cho từng biến.

```
cout << "Type of x: " << typeid(x).name() << endl;
cout << "Type of max_of_int64: " << typeid(max_of_int64).name() << endl;
cout << "Type of PI: " << typeid(PI).name() << endl;
cout << "Type of character: " << typeid(character).name() << endl;
cout << "Type of my_name: " << typeid(my_name).name() << endl;
```

Bên dưới là kết quả chạy chương trình của mình.



Compiler đã chọn đúng kiểu dữ liệu cho từng biến, nhưng chưa phải là tối ưu nhất. Ví dụ với giá trị **PI = 3.14**, chúng ta hoàn toàn có thể lưu trữ với kiểu dữ liệu **float (4 bytes)** thay vì kiểu **double (8 bytes)**. Nhưng vì **compiler** muốn đảm bảo an toàn cho dữ liệu, nên nó đã chọn kiểu có kích thước lớn hơn để đề phòng giá trị biến **PI** có thể bị thay đổi.

*Trong bài học này, mình chỉ mới hướng dẫn các bạn sử dụng từ khóa **auto** để làm việc với các kiểu dữ liệu cơ bản. Từ khóa **auto** còn có thể dùng để tự nhận dạng các kiểu dữ liệu mà chúng ta tự định nghĩa, kiểu con trỏ, các iterator trong bộ thư viện STL,... Vì thế, mình sẽ còn nhắc lại từ khóa **auto** trong những bài học sau.*

Từ khóa decltype (decltype keyword)

Cũng tương tự với từ khóa **auto**, từ khóa **decltype** giúp chương trình tự động xác định kiểu dữ liệu cho biến. Nhưng cách sử dụng từ khóa **decltype** có một chút khác biệt so với cách sử dụng từ khóa **auto**.

Để phân biệt:

- Từ khóa **auto** xác định kiểu dữ liệu dựa trên phần khởi tạo của biến.
- Từ khóa **decltype** xác định kiểu dữ liệu từ 1 biến hoặc 1 biểu thức khác.

Vì thế, khi sử dụng từ khóa **decltype**, chúng ta phải sử dụng kèm với 1 đối tượng cụ thể (1 biến, 1 biểu thức hoặc 1 đối tượng của class nào đó...).

Cách sử dụng từ khóa **decltype**:

```
decltype(<object or expression>) <variable_name> [= <initial_value>];
```

Giá trị khởi tạo (phần đặt trong ngoặc vuông) là không bắt buộc vì từ khóa **decltype** đã xác định được kiểu dữ liệu bằng cách lấy kiểu dữ liệu của đối tượng (object) hoặc biểu thức (expression).

```
int32_t i_value;
decltype(i_value) what_is_this;

cout << typeid(what_is_this).name() << endl; //int
```

Trong đoạn chương trình trên, mình khai báo 1 biến có tên là **i_value** với kiểu dữ liệu **int32_t**. Sau đó, mình dùng từ khóa **decltype** để lấy ra kiểu dữ liệu của biến **i_value** và dùng nó cho biến mà mình muốn sử dụng.

Thử so sánh kiểu dữ liệu của 2 biến:

```
int32_t i_value;
decltype(i_value) what_is_this;

if (typeid(i_value) == typeid(what_is_this))
	cout << "i_value and what_is_this have the same data type" << endl;
else
	cout << "Are you kidding me?" << endl;
```

Bởi vì từ khóa **decltype** lấy kiểu dữ liệu của đối tượng trước đó để khai báo cho đối tượng sau, nên hai đối tượng này luôn có cùng kiểu dữ liệu.

Kết hợp từ khóa auto và từ khóa decltype (C++14 standard)

Khi các bạn sử dụng **Visual studio 2015** thì sẽ được tích hợp luôn chuẩn C++14. Và các bạn có thể thực hiện khai báo như sau:

```
decltype(auto) <variable_name> = <initial_value>;
```

Từ khóa **decltype** sẽ lấy ra kiểu dữ liệu mà từ khóa **auto** đã xác định được qua giá trị khởi tạo. Vì thế, giá trị khởi tạo là thành phần bắt buộc.

```
decltype(auto) my_name = "Le Tran Dat";
cout << "Type of my_name: " << typeid(my_name).name() << endl; //char const [12]
```

Qua đoạn code mẫu trên, **compiler** đã xác định được kiểu dữ liệu dùng cho biến **my_name** là chuỗi kí tự gồm 12 kí tự.

Lưu ý: cách dùng này chỉ được hỗ trợ trong chuẩn C++14.

Tổng kết

Sử dụng từ khóa **auto** và **decltype** giúp chương trình của chúng ta dễ hiểu hơn, nhưng cũng có một số hạn chế khi để **compiler** tự động quyết định kiểu dữ liệu. Ví dụ:

```
int i_value = 10;
float f_value = 2.5f;

auto a_value = i_value * f_value;
cout << typeid(a_value).name() << ": " << a_value << endl;
```

Compiler xác định kiểu dữ liệu float cho biến a_value, nhưng giá trị in ra là 25 chứ không phải 25.0 như biến **float** thông thường.

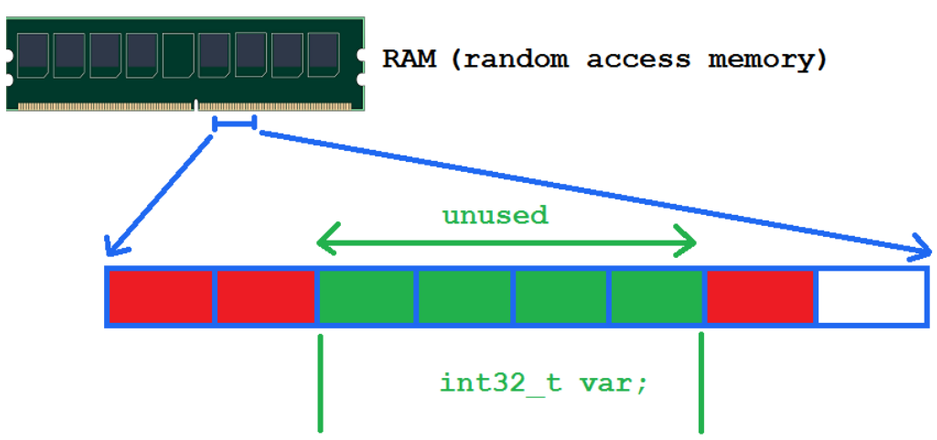
Vì thế, các bạn cần cân nhắc trước khi sử dụng những từ khóa này.

4.4 Địa chỉ của biến

Trong bài học này, mình sẽ đề cập đến vấn đề cũng khá quan trọng liên quan đến việc truy xuất biến (**variable**).

Điều gì xảy ra sau khi khai báo biến?

Đối với các kiểu dữ liệu cơ bản mà các bạn đã học trong loạt bài trước đây, và khi các bạn khai báo biến cục bộ (**local variable**), sau khi các bạn khai báo biến, hệ điều hành sẽ tìm đến 1 vùng nhớ trống trên các thiết bị lưu trữ tạm thời (RAM hoặc các vùng nhớ khác), nếu tìm được vùng nhớ có khoảng trống đủ cho kích thước của biến, biến đó sẽ nắm giữ vùng nhớ vừa tìm được.



Giả sử chương trình của chúng ta có khai báo biến `int32_t var`, và tạm thời mình cho rằng RAM là thiết bị lưu trữ duy nhất mà máy tính của bạn đang có, chương trình sẽ tìm đến vị trí có 4 bytes bộ nhớ trống và giao cho biến **var** quản lý.
*Với việc khai báo biến cục bộ, hoặc sử dụng các kiểu dữ liệu cơ bản mà các bạn đã học, chương trình sẽ cấp phát vùng nhớ cho các biến này trên một vùng nhớ được gọi là **call stack** (chúng ta sẽ có 1 bài học nói về vấn đề này).*

Vậy thì sau khi 1 vùng nhớ đã được giao cho biến quản lý, làm sao compiler biết được chính xác vị trí của biến đó trên vùng nhớ để thực hiện các lệnh truy xuất hoặc thay đổi giá trị trên biến đó?

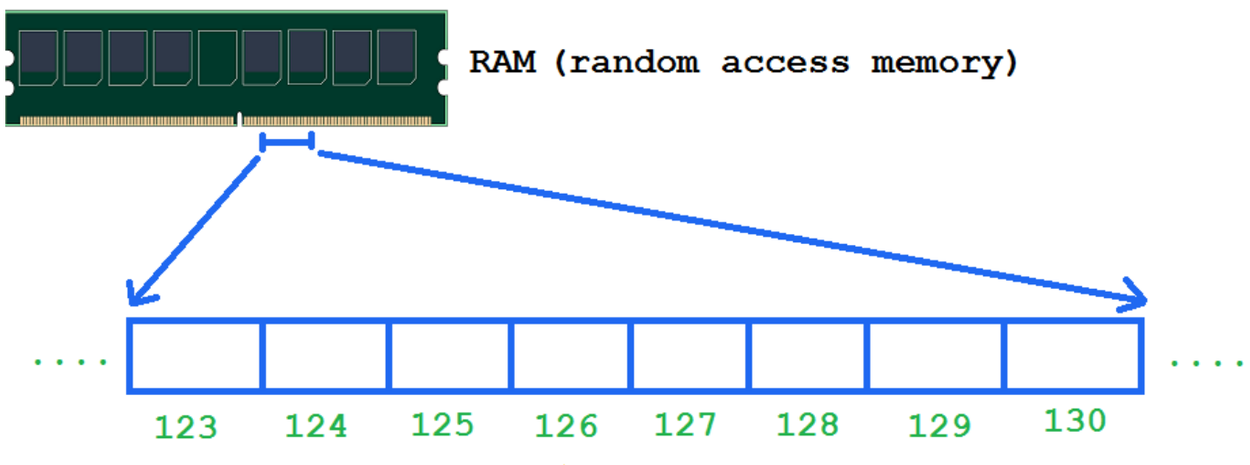
Compiler sẽ biết được vị trí của biến vì mỗi biến có 1 địa chỉ vùng nhớ trên thiết bị lưu trữ mà biến đó đang nắm giữ.

Địa chỉ của biến

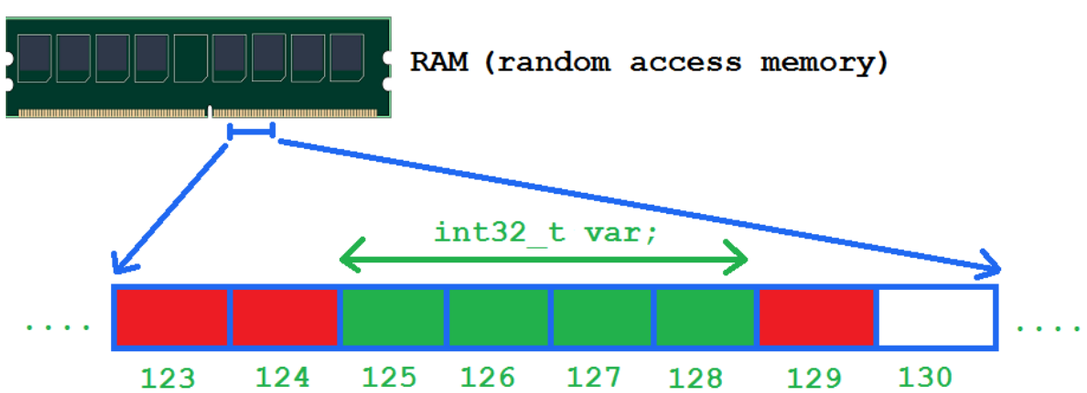
RAM hay các thiết bị cung cấp bộ nhớ tạm thời khác đều được tạo nên bởi các ô nhớ liên tiếp nhau, mỗi ô nhớ đều có 1 số thứ tự đại diện cho vị trí của ô nhớ đó trong thiết bị lưu trữ. Chúng ta có thể gọi con số đó địa chỉ của ô nhớ.

Những địa chỉ của ô nhớ chỉ là những con số ảo được tạo ra do hệ điều hành, còn về bản chất bên trong việc quản lý bộ nhớ của máy tính thì máy tính của chúng ta có những thiết bị riêng để làm điều đó.

Các bạn cứ tưởng tượng 1 ô nhớ trong thiết bị lưu trữ là một cái nhà trên con đường, để xác định được vị trí của 1 cái nhà, chúng ta cần biết địa chỉ của nhà cần tìm.



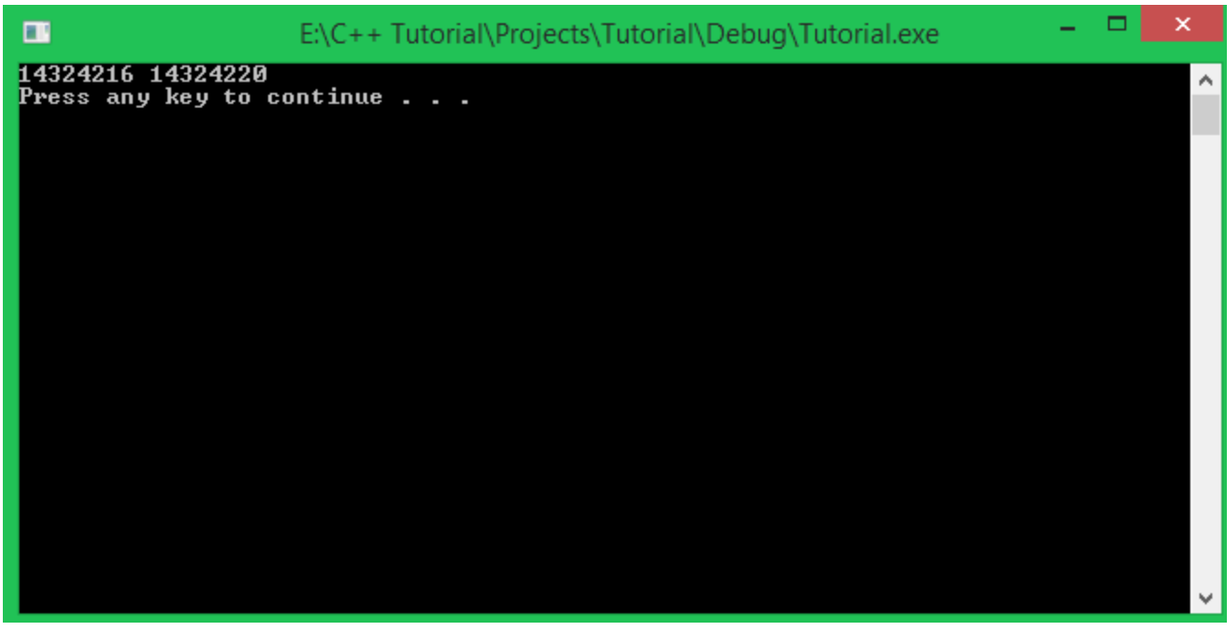
Địa chỉ ô nhớ đầu tiên được đánh số 0, và địa chỉ cuối cùng tương đương với số ô nhớ có trên thiết bị đó.



1.png?raw=true929x485

Giả sử biến **var** được khai báo bằng kiểu dữ liệu **int32_t**, và hệ điều hành tìm được vùng nhớ trống đủ 4 bytes để cung cấp cho biến **var** tại vị trí 125 đến 128, biến var sau khi được cấp phát vùng nhớ sẽ có địa chỉ 125 (là địa chỉ của ô nhớ đầu tiên mà biến nắm giữ).

Ở hình trên chỉ là minh họa cho việc cấp phát vùng nhớ cho biến có kích thước 4 bytes. Trên thực tế, địa chỉ của ô nhớ được cấp phát cho biến trong chương trình của chúng ta sẽ có giá trị rất lớn do các chương trình đang chạy trong hệ điều hành của chúng ta đã chiếm giữ trước đó.



Ở trên đây là kết quả của một chương trình mà mình viết. Mình đã tạo ra hai biến kiểu số nguyên **int32_t** có vùng nhớ nằm cạnh nhau, và mình thực hiện in ra địa chỉ của 2 biến đó.

