

HUỲNH VĂN DUY  
VIẾT TẶNG THẦY LÊ VIỆT KHẢI (04/2017)

**C++**

**NHỮNG VIÊN GẠCH ĐẦU TIÊN**



**THPT CHUYÊN HUỲNH MÃN ĐẠT**

## *Lời nói đầu*

Nhiều năm qua, C++ đã dần thay thế ngôn ngữ Pascal để trở thành ngôn ngữ chính trong học tập và bồi dưỡng học sinh giỏi môn Tin học ở nhiều trường Trung học phổ thông trên cả nước. Tuy nhiên, tài liệu học tập C++ thì còn chưa nhiều hoặc có chăng là nội dung kiến thức quá bao phủ, chưa bám sát trọng tâm trong công tác bồi dưỡng học sinh giỏi. Do đó, tôi viết cuốn sách này như là một đề tập hợp những thứ cần thiết nhất để phục vụ học sinh trong đội tuyển Tin học có thể lấy làm tài liệu tham khảo và nghiên cứu về ngôn ngữ này. Tất cả kiến thức trong cuốn sách này được tôi tổng hợp từ các thư viện mã nguồn mở như: Wikipedia, tutorial point, cpulusplus, vietjack, ... và sách giáo khoa tin học lớp 11. Do đó, dưới cách nhìn khách quan thì tôi không dám đảm bảo những gì tôi viết là hoàn toàn chính xác. Cuốn sách này được chia thành hai phần. Phần một là “Chương trình C++ cơ bản” cung cấp những kiến thức cơ bản về chương trình và cách viết chương trình C++ bằng Code::Blocks, phần này chủ yếu bám theo khung của sách giáo khoa tin học lớp 11. Phần hai là một số bài viết hay được tổng hợp từ nhiều nguồn về những kĩ thuật hay trong C++ và sẽ được update thêm nếu cần thiết. Để cuốn sách ngày càng hoàn thiện hơn, trong quá trình sử dụng, nếu phát hiện sai sót hoặc ý kiến đóng góp nào đó các bạn có thể gửi vào hộp thư: [hvd.huynhduy@gmail.com](mailto:hvd.huynhduy@gmail.com).

04/2017

Huỳnh Văn Duy

## MỤC LỤC

PHẦN MỘT: .....	2
CHƯƠNG TRÌNH C++ CƠ BẢN .....	2
CHƯƠNG I:    TỔNG QUAN VỀ C++.....	3
VÀ MÔI TRƯỜNG LẬP TRÌNH CODEBLOCKS .....	3
§1: C++ VÀ CODEBLOCKS .....	3
CHƯƠNG II:    NHỮNG KHÁI NIỆM ĐẦU TIÊN.....	6
§1: NHỮNG YẾU TỐ CƠ BẢN TRONG C++ .....	6
§2: LẬP TRÌNH C++ BẰNG CODE::BLOCKS.....	10
§3. CÁC KIẾN THỨC CƠ BẢN .....	12
§3. NHẬP XUẤT DỮ LIỆU TRONG C++ .....	27
CHƯƠNG III: CẤU TRÚC ĐIỀU KHIỂN .....	35
§1: CẤU TRÚC ĐIỀU KIỆN .....	35
§2: CẤU TRÚC LẶP .....	42
CHƯƠNG IV: DỮ LIỆU CÓ CẤU TRÚC.....	49
§1: MẢNG .....	49
§2: CHUỖI (XÂU KÍ TỰ) .....	53
§2: KIỂU DỮ LIỆU TỰ ĐỊNH NGHĨA .....	60
CHƯƠNG V: LẬP TRÌNH CÓ CẤU TRÚC.....	65
§1: HÀM.....	65
CHƯƠNG VI: TỆP.....	76
§1: KIỂU DỮ LIỆU TỆP.....	76
§2: THAO TÁC VỚI TỆP TRONG C++ .....	77
CHƯƠNG VII: CON TRỎ.....	82
§1: CON TRỎ.....	82
PHẦN HAI: .....	97
MỘT SỐ KỸ THUẬT NÂNG CAO .....	97
§1: DEBUG VỚI CODE::BLOCKS.....	98
§2: STANDARD TEMPLATE LIBRARY (STL) .....	105
§3: COMPARISON FUNCTION .....	124
§4: CHỈ THỊ TIỀN XỬ LÝ TRONG C++.....	127
§5: MỘT SỐ KỸ THUẬT TRONG C++11 .....	130
§6: TEMPLATE TRONG C++.....	135

PHẦN MỘT:

# CHƯƠNG TRÌNH C++ CƠ BẢN

# CHƯƠNG I: TỔNG QUAN VỀ C++ VÀ MÔI TRƯỜNG LẬP TRÌNH CODEBLOCKS

---

## §1: C++ VÀ CODEBLOCKS

### 1. Một số khái niệm

#### *a. Khái niệm ngôn ngữ lập trình*

Ngôn ngữ lập trình là một tập con của ngôn ngữ máy tính, được thiết kế và chuẩn hóa để truyền chỉ thị cho các máy có bộ xử lí. Ngôn ngữ lập trình được dùng để lập trình máy tính, tạo ra các chương trình nhằm mục đích điều khiển máy tính hoặc mô tả thuật toán để người khác đọc hiểu.

#### *b. Các loại ngôn ngữ lập trình*

Ngày nay có ba loại ngôn ngữ lập trình chính đó là: Ngôn ngữ máy, hợp ngữ và ngôn ngữ bậc cao.

Ngôn ngữ máy (machine language) là ngôn ngữ được viết bằng các câu lệnh nhị phân, chương trình của ngôn ngữ máy có thể nạp vào bộ nhớ máy tính và thực hiện ngay. Hợp ngữ (assembly language) là ngôn ngữ được viết bằng những từ viết tắt tiếng anh diễn tả câu lệnh, một chương trình viết bằng hợp ngữ phải được dịch sang ngôn ngữ máy bằng chương trình hợp dịch. Ngôn ngữ còn lại là ngôn ngữ bậc cao (high-level programming languages) hay ngôn ngữ lập trình thế hệ thứ ba (3GL, third-generation programming languages). Trong ngôn ngữ này các câu lệnh được viết gần với ngôn ngữ tự nhiên hơn. Chương trình viết bằng ngôn ngữ bậc cao phải được chuyển đổi thành chương trình trên ngôn ngữ máy mới có thể thực hiện được., do đó phải sử dụng chương trình biên dịch (khác với chương trình hợp dịch) để chuyển đổi. Lập trình ngôn ngữ bậc cao dễ nhất vì các câu lệnh thường dùng tiếng anh thông dụng, còn ngôn ngữ máy và hợp ngữ thì thường chỉ dành cho những chuyên gia.

### 2. Ngôn ngữ lập trình C++

C++ là một ngôn ngữ lập trình. Đây là một dạng ngôn ngữ đa mẫu hình tự do có kiểu tĩnh và hỗ trợ lập trình thủ tục, dữ liệu trừu tượng, lập trình hướng đối tượng và lập trình đa hình. Từ thập niên 1990, C++ đã trở thành ngôn ngữ lập trình

thương mại phổ biến nhất khi đó. Trước C++, ngôn ngữ lập trình C được phát triển trong năm 1972 bởi Dennis Ritchie tại phòng thí nghiệm Bell Telephone, C chủ yếu là một ngôn ngữ lập trình hệ thống, một ngôn ngữ để viết ra hệ điều hành, vào 1999, ủy ban ANSI đã phát hành một phiên bản mới của C là C99. Dựa trên ngôn ngữ lập trình C, C++ được tạo ra bởi Bjarne Stroustrup – một nhà khoa học máy tính người Đan Mạch tại phòng thí nghiệm AT&T Bell vào năm 1979, được ISO công nhận vào năm 1998, lần phê chuẩn tiếp theo là 2003 (người ta gọi là C++ 03). Hai lần phê chuẩn gần nhất là 2011 và 2014 (C++ 11 và C++ 14). Phiên bản C++ 17 đã được công bố, dự đoán sẽ hoàn thành trong năm 2017.

Ta sẽ xét một vài đặc trưng của ngôn ngữ lập trình C++.

Thứ nhất, C++ là một ngôn ngữ lập trình bậc trung. Nó có nghĩa là bạn có thể sử dụng C++ để phát triển những ứng dụng bậc cao, và cả những chương trình bậc thấp hoạt động tốt trên phần cứng.

Thứ hai, C++ là một ngôn ngữ lập trình hướng đối tượng. Khác với ngôn ngữ C – một ngôn ngữ lập trình hướng thủ tục, chương trình được tổ chức theo thuật ngữ chức năng. C++ được thiết kế với một cách tiếp cận hoàn toàn mới được gọi là lập trình hướng đối tượng, nơi mà chúng ta sử dụng những đối tượng, các lớp và sử dụng các khái niệm như: thừa kế, đa hình, tính đóng gói, tính trừu tượng... Tuy nhiên, do C++ là phiên bản mở rộng của C nên ta vẫn có thể lập trình hướng chức năng như C.

Thứ ba, C++ có thể chạy trên nhiều nền tảng hệ điều hành khác nhau (đa nền tảng) như Windows, Mac OS, một số biến thể của UNIX, ...

### 3. Môi trường lập trình Code::Blocks

Trước hết ta cần hiểu thế nào là môi trường lập trình (IDE). IDE là viết tắt của cụm từ Integrated Development Environment là phần mềm cung cấp cho các lập trình viên một môi trường tích hợp bao gồm nhiều công cụ khác nhau như chương trình viết mã lệnh hay code editor, chương trình sửa lỗi hay debugger, chương trình mô phỏng ứng dụng khi chạy thực tế hay simulator.... Nói cách khác thì IDE là một phần mềm bao gồm những gói phần mềm khác giúp phát triển ứng dụng phần mềm. Các IDE phổ biến đang được sử dụng gồm có Netbeans IDE, Eclipse, PhpStorm, XCode (sử dụng trên hệ điều hành MacOS để phát triển ứng dụng mobile)... Phần tiếp theo chúng ta sẽ cùng tìm hiểu một số IDE phổ biến.

Code::Blocks là một IDE miễn phí cho C, C++ và Fortan. Bản thân IDE cũng được viết bằng C++ sử dụng framework wxWidget.

Download Code::Blocks tại đây: <http://www.codeblocks.org/downloads/26>.  
Lưu ý chọn phiên bản phù hợp, thường thì với những người bắt đầu, ta nên chọn bản codeblocks-16.01mingw-setup.exe vì nó có tích hợp sẵn bộ cài trình biên dịch MinGW.

## CHƯƠNG II: NHỮNG KHÁI NIỆM ĐẦU TIÊN

### §1: NHỮNG YẾU TỐ CƠ BẢN TRONG C++

#### 1. Các thành phần cơ bản.

Một ngôn ngữ lập trình thường có ba thành phần cơ bản là *bảng chữ cái*, *cú pháp* và *ngữ nghĩa*.

##### a. Bảng chữ cái

Bảng chữ cái là toàn bộ những kí tự được sử dụng trong chương trình. Ngoài ra, không được sử dụng kí tự nào khác. Bảng chữ cái trong C++ gồm:

- Các chữ cái in thường và in hoa trong tiếng Anh:

a b c d e f g h i j k l m n o p q r s t u v w x y z  
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- Các số Ả Rập: 0 1 2 3 4 5 6 7 8 9

- Các kí tự đặc biệt:

+	-	*	/	=	<	>	[	]	.	,
;	#	^	\$	@	&	(	)	{	}	:
Dấu cách (ASCII 32)										“

##### b. Cú pháp

Cú pháp là bộ quy tắc của ngôn ngữ lập trình. Dựa vào chúng, người lập trình và chương trình dịch biết được tổ hợp nào của các kí tự trong bảng chữ cái là hợp lệ và tổ hợp nào là hợp lệ. Nhờ đó có thể miêu tả chính xác thuật toán để thực hiện.

##### c. Ngữ nghĩa

Là cách xác định ý nghĩa của từng tổ hợp cú pháp.

#### 2. Một số khái niệm

##### a. Tên

Mọi đối tượng trong chương trình đều phải được đặt tên theo quy tắc của ngôn ngữ lập trình và từng chương trình dịch cụ thể.

Trong C++, tên là một chuỗi kí tự liên tiếp không có khoảng cách. Tên phải bắt đầu bằng kí tự không phải số hoặc “\_”. Khác với Pascal, C++ là ngôn ngữ phân



biệt chữ hoa chữ thường. Vì vậy, hai tên Chuongtrinh và chuongtrinh là hoàn toàn khác nhau. Tên có ba loại chính:

- *Tên dành riêng*: Một số tên được quy định với ý nghĩa riêng xác định. Người lập trình không được sử dụng với ý nghĩa khác. Ví dụ:  
main, include, if, while, void, ...
- *Tên chuẩn*: Một số tên được quy định một ý nghĩa nhất định nào đó, nhưng người lập trình có thể sử dụng với ý nghĩa khác. Ví dụ:  
cin, cout, getchar, ...
- *Tên do người lập trình đặt*: Đây là tên dùng với ý nghĩa riêng do người lập trình quy định bằng cách khai báo trước khi sử dụng. Các tên này không được trùng với tên riêng.

### ***b. Hằng và biến.***

Hằng là đại lượng đặt trưng có giá trị không đổi trong quá trình thực hiện chương trình.

- Trong các ngôn ngữ lập trình thường có các hằng số học, hằng logic, và hằng xâu.
  - Hằng số học là các số nguyên hay số thực (dấu phẩy tính hoặc dấu phẩy động).
  - Hằng logic là giá trị đúng hoặc sai tương ứng với true or false.
  - Hằng xâu là chuỗi kí tự trong bộ mã ASCII. Khi viết kí tự này trong C++ phải để trong dấu nháy kép.

Biến là đại lượng đặt trưng được đặt tên, dùng để lưu giữ giá trị và giá trị có thể được thay đổi trong quá trình thực hiện chương trình.

### ***c. Chú thích***

Có thể đặt các đoạn chú thích (comment) trong chương trình nguồn. Các chú thích này chỉ giúp cho người đọc nhận biết được ý nghĩa của chương trình, chứ không ảnh hưởng đến nội dung chương trình và được chương trình dịch bỏ qua.

Trong C++ chú thích thường được đặt giữa /\* và \*/ hoặc phía sau dấu //.

## **3. Cấu trúc chương trình C++**

Với một chương trình C++ chuẩn ta thường sẽ có những phần sau:

- Phần khai báo gồm:
  - Khai báo header file (phần này là bắt buộc): Đây là phần khai báo những header file được dùng trong chương trình, đó có thể là một thư viện chuẩn (STL) hoặc một header file do người lập trình viết.

- Khai báo namespace (không bắt buộc): namespace là một cơ chế trong C++, cho phép ta nhóm các thực thể (class, object, function...) có liên quan thành từng nhóm khác nhau theo tên, mà theo đó tên của mọi thực thể trong mỗi namespace đều được gắn thêm tên của namespace đó như tiền tố. Trong C++, namespace std là namespace phổ biến nhất chứa đa số những hàm mà ta cần. Ta sẽ không tìm hiểu sâu về namespace ở phần này.
- Các khai báo toàn cục khác (không bắt buộc): Ta có thể khai báo biến toàn cục, hằng toàn cục, lớp toàn cục, hàm toàn cục hoặc kiểu dữ liệu tự định nghĩa và nhiều thứ khác.
- Phần thân chương trình: Thực chất phần thân chính là phần khai báo hàm main. Hàm main có một số đặt điểm sau:
  - Mọi chương trình C++ đều chạy bắt đầu từ hàm main.
  - Tất cả các hàm chỉ được thực thi nếu được gọi trong hàm main và thứ tự thực thi cũng được quy định trong hàm main.
  - Mỗi chương trình C++ chỉ có duy nhất một hàm main.

Có nhiều cách viết một chương trình và thứ tự khai báo khác nhau cùng cho ra một chương trình giống nhau.

Ta sẽ xét một ví dụ:

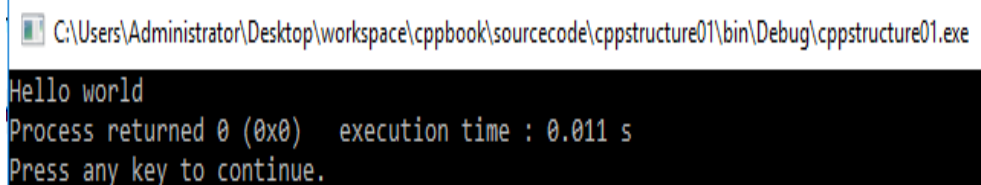
```
#include <bits/stdc++.h>

using namespace std;

int main() {
    cout << "Hello world";
    return 0;
}
```

- Dòng đầu tiên là khai báo thư viện `bit/stdc++.h`, đây là một thư viện chuẩn của C++. Hoặc nếu ta tạo một header file là `myheader.h` và muốn include vào chương trình thì chỉ cần thực hiện cú pháp:  
`#include <đường dẫn đến myheader.h>` ví dụ:  
`#include </headerfile/duy/code/myheader.h>`
- Dòng tiếp theo là `using namespace std` nghĩa là khai báo namespace std. Nếu không khai báo std thì khi sử dụng những hàm trong namespace std ta phải thực hiện cú pháp `std::<tên hàm>`. Ngược lại trong chương

- trình ta chỉ cần gọi tên hàm. Ví dụ với hàm `max`, khi không khai báo namespace `std` ta phải viết `std::max(a, b)`, nhưng nếu như đã khai báo thì chỉ cần gọi `max(a, b)` là được.
- `int main` như đã nói là phần khai báo hàm `main`. Trong hàm `main` chứa những câu lệnh chính của chương trình. Bao gồm `cout << "Hello world"` và `return 0`.
    - `cout << "Hello world"`: là câu lệnh dùng để in ra màn hình dòng chữ "Hello world".
    - `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.



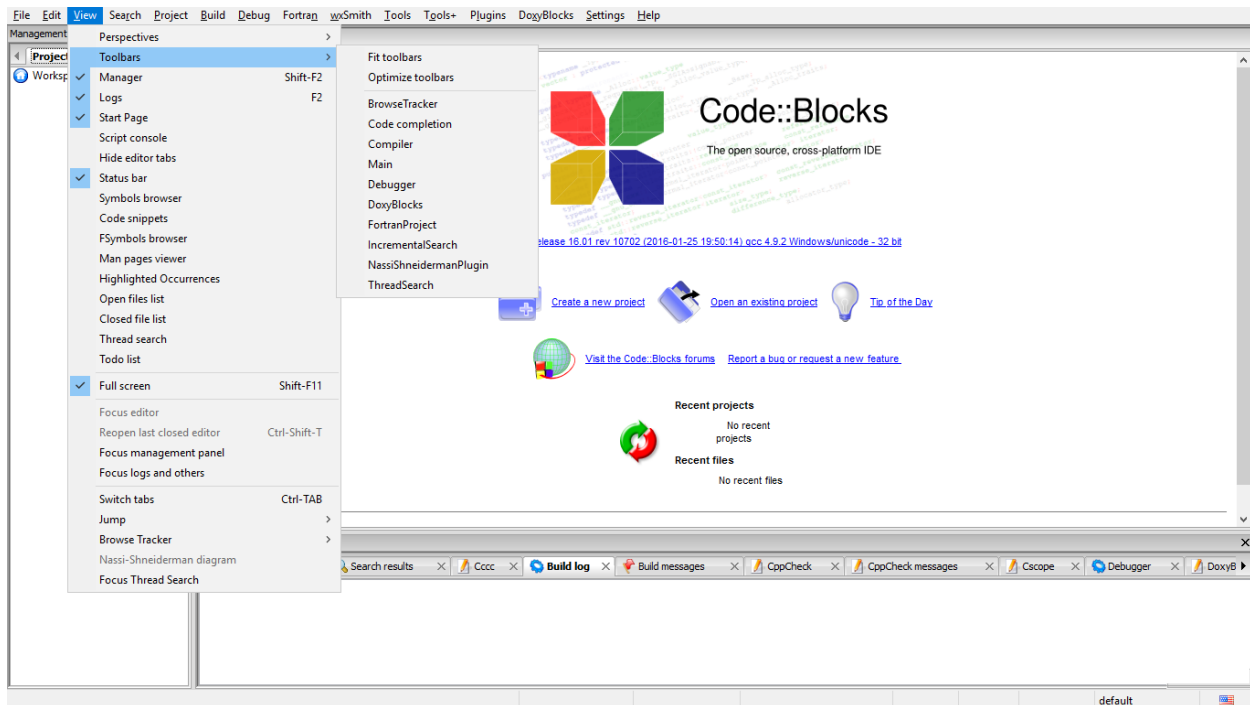
```

C:\Users\Administrator\Desktop\workspace\cppbook\sourcecode\cppstructure01\bin\Debug\cppstructure01.exe
Hello world
Process returned 0 (0x0)   execution time : 0.011 s
Press any key to continue.
  
```

## §2: LẬP TRÌNH C++ BẰNG CODE::BLOCKS

### 1. Giao diện người dùng

Code::Blocks có giao diện khá trực quan và dễ dùng. Mọi thứ đều có thể tùy chỉnh bằng việc tích hoặc bỏ tích phần phần view.



HÌNH 1

Những Toolbars thường dùng là compiler và debugger. Chứa những công cụ giúp chúng ta dịch, chạy và debug chương trình.

### 2. Viết chương trình đơn giản bằng Code::Blocks

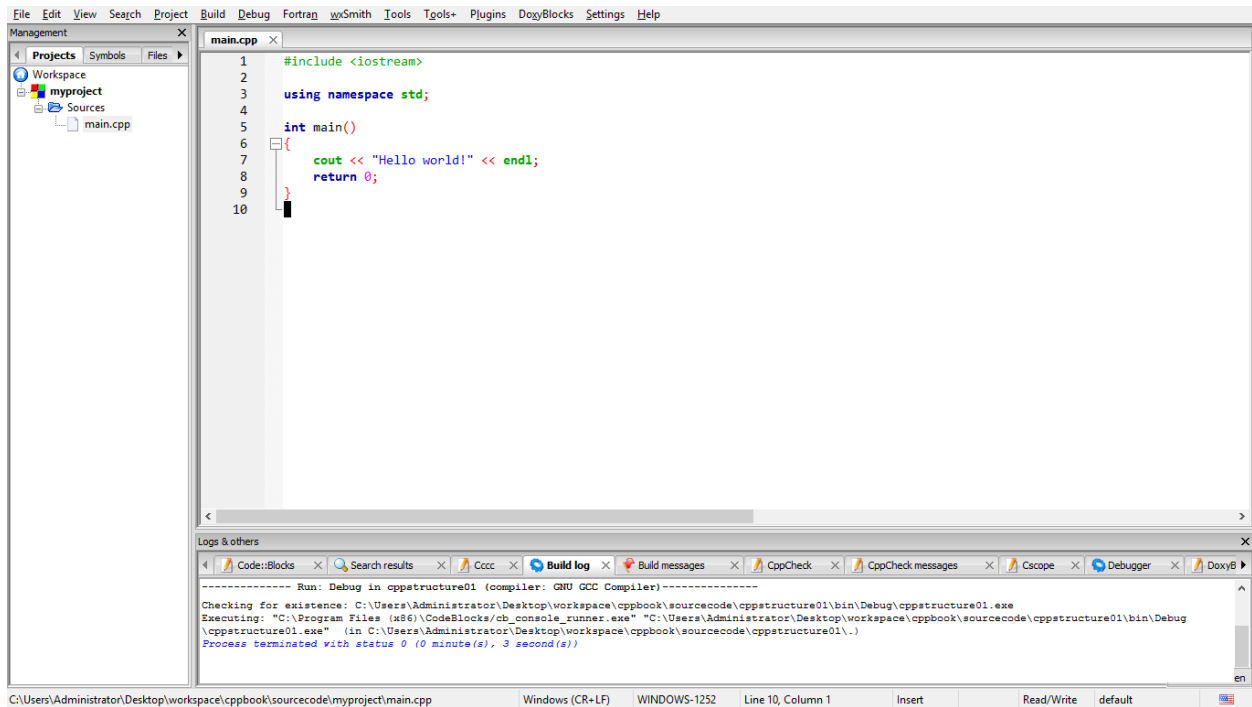
Đầu tiên ta sẽ mở một project C++.

**Bước 1:** Chọn File → New → Project

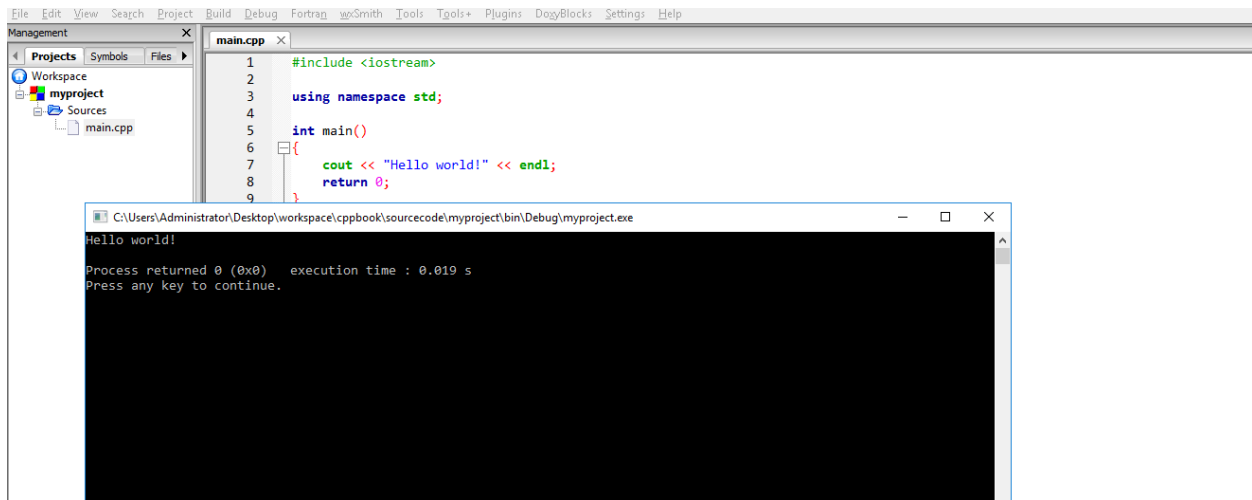
**Bước 2:** Cửa sổ New from template hiện lên, chọn Console application. Tiếp theo chọn C++.

**Bước 3:** Điền những thông tin như về project.

Sau khi thực hiện xong ta sẽ được một project mới và ta sẽ bắt đầu code với file main.cpp.



Để dịch và chạy chương trình ta nhấn phím **F9** hoặc chọn **Build → Build and Run**. Khi đó một cửa sổ console sẽ xuất hiện cho ta output của chương trình.



## §3. CÁC KIẾN THỨC CƠ BẢN

### 1. Một số kiểu dữ liệu chuẩn

Thông thường dữ liệu input phần lớn là số và chữ. Tuy nhiên, việc phân chia thành hai loại dữ liệu như vậy là không đủ cho nên đa số các ngôn ngữ lập trình bậc cao đều phân chia dữ liệu thành các kiểu khác nhau. Thông thường một kiểu dữ liệu trong C++ sẽ được xác định bởi ba đặc điểm:

- Tên kiểu: Là một tên dành riêng để xác định kiểu dữ liệu.
- Số byte trong bộ nhớ để lưu trữ một đơn vị dữ liệu kiểu này.
- Miền giá trị: Cho biết một đơn vị kiểu dữ liệu này có thể dùng cho những giá trị thuộc miền nào.

Dưới đây là bảng tóm tắt một số kiểu dữ liệu trong C++. Đầu tiên là các kiểu dữ liệu nguyên thủy. Tên tiếng Anh là Primitive Type, còn có thể gọi là kiểu dữ liệu gốc, kiểu dữ liệu có sẵn trong C/C++. Bên cạnh các kiểu dữ liệu gốc này, C/C++ cũng cung cấp các kiểu dữ liệu user-defined. Bảng dưới đây liệt kê 7 kiểu dữ liệu cơ bản trong C/C++:

Kiểu dữ liệu	Từ khóa
Boolean	bool
Ký tự	char
Số nguyên	int
Số thực	float
Số thực dạng Double	double
Kiểu không có giá trị	void

Kiểu Wide character	wchar_t
---------------------	---------

Một số kiểu dữ liệu được sửa đổi bởi một hoặc nhiều modifier như:

- signed(Kiểu có dấu)
- unsigned(Kiểu không có dấu)
- short
- long

Kiểu	Độ rộng bit	Dãy giá trị
char	1 byte	-127 tới 127 hoặc 0 tới 255
unsigned char	1 byte	0 tới 255
signed char	1 byte	-127 tới 127
int	4 byte	-2147483648 tới 2147483647
unsigned int	4 byte	0 tới 4294967295
signed int	4 byte	-2147483648 tới 2147483647
short int	2 byte	-32768 tới 32767
unsigned short int	Range	0 tới 65,535
signed short int	Range	-32768 tới 32767
long int	4 byte	-2,147,483,647 tới 2,147,483,647

signed long int	4 byte	Tương tự như long int
unsigned long int	4 byte	0 tới 4,294,967,295
float	4 byte	+/- 3.4e +/- 38 (~7 chữ số)
double	8 byte	+/- 1.7e +/- 308 (~15 chữ số)
long double	8 byte	+/- 1.7e +/- 308 (~15 chữ số)
wchar_t	2 hoặc 4 byte	1 wide character

Kích cỡ của các biến có thể khác với những gì hiển thị trên bảng, phụ thuộc vào compiler và máy tính bạn đang sử dụng. Dưới đây là ví dụ sẽ đưa ra kích cỡ chính xác của các kiểu dữ liệu đa dạng trên máy tính của bạn.

