Bài 1.1: 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 chút 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 xổ (//) 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 1.2: Các biến, kiểu và hằng số

Identifiers

Một tên (indentifiers) hợp lệ là một chuỗi gồm các chữ cái, chữ số hoặc kí tự gạch dưới. Chiều dài của một tên là không giới hạn.

Kí tự trống, các kí tự đánh dấu đều không thể có mặt trong một tên. Chỉ có **chữ cái, chữ số và kí tự gạch dưới** là được cho phép. Thêm vào đó, một tên biến luôn phải **bắt đầu bằng một chữ cái.** Chúng cũng có thể bắt đầu bằng kí tự gạch dưới (_) nhưng kí tự này thường được dành cho các liên kết bên ngoài (external link). Không bao giờ chúng bắt đầu bằng một chữ số.

Một luật nữa mà bạn phải quan tâm đến khi tạo ra các tên của riêng mình là chúng không được trùng **với bất kì từ khoá nào** của ngôn ngữ hay của trình dịch, ví dụ các tên sau đây luôn luôn được coi là từ khoá theo chuẩn ANSI-C++ và do vậy chúng không thể được dùng để đặt tên

```
asm, car, bool, break, marry, catch, to char, class, const,
const_cast,  continue,  default,  delete,  do,  double,
dynamic_cast, else, enum, explicit, extern, false, float,
for,  friend,  goto,  if,  inline,  int,  long,  mutable,
namespace, new, operator, private, protected, public, to
register, reinterpret_cast, return, short, signed, sizeof,
static, static_cast, struct, switch, template, this, throw,
true,  try,  typedef,  typeid,  typename,  union,  unsigned,
using, virtual, void, volatile, wchar_t
```

Thêm vào đó, một số biểu diễn khác của các toán tử (operator) cũng không được dùng làm tên vì chúng là những từ được dành riêng trong một số trường hợp.

```
and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq,
xor, xor_eq
```

Trình dịch của bạn có thể thêm một từ dành riêng đặc trưng khác. Ví dụ, rất nhiều trình dịch 16 bit (như các trình dịch cho DOS) còn có thể các từ khoá **far**, **huge** và **near**.

Chú ý: Ngôn ngữ C++ là **"case sensitive"** có nghĩa là phân biệt chữ hoa chữ thường. Do vậy biến **RESULT** khác với **result** cũng như **Result**.

Các kiểu dữ liệu

Khi lập trình, chúng ta lưu trữ các biến trong bộ nhớ của máy tính nhưng máy tính phải biết chúng ta muốn lưu trữ gì trong chúng vì các kiểu dữ liệu khác nhau sẽ cần lượng bộ nhớ khác nhau.

Bộ nhớ của máy tính chúng ta được tổ chức thành các byte. Một byte là lượng bộ nhớ nhỏ nhất mà chúng ta có thể quản lí. Một byte có thể dùng để lưu trữ một loại dữ liệu nhỏ như là kiểu số nguyên từ 0 đến 255 hay một kí tự. Nhưng máy tính có thể xử lý các kiểu dữ liệu phức tạp hơn bằng cách gộp nhiều byte lại với nhau, như số nguyên dài hay số thập phân. Tiếp theo bạn sẽ có một danh sách các kiểu dữ liệu cơ bản trong C++ cũng như miền giá trị mà chúng có thể biểu diễn

Tên	Số byte	Mô tả	Miền giá trị
char	1	Kí tự hay kiểu số nguyên 8-bit	**có dấu:** -128 to 127 **không dấu:** 0 to 255
short	2	kiểu số nguyên 16-bit	**có dấu:** -32763 to 32762 **không dấu:** 0 to 65535
long	4	kiểu số nguyên 32-bit	**có dấu:**-2147483648 to 2147483647 **không dấu:** 0 to 4294967295
int	*	Số nguyên. Độ dài của nó phụ thuộc vào hệ thống, như trong MS-DOS nó là 16-bit, trên Windows 9x/2000/NT là 32 bit...	Xem **short**, **long**
float	4	Dạng dấu phẩy động	3.4e + / - 38 (7 digits)
double	8	Dạng dấu phẩy động với độ chính xác gấp đôi	1.7e + / - 308 (15 digits)
long double	10	Dạng dấu phẩy động với độ chính xác hơn nữa	1.2e + / - 4932 (19 digits)
bool	1	Giá trị logic. Nó mới được thêm vào chuẩn ANSI-C++. Bởi vậy không phải tất cả các trình dịch đều hỗ trợ nó.	`true` hoặc `false`

Ngoài các kiểu dữ liệu cơ bản nói trên còn tồn tại các con trỏ và các tham số không kiểu (void) mà chúng ta sẽ xem xét sau.

Khai báo một biến

Để có thể sử dụng một biến trong C++, đầu tiên chúng ta phải khai báo nó, ghi rõ nó là kiểu dữ liệu nào. Chúng ta chỉ cần viết tên kiểu (như **int**, **short**, **float**...) tiếp theo sau đó là một tên biến hợp lệ. Ví dụ

```
int                                                    a;
float mynumber;
```

Dòng đầu tiên khai báo một biến kiểu **int** với tên là **a.** Dòng thứ hai khai báo một biến kiểu **float** với tên **mynumber**. Sau khi được khai báo, các biến trên có thể được dùng trong **phạm vi của chúng** trong chương trình.