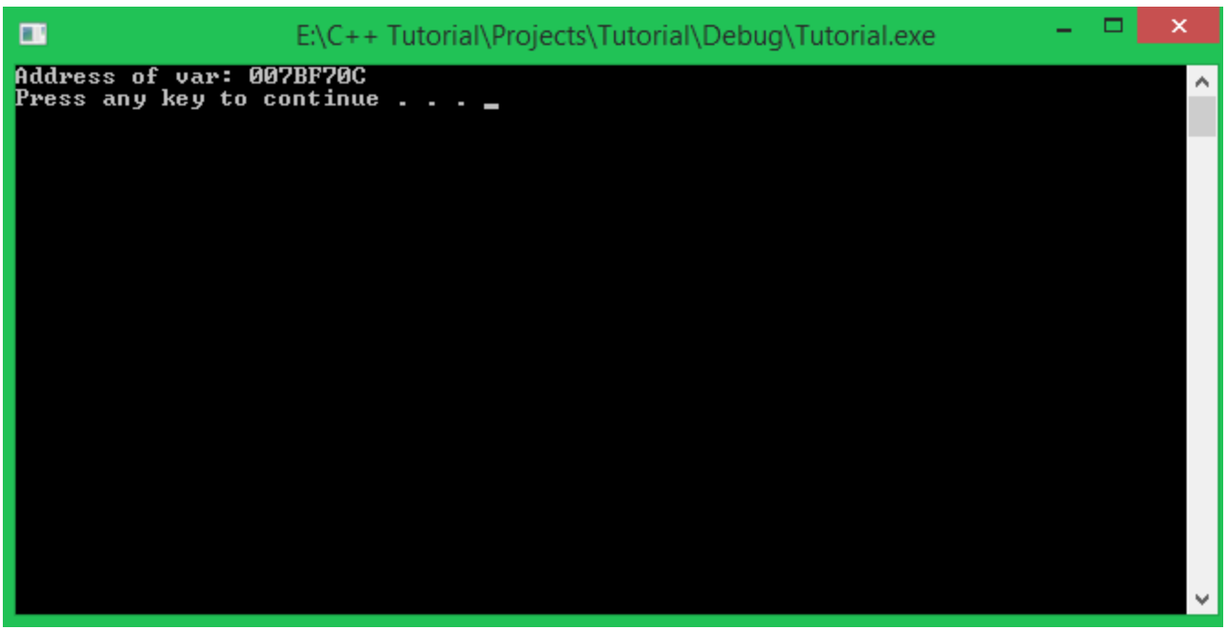
Như các bạn thấy, biến đầu tiên có địa chỉ **14324216** thì biến tiếp theo sẽ có địa chỉ cách biến đầu tiên 4 bytes (là **14324220**). Ở các bài học sau, bạn sẽ biết cách cấp phát những vùng nhớ liên tiếp nhau cho biến.

Làm thế nào để lấy được địa chỉ của biến trong ngôn ngữ C++?

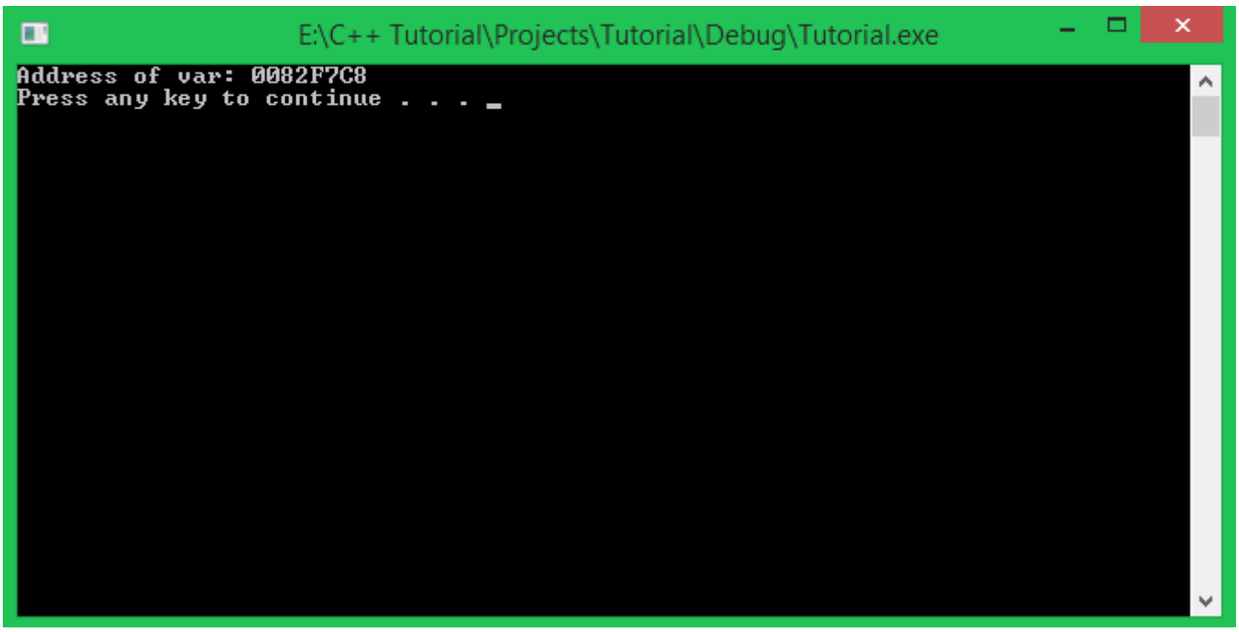
Ví dụ ta khai báo biến có tên **var** với kiểu dữ liệu bất kì mà bạn đã được học. Để lấy ra địa chỉ của biến **var** này, chúng ta đặt toán tử **&** trước tên của biến.

```
int32_t var;
cout << "Address of var: " << &var << endl;
```

Toán tử **&** được gọi là toán tử tham chiếu (**address-of operator**). Đoạn chương trình trên sẽ tìm đến chính xác địa chỉ mà biến **var** đang nắm giữ và in địa chỉ đó ra màn hình. Các bạn cùng xem kết quả bên dưới:



Thử chạy lại chương trình một lần nữa:



Chúng ta thấy qua 2 lần chạy chương trình thì địa chỉ của biến này có 2 vị trí khác nhau. Đồng nghĩa với việc chọn vị trí vùng nhớ để cấp phát cho biến hoàn toàn được thực thi tự động bởi hệ điều hành.

Địa chỉ của biến được định dạng theo hệ cơ số 16 chứ không phải hệ thập phân như chúng ta thường thấy.

Tham chiếu (Reference)

Một tham chiếu (**reference**) trong ngôn ngữ C++ cũng là một kiểu dữ liệu cơ bản, nó hoạt động như một tên giả của biến nó tham chiếu đến.

1) Cách khai báo 1 tham chiếu (reference)

Đặt toán tử & giữa kiểu dữ liệu và tên biến trong khi khai báo biến sẽ tạo thành một tham chiếu.

```
int32_t & var_reference; //use to refer to another int32_t variable
```

Khi viết đến đây, compiler sẽ báo lỗi tại dòng khai báo tham chiếu, vì 1 tham chiếu cần có giá trị khởi tạo là tên biến mà nó sẽ tham chiếu đến.

Một biến tham chiếu chỉ có thể tham chiếu đến một biến khác có cùng kiểu dữ liệu.

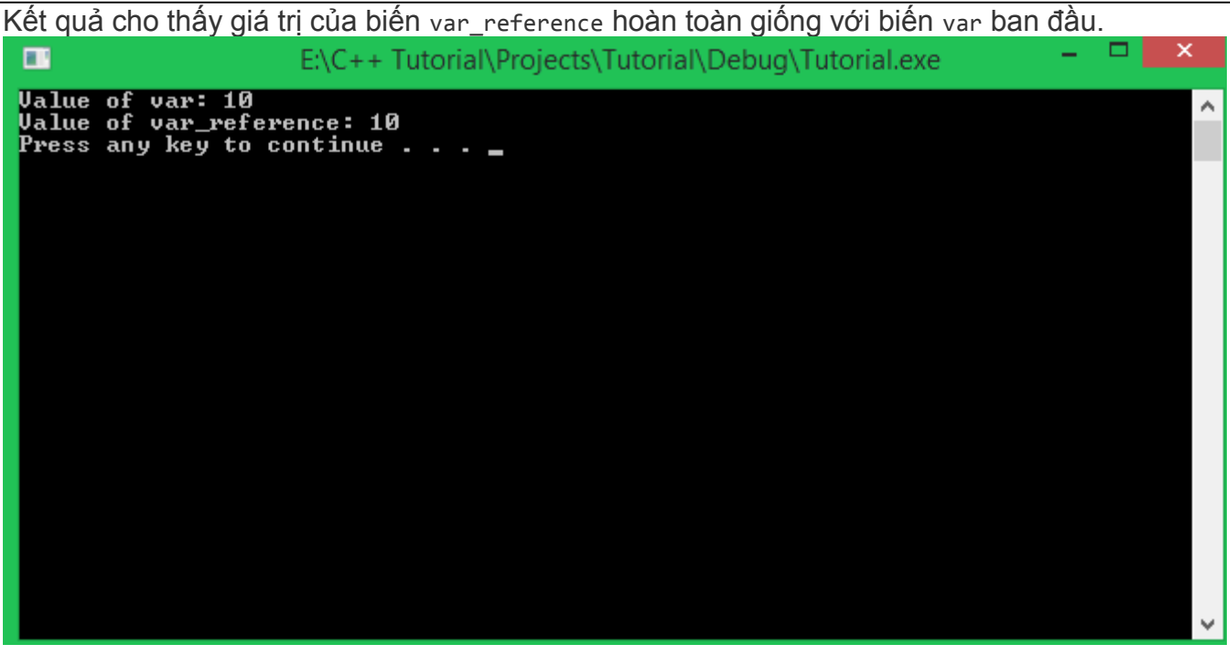
2) Thực hiện tham chiếu đến biến khác:

```
int32_t  var = 10;
int32_t &       var_reference = var;
```

3) Thử in ra giá trị của 2 biến `var` và `var_reference`:

```
cout << "Value of var: " << var << endl;
cout << "Value of var_reference: " << var_reference << endl;
```

Kết quả cho thấy giá trị của biến `var_reference` hoàn toàn giống với biến `var` ban đầu.

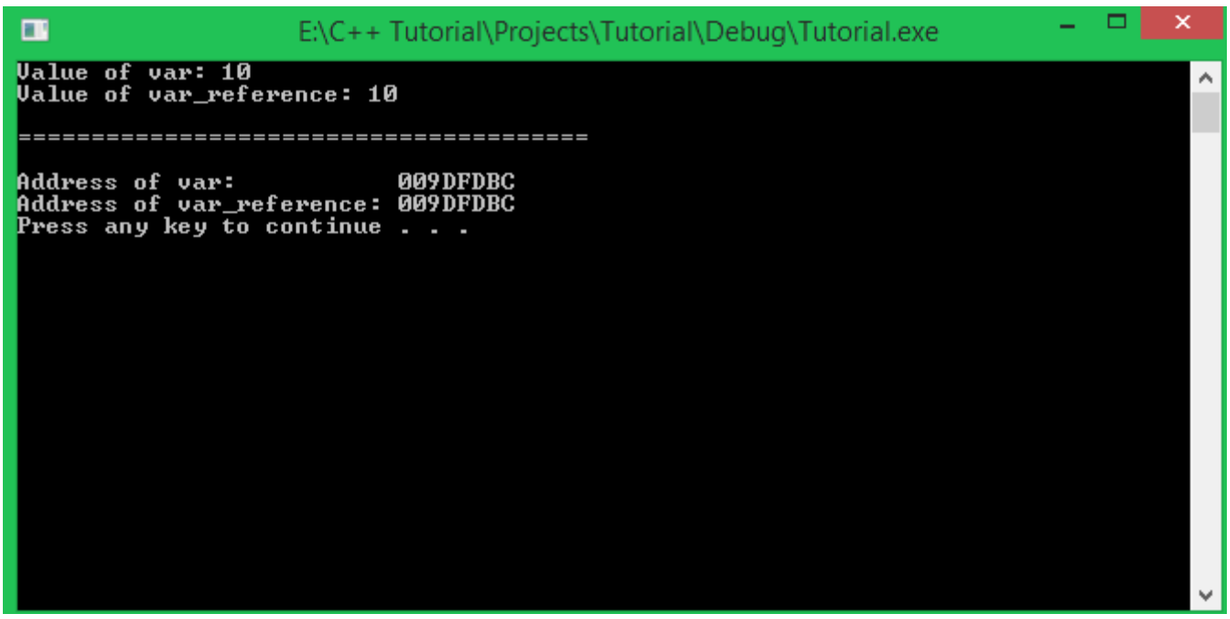


Điều gì đã xảy ra? Chúng ta cùng làm thêm 1 bước nữa trước khi đi vào kết luận.

4) In ra địa chỉ của 2 biến `var` và `var_reference`:

```
cout << "Address of var: " << &var << endl;
cout << "Address of var_reference: " << &var_reference << endl;
```

Và đây là kết quả chương trình:

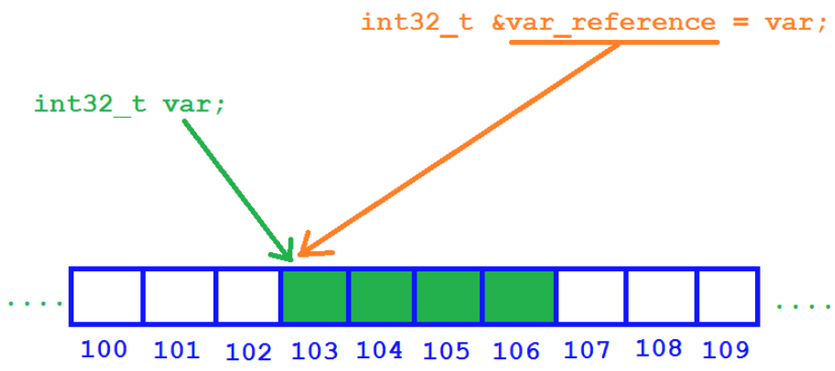


Kết quả cho thấy giá trị của tham chiếu `var_reference` và địa chỉ của `var_reference` hoàn toàn giống với biến `var` ban đầu. Vậy nó có phải là một bản sao của biến `var`? Hoàn toàn không phải nhé các bạn.
Về mặt ngữ nghĩa của dòng lệnh

```
int32_t & var_reference = var;
```

Toán tử **&** không mang ý nghĩa "**địa chỉ của**", mà nó có nghĩa "**tham chiếu đến**".

Khi thực hiện tham chiếu từ biến `var_reference` đến biến `var`, biến `var_reference` sẽ kiểm soát vùng nhớ có địa chỉ là địa chỉ của biến `var`.



Lúc này, biến var và biến var_reference vẫn là 2 tên biến khác nhau, nhưng chúng có cùng địa chỉ.

Điều này có nghĩa khi chúng ta thực hiện thay đổi giá trị cho biến `var_reference`, giá trị của biến `var` cũng thay đổi và ngược lại.

```
int32_t var = 10;
int32_t & var_reference = var;
```

```
cout << "Value of var: " << var << endl;
cout << "Value of var_reference: " << var_reference << endl;

var++;                          //Increase value of var
var_reference++; //Increase value of var_reference

cout << endl << "=====================================" << endl << endl;

cout << "New value of var: " << var << endl;
cout << "New value of var_reference: " << var_reference << endl;
```

Một số lưu ý khi sử dụng tham chiếu

1) Các bạn không thể khởi tạo tham chiếu không phải hằng số bằng một biến hằng số.

Đoạn code sau sẽ báo lỗi:

```
const int32_t var = 10;
int32_t & ref = var;
```

Vì biến tham chiếu `ref` có thể thay đổi giá trị bên trong vùng nhớ, nhưng lúc này, `var` là hằng số nên giá trị vùng nhớ không được phép thay đổi. Điều này dẫn đến xung đột nên compiler ngăn chặn chúng ta biên dịch chương trình.

2) Nhưng chúng ta có thể tham chiếu một biến tham chiếu hằng số đến một hằng số.

```
const int32_t var = 10;
const int32_t & ref = var;
```

3) Hoặc chúng ta có thể tham chiếu một biến tham chiếu hằng số đến một biến bình thường.

```
int32_t var = 10;
const int32_t & ref = var;
```

4) Chúng ta không có thể thực hiện nhiều lần tham chiếu đến nhiều biến khác nhau.

```
#include <iostream>

using namespace std;

int main() {

    int32_t i_value1 = 10;
    int32_t i_value2 = 20;

    cout << "Address of i_value1: " << &i_value1 << endl;

    int32_t & ref = i_value1;
    cout << "Address of ref: " << &ref << endl;
    cout << "Value of ref:" << ref << endl;

    ref = i_value2;
    cout << "Address of ref: " << &ref << endl;
    cout << "Value of ref:" << ref << endl;
    cout << "Value of i_value1:" << i_value1 << endl;
}
```

Đây là kết quả của chương trình này:

```
Address of i_value1: 0x6afef4
Address of ref: 0x6afef4
Value of ref:10
Address of ref: 0x6afef4
Value of ref:20
Value of i_value1:20
```

Như các bạn thấy, địa chỉ của ref không bị thay đổi, nghĩa là phép gán thứ hai chỉ là phép gán giá trị thông thường, chứ không phải tham chiếu.

