

Bài 1 : Cấu Trúc Của Một Chương Trình C++

Có lẽ một trong những cách tốt nhất để bắt đầu học một ngôn ngữ lập trình là bằng một chương trình. Vậy đây là chương trình đầu tiên của chúng ta :

```
// my first program in C++  
  
#include <iostream.h>  
  
int main ()  
{  
    cout << "Hello World!";  
    return 0;  
}
```

Hello World!

Chương trình trên đây là chương trình đầu tiên mà hầu hết những người học nghề lập trình viết đầu tiên và kết quả của nó là viết câu "Hello, World" lên màn hình. Đây là một trong những chương trình đơn giản nhất có thể viết bằng C++ nhưng nó đã bao gồm những phần cơ bản mà mọi chương trình C++ có. Hãy cùng xem xét từng dòng một :

// my first program in C++

Đây là dòng chú thích. Tất cả các dòng bắt đầu bằng hai dấu sổ (//) được coi là chú thích mà chúng không có bất kì một ảnh hưởng nào đến hoạt động của chương trình. Chúng có thể được các lập trình viên dùng để giải thích hay bình phẩm bên trong mã nguồn của chương trình. Trong trường hợp này, dòng chú thích là một giải thích ngắn gọn những gì mà chương trình chúng ta làm.

#include <iostream.h>

Các câu bắt đầu bằng dấu (#) được dùng cho preprocessor (ai dịch hộ tôi từ này với). Chúng không phải là những dòng mã thực hiện nhưng được dùng để báo hiệu cho trình dịch. Ở đây câu lệnh **#include <iostream.h>** báo cho trình dịch biết cần phải "include" thư viện **iostream**. Đây là một thư viện vào ra cơ bản trong C++ và nó phải được "include" vì nó sẽ được dùng trong chương trình. Đây là cách cổ điển để sử dụng thư viện **iostream**

int main ()

Dòng này tương ứng với phần bắt đầu khai báo hàm **main**. Hàm **main** là điểm mà tất cả các chương trình C++ bắt đầu thực hiện. Nó không phụ thuộc vào vị trí của hàm này (ở đầu, cuối hay ở giữa của mã nguồn) mà nội dung của nó luôn được thực hiện đầu tiên khi chương trình bắt đầu. Thêm vào đó, do nguyên nhân nói trên, mọi chương trình C++ đều phải tồn tại một hàm **main**.

Theo sau **main** là một cặp ngoặc đơn bởi vì nó là một hàm. Trong C++, tất cả các hàm mà sau đó là một cặp ngoặc đơn () thì có nghĩa là nó có thể có hoặc không có tham số (không bắt buộc). Nội dung của hàm main tiếp ngay sau phần khai báo chính thức được bao trong các ngoặc nhọn ({ }) như trong ví dụ của chúng ta

```
cout << "Hello World";
```

Dòng lệnh này làm việc quan trọng nhất của chương trình. **cout** là một dòng (stream) output chuẩn trong C++ được định nghĩa trong thư viện **iostream** và những gì mà dòng lệnh này làm là gửi chuỗi kí tự "Hello World" ra màn hình.

Chú ý rằng dòng này kết thúc bằng dấu chấm phẩy (;). Kí tự này được dùng để kết thúc một lệnh và bắt buộc phải có sau mỗi lệnh trong chương trình C++ của bạn (một trong những lỗi phổ biến nhất của những lập trình viên C++ là quên mất dấu chấm phẩy).

```
return 0;
```

Lệnh **return** kết thúc hàm main và trả về mã đi sau nó, trong trường hợp này là 0. Đây là một kết thúc bình thường của một chương trình không có một lỗi nào trong quá trình thực hiện. Như bạn sẽ thấy trong các ví dụ tiếp theo, đây là một cách phổ biến nhất để kết thúc một chương trình C++.

Chương trình được cấu trúc thành những dòng khác nhau để nó trở nên dễ đọc hơn nhưng hoàn toàn không phải bắt buộc phải làm vậy. Ví dụ, thay vì viết

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

ta có thể viết

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

cũng cho một kết quả chính xác như nhau.

Trong C++, các dòng lệnh được phân cách bằng dấu chấm phẩy (;). Việc chia chương trình thành các dòng chỉ nhằm để cho nó dễ đọc hơn mà thôi.

Các chú thích.

Các chú thích được các lập trình viên sử dụng để ghi chú hay mô tả trong các phần của chương trình. Trong C++ có hai cách để chú thích

```
// Chú thích theo dòng
/* Chú thích theo khối */
```

Chú thích theo dòng bắt đầu từ cặp dấu xô (//) cho đến cuối dòng. Chú thích theo khối bắt đầu bằng /* và kết thúc bằng */ và có thể bao gồm nhiều dòng. Chúng ta sẽ thêm các chú thích cho chương trình :

```
/* my second program in C++
   with more comments */
```

```
Hello World! I'm a C++ program
```

```
#include <iostream.h>

int main ()
{
    cout << "Hello World! ";
    // says Hello World!
    cout << "I'm a C++
program"; // says I'm a C++
program
    return 0;
}
```

Nếu bạn viết các chú thích trong chương trình mà không sử dụng các dấu //, /* hay */, trình dịch sẽ coi chúng như là các lệnh C++ và sẽ hiển thị các lỗi.

Bài 1 : Cấu Trúc Của Một Chương Trình C++

Có lẽ một trong những cách tốt nhất để bắt đầu học một ngôn ngữ lập trình là bằng một chương trình. Vậy đây là chương trình đầu tiên của chúng ta :

```
// my first program in C++

#include <iostream.h>

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

Hello World!

Chương trình trên đây là chương trình đầu tiên mà hầu hết những người học nghề lập trình viết đầu tiên và kết quả của nó là viết câu "Hello, World" lên màn hình. Đây là một trong những chương trình đơn giản nhất có thể viết bằng C++ nhưng nó đã bao gồm những phần cơ bản mà mọi chương trình C++ có. Hãy cùng xem xét từng dòng một :

// my first program in C++

Đây là dòng chú thích. Tất cả các dòng bắt đầu bằng hai dấu sổ (//) được coi là chú thích mà chúng không có bất kì một ảnh hưởng nào đến hoạt động của chương trình. Chúng có thể được các lập trình viên dùng để giải thích hay bình phẩm bên trong mã nguồn của chương trình. Trong trường hợp này, dòng chú thích là một giải thích ngắn gọn những gì mà chương trình chúng ta làm.

#include <iostream.h>

Các câu bắt đầu bằng dấu (#) được dùng cho preprocessor (ai dịch hộ tôi từ này với). Chúng không phải là những dòng mã thực hiện nhưng được dùng để báo hiệu cho trình dịch. Ở đây câu lệnh **#include <iostream.h>** báo cho trình dịch biết cần phải

"include" thư viện **iostream**. Đây là một thư viện vào ra cơ bản trong C++ và nó phải được "include" vì nó sẽ được dùng trong chương trình. Đây là cách cổ điển để sử dụng thư viện **iostream**

```
int main ()
```

Dòng này tương ứng với phần bắt đầu khai báo hàm **main**. Hàm **main** là điểm mà tất cả các chương trình C++ bắt đầu thực hiện. Nó không phụ thuộc vào vị trí của hàm này (ở đầu, cuối hay ở giữa của mã nguồn) mà nội dung của nó luôn được thực hiện đầu tiên khi chương trình bắt đầu. Thêm vào đó, do nguyên nhân nói trên, mọi chương trình C++ đều phải tồn tại một hàm **main**.

Theo sau **main** là một cặp ngoặc đơn bởi vì nó là một hàm. Trong C++, tất cả các hàm mà sau đó là một cặp ngoặc đơn () thì có nghĩa là nó có thể có hoặc không có tham số (không bắt buộc). Nội dung của hàm main tiếp ngay sau phần khai báo chính thức được bao trong các ngoặc nhọn ({ }) như trong ví dụ của chúng ta

```
cout << "Hello World";
```

Dòng lệnh này làm việc quan trọng nhất của chương trình. **cout** là một dòng (stream) output chuẩn trong C++ được định nghĩa trong thư viện **iostream** và những gì mà dòng lệnh này làm là gửi chuỗi kí tự "Hello World" ra màn hình.

Chú ý rằng dòng này kết thúc bằng dấu chấm phẩy (;). Kí tự này được dùng để kết thúc một lệnh và bắt buộc phải có sau mỗi lệnh trong chương trình C++ của bạn (một trong những lỗi phổ biến nhất của những lập trình viên C++ là quên mất dấu chấm phẩy).

```
return 0;
```

Lệnh **return** kết thúc hàm main và trả về mã đi sau nó, trong trường hợp này là 0. Đây là một kết thúc bình thường của một chương trình không có một lỗi nào trong quá trình thực hiện. Như bạn sẽ thấy trong các ví dụ tiếp theo, đây là một cách phổ biến nhất để kết thúc một chương trình C++.

Chương trình được cấu trúc thành những dòng khác nhau để nó trở nên dễ đọc hơn nhưng hoàn toàn không phải bắt buộc phải làm vậy. Ví dụ, thay vì viết

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

ta có thể viết

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

cũng cho một kết quả chính xác như nhau.

Trong C++, các dòng lệnh được phân cách bằng dấu chấm phẩy (;). Việc chia chương trình thành các dòng chỉ nhằm để cho nó dễ đọc hơn mà thôi.

Các chú thích.

Các chú thích được các lập trình viên sử dụng để ghi chú hay mô tả trong các phần của chương trình. Trong C++ có hai cách để chú thích

```
// Chú thích theo dòng
/* Chú thích theo khối */
```

Chú thích theo dòng bắt đầu từ cặp dấu xô (//) cho đến cuối dòng. Chú thích theo khối bắt đầu bằng /* và kết thúc bằng */ và có thể bao gồm nhiều dòng. Chúng ta sẽ thêm các chú thích cho chương trình :

```
/* my second program in C++
   with more comments */

#include <iostream.h>

int main ()
{
    cout << "Hello World! ";
    // says Hello World!
    cout << "I'm a C++
program"; // says I'm a C++
program
    return 0;
}
```

```
Hello World! I'm a C++ program
```

Nếu bạn viết các chú thích trong chương trình mà không sử dụng các dấu //, /* hay */, trình dịch sẽ coi chúng như là các lệnh C++ và sẽ hiển thị các lỗi.

Bài 3 : Các Toán Tử

Qua bài trước chúng ta đã biết đến sự tồn tại của các biến và các hằng. Trong C++, để thao tác với chúng ta sử dụng các toán tử, đó là các từ khoá và các dấu không có trong bảng chữ cái nhưng lại có trên hầu hết các bàn phím trên thế giới. Hiểu biết về chúng là rất quan trọng vì đây là một trong những thành phần cơ bản của ngôn ngữ C++.

Toán tử gán (=).

Toán tử gán dùng để gán một giá trị nào đó cho một biến

```
a = 5;
```

gán giá trị nguyên 5 cho biến **a**. Vế trái bắt buộc phải là một biến còn vế phải có thể là bất kì hằng, biến hay kết quả của một biểu thức.

Cần phải nhấn mạnh rằng toán tử gán luôn được thực hiện từ trái sang phải và không bao giờ đảo ngược

```
a = b;
```

gán giá trị của biến **a** bằng giá trị đang chứa trong biến **b**. Chú ý rằng chúng ta chỉ gán **giá trị** của **b** cho **a** và sự thay đổi của **b** sau đó sẽ không ảnh hưởng đến giá trị của **a**.

Một thuộc tính của toán tử gán trong C++ góp phần giúp nó vượt lên các ngôn ngữ lập trình khác là việc cho phép về phải có thể chứa các phép gán khác. Ví dụ:

```
a = 2 + (b = 5);
```

tương đương với

```
b = 5;  
a = 2 + b;
```

Vì vậy biểu thức sau cũng hợp lệ trong C++

```
a = b = c = 5;
```

gán giá trị 5 cho cả ba biến **a**, **b** và **c**

Các toán tử số học (+, -, *, /, %)

Năm toán tử số học được hỗ trợ bởi ngôn ngữ là:

- + cộng
- trừ
- * nhân
- / chia
- % lấy phần dư (trong phép chia)

Thứ tự thực hiện các toán tử này cũng giống như chúng được thực hiện trong toán học. Điều duy nhất có vẻ hơi lạ đối với bạn là phép lấy phần dư, ký hiệu bằng dấu phần trăm (%). Đây chính là phép toán lấy phần dư trong phép chia hai số nguyên với nhau. Ví dụ, nếu **a = 11 % 3**;, biến **a** sẽ mang giá trị 2 vì $11 = 3*3 + 2$.

Các toán tử gán phức hợp (+=, -=, *=, /=, %=>, <=>, &=>, ^=, |=)

Một đặc tính của ngôn ngữ C++ làm cho nó nổi tiếng là một ngôn ngữ súc tích chính là các toán tử gán phức hợp cho phép chỉnh sửa giá trị của một biến với một trong những toán tử cơ bản sau:

```

value += increase; tương đương với value = value +
increase;
a -= 5; tương đương với a = a - 5;
a /= b; tương đương với a = a / b;
price *= units + 1; tương đương với price = price *
(units + 1);

```

và tương tự cho tất cả các toán tử khác.

Tăng và giảm.

Một ví dụ khác của việc tiết kiệm khi viết mã lệnh là toán tử tăng (++) và giảm (--). Chúng tăng hoặc giảm giá trị chứa trong một biến đi 1. Chúng tương đương với +=1 hoặc -=1. Vì vậy, các dòng sau là tương đương:

```

a++;
a+=1;
a=a+1;

```

Một tính chất của toán tử này là nó có thể là *tiền tố* hoặc *hậu tố*, có nghĩa là có thể viết trước tên biến (++a) hoặc sau (a++) và mặc dù trong hai biểu thức rất đơn giản đó nó có cùng ý nghĩa nhưng trong các thao tác khác khi mà kết quả của việc tăng hay giảm được sử dụng trong một biểu thức thì chúng có thể có một khác biệt quan trọng về ý nghĩa: Trong trường hợp toán tử được sử dụng như là một tiền tố (++a) giá trị được tăng trước khi biểu thức được tính và giá trị đã tăng được sử dụng trong biểu thức; trong trường hợp ngược lại (a++) giá trị trong biến a được tăng sau khi đã tính toán. Hãy chú ý sự khác biệt :

Ví dụ 1