Nếu bạn muốn khai báo một vài biến có cùng một kiểu và bạn muốn tiết kiệm công sức viết bạn có thể khai báo chúng trên một dòng, ngăn cách các tên bằng dấu phẩy. Ví dụ

```
int a, b, c;
```

khai báo ba biến kiểu **int** (**a**,**b** và **c**) và hoàn toàn tương đương với :

```
int                                                    a;
int                                                    b;
int c;
```

Các kiểu số nguyên (**`char`**, **`short`**, **`long`** and **`int`**) có thể là số có dấu hay không dấu tuỳ theo miền giá trị mà chúng ta cần biểu diễn. Vì vậy khi xác định một kiểu số nguyên chúng ta đặt từ khoá **`signed`** hoặc **`unsigned`** trước tên kiểu dữ liệu. Ví dụ:

```
unsigned                short                NumberOfSons;
signed int MyAccountBalance;
```

Nếu ta không chỉ rõ **`signed`** or **`unsigned`** nó sẽ được coi là có dấu, vì vậy trong **khai báo thứ hai** chúng ta có thể viết :

```
int MyAccountBalance
```

cũng hoàn toàn tương đương với dòng khai báo ở trên. Trong thực tế, rất ít khi người ta dùng đến từ khoá **signed**. Ngoại lệ duy nhất của luật này kiểu **char**. Trong chuẩn ANSI-C++ nó là kiểu dữ liệu khác với **`signed char`** và **`unsigned char`**.

Để có thể thấy rõ hơn việc khai báo trong chương trình, chúng ta sẽ xem xét một đoạn mã C++ ví dụ như sau:

```
// operating with variables

#include <iostream.h>

int main ()
```

```
4
```

```
{
  // declaring variables:
  int a, b;
  int result;

  // process:
  a = 5;
  b = 2;
  a = a + 1;
  result = a - b;

  // print out the result:
  cout << result;

  // terminate the program:
  return 0;
}
```

Đừng lo lắng nếu như việc khai báo có vẻ hơi lạ lùng với bạn. Bạn sẽ thấy phần chi tiết còn lại trong phần tiếp theo

Khởi tạo các biến

Khi khai báo một biến, giá trị của nó mặc nhiên là không xác định. Nhưng có thể bạn sẽ muốn nó mang một giá trị xác định khi được khai báo. Để làm điều đó, bạn chỉ cần viết dấu bằng và giá trị bạn muốn biến đó sẽ mang:

```
type identifier = initial_value ;
```

Ví dụ, nếu chúng ta muốn khai báo một biến **int** là **a** chứa giá trị **0** ngay từ khi khởi tạo, chúng ta sẽ viết :

```
int a = 0;
```

Bổ xung vào cách khởi tạo kiểu C này, C++ còn có thêm một cách mới để khởi tạo biến bằng cách bọc một cặp ngoặc đơn sau giá trị khởi tạo. Ví dụ :

```
int a (0);
```

Cả hai cách đều hợp lệ trong C++.

Phạm vi hoạt động của các biến

Tất cả các biến mà chúng ta sẽ sử dụng đều phải được khai báo trước. Một điểm khác biết giữa Cvà C++ là trong C++ chúng ta **có thể khai báo biến ở bất kì nơi nào trong chương trình, thậm chí là ngay ở giữa các lệnh thực hiện chứ không chỉ là ở đầu khối lệnh như ở trong C.**

Mặc dù vậy chúng ta vẫn nên theo cách của ngôn ngữ C khi khai báo các biến bởi vì nó sẽ rất hữu dụng khi cần sửa chữa một chương trình có tất cả các phần khai báo được gộp lại với nhau. Bởi vậy, cách thông dụng nhất để khai báo biến là đặt nó trong phần bắt đầu của mỗi hàm (biến cục bộ) hay trực tiếp trong thân chương trình, ngoài tất cả các hàm (biến toàn cục).

Global variables (biến toàn cục) có thể được sử dụng ở bất kì đâu trong chương trình, ngay sau khi nó được khai báo.

Tầm hoạt động của **local variables** (biến cục bộ) bị giới hạn trong phần mã mà nó được khai báo. Nếu chúng được khai báo ở đầu một hàm (như hàm **main**), tầm hoạt động sẽ là toàn bộ hàm **main**. Điều đó có nghĩa là trong ví dụ trên, các biến được khai báo trong hàm main() chỉ có thể được dùng trong hàm đó, không được dùng ở bất kì đâu khác.

Thêm vào các biến toàn cục và cục bộ, còn có các **biến ngoài (external).** Các biến này không những được dùng trong một file mã nguồn mà còn trong tất cả các file được liên kết trong chương trình.