Lưu ý: Biến tham chiếu chỉ có thể tham chiếu một lần duy nhất ngay khi khai báo và khởi tạo. Chúng ta không thể tham chiếu đến biến có địa chỉ khác sau khi đã khởi tạo.

Tổng kết

Việc hiểu được địa chỉ của biến khá là quan trọng. Sau này khi học đến phần con trỏ trong C++, mình sẽ còn nhắc lại khái niệm này.

PHẦN 5: KIỂU DỮ LIỆU MẢNG

5.0 Mảng một chiều

Tiếp tục với bài học ngày hôm nay, chúng ta sẽ cùng tìm hiểu về một cách tổ chức dữ liệu cơ bản trong thiết bị lưu trữ tạm thời của máy tính giúp khắc phục một số nhược điểm của việc sử dụng các biến thông thường.

Đặt vấn đề

Giảng viên cần tìm ra điểm số cao nhất của bài kiểm tra môn lập trình cơ sở. Giả sử lớp học có 30 sinh viên có số thứ tự 1 đến 30.

Công việc của những lập trình viên chúng ta là giúp giảng viên này chỉ ra số thứ tự của sinh viên có điểm kiểm tra cao nhất, và điểm cao nhất đó là bao nhiêu bằng cách viết chương trình ngôn ngữ C++ trên máy tính để tiết kiệm thời gian suy nghĩ.

Tìm hướng giải quyết

Với yêu cầu như trên, chúng ta cần 30 biến để lưu lại điểm của 30 sinh viên.

```
int32_t score_of_student1;
int32_t score_of_student2;
//....
int32_t score_of_student30;
```

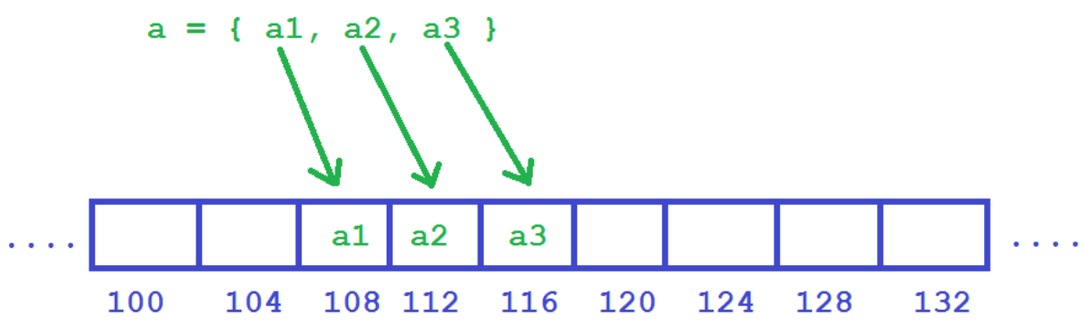
Vậy là chúng ta cần tới 30 dòng lệnh khai báo 30 biến, chưa kể mất thời gian viết thêm 30 dòng lệnh nhập dữ liệu vào là điểm của từng sinh viên, sau đó chúng ta còn phải tìm điểm cao nhất.

Một vấn đề khác nãy sinh: Sau khi tìm ra điểm số cao nhất từ 30 biến trên, làm thế nào chúng ta biết điểm số đó là của sinh viên có số thứ tự nào trong khi 30 biến này được cấp phát hoàn toàn tách biệt nhau (không theo 1 thứ tự nhất định)?

Rất may mắn cho chúng ta khi ngôn ngữ C/C++ đưa ra cho chúng ta một khái niệm về tổ chức dữ liệu liên tiếp nhau trên thiết bị cung cấp bộ nhớ. Chúng ta có thể gọi là **Mảng một chiều (Array)**.

Mảng một chiều (Array)

Mảng một chiều (**array**) là một dãy các phần tử có cùng kiểu dữ liệu được đặt liên tiếp nhau trong một vùng nhớ, chúng ta có thể ngay lập tức truy xuất đến một phần tử của dãy đó thông qua chỉ số của mỗi phần tử.



Như hình trên, giả sử mình khai báo mảng một chiều có 3 phần tử kiểu **int32_t**, mỗi phần tử sẽ có kích thước **4 bytes**.

Mình lấy ví dụ hệ điều hành tìm thấy vùng nhớ trống đủ chỗ chứa 3 phần tử của mảng tại địa chỉ **108**, thì phần tử đầu tiên a1 sẽ có địa chỉ là địa chỉ ô nhớ đầu tiên mà hệ điều hành cấp phát (là **108**). Khi

đó, phần tử thứ 2 sẽ có địa chỉ là địa chỉ của phần tử thứ nhất cộng thêm 4 (4 là kích thước kiểu dữ liệu **int32_t**), tương tự cho phần tử thứ 3.

Với kiểu tổ chức dữ liệu này, chúng ta chỉ cần quan tâm đến 2 điều:

- Địa chỉ ô nhớ đầu tiên trong mảng.
- Số phần tử của mảng.

Từ đó, chúng ta có thể truy xuất đến toàn bộ phần tử trong mảng.

Khai báo mảng một chiều

Chúng ta có nhiều cách để khai báo mảng một chiều khác nhau:

- Khai báo nhưng không khởi tạo các phần tử:

```
<data_type> <name_of_array>[<number_of_elements>];
```

Với cách khai báo này, chúng ta cần ghi rõ cho **compiler** biết số lượng phần tử mà bạn cần sử dụng đặt trong cặp dấu ngoặc vuông. Ví dụ:

```
int32_t age_of_students[30];
```

Mình vừa tạo ra một mảng dữ liệu kiểu **int32_t** để lưu trữ số tuổi của 30 sinh viên trong 1 lớp học.

Vì mình chưa khởi tạo giá trị cụ thể cho 30 phần tử trong mảng, nên khi truy xuất đến giá trị của từng phần tử, chúng ta có thể nhận được giá trị khởi tạo mặc định của kiểu int32_t là 0 hoặc giá trị rác (tùy vào **compiler**).

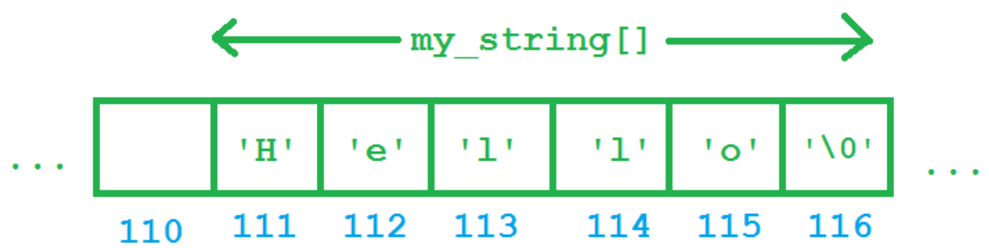
- Khai báo và khởi tạo giá trị cho mỗi phần tử:

```
<data_type> <name_of_array>[] = { <value1>, <value2>, ... <valueN> };
```

Với cách khai báo này, chúng ta không cần thiết xác định trước số phần tử của mảng. Compiler sẽ xác định số phần tử thông qua số lượng giá trị mà bạn khởi tạo.

```
char my_string[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

Mình vừa khai báo một mảng phần tử với kiểu kí tự (Chúng ta sẽ đi sâu hơn về chuỗi kí tự trong những bài học sau), **compiler** nhìn vào số lượng kí tự mình khởi tạo và cấp phát 6 ô nhớ liên tục nhau trên vùng nhớ còn trống.



Với kiểu kí tự, mỗi phần tử chỉ chiếm 1 byte, nên chúng ta có 6 bytes liên tiếp nhau để chứa được chuỗi kí tự trên.

Truy xuất đến các phần tử trong mảng một chiều

Sau khi biết cách khai báo mảng một chiều (**array**), điều tiếp theo chúng ta cần quan tâm là làm thế nào để truy xuất đến một phần tử trong mảng.

Mỗi phần tử trong mảng sẽ đi kèm với một chỉ số cho biết vị trí của phần tử có khoảng cách bao nhiêu so với phần tử đầu tiên của mảng. Phần tử đầu tiên của mảng mang chỉ số **0**, phần tử cuối cùng của mảng có **N** phần tử sẽ có chỉ số **(N - 1)**.
Cú pháp truy xuất phần tử trong mảng một chiều:

```
<name_of_array>[index];
```

Trong đó, **index** là một số nguyên đại diện cho chỉ số của phần tử trong mảng một chiều.

Ví dụ với một mảng một chiều kiểu int32_t có 5 phần tử được khai báo như sau:

```
int32_t values[] = { 2, 4, 6, 8, 10 };
```

Khi đó, các phần tử trong mảng lần lượt là:

```
values[0]; //2
values[1]; //4
values[2]; //6
values[3]; //8
values[4]; //10
```

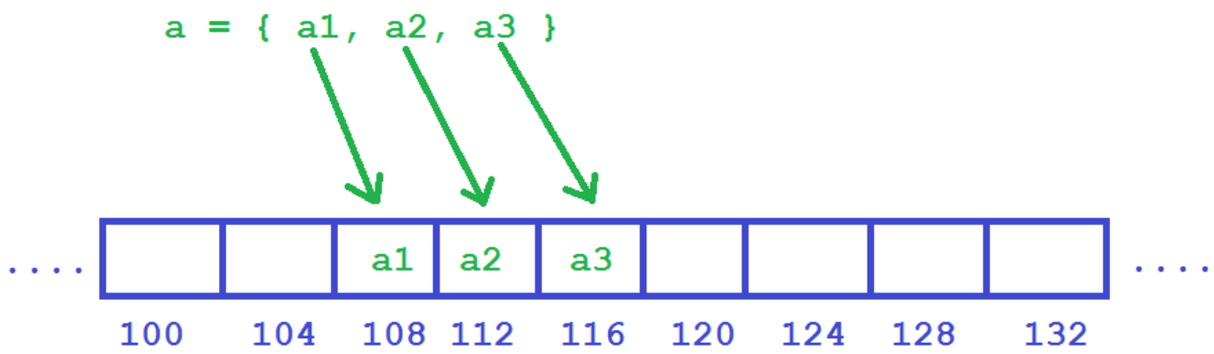
Giải thích cho việc tại sao chỉ số của mảng một chiều trong C/C++ bắt đầu từ 0:

Mỗi phần tử trong mảng sẽ đi kèm với một chỉ số cho biết vị trí của phần tử có khoảng cách bao nhiêu so với phần tử đầu tiên của mảng.

Sau khi khai báo mảng một chiều, địa chỉ của mảng ứng với địa chỉ của phần tử đầu tiên trong mảng. Vị trí của các phần tử sẽ được tính dựa trên công thức:

```
index = (address_of_current_element - address_of_the_first_element) / sizeof(data_type);
```

Lấy lại ví dụ mảng có 3 phần tử kiểu int32_t như trong mục **Mảng một chiều (Array)**



Cho rằng địa chỉ của mảng **a** (cũng là địa chỉ của phần tử a1) là **108**. Vậy chỉ số của phần tử đầu tiên a1 là:

```
index_of_a1 = (address_of_a1 - address_of_the_first_element) / sizeof(int32_t);
index_of_a1 = (108 - 108) / 4 = 0;
```

Như vậy, phần tử đầu tiên của mảng có chỉ số là **0**.

Đây chỉ là phần mình làm rõ cho các bạn tại sao chỉ số của mảng một chiều trong C/C++ bắt đầu từ 0 và kết thúc tại `(số_phần_tử - 1)`. Các bạn không cần quan tâm đến việc tính toán chỉ số của mỗi phần tử mà **compiler** sẽ làm giúp bạn.

In ra giá trị của tất cả phần tử trong mảng

Để quản lý mảng một chiều, chúng ta cần biết:

- Địa chỉ phần tử đầu tiên của mảng. (Có thể có được thông qua `<array_name>[0]`)
- Số lượng phần tử của mảng.

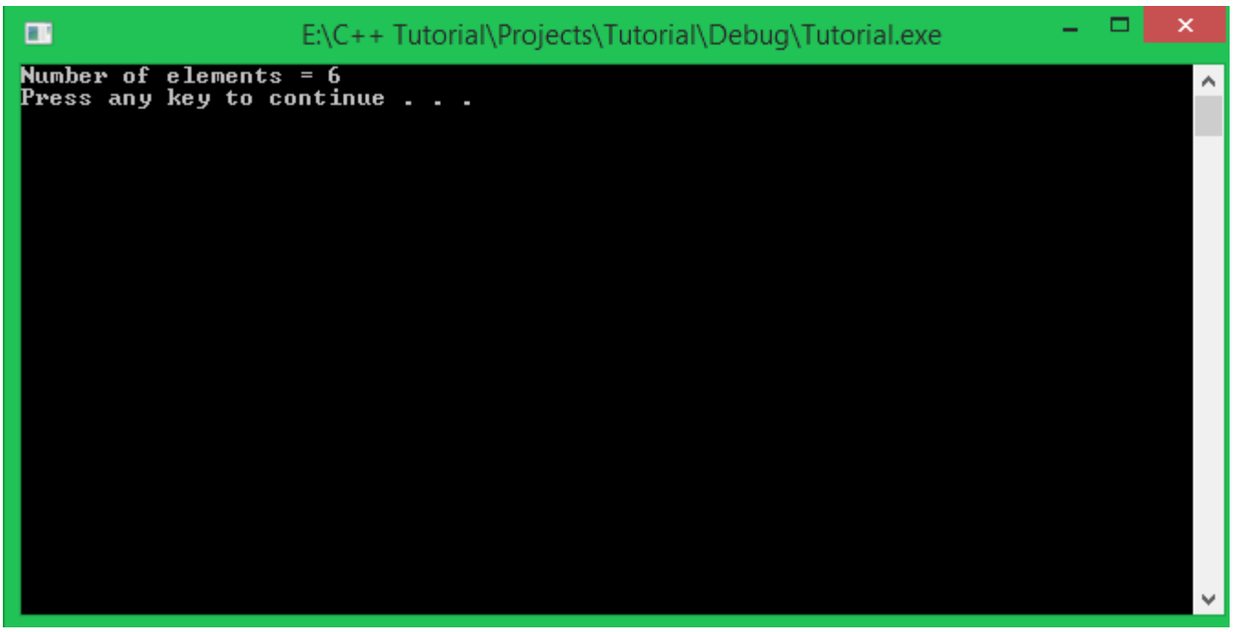
Mình sẽ thực hiện một phương pháp tổng quát để lấy ra số lượng phần tử của mảng:

```
<number_of_elements> = sizeof(<name_of_array>) / sizeof(<type_of_array>);
```

Chúng ta sử dụng toán tử **sizeof**, truyền vào tên của mảng chúng ta sẽ nhận được giá trị là tổng kích thước bộ nhớ sử dụng cho mảng, chia cho kích thước của một phần tử của mảng chúng ta sẽ có được số lượng phần tử. Ví dụ:

```
double d_values[] = { 2.08, 1.32, 6, 4.1, 12, 999.99 };
int32_t num_of_elements = sizeof(d_values) / sizeof(double);

//another way
num_of_elements = sizeof(d_values) / sizeof(d_values[0]);

cout << "Number of elements = " << num_of_elements << endl;
```

Kết quả chương trình sẽ cho ta thấy mảng có 6 phần tử:



Như cách thông thường, chúng ta thường định nghĩa trước số lượng phần tử tối đa mà mảng một chiều có thể chứa như sau:

```
#define ARRAY_SIZE 100

//........