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

int main()
{
    cout << "Kích co cua char la: " << sizeof(char) << endl;
    cout << "Kích co cua int la: " << sizeof(int) << endl;
    cout << "Kích co cua short int la: " << sizeof(short int) << endl;
    cout << "Kích co cua long int la: " << sizeof(long int) << endl;
    cout << "Kích co cua float la: " << sizeof(float) << endl;
    cout << "Kích co cua double la: " << sizeof(double) << endl;
    cout << "Kích co cua wchar_t la: " << sizeof(wchar_t) << endl;
    return 0;
}
```

## 2. Câu lệnh

Một câu lệnh (statement) xác định một công việc mà chương trình phải thực hiện để xử lý dữ liệu đã được mô tả và khai báo. Các câu lệnh được ngăn cách với nhau bởi dấu chấm phẩy (;).



Thông thường chúng ta có hai loại câu lệnh đó là lệnh đơn và lệnh có cấu trúc. Lệnh đơn là một lệnh không chứa câu lệnh khác. Các lệnh đơn gồm: lệnh gán, các câu lệnh nhập xuất dữ liệu, ..

Lệnh có cấu trúc là lệnh mà trong đó chứa các lệnh khác. Lệnh có cấu trúc bao gồm: Cấu trúc rẽ nhánh, cấu trúc điều kiện lựa chọn, cấu trúc lặp và cấu trúc lệnh hợp thành. Lệnh hợp thành (khối lệnh) là một nhóm các bao gồm nhiều khai báo biến và các câu lệnh được gom vào cặp dấu {}.

### 3. Phép toán và biểu thức

Tương tự trong toán học, trong các ngôn ngữ lập trình đều có những phép toán như cộng, trừ, nhân, chia trên các đại lượng thực, các phép toán chia lấy nguyên và lấy phần dư, các phép toán qua hệ, các phép toán với bit ...

Bảng dưới đây là kí hiệu các phép toán số học trong C++:

Toán tử	Miêu tả	Ví dụ
+	Cộng hai toán hạng	$A + B$ kết quả là 30
-	Trừ toán hạng thứ hai từ toán hạng đầu	$A - B$ kết quả là -10
*	Nhân hai toán hạng	$A * B$ kết quả là 200
/	Phép chia	$B / A$ kết quả là 2
%	Phép lấy số dư	$B \% A$ kết quả là 0
++	<b><u>Toán tử tăng (++)</u></b> , tăng giá trị toán hạng thêm một đơn vị	$A++$ kết quả là 11

--	<b>Toán tử giảm (--)</b> , giảm giá trị toán hạng đi một đơn vị	A-- kết quả là
----	---	----------------

Bảng các phép toán quan hệ. Giả sử biến A giữ giá trị 10, biến B giữ giá trị 30 thì:

Toán tử	Miêu tả	Ví dụ
==	Kiểm tra nếu 2 toán hạng bằng nhau hay không. Nếu bằng thì điều kiện là true.	(A == B) là không đúng
!=	Kiểm tra 2 toán hạng có giá trị khác nhau hay không. Nếu không bằng thì điều kiện là true.	(A != B) là true
>	Kiểm tra nếu toán hạng bên trái có giá trị lớn hơn toán hạng bên phải hay không. Nếu lớn hơn thì điều kiện là true.	(A > B) là không đúng
<	Kiểm tra nếu toán hạng bên trái nhỏ hơn toán hạng bên phải hay không. Nếu nhỏ hơn thì là true.	(A < B) là true
>=	Kiểm tra nếu toán hạng bên trái có giá trị lớn hơn hoặc bằng giá trị của toán hạng bên phải hay không. Nếu đúng là true.	(A >= B) là không đúng
<=	Kiểm tra nếu toán hạng bên trái có giá trị nhỏ hơn hoặc bằng toán	(A <= B) là true

	hạng bên phải hay không. Nếu đúng là true.	
--	--	--

Bảng các phép toán logic. Giả sử biến A có giá trị 1 và biến B có giá trị 0 thì:

Toán tử	Miêu tả	Ví dụ
&&	Được gọi là toán tử logic AND (và). Nếu cả hai toán tử đều có giá trị khác 0 thì điều kiện trở lên true.	(A && B) là false.
	Được gọi là toán tử logic OR (hoặc). Nếu một trong hai toán tử khác 0, thì điều kiện là true.	(A    B) là true.
!	Được gọi là toán tử NOT (phủ định). Sử dụng để đảo ngược lại trạng thái logic của toán hạng đó. Nếu điều kiện toán hạng là true thì phủ định nó sẽ là false.	!(A && B) là true.

Bảng các toán tử so sánh bit. Phép toán làm việc trên đơn vị bit, tính toán biểu thức và so sánh từng bit:

p	q	p & q	p   q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Giả sử nếu  $A = 60$  và  $B = 13$  thì:

Toán tử	Miêu tả	Ví dụ
&	Toán tử AND (và) nhị phân sao chép một bit tới kết quả nếu nó tồn tại trong cả hai toán hạng.	$(A \& B)$ sẽ cho kết quả là 12, tức là 0000 1100
	Toán tử OR (hoặc) nhị phân sao chép một bit tới kết quả nếu nó tồn tại trong một hoặc hai toán hạng.	$(A   B)$ sẽ cho kết quả là 61, tức là 0011 1101
^	Toán tử XOR nhị phân sao chép bit mà nó chỉ tồn tại trong một toán hạng mà không phải cả hai.	$(A \wedge B)$ sẽ cho kết quả là 49, tức là 0011 0001
~	Toán tử đảo bit (đảo bit 1 thành bit 0 và ngược lại).	$(\sim A)$ sẽ cho kết quả là -61, tức là 1100 0011.
<<	Toán tử dịch trái. Giá trị toán hạng trái được dịch chuyển sang trái bởi số các bit được xác định bởi toán hạng bên phải.	$A \ll 2$ sẽ cho kết quả 240, tức là 1111 0000 (dịch sang trái hai bit)
>>	Toán tử dịch phải. Giá trị toán hạng trái được dịch chuyển sang phải bởi số các bit được xác định bởi toán hạng bên phải.	$A \gg 2$ sẽ cho kết quả là 15, tức là 0000 1111 (dịch sang phải hai bit)

Bảng các phép toán hỗn hợp:

Toán tử	Miêu tả
sizeof	<b><u>Toán tử sizeof trong C++</u></b> trả về kích cỡ của một biến. Ví dụ: sizeof(a), với a là integer, sẽ trả về 4
Điều kiện ? X : Y	<b><u>Toán tử điều kiện trong C++</u></b> . Nếu Condition là true ? thì nó trả về giá trị X : nếu không thì trả về Y
,	<b><u>Toán tử Comma trong C++</u></b> làm cho một dãy hoạt động được thực hiện. Giá trị của toàn biểu thức comma là giá trị của biểu thức cuối cùng trong danh sách được phân biệt bởi dấu phẩy
. (dot) và -> (arrow)	<b><u>Toán tử thành viên trong C++</u></b> được sử dụng để tham chiếu các phần tử đơn của các lớp, các cấu trúc, và union
Cast	<b><u>Toán tử ép kiểu (Casting) trong C++</u></b> biến đổi một kiểu dữ liệu thành kiểu khác. Ví dụ: int(2.2000) sẽ trả về 2
&	<b><u>Toán tử con trỏ &amp; trong C++</u></b> trả về địa chỉ của một biến. Ví dụ: &a; sẽ trả về địa chỉ thực sự của biến này
*	<b><u>Toán tử con trỏ * trong C++</u></b> là trỏ tới một biến. Ví dụ: *var sẽ trỏ tới một biến var

Thứ tự ưu tiên của các phép toán trong C++ xác định cách biểu thức được tính toán. Ví dụ, toán tử nhân có quyền ưu tiên hơn toán tử cộng, và nó được thực hiện trước. Bảng dưới đây liệt kê thứ tự ưu tiên thấp nhất thì ở bên dưới còn cao nhất thì xuất hiện trên cùng của bảng, và các toán tử có quyền ưu tiên thấp nhất thì ở bên dưới cùng của bảng. Trong một biểu thức, các toán tử có quyền ưu tiên cao hơn sẽ được thực hiện trước.

Loại	Toán tử	Thứ tự ưu tiên
Postfix	() [] -> . ++ --	Trái sang phải
Unary	+ - ! ~ ++ -- (type)* & sizeof	Phải sang trái
Tính nhân	* / %	Trái sang phải
Tính cộng	+ -	Trái sang phải
Dịch chuyển	<< >>	Trái sang phải
Quan hệ	< <= > >=	Trái sang phải
Cân bằng	== !=	Trái sang phải
Phép AND bit	&	Trái sang phải
Phép XOR bit	^	Trái sang phải
Phép OR bit		Trái sang phải
Phép AND logic	&&	Trái sang phải
Phép OR logic		Trái sang phải
Điều kiện	?:	Phải sang trái
Gán	= += -= *= /= %= >>= <<= &= ^=  =	Phải sang trái

Dấu phẩy	,	Trái sang phải
----------	---	----------------

## 4. Biến và hằng

### a. Biến

Như đã nói ở bài trước, một biến cung cấp nơi lưu giữ được đặt tên để chúng ta có thể thao tác. Mỗi biến trong C/C++ có một kiểu cụ thể, mà quyết định: kích cỡ và cách bố trí bộ nhớ của biến; dãy giá trị có thể được lưu giữ bên trong bộ nhớ đó; và tập hợp hoạt động có thể được áp dụng cho biến đó.

Tên biến có thể gồm các ký tự, các chữ số, dấu gạch dưới. Nó phải bắt đầu bởi hoặc một ký tự hoặc một dấu gạch dưới. Các ký tự chữ hoa và chữ thường là khác nhau bởi C/C++ là ngôn ngữ phân biệt kiểu chữ. Biến có thể trong các kiểu giá trị đa dạng như kiểu char, int, float, ...

Cách khai báo biến: <Kiểu biến> <Danh sách biến>;

Ví dụ: `int x, y, z; // Khai báo ba biến x, y, z theo kiểu int.`

Lưu ý, Một scope (phạm vi) là một khu vực của chương trình nơi biến hoạt động, và nói chung có thể có 3 khu vực mà biến có thể được khai báo:

- Bên trong một hàm hoặc một khối, được gọi là biến cục bộ (local).
- Trong định nghĩa của các tham số hàm, được gọi là các tham số hình thức (formal).
- Bên ngoài của tất cả hàm, được gọi là biến toàn cục (global).

Chúng ta sẽ học hàm và các tham số của hàm là gì trong chương tới. Dưới đây chúng tôi sẽ giải thích khái niệm về biến cục bộ và biến toàn cục.

- Các biến được khai báo bên trong một hàm hoặc khối là các biến cục bộ (local). Chúng chỉ có thể được sử dụng bởi các lệnh bên trong hàm hoặc khối code đó. Các biến cục bộ không được biết ở bên ngoài hàm đó (tức là chỉ được sử dụng bên trong hàm hoặc khối code đó). Dưới đây là ví dụ sử dụng các biến cục bộ:

```
#include <bits/stdc++.h>

using namespace std;

int main() {

    int a, b, c;

    c = 10; b = 2;

    a = b + c;

}
```

- Biến toàn cục (global) trong C++ được định nghĩa bên ngoài các hàm, thường ở phần đầu chương trình. Các biến toàn cục giữ giá trị của nó trong suốt vòng đời chương trình của bạn. Một biến toàn cục có thể được truy cập bởi bất kỳ hàm nào. Tức là, một biến toàn cục là có sẵn cho bạn sử dụng trong toàn bộ chương trình sau khi đã khai báo nó. Dưới đây là ví dụ sử dụng biến toàn cục và biến nội bộ trong C++:

```
#include <bits/stdc++.h>

using namespace std;

int g;

int main() {

    int a, b;

    a = 10;

    b = 20;

    g = a + b;

}
```



```
}
```

Lưu ý, một chương trình có thể có các biến toàn cục và các biến cục bộ trùng tên nhau, nhưng trong một hàm thì giá trị của biến cục bộ sẽ được ưu tiên.

### ***b. Hằng***

Hằng liên quan đến các giá trị có định mà chương trình không thể thay đổi và chúng được gọi là Literals. Hằng là một kiểu dữ liệu thay thế cho Literals, còn literals thể hiện chính nó. Ví dụ: `const PI = 3.14` thì hằng ở đây là PI, còn literals là 14.

Hằng có thể là bất kì kiểu dữ liệu cơ bản nào trong C/C++, và có thể được phân chia thành hằng số nguyên, hằng số thực, hằng kí tự, hằng chuỗi và Boolean literals (tạm dịch là hằng logic). Ngoài ra, hằng được đối xử giống như biến thông thường, ngoại trừ việc giá trị của chúng không thể thay đổi sau khi khai báo.

Có 2 cách khai báo hằng trong C++:

- Tiền xử lí `#define`: `#define <tên hằng> <giá trị của hằng>`
- Sử dụng từ khóa `const`: `const <tên hằng> = <giá trị của hằng>;`

## **5. Lệnh gán**

Lệnh gán (assignment statement) dùng để gán giá trị của một biểu thức cho một biến.

Cú pháp: `<Tên biến> = <biểu thức>;`

Ví dụ:

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int x = 10;    // Gán hằng số 10 cho biến x
    int y = 2*x;   // Gán biểu thức 2*x = 2*10 = biến y
    return 0;
}
```

Nguyên tắc khi dùng phép lệnh gán là kiểu của biến và kiểu của biểu thức phải giống nhau, gọi là có sự tương thích giữa các kiểu dữ liệu. Ví dụ sau cho thấy một sự không tương thích về kiểu dữ liệu.

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int x, y;
    x = 10;
    y = "Xin chào";
    return 0;
}
```

Khi biên dịch chương trình này, C++ sẽ báo lỗi “Cannot convert ‘char’ to ‘int’” tức là C++ không thể tự động chuyển đổi kiểu từ char\* (chuỗi kí tự) sang int.

Tuy nhiên, trong đa số trường hợp sự tự động biến đổi kiểu để sự tương thích về kiểu được đảm bảo sẽ được thực hiện. Ví dụ:

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int x, y;
    float r;
    char ch;
    r = 9000;
    x = 10; // Gán hằng số 10 cho biến x
    y = 'd'; // y có kiểu int, còn 'd' là char, lúc này mã ASCII của 'd' sẽ
    được gán cho y
    r = 'e'; // r có kiểu float, 'e' có kiểu char, mã ASCII của 'e' cũng sẽ
    được gán cho r
    return 0;
}
```

Trong nhiều trường hợp để đảm bảo sự tương thích về kiểu, ta phải sử dụng đến cách chuyển đổi kiểu một cách tường minh. Cú pháp của phép ép kiểu này như sau:

(Tên kiểu) <Biểu thức>;

Nghĩa là ta chuyển đổi kiểu của biểu thức thành kiểu mới là <Tên kiểu>. Chẳng hạn như: `f = (float) 10/4;`

Chú ý:

- Khi một biểu thức được gán cho một biến thì giá trị của nó sẽ thay thế giá trị cũ mà biến đã lưu giữ trước đó.

- Trong câu lệnh gán, dấu “=” là một toán tử; do đó nó có thể sử dụng như một thành phần của biểu thức. Trong trường hợp này giá trị của biểu thức gán chính là giá trị của biến. Ví dụ:  

```
int x, y;
x = y = 3; // Lúc này x và y cùng bằng 3
```
- Ta có thể gán giá trị của biến lúc được khai báo theo cách như sau:  

```
<Tên kiểu><Tên biến> = <Biểu thức>;
```

  
 Ví dụ: `int x = 10, y = x;`

Ngoài dấu “=” ta có thêm một số toán tử gán khác:

Toán tử	Miêu tả	Ví dụ
=	Toán tử gán đơn giản. Gán giá trị toán hạng bên phải cho toán hạng trái.	$C = A + B$ sẽ gán giá trị của $A + B$ vào trong $C$
+=	Thêm giá trị toán hạng phải tới toán hạng trái và gán giá trị đó cho toán hạng trái.	$C += A$ tương đương với $C = C + A$
-=	Trừ đi giá trị toán hạng phải từ toán hạng trái và gán giá trị này cho toán hạng trái.	$C -= A$ tương đương với $C = C - A$
*=	Nhân giá trị toán hạng phải với toán hạng trái và gán giá trị này cho toán hạng trái.	$C *= A$ tương đương với $C = C * A$
/=	Chia toán hạng trái cho toán hạng phải và gán giá trị này cho toán hạng trái.	$C /= A$ tương đương với $C = C / A$
%=	Lấy phần dư của phép chia toán hạng trái cho toán hạng phải và gán cho toán hạng trái.	$C \% = A$ tương đương với $C = C \% A$

<<=	Dịch trái toán hạng trái sang số vị trí là giá trị toán hạng phải.	$C \ll 2$ tương đương với $C = C \ll 2$
>>=	Dịch phải toán hạng trái sang số vị trí là giá trị toán hạng phải.	$C \gg 2$ tương đương với $C = C \gg 2$
&=	Phép AND bit	$C \&= 2$ tương đương với $C = C \& 2$
^=	Phép OR loại trừ bit	$C \wedge= 2$ tương đương với $C = C \wedge 2$
=	Phép OR bit.	$C  = 2$ tương đương với $C = C   2$

## §3. NHẬP XUẤT DỮ LIỆU TRONG C++

Thư viện chuẩn trong C++ cung cấp nhiều khả năng để input/output và sẽ được bàn luận trong các chương sau. Trong chương này, chúng ta thảo luận rất cơ bản và phổ biến nhất về hoạt động I/O cần thiết cho lập trình C++.

I/O trong C++ diễn ra trong các stream (luồng) như là các dãy byte. Nếu các byte chảy từ một thiết bị, như bàn phím, disk drive, hoặc kết nối mạng ..., tới bộ nhớ chính thì được gọi là input. Ngược lại gọi là output.

### 1. Header file trong C++

Bảng dưới đây liệt kê các header file quan trọng trong chương trình C++:

Header File	Miêu tả
iostream	File này định nghĩa các đối tượng <b>cin</b> , <b>cout</b> , <b>cerr</b> và <b>clog</b> , tương ứng với Standard Input Stream (Luồng đầu vào chuẩn), Standard Output Stream (Luồng đầu ra chuẩn), Un-buffered Standard Error Stream (Luồng lỗi chuẩn không được đệm) và Buffered Standard Error Stream (Luồng lỗi chuẩn được đệm)
iomanip	File này khai báo các dịch vụ hữu ích để thực hiện hoạt động I/O được định dạng với các bộ thao tác luồng được tham số hóa như <b>setw</b> và <b>setprecision</b> .
fstream	File này khai báo các dịch vụ xử lý file được kiểm soát bởi người dùng. Chúng ta sẽ thảo luận chi tiết về nó trong chương File và Stream trong C++

### 2. Luồng nhập – xuất tiêu chuẩn trong C++ (Standard I/O stream)

#### a. Luồng nhập chuẩn (Standard Input Stream)

Để nhập dữ liệu, ta sử dụng đối tượng tiền định nghĩa **cin**, **cin** là một minh họa của lớp **istream**. Đối tượng **cin** được xem như đính kèm với thiết bị đầu vào chuẩn,

mà thường là bàn phím. Đối tượng cin được sử dụng với toán tử trích luồng (extraction operator), viết là >>, như trong ví dụ sau:

```
#include <iostream>
using namespace std;
int main() {
    string name;
    cin >> name;
}
```

Toán tử trích luồng có thể sử dụng nhiều hơn trong một lần lệnh. Để yêu cầu nhiều hơn một dữ liệu chuẩn, bạn có thể sử dụng:

```
cin >> name >> age;
```

Nó tương đương với:

```
cin >> name;
```

```
cin >> age;
```

### ***b. Luồng xuất tiêu chuẩn (Standard Output Stream)***

Để xuất dữ liệu ta sử dụng đối tượng tiền định nghĩa cout, cout là một minh họa của ostream. Đối tượng cout được sử dụng với toán tử chèn luồng (insertion operation), được viết là <<, như ví dụ dưới đây:

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    char str[] = "Hello World";
    cout << str;
}
```

### ***c. Standard Error Stream (cerr) và Standard Log Stream (clog)***

Đối tượng tiền định nghĩa cerr là một minh họa của ostream. Đối tượng cerr được xem như đính kèm với thiết bị lỗi chuẩn, ta thường dùng để xuất lỗi ra màn hình. Ta sẽ thấy hữu ích của nó khi làm việc với input/output source không phải là màn hình. Đối tượng cerr trong C++ cũng được kết hợp sử dụng với toán tử chèn luồng <<. Đối tượng clog cũng giống như cerr, tuy nhiên mỗi sự chèn luồng tới clog làm đầu ra đều được giữ trong một bộ đệm (buffer) tới khi bộ đệm này đầy hoặc tới khi bộ đệm này bị flush (bị lấy ra), còn cerr là un-buffer (không được đệm) và mỗi sự chèn luồng làm đầu ra của nó sẽ xuất hiện tức thì. Ví dụ:

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    string str1 = "world";
```

```
string str2 = "Hello";
clog << str1 << endl;
cerr << str2 << endl;
}
```

### 3. Sử dụng cách nhập trong C

Như chúng ta đã biết, C là tổ tiên của C++ cho nên hầu hết những thứ gì C làm được thì C++ cũng làm được. Do đó, ta có thể sử dụng cách nhập trong C cho C++.

#### a. Mô tả định dạng dữ liệu

Không giống như Standard I/O Stream của C++ tự phát hiện kiểu dữ liệu, thì khi muốn nhập hoặc xuất trong C, ta nhất thiết phải định dạng kiểu dữ liệu để C xử lý. Ta sẽ mô tả định dạng bằng một chuỗi được quy định trong C, bắt đầu bằng %. Chuỗi định dạng phải tương ứng với danh sách tham số về số lượng, kiểu dữ liệu và thứ tự. Chuỗi định dạng luôn được đặt bên trong cặp dấu "" và bao gồm một hoặc nhiều thành phần sau:

- Ký tự văn bản (Text characters): là những ký tự in được, bao gồm các chữ cái, chữ số và các ký tự đặc biệt trong bảng mã ASCII.
- Các ký tự không in được: bao gồm một số ký tự điều khiển như tab (\t), xuống dòng (\n), khoảng trắng, ... Khoảng trắng thường được sử dụng để phân cách các trường (field) dữ liệu được xuất ra.
- Lệnh định dạng: là các lệnh quy định cách mà dữ liệu được xuất ra màn hình.

Lệnh định dạng có cấu trúc như sau:

**%[flags][width][.precision][length]specifier**

Trong đó, flags, width, .precision, length là một số bổ từ tùy chọn để thay đổi đặc tả gốc cho phù hợp với mong muốn của lập trình viên.

Specifier là đặc tả cho kiểu dữ liệu. Một số đặc tả thường dùng:

- %c: Ký tự đơn.
- %s: Chuỗi.
- %d: Số nguyên có dấu.
- %f: Số thực chấm phẩy động (VD 5.54 sẽ in ra 5.540000).
- %e: Số thực chấm phẩy động (ký hiệu có số mũ).
- %g: Số thực chấm phẩy động (VD 5.54 sẽ in ra 5.54).

- %x: Số nguyên hex không dấu (hệ 16).
- %o: Số nguyên bát phân (hệ 8).
- l: Tiền tố đi với %d, %x, %g để chỉ số nguyên dài (VD: %ld).

### ***b. Nhập/xuất dữ liệu***

Thư viện nhập/xuất trong C là stdio.h, stdio.h cung cấp cho ta hai hàm để nhập xuất dữ liệu lần lượt là scanf() và printf().

Hàm print() có nguyên mẫu của hàm (prototype) như sau:

```
int printf("<chuỗi định dạng>", <danh sách tham số>);
```

Trong đó, int là kiểu trả về của hàm, là giá trị đại diện cho hàm sau khi hết phạm vi của hàm. Chuỗi định dạng (Format string) có nhiệm vụ định dạng dữ liệu xuất ra màn hình. Danh sách tham số có thể bao gồm biến, hằng số, biểu thức và hàm (function) và được phân cách bằng dấu ",".

Hàm scanf() có nguyên mẫu của hàm (prototype) như sau:

```
int scanf("<chuỗi định dạng>", <danh sách tham số>);
```

Tương tự như hàm printf, danh sách tham số của scanf cũng được phân cách bằng dấu ",". Tuy nhiên, tham số phải được truyền vào dưới dạng tham chiếu, tức truyền vào địa chỉ của biến. Tham chiếu của các kiểu dữ liệu cơ bản (primitive data type) như int, float, char, ... là & (address-of operator) cùng với tên biến. Đối với các kiểu dữ liệu dẫn xuất (ví dụ như chuỗi ký tự), tham chiếu đơn giản là tên biến.

Ví dụ:

```
#include <stdio.h>

int main ()
{
    printf ("Welcome to Stdio!\n");

    int a;
    float b;
    printf ("Enter an integer and a float: ");
    scanf ("%d %f", &a, &b);
    printf ("The integer you've entered is %d\nThe float you've entered
is %f\n", a, b);

    char *s = new char[100];
    printf ("Enter a string: ");
    scanf ("%s", s);
    printf ("The string you've entered is %s\n", s);
    delete[] s;
```



```

int dd, mm, yyyy;
printf ("Enter a date: "); //Format: dd/mm/yyyy
scanf ("%d/%d/%d", &dd, &mm, &yyyy);
printf ("The date you've entered is %2d/%2d/%4d\n", dd, mm, yyyy);

return 0;
}

```

### ***c. Một số vấn đề về nhập chuỗi trong C++***

Do một số lí do kĩ thuật mà phương thức `cin` trong C++ không thể nhập được chuỗi có chứa khoảng trắng. Ví dụ:

```

#include <iostream>
using namespace std;
int main() {
    string name;
    cout << "Enter your string: ";
    cin >> name;
    cout << name;
}

```

Nếu bạn nhập chuỗi “Nguyen Huynh Thao Nhi” thì lúc này `name` chỉ có giá trị là “Nguyen”.

Vậy làm sao để giải quyết vấn đề này ? Cách mà nhiều người hay dùng nhất là sử dụng phương thức nhập của C. Hàm `scanf()` trong C nhận diện khoảng trắng giống như ký tự xuống dòng ‘\n’, như một dấu hiệu để kết thúc chuỗi thu được từ bộ đệm `stdin` (khác với ký tự kết thúc chuỗi ‘\0’). Do đó khi nhập dữ liệu có khoảng trắng, khoảng trắng sẽ bị giữ lại trong `stdin` và hàm `scanf` sẽ không nhận đủ lượng dữ kiện cần thiết. Tuy vậy có một hàm thay thế, đó là hàm `gets()`, cú pháp như sau:

```
char* gets(char* str);
```

Hàm `gets` sẽ nhận khoảng trắng như một ký tự bình thường. Quá trình này sẽ kết thúc khi bạn nhấn enter.

### ***d. Một số vấn đề với nhập xuất dữ liệu***

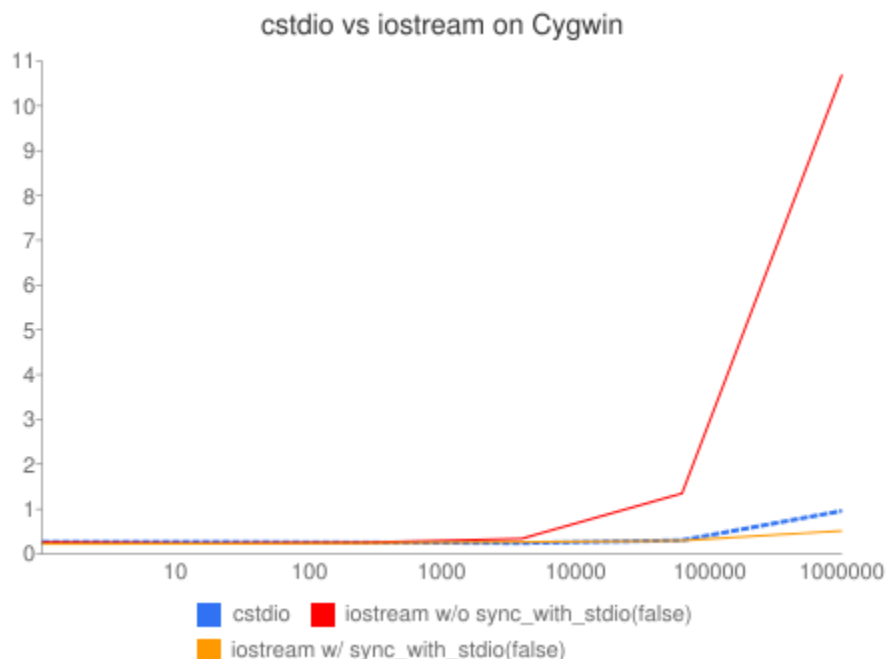
- Không truyền tham chiếu vào hàm `scanf`: Đây là lỗi mà các bạn thường gặp nhất. Với những project nhỏ thì lỗi này có thể phát hiện được dễ dàng. Tuy nhiên, khi làm việc với những dự án lớn, việc phát hiện và khắc phục lỗi này là cực kỳ khó khăn, vì đây không phải là lỗi trong quá trình Build nên ta không xác định được vị trí dòng code bị lỗi. Do đó lỗi này là đặc biệt nghiêm trọng và cần được khắc phục trong quá trình học tập. Cách khắc phục: thêm toán tử `&` vào trước tên biến (primitive data type).

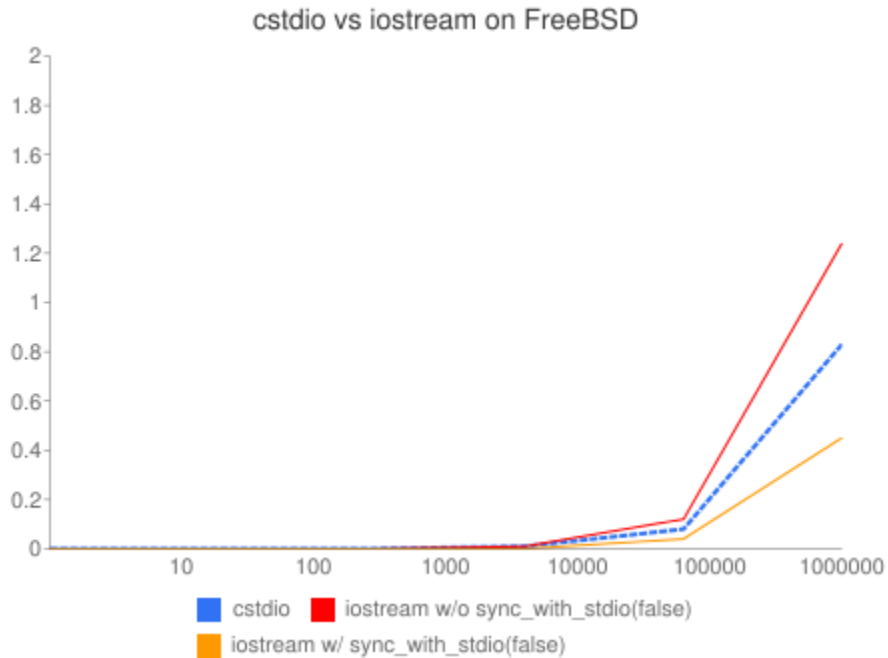
- Trôi dòng lệnh khi xử lý chuỗi: Lỗi này khá phổ biến và gây hoang mang cho người mới học. Trong C, dòng (stream) vào tiêu chuẩn là `stdin`, các hàm như `gets`, `scanf`, `getchar`, .. đều nhận dữ liệu từ `stdin`. Khi trên `stdin` không còn dữ liệu, các hàm nhập dữ liệu sẽ yêu cầu người dùng nhập từ bàn phím. Các hàm trên chỉ nhận đủ dữ kiện mà chúng yêu cầu (trong trường hợp này `scanf` chỉ nhận chuỗi không có khoảng trắng), do đó một phần dữ liệu bị sót lại trên `stdin`, có thể là ký tự '\n' hoặc phần dữ liệu phía sau khoảng trắng. Điều này ảnh hưởng đến các hàm nhập dữ liệu phía sau. Ta có thể khắc phục lỗi này bằng cách làm rỗng bộ đệm `stdin` trước mỗi lần nhập để bảo đảm sự chính xác của chương trình. Câu lệnh **`fflush(stdin)`** sẽ giúp ta làm điều đó.

#### 4. So sánh 2 cách nhập/xuất dữ liệu đã đề cập

Hiện nay, việc sử dụng `scanf/printf` hay `cin/cout` trong các bài tập lập trình tính toán đang là một đề tài tranh cãi lớn trong cộng đồng competitive programmer thế giới do trong một vài trường hợp 2 cách nhập/xuất này sẽ cho thời gian thực hiện khác nhau có khi là đến vài giây. Do đó việc lựa chọn cho mình một cách nhập/xuất hợp lý cũng là một lợi thế lớn trong các cuộc thi lập trình. Tuy nhiên, biểu hiện của `scanf/printf` với `cin/cout` còn phụ thuộc vào cấu hình máy, chương trình dịch, phiên bản C++ nên mọi đánh giá là tương đối, chỉ mang tính chất tham khảo.