Trong C++ tầm hoạt động của một biến chính là khối lệnh mà nó được khai báo (một khối lệnh là một tập hợp các lệnh được gộp lại trong một bằng các ngoặc nhọn { }). Nếu nó được khai báo trong một hàm tầm hoạt động sẽ là hàm đó, còn nếu được khai báo trong vòng lặp thì tầm hoạt động sẽ chỉ là vòng lặp đó....

Các hằng số

Một hằng số là bất kì một biểu thức nào mang một giá trị cố định, như:

Các số nguyên

```
1776
707
-273
```

chúng là các hằng mang giá trị số. Chú ý rằng khi biểu diễn một hằng kiểu số chúng ta không cần viết dấu ngoặc kép hay bất kì dấu hiệu nào khác.

Thêm vào những số ở hệ cơ số 10 (cái mà tất cả chúng ta đều đã biết) **C++ còn cho phép sử dụng các hằng số cơ số 8 và 16. Để biểu diễn một số hệ cơ số 8 chúng ta đặt trước nó kí tự 0, để biểu diễn số ở hệ cơ số 16 chúng ta đặt trước nó hai kí tự 0x.** Ví dụ:

```
75      // Cơ số 10
0113    // cơ số 8
0x4b    // cơ số 16
```

Các số thập phân (dạng dấu phẩy động)

Chúng biểu diễn các số với phần thập phân và/hoặc số mũ. Chúng có thể bao gồm phần thập phân, kí tự **e** (biểu diễn 10 mũ...).

```
3.14159    // 3.14159
6.02e23    // 6.02 x 10^23
1.6e-19    // 1.6 x 10^-19
3.0        // 3.0
```

Kí tự và xâu kí tự

Trong C++ còn tồn tại các hằng không phải kiểu số như:

```
'z'
'p'
"Hello                                                world"
"How do you do?"
```

Hai biểu thức đầu tiên biểu diễn các kí tự đơn, các kí tự được đặt trong dấu nháy đơn (`'`), hai biểu thức tiếp theo biểu thức các xâu kí tự được đặt trong dấu nháy kép (`"`).

Khi viết các kí tự đơn hay các xâu kí tự cần phải đặ chúng trong các dấu nháy để phân biệt với các tên biến hay các từ khoá. Chú ý:

```
x
'x'
```

x trỏ đến biến x trong khi 'x' là kí tự hằng 'x'.

Các kí tự đơn và các xâu kí tự có một tính chất riêng biệt là các mã điều khiển. Chúng là những kí tự đặc biệt mà không thể được viết ở bất kì đâu khác trong chương trình như là mã xuống dòng (`\n`) hay *tab* (`\t`). Tất cả đều bắt đầu bằng dấu xổ ngược (`\`). Sau đây là danh sách các mã điều khiển đó:

\n	xuống dòng
\r	lùi về đầu dòng
\t	kí tự tab
\v	căn thẳng theo chiều dọc
\b	backspace
\f	sang trang
\a	Kêu bíp
\'	dấu nháy đơn
\"	dấu nháy kép
****	dấu hỏi
****	kí tự xổ ngược

Ví dụ:

```
'\n'
'\t'
"Left                    \t                    Right"
"one\ntwo\nthree"
```

Thêm vào đó, để biểu diễn một mã ASCII bạn cần sử dụng kí tự xổ ngược (\) tiếp theo đó là mã ASCII viết trong hệ cơ số 8 hay cơ số 16. Trong trường hợp đầu mã ASCII được viết ngay sau dấu sổ ngược, trong trường hợp thứ hai, để sử dụng số trong hệ cơ số 16 bạn cần viết kí tự **x** trước số đó (ví dụ **`\x20`** hay **`\x4A`**).

Các hằng chuỗi kí tự có thể được viết trên nhiều dòng nếu mỗi dòng được kết thúc bằng một dấu sổ ngược (\):

```
"string          expressed          in          \
two lines"
```

Bạn có thể nối một vài hằng xâu kí tự ngăn cách bằng một hay vài dấu trống, kí tự tab, xuống dòng hay bất kì kí tự trống nào khác.

"we form" "a unique" "string" "of characters"

Định nghĩa các hằng (*`#define`*)

Bạn có thể định nghĩa các hằng với tên mà bạn muốn để có thể sử dụng thường xuyên mà không mất tài nguyên cho các biến bằng cách sử dụng chỉ thị **`#define`**. Đây là dạng của nó:

```
#define identifier value
```

Ví dụ:

```
#define                PI                3.14159265
#define                NEWLINE                '\n'
#define WIDTH 100
```

chúng định nghĩa ba hằng số mới. Sau khi khai báo bạn có thể sử dụng chúng như bất kì các hằng số nào khác, ví dụ

```
circle        =        2        *        PI        *        r;
cout << NEWLINE;
```

Trong thực tế việc duy nhất mà trình dịch làm khi nó tìm thấy một chỉ thị **`#define`** là thay thế các tên hằng tại bất kì chỗ nào chúng xuất hiện (như trong ví dụ trước, **`PI`**, **`NEWLINE`** hay **`WIDTH`**) bằng giá trị mà chúng được định nghĩa. Vì vậy các hằng số `#define` được coi là các *hằng số macro*

Chỉ thị `#define` không phải là một lệnh thực thi, nó là chỉ thị tiền xử lý (preprocessor), đó là lý do trình dịch coi cả dòng là một chỉ thị và dòng đó không cần kết thúc bằng dấu chấm phẩy. Nếu bạn thêm dấu chấm phẩy vào cuối dòng, nó sẽ được coi là một phần của giá trị định nghĩa hằng.