```

B=3;
A=++B;
// A is 4, B is 4

```

Ví dụ 2

```

B=3;
A=B++;
// A is 3, B is 4

```

Các toán tử quan hệ (==, !=, >, <, >=, <=)

Để có thể so sánh hai biểu thức với nhau chúng ta có thể sử dụng các toán tử quan hệ. Theo chuẩn ANSI-C++ thì giá trị của thao tác quan hệ chỉ có thể là giá trị logic - chúng chỉ có thể có giá trị **true** hoặc **false**, tùy theo biểu thức kết quả là đúng hay sai.

Sau đây là các toán tử quan hệ bạn có thể sử dụng trong C++

`==` Bằng
`!=` Khác
`>` Lớn hơn
`<` Nhỏ hơn
`>=` Lớn hơn hoặc bằng
`<=` Nhỏ hơn hoặc bằng

Ví dụ:

`(7 == 5)` sẽ trả giá trị **false**

`(6 >= 6)` sẽ trả giá trị **true**

tất nhiên thay vì sử dụng các số, chúng ta có thể sử dụng bất cứ biểu thức nào. Cho `a=2`, `b=3` và `c=6`

`(a*b >= c)` sẽ trả giá trị **true**.

`(b+4 < a*c)` sẽ trả giá trị **false**

Cần chú ý rằng `=` (một dấu bằng) là hoàn toàn khác với `==` (hai dấu bằng). Dấu đầu tiên là một toán tử gán (gán giá trị của biểu thức bên phải cho biến ở bên trái) và dấu còn lại (`==`) là một toán tử quan hệ nhằm so sánh xem hai biểu thức có bằng nhau hay không.

Trong nhiều trình dịch có trước chuẩn ANSI-C++ cũng như trong ngôn ngữ C, các toán tử quan hệ không trả về giá trị logic **true** hoặc **false** mà trả về giá trị **int** với **0** tương ứng với **false** còn giá trị khác 0 (thường là 1) thì tương ứng với **true**.

Các toán tử logic (`!`, `&&`, `||`).

Toán tử `!` tương đương với toán tử logic NOT, nó chỉ có một đối số ở phía bên phải và việc duy nhất mà nó làm là đổi ngược giá trị của đối số từ **true** sang **false** hoặc ngược lại. Ví dụ:

`!(5 == 5)` trả về **false** vì biểu thức bên phải `(5 == 5)` có giá trị **true**.

`!(6 <= 4)` trả về **true** vì `(6 <= 4)` có giá trị **false**.

!true trả về **false**.
!false trả về **true**.

Toán tử logic **&&** và **||** được sử dụng khi tính toán hai biểu thức để lấy ra một kết quả duy nhất. Chúng tương ứng với các toán tử logic *AND* và *OR*. Kết quả của chúng phụ thuộc vào mối quan hệ của hai đối số:

Đối số thứ nhất a	Đối số thứ hai b	Kết quả a && b	Kết quả a b
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Ví dụ:

(**(5 == 5) && (3 > 6)**) trả về **false** (*true && false*).
(**(5 == 5) || (3 > 6)**) trả về **true** (*true || false*).

Toán tử điều kiện (**?**).

Toán tử điều kiện tính toán một biểu thức và trả về một giá trị khác tùy thuộc vào biểu thức đó là đúng hay sai. Cấu trúc của nó như sau:

condition ? result1 : result2

Nếu *condition* là **true** thì giá trị trả về sẽ là *result1*, nếu không giá trị trả về là *result2*.

7==5 ? 4 : 3 trả về **3** vì 7 không bằng 5.
7==5+2 ? 4 : 3 trả về **4** vì 7 bằng 5+2.
5>3 ? a : b trả về **a**, vì 5 lớn hơn 3.
a>b ? a : b trả về giá trị lớn hơn, **a** hoặc **b**.

Các toán tử thao tác bit (**&, |, ^, ~, <<, >>**).

Các toán tử thao tác bit thay đổi các bit biểu diễn một biến, có nghĩa là thay đổi biểu diễn nhị phân của chúng

toán tử	asm	Mô tả
---------	-----	-------

&	AND	Logical AND
	OR	Logical OR
^	XOR	Logical exclusive OR
~	NOT	Đảo ngược bit
<<	SHL	Dịch bit sang trái
>>	SHR	Dịch bit sang phải

Các toán tử chuyển đổi kiểu

Các toán tử chuyển đổi kiểu cho phép bạn chuyển đổi dữ liệu từ kiểu này sang kiểu khác. Có vài cách để làm việc này trong C++, cách cơ bản nhất được thừa kế từ ngôn ngữ C là đặt trước biểu thức cần chuyển đổi tên kiểu dữ liệu được bọc trong cặp ngoặc đơn **()**, ví dụ:

```
int i;
float f = 3.14;
i = (int) f;
```

Đoạn mã trên chuyển số thập phân 3.14 sang một số nguyên (3). Ở đây, toán tử chuyển đổi kiểu là (int). Một cách khác để làm điều này trong C++ là sử dụng các constructors (ở một số sách thuật ngữ này được dịch là **cấu tử** nhưng tôi thấy nó có vẻ không xuôi tai lắm) thay vì dùng các toán tử: đặt trước biểu thức cần chuyển đổi kiểu tên kiểu mới và bao bọc **biểu thức** giữa một cặp ngoặc đơn.

```
i = int ( f );
```

Cả hai cách chuyển đổi kiểu đều hợp lệ trong C++. Thêm vào đó ANSI-C++ còn có những toán tử chuyển đổi kiểu mới đặc trưng cho lập trình hướng đối tượng.

sizeof()

Toán tử này có một tham số, đó có thể là một kiểu dữ liệu hay là một biến và trả về kích cỡ bằng byte của kiểu hay đối tượng đó.

```
a = sizeof (char);
```

a sẽ mang giá trị 1 vì kiểu **char** luôn có kích cỡ 1 byte trên mọi hệ thống. Giá trị trả về của **sizeof** là một hằng số vì vậy nó luôn luôn được tính trước khi chương trình thực hiện.

Các toán tử khác

Trong C++ còn có một số các toán tử khác, như các toán tử liên quan đến con trỏ hay lập trình hướng đối tượng. Chúng sẽ được nói đến cụ thể trong các phần tương ứng.

Thứ tự ưu tiên của các toán tử

Khi viết các biểu thức phức tạp với nhiều toán hạng các bạn có thể tự hỏi toán hạng nào được tính trước, toán hạng nào được tính sau. Ví dụ như trong biểu thức sau:

$$a = 5 + 7 \% 2$$

có thể có hai cách hiểu sau:

$a = 5 + (7 \% 2)$ với kết quả là **6**, hoặc
 $a = (5 + 7) \% 2$ với kết quả là **0**

Câu trả lời đúng là biểu thức đầu tiên. Vì nguyên nhân nói trên, ngôn ngữ C++ đã thiết lập một thứ tự ưu tiên giữa các toán tử, không chỉ riêng các toán tử số học mà tất cả các toán tử có thể xuất hiện trong C++. Thứ tự ưu tiên của chúng được liệt kê trong bảng sau theo thứ tự từ cao xuống thấp.

Thứ tự	Toán tử	Mô tả	Associativity
1	::	scope	Trái
2	() [] -> . sizeof		Trái
3	++ --	tăng/giảm	Phải
	~	Đảo ngược bit	
	!	NOT	
	& *	Toán tử con trỏ	
	(type)	Chuyển đổi kiểu	
	+ -	Dương hoặc âm	
4	* / %	Toán tử số học	Trái
5	+ -	Toán tử số học	Trái
6	<< >>	Dịch bit	Trái
7	< <= > >=	Toán tử quan hệ	Trái
8	== !=	Toán tử quan hệ	Trái

9	& ^	Toán tử thao tác bit	Trái
10	&&	Toán tử logic	Trái
11	? :	Toán tử điều kiện	Phải
12	= += -= *= /= %= >>= <<= &= ^= =	Toán tử gán	Phải
13	,	Dấu phẩy	Trái

Associativity định nghĩa trong trường hợp có một vài toán tử có cùng thứ tự ưu tiên thì cái nào sẽ được tính trước, toán tử ở phía xa nhất bên phải hay là xa nhất bên trái.

Nếu bạn muốn viết một biểu thức phức tạp mà lại không chắc lắm về thứ tự ưu tiên của các toán tử thì nên sử dụng các ngoặc đơn. Các bạn nên thực hiện điều này vì nó sẽ giúp chương trình dễ đọc hơn.

Bài 4 : Các Cấu Trúc Điều Khiển

Một chương trình thường không chỉ bao gồm các lệnh tuần tự nối tiếp nhau. Trong quá trình chạy nó có thể rẽ nhánh hay lặp lại một đoạn mã nào đó. Để làm điều này chúng ta sử dụng các cấu trúc điều khiển.

Cùng với việc giới thiệu các cấu trúc điều khiển chúng ta cũng sẽ phải biết tới một khái niệm mới: **khối lệnh**, đó là một nhóm các lệnh được ngăn cách bởi dấu chấm phẩy (;) nhưng được gộp trong một khối giới hạn bởi một cặp ngoặc nhọn: { và }.

Hầu hết các cấu trúc điều khiển mà chúng ta sẽ xem xét trong chương này cho phép sử dụng một lệnh đơn hay một khối lệnh làm tham số, tùy thuộc vào chúng ta có đặt nó trong cặp ngoặc nhọn hay không.

Cấu trúc điều kiện: *if* và *else*

Cấu trúc này được dùng khi một lệnh hay một khối lệnh chỉ được thực hiện khi một điều kiện nào đó thỏa mãn. Dạng của nó như sau:

```
if (condition) statement
```

trong đó *condition* là biểu thức sẽ được tính toán. Nếu điều kiện đó là **true**, *statement* được thực hiện. Nếu không *statement* bị bỏ qua (không thực hiện) và chương trình tiếp tục thực hiện lệnh tiếp sau cấu trúc điều kiện.

Ví dụ, đoạn mã sau đây sẽ viết **x is 100** chỉ khi biến **x** chứa giá trị 100:

```
if (x == 100)
    cout << "x is 100";
```

Nếu chúng ta muốn có hơn một lệnh được thực hiện trong trường hợp *condition* là **true** chúng ta có thể chỉ định một *khối lệnh* bằng cách sử dụng một cặp ngoặc nhọn { }:

```
if (x == 100)
{
    cout << "x is ";
    cout << x;
}
```

Chúng ta cũng có thể chỉ định điều gì sẽ xảy ra nếu điều kiện không được thoả mãn bằng cách sử dụng từ khoá *else*. Nó được sử dụng cùng với **if** như sau:

```
if (condition) statement1 else statement2
```

Ví dụ:

```
if (x == 100)
    cout << "x is 100";
else
    cout << "x is not 100";
```

Cấu trúc *if + else* có thể được móc nối để kiểm tra nhiều giá trị. Ví dụ sau đây sẽ kiểm tra xem giá trị chứa trong biến **x** là dương, âm hay bằng không.

```
if (x > 0)
    cout << "x is positive";
else if (x < 0)
    cout << "x is negative";
else
    cout << "x is 0";
```

Các cấu trúc lặp

Mục đích của các vòng lặp là lặp lại một thao tác với một số lần nhất định hoặc trong khi một điều kiện nào đó còn thoả mãn.

Vòng lặp *while* .

Dạng của nó như sau:

```
while (expression) statement
```

và chức năng của nó đơn giản chỉ là lặp lại *statement* khi điều kiện *expression* còn thoả mãn.

Ví dụ, chúng ta sẽ viết một chương trình đếm ngược sử dụng vào lặp *while*:

```
// custom countdown using while
```

```
Enter the starting number > 8
```

```
#include <iostream.h>
int main ()
{
    int n;
    cout << "Enter the starting
number > ";
    cin >> n;
    while (n>0) {
        cout << n << ", ";
        --n;
    }
    cout << "FIRE!";
    return 0;
}
```

```
8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

Khi chương trình chạy người sử dụng được yêu cầu nhập vào một số để đếm ngược. Sau đó, khi vòng lặp *while* bắt đầu nếu số mà người dùng nhập vào thoả mãn điều kiện điều kiện $n > 0$ khối lệnh sẽ được thực hiện một số lần không xác định chừng nào điều kiện ($n > 0$) còn được thoả mãn.

Chúng ta cần phải nhớ rằng vòng lặp phải kết thúc ở một điểm nào đó, vì vậy bên trong vòng lặp chúng ta phải cung cấp một phương thức nào đó để buộc *condition* trở thành sai nếu không thì nó sẽ lặp lại mãi mãi. Trong ví dụ trên vòng lặp phải có lệnh `--n;` để làm cho *condition* trở thành sai sau một số lần lặp.

Vòng lặp *do-while*

Dạng thức:

```
do statement while (condition);
```

Chức năng của nó là hoàn toàn giống vòng lặp *while* chỉ trừ có một điều là điều kiện điều khiển vòng lặp được tính toán sau khi *statement* được thực hiện, vì vậy *statement* sẽ được thực hiện ít nhất một lần ngay cả khi *condition* không bao giờ được thoả mãn. Ví dụ, chương trình dưới đây sẽ viết ra bất kì số nào mà bạn nhập vào cho đến khi bạn nhập số 0.

```
// number echoer
#include <iostream.h>
int main ()
{
    unsigned long n;
    do {
        cout << "Enter number (0 to
end): ";
        cin >> n;
        cout << "You entered: " <<
n << "\n";
```

```
Enter number (0 to end): 12345
You entered: 12345
Enter number (0 to end): 160277
You entered: 160277
Enter number (0 to end): 0
You entered: 0
```

```

    } while (n != 0);
    return 0;
}

```

Vòng lặp *do-while* thường được dùng khi điều kiện để kết thúc vòng lặp nằm trong vòng lặp, như trong ví dụ trên, số mà người dùng nhập vào là điều kiện kiểm tra để kết thúc vòng lặp. Nếu bạn không nhập số 0 trong ví dụ trên thì vòng lặp sẽ không bao giờ chấm dứt.

Vòng lặp *for* .

Dạng thức:

```
for (initialization; condition; increase) statement;
```

và chức năng chính của nó là lặp lại *statement* chừng nào *condition* còn mang giá trị đúng, như trong vòng lặp *while*. Nhưng thêm vào đó, **for** cung cấp chỗ dành cho lệnh khởi tạo và lệnh tăng. Vì vậy vòng lặp này được thiết kế đặc biệt lặp lại một hành động với một số lần xác định.

Cách thức hoạt động của nó như sau:

- 1, *initialization* được thực hiện. Nói chung nó đặt một giá trị ban đầu cho biến điều khiển. Lệnh này được thực hiện chỉ một lần.
- 2, *condition* được kiểm tra, nếu nó là đúng vòng lặp tiếp tục còn nếu không vòng lặp kết thúc và *statement* được bỏ qua.
- 3, *statement* được thực hiện. Nó có thể là một lệnh đơn hoặc là một khối lệnh được bao trong một cặp ngoặc nhọn.
- 4, Cuối cùng, *increase* được thực hiện để tăng biến điều khiển và vòng lặp quay trở lại bước 2.

Sau đây là một ví dụ đếm ngược sử dụng vòng *for*.

```

// countdown using a for loop
#include <iostream.h>
int main ()
{
    for (int n=10; n>0; n--) {
        cout << n << ", ";
    }
    cout << "FIRE!";
    return 0;
}

```

```

10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
FIRE!

```

Phần khởi tạo và lệnh tăng không bắt buộc phải có. Chúng có thể được bỏ qua nhưng vẫn phải có dấu chấm phẩy ngăn cách giữa các phần. Vì vậy, chúng ta có thể viết `for (;n<10;)` hoặc `for (;n<10;n++)`.

Bằng cách sử dụng dấu phẩy, chúng ta có thể dùng nhiều lệnh trong bất kì trường nào trong vòng `for`, như là trong phần khởi tạo. Ví dụ chúng ta có thể khởi tạo một lúc nhiều biến trong vòng lặp:

```
for ( n=0, i=100 ; n!=i ; n++, i-- )
{
    // cái gì ở đây cũng được...
}
```

Vòng lặp này sẽ thực hiện 50 lần nếu như `n` và `i` không bị thay đổi trong thân vòng lặp:

The diagram shows a `for` loop: `for ( n=0, i=100 ; n!=i ; n++, i-- )`. The parts are color-coded and labeled with arrows:
- `n=0, i=100` (red box) is labeled "Initialization".
- `n!=i` (yellow box) is labeled "Condition".
- `n++, i--` (blue box) is labeled "Increase".

Các lệnh rẽ nhánh và lệnh nhảy

Lệnh `break`.

Sử dụng `break` chúng ta có thể thoát khỏi vòng lặp ngay cả khi điều kiện để nó kết thúc chưa được thoả mãn. Lệnh này có thể được dùng để kết thúc một vòng lặp không xác định hay buộc nó phải kết thúc giữa chừng thay vì kết thúc một cách bình thường. Ví dụ, chúng ta sẽ dừng việc đếm ngược trước khi nó kết thúc:

```
// break loop example
#include <iostream.h>
int main ()
{
    int n;
    for (n=10; n>0; n--) {
        cout << n << ", ";
        if (n==3)
        {
            cout << "countdown
aborted!";
            break;
        }
    }
    return 0;
}
```

10, 9, 8, 7, 6, 5, 4, countdown
aborted!

Lệnh `continue`.

Lệnh *continue* làm cho chương trình bỏ qua phần còn lại của vòng lặp và nhảy sang lần lặp tiếp theo. Ví dụ chúng ta sẽ bỏ qua số 5 trong phần đếm ngược:

```
// break loop example
#include <iostream.h>
int main ()
{
    for (int n=10; n>0; n--) {
        if (n==5) continue;
        cout << n << ", ";
    }
    cout << "FIRE!";
    return 0;
}
```

10, 9, 8, 7, 6, 4, 3, 2, 1,
FIRE!

Lệnh *goto*.

Lệnh này cho phép nhảy vô điều kiện tới bất kì điểm nào trong chương trình. Nói chung bạn nên tránh dùng nó trong chương trình C++. Tuy nhiên chúng ta vẫn có một ví dụ dùng lệnh **goto** để đếm ngược:

```
// goto loop example
#include <iostream.h>
int main ()
{
    int n=10;
loop: ;
    cout << n << ", ";
    n--;
    if (n>0) goto loop;
    cout << "FIRE!";
    return 0;
}
```

10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
FIRE!

Hàm *exit*.

Mục đích của *exit* là kết thúc chương trình và trả về một mã xác định. Dạng thức của nó như sau

```
void exit (int exit code);
```

exit code được dùng bởi một số hệ điều hành hoặc có thể được dùng bởi các chương trình gọi. Theo quy ước, mã trả về 0 có nghĩa là chương trình kết thúc bình thường còn các giá trị khác 0 có nghĩa là có lỗi.

Cấu trúc lựa chọn: *switch*.