Ta sẽ xét một ví dụ với bài tập [Codeforces Beta Round #43 Problem E](#) trên trang web [codeforces.com](#). Dưới đây là biểu đồ cho thấy thời gian chạy của các cách thức nhập/xuất dữ liệu:





Cấu hình hai máy sử dụng để thử như sau:

Cygwin

- CPU: Core2Duo T7600 2.3Ghz
- RAM: 4GB
- OS: Windows XP SP3
- Compiler: gcc 4.3.4 on Cygwin 1.7.7

FreeBSD

- CPU: PentiumM 1.3GHz
- RAM: 1GB
- OS: FreeBSD 8.1-STABLE

Compiler: gcc 4.2.1

Lưu ý: câu lệnh `ios_base::sync_with_stdio(false)` là câu lệnh để tắt đồng bộ hóa hai luồng i/o của C++ và C, điều này giúp giảm thời gian đọc/xuất dữ liệu của `cin/cout` trong C++.

Qua so sánh sự khác biệt của đồ thị và quá trình code hãy tự rút ra cho mình cách nhập/xuất hiệu quả nhất.

## CHƯƠNG III: CẤU TRÚC ĐIỀU KHIỂN

### §1: CẤU TRÚC ĐIỀU KIỆN

Trong cuộc sống, không phải lúc nào những điều kiện, yêu cầu mà chúng ta đặt ra cũng được thỏa mãn. Do, khi đứng trước một kết quả của điều kiện thì ta luôn sẽ chuẩn bị một cách hành động phù hợp với kết quả đó. Và đó là một điều hết sức bình thường mà hằng ngày đang diễn ra quanh ta. Trong lập trình cũng vậy, cấu trúc điều kiện là một cấu trúc cực kì cơ bản mà bất kì ngôn ngữ lập trình nào cũng có. C++ cũng không ngoại lệ. Hôm nay ta sẽ tìm hiểu về cấu trúc điều kiện trong C++.

#### 1. Cấu trúc điều kiện If

Cấu trúc này có dạng như sau:

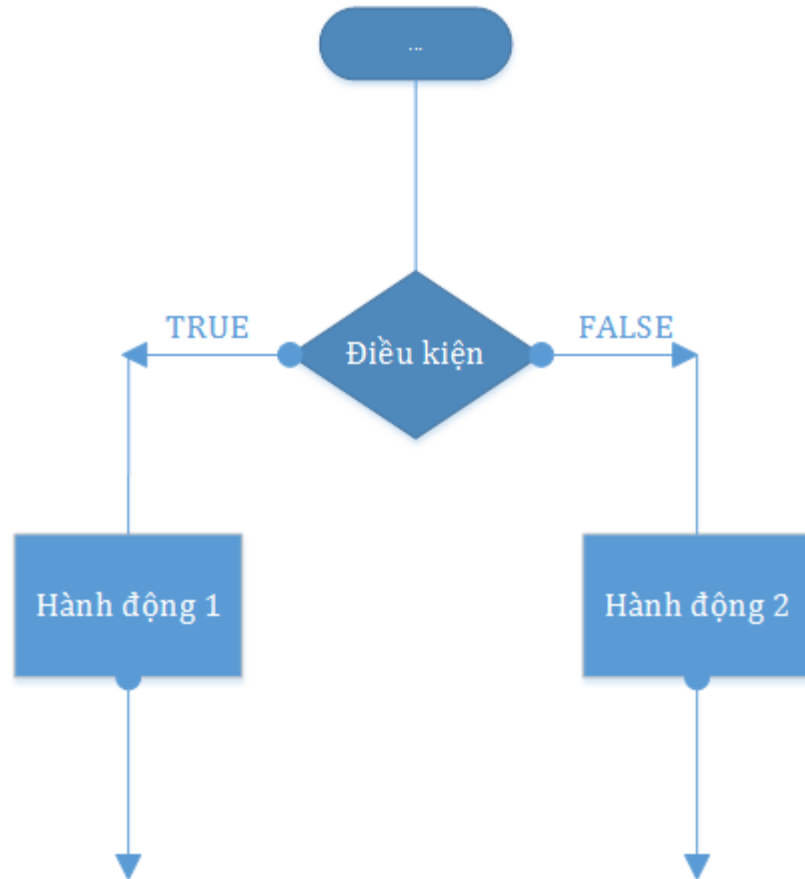
**If** <điều kiện> <hành động 1>; **else** <hành động 2>;

Trong đó:

- <điều kiện> là một biểu thức trả về giá trị true hoặc false.
- <hành động 1>, <hành động 2> là một lệnh hoặc khối lệnh mô tả công việc mà chương trình sẽ phải thực hiện.

Nếu <điều kiện> trả về kết quả là true thì <hành động 1> sẽ được thực hiện, ngược lại <hành động 2> sẽ được thực hiện. Ví dụ

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int a = 0;
    if (a == 0) {
        cout << 0;
    } else {
        cout << 1;
    }
}
```



Ở <hành động 2> và <hành động 1> ta có thể sử dụng thêm nhiều cấu trúc điều kiện lồng nhau. Do đó cấu trúc điều có lưu đồ thuật toán tổng quát có dạng cây nhị phân độ sâu  $n$  – với  $n$  là số lượng cấu trúc điều kiện lồng nhau. Ví dụ:

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

int main () {
    // Khai bao bien cuc bo:
    int a = 100;

    // kiem tra dieu kien cua bieu thuc boolean
    if (a == 10) {
        // Neu dieu kien la true thi in dong sau
        cout << "Gia tri cua a la 10" << endl;
    } else if (a == 20) {
        // neu dieu kien else if la true
        cout << "Gia tri cua a la 20" << endl;
    }
}
```

```

    } else if (a == 30) {
        // eu dieu kien else if la true
        cout << "Gia tri cua a la 30" << endl;
    } else {
        // neu cac dieu kien tren khong la true thi in dong sau
        cout << "Gia tri cua a khong ket noi voi cac dieu kien tren" <<
endl;
    }
    cout << "Gia tri chinh xac cua a la: " << a << endl;

    return 0;
}

```

Chạy đoạn chương trình trên ta sẽ được kết quả là:

```

C:\Users\Administrator\Desktop\workspace\cppbook\sourcecode\myproject\bin\Debug\myproject.exe
Gia tri cua a khong ket noi voi cac dieu kien tren
Gia tri chinh xac cua a la: 100

Process returned 0 (0x0)   execution time : 0.017 s
Press any key to continue.

```

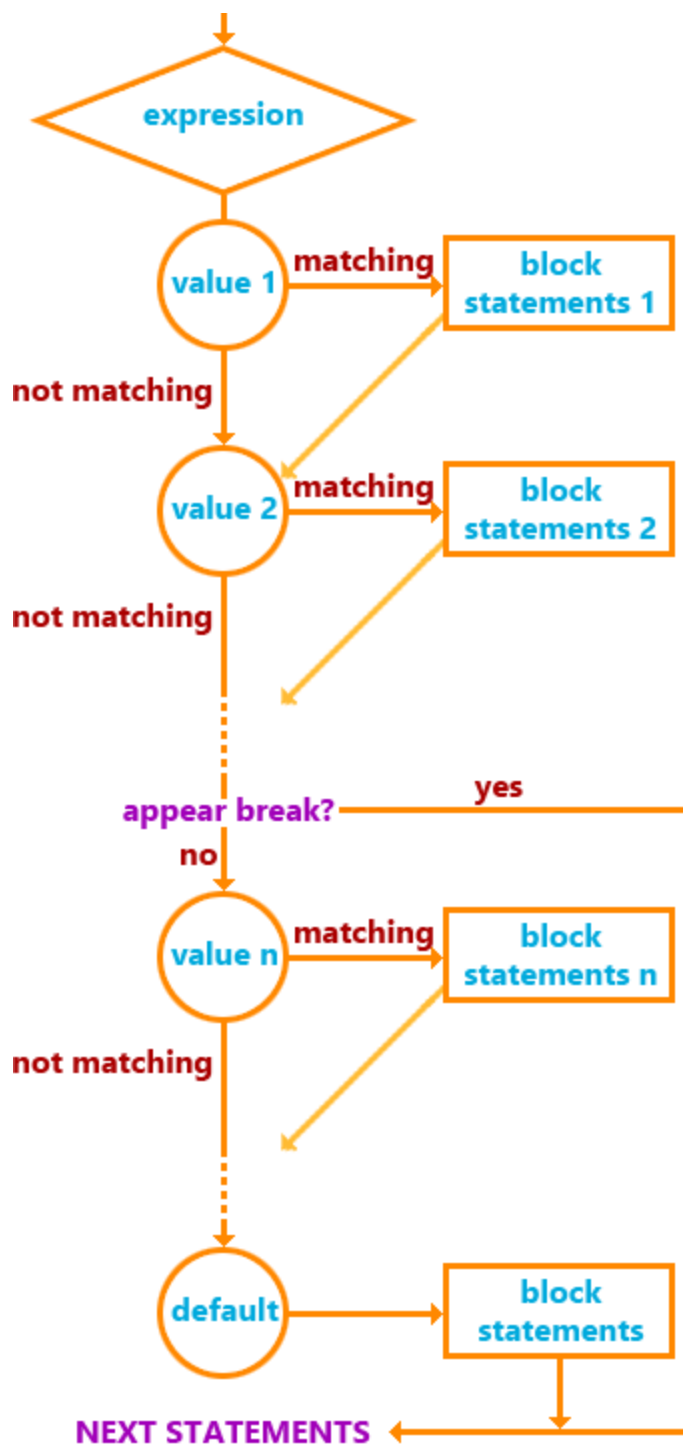
## 2. Cấu trúc điều kiện switch...case

Cũng giống như if..else, switch...case là một cấu trúc điều kiện trong C++.

Nhưng trong một cấu trúc điều kiện switch case thì sẽ có số lựa chọn công việc tùy ý, thay vì với if..else chỉ là hai. Switch so sánh một biểu thức nguyên với một danh sách giá trị các số nguyên, các hằng kí tự hoặc biểu thức hằng. Mỗi giá trị trong danh sách chính là một case (trường hợp) trong khối lệnh của switch. Ngoài ra, trong khối lệnh switch còn có thể có một default case (trường hợp mặc định) có thể có hoặc không. Mặt khác, trong mỗi trường hợp còn chứa các khối lệnh chờ được thực thi. Biểu thức nguyên trong switch được tính toán và kiểm tra lần lượt với giá trị của từng case. Đầu tiên, nó sẽ được so sánh với giá trị của case đầu tiên, nếu bằng nhau thì sẽ thực hiện các lệnh (statement) trong case này cho đến khi nó gặp được từ khoá break. Khi đó, cấu trúc switch...case kết thúc. Chương trình sẽ thực hiện tiếp những dòng lệnh sau cấu trúc switch...case. Ngược lại, nếu như giá trị biểu thức nguyên không bằng giá trị case đầu tiên thì nó sẽ tiếp tục so sánh đến giá trị của case thứ hai và tiếp tục thực hiện như những bước trên. Giả sử, đến cuối cùng vẫn không tìm được giá trị bằng nó thì các khối lệnh trong default sẽ được thực hiện nếu như có tồn tại default. Cấu trúc của câu lệnh điều kiện này như sau:

```
switch (expression)
{
    case constant_1:
    {
        Statements;
        break;
    }
    case constant_2:
    {
        Statements;
        break;
    }
    .
    .
    .
    case constant_n:
    {
        Statements;
        break;
    }
    default:
    {
        Statements;
    }
}
```





Ví dụ cụ thể:

```
#include <iostream>
using namespace std;
int main() {
    int month;
    cout << "Month: " << endl;
    cin >> month;

    switch (month) {
        case 1: {
            cout << "January" << endl;
            break;
        } case 2: {
            cout << "February" << endl;
            break;
        } case 3: {
            cout << "March" << endl;
            break;
        } case 4: {
            cout << "April" << endl;
            break;
        } case 5: {
            cout << "May" << endl;
            break;
        } case 6: {
            cout << "June" << endl;
            break;
        } case 7: {
            cout << "July" << endl;
            break;
        } case 8: {
            cout << "August" << endl;
            break;
        } case 9: {
            cout << "September" << endl;
            break;
        } case 10: {
            cout << "October" << endl;
            break;
        } case 11: {
            cout << "November" << endl;
            break;
        } case 12: {
            cout << "December" << endl;
            break;
        } default: {
            cout << "Input is false" << endl;
        }
    }
}
```

```

    return 0;
}

```

Chạy chương trình này ta sẽ được kết quả như sau:

```

C:\Users\Administrator\Desktop\workspace\cppbook\sourcecode\myproject\bin\Debug\myproject.exe
Month:
1
January

Process returned 0 (0x0)   execution time : 4.531 s
Press any key to continue.

```

Một số lưu ý khi dùng cấu trúc rẽ nhánh switch...case:

- Các giá trị của mỗi case phải cùng kiểu dữ liệu với giá trị của biểu thức được so sánh.
- Số lượng các case là không giới hạn nhưng chỉ có thể có duy nhất một default.
- Giá trị của các case là một hằng số và các giá trị của các case phải khác nhau.
- Từ khóa break có thể sử dụng hoặc không. Nếu không được sử dụng thì chương trình sẽ không kết thúc cấu trúc switch...case khi đã thực hiện hết khối lệnh của case đó. Thay vào đó, nó sẽ thực hiện tiếp các khối lệnh tiếp theo cho đến khi gặp từ khóa break hoặc dấu " } " cuối cùng của cấu trúc switch...case. Vì vậy, các bạn có thể sử dụng một khối lệnh cho nhiều trường hợp khác nhau.

## §2: CẤU TRÚC LẶP

### 1. Lặp

Trong nhiều bài toán thực tế, ta phải giải quyết bằng cách lặp đi, lặp lại một thao tác để tìm ra câu trả lời tối ưu. Ta xét một ví dụ:

*Đếm tất cả các số từ 1 đến 1000 chỉ chia hết cho 30 mà không chia hết cho 16.*

Với bài này chỉ cần nhẩm cũng tính ra kết quả. Tuy nhiên để trực quan ta sẽ giải quyết bằng thuật toán đơn giản như sau:

- **Bước 1:** Khởi tạo biến lưu kết quả bằng 0, biến chạy  $i = 1$ .
- **Bước 2:** Kiểm tra nếu  $i$  chia hết cho 30 mà không chia hết cho 16 thì:
  - Tăng kết quả lên 1.
  - Tăng biến chạy  $i$  lên 1.

Ngược lại, tăng biến chạy  $i$  lên 1.

- **Bước 3:** Kiểm tra nếu  $i > 1000$  thì chuyển đến bước 5.
- **Bước 4:** Quay lại bước 2.
- **Bước 5:** In kết quả và dừng chương trình.

Ta có thể thấy những bước từ bước 2 đến bước 4 phải thực hiện lặp lại đến 1000 lần. Khi mô tả bằng code ta không thể dòng hơn vài nghìn dòng code để làm những công việc giống nhau như vậy được. Do đó, cấu trúc lặp đã ra đời và là cấu trúc cơ bản của mọi loại ngôn ngữ lập trình.

### 2. Cấu trúc lặp for

Cấu trúc for có prototype tiêu chuẩn như sau:

```
for (Initialization statement; Condition; Update statement) {
    //Code while condition is true
}
```

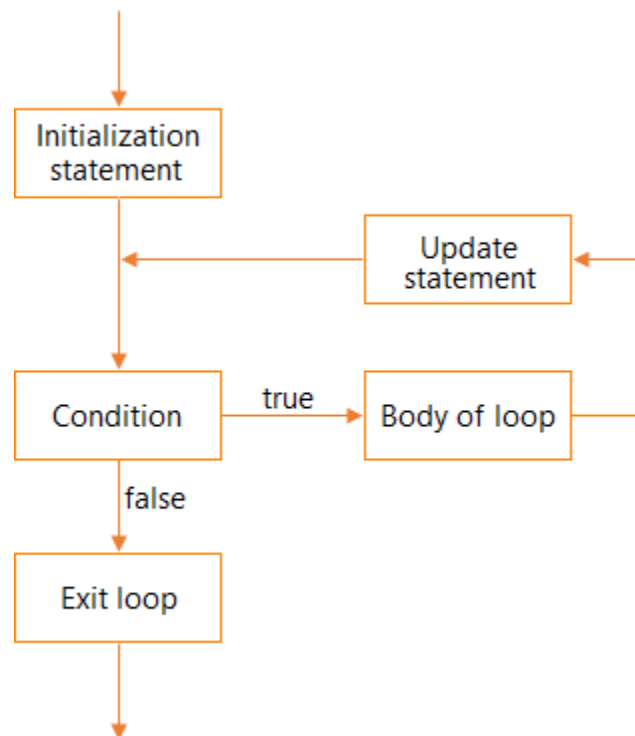
Trong đó:

- Initialization statement là biểu thức khởi tạo các giá trị ban đầu cho các tham số. Các tham số này dùng để kiểm tra điều kiện trong biểu thức Condition.
- Condition là biểu thức điều kiện của vòng lặp. Vòng lặp sẽ kết thúc khi biểu thức này trả về giá trị false.

- Update statement là biểu thức cập nhật giá trị cho các tham số.

Ngoài ra, nếu phần code trong vòng lặp for chỉ có 1 dòng, ta có thể bỏ cặp dấu {}.

**Nguyên tắc hoạt động:** Bắt đầu với việc khởi tạo giá trị ban đầu cho các biến, quá trình lặp sẽ diễn ra liên tục, cho tới khi biểu thức Condition không còn đúng nữa. Các biến sẽ được cập nhật trong phần Update statement hoặc cả ở trong vòng lặp (nếu có). Sơ đồ hoạt động của cấu trúc for như sau:



**Một số lưu ý:** Vòng lặp sẽ kết thúc khi và chỉ khi biểu thức điều kiện trả về giá trị false, do đó các bạn cần lưu ý biểu thức Update statement để thu được kết quả mong muốn. Sau dòng lệnh for không có dấu “;”, nếu có chương trình sẽ báo lỗi hoặc cho kết quả không mong muốn. Các biến nếu được khai báo trong dòng lệnh for sẽ chỉ có phạm vi sử dụng trong vòng lặp. Cũng giống như các biến được khai báo trong một scope thì chỉ có phạm vi sử dụng trong scope đó. Các biểu thức Initialization statement, Condition và Update statement có thể có hoặc không, nhưng bắt buộc phải có đủ hai dấu “;”. Khi vắng mặt các biểu thức đó, ta phải tự định nghĩa chúng để tránh chương trình bị lặp vô hạn. Tuy nhiên, ngoại trừ các trường hợp đặc biệt, ta nên định nghĩa các biểu thức đó trong dòng lệnh for để tạo tính nhất quán và chuẩn mực chung của các lập trình viên. Để dừng vòng lặp một cách ép buộc (force exit), ta

sử dụng lệnh break. Lệnh này cần được đặt bên trong đoạn code của vòng lặp. Ngoài ra còn có lệnh continue để bỏ qua lần lặp hiện tại và thực hiện lần lặp kế tiếp.

Ví dụ:

```
#include <stdio.h>

int main() {
    int s1 = 0;
    for (int i = 0; i < 10; i++) //Full type
        s1 += i;
    printf("%d\n", s1);
    int s2 = 0;
    for (int j = 0;;j++) {
        if (j % 2 == 0) {
            s2 += j;
        }
        if (j > 20) //Condition {
            break;
        }
    }
    printf("%d\n", s2);
    int s3 = 0;
    int k = 0;          //Initialization
    for (;;) {
        if (k % 3 == 0) {
            continue;
        }
        if (s3 > 100)    //Condition {
            break;
        }
        s3 += k;
        k++;            //Update
    }
    printf("%d\n", s3);
    return 0;
}
```

Trước khi chạy đoạn chương trình trên hãy dự đoán xem sẽ có chuyện gì xảy ra, và vì sao lại như vậy.

### 3. Cấu trúc While

Cấu trúc của vòng lặp while như sau:

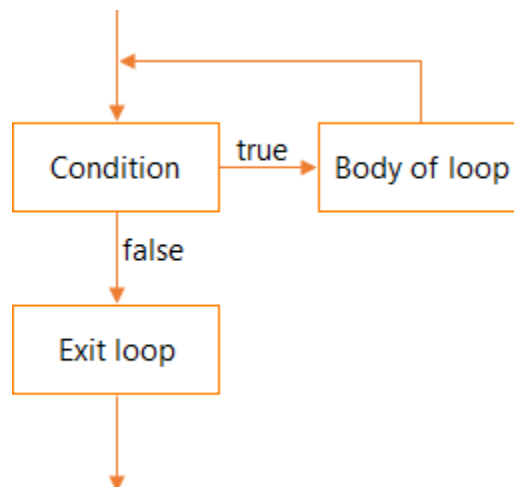
```
while (condition) {
```

```
// Code while condition is true
}
```

Các biểu thức Initialization statement và Update statement không xuất hiện trong cấu trúc while. Do đó ta cần hiện thực chúng trước và trong vòng lặp để chương trình không bị lặp vô hạn. Đoạn code trong thân vòng lặp (body of loop) có thể chứa các biểu thức tính toán logic, các cấu trúc điều kiện (if ... else, switch ... case) hoặc cả các cấu trúc lặp khác. Nhờ đó ta có thể xây dựng được các chu trình lặp lồng nhau (sử dụng để thao tác với mảng nhiều chiều, ...).

**Nguyên tắc hoạt động:** Giống như cấu trúc for, quá trình lặp trong while sẽ diễn ra liên tục cho đến khi biểu thức Condition trả về giá trị false. Một điều cần lưu ý là biểu thức Condition tồn tại trước khi quá trình lặp bắt đầu. Do đó vòng lặp có thể sẽ không được thực hiện lần nào, tùy thuộc vào việc khởi tạo giá trị cho các biến có liên quan.

Sơ đồ hoạt động của cấu trúc while như sau:



Vòng lặp while thường được sử dụng trong các trường hợp không biết rõ số lần lặp. Ví dụ sau sẽ giúp các bạn dễ dàng hình dung cách làm việc của vòng lặp while:

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

```

int main()
{
    int s = 0;
    int i = 0;

    while(s < 100)
    {
        i++;
        s += i;
    }

    s = 0;
    i = 0;

    while(i++, s < 100)
        s += i;

    s = 0;
    i = 0;

    while(1)
    {
        i++;
        s += i;

        if(s >= 100)
            break;
    }

    return 0;
}

```

Ba đoạn code trên đây thực hiện chính xác một công việc giống nhau (chỉ quan tâm đến kết quả, không quan tâm đến giá trị của các biến trong suốt quá trình). Bạn có thể lồng ghép nhiều biểu thức vào Condition, phân cách với nhau bởi dấu “,” như ở đoạn code thứ hai. Khi đó biểu thức sau dấu “,” cuối cùng chính là điều kiện chính để dừng vòng lặp.

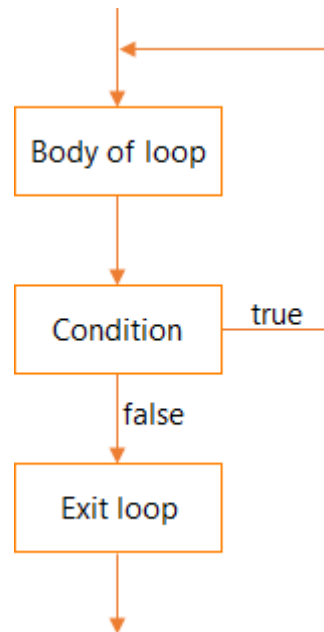
#### 4. Cấu trúc do...while

Vòng lặp while có cấu trúc như sau:



```
do
{
    // Code
} while (Condition);
```

Do...while có đầy đủ các tính chất mà một vòng lặp cần phải có. Tuy nhiên, khác với cấu trúc while, biểu thức điều kiện của do ... while được đặt phía sau đoạn code, do đó vòng lặp sẽ được thực hiện ít nhất một lần, bất kể các giá trị đầu vào có thoả biểu thức điều kiện hay không. Lưu đồ thuật toán như sau:



Ví dụ:

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

int main()
{
    int n;

    do
    {
        cin >> n;
    } while(n <= 0);

    int i;
    int s = 0;

    do
    {
```

```
        cin >> i;  
    } while(s += i, --n);  
  
    return 0;  
}
```

## CHƯƠNG IV: DỮ LIỆU CÓ CẤU TRÚC

### §1: MẢNG

Khi cần lưu trữ một số lượng lớn các phần tử có cùng kiểu dữ liệu, chẳng hạn như cần lưu trữ 30 số nguyên được nhập từ bàn phím, ta có thể dùng 30 biến số nguyên riêng biệt để lưu trữ. Tuy nhiên, cách trên có một số nhược điểm như sau:

- Khi cần lưu trữ một số lượng lớn các phần tử có cùng kiểu dữ liệu, chẳng hạn như cần lưu trữ 30 số nguyên được nhập từ bàn phím, ta có thể dùng 30 biến số nguyên riêng biệt để lưu trữ. Tuy nhiên, cách trên có một số nhược điểm như sau:
- Phải thực hiện thủ công các thao tác với biến đó. Do các biến là độc lập nên ta không thể sử dụng các cấu trúc lặp để thao tác với nó.
- Chương trình sẽ dài và việc nâng cấp cũng trở nên khó khăn.

Do đó, việc nhóm các phần tử đó lại với một cái tên duy nhất sẽ khắc phục được những nhược điểm trên. Các phần tử vẫn hoạt động độc lập, nhưng dưới một tên duy nhất, cùng với chỉ số (index) của nó để phân biệt với các phần tử khác.

#### 1. Khai báo và sử dụng mảng

Cách khai báo mảng như sau:

**<Kiểu dữ liệu> <Tên mảng>[số lượng phần tử];**

Kiểu dữ liệu của mảng sẽ quy định kiểu dữ liệu của các phần tử có trong mảng. Tên mảng được đặt tên theo quy tắc đặt tên biến trong C/C++. Khi đó, hệ thống sẽ tìm một vùng nhớ liên tiếp phù hợp để cấp phát cho biến mảng. Lưu ý, số lượng phần tử phải là một hằng số. Lưu ý, phần tử đầu tiên của mảng là 0, do đó nếu khai báo mảng n phần tử thì phần tử cuối cùng trong mảng là n - 1. Ví dụ, ta có mảng `balance = {1000.0, 2.0, 3.4, 7.0, 50.0}`.

	0	1	2	3	4
<b>balance</b>	1000.0	2.0	3.4	7.0	50.0

Để truy cập tới phần tử của mảng ta sẽ dùng cú pháp:

**<tên mảng>[<index cần truy cập>]**

Ví dụ, `balance[2] = 3.4`.

Ngoài ra, ta còn có thể khai báo mảng kèm theo giá trị khởi tạo của mảng. Ví dụ:

```
int main()
{
    int a[10];                //Explicitly Declare
    for(int i = 0; i < 10; i++)
    {
        scanf("%d", &a[i]);
    }
    for(int i = 0; i < 10; i++)
    {
        printf("%d ", a[i]);
    }
    printf("\n");

    int b[10] = {7, 4, 1, 1};    //Declare and set value
    for(int i = 0; i < 10; i++)
    {
        printf("%d ", *(b + i));    //Addition with pointer
    }
    printf("\n");

    char str [] = "Welcome to Stdio";    //Implicitly Declare
    printf("%d", strlen(str));

    return 0;
}
```

## 2. Mảng nhiều chiều

Khi mỗi phần tử của mảng có kiểu dữ liệu là một mảng khác, ta được mảng 2 chiều. Khi đó, ngoài index của các phần tử trong mảng, mỗi phần tử trong các phần tử đó cũng có index của nó. Do đó để truy xuất đến từng phần tử nhỏ, ta sử dụng cả 2 chỉ số index để chỉ ra vị trí của nó trong mảng 2 chiều.

Chúng ta xét ví dụ sau để nắm được cách khai báo và sử dụng mảng 2 chiều.