float f_values[ARRAY_SIZE];
```

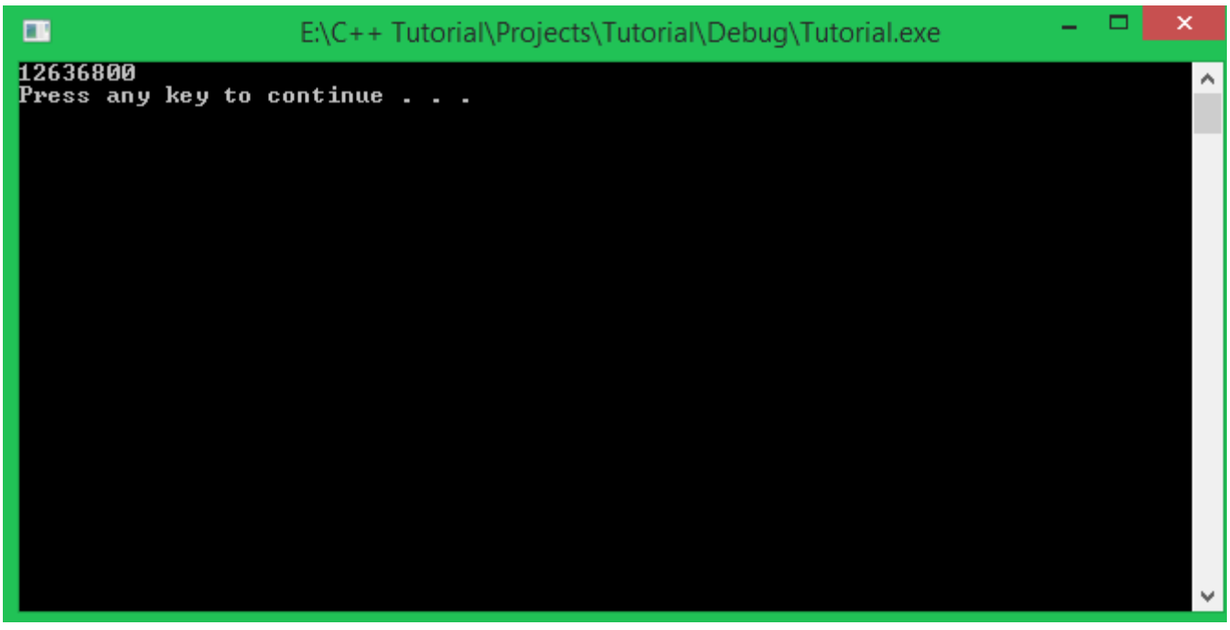
Lúc này, chúng ta chỉ cần sử dụng `ARRAY_SIZE` như là số lượng phần tử của mảng. Nhưng cách này có thể là hao tốn bộ nhớ khi số lượng phần tử thực sự cần sử dụng không đạt đến con số `ARRAY_SIZE`. Vì thế, mình thường tính số phần tử của mảng theo cách tổng quát mà mình trình bày ở trên.

Điều gì xảy ra nếu chúng ta truy xuất mảng bằng chỉ số lớn hơn số lượng phần tử?

Các bạn thử chạy đoạn chương trình sau:

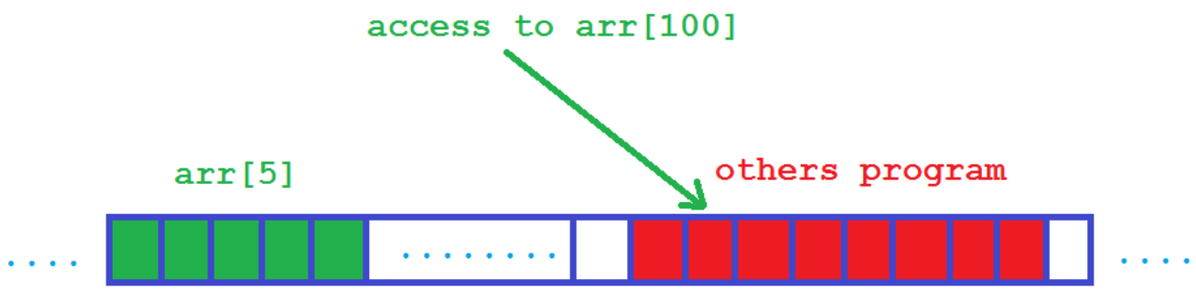
```
int32_t arr[] = { 1, 2, 3, 4, 5 }; //create an array with 5 elements
cout << arr[100] << endl;
```

Ở lần chạy đầu tiên của đoạn chương trình trên, máy mình cho ra kết quả:



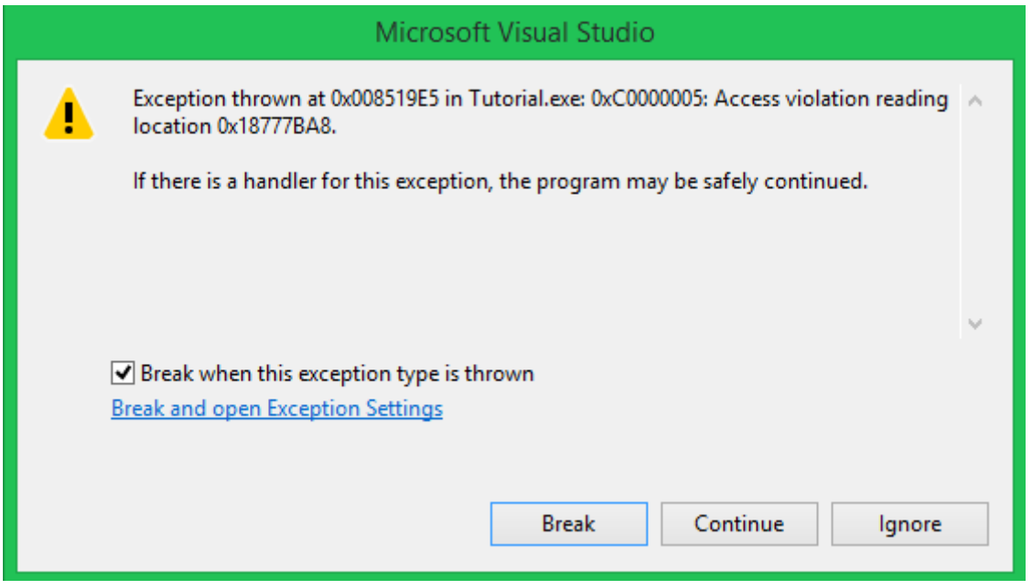
Thử chạy lại chương trình nhiều lần khác nhau, các bạn sẽ thấy được nhiều giá trị khác nhau. Những giá trị này ở đâu ra?

Đó chính là những giá trị thuộc vùng nhớ mà chương trình khác đang quản lý.



Có thể sau khi các chương trình khác sử dụng vùng nhớ đó và trả lại cho hệ điều hành quản lý, giá trị của ô nhớ vẫn còn giữ nguyên, nên khi truy cập mảng với chỉ số vượt quá số lượng phần tử tối đa, chúng ta nhận được những giá trị không có ý nghĩa.

Trường hợp xấu hơn có thể xảy ra là khi các chương trình khác đang sử dụng vùng nhớ mà bạn truy cập đến, Visual studio sẽ đưa ra cảnh báo về việc xung đột vùng nhớ và cho dừng chương trình của bạn.



Vì thế việc quản lý số lượng phần tử của mảng là rất quan trọng.

Nhập dữ liệu cho mảng một chiều (Array input)

Giả sử chúng ta có mảng một chiều dùng để chứa 10 số nguyên (có chỉ số từ 0 đến 9). Để nhập dữ liệu cho từng phần tử trong mảng này, chúng ta có thể sử dụng đối tượng **cin** trong thư viện **iostream** mà các bạn đã được học.

```
cin >> <name_of_array>[index];
```

Trong đó, **index** là chỉ số của phần tử của mảng mà chúng ta cần nhập giá trị từ bàn phím và đưa vào phần tử.

```
int32_t arr[10];
for(int32_t index = 0; index <= 9; index++)       {
        cin >> arr[index];
}
```

Mình vừa sử dụng vòng lặp **for** (vì mình biết được số lượng phần tử của mảng nên mình biết cần lặp bao nhiêu lần), trong vòng lặp **for** này, mình sử dụng biến **index** và cho nó di chuyển từ giá trị **0 đến 9** tương ứng với từng chỉ số của các phần tử trong mảng. Với mỗi giá trị **index** được gán, mình thực hiện nhập dữ liệu từ bàn phím bằng đối tượng **cin** cho phần tử **arr[index]**.

Một cách tổng quát hơn để nhập dữ liệu cho mảng một chiều

Ở ví dụ trên, mình cho mảng số nguyên có số lượng phần tử cố định là 10. Đối với mảng một chiều có số lượng phần tử khác nhau thì ta làm thế nào?

Việc đầu tiên chúng ta cần làm là tìm ra số lượng phần tử của mảng. Ví dụ:

```
int32_t i_values[100];
int32_t num_of_elements = sizeof(i_values) / sizeof(int32_t);

for(int32_t index = 0; index <= (num_of_elements - 1); index++)    {
        cin >> i_values[index];
}
```
Với cách này, chúng ta có thể không cần quan tâm đến số lượng phần tử hiện tại của mảng, mà mình để **compiler** tính giúp mình.

Mình cho biến index chạy từ 0 đến (num_of_elements - 1) vì như mình đã nói ở trên, mảng một chiều có chỉ số bắt đầu từ 0 đến `số_lượng_phần_tử` trừ đi 1.

Đưa ra nhắc nhở khi nhập dữ liệu cho mảng

Chúng ta nên thông báo cho người dùng biết là chúng ta đang nhập dữ liệu cho phần tử nào trong mảng.

```
int32_t i_values[100];
int32_t num_of_elements = sizeof(i_values) / sizeof(int32_t);

for(int32_t index = 0; index <= (num_of_elements - 1); index++)    {
        cout << "Value of element " << index << ": ";
        cin >> i_values[index];
}
```
Như vậy, người dùng sẽ tránh được việc nhập nhầm thứ tự dữ liệu cho các phần tử trong mảng.

Ngoài việc dùng đối tượng cin, chúng ta cũng có thể gán trực tiếp giá trị cho các phần tử trong mảng thông qua toán tử gán.

```
int32_t i_values[100];
int32_t num_of_elements = sizeof(i_values) / sizeof(int32_t);

for(int32_t index = 0; index <= (num_of_elements - 1); index++)    {
        i_values[index] = index + 1;
}
```

Nhập dữ liệu cho ô nhớ có chỉ số vượt quá số lượng phần tử

Cũng tương tự như việc bạn truy xuất đến phần tử với chỉ số vượt ngoài tầm số lượng phần tử trong mảng, Visual studio sẽ đưa ra cảnh báo xung đột vùng nhớ và dừng chương trình.

Tổng kết

Cùng nhìn lại vấn đề mình đặt ra ngay từ đầu bài học, mảng một chiều đã giúp chúng ta tiết kiệm thời gian hơn khi mà chỉ với 1 dòng lệnh khai báo mảng một chiều, chúng ta có thể quản lý 30 vùng nhớ liên tiếp nhau dùng để lưu trữ điểm của cả 30 sinh viên. Chúng ta cũng có thể biết được điểm số nào là của sinh viên nào thông qua chỉ số của mảng đó.

Mảng một chiều đã khắc phục nhiều nhược điểm của việc khai báo các biến đơn lẻ. Tuy nhiên, nó cũng có một số nhược điểm riêng như việc dư thừa vùng nhớ khi không dùng hết số lượng ô nhớ đã cấp phát, hoặc số lượng phần tử được yêu cầu quá lớn nên hệ điều hành không đủ khả năng cấp phát. Chúng ta sẽ tìm cách giải quyết những vấn đề này trong những bài học sau.

Bài tập cơ bản

Với yêu cầu đặt ra ban đầu, giảng viên cần biết điểm số cao nhất của 30 sinh viên trong lớp, đồng thời muốn biết điểm cao nhất là của sinh viên có số thứ tự bao nhiêu. Bạn hãy sử dụng mảng 1 chiều để giải quyết vấn đề này. (Điểm của sinh viên được nhập từ bàn phím)

5.1 Các thao tác cơ bản với mảng một chiều

Chào các bạn! Chúng ta tiếp tục đồng hành trong khóa học lập trình trực tuyến ngôn

Trong bài học này, mình sẽ hướng dẫn các bạn thực hiện một số thao tác cơ bản với mảng một chiều, giúp các bạn hình thành tư duy giải các bài toán có thể giải quyết được bằng mảng một chiều cơ bản.

Sao chép mảng một chiều

Để tạo ra một bản sao khác của mảng một chiều ban đầu, chúng ta cần khai báo thêm 1 mảng một chiều khác có cùng kích thước với mảng ban đầu. Ví dụ, ta có mảng một chiều cần sao chép như sau:

```
#define ARRAY_SIZE 50
//........

int32_t arr[ARRAY_SIZE];
int32_t arr_clone[ARRAY_SIZE];
```

Việc thực hiện sao chép giá trị từ mảng `arr` ban đầu sang mảng `arr_clone` đơn giản chỉ là gán giá trị của phần tử có cùng chỉ số ở mảng `arr` cho mảng `arr_clone`.

```
for(int32_t index = 0; index <= (ARRAY_SIZE - 1); index++){

        arr_clone[index] = arr[index];
}
```

Tìm kiếm một phần tử trong mảng một chiều

Vấn đề này cũng tương đương với việc kiểm tra sự tồn tại của một phần tử (hoặc giá trị) trong mảng một chiều.

Đặt vấn đề

Ví dụ:

```
char ch_array[] = { 'L', 'e', 'T', 'r', 'a', 'n', 'D', 'a', 't', '\n' };
```

Mình có mảng một chiều kiểu dữ liệu **char** đã được khởi tạo bằng các kí tự trong tên của mình như trên, mình muốn xác định xem liệu 1 kí tự mình nhập từ bàn phím có giống với kí tự nào trong tên của mình hay không.

Tìm hướng giải quyết

Giả sử kí tự mình nhập vào từ bàn phím là **'D'**, nếu chưa sử dụng đến máy tính mà chỉ dùng mắt thường thì chúng ta sẽ làm gì để nhận biết kí tự **'D'** có tồn tại trong mảng `ch_array` hay không? Mình sẽ lần lượt nhìn vào từng kí tự của mảng `ch_array`, so sánh từng kí tự mình đang xem xét với kí tự **'D'** mình đã nhập từ bàn phím. Phép so sánh sẽ được mình thực hiện từ kí tự có chỉ số 0 đến kí tự có chỉ số (10 - 1) trong mảng. Với mỗi lần kiểm tra, sẽ xảy ra 2 trường hợp:

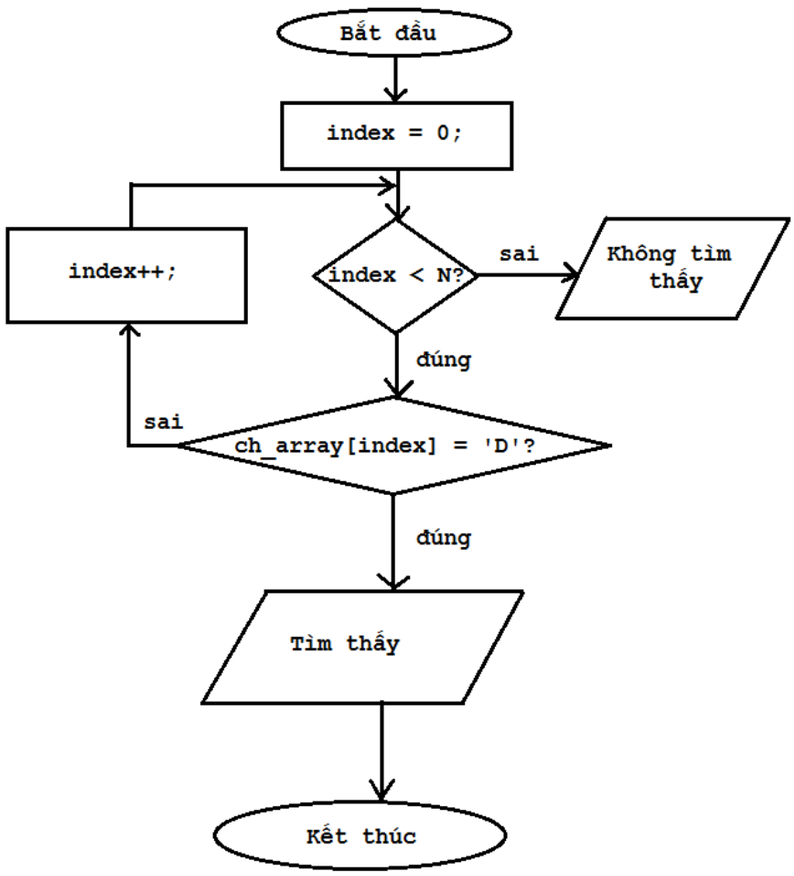
- Nếu mình bắt gặp kí tự giống với kí tự **'D'** mà mình đã nhập, mình sẽ không so sánh tiếp nữa, mà kết luận ngay là kí tự **'D'** có ồn tại trong mảng `ch_array`.

- Nếu kí tự mình đang xem xét khác kí tự **'D'** mà mình vừa nhập, mình chuyển đến kí tự tiếp theo và thực hiện so sánh tương tự.

Nếu đã so sánh hết phần tử trong mảng mà không tìm được kí tự nào trùng khớp với kí tự **'D'** mình đã nhập, lúc này mình có thể kết luận không có phần tử **'D'** nào trong mảng `ch_array`.

Định hình giải pháp dưới dạng sơ đồ khối

Chúng ta hoàn toàn có thể thực hiện giải pháp này trên máy tính, nhưng mình chưa bắt tay vào viết code ngay, mà mình sẽ vẻ ra sơ đồ khối để các bạn hình dung trước.



0.png?raw=true712x734

Viết code cho từng bước

Đầu tiên, chúng ta cần khai báo mảng một chiều kiểu char như yêu cầu, sau đó ta tính luôn số lượng phần tử có trong mảng:

```
char ch_array[] = { 'L', 'e', 'T', 'r', 'a', 'n', 'D', 'a', 't', '\n' };
int32_t N = sizeof(ch_array) / sizeof(char); //calculate the number of elements
```

Chúng ta phải nhập 1 kí tự từ bàn phím và dùng kí tự đó để so sánh, chúng ta cần 1 biến kiểu char để lưu trữ kí tự nhập vào:

```
char ch;
cout << "Enter a character: ";
cin >> ch;
```

Để so sánh biến **ch** với từng phần tử trong mảng `ch_array`, chúng ta sẽ dùng vòng lặp for để truy xuất đến tất cả các phần tử từ chỉ số 0 đến chỉ số (N - 1):

```
for (int32_t index = 0; index <= (N - 1); index++) {

}
```

Trong vòng lặp **for** này, chúng ta sẽ thực hiện so sánh biến **ch** với phần từ `ch_array[index]` để kiểm tra xem chúng có giống nhau hay không. Chúng ta sẽ dùng 1 biến kiểu **bool** khai báo ở trên vòng lặp **for** để lưu kết quả.