Cú pháp của lệnh *switch* hơi đặc biệt một chút. Mục đích của nó là kiểm tra một vài giá trị hằng cho một biểu thức, tương tự với những gì chúng ta làm ở đầu bài này khi liên kết một vài lệnh *if* và *else if* với nhau. Dạng thức của nó như sau:

```
switch (expression) {
    case constant1:
        block of instructions 1
        break;
    case constant2:
        block of instructions 2
        break;
    .
    .
    .
    default:
        default block of instructions
}
```

Nó hoạt động theo cách sau: **switch** tính biểu thức và kiểm tra xem nó có bằng *constant1* hay không, nếu đúng thì nó thực hiện *block of instructions 1* cho đến khi tìm thấy từ khoá **break**, sau đó nhảy đến phần cuối của cấu trúc lựa chọn *switch*. Còn nếu không, *switch* sẽ kiểm tra xem biểu thức có bằng *constant2* hay không. Nếu đúng nó sẽ thực hiện *block of instructions 2* cho đến khi tìm thấy từ khoá **break**. Cuối cùng, nếu giá trị biểu thức không bằng bất kì hằng nào được chỉ định ở trên (bạn có thể chỉ định bao nhiêu câu lệnh **case** tùy thích), chương trình sẽ thực hiện các lệnh trong phần **default**: nếu nó tồn tại vì phần này không bắt buộc phải có.

Hai đoạn mã sau là tương đương:

ví dụ switch

```
switch (x) {
    case 1:
        cout << "x is 1";
        break;
    case 2:
        cout << "x is 2";
        break;
    default:
        cout << "value of x
unknown";
}
```

if-else tương đương

```
if (x == 1) {
    cout << "x is 1";
}
else if (x == 2) {
    cout << "x is 2";
}
else {
    cout << "value of x unknown";
}
```

Tôi đã nói ở trên rằng cấu trúc của lệnh **switch** hơi đặc biệt. Chú ý sự tồn tại của lệnh **break** ở cuối mỗi khối lệnh. Điều này là cần thiết vì nếu không thì sau khi thực hiện *block of instructions 1* chương trình sẽ không nhảy đến cuối của lệnh *switch* mà sẽ thực hiện các khối lệnh tiếp theo cho đến khi nó tìm thấy lệnh **break** đầu tiên. Điều này khiến cho việc đặt cặp ngoặc nhọn { } trong mỗi trường hợp là không cần thiết và có thể được dùng khi bạn muốn thực hiện một khối lệnh cho nhiều trường hợp khác nhau, ví dụ:

```
switch (x) {
    case 1:
    case 2:
    case 3:
        cout << "x is 1, 2 or 3";
        break;
    default:
        cout << "x is not 1, 2 nor 3";
}
```

Chú ý rằng lệnh **switch** chỉ có thể được dùng để so sánh một biểu thức với các hằng. Vì vậy chúng ta không thể đặt các biến (**case (n*2) :**) hay các khoảng (**case (1..3) :**) vì chúng không phải là các hằng hợp lệ.

style="BORDER-RIGHT: medium none; PADDING-RIGHT: 0cm; BORDER-TOP: medium none; PADDING-LEFT: 0cm; PADDING-BOTTOM: 0cm; BORDER-LEFT: medium none; PADDING-TOP: 0cm; BORDER-BOTTOM: windowtext 3pt solid">

Nếu bạn cần kiểm tra các khoảng hay nhiều giá trị không phải là hằng số hãy kết hợp các lệnh **if** và **else if**

Bài 5 : Hàm (I)

Hàm là một khối lệnh được thực hiện khi nó được gọi từ một điểm khác của chương trình. Dạng thức của nó như sau:

type name (argument1, argument2, ...) statement

trong đó:

type là kiểu dữ liệu được trả về của hàm

name là tên gọi của hàm.

arguments là các tham số (có nhiều bao nhiêu cũng được tùy theo nhu cầu). Một tham số bao gồm tên kiểu dữ liệu sau đó là tên của tham số giống như khi khai báo biến (ví dụ `int x`) và đóng vai trò bên trong hàm như bất kì biến nào khác. Chúng dùng để truyền tham số cho hàm khi nó được gọi. Các tham số khác nhau được ngăn cách bởi các dấu phẩy.

statement là thân của hàm. Nó có thể là một lệnh đơn hay một khối lệnh.

Dưới đây là ví dụ đầu tiên về hàm:

```
// function example
#include <iostream.h>

int addition (int a, int b)
{
    int r;
    r=a+b;
```

The result is 8

```

    return (r);
}

int main ()
{
    int z;
    z = addition (5,3);
    cout << "The result is " << z;
    return 0;
}

```

Để có thể hiểu được đoạn mã này, trước hết hãy nhớ lại những điều đã nói ở bài đầu tiên: một chương trình C++ luôn bắt đầu thực hiện từ hàm **main**. Vì vậy chúng ta bắt đầu từ đây.

Chúng ta có thể thấy hàm **main** bắt đầu bằng việc khai báo biến **z** kiểu **int**. Ngay sau đó là một lời gọi tới hàm **addition**. Nếu để ý chúng ta sẽ thấy sự tương tự giữa cấu trúc của lời gọi hàm với khai báo của hàm:

```

int addition (int a, int b)
           ↑       ↑
z = addition ( 5   ,   3 );

```

Các tham số có vai trò thật rõ ràng. Bên trong hàm **main** chúng ta gọi hàm **addition** và truyền hai giá trị: 5 và 3 tương ứng với hai tham số **int a** và **int b** được khai báo cho hàm **addition**.

Vào thời điểm hàm được gọi từ **main**, quyền điều khiển được chuyển sang cho hàm **addition**. Giá trị của hai tham số (5 và 3) được copy sang hai biến cục bộ **int a** và **int b** bên trong hàm.

Dòng lệnh sau:

```
return (r);
```

kết thúc hàm **addition**, và trả lại quyền điều khiển cho hàm nào đã gọi nó (**main**) và tiếp tục chương trình ở cái điểm mà nó bị ngắt bởi lời gọi đến **addition**. Nhưng thêm vào đó, giá trị được dùng với lệnh **return (r)** chính là giá trị được trả về của hàm.

```

int addition (int a, int b)
  ↓ 8
z = addition ( 5   ,   3 );

```

Giá trị trả về bởi một hàm chính là giá trị của hàm khi nó được tính toán. Vì vậy biến **z** sẽ có giá trị được trả về bởi **addition (5, 3)**, đó là 8.

Phạm vi hoạt động của các biến [nhắc lại]

Bạn cần nhớ rằng phạm vi hoạt động của các biến khai báo trong một hàm hay bất kì một khối lệnh nào khác chỉ là hàm đó hay khối lệnh đó và không thể sử dụng bên ngoài chúng. Ví dụ, trong chương trình ví dụ trên, bạn không thể sử dụng trực tiếp các biến **a**, **b** hay **x** trong hàm **main** vì chúng là các biến cục bộ của hàm **addition**. Thêm vào đó bạn cũng không thể sử dụng biến **z** trực tiếp bên trong hàm **addition** vì nó là biến cục bộ của hàm **main**.

Tuy nhiên bạn có thể khai báo các biến toàn cục để có thể sử dụng chúng ở bất kì đâu, bên trong hay bên ngoài bất kì hàm nào. Để làm việc này bạn cần khai báo chúng bên ngoài mọi hàm hay các khối lệnh, có nghĩa là ngay trong thân chương trình.

Đây là một ví dụ khác về hàm:

```
// function example
#include <iostream.h>

int subtraction (int a, int b)
{
    int r;
    r=a-b;
    return (r);
}

int main ()
{
    int x=5, y=3, z;
    z = subtraction (7,2);
    cout << "The first result is " <<
    z << '\n';
    cout << "The second result is "
    << subtraction (7,2) << '\n';
    cout << "The third result is " <<
    subtraction (x,y) << '\n';
    z= 4 + subtraction (x,y);
    cout << "The fourth result is "
    << z << '\n';
    return 0;
}
```

```
The first result is 5
The second result is 5
The third result is 2
The fourth result is 6
```

Trong trường hợp này chúng ta tạo ra hàm **subtraction**. Chức năng của hàm này là lấy hiệu của hai tham số rồi trả về kết quả.

Tuy nhiên, nếu phân tích hàm **main** các bạn sẽ thấy chương trình đã vài lần gọi đến hàm **subtraction**. Tôi đã sử dụng vài cách gọi khác nhau để các bạn thấy các cách khác nhau mà một hàm có thể được gọi.

Để có hiểu cặn kẽ ví dụ này bạn cần nhớ rằng một lời gọi đến một hàm có thể hoàn toàn được thay thế bởi giá trị của nó. Ví dụ trong lệnh gọi hàm đầu tiên :

```
z = subtraction (7,2);  
cout << "The first result is " << z;
```

Nếu chúng ta thay lời gọi hàm bằng giá trị của nó (đó là 5), chúng ta sẽ có:

```
z = 5;  
cout << "The first result is " << z;
```

Tương tự như vậy

```
cout << "The second result is " << subtraction (7,2);
```

cũng cho kết quả giống như hai dòng lệnh trên nhưng trong trường hợp này chúng ta gọi hàm `subtraction` trực tiếp như là một tham số của `cout`. Chúng ta cũng có thể viết:

```
cout << "The second result is " << 5;
```

vì 5 là kết quả của `subtraction (7,2)`.

Còn với lệnh

```
cout << "The third result is " << subtraction (x,y);
```

Điều mới mẻ duy nhất ở đây là các tham số của `subtraction` là các biến thay vì các hằng. Điều này là hoàn toàn hợp lệ. Trong trường hợp này giá trị được truyền cho hàm `subtraction` là giá trị của `x` and `y`.

Trường hợp thứ tư cũng hoàn toàn tương tự. Thay vì viết

```
z = 4 + subtraction (x,y);
```

chúng ta có thể viết:

```
z = subtraction (x,y) + 4;
```

cũng hoàn toàn cho kết quả tương đương. Chú ý rằng dấu chấm phẩy được đặt ở cuối biểu thức chứ không cần thiết phải đặt ngay sau lời gọi hàm.

Các hàm không kiểu. Cách sử dụng `void`.

Nếu bạn còn nhớ cú pháp của một lời khai báo hàm:

```
type name ( argument1, argument2 ...) statement
```

bạn sẽ thấy rõ ràng rằng nó bắt đầu với một tên kiểu, đó là kiểu dữ liệu sẽ được hàm trả về bởi lệnh **return**. Nhưng nếu chúng ta không muốn trả về giá trị nào thì sao ?

Hãy tưởng tượng rằng chúng ta muốn tạo ra một hàm chỉ để hiển thị một thông báo lên màn hình. Nó không cần trả về một giá trị nào cả, hơn nữa cũng không cần nhận tham số nào hết. Vì vậy người ta đã nghĩ ra kiểu dữ liệu **void** trong ngôn ngữ C. Hãy xem xét chương trình sau:

```
// void function example
#include <iostream.h>

void dummyfunction (void)
{
    cout << "I'm a function!";
}

int main ()
{
    dummyfunction ();
    return 0;
}
```

I'm a function!

Từ khoá **void** trong phần danh sách tham số có nghĩa là hàm này không nhận một tham số nào. Tuy nhiên trong C++ không cần thiết phải sử dụng **void** để làm điều này. Bạn chỉ đơn giản sử dụng cặp ngoặc đơn () là xong.

Bởi vì hàm của chúng ta không có một tham số nào, vì vậy lời gọi hàm **dummyfunction** sẽ là :

```
dummyfunction ();
```

```
style="BORDER-RIGHT: medium none; PADDING-RIGHT: 0in; BORDER-TOP:
medium none; PADDING-LEFT: 0in; PADDING-BOTTOM: 0in; BORDER-LEFT:
medium none; PADDING-TOP: 0in; BORDER-BOTTOM: windowtext 3pt solid">
```

Hai dấu ngoặc đơn là cần thiết để cho trình dịch hiểu đó là một lời gọi hàm chứ không phải là một tên biến hay bất kì dấu hiệu nào khác.

Bài 6 : Hàm (II)

Truyền tham số theo *tham số giá trị* hay *tham số biến*.

Cho đến nay, trong tất cả các hàm chúng ta đã biết, tất cả các tham số truyền cho hàm đều được truyền theo *giá trị*. Điều này có nghĩa là khi chúng ta gọi hàm với các tham số, những gì chúng ta truyền cho hàm là các *giá trị* chứ không phải bản thân các biến. Ví dụ, giả sử chúng ta gọi hàm **addition** như sau:

```
int x=5, y=3, z;
z = addition ( x , y );
```

Trong trường hợp này khi chúng ta gọi hàm **addition** thì các giá trị 5 and 3 được truyền cho hàm, không phải là bản thân các biến.

```
void duplicate (int& a,int& b,int& c)
               ↑x   ↑y   ↑z
duplicate (   x   ,   y   ,   z   );
```

Đến đây các bạn có thể hỏi tôi: Như vậy thì sao, có ảnh hưởng gì đâu ? Điều đáng nói ở đây là khi các bạn thay đổi giá trị của các biến **a** hay **b** bên trong hàm thì các biến **x** và **y** vẫn không thay đổi vì chúng đâu có được truyền cho hàm chỉ có giá trị của chúng được truyền mà thôi.

Hãy xét trường hợp bạn cần thao tác với một biến ngoài ở bên trong một hàm. Vì vậy bạn sẽ phải truyền tham số dưới dạng tham số biến như ở trong hàm **duplicate** trong ví dụ dưới đây:

```
// passing parameters by reference
#include <iostream.h>

void duplicate (int& a, int& b,
int& c)
{
    a*=2;
    b*=2;
    c*=2;
}

int main ()
{
    int x=1, y=3, z=7;
    duplicate (x, y, z);
    cout << "x=" << x << ", y=" << y
<< ", z=" << z;
    return 0;
}
```

x=2, y=6, z=14

Điều đầu tiên làm bạn chú ý là trong khai báo của **duplicate** theo sau tên kiểu của mỗi tham số đều là dấu và (&), để báo hiệu rằng các tham số này được truyền theo tham số biến chứ không phải tham số giá trị.

Khi truyền tham số dưới dạng tham số biến chúng ta đang truyền bản thân biến đó và bất kì sự thay đổi nào mà chúng ta thực hiện với tham số đó bên trong hàm sẽ ảnh hưởng trực tiếp đến biến đó.


```
int addition (int a, int b)

      ↑5      ↑3
z = addition ( x , y );
```

Trong ví dụ trên, chúng ta đã liên kết **a**, **b** và **c** với các tham số khi gọi hàm (**x**, **y** và **z**) và mọi sự thay đổi với **a** bên trong hàm sẽ ảnh hưởng đến giá trị của **x** và hoàn toàn tương tự với **b** và **y**, **c** và **z**.

Kiểu khai báo tham số theo dạng tham số biến sử dụng *dấu và (&)* chỉ có trong C++. Trong ngôn ngữ C chúng ta phải sử dụng con trỏ để làm việc tương tự như thế.

Truyền tham số dưới dạng tham số biến cho phép một hàm trả về nhiều hơn một giá trị. Ví dụ, đây là một hàm trả về số liền trước và liền sau của tham số đầu tiên.

```
// more than one returning value
#include <iostream.h>

void prevnext (int x, int& prev,
int& next)
{
    prev = x-1;
    next = x+1;
}

int main ()
{
    int x=100, y, z;
    prevnext (x, y, z);
    cout << "Previous=" << y << ",
Next=" << z;
    return 0;
}
```

Previous=99, Next=101

Giá trị mặc định của tham số.

Khi định nghĩa một hàm chúng ta có thể chỉ định những giá trị mặc định sẽ được truyền cho các đối số trong trường hợp chúng bị bỏ qua khi hàm được gọi. Để làm việc này đơn giản chỉ cần gán một giá trị cho đối số khi khai báo hàm. Nếu giá trị của tham số đó vẫn được chỉ định khi gọi hàm thì giá trị mặc định sẽ bị bỏ qua. Ví dụ:

```
// default values in functions
#include <iostream.h>

int divide (int a, int b=2)
{
    int r;
    r=a/b;
}
```

6
5

```

    return (r);
}

int main ()
{
    cout << divide (12);
    cout << endl;
    cout << divide (20,4);
    return 0;
}

```

Nhưng chúng ta thấy trong thân chương trình, có hai lời gọi hàm **divide**. Trong lệnh đầu tiên:

divide (12)

chúng ta chỉ dùng một tham số nhưng hàm **divide** cho phép đến hai. Bởi vậy hàm **divide** sẽ tự cho tham số thứ hai giá trị bằng 2 vì đó là giá trị mặc định của nó (chú ý phân khai báo hàm được kết thúc bởi **int b=2**). Vì vậy kết quả sẽ là **6 (12/2)**.