```
#include <stdio.h>
#include <string.h>
int main()
{
```

```

int a[3][3];
for(int i = 0; i < 3; i++)
{
    for(int j = 0; j < 3; j++)
    {
        scanf("%d", &a[i][j]);
    }
}
for(int i = 0; i < 3; i++)
{
    for(int j = 0; j < 3; j++)
    {
        printf("%d ", a[i][j]);
    }
    printf("\n");
}

int b[3][3] = {{1, 3}, {2, 4, 6}};
for(int i = 0; i < 3; i++)
{
    for(int j = 0; j < 3; j++)
    {
        printf("%d ", b[i][j]);
    }
    printf("\n");
}

return 0;
}

```

Tương tự như vậy ta có thể sử dụng mảng n chiều trong C++ với cùng cách thức làm việc như mảng 1 chiều và 2 chiều.

### 3. Sử dụng mảng bằng cách cấp phát động

Khi lập trình, có những trường hợp chúng ta không xác định được số lượng ô nhớ cần thiết khi code, chỉ có thể xác định được khi chương trình đã được thực thi. Do đó sẽ là một sự lãng phí khi ta không dùng hết bộ nhớ được cấp phát, hoặc cấp phát không đủ bộ nhớ để sử dụng. Việc cấp phát động giúp chúng ta chủ động trong việc cấp phát bộ nhớ, tránh các trường hợp thừa/thiếu bộ nhớ.

Một ưu điểm khác của việc cấp phát động so với cấp phát tĩnh là ta có thể cấp phát vùng nhớ lớn, miễn sao chương trình tìm được một vùng nhớ liên tục trên RAM có kích thước phù hợp. Vùng nhớ Stack Segment được quản lý chặt chẽ bởi hệ thống, nhưng dung lượng khá nhỏ nên sẽ không đáp ứng hết được nhu cầu của lập trình viên.

Ngoài ra còn một số lợi ích khác của cấp phát động như sau:

- Có thể cấp phát vùng nhớ có kích thước bất kỳ bằng cách truyền tham số vào trong cặp dấu [ ] khi cấp phát vùng nhớ. Việc cấp phát tĩnh không cho phép truyền vào tham số để cấp phát vùng nhớ.
- Có thể tái sử dụng vùng nhớ sau khi được huỷ bằng toán tử delete.

Để cấp phát động bộ nhớ cho đối tượng, ta sử dụng toán tử new. Khi cần cấp phát nhiều ô nhớ cho biến, ta thay toán tử new bằng toán tử new[]. Cú pháp như sau:

```
int *a = new int;
int *b = new int[10];
```

Khi đó, chương trình sẽ cấp phát cho các biến một vùng nhớ liên tục có kích thước là chỉ số \* sizeof(kiểu dữ liệu). Các biến quản lý vùng nhớ được cấp phát là một biến con trỏ, lưu trữ địa chỉ ô nhớ đầu tiên của vùng nhớ đó. Các biến con trỏ này được lưu trên Stack Segment, trong khi vùng nhớ mà nó trỏ đến sẽ được lưu trên Data Segment (Vấn đề này chúng ta sẽ tìm hiểu kĩ hơn ở phần sau).

Sau khi thao tác xong, nếu không sử dụng vùng nhớ này nữa, ta sẽ huỷ nó bằng toán tử delete hoặc delete[] tương ứng. Nếu không huỷ vùng nhớ này trước khi chương trình kết thúc sẽ dẫn đến lỗi Memory Leak (rò rỉ vùng nhớ), vùng nhớ đó sẽ không thể được tái sử dụng cho đến khi khởi động lại máy tính (ngắt nguồn điện ở RAM). Do đó, chúng ta cần đặc biệt chú ý tới điều này, có new là phải có delete.

Ví dụ:

```
#include <stdio.h>

int main()
{
    int *a = new int[10];

    for (int i = 0; i < 10; i++)
    {
        scanf("%d", &a[i]);
    }
    for (int i = 0; i < 10; i++)
    {
        printf("%-4d", *(a + i));
    }

    delete[] a;
    return 0;
}
```

## §2: CHUỖI (XÂU KÍ TỰ)

Khái niệm chuỗi ký tự do con người đặt ra để thuận tiện trong việc sử dụng. Có thể hiểu đơn giản, chuỗi là tập hợp các ký tự được lưu trữ liên tiếp trong vùng nhớ máy tính. Mỗi ký tự có kích thước 1 byte, do đó kích thước của chuỗi ký tự sẽ bằng tổng số các ký tự có mặt trong chuỗi. Bài viết này sẽ giúp các bạn có được những kiến thức cơ bản trong thao tác xử lý chuỗi trong ngôn ngữ C/C++.

### 1. Chuỗi là gì ?

Khái niệm chuỗi ký tự do con người đặt ra để thuận tiện trong việc sử dụng. Có thể hiểu đơn giản, chuỗi là tập hợp các ký tự được lưu trữ liên tiếp trong vùng nhớ máy tính. Mỗi ký tự có kích thước 1 byte, do đó kích thước của chuỗi ký tự sẽ bằng tổng số các ký tự có mặt trong chuỗi.

Thực chất, chuỗi ký tự là một mảng dữ liệu (array), mỗi phần tử trong mảng có kích thước 1 byte. Do đó, chuỗi ký tự có đầy đủ các tính chất của một mảng dữ liệu. Ngoài ra, nó còn có nhiều đặc tính và phương thức riêng để thao tác được dễ dàng hơn.

### 2. Chuỗi trong C

Dạng chuỗi này bắt nguồn từ ngôn ngữ C và tiếp tục được hỗ trợ trong C/C++. Chuỗi trong ngôn ngữ lập trình C thực chất là mảng một chiều của các ký tự mà kết thúc bởi một ký tự null '\0'.

Phần khai báo và khởi tạo dưới đây tạo ra một chuỗi bao gồm một từ "Hello". Để giữ các giá trị null tại cuối của mảng, cỡ của mảng các ký tự bao gồm một chuỗi phải nhiều hơn số lượng các ký tự trong từ khóa "Hello".

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

Ngoài ra ta còn cách khác để khởi tạo chuỗi:

```
char loiChao[] = "Hello";
```

Dưới đây là phần biểu diễn ô nhớ cho đoạn chuỗi trên trong ngôn ngữ C/C++:

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

Thực tế, ta không đặt ký tự null tại vị trí cuối cùng của biến hằng số. Bộ biên dịch C tự động thêm '\0' tại vị trí cuối cùng của chuỗi khi nó khởi tạo chuỗi. Dưới đây là bảng các hàm hay dùng với chuỗi theo phong cách C.

STT	Hàm & Mục đích
1	<b>strcpy(s1, s2);</b> Sao chép chuỗi s2 cho chuỗi s1.
2	<b>strcat(s1, s2);</b> Nối chuỗi s2 vào cuối chuỗi s1.
3	<b>strlen(s1);</b> Trả về độ dài của chuỗi s1.
4	<b>strcmp(s1, s2);</b> Trả về 0 nếu s1 và s2 là như nhau; nhỏ hơn 0 nếu s1<s2; lớn hơn 0 nếu s1>s2.
5	<b>strchr(s1, ch);</b> Trả về con trỏ tới vị trí đầu tiên của ch trong s1.
6	<b>strstr(s1, s2);</b> Trả về con trỏ tới vị trí đầu tiên của chuỗi s2 trong chuỗi s1.



Có nhiều hạn chế của việc dùng chuỗi theo phong cách C có thể kể đến như là:

- Người lập trình phải chủ động kiểm soát bộ nhớ cấp phát cho chuỗi ký tự. Nói chung là phải am hiểu và rất thông thạo về kỹ thuật dùng bộ nhớ và con trỏ thì chương trình mới tránh được các lỗi về kỹ thuật.
- Không thể gán giá trị hay sử dụng phép toán + (ghép chuỗi) và các phép toán so sánh như: > (lớn hơn), < (nhỏ hơn),... mà phải gọi các hàm thư viện trong <string.h>;
- Nếu dùng kỹ thuật cấp phát động thì phải quản lý việc cấp thêm bộ nhớ khi chuỗi dài ra (chẳng hạn do ghép chuỗi) và phải hủy bộ nhớ (khi không dùng nữa) để tránh việc cạn kiệt bộ nhớ của máy tính trong trường hợp có nhiều chương trình hoạt động đồng thời.

### 3. Chuỗi trong STL (Standard Template Library) C++

#### a. String trong C++

Thư viện chuẩn STL (Standard Template Library) cung cấp kiểu string (xâu ký tự), giúp chúng ta tránh khỏi hoàn toàn các phiền phức nêu trên. Tuy vậy, nếu ta muốn sử dụng các hàm của C-string, ta cũng có thể chuyển đổi giữa hai loại chuỗi này:

- Chuyển từ string sang C-string: `string s; s.c_str();`
- Chuyển từ C-string sang string:  
`char* s_old = "ABC"; string s(s_old);`

#### b. Các phương thức phép toán với string (method & operation)

Các toán tử +, += dùng để ghép hai chuỗi và cũng để ghép một ký tự vào một chuỗi.

Các phép so sánh theo thứ tự từ điển: == (bằng nhau), != (khác nhau), > (lớn hơn), >= (lớn hơn hay bằng), < (nhỏ hơn), <= (nhỏ hơn hay bằng).

Hàm length() trả về kết quả là độ dài của chuỗi.

Do có tính chất mảng nên ta có thể xử lý chuỗi như một mảng các ký tự.

Các phương thức thông dụng:

- Phương thức substr(int pos, int nchar) trích ra chuỗi con của một chuỗi cho trước, ví dụ str.substr(2,4) trả về chuỗi con gồm 4 ký tự của chuỗi str kể từ ký tự ở vị trí thứ 2 (ký tự đầu tiên của chuỗi ở vị trí 0). Ví dụ:

```
//get substring
#include <iostream>
#include <string>
#include <conio.h>
using namespace std;
int main ()
{
    string s="ConCho chay qua rao";
    cout << s.substr(2,4) << endl;
    // cout << new string(str.begin()+2, str.begin()+2+4);
}
```

```

    getchar();
    return 0;
}

```

- Phương thức insert() chèn thêm ký tự hay chuỗi vào một vị trí nào đó của chuỗi str cho trước. Có nhiều cách dùng phương thức này:
  - str.insert(int pos, char\* s): chèn s (kiểu c-string) vào vị trí pos của str.
  - str.insert(int pos, string s): chèn s (kiểu string) vào vị trí pos của str.
  - str.insert(int pos, int n, int ch): chèn n lần ký tự ch vào vị trí pos của chuỗi str.
  - Ví dụ:

```

// inserting into a string
#include <iostream>
#include <string>
#include <conio.h>
using namespace std;
int main ()
{
    string str="day la .. xau thu";
    string istr = "them";
    str.insert(8, istr);
    cout << str << endl;
    getchar();
    return 0;
}

```

- Phương thức str.erase(int pos, int n) xóa n ký tự của chuỗi str kể từ vị trí pos. Nếu không quy định giá trị n thì tất cả các giá trị của str từ pos trở đi sẽ bị xóa. Ví dụ:

```

// erase from a string
#include <iostream>
#include <string>
#include <conio.h>
using namespace std;
int main () {
    string str="day cung la xau thu";
    str.erase(0, 3); // " cung la xau thu"
    cout << str << endl;
    str.erase(6, 2);
    cout << str << endl; // " cung xau thu"
    getchar();
    return 0;
}

```

- So sánh:

- Bạn có thể đơn giản là sử dụng những toán tử quan hệ (==, !=, <, <=, >=) được định nghĩa sẵn. Tuy nhiên, nếu muốn so sánh một phần của một chuỗi thì sẽ cần sử dụng phương thức compare():

```
int compare ( const string& str ) const;
int compare ( const char* s ) const;
int compare ( size_t pos1, size_t n1, const string& str )
const;
int compare ( size_t pos1, size_t n1, const char* s ) const;
int compare ( size_t pos1, size_t n1, const string& str,
size_t pos2, size_t n2 ) const;
int compare ( size_t pos1, size_t n1, const char* s, size_t
n2) const;
```

- Hàm trả về 0 khi hai chuỗi bằng nhau và lớn hơn hoặc nhỏ hơn 0 cho trường hợp khác Ví dụ:

```
// comparing apples with apples
#include <iostream>
#include <string>
using namespace std;
int main ()
{
    string str1 ("green apple");
    string str2 ("red apple");
    if (str1.compare(str2) != 0)
        cout << str1 << " is not " << str2 << "\n";
    if (str1.compare(6,5,"apple") == 0)
        cout << "still, " << str1 << " is an apple\n";
    if (str2.compare(str2.size()-5,5,"apple") == 0)
        cout << "and " << str2 << " is also an apple\n";
    if (str1.compare(6,5,str2,4,5) == 0)
        cout << "therefore, both are apples\n";
    return 0;
}
```

- Tìm kiếm và thay thế:

- Phương thức find() tìm kiếm xem một ký tự hay một chuỗi nào đó có xuất hiện trong một chuỗi str cho trước hay không. Có nhiều cách dùng phương thức này:

str.find(int ch, int pos = 0); tìm ký tự ch kể từ vị trí pos đến cuối chuỗi str

str.find(char \*s, int pos = 0); tìm s (mảng ký tự kết thúc '\0') kể từ vị trí pos đến cuối

str.find(string& s, int pos = 0); tìm chuỗi s kể từ vị trí pos đến cuối chuỗi.

- Nếu không quy định giá trị pos thì hiểu mặc nhiên là 0; nếu tìm có thì phương thức trả về vị trí xuất hiện đầu tiên, ngược lại trả về giá trị -1.

```
//find substring
#include <iostream>
#include <string>
#include <conio.h>
using namespace std;
int main ()
{
    string str="ConCho chay qua rao";
    cout << str.find("chay") << endl; // 7
    cout << (int)str.find("Chay") << endl; // -1
    getch();
    return 0;
}
```

- Hàm tìm kiếm ngược (rfind)

```
//find from back
#include <iostream>
#include <string>
#include <conio.h>
using namespace std;
int main ()
{
    string str="ConCho chay qua chay qua rao";
    cout << str.find("chay") << endl; // 7
    cout << (int)str.rfind("chay") << endl; // 16
    getch();
    return 0;
}
```

- Phương thức replace() thay thế một đoạn con trong chuỗi str cho trước (đoạn con kể từ một vị trí pos và đếm tới nchar ký tự ký tự về phía cuối chuỗi) bởi một chuỗi s nào đó, hoặc bởi n ký tự ch nào đó.

Có nhiều cách dùng, thứ tự tham số như sau:

```
str.replace(int pos, int nchar, char *s);
str.replace(int pos, int nchar, string s);
str.replace(int pos, int nchar, int n, int ch);
```

```
string str="con cho la con cho con. Con meo ko phai la con cho";
str.replace(4, 3, "CH0"); // "con CH0 la con cho con. Con meo ko
phai la con cho";
cout << str << endl;
getchar();
```

- Tách chuỗi:

- Trong việc xử lý xâu ký tự, không thể thiếu được các thao tác tách xâu ký tự thành nhiều xâu ký tự con thông qua các ký tự ngăn cách. Các hàm này có sẵn trong các ngôn ngữ khác như Visual Basic, Java,

hay thậm chí là trong <string.h>. Với STL, chúng ta có thể dễ dàng làm điều này với stringstream:

```
string S = "Xin chao tat ca cac ban"; // Khởi tạo giá trị của  
xâu  
stringstream ss(S); // Khởi tạo stringstream từ xâu S  
  
while (ss >> token) { // Đọc lần lượt các phần của xâu. Các  
phần tách nhau bởi dấu cách hoặc xuống dòng.  
    cout << token << endl;  
}
```

## §2: KIỂU DỮ LIỆU TỰ ĐỊNH NGHĨA

Bên cạnh những kiểu dữ liệu có sẵn như char, int, float,... C++ cũng cho phép chúng ta tự định nghĩa kiểu dữ liệu để phù hợp với mục đích cũng như tạo mã nguồn tối ưu nhất.

### 1. Struct

Struct trong C/C++ Các mảng trong C/C++ cho phép ta định nghĩa một vài loại biến có thể giữ giá trị của một vài thành viên cùng kiểu dữ liệu. Nhưng structure - cấu trúc là một loại dữ liệu khác trong ngôn ngữ lập trình C/C++, cho phép ta kết hợp các dữ liệu khác kiểu nhau. Các biến trong cùng 1 struct có thể cùng kiểu hoặc khác kiểu với nhau.

#### a. Định nghĩa một struct

Cú pháp định nghĩa một struct:

```
struct <Tên struct>
{
    <Kiểu dữ liệu thành viên số 1> <Tên thành viên số 1>;
    <Kiểu dữ liệu thành viên số 2> <Tên thành viên số 2>;
    ...
};
Ví dụ:
struct Subject
{
    string    m_name_subject;
    string    m_pre_subject;
    int       m_hour_per_week;
    string    m_trainer;
};
```

#### b. Sử dụng struct

Để truy cập vào các thành viên của struct ta sử dụng toán tử dấu chấm (.) đặt giữa tên biến kiểu struct và tên thành viên của struct cần truy cập.

Một biến thuộc kiểu struct cũng có thể trở thành tham số của một hàm.

Có thể truy cập thành phần của struct thông qua biến con trỏ.

Dưới đây là ví dụ cho cách sử dụng cấu trúc trong C++:

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

```

/*Declare a struct called Subject*/
struct Subject
{
    string      m_name_subject;           //declare member types
    string      m_pre_subject;
    int         m_hour_per_week;
    string      m_trainer;
};

void PrintSubjectInformation(Subject object)
{
    cout << object.m_name_subject      << endl;
    cout << object.m_trainer            << endl;
    cout << object.m_hour_per_week      << endl;
    cout << object.m_pre_subject        << endl;
}

int main()
{
    Subject subject;

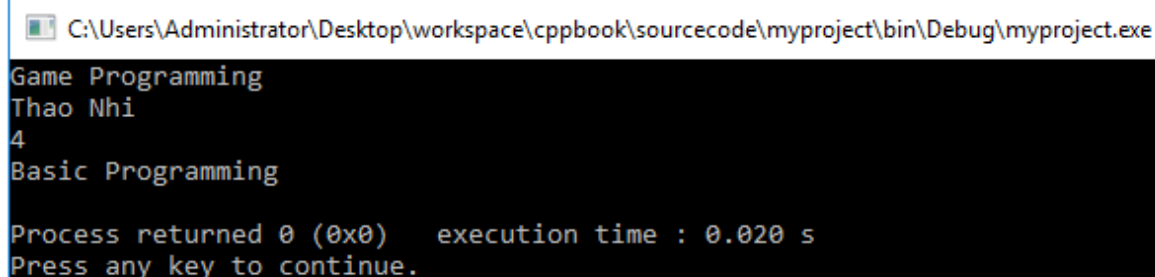
    // Access the member of struct Subject through subject
    subject.m_name_subject = "Game Programming";
    subject.m_pre_subject  = "Basic Programming";
    subject.m_hour_per_week = 4;
    subject.m_trainer      = "Thao Nhi";

    PrintSubjectInformation(subject);

    return 0;
}

```

Chạy chương trình trên ta sẽ được kết quả như sau:



```

C:\Users\Administrator\Desktop\workspace\cppbook\sourcecode\myproject\bin\Debug\myproject.exe
Game Programming
Thao Nhi
4
Basic Programming

Process returned 0 (0x0)   execution time : 0.020 s
Press any key to continue.

```

## 2. Union

Cũng giống như struct, union là một tập hợp các phần tử dữ liệu cùng kiểu hoặc khác kiểu được gộp chung thành một nhóm. Các phần tử này được gọi là thành viên của union

### *a. Định nghĩa một union*

Cú pháp:

```
union <Tên union>
{
    <Kiểu dữ liệu thành viên số 1> <Tên thành viên số 1>;
    <Kiểu dữ liệu thành viên số 2> <Tên thành viên số 2>;
    ...
};
Ví dụ:
union <Tên struct>
{
    <Kiểu dữ liệu thành viên số 1> <Tên thành viên số 1>;
    <Kiểu dữ liệu thành viên số 2> <Tên thành viên số 2>;
    ...
};
```

### ***b. Sử dụng union***

Để truy cập vào các thành viên của union ta sử dụng toán tử dấu chấm (.) đặt giữa tên biến kiểu union và tên thành viên của union cần truy cập.

Một biến thuộc kiểu union cũng có thể trở thành tham số của một hàm.

Có thể truy cập thành viên của union thông qua biến con trỏ.

Ta sẽ xét một ví dụ:

```
#include <iostream>

using namespace std;

/* Declare */
union Post
{
    int        m_iValue;
    float      m_fValue;           //declare member types
    long       m_lValue;
};

void PrintSubjectInformation(Post post)
{
    cout << post.m_iValue << endl;
    cout << post.m_fValue << endl;
    cout << post.m_lValue << endl;
}

int main()
{
    Post post;
    post.m_iValue = 169;           // Access the member
of union
    post.m_fValue = 39.01;
```



```

    post.m_lValue      = 123;

    PrintSubjectInformation(post);

    return 0;
}

```

Chạy chương trình trên ta sẽ được:

```

C:\Users\Administrator\Desktop\workspace\cppbook\sourcecode\myproject\bin\Debug\myproject.exe
123
1.7236e-043
123

Process returned 0 (0x0)   execution time : 0.018 s
Press any key to continue.

```

Khi thay đổi giá trị `post.m_lValue = 123;`. Ta đã làm ảnh hưởng đến giá trị của các thành viên còn lại. Điều này sẽ được giải thích ở phần sau.

### 3. So sánh struct và union

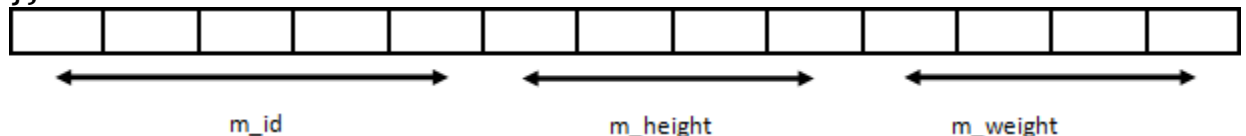
Về mặt ý nghĩa, struct và union cơ bản giống nhau. Tuy nhiên, về mặt lưu trữ trong bộ nhớ, chúng có sự khác biệt rõ rệt như sau:

- struct: Dữ liệu của các thành viên của struct được lưu trữ ở những vùng nhớ khác nhau. Do đó kích thước của một struct chắc chắn lớn hơn kích thước của các thành viên cộng lại. Vì sao lại như vậy? Câu trả lời là vì kích thước của khối nhớ là do hệ thống quy định. Ví dụ: Hệ thống nhớ quy định khối nhớ là 1 byte, thì kích thước của kiểu dữ liệu `sSinhVien` là 13 byte.

```

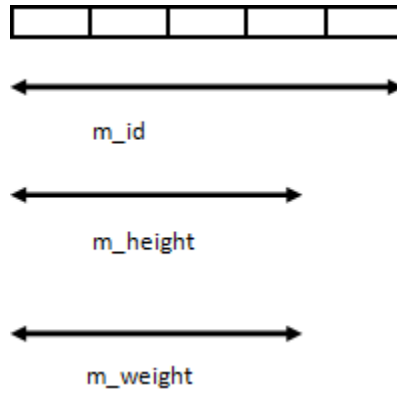
struct sSinhVien
{
    char  m_id[5];
    float m_height;
    float m_weight;
};

```



Thực chất struct trên nếu lấy kích thước thông thường sẽ có hiệu ứng Alignment nên kích thước sẽ không phải là 13 byte, nhưng đó là một vấn đề khác không bàn ở đây.

- union: với union, các thành viên sẽ dùng chung một vùng nhớ. Kích thước của union được tính là kích thước lớn nhất của kiểu dữ liệu trong union. Việc thay đổi nội dung của một thành viên sẽ dẫn đến thay đổi nội dung của các thành viên khác.



## CHƯƠNG V: LẬP TRÌNH CÓ CẤU TRÚC

### §1: HÀM

#### 1. Khái niệm hàm

Nói một cách dễ hiểu ta có thể tưởng tượng hàm là một chương trình con trong chương trình lớn. Hàm nhận (hoặc không) các đối số và trả lại (hoặc không) một giá trị cho chương trình gọi nó.

Ta có 2 số nguyên 3 và 4. Khi truyền vào hàm nó sẽ giúp bạn trả về một giá trị là 7. Hàm này là hàm thực hiện công việc tính tổng của 2 số nhận vào và trả về giá trị là tổng của hai số đó.

Một chương trình dạng C++ console là tập hợp các hàm, trong đó có một hàm chính với tên gọi main(). Khi chạy chương trình, hàm main() sẽ được chạy đầu tiên và gọi đến hàm khác. Kết thúc hàm main() cũng là kết thúc chương trình.

Hàm giúp cho việc phân đoạn chương trình thành những thành phần riêng lẻ, hoạt động độc lập với chương trình lớn, có nghĩa một hàm có thể được sử dụng trong chương trình này mà cũng có thể được sử dụng trong chương trình khác, dễ cho việc kiểm tra và bảo trì chương trình.

Ví dụ: Trường hợp không dùng hàm tính trung bình cộng 2 số

```
#include <iostream>
using namespace std;
int main()
{
    // Trung bình cộng của 3 và 7
    cout << (3 + 7)/2.0;
    // Trung bình cộng của 4 và 6
    cout << (4 + 6)/2.0;
    return 0;
}
```

Trường hợp dùng hàm tính trung bình cộng 2 số

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

```
// Định nghĩa hàm tính trung bình cộng 2 số
double avg(int a, int b)
{
    return (a + b)/2.0;
}

int main()
{
    // Trung bình cộng của 3 và 7
    cout << avg(3, 7);

    // Trung bình cộng của 4 và 6
    cout << avg(4, 6);

    return 0;
}
```

Ngoài ra, chúng ta có thể kể chi tiết những lợi ích khi sử dụng hàm như sau:

- Tránh việc lặp đi lặp lại một khối lệnh cùng chức năng.
- Hỗ trợ việc thực hiện các chương trình lớn.
- Phục vụ các quá trình trừu tượng hóa.
- Mở rộng khả năng ngôn ngữ.
- Thuận tiện phát triển, nâng cấp chương trình.

## 2. Khai báo hàm và định nghĩa hàm

### a. Khai báo hàm

Thông thường hàm thực hiện việc tính toán trên các tham số và cho lại giá trị kết quả, hoặc chỉ thực hiện một chức năng nào đó mà không cần trả lại kết quả. Kiểu giá trị trả về này được gọi là kiểu của hàm. Hàm thường được khai báo ở đầu chương trình. Cách để khai báo một hàm như sau:

**<kiểu của hàm> <tên hàm>(danh sách tham số) ;**

Trong đó:

- Kiểu của hàm: Là kiểu dữ liệu mà hàm trả về. Nếu không có giá trị trả về thì kiểu của hàm là void.
- Tên hàm: Là tên của hàm, do người lập trình đặt. Tên hàm không được chứa ký tự đặc biệt, không được bắt đầu bằng số.
- Danh sách tham số: là danh sách các tham số dùng như các biến cục bộ. Nếu có nhiều tham số, thì chúng sẽ được phân tách theo các dấu phẩy.
- Ví dụ:

```
- int test(int);           // Hàm test có tham số kiểu int và kiểu
    hàm là int int rand(); // Không tham số, kiểu hàm
```

```
(giá trị trả lại) là int void write(float) ;    //Hàm không trả lại giá trị (kiểu void).
```

### ***b. Định nghĩa hàm***

Cấu trúc một hàm bất kỳ được bố trí cũng giống như hàm main(). Cụ thể:

```
<kiểu hàm> <tên hàm>(danh sách tham số)
{
    dãy lệnh của hàm ;
    return (biểu thức trả về); // có thể nằm đâu đó trong dãy lệnh.
}
```

Trong đó return là để truyền dữ liệu cho nơi gọi hàm. Câu lệnh return bắt buộc là phải có đối với hàm có giá trị trả về. Và có thể có hoặc không đối với hàm không có giá trị trả về (có kiểu trả về là void). Đối với hàm có giá trị trả về thì kèm theo sau return là một biểu thức. Kiểu giá trị của biểu thức này chính là kiểu của hàm. Tùy theo mục đích của hàm mà chúng ta có thể đặt return bất cứ đâu trong dấu {} ở trên. Khi gặp return chương trình tức khắc thoát khỏi hàm và trả lại giá trị biểu thức sau return. Có thể có nhiều câu lệnh return trong hàm.

Ví dụ: Tính tổng 2 số nguyên (kiểu int). Hàm này có 2 tham số truyền vào (số nguyên a và số nguyên b) và giá trị trả về là số nguyên a+b.

```
int sum(int a, int b)
{
    return (a+b);
}
```

Một ví dụ về hàm không có giá trị trả về:

```
void printNumber(int a)
{
    std::cout << a;
}
```

Hàm trên thực hiện công việc in một số nguyên (được truyền vào) ra màn hình console. Không có kiểu trả về nên không cần return.

Danh sách tham số trong khai báo hàm có thể có hoặc không tên của biến, thông thường ta chỉ cần khai báo kiểu dữ liệu của tham số chứ không cần khai báo tên của biến. Trong khi trong định nghĩa hàm thì phải có đầy đủ. Hàm có thể có hoặc không có tham số, tuy nhiên cặp dấu () sau tên hàm vẫn phải được viết.

Ví dụ:

```
//Khai báo hàm
int sum(int, int); //không cần tên biến
int sum(int a, int b); //có tên biến

//Định nghĩa hàm
int sum(int a, int b) // có tên biến
{
    return (a + b);
}

void print()
{
    cout << "Hello";
}
```

Tuy nhiên, ta có thể vừa khai báo vừa định nghĩa hàm cùng lúc, cho nên để cho ngắn gọn người ta thường viết đoạn chương trình trên như sau:

```
//Định nghĩa hàm
int sum(int a, int b) // có tên biến
{
    return (a + b);
}

void print()
{
    cout << "Hello";
}
```

### 3. Tham số của hàm

Thông thường ta có 2 cách để truyền tham số cho hàm đó là truyền tham trị và tham chiếu. Ta sẽ lần lượt xem xét 2 cách truyền tham số này.

#### *a. Truyền tham trị*

Hiểu một cách đơn giản thì truyền theo tham trị là truyền giá trị của tham số. Khi hàm được gọi. Chương trình sẽ khởi tạo các ô nhớ tương ứng với danh sách tham số truyền vào. Sau đó sẽ sao chép các giá trị của tham số vào các ô nhớ mới này. Vì vậy những thay đổi trong ô nhớ mới này sẽ không ảnh hưởng đến các tham số truyền vào.

Cú pháp truyền tham trị: **<kiểu tham trị> <tên biến>;**

Để hiểu hơn về truyền theo kiểu tham trị cũng như cách thực hiện, ta xét các ví dụ sau:

Chương trình tính tổng 2 số:

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

int sum(int a, int b)
{
    return (a+b);
}

int main()
{
    int a = 2, b = 3;
    int s;
    s = sum(a, b); // Gán giá trị là tổng a và b cho biến s;
    cout << "Tong của a va b: " << s << endl;
    return 0;
}
```

Xét một ví dụ khác là chương trình hoán đổi 2 số:

```
#include <iostream>
using namespace std;
void swap(int a, int b)
{
    int tamp = a;
    a = b;
    b = tamp;
}

int main()
{
    int a = 2, b = 3;
    cout << "Truoc khi hoan doi: " << a << ' ' << b << endl;;
    swap(a, b); // hoán đổi
    cout << "Sau khi hoan doi: " << a << ' ' << b << endl;
    return 0;
}
```

Sau khi chạy chương trình trên sẽ thấy kết quả trước và sau vẫn không đổi. Lý do là bởi vì lúc truyền tham số, chương trình sẽ khởi tạo 2 ô nhớ khác và sao chép giá trị truyền vào. Nên mọi thay đổi trong hàm sẽ không ảnh hưởng đến chương trình chính. Do đó không thể dùng cách truyền tham trị cho chương trình trên.

### ***b. Truyền tham chiếu***

Tham chiếu nói một cách đơn giản là một bí danh của một biến. Nói cách khác, tham chiếu là một tên gọi khác của một biến đã có sẵn. Điều đó có nghĩa là, có thể thao tác trên biến đó thông qua tên biến hoặc tham chiếu.

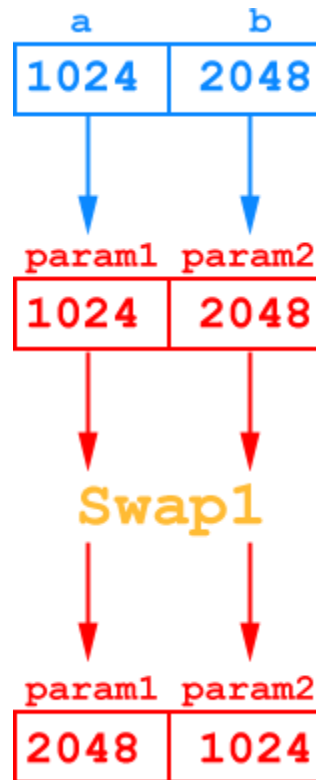
Cú pháp truyền tham chiếu: **<kiểu tham chiếu> &<tên biến>;**

Để hiểu rõ lí do nên dùng tham chiếu ta sẽ xét lại ví dụ hoán đổi hai số.

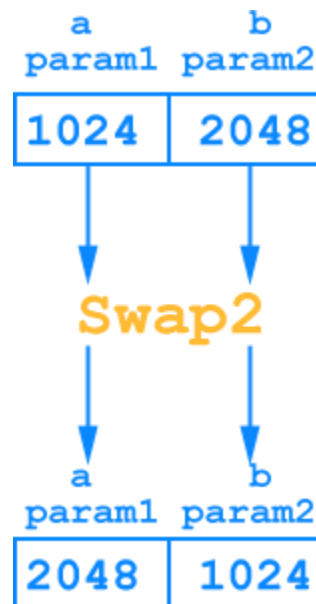
```
#include <iostream>
using namespace std;
void Swap1(int param1, int param2) {
    int temp = param1;
    param1 = param2;
    param2 = temp;
}
void Swap2(int& param1, int& param2) {
    int temp = param1;
    param1 = param2;
    param2 = temp;
}
int main() {
    int a = 1024;
    int b = 2048;
    cout << "a = " << a << ", b = " << b << endl;
    cout << "Swap1: ";
    Swap1(a, b);
    cout << "a = " << a << ", b = " << b << endl;
    cout << "Swap2: ";
    Swap2(a, b);
    cout << "a = " << a << ", b = " << b << endl;
    return 0;
}
// Difference between value parameter and reference parameter
// a = 1024, b = 2048
// Swap1: a = 1024, b = 2048
// Swap2: a = 2048, b = 1024
```

Hàm swap1 sử dụng cách truyền tham trị, trong khi đó hàm swap2 sử dụng cách truyền tham chiếu. Kết quả hoàn toàn khác nhau là vì: Khi truyền tham trị, chương trình sẽ tạo ra hai biến mới và copy giá trị của hai tham trị vào, cho nên khi đó hàm chỉ hoán đổi hai biến là bản sao của hai tham trị với nhau mà thôi, tham trị hoàn toàn không tham gia vào quá trình này; Trong khi đó, khi truyền tham chiếu, chương trình sẽ không tạo ra bản sao nào cả mà chỉ đơn giản là dùng chính hai tham chiếu cho những lệnh trong hàm, cho nên giá trị của các tham chiếu đã bị thay đổi.