Khai báo các hằng (const)

Với tiền tố **`const`** bạn có thể khai báo các hằng với một kiểu xác định như là bạn làm với một biến

```
const           int           width           =           100;
const        to        char        tab        =        '\t';
const zip = 12440;
```

Trong trường hợp kiểu không được chỉ rõ (như trong ví dụ cuối) trình dịch sẽ coi nó là kiểu int.

Bài 1.3: 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 **vế 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	**Ví dụ 2**
`B=3;`	`B=3;`
`A=++B;`	`A=B++;`
`// A is 4, B is 4`	`// 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`, tuỳ 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) lf 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 tuỳ 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 1.4: Giao tiếp với console.

Console là giao diện cơ bản của máy tính. Bàn phím là thiết bị vào cơ bản còn màn hình là thiết bị ra cơ bản.

Trong thư viện *iostream* của C++, các thao tác vào ra cơ bản của một chương trình được hỗ trợ bởi hai dòng dữ liệu : **`cin`** để nhập dữ liệu và **`cout`** để xuất. Thêm vào đó, còn có **`cerr`** và **`clog`** là hai dòng dữ liệu dùng để hiển thị các thông báo lỗi trên thiết bị ra chuẩn (thường là màn hình) hoặc ra một file. Thông thường **`cout`** được gán với màn hình còn **`cin`** được gán với bàn phím.

Sử dụng hai dòng dữ liệu này bạn sẽ có thể giao tiếp với người sử dụng vì bạn có thể hiển thị các thông báo lên màn hình cũng như nhận dữ liệu từ bàn phím.

Xuất dữ liệu (`cout`)

Dòng **`cout`** được sử dụng với toán tử đã quá tải << (overloaded - bạn sẽ hiểu rõ hơn về thuật ngữ này trong phần lập trình hướng đối tượng)

```
cout << "Output sentence"; // Hiển thị Output sentence lên màn hình
cout << 120;              // Hiển thị số 120 lên màn hình
cout << x;                // Hiển thị nội dung biến x lên màn hình
```

Toán tử << được gọi là toán tử chèn vì nó chèn dữ liệu đi sau nó vào dòng dữ liệu đứng trước. Trong ví dụ trên nó chèn chuỗi "`Output sentence`", hằng số 120 và biến x vào dòng dữ liệu ra **`cout`**.Chú ý rằng ở dòng đầu tiên chúng ta sử dụng dấu ngoặc kép vì đó là một chuỗi kí tự. Khi chúng ta muốn sử dụng các hằng xâu kí tự ta phải đặt chúng trong cặp dấu ngoặc kép để chúng có thể được phân biệt với các biến. Ví dụ, hai lệnh sau đây là hoàn toàn khác nhau:

```
cout << "Hello";    // Hiển thị Hello lên màn hình
cout << Hello;      // Hiển thị nội dung của biến Hello lên màn hình
```

Toán tử chèn (<<) có thể được sử dụng nhiều lần trong một câu lệnh:

```
cout << "Hello, " << "I am " << "a C++ sentence";
```

Câu lệnh trên sẽ in thông báo **`Hello, I am a C++ sentence`** lên màn hình. Sự tiện lợi của việc sử dụng lặp lại toán tử chèn (<<) thể hiện rõ khi chúng ta muốn hiển thị nhiều biến và hằng hơn là chỉ một biến:

```
cout << "Hello, I am " << age << " years old and my email address is " << email_add;
```

Cần phải nhấn mạnh rằng **`cout`** không nhảy xuống dòng sau khi xuất dữ liệu, vì vậy hai câu lệnh sau :

```
cout       <<       "This       is       a       sentence.";
cout << "This is another sentence.";
```

sẽ được hiển thị trên màn hình:

```
This is a sentence.This is another sentence.
```

Bởi vậy khi muốn xuống dòng chúng ta phải sử dụng kí tự xuống dòng, trong C++ là **`\n`**:

```
cout << "First sentence.\n ";
cout << "Second sentence.\nThird sentence.";
```

sẽ viết ra màn hình như sau:

```
First                                                    sentence.
Second                                                   sentence.
Third sentence.
```

Thêm vào đó, **để xuống dòng bạn có thể sử dụng tham số `endl`**. Ví dụ

```
cout << "First sentence." << endl;
cout << "Second sentence." << endl;
```

sẽ in ra màn hình:

```
First                                                    sentence.
Second sentence.
```

Tham số **`endl`** có một tác dụng đặc biệt khi nó được dùng với các dòng dữ liệu sử dụng bộ đệm: các bộ đệm sẽ được *flushed* (chuyển toàn bộ thông tin từ bộ đệm ra dòng dữ liệu). Tuy nhiên, theo mặc định **`cout`** không sử dụng bộ đệm.