Trong lệnh thứ hai:

divide (20,4)

có hai tham số, bởi vậy giá trị mặc định sẽ được bỏ qua. Kết quả của hàm sẽ là **5 (20/4)**.

Quá tải các hàm.

Hai hàm có thể có cùng tên nếu khai báo tham số của chúng khác nhau, điều này có nghĩa là bạn có thể đặt cùng một tên cho nhiều hàm nếu chúng có số tham số khác nhau hay kiểu dữ liệu của các tham số khác nhau (hay thậm chí là kiểu dữ liệu trả về khác nhau). Ví dụ:

```

// overloaded function
#include <iostream.h>

int divide (int a, int b)
{
    return (a/b);
}

float divide (float a, float b)
{
    return (a/b);
}

int main ()
{
    int x=5,y=2;

```

2
2.5

```
float n=5.0,m=2.0;
cout << divide (x,y);
cout << "\n";
cout << divide (n,m);
return 0;
}
```

Trong ví dụ này chúng ta định nghĩa hai hàm có cùng tên nhưng một hàm dùng hai tham số kiểu `int` và hàm còn lại dùng kiểu `float`. Trình biên dịch sẽ biết cần phải gọi hàm nào bằng cách phân tích kiểu tham số khi hàm được gọi.

Để đơn giản tôi viết cả hai hàm đều có mã lệnh như nhau nhưng điều này không bắt buộc. Bạn có thể xây dựng hai hàm có cùng tên nhưng hoạt động hoàn toàn khác nhau.

Các hàm *inline*.

Chỉ thị *inline* có thể được đặt trước khai báo của một hàm để chỉ rõ rằng lời gọi hàm sẽ được thay thế bằng mã lệnh của hàm khi chương trình được dịch. Việc này tương đương với việc khai báo một macro, lợi ích của nó chỉ thể hiện với các hàm rất ngắn, tốc độ chạy chương trình sẽ được cải thiện vì nó không phải gọi một thủ tục con.

Cấu trúc của nó như sau:

```
inline type name ( arguments ... ) { instructions ... }
```

lời gọi hàm cũng như bất kì một hàm nào khác. Không cần thiết phải đặt từ khoá *inline* trong lệnh gọi, chỉ cần trong lời khai báo hàm là đủ.

Đệ qui.

Các hàm có thể gọi chính nó. Điều này có thể có ích với một số tác vụ như là một số phương pháp sắp xếp hay tính giai thừa của một số. Ví dụ, để tính giai thừa của một số (n), công thức toán học của nó như sau:

$$n! = n * (n-1) * (n-2) * (n-3) \dots * 1$$

và một hàm đệ qui để tính toán sẽ như sau:

```
// factorial calculator
#include <iostream.h>

long factorial (long a)
{
    if (a > 1)
        return (a * factorial (a-1));
    else
        return (1);
}
```

```
Type a number: 9
!9 = 362880
```

```

}

int main ()
{
    long l;
    cout << "Type a number: ";
    cin >> l;
    cout << "!" << l << " = " <<
factorial (l);
    return 0;
}

```

Chú ý trong hàm **factorial** chúng ta có thể lệnh gọi chính nó nhưng chỉ khi tham số lớn hơn 1, nếu không thì hàm sẽ thực hiện một vòng lặp vô hạn vì sau khi đến 0 nó sẽ tiếp tục nhân cả những số âm.

Hàm này có một hạn chế là kiểu dữ liệu mà nó dùng (`long`) không cho phép tính giai thừa quá 12!.

Khai báo mẫu cho hàm.

Cho đến giờ chúng ta hoàn toàn phải định nghĩa hàm trước lệnh gọi đầu tiên đến nó, mà thường là trong **main**, vì vậy hàm **main** luôn phải nằm cuối chương trình. Nếu bạn thử lặp lại một vài ví dụ về hàm trước đây nhưng thử đặt hàm **main** trước bất kì một hàm được gọi từ nó, bạn gần như chắc chắn sẽ nhận được thông báo lỗi. Nguyên nhân là một hàm phải được khai báo trước khi nó được gọi như những gì chúng ta đã làm trong tất cả các ví dụ.

Nhưng có một cách khác để tránh phải viết tất cả mã chương trình trước khi chúng có thể được dùng trong **main** hay bất kì một hàm nào khác. Đó chính là *khai báo mẫu cho hàm*. Cách này bao gồm việc khai báo hàm một cách ngắn gọn nhưng đủ để cho trình dịch có thể biết các tham số và kiểu dữ liệu trả về của hàm.

Dạng của nó như sau:

```
type name ( argument_type1, argument_type2, ... );
```

Đây chính là phần đầu của định nghĩa hàm, ngoại trừ:

- Nó không có bất kì lệnh nào cho hàm. Điều này có nghĩa là nó không bao gồm thân hàm với tất cả các lệnh thường được bọc trong cặp ngoặc nhọn { }.
- Nó kết thúc bằng dấu chấm phẩy (;).
- Trong phần liệt kê các tham số chỉ cần viết kiểu của chúng là đủ. Việc viết tên của các tham số trong phần khai báo mẫu là không bắt buộc.

Ví dụ:

```
// prototyping
#include <iostream.h>

void odd (int a);
void even (int a);

int main ()
{
    int i;
    do {
        cout << "Type a number: (0 to
exit)";
        cin >> i;
        odd (i);
    } while (i!=0);
    return 0;
}

void odd (int a)
{
    if ((a%2)!=0) cout << "Number is
odd.\n";
    else even (a);
}

void even (int a)
{
    if ((a%2)==0) cout << "Number is
even.\n";
    else odd (a);
}
```

```
Type a number (0 to exit): 9
Number is odd.
Type a number (0 to exit): 6
Number is even.
Type a number (0 to exit): 1030
Number is even.
Type a number (0 to exit): 0
Number is even.
```

Ví dụ này rõ ràng không phải là một ví dụ về sự hiệu quả. Tôi chắc chắn rằng các bạn có thể nhận được kết quả như trên chỉ với một nửa số dòng lệnh. Tuy nhiên nó giúp cho chúng ta thấy được việc khai báo mẫu các hàm là như thế nào. Hơn nữa, trong ví dụ này việc khai báo mẫu ít nhất một hàm là bắt buộc.

Đầu tiên chúng ta thấy khai báo mẫu của hai hàm **odd** và **even**:

```
void odd (int a);
void even (int a);
```

cho phép hai hàm này có thể được sử dụng trước khi chúng được định nghĩa hoàn chỉnh. Tuy nhiên lý do đặc biệt giải thích tại sao chương trình này lại cần ít nhất một hàm phải được khai báo mẫu là trong **odd** có một lời gọi đến **even** và trong **even** có một lời gọi đến **odd**. Vì vậy nếu không có hàm nào được khai báo trước thì lỗi chắc chắn sẽ xảy ra.

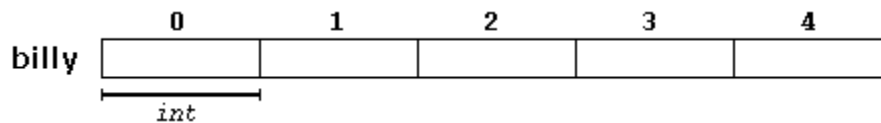
style="BORDER-RIGHT: medium none; PADDING-RIGHT: 0in; BORDER-TOP: medium none; PADDING-LEFT: 0in; PADDING-BOTTOM: 0in; BORDER-LEFT: medium none; PADDING-TOP: 0in; BORDER-BOTTOM: windowtext 3pt solid">

Rất nhiều lập trình viên kinh nghiệm khuyên rằng tất cả các hàm nên được khai báo mẫu. Đó cũng là lời khuyên của tôi, nhất là trong trường hợp có nhiều hàm hoặc chúng rất dài, khi đó việc khai báo tất cả các hàm ở cùng một chỗ cho phép chúng ta biết phải gọi các hàm như thế nào, vì vậy tiết kiệm được thời gian.

Bài 7 : Mảng

Mảng là một dãy các phần tử có cùng kiểu được đặt liên tiếp trong bộ nhớ và có thể truy xuất đến từng phần tử bằng cách thêm một chỉ số vào sau tên của mảng.

Điều này có nghĩa là, ví dụ, chúng ta có thể lưu 5 giá trị kiểu `int` mà không cần phải khai báo 5 biến khác nhau. Ví dụ, một mảng chứa 5 giá trị nguyên kiểu `int` có tên là *billy* có thể được biểu diễn như sau:



trong đó mỗi một ô trống biểu diễn một phần tử của mảng, trong trường hợp này là các giá trị nguyên kiểu `int`. Chúng được đánh số từ 0 đến 4 vì phần tử đầu tiên của mảng luôn là 0 bất kể độ dài của nó là bao nhiêu.

Như bất kì biến nào khác, một mảng phải được khai báo trước khi có thể sử dụng. Một khai báo điển hình cho một mảng trong C++ như sau:

```
type name [elements];
```

trong đó *type* là một kiểu dữ liệu hợp lệ (**int**, **float**...), *name* là một tên biến hợp lệ và trường *elements* chỉ định mảng đó sẽ chứa bao nhiêu phần tử

Vì vậy, để khai báo *billy* như đã trình bày ở trên chúng ta chỉ cần một dòng đơn giản như sau:

```
int billy [5];
```

Chú ý: Trường *elements* bên trong cặp ngoặc `[]` phải là một giá trị hằng khi khai báo một mảng, vì mảng là một khối nhớ tĩnh có kích cỡ xác định và trình biên dịch phải có khả năng xác định xem cần bao nhiêu bộ nhớ để cấp phát cho mảng trước khi các lệnh có thể được thực hiện.

Khởi tạo một mảng.

Khi khai báo một mảng với tầm hoạt động địa phương (trong một hàm), theo mặc định nó sẽ không được khởi tạo, vì vậy nội dung của nó là không xác định cho đến khi chúng ta lưu các giá trị lên đó.

Nếu chúng ta khai báo một mảng toàn cục (bên ngoài tất cả các hàm) nó sẽ được khởi tạo và tất cả các phần tử được đặt bằng 0. Vì vậy nếu chúng ta khai báo mảng toàn cục:

```
int billy [5];
```

mọi phần tử của *billy* sẽ được khởi tạo là 0:

	<code>billy[0]</code>	<code>billy[1]</code>	<code>billy[2]</code>	<code>billy[3]</code>	<code>billy[4]</code>
billy					

Nhưng thêm vào đó, khi chúng ta khai báo một mảng, chúng ta có thể gán các giá trị khởi tạo cho từng phần tử của nó. Ví dụ:

```
int billy [5] = { 16, 2, 77, 40, 12071 };
```

lệnh trên sẽ khai báo một mảng như sau:

	0	1	2	3	4
billy	16	2	77	40	12071

Số phần tử trong mảng mà chúng ta khởi tạo với cặp ngoặc nhọn { } phải bằng số phần tử của mảng đã được khai báo với cặp ngoặc vuông []. Bởi vì điều này có thể được coi là một sự lặp lại không cần thiết nên C++ cho phép để trống giữa cặp ngoặc vuông, kích thước của mảng được xác định bằng số giá trị giữa cặp ngoặc nhọn.

Truy xuất đến các phần tử của mảng.

Ở bất kì điểm nào của chương trình trong tầm hoạt động của mảng, chúng ta có thể truy xuất từng phần tử của mảng để đọc hay chỉnh sửa như là đối với một biến bình thường. Cấu trúc của nó như sau:

`name[index]`

Như ở trong ví dụ trước ta có mảng *billy* gồm 5 phần tử có kiểu `int`, chúng ta có thể truy xuất đến từng phần tử của mảng như sau:

	0	1	2	3	4
billy	0	0	0	0	0

Ví dụ, để lưu giá trị 75 vào phần tử thứ ba của *billy* ta viết như sau:

```
billy[2] = 75;
```

và, ví dụ, để gán giá trị của phần tử thứ 3 của *billy* cho biến *a*, chúng ta viết:

```
a = billy[2];
```

Vì vậy, xét về mọi phương diện, biểu thức `billy[2]` giống như bất kì một biến kiểu `int`.

Chú ý rằng phần tử thứ ba của *billy* là `billy[2]`, vì mảng bắt đầu từ chỉ số 0. Vì vậy, phần tử cuối cùng sẽ là `billy[4]`. Vì vậy nếu chúng ta viết `billy[5]`, chúng ta sẽ truy xuất đến phần tử thứ 6 của mảng và vượt quá giới hạn của mảng.

Trong C++, việc vượt quá giới hạn chỉ số của mảng là hoàn toàn hợp lệ, tuy nhiên nó có thể gây ra những vấn đề thực sự khó phát hiện bởi vì chúng không tạo ra những lỗi trong quá trình dịch nhưng chúng có thể tạo ra những kết quả không mong muốn trong quá trình thực hiện. Nguyên nhân của việc này sẽ được nói đến kĩ hơn khi chúng ta bắt đầu sử dụng con trỏ.

Cần phải nhấn mạnh rằng chúng ta sử dụng cặp ngoặc vuông cho hai tác vụ: đầu tiên là đặt kích thước cho mảng khi khai báo chúng và thứ hai, để chỉ định chỉ số cho một phần tử cụ thể của mảng khi xem xét đến nó.

```
int billy[5];           // khai báo một mảng mới.  
billy[2] = 75;         // truy xuất đến một phần tử của  
mảng.
```

Một vài thao tác hợp lệ khác với mảng:

```
billy[0] = a;  
billy[a] = 75;
```



```
b = billy [a+2];
billy[billy[a]] = billy[2] + 5;
```

```
// ví dụ về mảng
#include <iostream.h>

int billy [] = {16, 2, 77, 40,
12071};
int n, result=0;

int main ()
{
    for ( n=0 ; n<5 ; n++ )
    {
        result += billy[n];
    }
    cout << result;
    return 0;
}
```

12206

Mảng nhiều chiều.

Mảng nhiều chiều có thể được coi như mảng của mảng, ví dụ, một mảng hai chiều có thể được tưởng tượng như là một bảng hai chiều gồm các phần tử có kiểu dữ liệu cụ thể và giống nhau.

		0	1	2	3	4
jimmy	0					
	1					
	2					

`jimmy` biểu diễn một mảng hai chiều kích thước 3x5 có kiểu `int`. Cách khai báo mảng này như sau:

```
int jimmy [3][5];
```

và, ví dụ, cách để truy xuất đến phần tử thứ hai theo chiều dọc và thứ tư theo chiều ngang trong một biểu thức như sau:

		0	1	2	3	4
jimmy	0					
	1					
	2					

↓
`jimmy[1][3]`

(hãy nhớ rằng chỉ số của mảng luôn bắt đầu từ 0).

Mảng nhiều chiều không bị giới hạn bởi hai chỉ số (hai chiều), Chúng có thể chứa bao nhiêu chỉ số tùy thích mặc dù ít khi cần phải dùng đến mảng lớn hơn 3 chiều. Hãy thử xem xét lượng bộ nhớ mà một mảng có nhiều chỉ số cần đến. Ví dụ:

```
char century [100][365][24][60][60];
```

gán một giá trị **char** cho mỗi giây trong một thế kỉ, phải cần đến hơn 3 tỷ giá trị **chars**! Chúng ta sẽ phải cần khoảng 3GB RAM để khai báo nó.

Mảng nhiều chiều thực ra là một khái niệm trừu tượng vì chúng ta có thể có kết quả tương tự với mảng một chiều bằng một thao tác đơn giản giữa các chỉ số của nó:

```
int jimmy [3][5]; tương đương với  
int jimmy [15]; (3 * 5 = 15)
```

Dưới đây là hai ví dụ với cùng một kết quả như nhau, một sử dụng mảng hai chiều và một sử dụng mảng một chiều:

```
// multidimensional array  
#include <iostream.h>  
  
#define WIDTH 5  
#define HEIGHT 3  
  
int jimmy [HEIGHT][WIDTH];  
int n,m;  
  
int main ()  
{  
    for (n=0;n<HEIGHT;n++)  
        for (m=0;m<WIDTH;m++)  
        {  
            jimmy[n][m]=(n+1)*(m+1);  
        }  
    return 0;  
}
```

```
// pseudo-multidimensional array  
#include <iostream.h>  
  
#define WIDTH 5  
#define HEIGHT 3  
  
int jimmy [HEIGHT * WIDTH];  
int n,m;  
  
int main ()  
{  
    for (n=0;n<HEIGHT;n++)  
        for (m=0;m<WIDTH;m++)  
        {  
            jimmy[n * WIDTH +  
m]=(n+1)*(m+1);  
        }  
    return 0;  
}
```

không một chương trình nào viết gì ra màn hình nhưng cả hai đều gán giá trị vào khối nhớ có tên **jimmy** theo cách sau:

jimmy {		0	1	2	3	4
	0	1	2	3	4	5
	1	2	4	6	8	10
	2	3	6	9	12	15

Chúng ta đã định nghĩa hằng (**#define**) để đơn giản hóa những chỉnh sửa sau này của chương trình, ví dụ, trong trường hợp chúng ta quyết định tăng kích thước của mảng với chiều cao là 4 thay vì là 3, chúng ta chỉ cần thay đổi dòng:

```
#define HEIGHT 3
```

thành

```
#define HEIGHT 4
```

và không phải có thêm sự thay đổi nào nữa đối với chương trình.