Khi truyền tham trị giá trị của `a`, `b` sẽ được sao chép vào 2 biến `param1`, `param2` hay nói cách khác là đồ “fake”



Còn với tham chiếu thì chương trình sử dụng chính bản gốc “official”

### ***c. Hàm với tham số có giá trị ngầm định***

Xét một ví dụ sau:

```
#include <iostream>

int sum(int a, int b, int c, int d)
{
    return (a + b + c + d);
}

int main()
{
    int s = sum(1, 3, 4, 5);
    std::cout << "Tong la: " << s;
    return 0;
}
```

Hàm sum trong chương trình trên thực hiện việc tính tổng 4 số nguyên. Tuy nhiên, chúng ta chỉ cần tính tổng của 2 hay 3 số nguyên mà không phải là 4 số nguyên. Vấn đề được giải quyết khi chúng ta viết thêm một hàm tính 2 số nguyên và 3 số nguyên khác. Hoặc cũng có thể gọi hàm với giá trị có tham số bằng 0 như sau:

```
int s = sum(1, 2, 0, 0);
```

Lời gọi hàm như vậy sẽ giúp chúng ta tính được tổng 2 số nguyên một và 2 từ hàm tính tổng 4 số nguyên ở trên. Một vấn đề khác được đặt ra khi trong chương trình có quá nhiều lần hàm được gọi để tính tổng 2 số nguyên. Như vậy, cứ mỗi lần gọi hàm, lúc nào cũng phải viết thêm 2 tham số ngầm định giá trị 0 ở sau.

Đó là trường hợp với ví dụ trên. Trong thực tế, hãy thử tưởng tượng nếu một hàm được gọi nhiều lần với nhiều tham số có giá trị ngầm định (không thay đổi giá trị qua các lần gọi). Mỗi lần như vậy lúc nào cũng phải viết thêm một danh sách dài các tham số có giá trị giống nhau. Đó thật sự là một công việc không mấy thú vị. Từ đó hàm có tham số với giá trị ngầm định được C++ mở rộng ra (so với C) để giải quyết vấn đề này.

Cú pháp:

```
<Kiểu hàm> <Tên hàm> (t/s1, t/s2, ...,t/sn, t/smd1 = gt1, t/smd2 = gt2,...,
t/smdn = gtn);
```

Trong đó:

- t/s1, t/s2,...,t/sn là các tham số được khai báo bình thường. Tức là gồm có kiểu dữ liệu và tên tham số.
- Các tham số ngầm định t/smd1 ,t/smd2, ...,t/smdn có gán thêm các giá trị ngầm định gt1, gt2,..., gtn.
- Một lời gọi bất kỳ đến hàm này đều phải có đầy đủ các tham số ứng với t/s1,..., t/sn (không phải tham số ngầm định). Nhưng có thể có hoặc không

các tham số ngầm định ứng với  $t/smd1, \dots, t/smdn$ . Nếu tham số nào không có thì nó sẽ tự động gán với giá trị ngầm định đã khai báo.

Ví dụ: Hàm tính giá trị lũy thừa.

```
int exponential(int x, int n = 2);
```

Hàm này có một tham số có giá trị ngầm định là số mũ  $n$ . Nếu lời gọi hàm bỏ qua số mũ này thì chương trình sẽ hiểu là  $n = 2$  và tính bình phương của  $x$ .

Khi gọi `int e = exponential(4, 3);` chương trình sẽ tính  $e = 4^3$ .

Khi gọi `int e = exponential(4);` chương trình sẽ tính  $e = 4^2$ .

Xét hàm tính tổng 4 số nguyên.

```
int sum(int a, int b, int c = 0, int d = 0);
```

Chúng ta có thể:

- Tính tổng của 4 số nguyên 1, 2, 3, 4 bằng cách gọi `sum(1, 2, 3, 4)`.
- Tính tổng của 3 số nguyên 1, 2, 3 bằng cách gọi `sum(1, 2, 3)`.
- Tính tổng của 2 số nguyên 1, 2 bằng cách gọi `sum(1, 2)`.

**Chú ý:**

- Các tham số với giá trị ngầm định phải được khai báo liên tục và xuất hiện cuối cùng trong danh sách tham số.

```
int sum(int a, int b = 0, int c, int d = 0);
```

Trường hợp này sai vì các tham số ngầm định không liên tục.

```
int sum(int a, int b = 0, int c = 0, int d);
```

Trường hợp này sai vì tham số có giá trị ngầm định không ở cuối.

#### 4. Nạp chồng hàm (Function overloading)

Xét một ví dụ:

```
#include <iostream>

int max(int a, int b)
{
    if(a > b) return a;
    else return b;
}

int main()
{
    int c = max(4, 5);
}
```

```
std::cout << "c = " << c;
return 0;
}
```

Khi chạy chương trình chúng ta nhận được kết quả  $c = 5$  đúng như mong muốn.

Vậy nếu thay dòng lệnh  $c = \max(4, 5)$  thành  $\max(4.3, 5.5)$  thì ta lại được kết quả là 5, không phải kết quả ta mong muốn (5.5). Trường hợp này ta phải viết chương trình tính max của hai số thực:

```
float fmax(float a, float b)
{
    if(a > b) return a;
    else return b;
}
```

Tương tự ta sẽ có:

```
double dmax(double a, double b)
{
    if(a > b) return a;
    else return b;
}
```

hay:

```
char cmax(char a, char b)
{
    if(a > b) return a;
    else return b;
}
```

Vậy chúng ta sẽ gặp phải 2 bất lợi lớn đó là:

- Có quá nhiều tên hàm khác nhau.
- Viết quá nhiều lần một hàm giống nhau.

Ta có cách để giải quyết cả hai vấn đề này. Tuy nhiên, nó sẽ ở phần sau. Bây giờ ta chỉ xét cách để giải quyết vấn đề thứ nhất. Đó là nạp chồng hàm (function overloading). Nạp chồng hàm cho phép ta khai báo và định nghĩa các hàm trên cùng với một tên gọi. Khi đó ta sẽ được:

```
int max(int a, int b)
{
    if(a > b) return a;
    else return b;
}

float max(float a, float b)
{
    if(a > b) return a;
    else return b;
}
```

```
}  
  
double max(double a, double b)  
{  
    if(a > b) return a;  
    else return b;  
}  
  
char max(char a, char b)  
{  
    if(a > b) return a;  
    else return b;  
}
```

Và khi gọi hàm ở bất kì dạng nào như:

`max(4, 5)`

`max(4.3, 5.5)`

`max('A', 'B')`

Ta đều có được kết quả mong muốn.

Lưu ý: Ta không thể nạp chồng hàm nếu như chỉ khác biệt về kiểu trả về.

## CHƯƠNG VI: TẬP

---

# §1: KIỂU DỮ LIỆU TẬP

### 1. Vai trò của tập

Tất cả các dữ liệu ở thuộc các kiểu dữ liệu đã xét đều lưu ở bộ nhớ trong (RAM) và do đó sẽ bị mất khi tắt máy. Với một số bài toán khối lượng dữ liệu lớn, có yêu cầu lưu trữ xử lý nhiều lần, cần có kiểu dữ liệu tập (file).

Kiểu dữ liệu tập có những đặc điểm sau:

- Dữ liệu tập được lưu trữ lâu dài ở bộ nhớ ngoài, không bị mất đi khi tắt nguồn điện.
- Lượng dữ liệu của tập có thể rất lớn, phụ thuộc vào dung lượng đĩa.

### 2. Phân loại tập

Xếp theo cách tổ chức dữ liệu, tập có thể phân thành hai loại:

- Tập văn bản: là tập mà dữ liệu được ghi dưới dạng các kí tự theo mã ASCII. Trong tập văn bản, dãy kí tự kết thúc bởi kí tự xuống dòng hay kí tự kết thúc tập tạo thành một dòng.
- Tập có cấu trúc là tập mà thành phần của nó được tổ chức theo một cấu trúc nhất định. Tập nhị phân là một trường hợp riêng của tập có cấu trúc. Dữ liệu ảnh, âm thanh, .. thường được lưu trữ dưới dạng tập có cấu trúc.

Xếp theo cách truy cập tập, ta có thể phân thành hai loại sau:

- Tập truy cập tuần tự: cho phép truy cập đến một dữ liệu nào đó trong tập chỉ bằng cách bắt đầu từ đầu tập và đi qua lần lượt các dữ liệu trước nó.
- Tập truy cập trực tiếp là tập cho phép tham chiếu đến dữ liệu cần truy cập bằng cách xác định trực tiếp vị trí (thường là số hiệu) của dữ liệu đó.

## §2: THAO TÁC VỚI TẬP TRONG C++

### 1. Đọc dữ liệu từ fstream

Tới bây giờ, chúng ta đã sử dụng thư viện chuẩn iostream, cung cấp các phương thức cin và cout để đọc từ Standard Input và ghi tới Standard Output tương ứng. Bây giờ ta cần phải đọc và ghi dữ liệu với file. Điều này cần một Thư viện chuẩn C++ khác là fstream, mà định nghĩa 3 kiểu dữ liệu mới:

Kiểu dữ liệu	Miêu tả
ofstream	Kiểu dữ liệu này biểu diễn Output File Stream và được sử dụng để tạo các file và để ghi thông tin tới các file đó
ifstream	Kiểu dữ liệu này biểu diễn Input File Stream và được sử dụng để đọc thông tin từ các file
fstream	Kiểu dữ liệu này nói chung biểu diễn File Stream, và có các khả năng của cả ofstream và ifstream, nghĩa là nó có thể tạo file, ghi thông tin tới file và đọc thông tin từ file

Để thực hiện tiến trình xử lý file trong C++, bạn bao các header file là `<iostream>` và `<fstream>` trong source file của chương trình C++ của chúng ta.

#### ***a. Mở một file trong C++ với fstream***

Một file phải được mở trước khi bạn có thể đọc thông tin từ nó hoặc ghi thông tin tới nó. Hoặc đối tượng ofstream hoặc đối tượng fstream có thể được sử dụng để mở một file với mục đích viết hoặc đối tượng ifstream được sử dụng để mở file chỉ với mục đích đọc.

Dưới đây là prototype của hàm open(), là một thành viên của các đối tượng fstream, ifstream và ofstream trong C++:

```
void open(const char *ten_file, ios::mode);
```

Tại đây, tham số đầu tiên xác định tên và vị trí của file để được mở và tham số thứ hai của hàm thành viên open() định nghĩa chế độ mà file nên được mở.

Chế độ	Miêu tả
<code>ios::app</code>	Chế độ Append. Tất cả output tới file đó được phụ thêm vào cuối file đó
<code>ios::ate</code>	Mở một file cho output và di chuyển điều khiển read/write tới cuối của file
<code>ios::in</code>	Mở một file để đọc
<code>ios::out</code>	Mở một file để ghi
<code>ios::trunc</code>	Nếu file này đã tồn tại, nội dung của nó sẽ được cắt (truncate) trước khi mở file

Bạn có thể kết hợp hai hoặc nhiều giá trị này bằng việc hoặc chúng cùng với nhau (sử dụng `|`). Ví dụ, nếu bạn muốn mở một file trong chế độ ghi và muốn cắt (truncate) nó trong trường hợp nó đã tồn tại, bạn theo cú pháp sau:

```
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

Tuy nhiên, đó là cách tổng quát. Ta có cách ngắn gọn hơn để làm điều này:

```
ofstream fo("file.dat");
ifstream fi("file.dat");
```

Tương đương với:

```
ofstream fo;
fo.open("file.dat", ios::out);
ifstream fi;
fi.open("file.dat", ios::in);
```

### ***b. Đóng một file trong C++ với `fstream`***



Để đóng một file ta thực hiện cú pháp:

**<tên biến file>.close();**

Ví dụ: **outfile.close();**

Ví dụ làm việc với file

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

int main ()
{
    char data[100];

    // mo mot file trong che do write.
    ofstream outfile;
    outfile.open("hdi.dat");

    cout << "Ghi du lieu toi file!" << endl;
    cout << "Nhap ten cua ban: ";
    cin.getline(data, 100);

    // ghi du lieu da nhap vao trong file.
    outfile << data << endl;

    cout << "Nhap tuoi cua ban: ";
    cin >> data;
    cin.ignore();

    // ghi du lieu da nhap vao trong file.
    outfile << data << endl;

    // dong file da mo.
    outfile.close();

    // mo mot file trong che do read.
    ifstream infile;
    infile.open("hdi.dat");

    cout << "\n=====\\n" ;
    cout << "Doc du lieu co trong file!" << endl;
    infile >> data;

    // ghi du lieu tren man hinh.
    cout << data << endl;

    // tiep tuc doc va hien thi du lieu.
    infile >> data;
```

```

    cout << data << endl;

    // dòng file đã mở.
    infile.close();

    return 0;
}

```

## 2. Hàm freopen()

Thông thường trong các cuộc thi thông thường, vấn đề làm việc với file thường chỉ là đọc và ghi file text. Cho nên có một cách thao tác với file khá đơn giản là sử dụng hàm mở file `freopen()` trong thư viện `stdio.h` của C. Prototype của hàm này như sau:

```
FILE * freopen ( const char * filename, const char * mode, FILE * stream );
```

Trong đó:

- `filename`: Là chuỗi chứa tên file được mở (có thể là đường dẫn đến file).
- `mode`: Chế độ truy cập file. Mode có thể là một trong những giá trị dưới đây:

mode	Miêu tả
"r"	Mở một file để đọc. File phải tồn tại
"w"	Tạo một file trống để ghi. Nếu một file với cùng tên đã tồn tại, nội dung của nó bị tẩy đi và file được xem như một file trống mới
"a"	Phụ thêm (append) tới một file. Với các hoạt động ghi, phụ thêm dữ liệu tại cuối file. File được tạo nếu nó chưa tồn tại
"r+"	Mở một file để ghi và đọc. File phải tồn tại
"w+"	Tạo một file trống để ghi và đọc
"a+"	Mở một file để đọc và phụ thêm

- `stream`: Là con trỏ tới đối tượng file mà nhận diện stream đang được mở.

Nếu file được mở lại thành công, hàm trả về một con trỏ tới một đối tượng nhận diện Stream đó, nếu không thì con trỏ null được trả về. Tức là sau khi mở file bằng `freopen()` ta vẫn sẽ sử dụng luồng dữ liệu giống như standard i/o stream (`scanf`, `printf`, `cin`, `cout`, ...) hoạt động bình thường như khi làm việc với màn hình. Ví dụ:

```
#include <bits/stdc++.h>

using namespace std;

int main() {
    freopen("IN.TXT", "r", stdin);
    freopen("OUT.TXT", "w", stdout);

    int a, b, ans = 0;
    cin >> a >> b;
    for (int i = a; i <= b; i++)
        ans += b;
    cout << ans;
    return 0;
}
```

## CHƯƠNG VII: CON TRỎ

---

### §1: CON TRỎ

Con trỏ - Pointer trong ngôn ngữ C /C++ rất dễ học và thú vị. Một vài tác vụ trong ngôn ngữ C /C++ được thực hiện dễ dàng hơn nhờ con trỏ, và những tác vụ khác trở nên linh hoạt hơn, như trong việc cấp phát bộ nhớ, không thể thực hiện mà không dùng con trỏ

#### 1. Khái niệm con trỏ

Như bạn biết, mỗi biến trong một vùng nhớ nhất định và mỗi vùng nhớ này có địa chỉ có nó được định nghĩa để dễ dàng trong việc truy cập sử dụng toán tử (&) tương ứng với địa chỉ của nó trong bộ nhớ. Xem xét ví dụ dưới đây, sẽ in ra địa chỉ của biến được định nghĩa:

```
#include <iostream>

using namespace std;

int main ()
{
    int  bien1;
    char bien2[10];

    cout << "Địa chỉ của bien1 là: ";
    cout << &bien1 << endl;

    cout << "Địa chỉ của bien2 là: ";
    cout << &bien2 << endl;

    return 0;
}
```

Chạy chương trình trên sẽ được kết quả như sau:

```

C:\Users\Administrator\Desktop\workspace\cppbook\sourcecode\myproject\bin\Debug\myproject.exe
Dia chi cua bien1 la: 0x6afefc
Dia chi cua bien2 la: 0x6afef2

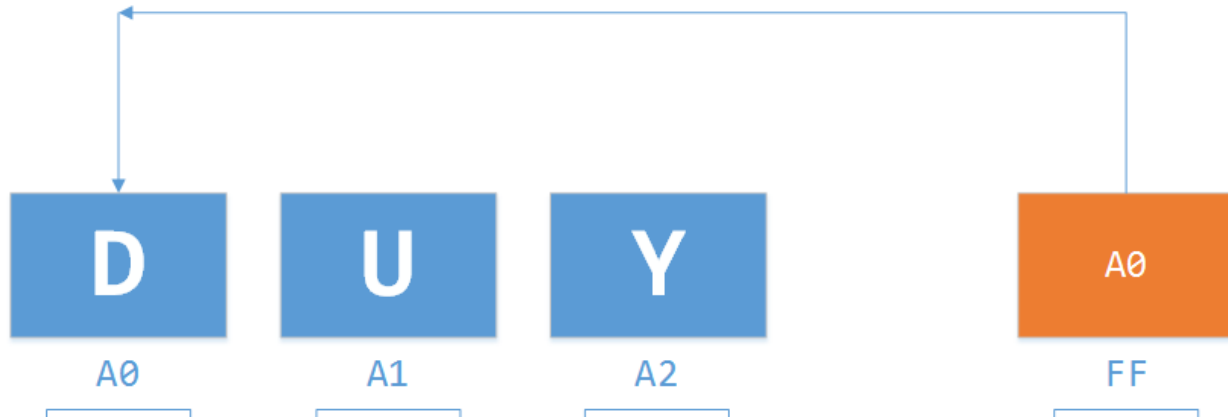
Process returned 0 (0x0)   execution time : 0.019 s
Press any key to continue.

```

Vậy con trỏ là gì ?

Một con trỏ là một biến mà trong đó giá trị của nó là địa chỉ của biến khác. Ví dụ như địa chỉ của vùng nhớ. Giống như các biến và hằng số, ta phải khai báo con trỏ trước khi có thể sử dụng nó để lưu trữ bất kỳ địa chỉ của biến nào.

Nếu chưa hình dung được ta sẽ xem xét ví dụ sau:



Trong hình minh họa trên, biến con trỏ (có địa chỉ là FF) sẽ quản lý vùng nhớ chuỗi “DUY” (bắt đầu từ địa chỉ A0). Dĩ nhiên đây chỉ là hình minh họa cho định nghĩa, bản chất nó còn nhiều điểm cần làm rõ bên dưới.

Dạng tổng quát của việc khai báo con trỏ như sau:

**<kiểu dữ liệu> \*<tên biến>;**

Ở đây, **<kiểu dữ liệu>** là kiểu dữ liệu cơ bản con trỏ, nó là kiểu hợp lệ trong ngôn ngữ C++ và **<tên biến>** là tên giá trị của con trỏ. Phần ký tự \* sử dụng trong khai báo con trỏ giống như việc bạn sử dụng cho phép nhân. Mặc dù vậy, trong khai báo này, ký tự \* được thiết kế để sử dụng các biến của con trỏ. Dưới đây là một số cách khai báo hợp lệ của con trỏ:

```

int    *sv;    // tro toi mot gia tri nguyen
double *nv;    // tro toi mot gia tri double

```

```
float *luong;    // tro toi mot gia tri float
char *ten       // tro toi mot ky tu
```

Kiểu dữ liệu thực sự của giá trị của tất cả các con trỏ, có thể là integer, float, character, hoặc kiểu khác, là giống như, một số long hexa biểu diễn một địa chỉ bộ nhớ. Điểm khác nhau duy nhất của các con trỏ của các kiểu dữ liệu khác nhau là kiểu dữ liệu của biến hoặc hằng số mà con trỏ chỉ tới.

## 2. Sử dụng con trỏ trong C++

Bởi vì con trỏ chỉ lưu giữ địa chỉ, nên khi bạn gán giá trị cho một con trỏ, giá trị đó phải là địa chỉ. Để lấy địa chỉ của một biến, chúng ta sử dụng address-of operator (&).

```
int main()
{
    int a = 5;
    int *ip = &a;
    return 0;
}
```

Khi đó biến ip sẽ lưu trữ địa chỉ của byte đầu tiên của biến a. Lưu ý toán tử & ở đây là unary operator – toán tử một ngôi, hoàn toàn khác với toán tử & hai ngôi (bitwise).

Một toán tử khác thường sử dụng với con trỏ là dereference operator (\*). Dereference operator dùng để lấy nội dung của địa chỉ mà biến đó trỏ vào.

```
#include <iostream>

using namespace std;

int main()
{
    int a = 5;

    int *ip = &a;
    cout << *ip << endl;

    void *vp = &a;
    cout << *(int *)vp << endl;
```

```

    return 0;
}

```

Trong chương trình trên, để lấy giá trị được lưu trữ trong biến `a` có hai cách: gọi trực tiếp biến `a` hoặc gọi gián tiếp qua con trỏ `ip`. Khi đó, ta có thể thay đổi giá trị của biến `a` thông qua con trỏ `ip` như sau: `*ip = 11;`

Một lưu ý là con trỏ kiểu `void` có thể lưu địa chỉ của biến bất kì, nhưng không thể dereference trực tiếp. Nghĩa là nó không thể lấy nội dung của memory address chứa trong nó, vì con trỏ `void` không biết kiểu dữ liệu của đối tượng mà nó trỏ đến. Do đó, lập trình viên cần định kiểu hoặc gán nó cho một con trỏ khác có kiểu dữ liệu cụ thể trước khi sử dụng.

Ngoài ra ta có thể gán trực tiếp một con trỏ cho một con trỏ khác. Tuy nhiên, khi đó hai con trỏ sẽ cùng lưu trữ memory address giống nhau, dễ xảy ra tình trạng "dangling pointer" và gây ra lỗi memory leak nếu người lập trình không quản lý chương trình tốt.

### 3. Kích thước của con trỏ

Như ta biết, những biến có kiểu dữ liệu khác nhau sẽ được cấp phát vùng nhớ có kích thước khác nhau. Còn với biến con trỏ thì như thế nào? Để trả lời cho câu hỏi trên, trước tiên chúng ta cần trả lời câu hỏi: Biến con trỏ dùng để làm gì? Như đề cập ở trên, biến con trỏ dùng để lưu trữ memory address của một biến khác. Dù cho biến con trỏ được khai báo như thế nào đi nữa thì giá trị được lưu trữ trong nó cũng là một memory address. Đến đây người không có kiến thức lập trình cũng thấy được rằng đối với biến con trỏ, kích thước của nó là một hằng số, bất kể kiểu dữ liệu mà nó được khai báo. Hằng số này phụ thuộc vào OS, IDE, Compiler, ... Trên hệ điều hành 32-bit, hằng số này là 4 byte.

Vậy nếu kích thước của con trỏ là một hằng số, tại sao lại có nhiều kiểu biến con trỏ như vậy? Chúng ta cùng xét ví dụ sau:

```

#include <iostream>

using namespace std;

int main ()
{
    int a = 257;
    char *cp = (char *)&a;

    cout << *cp;
}

```

```
    return 0;
}
```

Kết quả in ra màn hình là ký tự ☺. Để lý giải cho trường hợp này, minh họa bằng cách cụ thể hóa vùng nhớ của biến a như sau: 0001 0001 0000 0000

Do kiểu char có kích thước 1 byte, khi ép kiểu (char \*), biến cp chỉ nhận được dữ liệu trong byte đầu tiên (byte trái nhất) của biến a, dẫn đến việc đọc thiếu một phần dữ liệu có trong biến a. Kiểu dữ liệu sẽ quy định cho biến con trỏ đọc số byte tương ứng kể từ ô nhớ đầu tiên đó.

#### 4. Các phép toán trên con trỏ

C++ cho phép thực hiện các phép toán cộng, trừ trên con trỏ thông qua các toán tử +, -, ++, --. Khi đó, giá trị địa chỉ lưu trữ trong con trỏ sẽ được cộng/trừ số byte tương ứng với kiểu dữ liệu của con trỏ. Thao tác tương tự như với dữ liệu kiểu số:

```
int main()
{
    int a = 5;
    int *ip = &a;

    cout << ip + 1;
    cout << ip++;    //cout << ip; ip = ip + 1;
    cout << ++ip;    //ip = ip + 1; cout << ip;

    return 0;
}
```

Ở dòng thứ 6, giá trị in ra màn hình là địa chỉ của biến a + 1, con trỏ ip vẫn mang giá trị là địa chỉ của biến a. Để dễ hiểu hơn, giả sử biến a có địa chỉ là D0, giá trị in ra màn hình ở dòng 6 là D4 (không phải là D1!). Dòng 7 và 8 thao tác tương tự như với dữ liệu kiểu số, sau khi thực hiện, giá trị chứa trong con trỏ ip đã bị thay đổi. Ngoài các phép toán giữa con trỏ với số, ta còn có thể thực hiện phép trừ hai con trỏ cùng kiểu. Khi đó, giá trị trả về là một số nguyên độ chênh lệch địa chỉ giữa hai con trỏ đó với đơn vị là kiểu dữ liệu của con trỏ.

#### 5. Một vài vấn đề khác về con trỏ

##### a. Con trỏ NULL

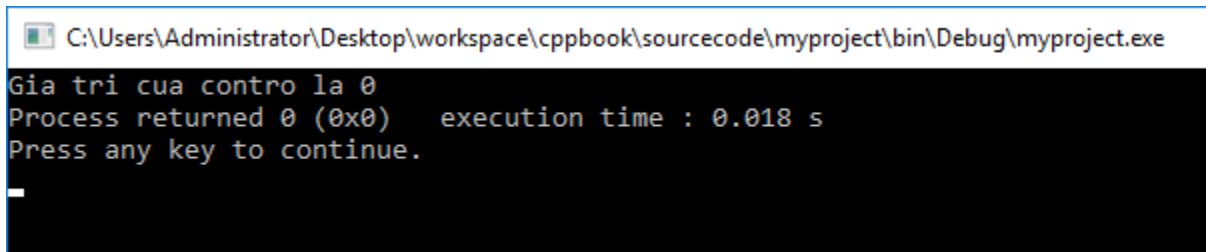


Con trỏ NULL là một hằng với một giá trị là 0 được định nghĩa trong một vài thư viện chuẩn, gồm iostream. Xét ví dụ sau:

```
#include <iostream>
using namespace std;
int main ()
{
    int *ptr = NULL;
    cout << "Gia tri cua contro la " << ptr ;

    return 0;
}
```

Kết quả nếu chạy chương trình là:



```
C:\Users\Administrator\Desktop\workspace\cppbook\sourcecode\myproject\bin\Debug\myproject.exe
Gia tri cua contro la 0
Process returned 0 (0x0)   execution time : 0.018 s
Press any key to continue.
```

Trên hầu hết Hệ điều hành, các chương trình không được cho phép để truy cập bộ nhớ tại địa chỉ 0, bởi vì, bộ nhớ đó được dự trữ bởi Hệ điều hành. Tuy nhiên, địa chỉ bộ nhớ 0 có ý nghĩa đặc biệt; nó báo hiệu rằng con trỏ không được trỏ tới một vị trí ô nhớ có thể truy cập. Nhưng theo qui ước, nếu một con trỏ chứa giá trị 0, nó được xem như là không trỏ tới bất cứ thứ gì.

### ***b. Con trỏ và mảng***

Con trỏ và Mảng có mối liên hệ chặt chẽ. Thực tế, con trỏ và mảng là có thể thay thế cho nhau trong một số trường hợp. Ví dụ, một con trỏ mà trỏ tới phần đầu mảng có thể truy cập mảng đó bởi sử dụng: hoặc con trỏ số học hoặc chỉ mục mảng. Xét ví dụ sau:

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main ()
{
    int mang[MAX] = {10, 100, 200};
    int *contro;

    // chung ta co mot mang dia chi trong con tro.
    contro = mang;
```

```

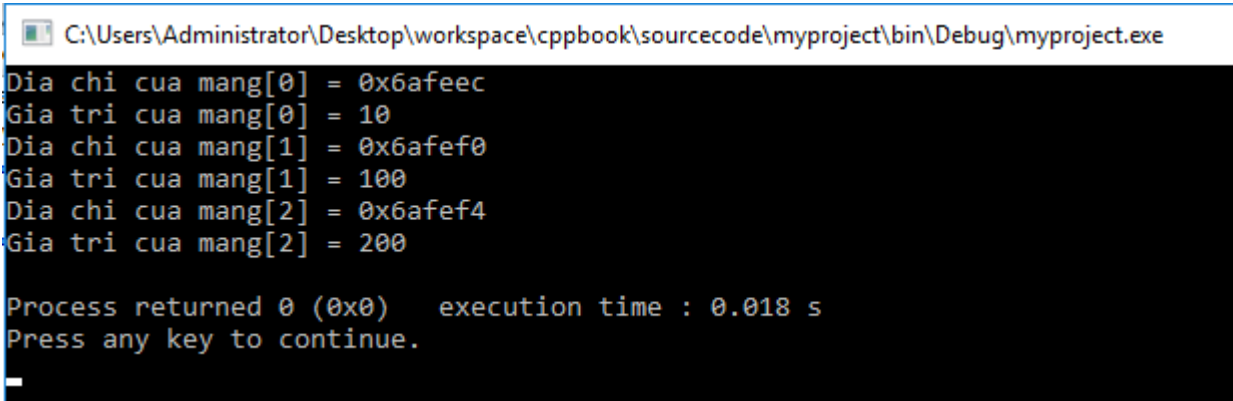
for (int i = 0; i < MAX; i++)
{
    cout << "Dia chi cua mang[" << i << "] = ";
    cout << contro << endl;

    cout << "Gia tri cua mang[" << i << "] = ";
    cout << *contro << endl;

    // tro toi vi tri tiep theo
    contro++;
}
return 0;
}

```

Ouput:



```

C:\Users\Administrator\Desktop\workspace\cppbook\sourcecode\myproject\bin\Debug\myproject.exe
Dia chi cua mang[0] = 0x6afeec
Gia tri cua mang[0] = 10
Dia chi cua mang[1] = 0x6afef0
Gia tri cua mang[1] = 100
Dia chi cua mang[2] = 0x6afef4
Gia tri cua mang[2] = 200

Process returned 0 (0x0)   execution time : 0.018 s
Press any key to continue.

```

Tuy nhiên, con trỏ và mảng không hoàn toàn thay thế được cho nhau. Ví dụ:

```

#include <iostream>

using namespace std;
const int MAX = 3;

int main ()
{
    int mang[MAX] = {10, 100, 200};

    for (int i = 0; i < MAX; i++)
    {
        *mang = i;    // Day la cu phap chinh xac
        mang++;       // cu phap nay la sai, nen chu y.
    }
    return 0;
}

```

Việc áp dụng toán tử con trỏ \* tới biến mang là hoàn hảo, nhưng nó không hợp lệ khi sửa đổi giá trị biến mang. Lý do là biến mang là một constant mà trỏ tới phần đầu mảng và không thể được sử dụng như là l-value. Bởi vì, một tên mảng tạo

một hằng con trỏ, nó có thể vẫn được sử dụng trong các biểu thức con trỏ, miễn là nó không bị sửa đổi. Ví dụ sau là một lệnh hợp lệ mà gán mang[2] giá trị 500.