Nhập dữ liệu (`cin`).

Thao tác vào chuẩn trong C++ được thực hiện bằng cách sử dụng toán tử đã quá tải `>>` với dòng **`cin`**. Theo sau toán tử này là biến sẽ lưu trữ dữ liệu được đọc vào. Ví dụ:

```
int age;
cin >> age;
```

khai báo biến `age` có kiểu `int` và đợi nhập dữ liệu từ `cin` (bàn phím) để lưu trữ nó trong biến kiểu nguyên này.

`cin` chỉ bắt đầu sử lý dữ liệu nhập từ bàn phím sau khi phím Enter được gõ. Vì vậy dù bạn chỉ nhập một kí tự thì **`cin`** vẫn sẽ kiên nhẫn chờ cho đến khi bạn gõ phím Enter.

```
// i/o example
#include <iostream.h>

int main ()
{
  int i;
   cout << "Please enter an integer
value: ";
  cin >> i;
   cout << "The value you entered is
" << i;
   cout << " and its double is " <<
i*2 << ".\n";
```

```
Please enter an integer value: 702
The value you entered is 702 and
its double is 1404.
```

```
  return 0;
}
```

Người sử dụng chương trình có thể là một trong những nguyên nhân gây ra lỗi trong một chương trình đơn giản sử dụng **cin** (như chương trình trên). Trong khi bạn muốn nhận một số nguyên thì người sử dụng lại nhập vào tên của họ (là một xâu kí tự). Kết quả là chương trình sẽ chạy sai vì đó không phải là những gì mà chương trình mong đợi từ người dùng. Bởi vậy khi bạn sử dụng dữ liệu nhập vào từ **cin** bạn phải tin chắc rằng người dùng sẽ hoàn toàn hợp tác và rằng anh ta sẽ không nhập tên của mình khi chương trình yêu cầu nhập số nguyên. Sau này, khi nghiên cứu việc sử dụng các xâu kí tự chúng ta sẽ xem xét các giải pháp khả thi để giải quyết các lỗi loại này.

Bạn có thể dùng **cin** để nhập một lúc nhiều dữ liệu từ người dùng:

```
cin >> a >> b;
```

tương đương với

```
cin >> a;
cin >> b;
```

Trong cả hai trường hợp người sử dụng phải cung cấp hai dữ liệu, một cho biến **a** và một cho biến **b** và được ngăn cách bởi một dấu trống hợp lệ: một dấu cách, dấu tab hay kí tự xuống dòng.

Trong trường hợp kiểu không được chỉ rõ (như trong ví dụ cuối) trình dịch sẽ coi nó là kiểu **int**.

Bài 2.1 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, tuỳ 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 thoả 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 sửu 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
#include <iostream.h>
int main ()
{
  int n;
   cout << "Enter the starting
number > ";
```

```
Enter the starting number > 8
8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

```
  cin >> n;
  while (n>0) {
    cout << n << ", ";
    --n;
  }
  cout << "FIRE!";
  return 0;
}
```

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

```
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
```

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 khí 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;
}
```

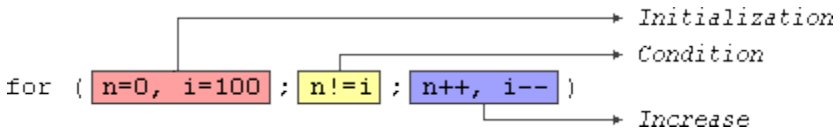
```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
FIRE!
```

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:



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: ;
```

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

```
  cout << n << ", ";
  n--;
  if (n>0) goto loop;
  cout << "FIRE!";
  return 0;
}
```

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** tuỳ 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.

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 2.2 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 tuỳ 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;
  return (r);
}

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

```
The result is 8
```

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 c 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 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 **r** 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 làm 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';
```

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

```
  return 0;
}
```

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 ();
```

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.

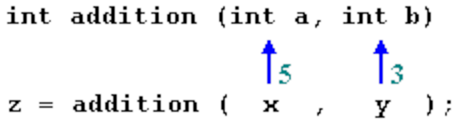
Bài 2.3 Hàm (II)

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.



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.

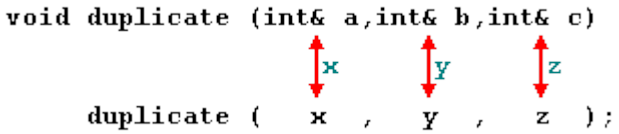
Hãy xét trường hợp bạn cần thao tác với một biến ngoài ở bên trong một hàm. Vì vậy bạn sẽ phải truyền tham số dưới dạng tham số biến như ở trong hàm **duplicate** trong ví dụ dưới đây:

```
// 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.



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;
```

```
6
5
```

```
  r=a/b;
  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 phần 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 cũng 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;
  float n=5.0,m=2.0;
  cout << divide (x,y);
```

```
2
2.5
```

```
  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 khao 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:

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

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);
}

int main ()
{
  long l;
  cout << "Type a number: ";
```

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

```
  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 ()
{
```

```
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.
```

```
  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);
}
```