Dùng mảng làm tham số.

Vào một lúc nào đó có thể chúng ta cần phải truyền một mảng tới một hàm như là một tham số. Trong C++, việc truyền theo tham số giá trị một khối nhớ là không hợp lệ, ngay cả khi nó được tổ chức thành một mảng. Tuy nhiên chúng ta lại được phép truyền địa chỉ của nó, việc này cũng tạo ra kết quả thực tế giống thao tác ở trên nhưng lại nhanh hơn nhiều và hiệu quả hơn.

Để có thể nhận mảng là tham số thì điều duy nhất chúng ta phải làm khi khai báo hàm là chỉ định trong phần tham số kiểu dữ liệu cơ bản của mảng, tên mảng và cặp ngoặc vuông trống. Ví dụ, hàm sau:

```
void procedure (int arg[])
```

nhận vào một tham số có kiểu "mảng của **char**" và có tên **arg**. Để truyền tham số cho hàm này một mảng được khai báo:

```
int myarray [40];
```

chỉ cần gọi hàm như sau:

```
procedure (myarray);
```

Dưới đây là một ví dụ cụ thể

```
// arrays as parameters
#include <iostream.h>

void printarray (int arg[], int
length) {
    for (int n=0; n<length; n++)
        cout << arg[n] << " ";
    cout << "\n";
}
```

```
5 10 15
2 4 6 8 10
```

```
int main ()
{
    int firstarray[] = {5, 10, 15};
    int secondarray[] = {2, 4, 6, 8, 10};
    printarray (firstarray,3);
    printarray (secondarray,5);
    return 0;
}
```

Như bạn có thể thấy, tham số đầu tiên (`int arg[]`) chấp nhận mọi mảng có kiểu cơ bản là `int`, bất kể độ dài của nó là bao nhiêu, vì vậy cần thiết phải có tham số thứ hai để báo cho hàm này biết độ dài của mảng mà chúng ta truyền cho nó.

Trong phần khai báo hàm chúng ta cũng có thể dùng tham số là các mảng nhiều chiều. Cấu trúc của mảng 3 chiều như sau:

```
base_type[][depth][depth]
```

ví dụ, một hàm với tham số là mảng nhiều chiều có thể như sau:

```
void procedure (int myarray[][3][4])
```

chú ý rằng cặp ngoặc vuông đầu tiên để trống nhưng các cặp ngoặc sau thì không. Bạn luôn luôn phải làm vậy vì trình biên dịch C++ phải có khả năng xác định độ lớn của các chiều thêm vào của mảng.

style="BORDER-RIGHT: medium none; PADDING-RIGHT: 0in; BORDER-TOP: medium none; PADDING-LEFT: 0in; PADDING-BOTTOM: 0in; BORDER-LEFT: medium none; PADDING-TOP: 0in; BORDER-BOTTOM: windowtext 3pt solid">

Mảng, cả một chiều và nhiều chiều, khi truyền cho hàm như là một tham số thường là nguyên nhân gây lỗi cho những lập trình viên thiếu kinh nghiệm. Các bạn nên đọc bài **3.3. Con trỏ** để có thể hiểu rõ hơn mảng hoạt động như thế nào.

Bài 8 : Xâu Ký Tự

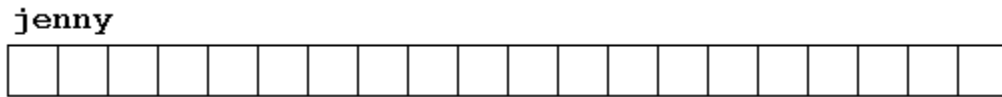
Trong tất cả các chương trình chúng ta đã thấy cho đến giờ, chúng ta chỉ sử dụng các biến kiểu số, chỉ dùng để biểu diễn các số. Nhưng bên cạnh các biến kiểu số còn có các xâu ký tự, chúng cho phép chúng ta biểu diễn các chuỗi ký tự như là các từ, câu, đoạn văn bản... Cho đến giờ chúng ta mới chỉ dùng chúng dưới dạng hằng chứ chưa quan tâm đến các biến có thể chứa chúng.

Trong C++ không có kiểu dữ liệu *cơ bản* để lưu các xâu ký tự. Để có thể thỏa mãn nhu cầu này, người ta sử dụng mảng có kiểu `char`. Hãy nhớ rằng kiểu dữ liệu này (`char`) chỉ có thể lưu trữ một ký tự đơn, bởi vậy nó được dùng để tạo ra xâu của các ký tự đơn.

Ví dụ, mảng sau (hay là chuỗi ký tự):

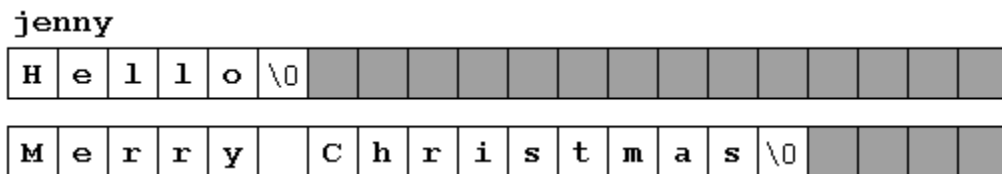
```
char jenny [20];
```

có thể lưu một chuỗi ký tự với độ dài cực đại là 20 ký tự. Bạn có thể tưởng tượng nó như sau:



Kích thước cực đại này không cần phải luôn luôn dùng đến. Ví dụ, **jenny** có thể lưu chuỗi "Hello" hay "Merry christmas". Vì các mảng ký tự có thể lưu các chuỗi ký tự ngắn hơn độ dài của nó, trong C++ đã có một quy ước để kết thúc một nội dung của một chuỗi ký tự bằng một ký tự null, có thể được viết là '\0'.

Chúng ta có thể biểu diễn **jenny** (một mảng có 20 phần tử kiểu **char**) khi lưu trữ chuỗi ký tự "Hello" và "Merry Christmas" theo cách sau:



Chú ý rằng sau nội dung của chuỗi, một ký tự null ('\0') được dùng để báo hiệu kết thúc chuỗi. Những ô màu xám biểu diễn những giá trị không xác định.

Khởi tạo các chuỗi ký tự.

Vì những chuỗi ký tự là những mảng bình thường nên chúng cũng như các mảng khác. Ví dụ, nếu chúng ta muốn khởi tạo một chuỗi ký tự với những giá trị xác định chúng ta có thể làm điều đó tương tự như với các mảng khác:

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

Tuy nhiên, chúng ta có thể khởi tạo giá trị cho một chuỗi ký tự bằng cách khác: sử dụng các *hằng chuỗi ký tự*.

Trong các biểu thức chúng ta đã sử dụng trong các ví dụ trong các chương trước các hằng chuỗi ký tự để xuất hiện vài lần. Chúng được biểu diễn trong cặp ngoặc kép ("), ví dụ:

```
"the result is: "
```

là một hằng chuỗi ký tự chúng ta sử dụng ở một số chỗ.

Không giống như dấu nháy đơn (') cho phép biểu diễn hằng kí tự, cặp ngoặc kép (") là hằng biểu diễn một chuỗi kí tự liên tiếp, và ở cuối chuỗi một kí tự null ('\0') luôn được tự động thêm vào.

Vì vậy chúng ta có thể khởi tạo xâu **mystring** theo một trong hai cách sau đây:

```
char mystring [] = { 'H', 'e', 'l', 'l', 'o', '\0' };  
char mystring [] = "Hello";
```

Trong cả hai trường hợp mảng (hay xâu kí tự) **mystring** được khai báo với kích thước 6 kí tự: 5 kí tự biểu diễn **Hello** cộng với một kí tự null.

Trước khi tiếp tục, tôi cần phải nhắc nhở bạn rằng việc gán nhiều hằng như việc sử dụng dấu ngoặc kép (") chỉ hợp lệ khi *khởi tạo* mảng, tức là lúc khai báo mảng. Các biểu thức trong chương trình như:

```
mystring = "Hello";  
mystring[] = "Hello";
```

là không hợp lệ, cả câu lệnh dưới đây cũng vậy:

```
mystring = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

Vậy hãy nhớ: Chúng ta chỉ có thể "gán" nhiều hằng cho một mảng vào lúc khởi tạo nó. Nguyên nhân là một thao tác gán (=) không thể nhận về trái là cả một mảng mà chỉ có thể nhận một trong những phần tử của nó. Vào thời điểm khởi tạo mảng là một trường hợp đặc biệt, vì nó không thực sự là một lệnh gán mặc dù nó sử dụng dấu bằng (=).

Gán giá trị cho xâu kí tự

Vì về trái của một lệnh gán chỉ có thể là một phần tử của mảng chứ không thể là cả mảng, chúng ta có thể gán một xâu kí tự cho một mảng kiểu **char** sử dụng một phương pháp như sau:

```
mystring[0] = 'H';  
mystring[1] = 'e';  
mystring[2] = 'l';  
mystring[3] = 'l';  
mystring[4] = 'o';  
mystring[5] = '\0';
```

Nhưng rõ ràng đây không phải là một phương pháp thực tế. Để gán giá trị cho một xâu kí tự, chúng ta có thể sử dụng loạt hàm kiểu **strcpy** (**string copy**), hàm này được định nghĩa trong **string.h** và có thể được gọi như sau:

```
strcpy (string1, string2);
```

Lệnh này copy nội dung của *string2* sang *string1*. *string2* có thể là một mảng, con trỏ hay một hằng chuỗi kí tự, bởi vậy lệnh sau đây là một cách đúng để gán chuỗi hằng "Hello" cho *mystring*:

```
strcpy (mystring, "Hello");
```

Ví dụ:

```
// setting value to string
#include <iostream.h>
#include <string.h>

int main ()
{
    char szMyName [20];
    strcpy (szMyName, "J. Soulie");
    cout << szMyName;
    return 0;
}
```

J. Soulie

Đề ý rằng chúng ta phải include file **<string.h>** để có thể sử dụng hàm **strcpy**.

Mặc dù chúng ta luôn có thể viết một hàm đơn giản như hàm **setstring** dưới đây để thực hiện một thao tác giống như **strcpy**:

```
// setting value to string
#include <iostream.h>

void setstring (char szOut [], char
szIn [])
{
    int n=0;
    do {
        szOut[n] = szIn[n];
        n++;
    } while (szIn[n] != 0);
}

int main ()
{
    char szMyName [20];
    setstring (szMyName, "J. Soulie");
    cout << szMyName;
    return 0;
}
```

J. Soulie

Một phương thức thường dùng khác để gán giá trị cho một mảng là sử dụng trực tiếp dòng nhập dữ liệu (**cin**). Trong trường hợp này giá trị của chuỗi kí tự được gán bởi người dùng trong quá trình chương trình thực hiện.

Khi **cin** được sử dụng với các chuỗi ký tự nó thường được dùng với phương thức **getline** của nó, phương thức này có thể được gọi như sau:

```
cin.getline ( char buffer[], int length, char delimiter = '  
\n' );
```

trong đó **buffer** (bộ đệm) là địa chỉ nơi sẽ lưu trữ dữ liệu vào (như là một mảng chẳng hạn), **length** là độ dài cực đại của bộ đệm (kích thước của mảng) và **delimiter** là ký tự được dùng để kết thúc việc nhập, mặc định - nếu chúng ta không dùng tham số này - sẽ là ký tự xuống dòng ('\n').

Ví dụ sau đây lặp lại tất cả những gì bạn gõ trên bàn phím. Nó rất đơn giản nhưng là một ví dụ cho thấy bạn có thể sử dụng **cin.getline** với các chuỗi ký tự như thế nào:

```
// cin with strings  
#include <iostream.h>
```

```
int main ()  
{  
    char mybuffer [100];  
    cout << "What's your name? ";  
    cin.getline (mybuffer,100);  
    cout << "Hello " << mybuffer <<  
    ".\n";  
    cout << "Which is your favourite  
team? ";  
    cin.getline (mybuffer,100);  
    cout << "I like " << mybuffer <<  
    " too.\n";  
    return 0;  
}
```

```
What's your name? Juan  
Hello Juan.  
Which is your favourite team? Inter  
Milan  
I like Inter Milan too.
```

Chú ý trong cả hai lời gọi **cin.getline** chúng ta sử dụng cùng một biến chuỗi (**mybuffer**). Những gì chương trình làm trong lời gọi thứ hai đơn giản là thay thế nội dung của **buffer** trong lời gọi cũ bằng nội dung mới.

Nếu bạn còn nhớ phần nói về giao tiếp với, bạn sẽ nhớ rằng chúng ta đã sử dụng toán tử **>>** để nhận dữ liệu trực tiếp từ đầu vào chuẩn. Phương thức này có thể được dùng với các chuỗi ký tự thay cho **cin.getline**. Ví dụ, trong chương trình của chúng ta, khi chúng ta muốn nhận dữ liệu từ người dùng chúng ta có thể viết:

```
cin >> mybuffer;
```

lệnh này sẽ làm việc như nó có những hạn chế sau mà **cin.getline** không có:

- Nó chỉ có thể nhận những từ đơn (không nhận được cả câu) vì phương thức này sử dụng ký tự trống (bao gồm cả dấu cách, dấu tab và dấu xuống dòng) làm dấu hiệu kết thúc..

- Nó không cho phép chỉ định kích thước cho bộ đệm. Chương trình của bạn có thể chạy không ổn định nếu dữ liệu vào lớn hơn kích cỡ của mảng chứa nó.

Vì những nguyên nhân trên, khi muốn nhập vào các xâu kí tự bạn nên sử dụng `cin.getline` thay vì `cin >>`.

Chuyển đổi xâu kí tự sang các kiểu khác.

Vì một xâu kí tự có thể biểu diễn nhiều kiểu dữ liệu khác như dạng số nên việc chuyển đổi nội dung như vậy sang dạng số là rất hữu ích. Ví dụ, một xâu có thể mang giá trị "1977" nhưng đó là một chuỗi gồm 5 kí tự (kể cả kí tự null) và không dễ gì chuyển thành một số nguyên. Vì vậy thư viện `cstdlib` (`stdlib.h`) đã cung cấp 3 macro/hàm hữu ích sau:

- **atoi**: chuyển xâu thành kiểu `int`.
- **atol**: chuyển xâu thành kiểu `long`.
- **atof**: chuyển xâu thành kiểu `float`.

Tất cả các hàm này nhận một tham số và trả về giá trị số (`int`, `long` hoặc `float`). Các hàm này khi kết hợp với phương thức `getline` của `cin` là một cách đáng tin cậy hơn phương thức `cin>>` cổ điển khi yêu cầu người sử dụng nhập vào một số:

```
// cin and ato* functions
#include <iostream.h>
#include <stdlib.h>

int main ()
{
    char mybuffer [100];
    float price;
    int quantity;
    cout << "Enter price: ";
    cin.getline (mybuffer,100);
    price = atof (mybuffer);
    cout << "Enter quantity: ";
    cin.getline (mybuffer,100);
    quantity = atoi (mybuffer);
    cout << "Total price: " <<
    price*quantity;
    return 0;
}
```