```
*(mang + 2) = 500;
```

Lệnh trên là hợp lệ và sẽ biên dịch thành công bởi vì mang không bị thay đổi.

### c. Mảng con trỏ

Trước khi chúng ta hiểu về khái niệm mảng các con trỏ, chúng ta xem xét ví dụ sau, mà sử dụng một mảng gồm 3 số integer:

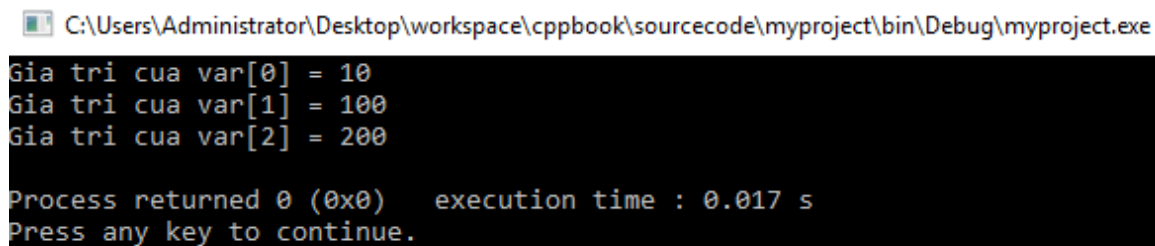
```
#include <iostream>

using namespace std;
const int MAX = 3;

int main ()
{
    int var[MAX] = {10, 100, 200};

    for (int i = 0; i < MAX; i++)
    {
        cout << "Gia tri cua var[" << i << "] = ";
        cout << var[i] << endl;
    }
    return 0;
}
```

Khi code trên được biên dịch và thực thi, nó cho kết quả sau:



```
C:\Users\Administrator\Desktop\workspace\cppbook\sourcecode\myproject\bin\Debug\myproject.exe
Gia tri cua var[0] = 10
Gia tri cua var[1] = 100
Gia tri cua var[2] = 200

Process returned 0 (0x0)   execution time : 0.017 s
Press any key to continue.
```


Có một tình huống khi chúng ta muốn duy trì một mảng, mà có thể lưu giữ các con trỏ tới một kiểu dữ liệu int hoặc char hoặc bất kỳ kiểu nào khác. Sau đây là khai báo một mảng của các con trỏ tới một integer: **int \*contro[MAX];**

Nó khai báo contro như là một mảng các con trỏ MAX kiểu integer. Vì thế, mỗi phần tử trong contro, bây giờ giữ một con trỏ tới một giá trị int. Ví dụ sau sử dụng 3 số integer, mà sẽ được lưu giữ trong một mảng các con trỏ như sau:

```
#include <iostream>
```

```
using namespace std;
const int MAX = 3;
int main ()
{
    int var[MAX] = {10, 100, 200};
    int *contro[MAX];
    for (int i = 0; i < MAX; i++)
    {
        contro[i] = &var[i]; // gán địa chỉ của số nguyên.
    }
    for (int i = 0; i < MAX; i++)
    {
        cout << "Giá trị của var[" << i << "] = ";
        cout << *contro[i] << endl;
    }
    return 0;
}
```

Output:

 C:\Users\Administrator\Desktop\workspace\cppbook\sourcecode\myproject\bin\Debug\myproject.exe

```
Gia tri cua var[0] = 10
Gia tri cua var[1] = 100
Gia tri cua var[2] = 200
```

```
Process returned 0 (0x0)   execution time : 0.017 s
Press any key to continue.
```

Ta có thể sử dụng một mảng các con trỏ tới ký tự để lưu giữ một danh sách các chuỗi như sau:

```
#include <iostream>

using namespace std;
const int MAX = 4;

int main ()
{
    char *tensv[MAX] = {
        "Nguyen Thanh Tung",
        "Tran Minh Chinh",
        "Ho Ngoc Ha",
        "Hoang Minh Hang",
    };

    for (int i = 0; i < MAX; i++)
    {
```

```

        cout << "Gia tri cua tensv[" << i << "] = ";
        cout << tensv[i] << endl;
    }
    return 0;
}

```

Output:

```

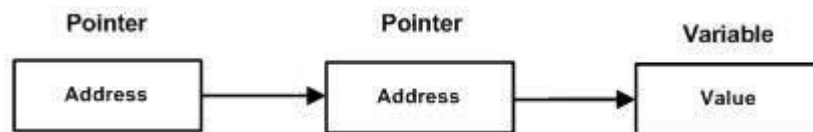
C:\Users\Administrator\Desktop\workspace\cppbook\sourcecode\myproject\bin\Debug\myproject.exe
Gia tri cua tensv[0] = Nguyen Thanh Tung
Gia tri cua tensv[1] = Tran Minh Chinh
Gia tri cua tensv[2] = Ho Ngoc Ha
Gia tri cua tensv[3] = Hoang Minh Hang

Process returned 0 (0x0)   execution time : 0.019 s
Press any key to continue.

```

#### d. Con trỏ tới con trỏ

Một con trỏ tới một con trỏ là một form không định hướng hoặc một chuỗi con trỏ. Thông thường, một con trỏ chứa địa chỉ của một biến. Khi chúng ta định nghĩa một con trỏ tới một con trỏ, con trỏ đầu tiên chứa địa chỉ của con trỏ thứ hai, mà trỏ tới vị trí mà chứa giá trị thực sự như hiển thị trong sơ đồ dưới đây:



Một biến, mà là một con trỏ tới một con trỏ, phải được khai báo. Điều này được thực hiện bởi việc đặt một dấu sao (\*) ở trước tên của nó. Ví dụ, sau đây là khai báo một con trỏ tới một con trỏ trong kiểu int: **int \*\*var;**

Khi một giá trị mục tiêu được trỏ không định hướng bởi một con trỏ tới một con trỏ, truy cập giá trị đó yêu cầu rằng toán tử dấu sao được áp dụng hai lần, như dưới ví dụ:

```

#include <iostream>

using namespace std;

int main ()
{
    int var;
    int *ptr;
    int **pptr;
}

```

```

var = 125;

// lay dia chi cua var
ptr = &var;

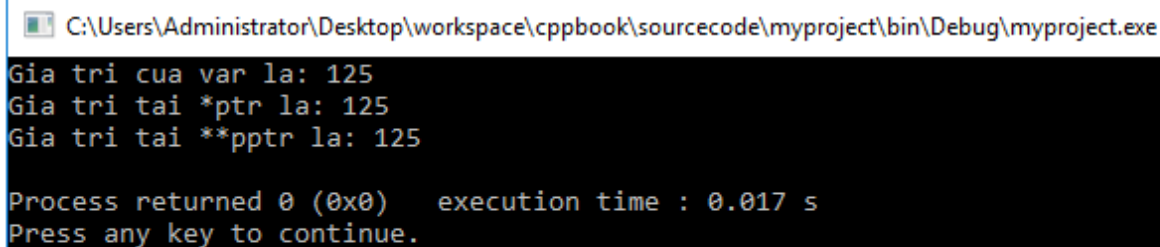
// lay dia chi cua ptr boi su dung dia chi cua toan tu &
pptr = &ptr;

// Lay gia tri boi su dung pptr
cout << "Gia tri cua var la: " << var << endl;
cout << "Gia tri tai *ptr la: " << *ptr << endl;
cout << "Gia tri tai **pptr la: " << **pptr << endl;

return 0;
}

```

Output:



```

C:\Users\Administrator\Desktop\workspace\cppbook\sourcecode\myproject\bin\Debug\myproject.exe
Gia tri cua var la: 125
Gia tri tai *ptr la: 125
Gia tri tai **pptr la: 125

Process returned 0 (0x0)   execution time : 0.017 s
Press any key to continue.

```

### *e. Truyền con trỏ tới hàm*

C++ cho phép truyền một con trỏ tới một hàm. Để làm điều này, đơn giản chỉ cần khai báo tham số hàm như ở dạng một kiểu con trỏ. Ở ví dụ đơn giản dưới đây, chúng ta truyền một con trỏ unsigned long tới một hàm và thay đổi giá trị bên trong hàm, mà phản chiếu trở lại trong khi gọi hàm:

```

#include <iostream>
#include <ctime>

using namespace std;
void laySoGiay(unsigned long *par);

int main ()
{

```

```

unsigned long sogiay;

laySoGiay( &sogiay );

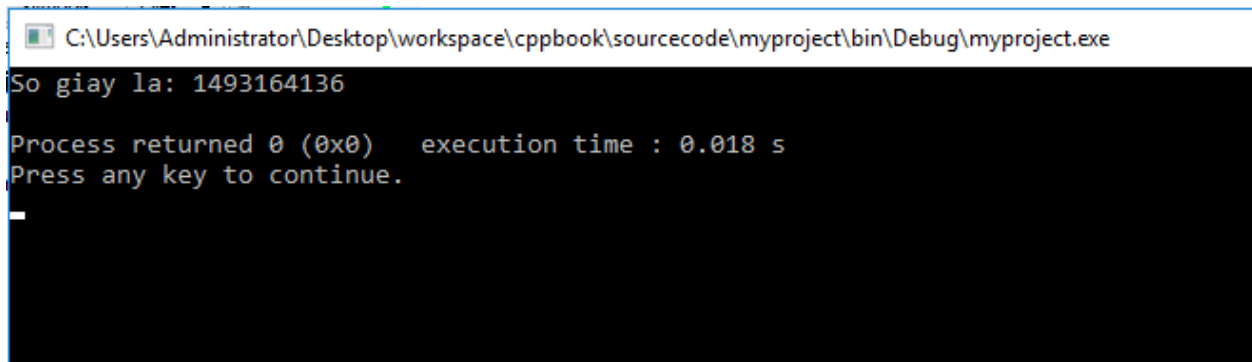
// in gia tri
cout << "So giay la: " << sogiay << endl;

return 0;
}

void laySoGiay(unsigned long *par)
{
    // Lay so giay hien tai
    *par = time( NULL );
    return;
}

```

Output:



```

C:\Users\Administrator\Desktop\workspace\cppbook\sourcecode\myproject\bin\Debug\myproject.exe
So giay la: 1493164136
Process returned 0 (0x0) execution time : 0.018 s
Press any key to continue.

```

Hàm, mà có thể chấp nhận một con trỏ, cũng có thể chấp nhận một mảng như ví dụ sau:

```

#include <iostream>
using namespace std;

// khai bao ham:
double giaTriTB(int *arr, int size);

int main ()
{
    // mot mang so nguyen co 5 phan tu.
    int doanhSoBH[5] = {122, 35, 27, 235, 60};
    double trungbinh;

    // truyen con tro toi mang duoi dang mot tham so.
    trungbinh = giaTriTB( doanhSoBH, 5 ) ;

    // hien thi ket qua
    cout << "Gia tri trung binh la: " << trungbinh << endl;
}

```

```

    return 0;
}

double giaTriTB(int *arr, int size)
{
    int    i, sum = 0;
    double trungbinh;

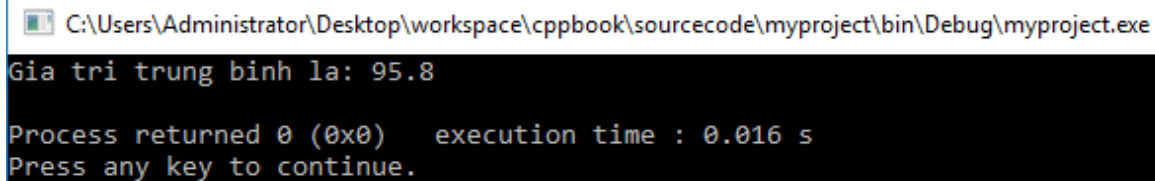
    for (i = 0; i < size; ++i)
    {
        sum += arr[i];
    }

    trungbinh = double(sum) / size;

    return trungbinh;
}

```

Output:



```

C:\Users\Administrator\Desktop\workspace\cppbook\sourcecode\myproject\bin\Debug\myproject.exe
Gia tri trung binh la: 95.8

Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
_

```

### *f. Trả về con trỏ từ hàm*

Như chúng ta đã thấy cách C++ cho phép trả về một mảng từ một hàm, tương tự như vậy, C++ cho phép trả về một con trỏ từ một hàm. Để làm điều này, phải khai báo một hàm trả về một con trỏ như sau:

```

int * tenHam()
{
    .
    .
    .
}

```



Điều thứ hai cần ghi nhớ là, nó không là ý kiến tốt để trả về địa chỉ của một biến cục bộ tới ngoại vi của một hàm, vì thế bạn sẽ phải định nghĩa biến cục bộ như là biến static.

Bây giờ, giả sử hàm sau sẽ tạo 10 số ngẫu nhiên và trả về chúng bởi sử dụng một tên mảng mà biểu diễn một con trỏ, ví dụ, địa chỉ đầu tiên của phần tử mảng đầu tiên.

```
#include <iostream>
#include <ctime>
#include <stdlib.h>
using namespace std;

// phan dinh nghia ham de tao va tra ve cac so ngau nhien.
int * soNgauNhien( )
{
    static int  r[10];

    srand( (unsigned)time( NULL ) );
    for (int i = 0; i < 10; ++i)
    {
        r[i] = rand();
        cout << r[i] << endl;
    }

    return r;
}

// ham main de goi phan dinh nghia ham tren.
int main ()
{
    // mot con tro tro toi mot so nguyen.
    int *p;

    p = soNgauNhien();
    for ( int i = 0; i < 10; i++ )
    {
        cout << "Gia tri cua *(p + " << i << " ) : ";
        cout << *(p + i) << endl;
    }

    return 0;
}
```

Output:

```
C:\Users\Administrator\Desktop\workspace\cppbook\sourcecode\myprojec
4581
6753
29400
12640
1263
26616
17456
1091
22470
11319
Gia tri cua *(p + 0) : 4581
Gia tri cua *(p + 1) : 6753
Gia tri cua *(p + 2) : 29400
Gia tri cua *(p + 3) : 12640
Gia tri cua *(p + 4) : 1263
Gia tri cua *(p + 5) : 26616
Gia tri cua *(p + 6) : 17456
Gia tri cua *(p + 7) : 1091
Gia tri cua *(p + 8) : 22470
Gia tri cua *(p + 9) : 11319

Process returned 0 (0x0)   execution time : 0.023 s
Press any key to continue.
```

**PHẦN HAI:**

# **MỘT SỐ KỸ THUẬT NÂNG CAO**

# §1: DEBUG VỚI CODE::BLOCKS

## 1. Debug là gì ?

Lập trình vốn là một công việc nặng nhọc và căng thẳng, vì thế không phải lúc nào người viết code cũng thật sự tỉnh táo để thực hiện chương trình của mình một cách trơn tru nhất. Đa phần trong chúng ta đều không thể tránh khỏi những lỗi khi lập trình, do đó sinh ra “bug”.

Bug hiểu theo nghĩa đen là những con bọ, ám chỉ các lỗi xảy ra trong logic, hay bất kì vấn đề gì khiến cho chương trình không thực hiện đúng như ý muốn của người lập trình. Bug luôn tìm tòi khắp mọi nơi, trên từng dòng code và có thể nói là chúng rất khó kiểm soát.

Ví dụ: *Viết chương trình tính tổng hai số được nhập vào từ bàn phím.*

Ta xét đoạn code:

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int a, b;
    cout << "Nhập a và b: ";
    cin >> a >> b;
    cout << a + b;
    return 0;
}
```

Khi nhập  $a = 3$ ,  $b = 4$  thì hiển nhiên ta sẽ được kết quả là 7. Tuy nhiên ta thử nhập 3.1 và 4.1 thì kết quả vẫn là 7, đây là kết quả không chính xác. Do đó ta xem như đoạn chương trình này toàn tại bug khiến cho kết quả của chương trình là không đúng.

Để khắc phục lỗi trên, ta có thể kiểm tra cụ thể từng dòng code để xem lỗi xảy ra ở đâu. Tuy nhiên, với những chương trình lớn hàng trăm, hàng nghìn dòng code thì việc này là dường như bất khả thi. Do đó trên hầu hết các IDE hiện đại đều hỗ trợ khả năng gỡ rối (debug) chương trình bằng cách chạy từng dòng lệnh (step by step) và xem xét sự thay đổi giá trị của biến (watch variables).

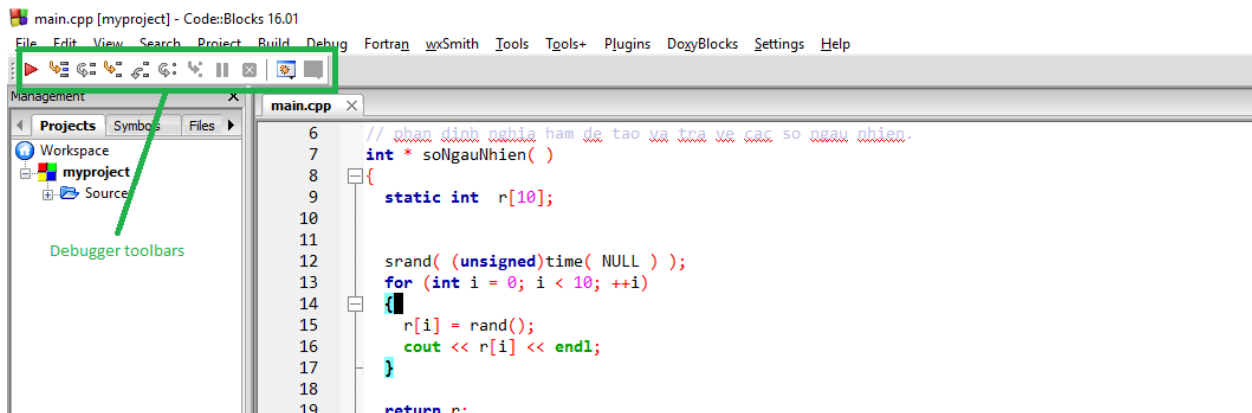
## 2. Debug với Code::Blocks

### a. Debugger toolbars

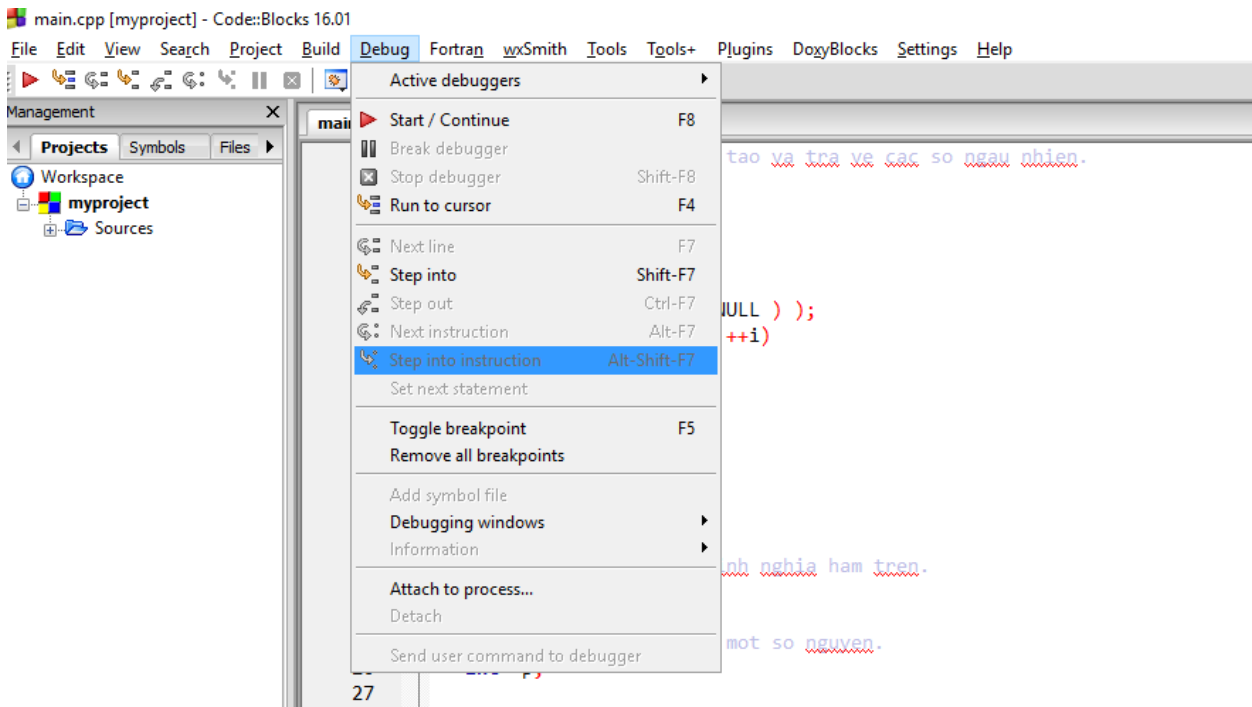
Để mở debugger toolbars trong Code::Blocks ta lần lượt thao tác như sau:

**View → Toolbars → Debugger**

Khi đó ta sẽ thấy xuất hiện những công cụ của debugger trên thanh toolbars.



Từ đây ta có thể sử dụng toolbars này để debug hoặc sử dụng option **Debug**



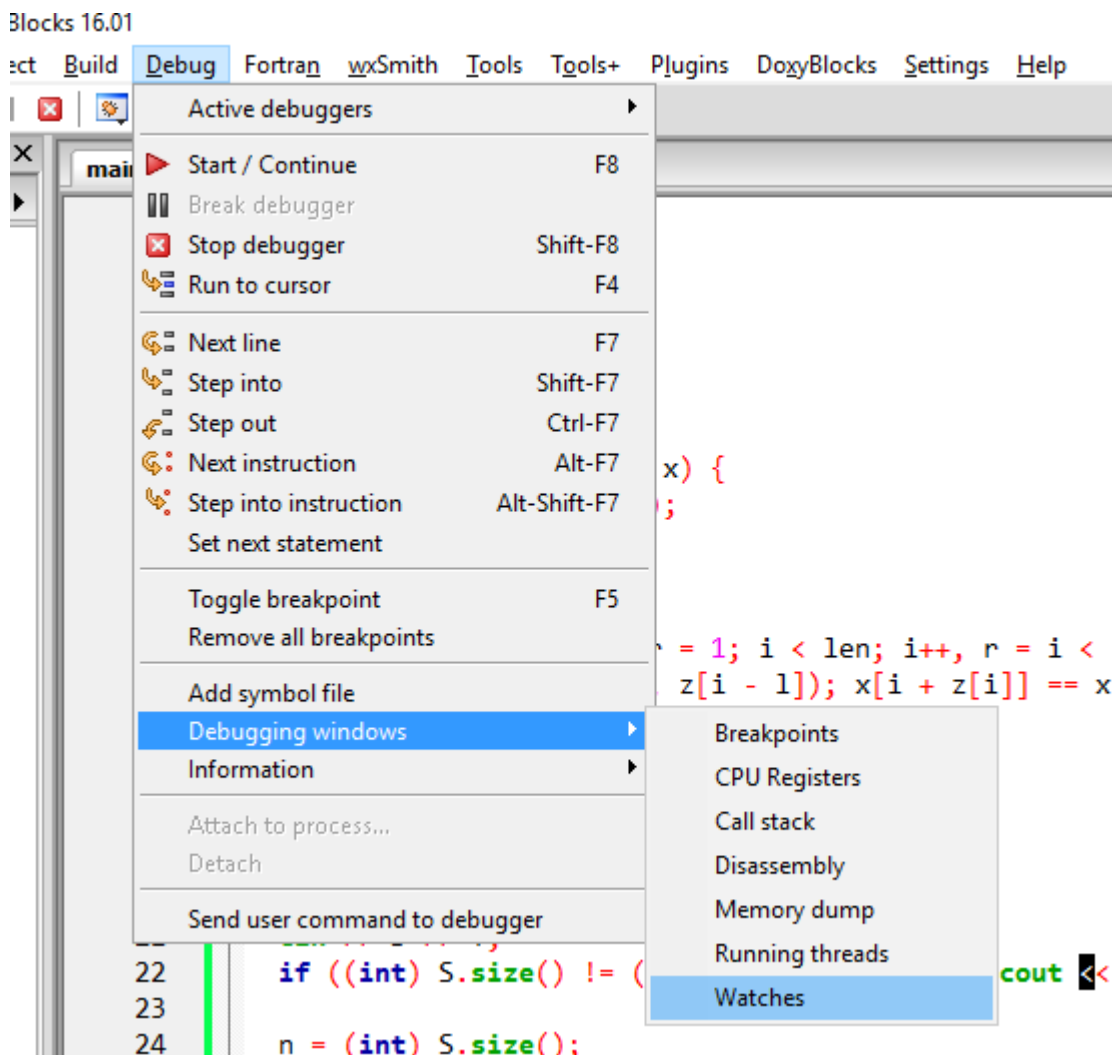
Hoặc sử dụng các phím tắt thường dùng như đã được mô tả trong option **Debug**:

- **F8**: Bắt đầu hoặc tiếp tục việc debug còn đang dở.
- **Shift + F8**: Dừng việc debug.
- **F4**: Chuyển đến dòng đang chứa con trỏ.
- **F7**: Đến dòng chứa lệnh tiếp theo.
- **Shift + F7**: Vào thân hàm xuất hiện tại dòng hiện tại.
- **Ctrl + F7**: Thoát khỏi hàm.
- **F5**: Đặt hoặc xóa bỏ breakpoints.

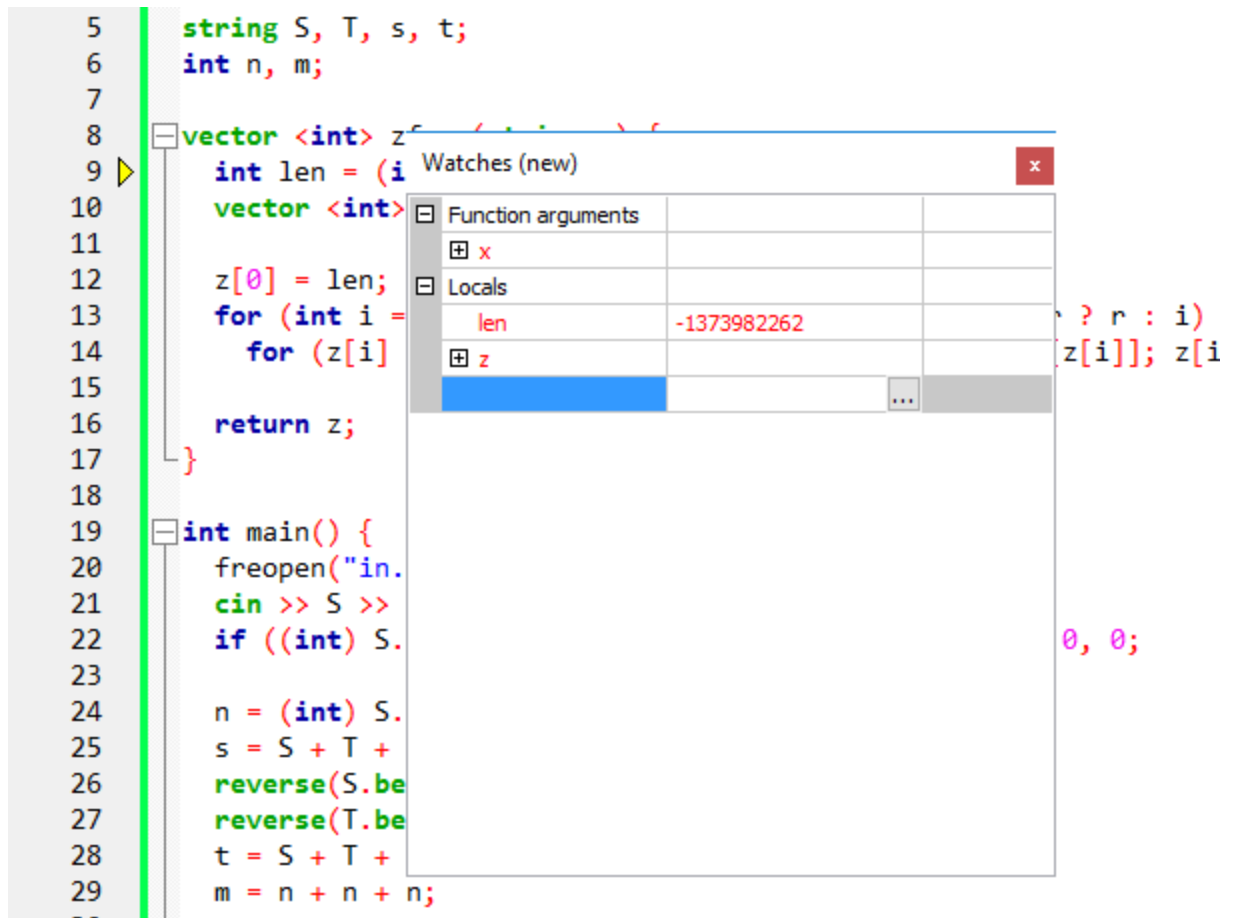
### *b. Những cách debug thường dùng*

**Watches windows:** Đây là một cửa sổ cho phép ta xem giá trị của biến hoặc thậm chí là kết quả của một số biểu thức với biến. Để mở **watches windows** ta phải đảm bảo debugger đã được chạy và thao tác:

Debug → Debugging windows → watches



Watches windows sẽ hiện ra:



Bảng này sẽ bao gồm:

- Function arguments: Danh sách các tham số của hàm chứa con trỏ debugger.
- Locals: Chứa danh sách biến cục bộ trong hàm chứa con trỏ debugger.

Ta có thể gõ bất kì biến hợp lệ nào vào ô màu xanh để debugger hiện ra giá trị của biến, hoặc thậm chí là một biểu thức hợp lệ.

Dưới đây là chi tiết cách thức hoạt động của một số phương thức thông dụng.

- **F8/Shift + F8**: Đây chỉ đơn giản là bắt đầu hoặc dừng việc debug.

- **F4 (Run to cursor):** Dịch nôm na là chuyển đến dòng chứa con trỏ hiện tại.