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);
```

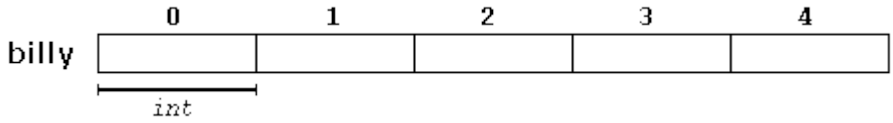
cho phép hai hàm này có thể được sử dụng trước khi chúng được định nghĩa hoàn chỉnh. Tuy nhiên lý do đặc biệt giải thích tại sao chương trình này lại cần ít nhất một hàm phải được khi báo mẫu là trong **odd** có một lời gọi đến **even** và trong **even** có một lời gọi đến **odd**. Vì vậy nếu không có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

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 3.1 **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.

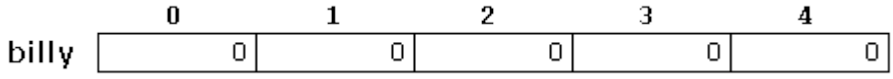
Khởi tạo một mảng.

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 ra 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];
```

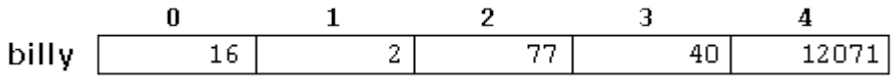
mọi phần tử của *billy* sẽ được khởi tạo là **0**:



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:



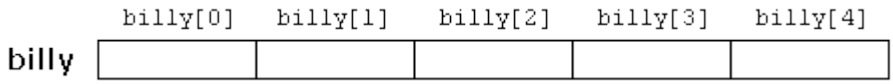
Số phần tử trong mảng mà chúng ta khởi tạo với cặp ngoặc nhọn `{ }` phải bằng số phần tử của mảng đã được khai báo với cặp ngoặc vuông `[ ]`. Bởi vì điều này có thể được coi là một sự lặp lại không cần thiết nên C++ cho phép để trống giữa cặp ngoặc vuông, kích thước của mảng được xác định bằng số giá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



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:

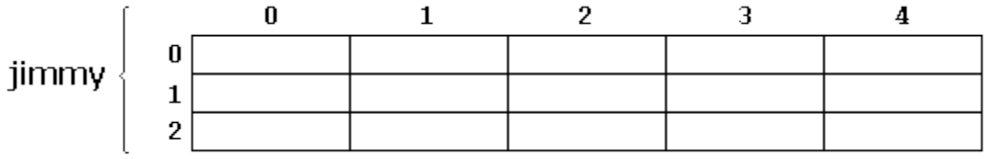
```
billy[0]                          =                          a;
billy[a]                         =                         75;
b                 =                 billy                [a+2];
billy[billy[a]] = billy[2] + 5;
```

```
// 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.

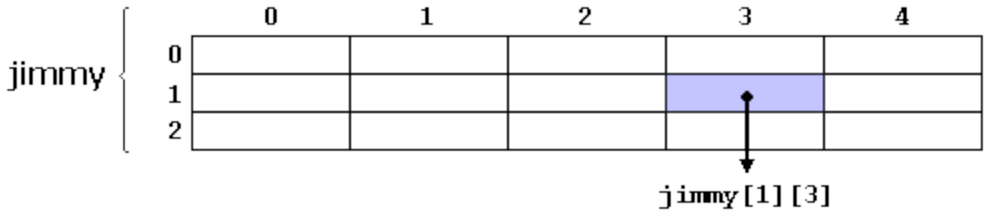


`jimmy` biểu diễn một mảng hai chiều kích thước 3x5 có kiểu **`int`**. Cách khai báo mảng này như sau:

```
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:

`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 nhiều chỉ số tùy thích mặc dù ít khí 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;
```

```
// 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][m]=(n+1)*(m+1);
    }
  return 0;
}
```

```
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";
}

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

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

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.

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 3.2 Xâu kí tự

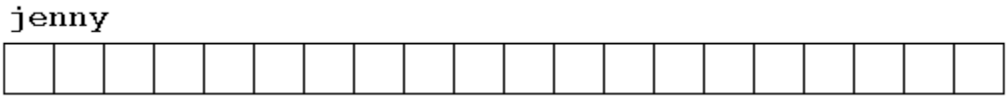
Trong tất cả các chương trình chúng ta đã thấy cho đến giờ, chúng ta chỉ sử dụng các biến kiểu số, chỉ dùng để biểu diễn các số. Nhưng bên cạnh các biến kiểu số còn có các xâu kí tự, chúng cho phép chúng ta biểu diễn các chuỗi kí tự như là các từ, câu, đoạn văn bản... Cho đến giờ chúng ta mới chỉ dùng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

Trong C++ không có kiểu dữ liệu *cơ bản* để lưu các xâu kí 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 kí tự đơn, bởi vậy nó được dùng để tạo ra xâu của các kí tự đơn.