```
Enter price: 2.75
Enter quantity: 21
Total price: 57.75
```

Các hàm để thao tác trên chuỗi

Thư viện `cstring` (`string.h`) không chỉ có hàm `strcpy` mà còn có nhiều hàm khác để thao tác trên chuỗi. Dưới đây là giới thiệu lướt qua của các hàm thông dụng nhất:

strcat: `char* strcat (char* dest, const char* src);`
Gắn thêm chuỗi *src* vào phía cuối của *dest*. Trả về *dest*.

strcmp: `int strcmp (const char* string1, const char* string2);`
So sánh hai chuỗi *string1* và *string2*. Trả về 0 nếu hai chuỗi là bằng nhau.

strcpy: `char* strcpy (char* dest, const char* src);`
Copy nội dung của *src* cho *dest*. Trả về *dest*.

strlen: `size_t strlen (const char* string);`
Trả về độ dài của *string*.

Chú ý: **char*** hoàn toàn tương đương với **char[]**

Bài 9 : Con Trỏ

Chúng ta đã biết các biến chính là các ô nhớ mà chúng ta có thể truy xuất dưới các tên. Các biến này được lưu trữ tại những chỗ cụ thể trong bộ nhớ. Đối với chương trình của chúng ta, bộ nhớ máy tính chỉ là một dãy gồm các ô nhớ 1 byte, mỗi ô có một địa chỉ xác định.

Một sự mô hình tốt đối với bộ nhớ máy tính chính là một phố trong một thành phố. Trên một phố tất cả các ngôi nhà đều được đánh số tuần tự với một cái tên duy nhất nên nếu chúng ta nói đến số 27 phố Trần Hưng Đạo thì chúng ta có thể tìm được nơi đó mà không lầm lẫn vì chỉ có một ngôi nhà với số như vậy.

Cũng với cách tổ chức tương tự như việc đánh số các ngôi nhà, hệ điều hành tổ chức bộ nhớ thành những số đơn nhất, tuần tự, nên nếu chúng ta nói đến vị trí 1776 trong bộ nhớ chúng ta biết chính xác ô nhớ đó vì chỉ có một vị trí với địa chỉ như vậy.

Toán tử lấy địa chỉ (&).

Vào thời điểm mà chúng ta khai báo một biến thì nó phải được lưu trữ trong một vị trí cụ thể trong bộ nhớ. Nói chung chúng ta không quyết định nơi nào biến đó được đặt - thật may mắn rằng điều đó đã được làm tự động bởi trình biên dịch và hệ điều hành, nhưng một khi hệ điều hành đã gán một địa chỉ cho biến thì chúng ta có thể muốn biết biến đó được lưu trữ ở đâu.

Điều này có thể được thực hiện bằng cách đặt trước tên biến một dấu và (&), có nghĩa là "**địa chỉ của**". Ví dụ:

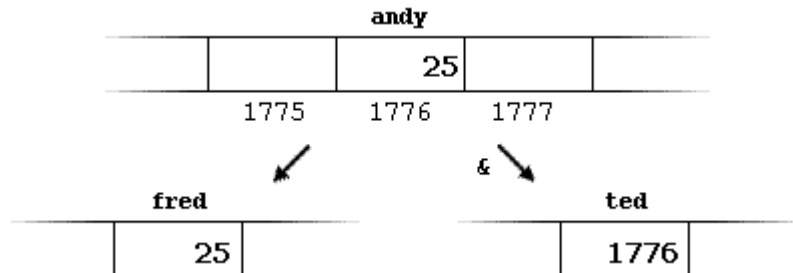
```
ted = &andy;
```

sẽ gán cho biến **ted** địa chỉ của biến **andy**, vì khi đặt trước tên biến **andy** dấu và (&) chúng ta không còn nói đến nội dung của biến đó mà chỉ nói đến địa chỉ của nó trong bộ nhớ.

Giả sử rằng biến **andy** được đặt ở ô nhớ có địa chỉ **1776** và chúng ta viết như sau:

```
andy = 25;
fred = andy;
ted = &andy;
```

kết quả sẽ giống như trong sơ đồ dưới đây:



Chúng ta đã gán cho **fred** nội dung của biến **andy** như chúng ta đã làm rất lần nhiều khác trong những phần trước nhưng với biến **ted** chúng ta đã gán địa chỉ mà hệ điều hành lưu giá trị của biến **andy**, chúng ta vừa giả sử nó là 1776.

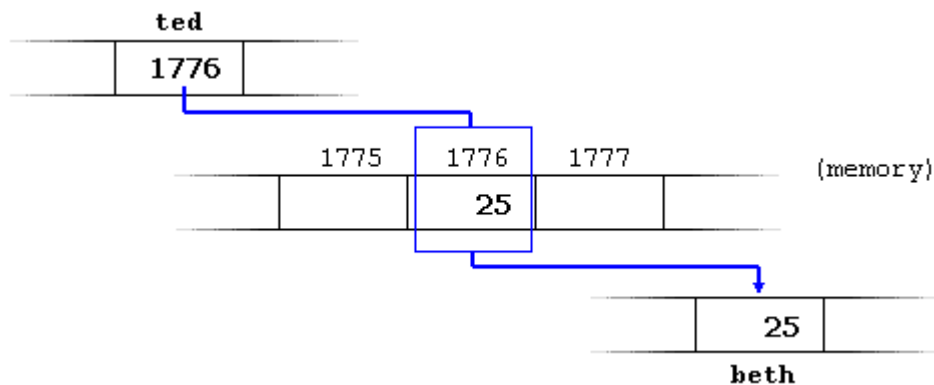
Những biến lưu trữ địa chỉ của một biến khác (như **ted** ở trong ví dụ trước) được gọi là **con trỏ**. Trong C++ con trỏ có rất nhiều ưu điểm và chúng được sử dụng rất thường xuyên, Tiếp theo chúng ta sẽ thấy các biến kiểu này được khai báo như thế nào.

Toán tử tham chiếu (*)

Bằng cách sử dụng con trỏ chúng ta có thể truy xuất trực tiếp đến giá trị được lưu trữ trong biến được trỏ bởi nó bằng cách đặt trước tên biến con trỏ một dấu sao (*) - ở đây có thể được dịch là "**giá trị được trỏ bởi**". Vì vậy, nếu chúng ta viết:

```
beth = *ted;
```

(chúng ta có thể đọc nó là: "beth bằng giá trị được trỏ bởi ted" **beth** sẽ mang giá trị 25, vì **ted** bằng 1776 và giá trị trỏ bởi 1776 là 25.



Bạn phải phân biệt được rằng `ted` có giá trị 1776, nhưng `*ted` (với một dấu sao đằng trước) trở tới giá trị được lưu trữ trong địa chỉ 1776, đó là 25. Hãy chú ý sự khác biệt giữa việc có hay không có dấu sao tham chiếu.

```
beth = ted;    // beth bằng ted ( 1776 )
beth = *ted;   // beth bằng giá trị được trữ bởi ( 25 )
```

Toán tử lấy địa chỉ (&)

Nó được dùng như là một tiền tố của biến và có thể được dịch là "**địa chỉ của**", vì vậy `&variable1` có thể được đọc là "địa chỉ của `variable1`".

Toán tử tham chiếu (*)

Nó chỉ ra rằng cái cần được tính toán là nội dung được trữ bởi biểu thức được coi như là một địa chỉ. Nó có thể được dịch là "**giá trị được trữ bởi**".

`*mypointer` được đọc là "*giá trị được trữ bởi `mypointer`*".

Vào lúc này, với những ví dụ đã viết ở trên

```
andy = 25;
ted = &andy;
```

bạn có thể dễ dàng nhận ra tất cả các biểu thức sau là đúng:

```
andy == 25
&andy == 1776
ted == 1776
*ted == 25
```

Khai báo biến kiểu con trỏ

Vì con trỏ có khả năng tham chiếu trực tiếp đến giá trị mà chúng trỏ tới nên cần thiết phải chỉ rõ kiểu dữ liệu nào mà một biến con trỏ trỏ tới khai báo nó. Vì vậy, khai báo của một biến con trỏ sẽ có mẫu sau:

```
type * pointer_name;
```

trong đó **type** là kiểu dữ liệu được trỏ tới, không phải là kiểu của bản thân con trỏ. Ví dụ:

```
int * number;
char * character;
float * greatnumber;
```

đó là ba khai báo của con trỏ. Mỗi biến đều trỏ tới một kiểu dữ liệu khác nhau nhưng cả ba đều là con trỏ và chúng đều chiếm một lượng bộ nhớ như nhau (kích thước của một biến con trỏ tùy thuộc vào hệ điều hành). nhưng dữ liệu mà chúng trỏ tới không chiếm lượng bộ nhớ như nhau, một kiểu `int`, một kiểu `char` và cái còn lại kiểu `float`.

Tôi phải nhấn mạnh lại rằng dấu sao (*) mà chúng ta đặt khi khai báo một con trỏ chỉ có nghĩa rằng: đó là một con trỏ và hoàn toàn không liên quan đến toán tử tham chiếu mà chúng ta đã xem xét trước đó. Đó đơn giản chỉ là hai tác vụ khác nhau được biểu diễn bởi cùng một dấu.

```
// my first pointer
#include <iostream.h>

int main ()
{
    int value1 = 5, value2 = 15;
    int * mypointer;

    mypointer = &value1;
    *mypointer = 10;
    mypointer = &value2;
    *mypointer = 20;
    cout << "value1==" << value1 <<
    "/ value2==" << value2;
    return 0;
}
```

```
value1==10 / value2==20
```

Chú ý rằng giá trị của **value1** và **value2** được thay đổi một cách gián tiếp. Đầu tiên chúng ta gán cho **mypointer** địa chỉ của **value1** dùng toán tử lấy địa chỉ (&) và sau đó chúng ta gán 10 cho giá trị được trỏ bởi **mypointer**, đó là giá trị được trỏ bởi **value1** vì vậy chúng ta đã sửa biến **value1** một cách gián tiếp

Để bạn có thể thấy rằng một con trỏ có thể mang một vài giá trị trong cùng một chương trình chúng ta sẽ lặp lại quá trình với **value2** và với cùng một con trỏ.

Đây là một ví dụ phức tạp hơn một chút:

```
// more pointers
#include <iostream.h>

int main ()
{
    int value1 = 5, value2 = 15;
    int *p1, *p2;

    p1 = &value1;      // p1 = địa chỉ
    của value1
    p2 = &value2;      // p2 = địa chỉ
    của value2
    *p1 = 10;          // giá trị trỏ
    bởi p1 = 10
    *p2 = *p1;         // giá trị trỏ
    bởi p2 = giá trị trỏ bởi p1
    p1 = p2;           // p1 = p2
    (phép gán con trỏ)
```

```
value1==10 / value2==20
```

```

    *p1 = 20;           // giá trị trở
    bởi p1 = 20

    cout << "value1==" << value1 <<
    "/ value2==" << value2;
    return 0;
}

```

Một dòng có thể gây sự chú ý của bạn là:

```
int *p1, *p2;
```

dòng này khai báo hai con trỏ bằng cách đặt dấu sao (*) trước mỗi con trỏ. Nguyên nhân là kiểu dữ liệu khai báo cho cả dòng là `int` và vì theo thứ tự từ phải sang trái, dấu sao được tính trước tên kiểu. Chúng ta đã nói đến điều này trong bài [1.3: Các toán tử](#).

Con trỏ và mảng.

Trong thực tế, tên của một mảng tương đương với địa chỉ phần tử đầu tiên của nó, giống như một con trỏ tương đương với địa chỉ của phần tử đầu tiên mà nó trỏ tới, vì vậy thực tế chúng hoàn toàn như nhau. Ví dụ, cho hai khai báo sau:

```
int numbers [20];
int * p;
```

lệnh sau sẽ hợp lệ:

```
p = numbers;
```

Ở đây `p` và `numbers` là tương đương và chúng có cùng thuộc tính, sự khác biệt duy nhất là chúng ta có thể gán một giá trị khác cho con trỏ `p` trong khi `numbers` luôn trỏ đến phần tử đầu tiên trong số 20 phần tử kiểu `int` mà nó được định nghĩa với. Vì vậy, không giống như `p` - đó là một biến con trỏ bình thường, `numbers` là một con trỏ hằng. Lệnh gán sau đây là không hợp lệ:

```
numbers = p;
```

bởi vì `numbers` là một mảng (con trỏ hằng) và không có giá trị nào có thể được gán cho các hằng.

Vì con trỏ cũng có mọi tính chất của một biến nên tất cả các biểu thức có con trỏ trong ví dụ dưới đây là hoàn toàn hợp lệ:

```
// more pointers
#include <iostream.h>
```

```
10, 20, 30, 40, 50,
```

```

int main ()
{
    int numbers[5];
    int * p;
    p = numbers; *p = 10;
    p++; *p = 20;
    p = &numbers[2]; *p = 30;
    p = numbers + 3; *p = 40;
    p = numbers; *(p+4) = 50;
    for (int n=0; n<5; n++)
        cout << numbers[n] << ", ";
    return 0;
}

```

Trong bài "mảng" chúng ta đã dùng dấu ngoặc vuông để chỉ ra phần tử của mảng mà chúng ta muốn trở đến. Cặp ngoặc vuông này được coi như là toán tử offset và ý nghĩa của chúng không đổi khi được dùng với biến con trỏ. Ví dụ, hai biểu thức sau đây:

```

a[5] = 0;           // a [offset of 5] = 0
*(a+5) = 0;         // pointed by (a+5) = 0

```

là hoàn toàn tương đương và hợp lệ bất kể **a** là mảng hay là một con trỏ.

Khởi tạo con trỏ

Khi khai báo con trỏ có thể chúng ta sẽ muốn chỉ định rõ ràng chúng sẽ trỏ tới biến nào,

```

int number;
int *tommy = &number;

```

là tương đương với:

```

int number;
int *tommy;
tommy = &number;

```

Trong một phép gán con trỏ chúng ta phải luôn luôn gán địa chỉ mà nó trỏ tới chứ không phải là giá trị mà nó trỏ tới. Bạn cần phải nhớ rằng khi khai báo một biến con trỏ, dấu sao (*) được dùng để chỉ ra nó là một con trỏ, và hoàn toàn khác với toán tử tham chiếu. Đó là hai toán tử khác nhau mặc dù chúng được viết với cùng một dấu. Vì vậy, các câu lệnh sau là không hợp lệ:

```

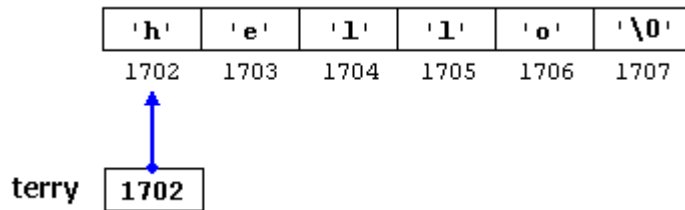
int number;
int *tommy;
*tommy = &number;

```

Như đối với mảng, trình biên dịch cho phép chúng ta khởi tạo giá trị mà con trỏ trỏ tới bằng giá trị hằng vào thời điểm khai báo biến con trỏ:

```
char * terry = "hello";
```

trong trường hợp này một khối nhớ tĩnh được dành để chứa "hello" và một con trỏ trỏ tới kí tự đầu tiên của khối nhớ này (đó là kí tự h) được gán cho **terry**. Nếu "hello" được lưu tại địa chỉ 1702, lệnh khai báo trên có thể được hình dung như thế này:

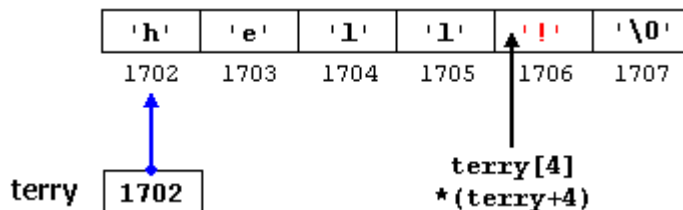


cần phải nhắc lại rằng **terry** mang giá trị 1702 chứ không phải là 'h' hay "hello".

Biến con trỏ **terry** trỏ tới một chuỗi kí tự và nó có thể được sử dụng như là đối với một mảng (hãy nhớ rằng một mảng chỉ đơn thuần là một con trỏ hằng). Ví dụ, nếu chúng ta muốn thay kí tự 'o' bằng một dấu chấm than, chúng ta có thể thực hiện việc đó bằng hai cách:

```
terry[4] = '!';  
*(terry+4) = '!';
```

hãy nhớ rằng viết **terry[4]** là hoàn toàn giống với viết ***(terry+4)** mặc dù biểu thức thông dụng nhất là cái đầu tiên. Với một trong hai lệnh trên chuỗi do **terry** trỏ đến sẽ có giá trị như sau:



Các phép tính số học với pointer

Việc thực hiện các phép tính số học với con trỏ hơi khác so với các kiểu dữ liệu số nguyên khác. Trước hết, chỉ phép cộng và trừ là được phép dùng. Nhưng cả cộng và trừ đều cho kết quả phụ thuộc vào kích thước của kiểu dữ liệu mà biến con trỏ trỏ tới.

Chúng ta thấy có nhiều kiểu dữ liệu khác nhau tồn tại và chúng có thể chiếm chỗ nhiều hơn hoặc ít hơn các kiểu dữ liệu khác. Ví dụ, trong các kiểu số nguyên, *char* chiếm 1 byte, *short* chiếm 2 byte và *long* chiếm 4 byte.

Giả sử chúng ta có 3 con trỏ sau:


```
char *mychar;  
short *myshort;  
long *mylong;
```

và chúng lần lượt trở tới ô nhớ 1000, 2000 and 3000.

Nếu chúng ta viết

```
mychar++;  
myshort++;  
mylong++;
```

`mychar` - như bạn mong đợi - sẽ mang giá trị 1001. Tuy nhiên `myshort` sẽ mang giá trị 2002 và `mylong` mang giá trị 3004. Nguyên nhân là khi cộng thêm 1 vào một con trỏ thì nó sẽ trở tới phần tử tiếp theo có cùng kiểu mà nó đã được định nghĩa, vì vậy kích thước tính bằng byte của kiểu dữ liệu nó trở tới sẽ được cộng thêm vào biến con trỏ.

Điều này đúng với cả hai phép toán cộng và trừ đối với con trỏ. Chúng ta cũng hoàn toàn thu được kết quả như trên nếu viết:

```
mychar = mychar + 1;  
myshort = myshort + 1;  
mylong = mylong + 1;
```

Cần phải cảnh báo bạn rằng cả hai toán tử tăng (++) và giảm (--) đều có quyền ưu tiên lớn hơn toán tử tham chiếu (*), vì vậy biểu thức sau đây có thể dẫn tới kết quả sai:

```
*p++;  
*p++ = *q++;
```

Lệnh đầu tiên tương đương với `*(p++)` điều mà nó thực hiện là tăng `p` (địa chỉ ô nhớ mà nó trở tới chứ không phải là giá trị trở tới).

Lệnh thứ hai, cả hai toán tử tăng (++) đều được thực hiện sau khi giá trị của `*q` được gán cho `*p` và sau đó cả `q` và `p` đều tăng lên 1. Lệnh này tương đương với:

```
*p = *q;  
p++;  
q++;
```

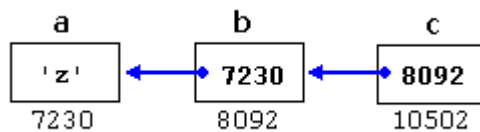
Như đã nói trong các bài trước, tôi khuyên các bạn nên dùng các cặp ngoặc đơn để tránh những kết quả không mong muốn.

Con trỏ trở tới con trỏ

C++ cho phép sử dụng các con trỏ trỏ tới các con trỏ khác giống như là trỏ tới dữ liệu. Để làm việc đó chúng ta chỉ cần thêm một dấu sao (*) cho mỗi mức tham chiếu.

```
char a;  
char * b;  
char ** c;  
a = 'z';  
b = &a;  
c = &b;
```

giả sử rằng a,b,c được lưu ở các ô nhớ 7230, 8092 and 10502, ta có thể mô tả đoạn mã trên như sau:



Điểm mới trong ví dụ này là biến **c**, chúng ta có thể nói về nó theo 3 cách khác nhau, mỗi cách sẽ tương ứng với một giá trị khác nhau:

```
c là một biến có kiểu (char **) mang giá trị 8092  
*c là một biến có kiểu (char*) mang giá trị 7230  
**c là một biến có kiểu (char) mang giá trị 'z'
```

Con trỏ không kiểu

Con trỏ không kiểu là một loại con trỏ đặc biệt. Nó có thể trỏ tới bất kì loại dữ liệu nào, từ giá trị nguyên hoặc thực cho tới một chuỗi ký tự. Hạn chế duy nhất của nó là dữ liệu được trỏ tới không thể được tham chiếu tới một cách trực tiếp (chúng ta không thể dùng toán tử tham chiếu * với chúng) vì độ dài của nó là không xác định và vì vậy chúng ta phải dùng đến toán tử chuyển kiểu dữ liệu hay phép gán để chuyển con trỏ không kiểu thành một con trỏ trỏ tới một loại dữ liệu cụ thể.

Một trong những tiện ích của nó là cho phép truyền tham số cho hàm mà không cần chỉ rõ kiểu

```
// integer increaser  
#include <iostream.h>  
  
void increase (void* data, int  
type)  
{  
    switch (type)  
    {  
        case sizeof(char) :  
            (*(char*)data)++; break;  
        case sizeof(short):
```

6, 10, 13

```

        (*(short*)data)++; break;
        case sizeof(long) :
        (*(long*)data)++; break;
    }
}

int main ()
{
    char a = 5;
    short b = 9;
    long c = 12;
    increase (&a,sizeof(a));
    increase (&b,sizeof(b));
    increase (&c,sizeof(c));
    cout << (int) a << ", " << b <<
    ", " << c;
    return 0;
}

```

sizeof là một toán tử của ngôn ngữ C++, nó trả về một giá trị hằng là kích thước tính bằng byte của tham số truyền cho nó, ví dụ **sizeof(char)** bằng 1 vì kích thước của **char** là 1 byte.

Con trỏ hàm

C++ cho phép thao tác với các con trỏ hàm. Tiện ích tuyệt vời này cho phép truyền một hàm như là một tham số đến một hàm khác. Để có thể khai báo một con trỏ trỏ tới một hàm chúng ta phải khai báo nó như là khai báo mẫu của một hàm nhưng phải bao trong một cặp ngoặc đơn () tên của hàm và chèn dấu sao (*) đằng trước.

```

// pointer to functions
#include <iostream.h>

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

int subtraction (int a, int b)
{ return (a-b); }

int (*minus)(int,int) =
subtraction;

int operation (int x, int y, int
(*functocall)(int,int))
{
    int g;
    g = (*functocall)(x,y);
    return (g);
}

int main ()

```

8

```
{
    int m,n;
    m = operation (7, 5, &addition);
    n = operation (20, m, minus);
    cout <<n;
    return 0;
}
```

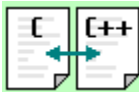
Trong ví dụ này, **minus** là một con trỏ toàn cục trỏ tới một hàm có hai tham số kiểu **int**, con trỏ này được gán để trỏ tới hàm **subtraction**, tất cả đều trên một dòng:

```
int (* minus)(int,int) = subtraction;
```

Bài 10 : Bộ Nhớ Động

Cho đến nay, trong các chương trình của chúng ta, tất cả những phần bộ nhớ chúng ta có thể sử dụng là các biến các mảng và các đối tượng khác mà chúng ta đã khai báo. Kích cỡ của chúng là cố định và không thể thay đổi trong thời gian chương trình chạy. Nhưng nếu chúng ta cần một lượng bộ nhớ mà kích cỡ của nó chỉ có thể được xác định khi chương trình chạy, ví dụ như trong trường hợp chúng ta nhận thông tin từ người dùng để xác định lượng bộ nhớ cần thiết.

Giải pháp ở đây chính là *bộ nhớ động*, C++ đã tích hợp hai toán tử *new* và *delete* để thực hiện việc này



Hai toán tử **new** và **delete** chỉ có trong C++. Ở phần sau của bài chúng ta sẽ biết những thao tác tương đương với các toán tử này trong C.

Toán tử **new** và **new[]**

Để có thể có được bộ nhớ động chúng ta có thể dùng toán tử **new**. Theo sau toán tử này là tên kiểu dữ liệu và có thể là số phần tử cần thiết được đặt trong cặp ngoặc vuông. Nó trả về một con trỏ trỏ tới đầu của khối nhớ vừa được cấp phát. Dạng thức của toán tử này như sau:

```
pointer = new type
```

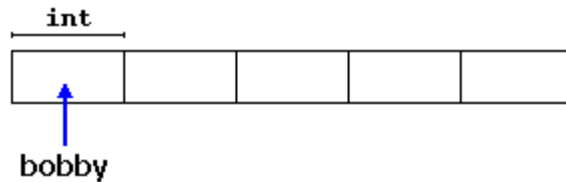
hoặc

```
pointer = new type [elements]
```

Biểu thức đầu tiên được dùng để cấp phát bộ nhớ chứa một phần tử có kiểu *type*. Lệnh thứ hai được dùng để cấp phát một khối nhớ (một mảng) gồm các phần tử kiểu *type*. Ví dụ:

```
int * bobby;
bobby = new int [5];
```

trong trường hợp này, hệ điều hành dành chỗ cho 5 phần tử kiểu `int` trong bộ nhớ và trả về một con trỏ trỏ đến đầu của khối nhớ. Vì vậy lúc này `bobby` trỏ đến một khối nhớ hợp lệ gồm 5 phần tử `int`.



Bạn có thể hỏi tôi là có gì khác nhau giữa việc khai báo một mảng với việc cấp phát bộ nhớ cho một con trỏ như chúng ta vừa làm. Điều quan trọng nhất là kích thước của một mảng phải là một hằng, điều này giới hạn kích thước của mảng đến kích thước mà chúng ta chọn khi thiết kế chương trình trong khi đó cấp phát bộ nhớ động cho phép cấp phát bộ nhớ trong quá trình chạy với kích thước bất kì.

Bộ nhớ động nói chung được quản lý bởi hệ điều hành và trong các môi trường đa nhiệm có thể chạy một lúc vài chương trình có một khả năng có thể xảy ra là hết bộ nhớ để cấp phát. Nếu điều này xảy ra và hệ điều hành không thể cấp phát bộ nhớ như chúng ta yêu cầu với toán tử `new`, một con trỏ null (zero) sẽ được trả về. Vì vậy các bạn nên kiểm tra xem con trỏ trả về bởi toán tử `new` có bằng null hay không:

```
int * bobby;
bobby = new int [5];
if (bobby == NULL) {
    // error assigning memory. Take measures.
};
```

Toán tử `delete`.

Vì bộ nhớ động chỉ cần thiết trong một khoảng thời gian nhất định, khi nó không cần dùng đến nữa thì nó sẽ được giải phóng để có thể cấp phát cho các nhu cầu khác trong tương lai. Để thực hiện việc này ta dùng toán tử `delete`, dạng thức của nó như sau:

```
delete pointer;
```

hoặc

```
delete [] pointer;
```

Biểu thức đầu tiên nên được dùng để giải phóng bộ nhớ được cấp phát cho một phần tử và lệnh thứ hai dùng để giải phóng một khối nhớ gồm nhiều phần tử (mảng). Trong hầu hết các trình dịch cả hai biểu thức là tương đương mặc dù chúng là rõ ràng là hai toán tử khác nhau.

```
// rememb-o-matic
#include <iostream.h>
#include <stdlib.h>
```

```
How many numbers do you want to
type in? 5
Enter number : 75
```

```

int main ()
{
    char input [100];
    int i,n;
    long * l, total = 0;
    cout << "How many numbers do you
want to type in? ";
    cin.getline (input,100); i=atoi
(input);
    l= new long[i];
    if (l == NULL) exit (1);
    for (n=0; n<i; n++)
    {
        cout << "Enter number: ";
        cin.getline (input,100);
        l[n]=atol (input);
    }
    cout << "You have entered: ";
    for (n=0; n<i; n++)
        cout << l[n] << ", ";
    delete[] l;
    return 0;
}

```

```

Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436, 1067, 8,
32,

```

NULL là một hằng số được định nghĩa trong thư viện C++ dùng để biểu thị con trỏ null. Trong trường hợp hằng số này chưa định nghĩa bạn có thể tự định nghĩa nó:

```
#define NULL 0
```

Dùng 0 hay **NULL** khi kiểm tra con trỏ là như nhau nhưng việc dùng **NULL** với con trỏ được sử dụng rất rộng rãi và điều này được khuyến khích để giúp cho chương trình dễ đọc hơn.

Bộ nhớ động trong ANSI-C

Toán tử *new* và *delete* là độc quyền C++ và chúng không có trong ngôn ngữ C. Trong ngôn ngữ C, để có thể sử dụng bộ nhớ động chúng ta phải sử dụng thư viện **stdlib.h**. Chúng ta sẽ xem xét cách này vì nó cũng hợp lệ trong C++ và nó vẫn còn được sử dụng trong một số chương trình.

Hàm *malloc*

Đây là một hàm tổng quát để cấp phát bộ nhớ động cho con trỏ. Cấu trúc của nó như sau:

```
void * malloc (size_t nbytes);
```

trong đó *nbytes* là số byte chúng ta muốn gán cho con trỏ. Hàm này trả về một con trỏ kiểu `void*`, vì vậy chúng ta phải chuyển đổi kiểu sang kiểu của con trỏ đích, ví dụ:

```
char * ronny;  
ronny = (char *) malloc (10);
```

Đoạn mã này cấp phát cho con trỏ `ronny` một khối nhớ 10 byte. Khi chúng ta muốn cấp phát một khối dữ liệu có kiểu khác `char` (lớn hơn 1 byte) chúng ta phải nhân số phần tử mong muốn với kích thước của chúng. Thật may mắn là chúng ta có toán tử `sizeof`, toán tử này trả về kích thước của một kiểu dữ liệu cụ thể.

```
int * bobby;  
bobby = (int *) malloc (5 * sizeof(int));
```

Đoạn mã này cấp phát cho `bobby` một khối nhớ gồm 5 số nguyên kiểu `int`, kích cỡ của kiểu dữ liệu này có thể bằng 2, 4 hay hơn tùy thuộc vào hệ thống mà chương trình được dịch.

Hàm *calloc*.

calloc hoạt động rất giống với *malloc*, sự khác nhau chủ yếu là khai báo mẫu của nó:

```
void * calloc (size_t nelements, size_t size);
```

nó sử dụng hai tham số thay vì một. Hai tham số này được nhân với nhau để có được kích thước tổng cộng của khối nhớ cần cấp phát. Thông thường tham số đầu tiên (*nelements*) là số phần tử và tham số thứ hai (*size*) là kích thước của mỗi phần tử. Ví dụ, chúng ta có thể định nghĩa `bobby` với *calloc* như sau:

```
int * bobby;  
bobby = (int *) calloc (5, sizeof(int));
```

Một điểm khác nhau nữa giữa *malloc* và *calloc* là *calloc* khởi tạo tất cả các phần tử của nó về 0.

Hàm *realloc*.

Nó thay đổi kích thước của khối nhớ đã được cấp phát cho một con trỏ.

```
void * realloc (void * pointer, size_t size);
```

tham số *pointer* nhận vào một con trỏ đã được cấp phát bộ nhớ hay một con trỏ null, và *size* chỉ định kích thước của khối nhớ mới. Hàm này sẽ cấp phát *size* byte bộ nhớ cho con trỏ. Nó có thể phải thay đổi vị trí của khối nhớ để có thể đủ chỗ cho kích thước mới của khối nhớ, trong trường hợp này nội dung hiện thời của khối nhớ được copy tới vị trí mới để đảm bảo dữ liệu không bị mất. Con trỏ mới trỏ tới khối nhớ được hàm trả về. Nếu không thể thay đổi kích thước của khối nhớ thì hàm sẽ trả về một con trỏ null nhưng tham số *pointer* và nội dung của nó sẽ không bị thay đổi.

Hàm *free*.

Hàm này giải phóng một khối nhớ động đã được cấp phát bởi *malloc*, *calloc* hoặc *realloc*.