```
bool check = false;
for (int32_t index = 0; index <= (N - 1); index++) {

}
```

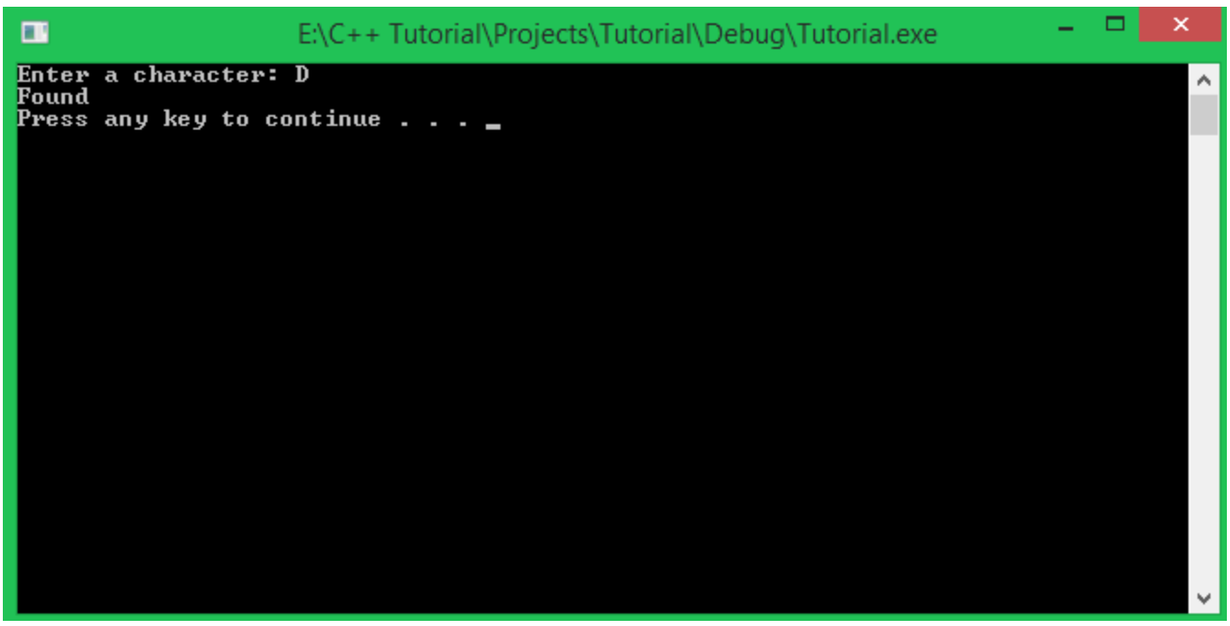
Biến **check** ban đầu có giá trị **false**, nghĩa là hiện tại không tìm thấy phần tử nào giống với biến **ch** đã nhập vào. Nếu bắt gặp phần tử có kí tự giống với kí tự mà biến **ch** lưu trữ, biến check sẽ chuyển sang giá trị **true**.

```
bool check = false;
for (int32_t index = 0; index <= (N - 1); index++) {

        if(ch_array[index] == ch)        {
                check = true;
        }
}
```

Cuối cùng, chúng ta dựa vào giá trị của biến check để chúng ta kết luận:

```
if(check == true)
        cout << "Found" << endl;
```

```
else
	cout << "Not found" << endl;
```
Thử chạy chương trình và nhập vào kí tự **'D'** và xem kết quả:



Các bạn thử chạy lại chương trình và nhập những kí tự khác để kiểm tra lại.

Dưới đây là mã nguồn đầy đủ của mình:

```
#include <iostream>
#include <cstdint>
using namespace std;

int main()
{
	char ch_array[] = { 'L', 'e', 'T', 'r', 'a', 'n', 'D', 'a', 't', '\n' };
	int32_t N = sizeof(ch_array) / sizeof(char); //calculate the number of elements

	char ch;
	cout << "Enter a character: ";
	cin >> ch;

	bool check = false;
	for (int32_t index = 0; index <= (N - 1); index++) {

		if (ch_array[index] == ch) {
			check = true;
			break; //break the loop immediately when ch is found.
		}
	}

	if (check == true)
		cout << "Found" << endl;
	else
		cout << "Not found" << endl;

	system("pause");
	return 0;
}
```
Tại câu lệnh điều kiện **if** trong vòng lặp **for**, mình thực hiện lệnh **break** để thoát ra khỏi vòng lặp khi tìm thấy kí tự giống với biến **ch**. Làm như thế có thể tiết kiệm thời gian tính toán của máy tính.

Chèn một phần tử mới vào vị trí bất kì trong mảng một chiều

Đặt vấn đề

Chúng ta có một mảng được khai báo với số phần tử tối đa được định nghĩa trước. Ví dụ:

```
#define MAX_SIZE = 100;
//.........
int32_t arr[MAX_SIZE];
```
Và N là số phần tử đang được sử dụng trong mảng `(0 < N < MAX_SIZE)`. Ví dụ:
```
int32_t N = 5;
```

Có người yêu cầu bạn thực hiện công việc chèn 1 giá trị số nguyên `insert_value` nào đó vào vị trí `insert_position` (với `insert_value` và `insert_position` là 2 giá trị được nhập từ bàn phím).

Tìm giải pháp

Chúng ta cùng thử tự đặt ra một số câu hỏi liên quan đến vấn đề trên và tự tìm ra câu trả lời để đưa ra giải pháp.

Điều gì xảy ra nếu 1 phần tử mới được chèn vào mảng?

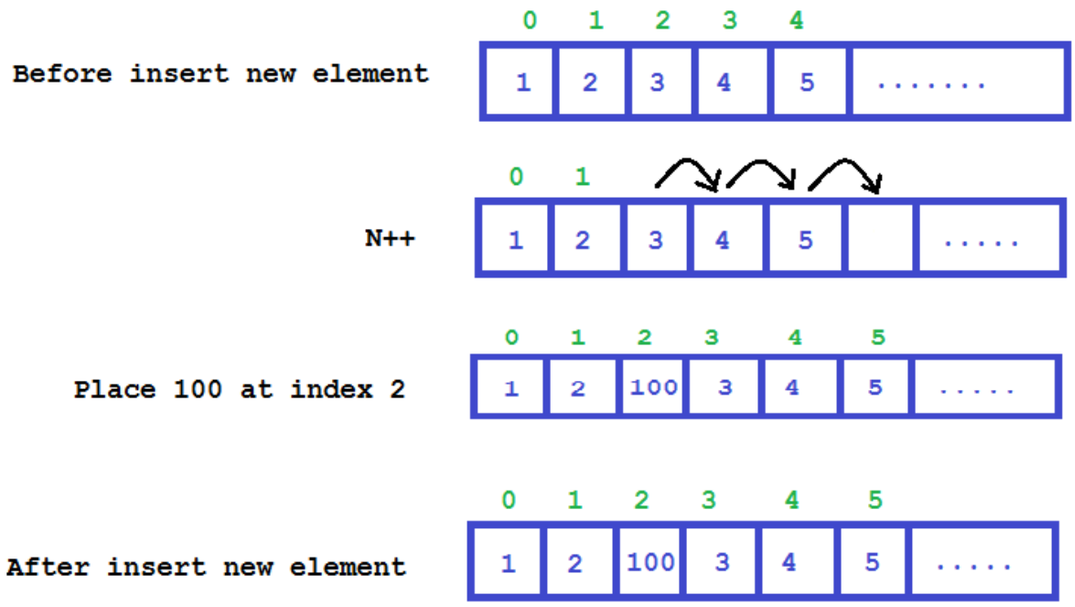
Điều đầu tiên dễ nhận thấy nhất là số lượng phần tử **N** hiện có trong mảng sẽ tăng lên 1. Vì thế, chúng ta cần tăng giá trị của biến **N** lên 1 để có thêm chỗ trống chứa phần tử mới được thêm vào.

```
N++;
```

Có một yêu cầu nhỏ khác là vị trí chèn phần tử `insert_position` sẽ phải nằm trong khoảng từ **0** đến **(N - 1)**, lúc đó, phần tử mới được chèn vào mới có chỉ số hợp lệ.
Ví dụ mảng ban đầu có 5 phần tử như sau:

```
arr[0] = 1;
arr[1] = 2;
arr[2] = 3;
arr[3] = 4;
arr[4] = 5;
```

Nếu phần tử cần chèn có giá trị 100 và vị trí chèn là phần tử có chỉ số 2. Mảng kết quả sau khi chèn sẽ là gì?



2.png?raw=true833x493

Phần tử đầu tiên của mảng gắn liền với địa chỉ ô nhớ đầu tiên mà hệ điều hành cấp phát cho mảng, vì thế, các phần tử có chỉ số nhỏ hơn vị trí cần chèn không thể thay đổi vị trí. Cách duy nhất là đẩy tất cả các phần tử có chỉ số từ vị trí cần chèn lui sau 1 ô nhớ, và chúng ta sẽ đặt phần tử mới vào ví trí cần chèn.

Full source code:

```
#include <iostream>
#include <cstdint>
#include <typeinfo>
using namespace std;

#define MAX_SIZE 100

int main()
{
        //initialize array
        int32_t arr[MAX_SIZE];
        int32_t N = 5;
```

```
        for (int32_t index = 0; index <= N - 1; index++) {
                arr[index] = index + 1;
        }

        //input insert_value and insert_position from keyboard
        int32_t insert_value, insert_position;
        cout << "Enter insert_value: "; cin >> insert_value;
        cout << "Enter insert_position: "; cin >> insert_position;

        //inserting
        N++;
        for (int32_t i = N - 2; i >= insert_position; i--) {

                int32_t after_i = i + 1;
                arr[after_i] = arr[i];
        }
        arr[insert_position] = insert_value;

        //output array
        for (int32_t index = 0; index <= N - 1; index++)
                cout << arr[index] << " ";
        cout << endl;

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

Các bạn thử giải thích trong đoạn code thực hiện đẩy các phần tử đứng sau vị trí `insert_position` này lui sau một vị trí, tại sao mình cho chỉ số bắt đầu từ **(N - 2)** nhé.

```
// why (N - 2)?
for (int32_t i = N - 2; i >= insert_position; i--) {

        int32_t after_i = i + 1;
        arr[after_i] = arr[i];
}
```

Xóa một phần tử có giá trị nào đó trong mảng một chiều

Việc thực hiện xóa 1 phần tử có giá trị `delete_value` nào đó đơn giản hơn việc chèn 1 phần tử mới vào mảng. Chúng ta chỉ cần làm ngược lại công đoạn chèn phần tử.
Giả sử chúng ta có mảng một chiều được khai báo và khởi tạo như sau:

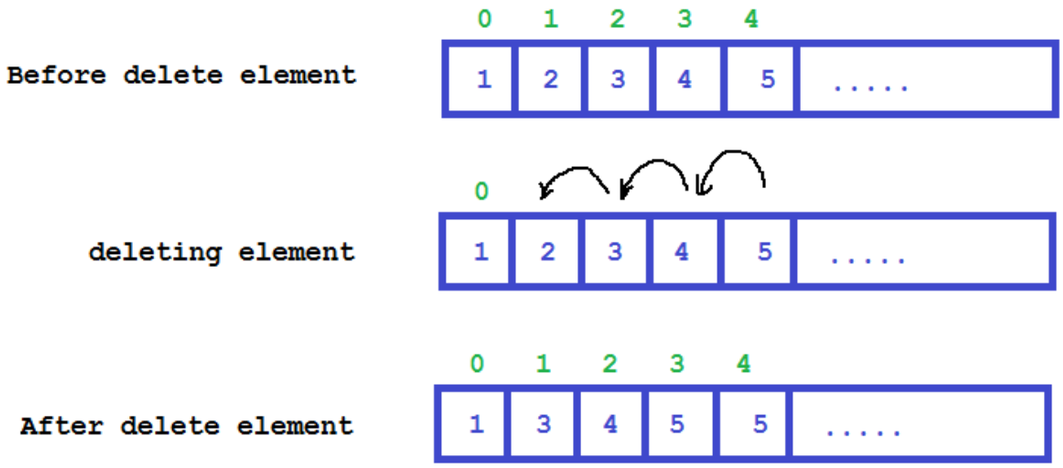
```
//initialize array
int32_t arr[MAX_SIZE]; //MAX_SIZE = 100
int32_t N = 5;

for (int32_t index = 0; index <= N - 1; index++) {
        arr[index] = index + 1;
}
```

Chúng ta phải tìm vị trí cần xóa trước đã. Phương pháp tìm kiếm phần tử trong mảng một chiều đã được mình trình bày ở phần trên, nhưng trong trường hợp tìm kiếm này, chúng ta có một chút thay đổi. Kết quả nhận được sau khi tìm kiếm không còn là giá trị **đúng/sai** nữa, mà là chỉ số của phần tử cần được xóa (nếu tìm thấy).

```
//input delete_value
int32_t delete_value;
cout << "Enter delete_value: "; cin >> delete_value;

//finding the delete_position
int32_t delete_position = -1;
for (int32_t index = 0; index <= N - 1; index++) {

        if (delete_value == arr[index]) {
                delete_position = index;
                break;
        }
}

//Check if program found the delete_value in arr
if (delete_position != -1) {

        //remove the element at index delete_position from arr
}
```

Sau khi tìm kiếm phần tử `delete_value` trong mảng `arr`, nếu biến `delete_position` bị thay đổi thì chúng ta hiểu rằng phần tử `delete_value` được tìm thấy. Việc còn lại chúng ta chỉ cần lấp những phần tử đứng sau vị trí `delete_position` lên trước 1 chỉ số thì phần tử tại vị trí `delete_position` sẽ bị ghi đè lên.

```
delete_value = 2
delete_position = 1;
```



3.png?raw=true833x461

Cuối cùng, chúng ta giảm số lượng phần tử **N** hiện có trong mảng đi 1.

```
//Check if program found the delete_value in arr
if (delete_position != -1) {

        for (int32_t i = delete_position + 1; i <= N - 1; i++) {

                int32_t before_i = i - 1;
                arr[before_i] = arr[i];
        }
        N--;
}

//output array
for (int32_t index = 0; index <= N - 1; index++)
        cout << arr[index] << " ";
cout << endl;
```

Sắp xếp mảng một chiều

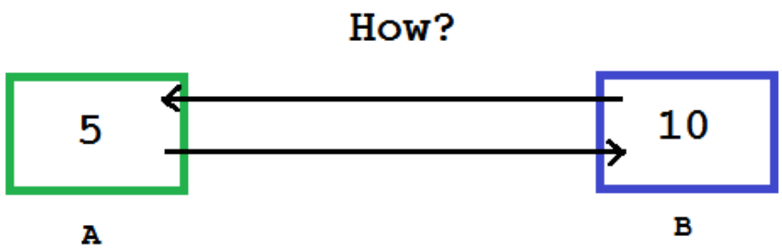
Ngày nay, chúng ta có rất nhiều cách sắp xếp các phần tử trong mảng một chiều theo thứ tự ***tăng/giảm*** dần. Trong bài học này, mình giới thiệu đến các bạn phương pháp **Selection sort** để sắp xếp mảng một chiều theo thứ tự tăng dần.

Selection sort có cách cài đặt và vận hành khá giống với việc sắp xếp mà con người chúng ta thường làm. Giả sử mình khởi tạo 1 mảng có 10 phần tử với các giá trị được khởi tạo có độ lớn giảm dần:

```
//initialize array
int32_t arr[MAX_SIZE];
int32_t N = 10;

for (int32_t i = 0; i <= N - 1; i++) {
        arr[i] = N - i;
}
//10 9 8 7 6 5 4 3 2 1
```

Công việc của chúng ta là sử dụng thuật toán **Selection sort** để hoán vị các phần tử trong mảng theo cách nào đó để kết quả ta thu được là mảng `arr` có giá trị tăng dần: 1 2 3 4 5 6 7 8 9 10.