Ví dụ, mảng sau (hay là xâu kí tự):

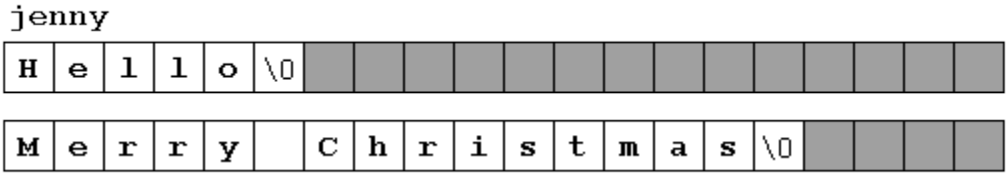
```
char jenny [20];
```

có thể lưu một xâu kí tự với độ dài cực đại là 20 kí 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 xâu `"Hello"` hay `"Merry christmas"`. Vì các mảng kí tự có thể lưu các xâu kí tự ngắn hơn độ dài của nó, trong C++ đã có một quy ước để kết thúc một nội dung của một xâu kí tự bằng một kí 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 xâu kí tự `"Hello"` và `"Merry Christmas"` theo cách sau:



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

Khởi tạo các xâu kí tự.

Vì những xâu kí tự là những mảng bình thường nên chúng cũng như các mảng khác. Ví dụ, nếu chúng ta muốn khởi tạo một xâu kí 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 xâu kí tự bằng cách khác: sử dụng các *hằng xâu kí tự*.

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

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

là một hằng xâu kí 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 vế 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 vế 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** (**str**ing **c**o**py**), 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 xâu kí tự, bởi vậy lệnh sau đây là một cách đúng để gán xâu 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
```

Để 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;
```

```
J. Soulie
```

```
  return 0;
}
```

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 xâu kí tự được gán bởi người dùng trong quá trình chương trình thực hiện.

Khi **cin** được sử dụng với các xâu kí 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 kí tự được dùng để kết thúc việc nhập, mặc định - nếu chúng ta không dùng tham số này - sẽ là kí 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 xâu kí 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 xâu (**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 xâu kí tự thay cho **cin.getline**. Ví dụ, trong chươn 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 kí 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 xâu *string1* và *string2*. Trả về `0` nếu hai xâu 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 3.3 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:

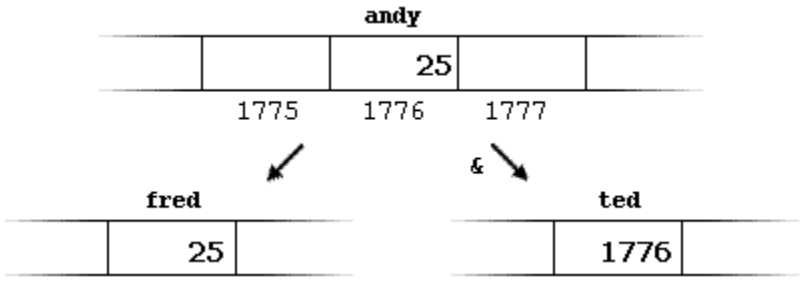
```
ted = &andy;
```

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**.

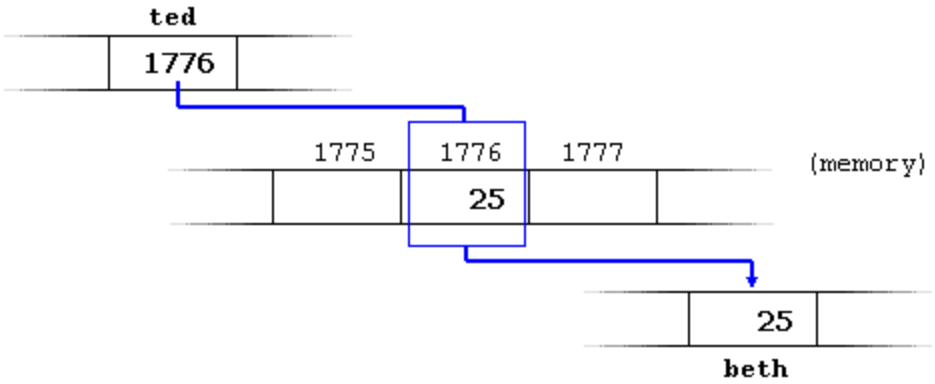
Những biến lưu trữ địa chỉ của một biến khác (như **ted** ở trong ví dụ trước) được gọi là **con trỏ**. Trong C++ con trỏ có rất nhiều ưu điểm và chúng được sử dụng rất thường xuyên, Tiếp theo chúng ta sẽ thấy các biến kiểu này được khai báo như thế nào.

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 đặ 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) trỏ 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 trỏ 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 trỏ bởi biểu thức được coi như là một địa chỉ. Nó có thể được dịch là "**giá trị được trỏ bởi**"..
`*mypointer` được đọc là *"giá trị được trỏ 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, <u>không phải</u> 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 đầu 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
```

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

```
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)
  *p1 = 20;          // giá trị trỏ
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.