```
void free (void * pointer);
```

Hàm này chỉ được dùng để giải phóng bộ nhớ được cấp phát bởi các hàm *malloc*, *calloc* and *realloc*.

Bài 11 : Các Cấu Trúc

Các cấu trúc dữ liệu.

Một cấu trúc dữ liệu là một tập hợp của những kiểu dữ liệu khác nhau được gộp lại với một cái tên duy nhất. Dạng thức của nó như sau:

```
struct model_name {  
    type1 element1;  
    type2 element2;  
    type3 element3;  
    .  
    .  
} object_name;
```

trong đó *model_name* là tên của mẫu kiểu dữ liệu và tham số tùy chọn *object_name* một tên hợp lệ cho đối tượng. Bên trong cặp ngoặc nhọn là tên các phần tử của cấu trúc và kiểu của chúng.

Nếu định nghĩa của cấu trúc bao gồm tham số *model_name* (tùy chọn), tham số này trở thành một tên kiểu hợp lệ tương đương với cấu trúc. Ví dụ:

```
struct products {  
    char name [30];  
    float price;  
} ;  
  
products apple;  
products orange, melon;
```

Chúng ta đã định nghĩa cấu trúc **products** với hai trường: **name** và **price**, mỗi trường có một kiểu khác nhau. Chúng ta cũng đã sử dụng tên của kiểu cấu trúc (**products**) để khai báo ba đối tượng có kiểu đó : **apple**, **orange** và **melon**.

Sau khi được khai báo, **products** trở thành một tên kiểu hợp lệ giống các kiểu cơ bản như *int*, *char* hay *short*.

Trường tùy chọn *object_name* có thể nằm ở cuối của phần khai báo cấu trúc dùng để khai báo trực tiếp đối tượng có kiểu cấu trúc. Ví dụ, để khai báo các đối tượng **apple**, **orange** và **melon** như đã làm ở phần trước chúng ta cũng có thể làm theo cách sau:

```
struct products {  
    char name [30];  
    float price;
```



```
} apple, orange, melon;
```

Hơn nữa, trong trường hợp này tham số `model_name` trở thành tùy chọn. Mặc dù nếu `model_name` không được sử dụng thì chúng ta sẽ không thể khai báo thêm các đối tượng có kiểu mẫu này.

Một điều quan trọng là cần phân biệt rõ ràng đâu là **kiểu mẫu** cấu trúc, đâu là **đối tượng** cấu trúc. Nếu dùng các thuật ngữ chúng ta đã sử dụng với các biến, kiểu mẫu là tên kiểu dữ liệu còn đối tượng là các biến.

Sau khi đã khai báo ba đối tượng có kiểu là một mẫu cấu trúc xác định (**apple**, **orange** and **melon**) chúng ta có thể thao tác với các trường tạo nên chúng. Để làm việc này chúng ta sử dụng một dấu chấm (.) chèn ở giữa tên đối tượng và tên trường. Ví dụ, chúng ta có thể thao tác với bất kì phần tử nào của cấu trúc như là đối với các biến chuẩn :

```
apple.name  
apple.price  
orange.name  
orange.price  
melon.name  
melon.price
```

mỗi trường có kiểu dữ liệu tương ứng: **apple.name**, **orange.name** và **melon.name** có kiểu **char[30]**, và **apple.price**, **orange.price** và **melon.price** có kiểu **float**.

Chúng ta tạm biệt apples, oranges và melons để đến với một ví dụ về các bộ phim:

```
// example about structures  
#include <iostream.h>  
#include <string.h>  
#include <stdlib.h>  
  
struct movies_t {  
    char title [50];  
    int year;  
} mine, yours;  
  
void printmovie (movies_t movie);  
  
int main ()  
{  
    char buffer [50];  
  
    strcpy (mine.title, "2001 A Space  
Odyssey");  
    mine.year = 1968;  
  
    cout << "Enter title: ";  
    cin.getline (yours.title, 50);  
    cout << "Enter year: ";  
    cin.getline (buffer, 50);
```

```
Enter title: Alien  
Enter year: 1979  
  
My favourite movie is:  
    2001 A Space Odyssey (1968)  
And yours:  
    Alien (1979)
```

```

    yours.year = atoi (buffer);

    cout << "My favourite movie is:\n
";
    printmovie (mine);
    cout << "And yours:\n ";
    printmovie (yours);
    return 0;
}

void printmovie (movies_t movie)
{
    cout << movie.title;
    cout << " (" << movie.year <<
    ") \n";
}

```

Ví dụ này cho chúng ta thấy cách sử dụng các phần tử của một cấu trúc và bản thân cấu trúc như là các biến thông thường. Ví dụ, **yours.year** là một biến hợp lệ có kiểu **int** cũng như **mine.title** là một mảng hợp lệ với 50 phần tử kiểu *chars*.

Chú ý rằng cả **mine** and **yours** đều được coi là các biến hợp lệ kiểu **movie_t** khi được truyền cho hàm **printmovie()**. Hơn nữa một lợi thế quan trọng của cấu trúc là chúng ta có thể xét các phần tử của chúng một cách riêng biệt hoặc toàn bộ cấu trúc như là một khối.

Các cấu trúc được sử dụng rất nhiều để xây dựng cơ sở dữ liệu đặc biệt nếu chúng ta xét đến khả năng xây dựng các mảng của chúng.

```

// array of structures
#include <iostream.h>
#include <stdlib.h>

#define N_MOVIES 5

struct movies_t {
    char title [50];
    int year;
} films [N_MOVIES];

void printmovie (movies_t movie);

int main ()
{
    char buffer [50];
    int n;
    for (n=0; n<N_MOVIES; n++)
    {
        cout << "Enter title: ";
        cin.getline

```

```

Enter title: Alien
Enter year: 1979
Enter title: Blade Runner
Enter year: 1982
Enter title: Matrix
Enter year: 1999
Enter title: Rear Window
Enter year: 1954
Enter title: Taxi Driver
Enter year: 1975

```

```

You have entered these movies:
Alien (1979)
Blade Runner (1982)
Matrix (1999)
Rear Window (1954)
Taxi Driver (1975)

```

```

(films[n].title,50);
    cout << "Enter year: ";
    cin.getline (buffer,50);
    films[n].year = atoi (buffer);
}
    cout << "\nYou have entered these
movies:\n";
    for (n=0; n<N_MOVIES; n++)
        printmovie (films[n]);
    return 0;
}

void printmovie (movies_t movie)
{
    cout << movie.title;
    cout << " (" << movie.year <<
")\n";
}

```

Con trỏ trỏ đến cấu trúc

Như bất kì các kiểu dữ liệu nào khác, các cấu trúc có thể được trỏ đến bởi con trỏ. Quy tắc hoàn toàn giống như đối với bất kì kiểu dữ liệu cơ bản nào:

```

struct movies_t {
    char title [50];
    int year;
};

movies_t amovie;
movies_t * pmovie;

```

Ở đây **amovie** là một đối tượng có kiểu **movies_t** và **pmovie** là một con trỏ trỏ tới đối tượng **movies_t**. OK, bây giờ chúng ta sẽ đến với một ví dụ khác, nó sẽ giới thiệu một toán tử mới:

```

// pointers to structures
#include <iostream.h>
#include <stdlib.h>

struct movies_t {
    char title [50];
    int year;
};

int main ()
{
    char buffer[50];

```

```

Enter title: Matrix
Enter year: 1999

You have entered:
Matrix (1999)

```

```

movies_t amovie;
movies_t * pmovie;
pmovie = & amovie;

cout << "Enter title: ";
cin.getline (pmovie->title, 50);
cout << "Enter year: ";
cin.getline (buffer, 50);
pmovie->year = atoi (buffer);

cout << "\nYou have entered:\n";
cout << pmovie->title;
cout << " (" << pmovie->year <<
")\n";

return 0;
}

```

Đoạn mã trên giới thiệu một điều quan trọng: toán tử `->`. Đây là một toán tử tham chiếu chỉ dùng để trở tới các cấu trúc và các lớp (class). Nó cho phép chúng ta không phải dùng ngoặc mỗi khi tham chiếu đến một phần tử của cấu trúc. Trong ví dụ này chúng ta sử dụng:

```
movies->title
```

nó có thể được dịch thành:

```
(*movies).title
```

cả hai biểu thức `movies->title` và `(*movies).title` đều hợp lệ và chúng đều dùng để tham chiếu đến phần tử `title` của cấu trúc được trả bởi `movies`. Bạn cần phân biệt rõ ràng với:

```
*movies.title
```

nó tương đương với

```
*(movies.title)
```

lệnh này dùng để tính toán giá trị được trả bởi phần tử `title` của cấu trúc `movies`, trong trường hợp này (title không phải là một con trỏ) nó chẳng có ý nghĩa gì nhiều. Bản dưới đây tổng kết tất cả các kết hợp có thể được giữa con trỏ và cấu trúc:

Biểu thức	Mô tả	Tương đương với
<code>movies.title</code>	Phần tử <code>title</code> của cấu trúc <code>movies</code>	
<code>movies->title</code>	Phần tử <code>title</code> của cấu trúc được trả bởi	<code>(*movies).title</code>

	movies	
*movies.title	Giá trị được trả bởi phần tử title của cấu trúc movies	*(movies.title)

Các cấu trúc lồng nhau

Các cấu trúc có thể được đặt lồng nhau vì vậy một phần tử hợp lệ của một cấu trúc có thể là một cấu trúc khác.

```
struct movies_t {
    char title [50];
    int year;
}

struct friends_t {
    char name [50];
    char email [50];
    movies_t favourite_movie;
} charlie, maria;

friends_t * pfriends = &charlie;
```

Vì vậy, sau phần khai báo trên chúng ta có thể sử dụng các biểu thức sau:

```
charlie.name
maria.favourite_movie.title
charlie.favourite_movie.year
pfriends->favourite_movie.year
```

(trong đó hai biểu thức cuối cùng là tương đương).

Các khái niệm cơ bản về cấu trúc được đề cập đến trong phần này là hoàn toàn giống với ngôn ngữ C, tuy nhiên trong C++, cấu trúc đã được mở rộng thêm các chức năng của một lớp với tính chất đặc trưng là tất cả các phần tử của nó đều là công cộng (public). Bạn sẽ có thêm các thông tin chi tiết trong phần [4.1, Lớp](#).

Bài 12: Các Kiểu Dữ Liệu Do Người Dùng Định Nghĩa

Trong bài trước chúng ta đã xem xét một loại dữ liệu được định nghĩa bởi người dùng (người lập trình): cấu trúc. Nhưng có còn nhiều kiểu dữ liệu tự định nghĩa khác:

Tự định nghĩa các kiểu dữ liệu (**typedef**).

C++ cho phép chúng ta định nghĩa các kiểu dữ liệu của riêng mình dựa trên các kiểu dữ liệu đã có. Để có thể làm việc đó chúng ta sẽ sử dụng từ khoá **typedef**, dạng thức như sau:

```
typedef    existing_type    new_type_name ;
```

trong đó *existing_type* là một kiểu dữ liệu cơ bản hay bất kì một kiểu dữ liệu đã định nghĩa và *new_type_name* là tên của kiểu dữ liệu mới. Ví dụ

```
typedef char C;
typedef unsigned int WORD;
typedef char * string_t;
typedef char field [50];
```

Trong trường hợp này chúng ta đã định nghĩa bốn kiểu dữ liệu mới: **C**, **WORD**, **string_t** và **field** kiểu **char**, **unsigned int**, **char*** kiểu **char[50]**, chúng ta hoàn toàn có thể sử dụng chúng như là các kiểu dữ liệu hợp lệ:

```
C achar, anotherchar, *ptchar1;
WORD myword;
string_t ptchar2;
field name;
```

typedef có thể hữu dụng khi bạn muốn định nghĩa một kiểu dữ liệu được dùng lặp đi lặp lại trong chương trình hoặc kiểu dữ liệu bạn muốn dùng có tên quá dài và bạn muốn nó có tên ngắn hơn.

Union

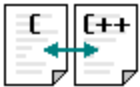
Union cho phép một phần bộ nhớ có thể được truy xuất dưới dạng nhiều kiểu dữ liệu khác nhau mặc dù tất cả chúng đều nằm cùng một vị trí trong bộ nhớ. Phần khai báo và sử dụng nó tương tự với cấu trúc nhưng chức năng thì khác hoàn toàn:

```
union model_name {
    type1 element1;
    type2 element2;
    type3 element3;
    .
    .
} object_name;
```

Tất cả các phần tử của *union* đều chiếm cùng một chỗ trong bộ nhớ. Kích thước của nó là kích thước của phần tử lớn nhất. Ví dụ:

```
union mytypes_t {
    char c;
    int i;
    float f;
} mytypes;
```

định nghĩa ba phần tử



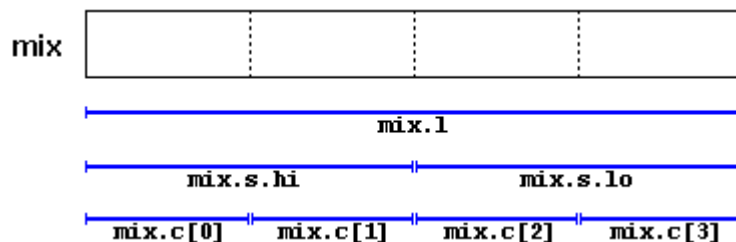
```
mytypes.c
mytypes.i
mytypes.f
```

mỗi phần tử có một kiểu dữ liệu khác nhau. Nhưng vì tất cả chúng đều nằm cùng một chỗ trong bộ nhớ nên bất kì sự thay đổi nào đối với một phần tử sẽ ảnh hưởng tới tất cả các thành phần còn lại.

Một trong những công dụng của *union* là dùng để kết hợp một kiểu dữ liệu cơ bản với một mảng hay các cấu trúc gồm các phần tử nhỏ hơn. Ví dụ:

```
union mix_t{
    long l;
    struct {
        short hi;
        short lo;
    } s;
    char c[4];
} mix;
```

định nghĩa ba phần tử cho phép chúng ta truy xuất đến cùng một nhóm 4 byte: **mix.l**, **mix.s** và **mix.c** mà chúng ta có thể sử dụng tùy theo việc chúng ta muốn truy xuất đến nhóm 4 byte này như thế nào. Tôi dùng nhiều kiểu dữ liệu khác nhau, mảng và cấu trúc trong union để bạn có thể thấy các cách khác nhau mà chúng ta có thể truy xuất dữ liệu.



Các unions vô danh

Trong C++ chúng ta có thể sử dụng các unions vô danh. Nếu chúng ta đặt một union trong một cấu trúc mà không đề tên (phần đi sau cặp ngoặc nhọn { }) union sẽ trở thành vô danh và chúng ta có thể truy xuất trực tiếp đến các phần tử của nó mà không cần đến tên của union (có cần cũng không được). Ví dụ, hãy xem xét sự khác biệt giữa hai phần khai báo sau đây:

union

```
struct {
    char title[50];
    char author[50];
    union {
        float dollars;
```

union vô danh

```
struct {
    char title[50];
    char author[50];
    union {
        float dollars;
```

```
int yens;  
} price;  
} book;
```

```
int yens;  
};  
} book;
```

Sự khác biệt duy nhất giữa hai đoạn mã này là trong đoạn mã đầu tiên chúng ta đặt tên cho union (**price**) còn trong cái thứ hai thì không. Khi truy nhập vào các phần tử **dollars** và **yens**, trong trường hợp thứ nhất chúng ta viết:

```
book.price.dollars  
book.price.yens
```

còn trong trường hợp thứ hai:

```
book.dollars  
book.yens
```

Một lần nữa tôi nhắc lại rằng vì nó là một union, hai trường **dollars** và **yens** đều chiếm cùng một chỗ trong bộ nhớ nên chúng không thể giữ hai giá trị khác nhau.

Kiểu liệt kê (**enum**)

Kiểu dữ liệu liệt kê dùng để tạo ra các kiểu dữ liệu chứa một cái gì đó hơi đặc biệt một chút, không phải kiểu số hay kiểu kí tự hoặc các hằng **true** và **false**. Dạng thức của nó như sau:

```
enum model_name {  
    value1,  
    value2,  
    value3,  
    .  
    .  
} object_name;
```

Ví dụ, chúng ta có thể tạo ra một kiểu dữ liệu mới có tên **color** để lưu trữ các màu với phần khai báo như sau:

```
enum colors_t {black, blue, green, cyan, red, purple,  
yellow, white};
```

Chú ý rằng chúng ta không sử dụng bất kì một kiểu dữ liệu cơ bản nào trong phần khai báo. Chúng ta đã tạo ra một kiểu dữ liệu mới mà không dựa trên bất kì kiểu dữ liệu nào có sẵn: kiểu **color_t**, những giá trị có thể của kiểu **color_t** được viết trong cặp ngoặc nhọn {}. Ví dụ, sau khi khai báo kiểu liệt kê, biểu thức sau sẽ là hợp lệ:

```
colors_t mycolor;  
  
mycolor = blue;  
if (mycolor == green) mycolor = red;
```


Trên thực tế kiểu dữ liệu liệt kê được dịch là một số nguyên và các giá trị của nó là các hằng số nguyên được chỉ định. Nếu điều này không được chỉ định, giá trị nguyên tương đương với phần tử đầu tiên là 0 và các giá trị tiếp theo cứ thế tăng lên 1. Vì vậy, trong kiểu dữ liệu `colors_t` mà chúng ta định nghĩa ở trên, **white** tương đương với 0, **blue** tương đương với 1, **green** tương đương với 2 và cứ tiếp tục như thế.

Nếu chúng ta chỉ định một giá trị nguyên cho một giá trị nào đó của kiểu dữ liệu liệt kê (trong ví dụ này là phần tử đầu tiên) các giá trị tiếp theo sẽ là các giá trị nguyên tiếp theo, ví dụ:

```
enum months_t { january=1, february, march, april,
                may, june, july, august,
                september, october, november, december}
y2k;
```

trong trường hợp này, biến `y2k` có kiểu dữ liệu liệt kê `months_t` có thể chứa một trong 12 giá trị từ **january** đến **december** và tương đương với các giá trị nguyên từ 1 đến 12, không phải 0 đến 11 vì chúng ta đã đặt **january** bằng 1

Nguồn internet