Làm thế nào để hoán vị giá trị của hai biến có cùng kiểu dữ liệu?

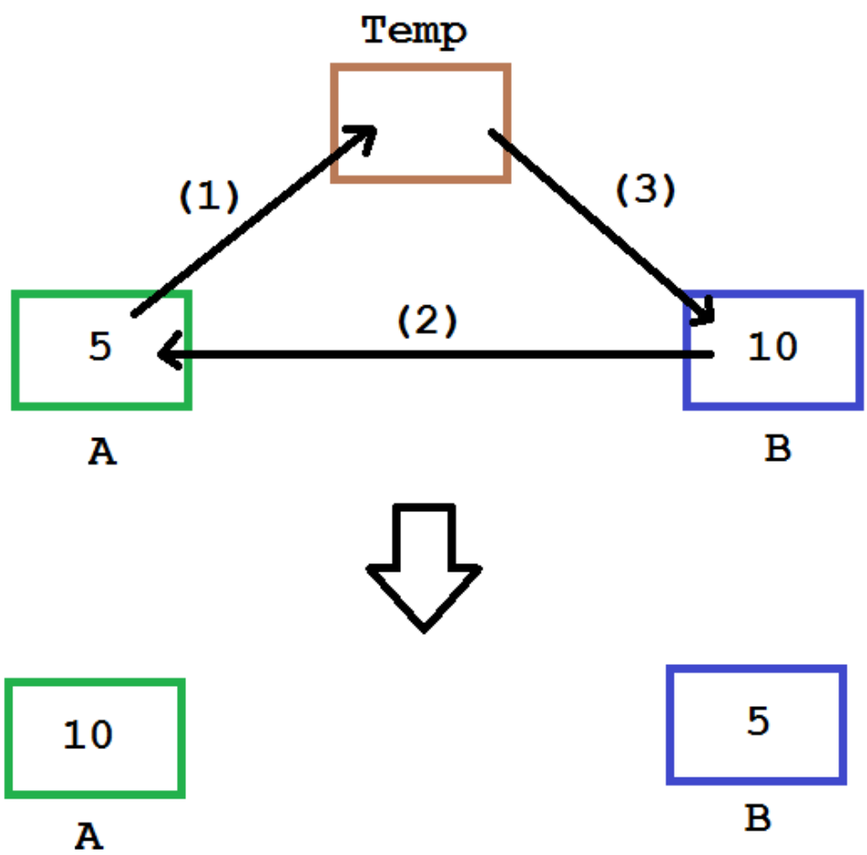


4.png?raw=true746x287

Hoán vị giá trị của hai biến là trao đổi giá trị của hai biến đó. Ví dụ:

```
//Before swap value
int32_t a = 5;
int32_t b = 10;

//After swap value
a = 10;
b= 5;
```

Có nhiều cách để hoán vị giá trị hai biến, mình sẽ đưa ra một cách sử dụng phổ biến nhất, đó là dùng thêm 1 biến để tạm lưu trữ giá trị của một trong hai biến cần hoán vị.



5.png?raw=true746x544

Bằng cách sử dụng thêm một biến `Temp` để lưu trữ một trong hai giá trị của biến (A hoặc B), chúng ta có thể dễ dàng thực hiện hoán vị qua 3 bước.

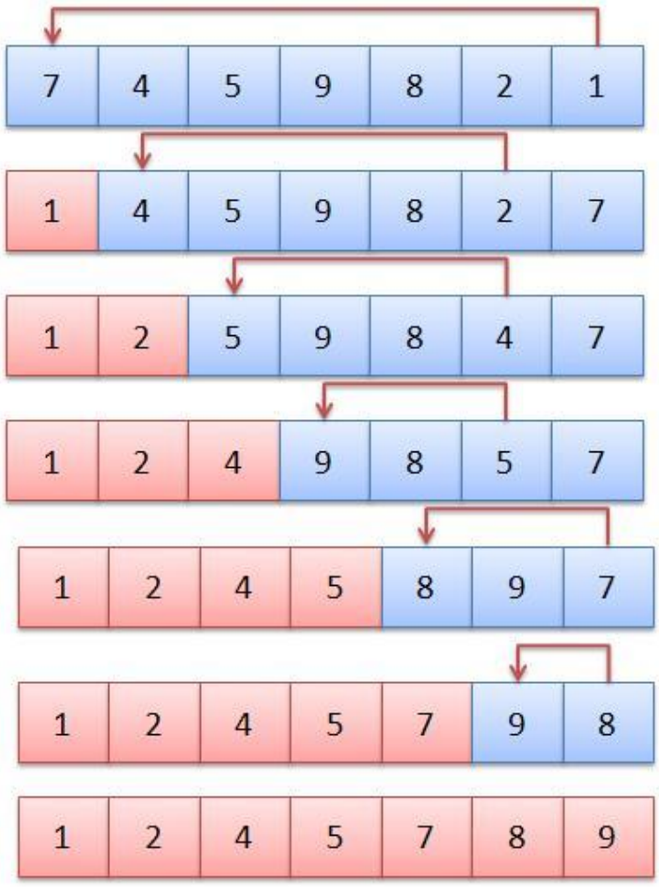
```
int32_t temp = a;
a = b;
b = temp; //b = old_value_of_a
```

Chúng ta sẽ thực hiện hoán vị các phần tử trong mảng một chiều để đưa mảng một chiều về dạng tăng dần trong bài học này.

Thuật toán selection sort

Tư tưởng của thuật toán này là chia mảng thành hai phần, phần đã được sắp xếp có chỉ số thấp, phần chưa được sắp xếp là những phần tử có chỉ số đứng sau chỉ số của phần tử cuối cùng đã được sắp xếp.

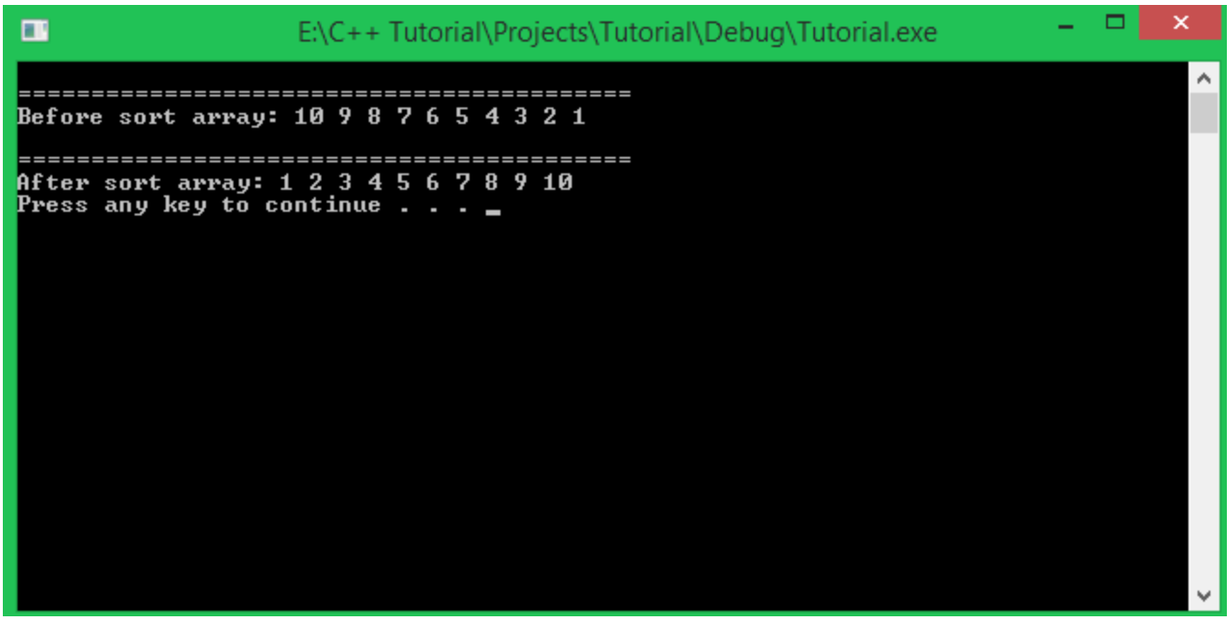
Khi chúng ta cần sắp xếp mảng một chiều theo thứ tự tăng dần, chúng ta sẽ tìm trong phần mảng chưa được sắp xếp ra một phần tử nhỏ nhất, và hoán vị với phần tử đứng sau chỉ số cuối cùng của phần đã được sắp xếp.



6.png?raw=true432x527

Dưới đây là phần code cho việc sắp xếp mảng một chiều bằng thuật toán **selection sort**:

```
//sorting
for (int32_t after_sorted_part = 0; after_sorted_part <= N - 1; after_sorted_part++) {

        int32_t min_index = after_sorted_part;
        for (int32_t find_min_index = after_sorted_part; find_min_index <= N - 1;
find_min_index++) {

                if (arr[find_min_index] < arr[min_index]) {
                        min_index = find_min_index;
                }
        }

        //swap value of arr[min_index] and arr[after_sorted_part]
        int32_t temp = arr[min_index];
        arr[min_index] = arr[after_sorted_part];
        arr[after_sorted_part] = temp;
}
```



Tổng kết

Qua bài học này, hi vọng các bạn đã có thể tự mình hình thành tư duy các bài toán liên quan đến mảng một chiều với các thao tác xử lý mảng một chiều mà mình vừa đưa ra.

Bài tập cơ bản

1/ Cho mảng một chiều như sau:

```
int32_t arr[] = { 2, 6, 5, 7, 9, 1, 3 };
```

Viết chương trình đảo ngược mảng trên. Mảng kết quả sau khi thực hiện đảo ngược là.

```
3 1 9 7 5 6 2
```

2/ Viết chương trình nhập vào một dãy các số nguyên từ bàn phím và lưu vào mảng một chiều, so sánh tổng các phần tử chẵn với tổng các phần tử lẻ và đưa ra màn hình kết quả.

3/ Viết chương trình in ra tất cả các phần tử của mảng nhưng bỏ qua các giá trị bị trùng lặp. Ví dụ với mảng một chiều như sau:

```
4 6 2 2 1 6 9
```

Kết quả in ra màn hình sẽ là:

```
4 6 2 1 9
```

5.2 Thư viện array trong STL

Đến với bài học ngày hôm nay, chúng ta sẽ tiếp tục làm việc với kiểu dữ liệu mảng một chiều, nhưng chúng ta sẽ sử dụng thư viện **array** trong **namespace std**.

Đây là một thư viện giúp chúng ta sử dụng mảng một chiều một cách hiệu quả, rõ ràng hơn, ngoài ra nó còn giúp chúng ta hạn chế được lỗi thường gặp như truy cập đến chỉ số vượt ngoài giới hạn số phần tử đang sử dụng.

Thư viện array

Để sử dụng thư viện **array**, các bạn chỉ cần include thư viện này như sau:

```
#include <array>
using namespace std;
```

Chúng ta cũng cần có thêm dòng khai báo **namespace std** vì thư viện **array** được định nghĩa bên trong nó.

Thư viện **array** cung cấp cho chúng ta kiểu dữ liệu **array**, biến được tạo ra bởi kiểu dữ liệu này chỉ là một biến đơn, nhưng vùng nhớ mà nó quản lý sẽ tương đương với số lượng phần tử tối đa mà chúng ta khai báo từ trước (gần giống như mảng một chiều).

Đối tượng được tạo ra bởi lớp **array** chỉ cung cấp cho chúng ta một vùng nhớ để lưu trữ một số lượng phần tử xác định trước, nhưng thông qua một số phương thức được định nghĩa bên trong lớp **array** này, chúng ta còn có thể truy xuất một số thông tin liên quan như số lượng phần tử, kiểm tra mảng có rỗng hay không, ...

Khai báo biến với kiểu dữ liệu array

Một đối tượng có kiểu array khi được khai báo cần xác định được 2 điều:

- Kiểu dữ liệu của các phần tử mà biến array sẽ chứa.
- Số lượng phần tử tối đa của mảng.

Cú pháp khai báo biến kiểu **array**:

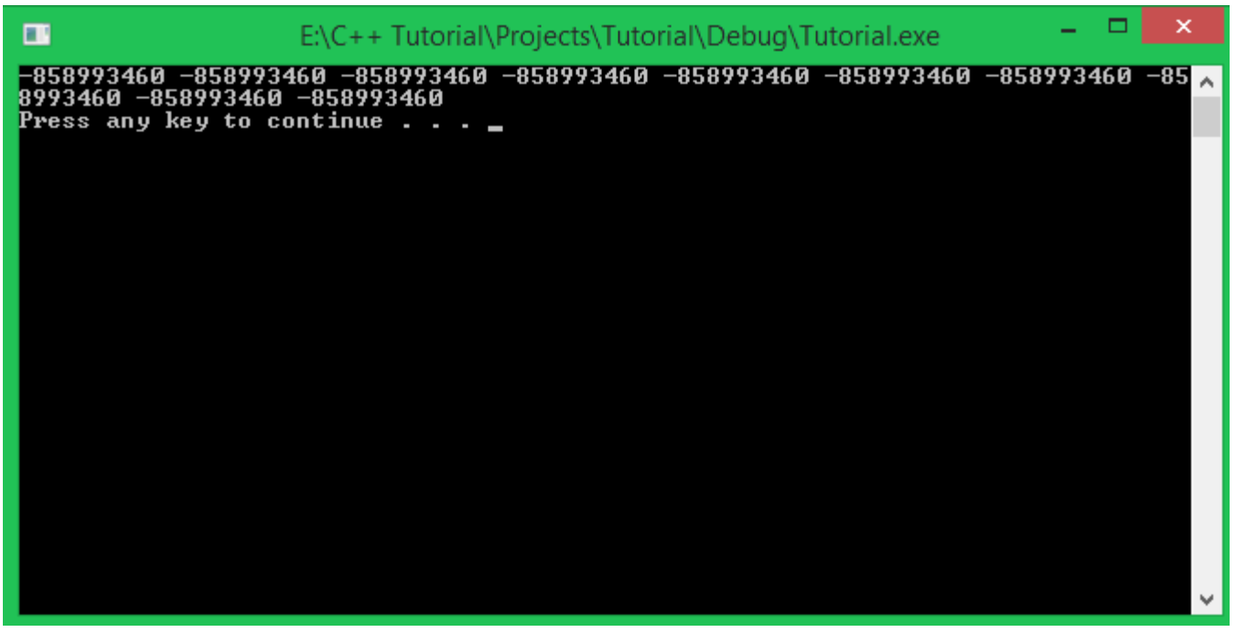
```
array< <data_type>, <number_of_elements> > <array_name>;
```

Ví dụ chúng ta cần sử dụng một mảng kiểu **int32_t** có 10 phần tử, chúng ta khai báo như sau:

```
array<int32_t, 10> arr;
```

Khởi tạo giá trị

Khi chưa khởi tạo giá trị cho biến kiểu **array**, chúng ta sẽ nhận được những giá trị không có ý nghĩa khi in chúng ra màn hình. Ví dụ với mảng **arr** trên:



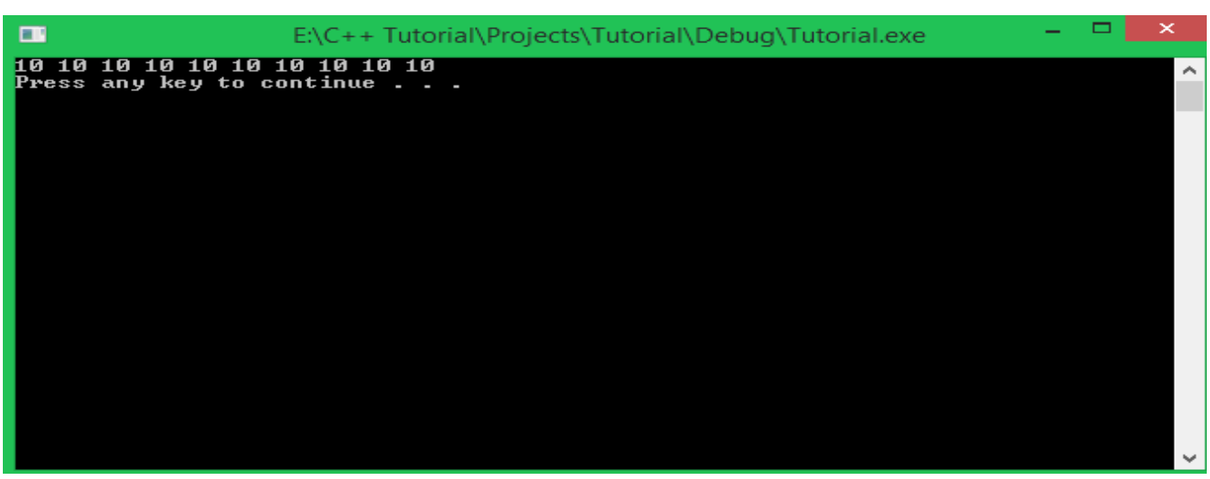
Chúng ta có thể khởi tạo giá trị cho toàn bộ phần tử trong mảng chỉ với 1 dòng lệnh:

```
<array_name>.assign(<value>);
```

Giả sử mình muốn gán giá trị 10 cho toàn bộ phần tử trong mảng **arr**, mình viết như sau:

```
arr.assign(10);
```

Kết quả của việc in mảng arr ra màn hình sau khi khởi tạo:



Truy xuất đến các thành phần trong biến có kiểu array

Chúng ta có thể truy cập đến một phần tử của đối tượng của lớp **array** bằng toán tử [ ] như lúc các bạn sử dụng mảng một chiều. Ví dụ:

```
arr[1];
```

Hoặc các bạn có thể sử dụng phương thức **at(<index>)** được định nghĩa trong lớp **array** như sau:

```
arr.at(1);
```

Sử dụng **arr[1]** và **arr.at(1)** đều trả về kết quả là giá trị của phần tử thứ 2 trong mảng.