Ví dụ:

```
main.cpp x
1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  string S, T, s, t;
6  int n, m;
7
8  vector<int> zfunc(string x) {
9      int len = (int) x.size();
10     vector<int> z(len);
11
12     z[0] = len;
13     for (int i = 1, l = 0, r = 1; i < len; i++, r = i < r ? r : i)
14         for (z[i] = min(r - i, z[i - 1]); x[i + z[i]] == x[z[i]]; z[i]++, r = i + z[i], l = i);
15
16     return z;
17 }
```

Ta nghi ngờ dòng số 14 là nguyên nhân gây ra lỗi. Cho nên ta đặt con trỏ chuột ở dòng thứ 14. Sau đó nhấn **F4** thì ta sẽ được con trỏ debug sẽ dừng lại ở dòng 14.

```
main.cpp x IN.TXT x
1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  string S, T, s, t;
6  int n, m;
7
8  vector<int> zfunc(string x) {
9      int len = (int) x.size();
10     vector<int> z(len);
11
12     z[0] = len;
13     for (int i = 1, l = 0, r = 1; i < len; i++, r = i < r ? r : i)
14         for (z[i] = min(r - i, z[i - 1]); x[i + z[i]] == x[z[i]]; z[i]++, r = i + z[i], l = i);
15
16     return z;
17 }
18
```

- **Shift + F7 và F7:** Có thể nói đây là hai tổ hợp phím dùng nhiều nhất khi debug. Để bắt đầu debug ta nhấn tổ hợp phím **Shift + F7** chức năng của nó là (Step into) và vì main là một hàm cho nên nó sẽ dừng lại ở ngay dòng đầu tiên của hàm main để kết thúc việc debug ta có thể nhấn **F8** hoặc **Shift + F8**. Để dịch chuyển đến dòng lệnh tiếp theo thì nhấn **F7**. Còn nếu muốn vào một hàm nào đó thì phải nhấn **Shift + F7** nếu không debugger sẽ tự động bỏ qua việc debug bên trong hàm đó. Ví dụ tại dòng 31, muốn xem hàm `zfunc(s)` hoạt động như thế nào thì ta phải nhấn **Shift + F7**.



```

28     t = S + T + T;
29     m = n + n + n;
30
31     vector<int> ZL = zfunc(s);
32     vector<int> ZR = zfunc(t);

```

Khi đó, con trỏ debugger sẽ dừng lại ở dòng đầu tiên bên trong hàm `zfunc(s)`:

```

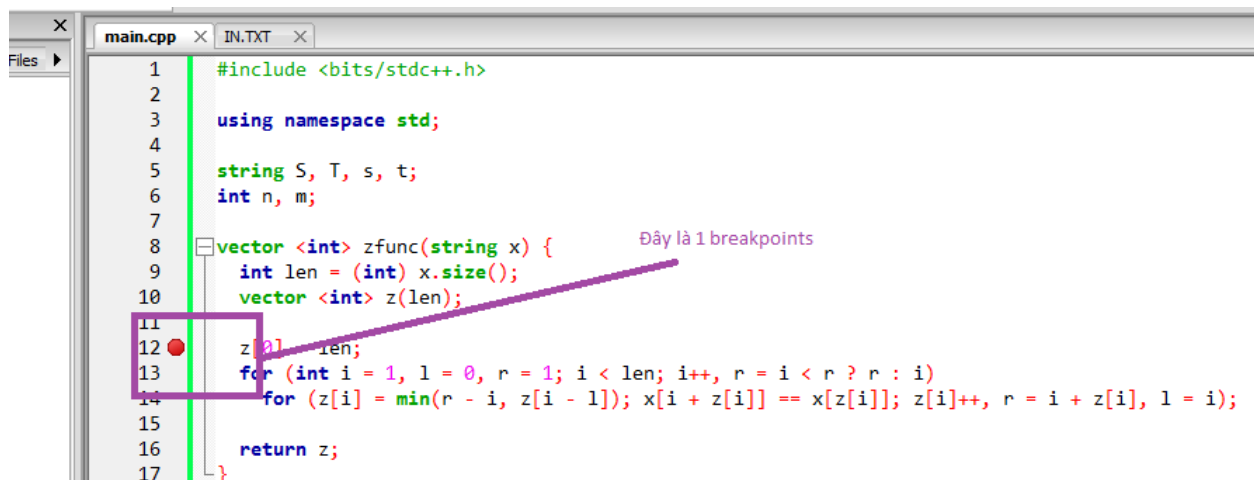
2
3     using namespace std;
4
5     string S, T, s, t;
6     int n, m;
7
8     vector<int> zfunc(string x) {
9     int len = (int) x.size();
10    vector<int> z(len);
11
12    z[0] = len;
13    for (int i = 1, l = 0, r = 1; i < len; i++, r = i < r ? r : i)
14        for (z[i] = min(r - i, z[i - 1]); x[i + z[i]] == x[z[i]]; z[i]++, r = i + z[i], l = i);
15
16    return z;
17 }

```

Ngược lại, tại một vị trí bất kì trong hàm thì để thoát ra khỏi hàm đó ta chỉ cần nhấn **Ctrl + F7**.

### c. Breakpoints

Breakpoints là vị trí mà chương trình sẽ dừng lại để người lập trình xem xét sự thay đổi của các biến qua từng dòng lệnh, từ đó phát hiện ra vị trí dòng code bị lỗi. Breakpoint được kí hiệu bằng chấm tròn màu đỏ ở đầu dòng code. Để tạo ra một breakpoint, cách đơn giản nhất là click chuột vào đầu dòng code (như trong hình). Để huỷ breakpoint, chỉ cần click chuột vào breakpoint đó một lần nữa. Ngoài ra cũng có thể tạo/huỷ breakpoint bằng phím F5



Công dụng của break point là khi ta nhấn phím **F8** để debug toàn bộ chương trình thì chương trình sẽ dừng lại ở vị trí được đặt breakpoints, giúp chúng ta có thể nhanh chóng tìm ra lỗi hơn là **F7** từng dòng code.

## §2: STANDARD TEMPLATE LIBRARY (STL)

Có thể nói STL chính là một trong những lí do chính giúp C++ đang dần thay thế pascal trong các cuộc thi lập trình ở Việt Nam. Việc sử dụng thành thạo STL sẽ là rất quan trọng nếu chúng ta có ý định tham gia các kì thi về Tin Học. Như thầy Lê Minh Hoàng đã từng nói “STL sẽ nổi dài khả năng lập trình của các bạn”. Đúng như vậy, STL là một kho thư viện khổng lồ chứa các công cụ giúp học sinh giải được bài toán một cách dễ dàng hơn rất nhiều. Tuy nhiên, khi học STL ta cũng phải chắc rằng mình đã nắm vững những thuật toán được sử dụng trong STL, bởi vì trong nhiều trường hợp việc sử dụng STL sẽ trở thành con dao hai lưỡi nếu ta lạm dụng quá nhiều, do STL bao gồm các template được quy định bởi ngôn ngữ lập trình nên việc debug, gỡ rối là không hề dễ dàng và chúng rất khó kiểm soát. Trong bài này ta sẽ tìm hiểu một số STL hay dùng nhất trong C++.

### 1. STL là gì ?

Standard Template Library (STL) trong C++ là một tập hợp các lớp Template mạnh mẽ trong C++ để cung cấp các lớp và các hàm được tạo theo khuôn mẫu cho mục đích lập trình tổng quát, mà triển khai nhiều thuật toán và cấu trúc dữ liệu được sử dụng phổ biến và thông dụng như vector, list, queue và stack, ....

Thư viện chuẩn C++ (C++ Standard Library) có thể chia thành hai phần chính:

- Thư viện hàm chuẩn (Standard Function Library): Thư viện này gồm các hàm mang mục đích tổng quát mà không là một phần của bất kì lớp nào (class). SFL mang tính kế thừa từ C. Ví dụ: I/O, xử lí chuỗi kí tự, toán học, Date, Time, Place, cấp phát động, hỗn hợp, các hàm cho wide-character, ..
- Thư viện lớp hướng đối tượng (Object Oriented Class Library): Đây là một tập hợp của các lớp và các hàm liên kết. Ví dụ: các lớp I/O chuẩn C++, string, numeric, stl container, stl algorithm, các đối tượng thuộc stl function, stl iterator, stl allocator, localization, các lớp exception handling, thư viện hàm hỗn hợp, ...

Tuy nhiên, ba cấu trúc thực sự đã tạo ra sự mạnh mẽ và khác biệt cho C++ đó chính là Containers, Algorithms và Iterators.

### 2. Một số lưu ý khi dùng STL

Khi sử dụng một STL nào đó, ta buộc phải khai báo bằng cú pháp sau ở đầu chương trình:

```
#include <tên thư viện>;
```

Ví dụ, nếu muốn sử dụng thư viện vector ta phải khai báo:

```
#include <vector>;
```

Ta nên khai báo namespace std, vì đa số các thư viện dùng chung namespace này, ta khai báo như sau:

```
using namespace std;
```

Ngoài ra, khi khai báo một đối tượng lưu trữ với kiểu dữ liệu nào đó thì ta phải khai báo theo cú pháp tương tự như khi báo một vector với kiểu int như sau:

```
vector <int> a;
```

Hãy đặt biệt chú ý đến cặp dấu "<>", khi khai báo một cấu trúc đối tượng trong đối tượng thì những cặp dấu "<>" không được viết liền nhau để tránh nhầm lẫn với toán tử trích luồng. Ví dụ

```
vector < vector<int> > N; // Khai báo đúng
```

```
vector <vector<int>> N; // Khai báo sai
```

Tuy nhiên, kể từ phiên bản C++ 11 thì điều phiền toái này không xảy ra nữa.

Để truy cập hàm hoặc phương thức thành viên của một STL, ta có thể thực hiện tương tự như ví dụ dưới đây.

```
vector <int> v;
```

```
v.clear(); // Phương thức này cho phép xóa toàn bộ phần tử của vector v
```

### 3. Iterator

Trong C++, một biến lặp là một đối tượng bất kì, trỏ tới một số phần tử trong phạm vi của các phần tử (như mảng hoặc container), có khả năng để lặp các phần tử trong phạm vi bằng cách sử dụng một tập các toán tử (operators) (như so sánh, tăng (++),...). Dạng rõ ràng nhất của iterator là một con trỏ: Một con trỏ có thể trỏ tới các phần tử trong mảng, và có thể lặp thông qua sử dụng toán tử tăng (++). Tuy nhiên, cũng có các dạng khác của iterator. Ví dụ: mỗi loại container (chẳng hạn như vector) có một loại iterator được thiết kế để lặp các phần tử của nó một cách hiệu quả.

Để khai báo iterator, ta dùng cú pháp:

```
<containers>::iterator it;
<containers>::reverse_iterator rit; // Đối với iterator ngược
```

Chẳng hạn:

```
vector<int>::iterator it;
vecotr<int>::reverse_iterator rit;
```

Ta có thể thực hiện một số thao tác với iterator như:

- So sánh: “==” và “!=”.
- Gán “=”.
- Cộng trừ.
- Lấy giá trị “\*”.

Ta có một vài iterator hay dùng trong các containers như.

- begin(): Trả về iterator đầu tiên của containers.
- end(): Trả về iterator cuối cùng của containers.
- rbegin(): Trả về iterator đầu tiên của vector theo chiều ngược.
- rend(): Trả về iterator cuối cùng của vector theo chiều ngược.

#### 4. Containers

Containers thực chất là một nhóm các thư viện cung cấp “dịch vụ” lưu trữ các đối tượng cụ thể. Ở mỗi containers sẽ cho phép thực hiện một số thao tác nhất định với đối tượng được containers lưu trữ. Các containers có thể chia thành những nhóm sau:

- **Sequence containers:** Vector, list, deque, ...
- **Associative containers:** Set, multiset, map, multimap, ...
- **Containers adapters:** Stack, queue, priority\_queue, ...
- **String:** String, rope, ...
- **Bitset**

##### a. Vector

Vector thật chất là một cấu trúc mảng động, người lập trình không cần quy định kích thước của vector như một mảng thông thường. Vector sẽ rất hữu ích khi cài đặt những mảng có số lượng phần tử không biết trước, tuy nhiên chúng ta vẫn có thể quy định kích thước của vector. Ví dụ

```
vector <int> v; // Khai báo vector khi chưa biết số lượng phần tử
```

```
vector<int> v(10); // Khai báo vector với 10 phần tử rỗng
vector<int> v(10, 1); // Khai báo vector với 10 phần tử có giá trị 1
```

Lưu ý là cũng như mảng, index của vector cũng bắt đầu từ 0.

Dưới đây là một số hàm thành viên của STL vector:

- empty(): Trả về giá trị true nếu vector rỗng, ngược lại là giá trị false.
- size(): Trả về giá trị là số lượng phần tử của vector.
- resize(n): Xóa phần tử dư hoặc thêm một số phần tử để số lượng phần tử của vector đúng bằng n.
- at(i): Trả về giá trị tại vị trí i của vector, v.at(i) tương đương với v[i].
- front(): Trả về giá trị đầu tiên của vector từ trái sang phải. v.front() = v[0].
- push\_back(value): Thêm phần tử value vào cuối vector.
- pop\_back(): Xóa phần tử cuối của vector.
- assign(n, t): Khởi tạo n phần tử đầu tiên của vector với giá trị bằng t.
- swap(v2): Hoán đổi với vector v2.
- erase(v): Xóa phần tử tại vị trí v (v là một iterator). Hoặc có thể dùng erase(v1, v2) để xóa phần tử từ vị trí v1 đến v2 của vector (v1, v2 là iterator). Thậm chí là xóa toàn bộ vector với erase().
- insert(i, T): Chèn phần tử T tại iterator i. Tương tự như erase ta cũng có insert(i, n, T) = chèn n phần tử T bắt đầu từ iterator i. Hoặc insert(i, v1, v2) tức là chèn vào vị trí iterator i những phần tử từ iterator v1 đến iterator v2 (chú ý v1, v2 có thể là iterator của một vector khác).

Ví dụ:

```
#include <iostream>
#include <vector>

using namespace std;

void view(vector<int> vec) {
    for (vector<int>::iterator it = vec.begin(); it != vec.end(); it++)
        cout << *it << " ";
    cout << "\n";
}

int main() {
    vector<int> v(10, 1);
    view(v); // 1 1 1 1 1 1 1 1 1 1
    cout << v.size() << "\n"; // 10
    cout << v.front() << "\n"; // 1

    for (int i = 0; i < v.size(); i++)
        v[i] = i;
}
```

```

v.erase(v.begin() + 5);
view(v); // 0 1 2 3 4 6 7 8 9

v.erase(v.begin() + 5, v.end());
view(v); // 0 1 2 3 4

vector<int> V;

for (int i = 1; i <= 10; i++)
    V.push_back((i + 1) * (i - 1));

V.swap(v);
view(V); // 0 1 2 3 4

V.swap(v);
view(V); // 0 3 8 15 24 35 48 63 80 99

vector<int>::iterator it = v.begin();
V.insert(V.begin() + 5, it, it + 3);
view(V); // 0 3 8 15 24 0 1 2 35 48 63 80 99

V.pop_back();
view(V); // 0 3 8 15 24 0 1 2 35 48 63 80

V.resize(10);
view(V); // 0 3 8 15 24 0 1 2 35 48

V.assign(10, 0);
view(V); // 0 0 0 0 0 0 0 0 0 0

return 0;
}

```

### ***b. List***

List thực chất là cấu trúc dữ liệu Double-Linked List, trước đây chúng ta thường phải cài đặt bằng con trỏ và làm việc với nó khá nhọc nhằn. Tuy nhiên, C++ đã cung cấp hẳn một thư viện hỗ trợ cho cấu trúc dữ liệu này. Nếu chưa biết về cách hoạt động của Double-Linked List, các bạn có thể tìm hiểu ở một tài liệu khác. Ở đây chúng ta chỉ nói về cách sử dụng STL list trong C++.

Cách khai báo một list: **list<kiểu dữ liệu> tên biến;**

Ví dụ: **list<int> mylist;**

Do cùng là sequence containers cho nên list cũng có một số phương thức giống như vector và hoạt động tương tự: front(), back(), push\_back(), pop\_back(), assign(), size(), empty(), swap(), erase(), resize(), size(), ...

Ngoài ra, ta cũng có một số phương thức mà list có còn vector thì không:

- `push_front(value)`: Thêm phần tử `value` vào đầu list.
- `pop_front()`: Xóa phần tử đầu của list.
- `sort(comparison_function)`: Hàm sắp xếp list theo thứ tự được quy định bởi `compare_function` (Cách viết `compare function` sẽ được đề cập ở một bài khác). Mặc định nếu không có `comparison_function` hàm `sort` sẽ sắp xếp list theo thứ tự tăng dần.

- `merge(s, comparison_function)`: Trộn list hiện tại với list `s`. Sau khi trộn ta sẽ được một list đã được sắp xếp. Lưu ý bản chất hàm này dựa trên thuật toán sắp xếp trộn (`merge sort`) cho nên hai list trước khi trộn phải là list đã được sắp xếp.

Ví dụ:

```
#include <iostream>
#include <list>

using namespace std;

void view(list<int> L) {
    for (list<int>::iterator it = L.begin(); it != L.end(); it++)
        cout << *it << " ";
    cout << "\n";
}

bool comparision(const int &first, const int &second) {
    return first > second;
}

int main() {
    list<int> myList;

    for (int i = 1; i <= 10; i++)
        myList.push_back(i);
    view(myList); // 1 2 3 4 5 6 7 8 9 10

    myList.push_front(100);
    myList.pop_back();
    view(myList); // 100 1 2 3 4 5 6 7 8 9

    list<string> L;
    L.push_back("one");
    L.push_back("two");
    L.push_back("three");
```



```

L.sort();
list<string>::iterator it;
for (it = L.begin(); it != L.end(); it++)
    cout << *it << " ";
cout << "\n"; // one three two

list<int> L2;
for (int i = 1; i <= 10; i++)
    L2.push_front(i);
view(L2);

myList.sort();
view(myList); // 1 2 3 4 5 6 7 8 9 100

myList.sort(comparision);
view(myList); // 100 9 8 7 6 5 4 3 2 1

myList.merge(L2, comparision);
view(myList); // 100 10 9 9 8 8 7 7 6 6 5 5 4 4 3 3 2 2 1 1

return 0;
}

```

### c. Pair

Đôi khi trong lúc lập trình, ta cần lưu giữ một đối tượng với một cặp giá trị. Khi đó ta thường sẽ nghĩ đến struct. Ví dụ:

```

struct p {
    int a;
    char b;
};

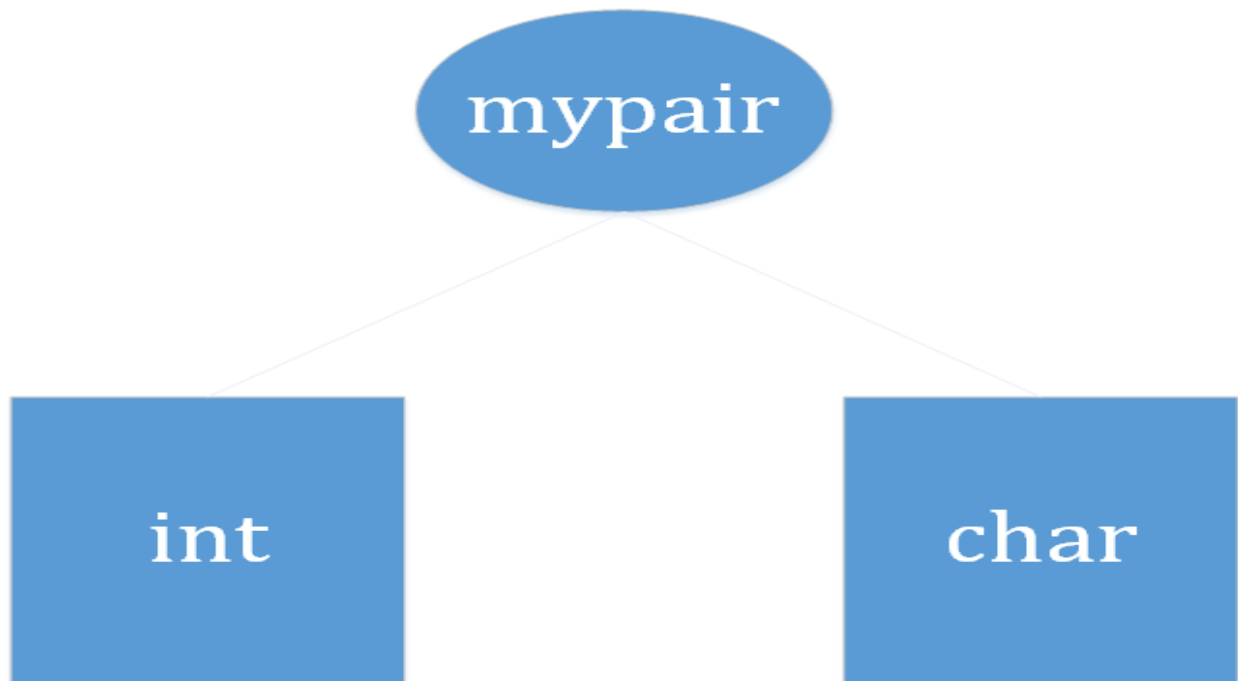
```

Tuy nhiên, có một cách khác đó là dùng pair. Pair cho phép ta gộp 2 đối tượng thành một cặp, có thể 2 đối tượng cùng kiểu hoặc khác kiểu, với thuộc tính "first" là đối tượng đầu tiên, "second" là đối tượng thứ 2 trong cặp đối tượng "position/ data". Để khai báo pair ta dùng cú pháp:

**pair** <kiểu giá trị 1, kiểu giá trị 2> tên đối tượng;

Ví dụ: **pair** <int, char> mypair;

Khi đó mypair sẽ có cấu trúc:



Để lấy dữ liệu trong pair, ta dùng hai thuộc tính là first và second. Để tạo một pair với một cặp dữ liệu có sẵn ta dùng constructor `make_pair(x, y)`, trong đó `x` là giá trị của thuộc tính first, `y` là giá trị của thuộc tính second. Ví dụ:

```
pair<int, char> p = make_pair(2, 'D');
cout << p.first << " " << p.second; // 2 D
```

Hoặc ta cũng có thể khởi tạo pair bằng cú pháp:

```
pair<int, char> p(2, 'D');
```

Ví dụ về sử dụng pair:

```
#include <cstdio>
#include <iostream>
#include <vector>

using namespace std;

int main() {
    pair<int, string> p;
    p = make_pair(3, "NHTN");
    cout << p.first << " " << p.second << "\n"; // 2 NHTN

    vector< pair<int, int> > p2;
    for (int i = 1; i <= 10; i++)
        p2.push_back(make_pair(i, 10 - i));
```

```

vector < pair<int, int> >::iterator it;
for (it = p2.begin(); it != p2.end(); it++)
    cout << (*it).first << " " << (*it).second << "\n";
/*
1 9
2 8
3 7
4 6
5 5
6 4
7 3
8 2
9 1
10 0
*/

return 0;
}

```

#### d. Map

Map thuộc loại Associative containers (Cấu trúc liên kết) được thiết kế như là một cây đỏ-đen tiêu chuẩn (red-black tree). Map được tổ chức theo kiểu key-value tức là một key sẽ được ánh xạ với một value. Các key sẽ được sắp xếp theo một thứ tự do hàm Compare quy định. Các thao tác chèn, xóa, tìm kiếm, ... với map được thực hiện với độ phức tạp logarit. Cú pháp khai báo:

**map<kiểu dữ liệu, kiểu dữ liệu> tên biến;**

Ví dụ:

```

map<string, int> m;
m["duy"] = 1;
m["crush"] = 100000;

```

Trong ví dụ trên, ta đã khai báo map m với key là kiểu string và value thuộc kiểu int. Ta có thể duyệt qua các phần tử của map bằng iterator, và các hàm thành viên cũng hoạt động giống như vector và list. Ví dụ với hàm insert:

```

map<int, char*> mymap, copymymap;
mymap[0] = "a";
mymap[1] = "b";
mymap[5] = "c";

// chèn vào copymymap cặp đối tượng (10, "c")
copymymap.insert(pair<int, char*>(10, "c"));
// chèn (-1, "d") vào copymymap từ vị trí bắt đầu của copymymap
copymymap.insert(copymymap.begin(), pair<int, char*>(-1, "d"));
// chèn mymap vào copymymap

```

```
copymymap.insert(mymap.begin(), mymap.end());
// => copymymap = {(-1,"d"),(0,"a"),(1,"b"),(5,"c"),(10,"c")}
```

Tuy nhiên, thật ra việc chèn ở một vị trí nào đó là vô nghĩa, Vì các key trong map đều được sắp xếp theo một quy tắc được quy định sẵn. Như ở trên key là các số nguyên nên chúng được sắp xếp tăng dần, cho nên dù không cần xác định vị trí iterator cần chèn thì vị trí của value được chèn vào vẫn được chương trình tự xác định.

Hàm thành viên erase() cho phép ta xóa một đối tượng hay hàm clear() cho phép ta xóa tất cả đối tượng trong "map", hàm erase() được override với các chức năng trong ví dụ sau:

```
map<int, char*> mymap, copymymap;
mymap[0] = "a";
mymap[1] = "b";
mymap[5] = "c";
mymap[7] = "d";
mymap[9] = "e";

// xóa cặp đối tượng với "position" là 5
mymap.erase(5);
// => mymap = {(0,"a"),(1,"b"),(7,"d"),(9,"e")}

map<int, char*>::iterator var = mymap.begin();
// xóa cặp đối tượng mà var đang truy cập
mymap.erase(var); // => mymap = {(1,"b"),(7,"d"),(9,"e")}

var = mymap.find(7); // => var truy cập đến (7,"d")

// xóa từ vị trí var đang truy cập cho đến (9,"e")
mymap.erase(var, mymap.end()); // => mymap = {(1,"b")}
```

Hàm thành viên find() cho phép ta tìm kiếm theo "position" của cặp giá trị "position/data"

```
map<int, char*> mymap;
mymap[0] = "a";
mymap[1] = "b";
mymap[5] = "c";
mymap[9] = "e";
map<int, char*>::iterator var = mymap.find(5);
// var -> (5,"c")
```

#### *d. Set*

Set là một cấu trúc dữ liệu liên kết lưu trữ một danh sách các đối tượng được sắp xếp theo thứ tự được quy định bởi hàm compare tiêu chuẩn hoặc do người lập trình quy định. Ứng dụng mạnh nhất của set là thêm vào set một đối tượng nào đó hoặc kiểm tra đối tượng đó có xuất hiện trong set hay chưa với độ phức tạp logarit.

Một số hàm hữu ích:

- insert(object): Chèn đối tượng object vào set.
- erase(v): Xóa đối tượng khỏi erase (v ở đây có thể là object, iterator). Ví dụ:

```
// erasing from set
#include <iostream>
#include <set>

int main ()
{
    std::set<int> myset;
    std::set<int>::iterator it;

    // insert some values:
    for (int i=1; i<10; i++) myset.insert(i*10); // 10 20 30 40 50
    60 70 80 90

    it = myset.begin();
    ++it; // "it" points now
    to 20

    myset.erase (it);

    myset.erase (40);

    it = myset.find (60);
    myset.erase (it, myset.end());

    std::cout << "myset contains:";
    for (it=myset.begin(); it!=myset.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';

    return 0;
}
```

- find(value): Trả về iterator chứa value. Ví dụ:

```
// set::find
#include <iostream>
#include <set>

int main ()
```

```

{
    std::set<int> myset;
    std::set<int>::iterator it;

    // set some initial values:
    for (int i=1; i<=5; i++) myset.insert(i*10);    // set: 10 20 30
40 50

    it=myset.find(20);
    myset.erase (it);
    myset.erase (myset.find(40));

    std::cout << "myset contains:";
    for (it=myset.begin(); it!=myset.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';

    return 0;
}

```

- count(value): Trả về số lượng phần tử bằng với value. Ví dụ:

```

// set::count
#include <iostream>
#include <set>

int main ()
{
    std::set<int> myset;

    // set some initial values:
    for (int i=1; i<5; ++i) myset.insert(i*3);    // set: 3 6 9 12

    for (int i=0; i<10; ++i)
    {
        std::cout << i;
        if (myset.count(i)!=0)
            std::cout << " is an element of myset.\n";
        else
            std::cout << " is not an element of myset.\n";
    }

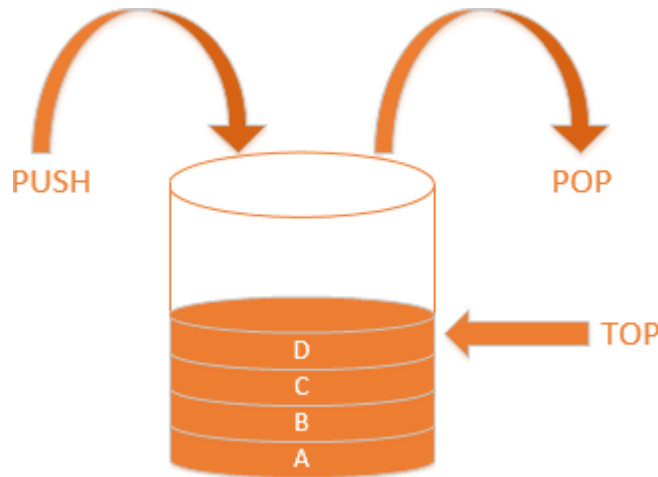
    return 0;
}

```

### *e. Stack*

Stack và queue là hai cấu trúc dữ liệu quan trọng trong lập trình. Trong C++, hai cấu trúc dữ liệu này cũng được cung cấp với khá đầy đủ những chức năng cần thiết. Là một.

Stack là một kiểu cấu trúc dữ liệu và cơ chế của nó là LIFO (Last In First Out) nghĩa là vào sau ra trước. Ta có thể hình dung Stack như một chồng đĩa, ta chỉ có thể lấy chiếc đĩa ra hoặc thêm một chiếc đĩa khác vào trên đỉnh của nó và chiếc đĩa nằm trên đỉnh đó được gọi là Top.



Các phương thức quan trọng của stack là:

- `empty()`: Dùng để kiểm tra stack có rỗng hay không. Nếu stack rỗng trả về `true` và ngược lại trả về `false`.
- `push(value)`: Thêm một phần tử `value` vào đỉnh stack.
- `top()`: Trả về giá trị của phần tử ở đỉnh stack.
- `pop()`: Lấy phần tử ở đỉnh stack ra ngoài (xóa).
- `clear()`: Xóa toàn bộ stack.

Ví dụ về cách dùng stack:

```
#include <iostream>
#include <stack>

using namespace std;

int main() {
    stack<int> st;
    for (int i = 1; i <= 5; i++)
        st.push(i);
    // st = 1 2 3 4 5
    cout << st.top() << "\n"; // 5
    st.pop();
}
```

```

cout << st.top() << "\n"; // 4

while (!st.empty()) {
    cout << st.top() << " ";
    st.pop();
}
// 4 3 2 1
}

```

### *f. Queue*

Tương tự như stack, nhưng queue tổ chức theo cấu trúc FIFO (First In First Out), đó gọi là cấu trúc hàng đợi. Các phương thức quan trọng trong queue là:

- `empty()`: Giống như stack hàm này cũng dùng để kiểm tra liệu queue có rỗng hay không.
- `push(value)`: Đẩy giá trị value vào hàng đợi.
- `pop()`: Lấy giá trị ra khỏi hàng đợi.
- `front()`: Lấy giá trị đang chờ ở đỉnh trong hàng đợi.

Ví dụ:

```

// queue::push/pop
#include <iostream>          // std::cin, std::cout
#include <queue>             // std::queue

int main ()
{
    std::queue<int> myqueue;
    int myint;

    std::cout << "Please enter some integers (enter 0 to end):\n";

    do {
        std::cin >> myint;
        myqueue.push (myint);
    } while (myint);

    std::cout << "myqueue contains: ";
    while (!myqueue.empty())
    {
        std::cout << ' ' << myqueue.front();
        myqueue.pop();
    }
    std::cout << '\n';

    return 0;
}

```



*g. Priority queue (Hàng đợi ưu tiên)*

Priority queue là một loại container adaptor, được thiết kế đặc biệt để phần tử ở đầu luôn luôn lớn nhất (theo một quy ước về độ ưu tiên nào đó) so với các phần tử khác. Priority Queue rất giống với heap, cho nên nó thường được dùng thay cho heap. Phép toán mặc định là less, có nghĩa là phần tử lớn nhất sẽ ở đầu. Tuy nhiên để linh hoạt ta có thể viết hàm so sánh (ở bài sau) để chủ động trong việc sử dụng priority queue. Khai báo:

```
priority_queue<int> pq; // Dùng phép toán mặc định less
priority_queue<int, vector<int>, greater<int> > pq; // Phép toán greater
```

Các phương thức quan trọng:

- empty(): Kiểm tra pq có rỗng hay không.
- push(value): Thêm phần tử value vào pq.
- pop(): Loại bỏ phần tử ở đỉnh pq (phần tử có ưu tiên cao nhất).
- top(): Trả về phần tử ở đỉnh pq (phần tử có ưu tiên cao nhất).