Truy xuất một số thông tin bên trong đối tượng của lớp array

Chúng ta có thể truy xuất một vài thông tin liên quan đến mảng một chiều bằng một số phương thức bên trong đối tượng của lớp **array**.

- Xem số lượng phần tử mà đối tượng của lớp **array** có thể chứa:

  ```
  cout << "Number of elements: " << arr.size() << endl;
  ```

  Phương thức size() trả về số lượng phần tử mà bạn đã khai báo lúc tạo ra đối tượng của class **array**.

- Kiểm tra xem mảng một chiều được chứa bên trong đối tượng của lớp **array** có rỗng hay không:

  Mảng một chiều rỗng nghĩa là số lượng phần tử bằng 0.

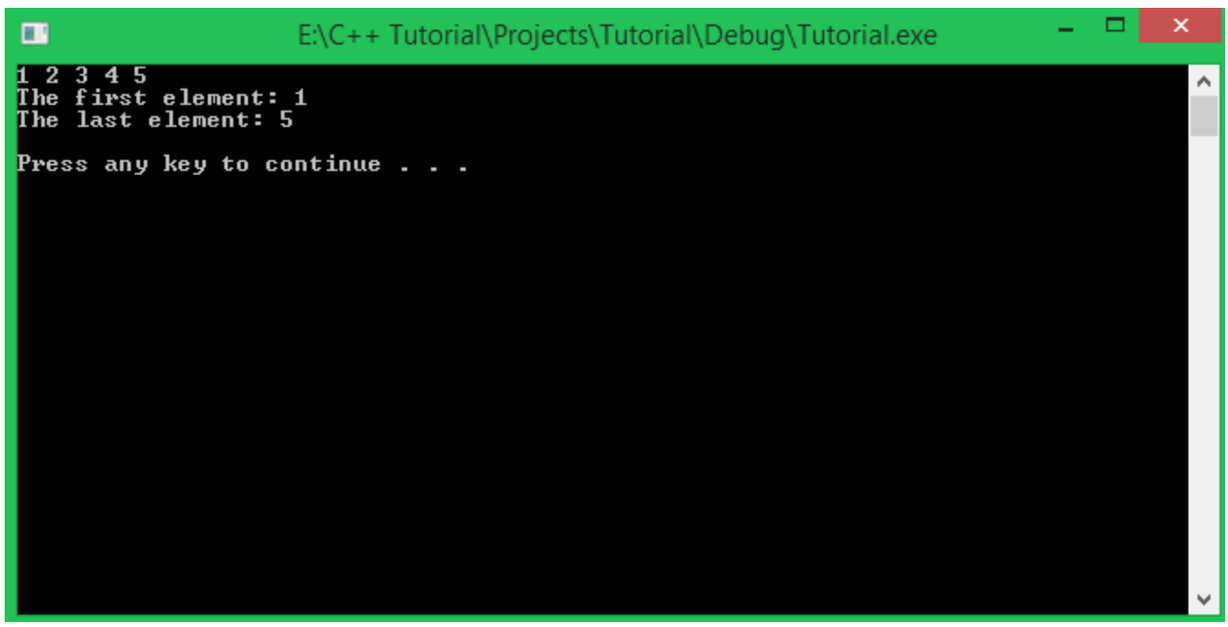
  ```
  if(arr.empty())
    cout << "Array is empty." << endl;
  else
    cout << "Number of elements: " << arr.size() << endl;
  ```

  Phương thức **empty()** trả về giá trị **true** nếu mảng bên trong đối tượng **arr** có số lượng phần tử là 0.

- Truy xuất đến phần tử đầu tiên và phần tử cuối cùng của mảng bên trong đối tượng của lớp **array**:

```
cout << "The first element: " << arr.front() << endl;
cout << "The last element: " << arr.back() << endl;
```

  Ví dụ mảng một chiều của mình được khởi tạo giá trị là 1 2 3 4 5. Kết quả in ra màn hình sẽ là:



  Phương thức **front()** sẽ trả về giá trị của phần tử đầu tiên trong mảng, ngược lại, phương thức **back()** sẽ trả về giá trị của phần tử cuối cùng trong mảng.

Nhập dữ liệu cho đối tượng của lớp array

Tương tự lúc các bạn nhập dữ liệu cho mảng một chiều thông thường, chúng ta sử dụng đối tượng **cin** để đưa giá trị được nhập từ bàn phím vào trong mỗi phần tử mà đối tượng của lớp **array** đang nắm giữ.

```
for (int i = 0; i < arr.size(); i++) {

        cout << "Enter value to element " << i + 1 << ": ";
        cin >> arr[i];
}
```

Lớp array ngăn chặn hành vi truy cập phần tử có chỉ số không phù hợp

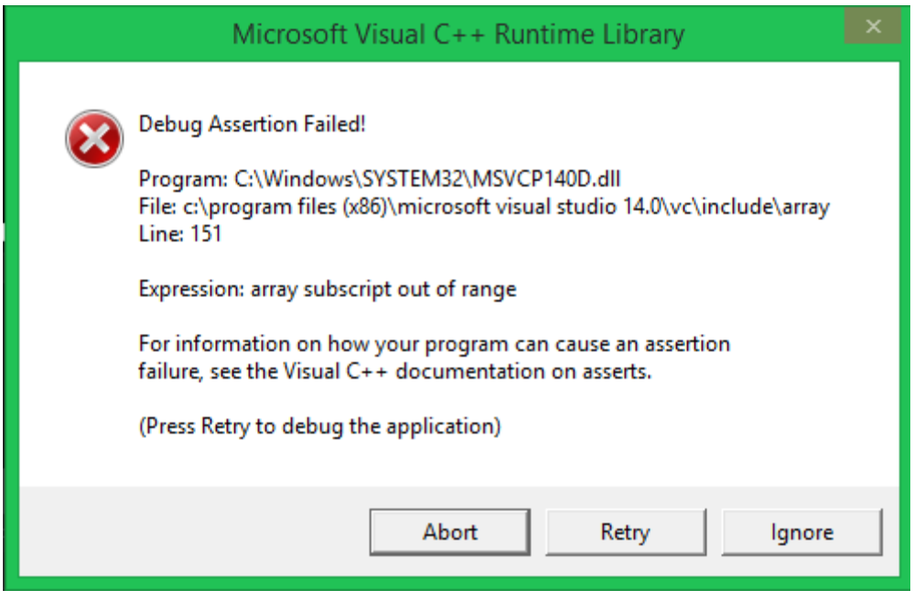
Chúng ta chỉ có thể truy xuất đến các phần tử trong đối tượng của lớp **array** với chỉ số trong phạm vi từ `0` đến `(size() - 1)`. Sau đây là những hành vi truy xuất hợp lệ:

```
#define ARRAY_SIZE 10
array<int32_t, ARRAY_SIZE> arr;
arr.assign(10);

//Access to all of elements of arr object
for (int32_t index = 0; index <= arr.size() - 1; index++) {
        cout << arr[index] << " ";
}
cout << endl;
```

Và dưới đây là một số hành vi truy xuất giá trị của đối tượng arr bằng những chỉ số không hợp lệ:

```
//Try to access array with wrong index
arr[-1];
arr[arr.size() + 10];
```

Khi gặp những dòng lệnh này, **compiler** sẽ đưa ra cảnh báo:



Vì bên trong lớp **array** có sử dụng thư viện **cassert** để đặt ra những **Assertion**, những Assertion này kiểm tra về chỉ số mà bạn đưa vào cho toán tử **[ ]** và phương thức **at()** để kiểm tra sự hợp lệ của chỉ số trước khi thực hiện lệnh. Mọi hành vi không phù hợp với điều kiện trong **Assertion** sẽ bị ngăn chặn.

Các bạn cũng có thể tự mình tạo ra những **Assertion** bằng cách sử dụng thư viện **cassert**.

Thư viện cassert

Thư viện **cassert** cung cấp cho chúng ta macro có tên là `assert(expression)` giúp chúng ta tạo ra những **Assertion** trong chương trình.
Khi gặp macro **assert(expression)**, chương trình sẽ kiểm tra biểu thức *expression* (là một biểu thức điều kiện có thể trả về giá trị **true/false**) và có hai trường hợp có thể xảy ra:

- **expression** trả về giá trị **true**:

  Chương trình sẽ tiếp tục thực hiện các dòng lệnh phía sau **Assertion** một cách bình thường. Ví dụ:

  ```
  float f_value = 1.0;
  assert(typeid(f_value) == typeid(float));

  f_value++;
  cout << f_value << endl;
  ```

Đoạn chương trình trên có 1 **Assertion** thực hiện công việc kiểm tra kiểu dữ liệu của biến **f_value**. Vì biểu thức `typeid(f_value) == typeid(float)` trả về giá trị **true**, nên chương trình vẫn được tiếp tục hoạt động.

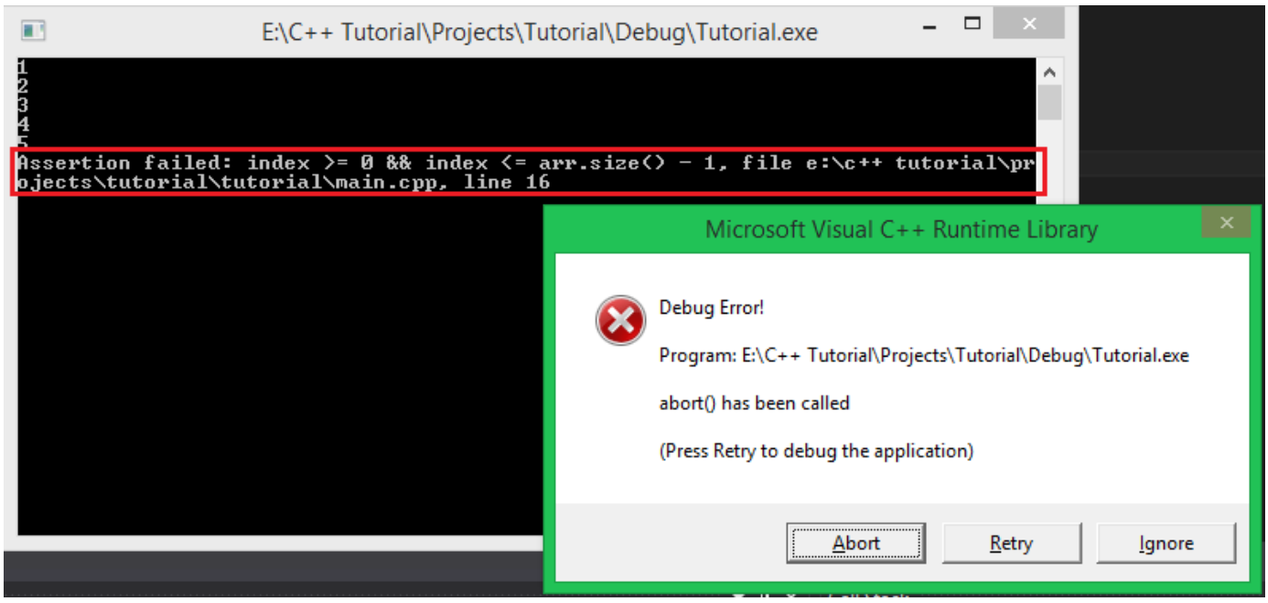
- **expression** trả về giá trị **false**:

  Chương trình sẽ dừng lại tại thời điểm phát hiện biểu thức bên trong **Assertion** cho giá trị **false**.

```
#define ARRAY_SIZE 5
array<int32_t, ARRAY_SIZE> arr;

for(int32_t index = 0; index <= arr.size(); index++)        {

  assert(index >= 0 && index <= arr.size() - 1);
  cin >> arr[index];
}
```

Đoạn chương trình trên thực hiện nhập dữ liệu cho đối tượng **arr** có kiểu **array**. Bên trong vòng lặp **for**, mình đặt 1 **Assertion** nhằm kiểm tra chỉ số của mảng có được cung cấp chính xác hay không. Chỉ số chính xác sẽ nằm trong khoảng từ `0` đến `(arr.size() - 1)`.

Bây giờ mình sẽ chạy đoạn chương trình trên để xem kết quả:



4.png?raw=true793x372

Ngoài việc chương trình đưa ra cửa sổ thông báo lỗi và bắt các bạn Abort chương trình đang chạy, trên cửa sổ console còn đưa ra thông báo lỗi tại dòng mình đặt **Assertion**.

Trước khi lỗi xảy ra, mình vẫn nhập dữ liệu bình thường. Vì lúc đó chỉ số index của vòng lặp **for** vẫn thõa mãn biểu thức điều kiện bên trong **Assertion**. Nhưng mà vòng lặp for của mình lại lặp với biến index chạy từ 0 đến **arr.size()**, vì thế, giá trị index tại lần lặp cuối cùng đã vi phạm biểu thức trong **Assertion** mà mình tự đặt ra.

Tổng kết

Trong bài học hôm nay, các bạn đã được tìm hiểu thêm thư viện **array** hỗ trợ cho các bạn quản lý mảng một chiều một cách hiệu quả và dễ dàng hơn. Mình cũng đã hướng dẫn cho các bạn cách để tạo ra những **Assertion** cho chương trình của các bạn với thư viện **cassert**. Bất cứ khi nào các bạn cần đảm bảo chương trình của các bạn không vi phạm quy tắc nào đó, các bạn có thể dùng macro **assert(expression)** của thư viện **cassert** để hạn chế những lỗi có thể xảy ra.

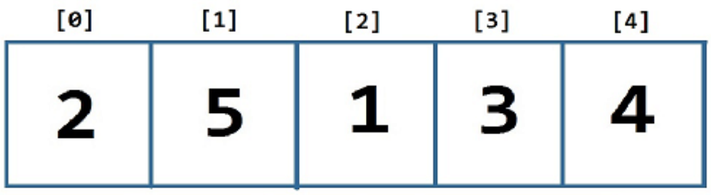
5.3 Mảng hai chiều

Trong các bài học trước, mình đã giới thiệu đến các bạn về mảng một chiều trong ngôn ngữ C/C++.

Mảng một chiều có thể được hiểu là một dãy các phần tử ***có cùng kiểu dữ liệu*** được đặt liên tiếp nhau trong một vùng nhớ, chúng ta có thể ngay lập tức truy xuất đến một phần tử của dãy đó thông qua chỉ số của mỗi phần tử.
Bây giờ các bạn thử tưởng tượng nếu kiểu dữ liệu của mảng một chiều là mảng một chiều? Hay nói cách khác, chúng ta có một mảng chứa các mảng một chiều? Lúc này, chúng trở thành **mảng 2 chiều**.

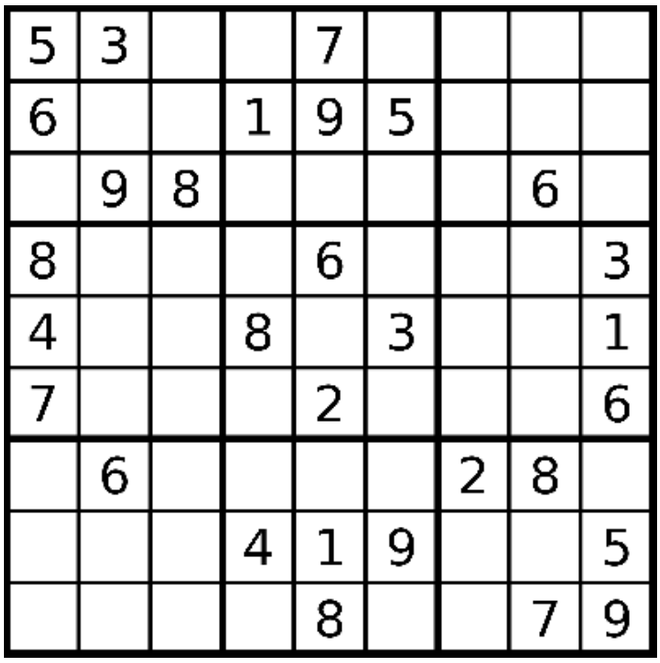
2D Array

Trước hết, mình cho các bạn xem lại hình ảnh minh họa cho **mảng một chiều** trên máy tính:



Đây là **mảng 1 chiều** gồm có 5 phần tử được đánh chỉ số từ 0 đến 4.

Và dưới đây là hình ảnh minh họa cho cách tổ chức dữ liệu **mảng hai chiều**:



Đây là bảng câu đố của game **Sudoku** được tạo thành từ 9x9 ô vuông (9 dòng và 9 cột). Giả sử mình tách dòng đầu tiên của bảng game này ra đứng riêng biệt:

Nó lại trở thành mảng **1 chiều** có 9 phần tử.

Vậy, **mảng một chiều** khi mô phỏng nó bằng hình ảnh, chúng ta chỉ thấy được 1 hàng ngang có nhiều cột phân chia thành các ô (tượng trưng cho các ô nhớ trong máy tính). Còn khi chúng ta nhìn vào **mảng hai chiều**, chúng ta thấy có nhiều hàng, mỗi hàng lại có nhiều cột, đặc biệt hơn là số lượng cột ở mỗi hàng đều bằng nhau.

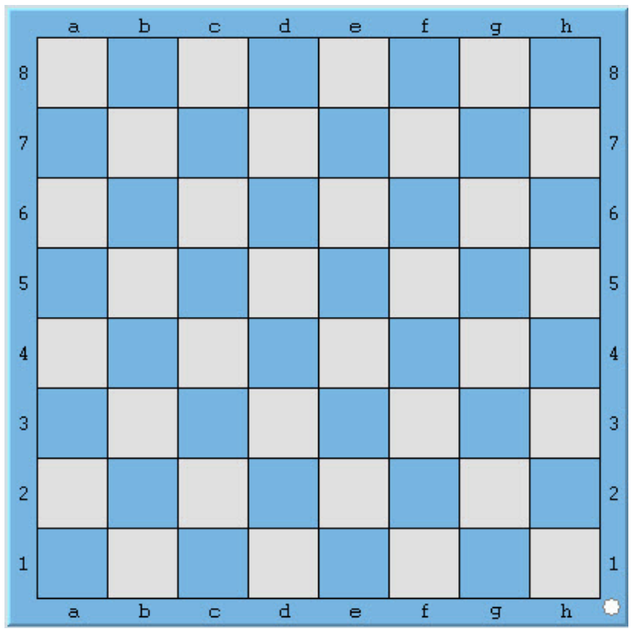
Ngôn ngữ C/C++ có hỗ trợ cho chúng ta tổ chức dữ liệu theo dạng bảng như trên, hay thường gọi là **mảng hai chiều**. Thế thì khi nào chúng ta cần sử dụng mảng hai chiều trong chương trình máy

tính? Trong thực tế, chúng ta gặp rất nhiều thứ được bố trí dưới dạng mảng 2 chiều. Dưới đây là một số ví dụ thực tế:

- Phòng học:

Như hình minh họa, chúng ta có một phòng học có 2 dãy bàn hàng ngang, mỗi dãy bàn ngang có thể đủ chỗ cho 3 sinh viên. Như vậy mình gọi đây là mảng hai chiều 2x3 (2 hàng, 3 cột).

- Bàn cờ vua:



Bàn cờ vua là một bảng hình vuông có 8 hàng, mỗi hàng có 8 cột, tổng cộng có 64 ô vuông, mỗi ô có thể đặt 1 quân cờ. Chúng ta có thể gọi đây là một mảng hai chiều 8x8 (8 dòng, 8 cột).

- Trò chơi Tic Tac Toe:

Trò chơi này được chơi trên một bảng 3x3 (3 hàng, 3 cột). Nếu trò chơi này được mô phỏng trên máy tính, chúng ta có thể sử dụng một mảng hai chiều 3x3 để lưu trữ các kí tự **'x'** hoặc **'o'**.

Qua một số hình ảnh minh họa như trên, hi vọng các bạn đã có thể hình dung được mảng hai chiều là như thế nào. Bây giờ mình sẽ đi vào chi tiết về cách khai báo, khởi tạo giá trị và cách sử dụng mảng hai chiều trong ngôn ngữ C++.

Khai báo mảng hai chiều

Đối với mảng một chiều, chúng ta chỉ cần khai báo số lượng phần tử (số lượng cột) cho một hàng duy nhất, do đó, khai báo mảng một chiều có dạng:

```
<data_type> <name_of_array>[num_of_columns];
```

Ví dụ:

```
int iArray[100]; //declare an array of integer can hold 100 elements
```

Bây giờ, khi quản lý mảng hai chiều, chúng ta còn phải quan tâm thêm về số hàng mà mảng hai chiều cần cấp phát:

```
<data_type> <name_of_array>[num_of_rows][num_of_columns];
```

Lưu ý, khi khai báo số lượng phần tử của mảng hai chiều, số hàng phải đặt trước số cột.

Ví dụ:

```
int array2D[3][5]; // 3x5 elements (3 rows, 5 columns)
```

Có thể nói cách khác, mảng có tên **array2D** có kiểu dữ liệu **int**, mảng **array2D** gồm có 3 mảng một chiều, mỗi mảng một chiều trong đó có thể chứa được tối đa 5 phần tử.

Khởi tạo mảng hai chiều

Mình lấy lại ví dụ về mảng có tên **array2D** như trên, mình sẽ khởi tạo giá trị cho mảng như sau:

```
int array2D[3][5] =
{
        { 1,  2,  3,  4,  5 },  //row 1
        { 6,  7,  8,  9,  10 }, //row 2
        { 11, 12, 13, 14, 15 }  //row 3
};
```

Do mảng **array2D** có 3 hàng, mỗi hàng lại là một mảng một chiều khác nhau, nên mình đã sử dụng cách khởi tạo của mảng một chiều, áp dụng cho mỗi hàng trong mảng hai chiều **array2D**.

Các bạn có thể khởi tạo mảng hai chiều theo cách sau:

```
int array2D[3][5] =
{
        1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
};
```

Nhưng mình vẫn khuyến khích các bạn sử dụng cách mình trình bày ở trước để tránh nhầm lẫn trong việc tổ chức dữ liệu.

Những phần tử chưa được khởi tạo giá trị sẽ được gán bằng giá trị mặc định tùy vào mỗi kiểu dữ liệu khác nhau. Như ví dụ sau mình sử dụng kiểu **int** để khai báo mảng hai chiều:

```
int seats[3][5] =
{
        { 1, 2 },               //row 1 = 1, 2, 0, 0, 0
        { 6, 7, 8 },     //row 2 = 6, 7, 8, 0, 0
        { 11 },                 //row 3 = 11, 0, 0, 0, 0
};
```

Tương tự mảng một chiều, nếu các bạn khởi tạo mảng hai chiều ngay khi khai báo, compiler có thể tự xác định số hàng cần cấp phát:

```
int array2D[][4] =
{
        { 1, 2, 3, 4 },
        { 5, 6, 7, 8 }
};
```

Các bạn có thể bỏ trống phần khai báo số lượng hàng, nhưng không thể không khai báo số lượng cột.

Truy cập các phần tử trong mảng hai chiều

Lấy ví dụ mình có một mảng hai chiều có 3 hàng và 4 cột tạo thành bảng như sau:

```
int board[3][4];
```

	Column 1	Column 2	Column 3	Column 4
Row 1	[0][0]	[0][1]	[0][2]	[0][3]
Row 2	[1][0]	[1][1]	[1][2]	[1][3]
Row 3	[2][0]	[2][1]	[2][2]	[2][3]

Để xác định tọa độ (ví trị) của một phần tử trong một mảng hai chiều, chúng ta cần xác định hai tham số là chỉ số dòng và chỉ số cột. Chúng ta truy cập vào chỉ số dòng trước và chỉ số cột sau. Ví dụ:

```
board[1][2]; //Access element on row 2 and column 3
```

Thực hiện truy cập mảng board với chỉ số dòng là 1 và chỉ số cột là 2 sẽ trỏ đến ô nhớ tại dòng thứ 2 và cột thứ 3, do chỉ số của mảng sẽ bắt đầu từ 0. Tương tự, để truy cập phần tử của cùng của mảng hai chiều 3x4, chúng ta truy cập với chỉ số (2, 3).

Để truy cập toàn bộ mảng hai chiều, chúng ta có thể sử dụng 2 vòng lặp: vòng lặp ngoài sẽ truy cập lần lượt các dòng, vòng lặp bên trong sẽ truy cập tất cả các cột của dòng hiện tại mà vòng lặp ngoài đang truy cập đến.

```
int board[3][4] =
{
        { 1, 1, 1, 1 },
        { 2, 2, 2, 2 },
        { 3, 3, 3, 3}
};

for(int row = 0; row < 3; row++)
{
        for(int col = 0; col < 4; col++)
        {
                cout << board[row][col] << " ";
        }
        cout << endl;
}
```

Nhập dữ liệu cho mảng hai chiều

Cũng tương tự việc các bạn nhập dữ liệu cho mảng một chiều, chúng ta sử dụng đối tượng **cin** trong thư viện **iostream**. Các bạn chỉ cần lưu ý rằng khi thao tác với các phần tử trong mảng hai chiều, chúng ta phải cung cấp đủ 2 chỉ số (hàng và cột) thì mới xác định được địa chỉ phần tử mà chúng ta cần thao tác.

```
cin >> <name_of_array>[row_index][col_index];
```

Trong đó, `row_index` là chỉ số dòng của phần tử, `col_index` là chỉ số cột của phần tử.
Ví dụ:

```
int board[3][3];

for(int row = 0; row < 3; row++)
{
        for(int col = 0; col < 3; col++)
        {
                cin >> board[row][col];
        }
}
```

Tổng kết

Trong bài học này, chúng ta đã cùng tìm hiểu về một cách tổ chức dữ liệu mới trên máy tính. Mảng hai chiều được sử dụng khá phổ biến để giải quyết một số thuật toán yêu cầu tối ưu như Quy Hoạch Động, bài toán đồ thị, ... Cũng có thể được sử dụng trong việc thiết kế một số trò chơi đơn giản, ví dụ game Minesweeper. Chúng ta sẽ còn ứng dụng nhiều về mảng hai chiều trong các bài học sau.

Bài tập cơ bản

1/ Viết chương trình nhập dữ liệu cho mảng hai chiều có số dòng, số cột dương (tùy ý bạn). In ra màn hình kết quả là tổng của mỗi dòng trong mảng hai chiều bạn vừa nhập.

Ví dụ mình nhập mảng hai chiều 3x3 như sau:

```
1 3 4
2 1 6
3 3 5
```

Kết quả in ra màn hình sẽ là:

```
8
9
11
```

Trong đó, 8 là tổng các giá trị trong dòng đầu tiên, 9 là tổng các giá trị của dòng thứ 2, 11 là tổng các giá trị của dòng thứ 3.

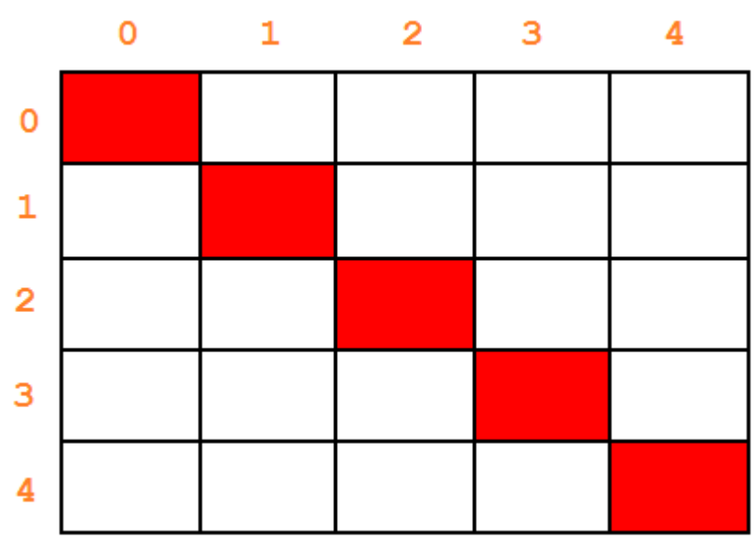
2/ Viết chương trình tìm kiếm sự xuất hiện của giá trị X nhập từ bàn phím trong mảng hai chiều.

5.4 Các thao tác cơ bản với mảng hai chiều

Trong bài học này, mình sẽ hướng dẫn các bạn thực hiện một số thao tác cơ bản với mảng hai chiều, cũng có thể coi đây là giải một số bài tập mẫu cơ bản, giúp các bạn hình thành tư duy giải các bài toán có thể giải quyết được bằng mảng hai chiều cơ bản.

Tính tổng các phần tử trên đường chéo chính

Trường hợp mảng hai chiều có đường chéo chính và đường chéo phụ chỉ tồn tại khi số hàng bằng số cột (có nghĩa là ma trận vuông). Khi đó, đường chéo chính có dạng:



Đặc điểm của các phần tử nằm trên đường chéo chính của ma trận vuông là chỉ số hàng luôn bằng chỉ số cột.

```
{ a[i][i] | 0 <= i <= n-1 }
```

Giả sử số hàng (hoặc số cột) của ma trận vuông này là N, chúng ta chỉ cần sử dụng vòng lặp for để lặp từ giá trị 0 đến N-1, cứ mỗi lần lặp với biến vòng lặp index, chúng ta cộng dồn giá trị của phần tử `Array[index][index]` vào biến tổng nào đó.

```
int main()
{
	int myArr[100][100];
	int level;
	cout << "Enter level of squared matrix: ";
	cin >> level;

	//input
	for (int row = 0; row < level; row++)
	{
		for (int col = 0; col < level; col++)
		{
			cin >> myArr[row][col];
		}
	}

	//calculate
	int sum = 0;
	for (int index = 0; index < level; index++)
	{
		sum += myArr[index][index];
	}

	//output
	cout << "Result: " << sum << endl;

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

Trong chương trình trên, mảng hai chiều **myArr** chưa được khởi tạo khi khai báo, nên mình phải cung cấp thông tin số hàng và số cột cụ thể cho compiler.

Xóa một dòng trong mảng hai chiều

Về phần input, chúng ta nhập dữ liệu bao gồm số hàng, số cột và giá trị của mỗi phần tử trong mảng hai chiều.

Trong phần xử lý, chúng ta cần nhập số dòng cần loại bỏ khỏi mảng hai chiều. Mình chưa thiết kế phần xử lý trường hợp nhập sai số dòng. Sau đó, tương tự việc xóa một phần tử trong mảng một chiều, ở mảng hai chiều, một phần tử chính là một mảng một chiều. Do đó, chúng ta không phải ghi đè giá trị sau lên giá trị trước, mà chúng ta cần ghi đè dữ liệu của dòng sau lên dòng trước đó.

```
int main()
{
	int myArr[100][100];
	int num_of_row, num_of_col;

	//input
	cout << "Enter number of rows: "; cin >> num_of_row;
	cout << "Enter number of columns: "; cin >> num_of_col;
	for (int row = 0; row < num_of_row; row++)
	{
		for (int col = 0; col < num_of_col; col++)
		{
			cin >> myArr[row][col];
		}
	}

	//process
	int removeRow;
	cout << "Enter the row you want to remove: ";
	cin >> removeRow;

	//Overide the next row onto the previous row
	for (int row = removeRow; row < num_of_row - 1; row++)
	{
		for (int col = 0; col < num_of_col; col++)
		{
			myArr[row][col] = myArr[row + 1][col];
		}
	}
	num_of_row--;
```

```
        //output
        for (int row = 0; row < num_of_row; row++)
        {
                for (int col = 0; col < num_of_col; col++)
                {
                        cout << myArr[row][col] << " ";
                }
                cout << endl;
        }

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

Tổng kết

Trên đây chỉ mới là một số thao tác cơ bản khi cần sử dụng đến mảng hai chiều. Hi vọng bài học này có thể giúp các bạn hiểu rõ hơn về bản chất của mảng hai chiều khi lưu trữ trong máy tính.

Bài tập cơ bản

Dựa trên chương trình xóa một dòng trong mảng hai chiều mà mình đã làm mẫu ở trên, các bạn hãy viết chương trình xóa một cột X được nhập từ bàn phím trong mảng hai chiều.