Ví dụ:

```
// priority_queue::push/pop
#include <iostream>          // std::cout
#include <queue>              // std::priority_queue

int main ()
{
    std::priority_queue<int> mypq;

    mypq.push(30);
    mypq.push(100);
    mypq.push(25);
    mypq.push(40);

    std::cout << "Popping out elements...";
    while (!mypq.empty())
    {
        std::cout << ' ' << mypq.top();
        mypq.pop();
    }
    std::cout << '\n';

    return 0;
}
```

## 5. STL Algorithm (Thư viện thuật toán).

Thư viện này chứa khá nhiều thuật toán hay và quan trọng. Ta sẽ chỉ xem xét ngắn gọn một số thuật toán hữu ích trong việc lập trình giải các bài toán trong thực tế.

### ***a. Min/Max***

Min/Max sẽ trả về giá trị bé/lớn hơn khi so sánh hai đối tượng cùng kiểu. ví dụ:

`min('a', 'b') = 'a';`

`min(2, 3) = 2`

### ***b. prev\_permutation/next\_permutation***

`next_permutation` trả về kết quả là `true` nếu có hoán vị tiếp theo, ngược lại thì kết quả là `false`. Hàm `prev_permutation` thì trả về `true` nếu hoán vị đang xét không phải là hoán vị đầu tiên của dãy các hoán vị được xếp theo thứ tự từ điển. Ví dụ:

```
#include <iostream>
#include <cstdio>
#include <algorithm>

using namespace std;

int main() {
    int myint[] = {1, 2, 3};
    do {
        cout << myint[0] << " " << myint[1] << " " << myint[2] << "\n";
    } while (next_permutation(myint, myint + 3));

    vector<int> a;
    a.push_back(3);
    a.push_back(2);
    a.push_back(1);
    do {
        cout << a[0] << " " << a[1] << " " << a[2] << "\n";
    } while (prev_permutation(a.begin(), a.end()));
}
```

Output:

```

C:\Users\Administrator\Desktop\workspace\cppbook\sourcecode\myproject\bin\Debug\myproject.exe
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
3 2 1
3 1 2
2 3 1
2 1 3
1 3 2
1 2 3

Process returned 0 (0x0)   execution time : 0.020 s
Press any key to continue.

```

### *c. sort*

Hàm sort cho phép ta sắp xếp các phần tử theo một thứ tự ưu tiên nào đó với độ phức tạp là  $O(N \log N)$ . Ví dụ:

```

#include <iostream>
#include <cstdio>
#include <algorithm>

using namespace std;

int main() {
    int myint[5] = {1, 3, 2, 5, 4};

    // sử dụng toán tử mặc định less
    // => sắp xếp tăng dần
    sort(myint, myint + 5);
    for (int i = 0; i < 5; i++)
        cout << myint[i] << " "; // 1 2 3 4 5
    cout << "\n";

    // sử dụng toán tử greater
    // => sắp xếp giảm dần
    sort(myint, myint + 5, greater<int>());
    for (int i = 0; i < 5; i++)
        cout << myint[i] << " "; // 5 4 3 2 1
    cout << "\n";
}

```

*d. binary\_search*

Tìm kiếm nhị phân xem khóa có trong đoạn cần tìm không với độ phức tạp  $O(N \log N)$ . Đoạn tìm kiếm phải là đoạn có thứ tự (đã sắp xếp). Nếu có thì trả về true, ngược lại là false. Ví dụ:

```
// binary_search example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool myfunction (int i,int j) { return (i<j); }

int main () {
    int myints[] = {1,2,3,4,5,4,3,2,1};
    vector<int> v(myints,myints+9); // 1 2 3 4 5 4 3 2 1

    // Sử dụng toán tử so sánh mặc định
    sort (v.begin(), v.end());

    cout << "looking for a 3... ";

    if (binary_search (v.begin(), v.end(), 3))
        cout << "found!\n"; else cout << "not found.\n";

    // sử dụng hàm so sánh tự định nghĩa:
    sort (v.begin(), v.end(), myfunction);

    cout << "looking for a 6... ";

    if (binary_search (v.begin(), v.end(), 6, myfunction))
        cout << "found!\n"; else cout << "not found.\n";

    return 0;
}
```

*e. lower\_bound và upper\_bound*

Hàm lower\_bound và upper\_bound tương đối giống nhau. Hàm lower\_bound sẽ trả về phần tử đầu tiên lớn hơn hoặc bằng khóa cho trước. Còn upper\_bound thì trả về phần tử đầu tiên lớn hơn hẳn khóa cho trước. Cả hai hàm này đều có độ phức tạp  $O(N \log N)$ . Ví dụ:

```
// lower_bound/upper_bound example
#include <iostream>
#include <algorithm>
#include <vector>
```

```

using namespace std;

int main () {
    int myints[] = {10,20,30,30,20,10,10,20};
    vector<int> v(myints,myints+8); // 10 20 30 30 20 10 10 20
    vector<int>::iterator low,up;

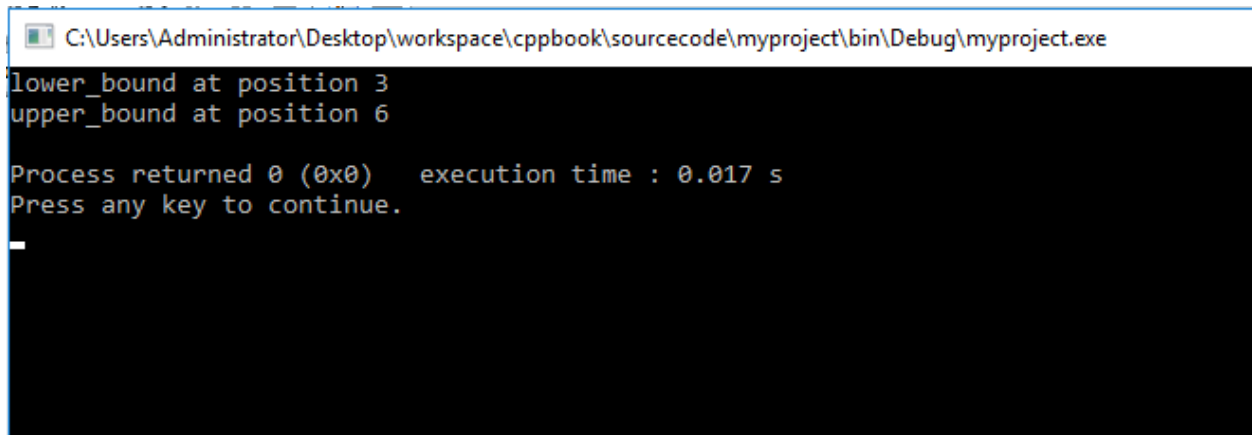
    sort (v.begin(), v.end()); // 10 10 10 20 20 20 30 30
    low = lower_bound (v.begin(), v.end(), 20); // ^
    up = upper_bound (v.begin(), v.end(), 20); // ^

    cout << "lower_bound at position " << int(low- v.begin()) << endl;
    cout << "upper_bound at position " << int(up - v.begin()) << endl;

    return 0;
}

```

Output:



```

C:\Users\Administrator\Desktop\workspace\cppbook\sourcecode\myproject\bin\Debug\myproject.exe
lower_bound at position 3
upper_bound at position 6

Process returned 0 (0x0)   execution time : 0.017 s
Press any key to continue.
_

```

## 6.Kết

STL là một thư viện cực kì lớn, được xây dựng trong một thời gian dài, do đó việc nắm vững STL trong ngày một, ngày hai là đều không thể. Cho nên ở trên đây, tôi đã nêu ra những cái cơ bản và ngắn gọn nhất của STL, phần còn lại là do người học tự tìm tòi và nghiên cứu, kiến thức sẽ được tích lũy trong một thời gian dài.

## §3: COMPARISION FUNCTION

Ở bài trước ta đã xét đến một số hàm thường dùng trong STL có dùng đến thứ tự ưu tiên như là `sort`, `lower_bound`, `upper_bound`, ... Tất cả những hàm đó xác định tính ưu tiên dựa trên một hàm so sánh mặc định, tuy vậy trong nhiều trường hợp ta không thể hoặc không muốn dùng hàm so sánh mặc định thì ta vẫn có thể viết cho mình một hàm so sánh riêng. Ở bài này, ta sẽ tìm hiểu những cách viết các hàm đó.

### 1. Định nghĩa toán tử “bé hơn” (less operator)

Toán tử bé hơn “<” trong C++ đã được mặc định sẵn với những kiểu dữ liệu và đối tượng (object) tiêu chuẩn. Tuy nhiên, khi chúng ta sử dụng các đối tượng tự định nghĩa như `struct`, `class`, ... thì toán tử “<” không hề có sẵn trong trường hợp này. Do đó ta phải định nghĩa lại toán tử “<” cho đối tượng mà ta đã tạo ra. Ví dụ ta có một `struct` biểu diễn một cạnh trong đồ thị như sau:

```
struct Edge {
    int from, to, weight;
};
```

Giả sử vấn đề là ta có danh sách cạnh và muốn sắp xếp chúng giảm dần để thực hiện thuật toán Kruskal. Và như đã biết thuật toán `sort` trong thư viện `algorithm` có chiều sắp xếp mặc định là tăng dần, tức là sử dụng hàm `less` trong thư viện `functional`. Hàm này có dạng  $f(x, y) = \text{true}$  nếu  $x < y$ , ngược lại thì `false`. Do đó với đối tượng `Edge` ta phải định nghĩa một cạnh như thế nào gọi là bé hơn cạnh kia. Ta có thể làm như sau:

```
struct Edge {
    int from, to, weight;
    bool operator < (Edge other) const {
        return weight < other.weight;
    }
};
```

Khi đó, chương trình sẽ hiểu khi so sánh với hai đối tượng `Edge` `a` và `b`:  $a < b = \text{true}$  nếu  $a.\text{weight} < b.\text{weight}$ . Sau đó chỉ cần viết hàm `sort` một cách rất bình thường như khi sắp xếp tăng dần một dãy số. Ví dụ:

```
struct Edge {
    int from, to, weight;
    bool operator < (Edge other) const {
        return weight < other.weight;
    }
} e[N];
...
sort(e, e + N);
```

Thật ra cách định nghĩa toán tử này là một cách tổng quát, ta có thể định nghĩa các toán tử khác với ý nghĩa theo ý muốn của chúng ta như: +, -, \*, /, ... cho một đối tượng. Tuy nhiên, ở đây tôi trình bày cách định nghĩa toán tử "<" vì đây là toán tử so sánh mặc định cho đa số các hàm hay containers trong STL. Chẳng hạn ta có thể viết:

```
struct Edge {
    int from, to, weight;
    bool operator > (Edge other) const {
        return weight < other.weight; // a lớn hơn b nếu a.weight <
b.weight
    }
    int operator + (Edge other) const {
        return weight - other.weight; // a + b = a.weight - b.weight
    }
    int operator / (Edge other) const {
        return weight * other.weight; // a / b = a.weight * b.weight
    }
}
```

## 2. Định nghĩa một hàm so sánh của riêng mình

Trong nhiều trường hợp, ta sử dụng những kiểu dữ liệu có sẵn trong thư viện. Tuy nhiên ta lại không muốn sử dụng hàm so sánh mặc định của kiểu dữ liệu đó. Mặt khác ta cũng không thể tự ý thay đổi cơ chế mặc so sánh mặc định của một kiểu dữ liệu có sẵn như kiểu dữ liệu tự định nghĩa ở trên. Cho nên ta cần phải viết một hàm so sánh của riêng mình với mục đích chuyên biệt. Hàm này sẽ có dạng:

```
bool cmp(T a, T b)
```

Trong đó, T là một kiểu dữ liệu. Hàm cmp(a, b) sẽ trả về true nếu a ưu tiên cao hơn b vào ngược lại. Ví dụ với đối tượng Edge ở trên nếu không muốn định nghĩa toán tử "<" thì ta có thể viết một hàm cmp như sau:

```
struct Edge {
    int from, to, weight;
} e[N];
...
bool cmp(Edge a, Edge b) {
    return a.weight < b.weight;
}
int main() {
    ...
```

```
        sort(e, e + N, cmp);
    }
```

### 3. Functor

Trong C++ có những kiểu lập trình khá “lạ”, trong đó có Functor hay Function Object. Functor được thiết kế để có thể dùng như một hàm và cũng có thể dùng như một đối tượng (bản chất nó là một object trong lập trình hướng đối tượng). Vì liên quan đến lập trình hướng đối tượng nên ta sẽ không đi sâu vào functor mà chỉ tìm hiểu cách dùng nó để viết một hàm so sánh.

Trong các containers có cấu trúc liên kết như map hay set thì thứ tự rất quan trọng. Vì các đối tượng trong cùng một tập đối tượng luôn được sắp xếp theo một thứ tự mặc định. Nhưng không phải lúc nào thứ tự này cũng đáp ứng như cầu của chúng ta. Vì vậy, functor là một giải pháp giúp ta tùy chỉnh thứ tự ưu tiên của các khóa trong một tập đối tượng. Một functor dùng với mục đích xác định thứ tự ưu tiên có thể viết như sau:

```
struct cmp {
    bool operator()(int a, int b) {
        return a > b;
    }
}
```

Khi đó ta có thể tùy chỉnh các containers theo thứ tự mà mình mong muốn. Ví dụ:

```
set<int, cmp> s; // tập hợp s với các giá trị được xếp giảm dần
priority_queue<int, vector<int>, cmp> pq; // hàng đợi ưu tiên với phần tử
có ưu tiên cao nhất là phần tử bé nhất
map<int, char, cmp> m; // map m với khóa int được sắp xếp giảm dần
```

Hoặc có thể dùng với các hàm `binary_search`, `lower_bound`, `upper_bound`, ... Ví dụ:

```
sort(v.begin(), v.end(), cmp);
binary_search(v.begin(), v.end(), 6, cmp); // tìm xem có phần tử nào lớn
hơn 6 trong v hay không
```

Ngoài ra, C++ cũng cung cấp sẵn một số function như `less<T>`, `greater<T>`, ... Trong đó, như đã nói `less<T>` làm hàm so sánh mặc định của đa số các containers hay hàm.



## §4: CHỈ THỊ TIỀN XỬ LÝ TRONG C++

Tác giả: PHẠM HOÀI NGUYỄN – STUDIO.VN

### 1. Chỉ thị tiền xử lý là gì ?

Chỉ thị tiền xử lý (preprocessor directives) là những chỉ thị cung cấp cho trình biên dịch để xử lý những thông tin trước khi bắt đầu biên dịch thật sự. Tất cả các chỉ thị tiền xử lý đều bắt đầu với “#” và các chỉ thị tiền xử lý không phải là lệnh trong C++ cho nên không kết thúc bằng dấu “;”. Ta thường có ba nhóm chỉ thị tiền xử lý chính đó là:

- Chỉ thị khai báo header file (`#include`).
- Chỉ thị định nghĩa cho tên (`#define marco`).
- Chỉ thị có điều kiện (`#if`, `#else`, `#elif`, `#endif`).

### 2. Chỉ thị khai báo header file

Có duy nhất một chỉ thị để khai báo header file đó là `#include`. Chỉ thị này cho phép ta chèn một file khác vào file chúng ta đang làm việc. Có 2 cú pháp để thực hiện chỉ thị này:

**`#include <file_name>`** ví dụ **`#include <iostream>`**

Khi dùng cú pháp này, bộ tiền xử lý (preprocessor) sẽ tìm `file_name` được cung cấp trong IDE để biên dịch cùng với file bạn đang làm việc, nếu không tìm thấy trình biên dịch sẽ báo lỗi.

**`#include "file_name"`** ví dụ **`#include "iostream"`**

Khi dùng cú pháp này, cách bộ tiền xử lý làm việc có khác đôi chút đó là trước khi tìm xem `file_name` có được cung cấp bởi IDE hay không thì preprocessor sẽ tìm trong máy của chúng ta trước. Nếu kết quả vẫn là hoàn toàn không có thì trình biên dịch sẽ báo lỗi.

### 3. Chỉ thị định nghĩa cho tên (`#define marco`)

Có hai chỉ thị hay dùng trong nhóm này đó là `#define` và `#undef`

Chỉ thị `#define` có cú pháp như sau:

**`#define ten_moi ten_duoc_thay_the`**

Chỉ thị này có tác dụng báo cho trình biên dịch biết rằng `ten_moi` khi biên dịch sẽ được thay thế bằng `ten_duoc_thay_the`. Nếu `ten_duoc_thay_the` quá dài thì ở cuối mỗi dòng ta thêm “\”. Ví dụ:

```
#define ll long long
#define view(x) for (int i = 0; i < x.size(); i++)\
    cout << x[i] << " ";\
```

Từ đó khi biên dịch chương trình cứ hễ bắt gặp “ll” thì trình biên dịch sẽ hiểu là “long long” hay nếu bắt gặp view(x) thì trình biên dịch sẽ hiểu là cả dòng for phía sau. Phạm vi của tên được định nghĩa bởi #define là lúc từ khi nó được định nghĩa cho đến cuối tệp. Có thể dùng #define định nghĩa như tên hàm, một biểu thức, một đoạn chương trình bằng một tên, với cách sử dụng này thì chương trình của chúng ta sẽ ngắn gọn và dễ hiểu hơn. Ví dụ ta có thể thay cả một đoạn code chỉ bằng 1 từ như sau:

```
#define HELLO { printf("Hello STDIO\n"); printf("stdio.vn"); }
void main() {
    bool x = true;
    if(x) HELLO;
}
```

Hay thậm chí là thay cho một hàm:

```
#define SUM(x,y) (x)+(y)
```

Chỉ thị #undef được dùng khi ta cần định nghĩa lại một tên trước đó ta đã định nghĩa, mục đích của #undef là hủy định nghĩa đó để định nghĩa lại bằng #undef. Ví dụ:

```
#define STDIO "Hello STDIO" // Định nghĩa cho tên STDIO là "Hello STDIO"
#undef STDIO // Hủy bỏ định nghĩa cho tên STDIO
#define STDIO "Welcome to STDIO" // Định nghĩa lại cho tên STDIO là "Welcome to STDIO"
```

#### 4. Chỉ thị biên dịch có điều kiện

Ở nhóm này gồm các chỉ thị #if, #elif, #else, #ifdef, #ifndef.

Đầu tiên ta xét các chỉ thị: #if, #elif, #else.

Cú pháp:

```
#if constant-expression_1
// Đoạn chương trình 1
#elif constant-expression_2
// Đoạn chương trình 2
#else
//Đoạn chương trình 3
#endif
```

Nếu constant-expression\_1 true thì chỉ có đoạn chương trình 1 sẽ được biên dịch, trái lại nếu constant-expression\_1 false thì sẽ tiếp tục kiểm tra đến constant-expression\_2. Nếu vẫn chưa đúng thì đoạn chương trình trong chỉ thị #else được

biên dịch. Các constant-expression là biểu thức mà các toán hạng trong đó đều là hằng, các tên đã được định nghĩa bởi các #define cũng được xem là các hằng.

Các chỉ thị #ifdef, #ifndef. Một cách biên dịch có điều kiện khác đó là sử dụng #ifdef và #ifndef, được hiểu như là Nếu đã định nghĩa và Nếu chưa được định nghĩa.

Chỉ thị #ifdef

```
#ifdef identifier
    //Đoạn chương trình 1

#else
    //Đoạn chương trình 2

#endif
```

Nếu identifier đã được định nghĩa thì đoạn chương trình 1 sẽ được thực hiện. Ngược lại nếu identifier chưa được định nghĩa thì đoạn chương trình 2 sẽ được thực hiện.

Chỉ thị #ifndef

```
#ifndef identifier
    //Đoạn chương trình 1

#else
    //Đoạn chương trình 2

#endif
```

# §5: MỘT SỐ KỸ THUẬT TRONG C++11

Tác giả: ALFONSO PETERSSEN – CODEFORCES

Từ trước đến giờ ta đã học về C++ dựa trên nền tảng C++98, nhưng bây giờ khi mà C++ 11 đã được chấp nhận trong các kì thi quốc gia, quốc tế thì việc tìm hiểu về các kỹ thuật mới trong C++11 là một điều hết sức cần thiết. Lưu ý bài viết này chỉ mang tính giới thiệu để người đọc nghiên cứu chứ không nhằm mục đích hướng dẫn chi tiết.

## 1. Từ khóa “auto”

Có thể nói từ khóa “auto” là một từ khóa cực kì hay trong C++11. Nói nôm na là nó có thể tự động nhận diện kiểu dữ liệu của một biến nào đó. Ví dụ trước đây để khai báo và khởi tạo một map với C++98 ta phải viết:

```
map< string, pair< int, int > > someLongTypeName = map< string, pair< int, int > >();
```

Tuy nhiên nếu dùng auto thì mọi chuyện sẽ đơn giản hơn:

```
auto longTypeNamesAreHistory = map< string, pair< int, int > >();
    Hoặc với iterator:

for (map< string, pair< int, int > >::iterator it = m.begin(); it !=
m.end(); ++it) {

    /* do something with it */

}
```

Có thể thay bằng:

```
for (auto it = m.begin(); it != m.end(); ++it) {

    /* do something with it */

}
```

## 2. Khởi tạo một danh sách

Chúng ta dễ dàng khởi tạo các containers dạng danh sách chỉ với cặp dấu "{}":

```
vector< int > primeDigits = {2, 3, 5, 7};

vector< string > days = {"Monday", "Tuesday", "Wednesday", "Thursday",
"Friday", "Saturday", "Sunday"};

pair< int, int > p = {1, 2}; // equivalent to make_pair(1, 2)

map< int, string > numeral = {

    {0, "zero"},

    {1, "one"},

    {2, "two"},

    {3, "three"},

    {4, "four"},

    {5, "five"},

    {6, "six"},

    {7, "seven"},

    {8, "eight"},

    {9, "nine"}

};
```

### 3. Duyệt toàn bộ containers

Phương pháp cũ:

```
for (containers::iterator it = container.begin(); it != container.end(); ++it) {

    cout << *it << endl;

}
```

Tuy nhiên với C++11 thì:

```
for (auto x : container) {

    cout << x << endl;

}
```

Chúng ta thậm chí có thể thay đổi giá trị của một containers, hãy thử đoạn code dưới đây

```
vector< int > numbers = {2, 3, 5, 7};

for (auto& x : numbers)

    x *= 2;

for (auto x : numbers)

    cout << x << endl;
```

### 4. Kiểu mảng mới

C++11 xây dựng một kiểu mảng hẳn hoi là array, nó trong như thế này:

```
auto arr = array< int, 10 >{5, 8, 1, 9, 0, 3, 4, 2, 7, 6}; // array of 10 ints
```

```

sort(arr.begin(), arr.end()); // yes, same as sort(arr, arr + n) for normal
arrays

arr[0] += arr[5]; // normal [] indexing

for (auto i: range(arr.size())) {

    /* do something */

}

auto matrix = array< array<double, 10>, 10>(); // 10 by 10 matrix (not the
same as double[10][10])

```

Thực sự thì nó trông không tốt hơn cách cài đặt mảng truyền thống là mấy, nếu không muốn nói là rườm rà hơn.

## 5. Lambda functions

Trước C++11, chúng ta thường hay sử dụng functor (function object) để tạo một hàm so sánh cho các containers và method trong algorithm. Tuy nhiên, từ C++11 trở đi ta có một dạng function mới đó là Lambda function. Chúng ta cùng xét ví dụ:

```

auto heights = vector< int >{ ...some values here... };

auto order = vector< int >(heights.size());

for (int i: range(heights.size()))

    order[i] = i;

sort(order.begin(), order.end(),

    [&] (const int& a, const& int b) -> bool {

```

```
        return heights[a] < heights[b];  
  
    }  
  
);
```

Hoặc với `for_each`:

```
for_each(v.begin(), v.end(),  
  
    [&] (int x) {  
  
        cout << x << endl;  
  
    }  
  
)
```



## §6: TEMPLATE TRONG C++

Tác giả: NGUYỄN HOÀNG MINH TRUNG – STUDIO.VN

### 1. Template là gì ?

“Template” là từ khóa trong C++, chúng ta có thể hiểu rằng là nó một kiểu dữ liệu trừu tượng, đặc trưng cho các kiểu dữ liệu cơ bản. “Template” là từ khóa báo cho trình biên dịch rằng đoạn mã sau đây định nghĩa cho nhiều kiểu dữ liệu và mã nguồn của nó sẽ được compile sinh ra tương ứng cho từng kiểu dữ liệu trong quá trình biên dịch.

Có 2 loại “template” cơ bản:

- Function template: là một khuôn mẫu hàm, cho phép định nghĩa các hàm tổng quát thao tác cho nhiều kiểu dữ liệu.
- Class template: là một khuôn mẫu lớp, cho phép định nghĩa các lớp tổng quát cho nhiều kiểu dữ liệu.

### 2. Function template

```
void Swap( int &x, int &y)
{
    int Temp = x;
    x = y;
    y = Temp;
}
```

Với đoạn code trên dùng để hoán vị giữa 2 số nguyên, nếu ta cần hoán vị giữa 2 số kiểu float hoặc double,... thì ta lại phải định nghĩa các hàm cho từng loại kiểu dữ liệu như thế. Tôi tự hỏi có cách nào mà có thể định nghĩa một hàm duy nhất mà có thể dùng cho nhiều kiểu dữ liệu hay không, sau một hồi tìm hiểu thì tôi đã có câu trả lời đó là “template”, với từ khóa “template” ta chỉ cần định nghĩa một hàm duy nhất cho các kiểu dữ liệu: int, float, double, ...

```
template <class Type>
void Swap( Type &x, Type &y)
{
    Type Temp = x;
    x = y;
    y = Temp;
}

void main()
{
    int x = 5, y = 10;
    float a = 5.5f, b = 3.0f;
```

```

Swap(x, y);
Swap(a, b);
}

```

Vậy cách thức nó hoạt động ra sao, ta hãy xét hoạt động của trình biên dịch khi gặp lời gọi hàm Swap(x, y) với tham số truyền vào là kiểu int, trước hết khi gặp lời gọi hàm Swap(x, y) thì trình biên dịch tìm xem có hàm Swap() nào đã được khai báo với tham số kiểu int hay chưa, nếu có thì sẽ liên kết với hàm đó, nếu chưa nhưng lại tìm thấy từ khóa “template” với hàm Swap() được truyền vào 2 tham số cùng kiểu với nhau (lúc này là kiểu Type), trình biên dịch chỉ cần kiểm tra xem lời gọi hàm Swap(x, y) có 2 tham số có cùng kiểu dữ liệu với nhau hay không( trong ví dụ trên là 2 tham số kiểu int -> cùng kiểu dữ liệu), nếu cùng kiểu thì trình biên dịch lại kiểm tra xem hàm Swap() với 2 tham số kiểu int đã được sinh ra trước đó hay chưa, nếu có thì lời gọi hàm sẽ liên kết với hàm Swap() đã được sinh ra, nếu chưa thì khi đó trình biên dịch sẽ sinh ra hàm Swap(int &x, int &y), tương tự với Swap(a, b) kiểu float cũng tương tự như vậy. Vì vậy trong quá trình biên dịch sẽ sinh ra 2 hàm Swap() cho 2 loại kiểu dữ liệu trên. Trong ví dụ trên tôi đã chỉ ra cách khai báo hàm mẫu, trong đó:

- “Type” chỉ là một tên riêng thể hiện cho một kiểu dữ liệu tổng quát.
- “class” ta có thể thay thế bằng “typename”, ở đây nó không có sự khác biệt.

Ngoài ra ta có thể dùng prototype cho nguyên mẫu hàm giống như ta làm cho các hàm thông thường

```

template <class T> void Swap( T &x, T &y);

void main()
{
    int x = 3, y = 4;
    float a = 1.2f, b = 2.2f;

    Swap(x, y);
    Swap(a, b);
}

template <class T> void Swap( T &x, T &y)
{
    T z = x;
    x = y;
    y = z;
}

```

Trong ví dụ trên ta chỉ hoán vị giữa 2 tham số cùng kiểu, vậy với khác kiểu thì sao? Ta chỉ cần khai báo thêm một kiểu dữ liệu tổng quát

```

template <class T, class X> void Swap( T &x, X &y);

```

### 3. Overloading Function Templates

Nguyên mẫu hàm (function template) đều có tính chất của một hàm thông thường, nó cho phép chúng ta nạp chồng (overload function).

```
template <class T, class X>
T Sum( T x, X y)
{
    T sum = x + y;
    return sum;
}

template <class T, class X>
X Sum( T x, X y, T z)
{
    X sum = x + y + z;
    return sum;
}
```

Lưu ý: muốn nạp chồng hàm thì các tham số truyền vào ở các hàm phải khác nhau

### 4. Class template

Ta xét ví dụ sau

```
class Point
{
    int x;
    int y;
};
```

Nếu muốn khai báo khuôn mẫu lớp với kiểu tùy ý ta làm như sau

```
template <class Type>
class Point
{
    Type x;
    Type y;
};
```

Lúc này nếu như muốn khai báo một thể hiện template Point: **Point<int> P1;**

Trong ví dụ trên là khai báo Point với 2 thuộc tính cùng kiểu với nhau, chúng ta có thể định nghĩa với 2 thuộc tính có kiểu dữ liệu khác nhau

```
template <class TypeOne, class TypeTwo>
class Point
{
    TypeOne x;
    TypeTwo y;
Public:
```

```

    Point();
};

template <class TypeOne, class TypeTwo>
Point<TypeOne, TypeTwo>::Point()
{
    x = 0;
    y = 0;
}

```

Trong ví dụ trên, khi muốn định nghĩa hàm của một lớp mẫu ở file khác thì ta phải khai báo template với các tham số kiểu trước mỗi hàm ta muốn định nghĩa. None-type template parameters (None-type template parameters) là tham số mẫu mặc định, được xác định và trình biên dịch không sinh ra kiểu khác trong suốt thời gian biên dịch.

```

template <class Type, int N>
class Stack
{
    //do something
};

```

Trong ví dụ trên N là kiểu int đã được xác định rõ từ trước và không thay đổi trong thời gian biên dịch.

## 5. Default type

Xét ví dụ sau

```

template <class Type = int> //defaults to type int
class Point
{
    Type x;
    Type y;
};

Point<> point;

```

Trong ví dụ trên khi khai báo thể hiện của khuôn mẫu lớp là `Point<> point;` do không có kiểu truyền vào, lúc này trình biên dịch sẽ mặc định là kiểu đó là int do lúc đầu đã gán tham số kiểu mặc định là int khi khai báo kiểu trong template. Lưu ý: tham số kiểu mặc định phải nằm ngoài cùng bên phải trong danh sách các tham số kiểu mẫu.

**Tư liệu tham khảo:**

*Sách giáo khoa tin học lớp 11.*

*C++ STL for newbie – Đỗ Xuân Mạnh*

<http://vnoi.info/>

<https://www.wikipedia.org/>

<http://www.vietjack.com/>

<https://www.stdio.vn/>

<http://www.cplusplus.com/>

<http://en.cppreference.com/w/>