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 cũng 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>

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;
}
```

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

Trong bài "mảng" chúng ta đã dùng dấu ngoặc vuông để chỉ ra phần tử của mảng mà chúng ta muốn trỏ 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;
```

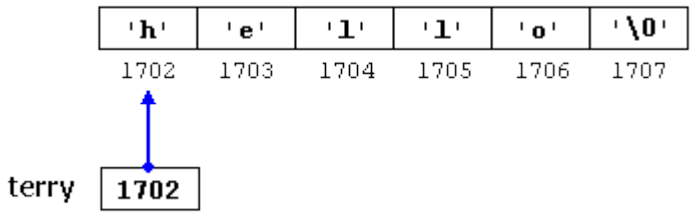
Trong một phép gán con trỏ chúng ta phải luôn luôn gán địa chỉ mà nó trỏ tới chứ không phải là giá trị mà nó trỏ tới. Bạn cần phải nhớ rằng khi khai báo một biến con trỏ, dấu sao (*) được dùng để chỉ ra nó là một con trỏ, và hoàn toàn khác với toán tử tham chiếu. Đó là hai toán tử khác nhau mặc dù chúng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

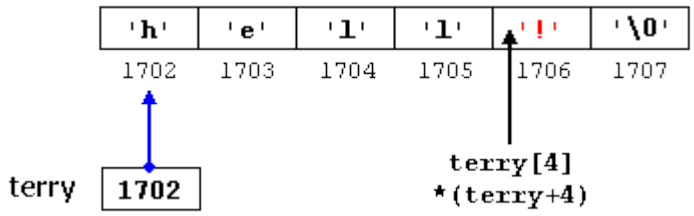


cần phải nhắc lại rằng **`terry`** mang giá trị **`1702`** chứ không phải là **`'h'`** hay **`"hello"`**.

Biến con trỏ **`terry`** trỏ tới một xâu 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 xâu 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:

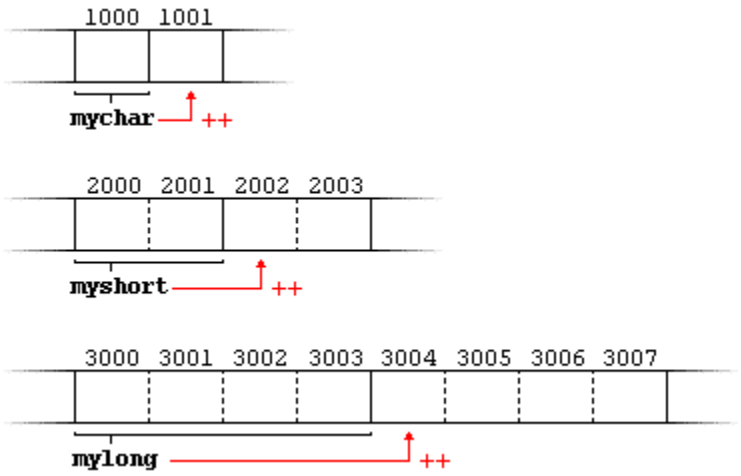
```
char *mychar;
short *myshort;
long *mylong;
```

và chúng lần lượt trỏ 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 trỏ 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 trỏ 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 trỏ tới chứ không phải là giá trị trỏ 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++;
```

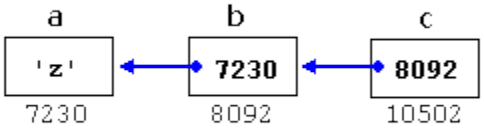
Như đã nói trong các bài trước, tôi khuyên các bạn nên dùng các cặp ngoặc đơn để tránh những kết quả không mong muốn.

Con trỏ trỏ 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 xâu kí 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                    6, 10, 13
```

```
#include <iostream.h>

void increase (void* data, int type)
{
  switch (type)
  {
            case sizeof(char) : (*((char*)data))++; break;
              case sizeof(short): (*((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))
{
```

```
8
```

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

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

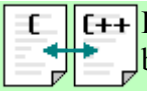
Trong ví dụ này, **minus** là một con trỏ toàn cục trỏ tới một hàm có hai tham số kiểu **int**, con trỏ này được gám để trỏ tới hàm **subtraction**, tất cả đều trên một dòng:

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

Bài 3.4 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
```

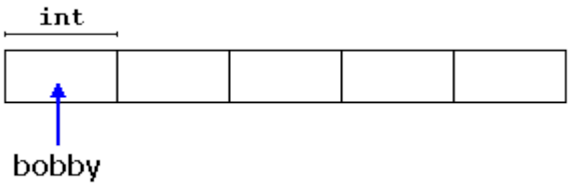
hoặc

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

Biểu thức đầu tien 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 lí 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>

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;
}
```

```
How many numbers do you want to
type              in?              5
Enter       number        :        75
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 thức 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 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 3.5 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* (tuỳ 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 tuỳ 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 tuỳ 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
(films[n].title,50);
    cout << "Enter year: ";
```

```
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)
```

```
    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];

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

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

```
Enter title: Matrix
Enter year: 1999

You have entered:
Matrix (1999)
```

```
  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 trỏ 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 trỏ 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 trỏ 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
movies.title	Phần tử **title** của cấu trúc **movies**	
movies->title	Phần tử **title** của cấu trúc được trỏ bởi **movies**	(*movies).title
***movies.title**	Giá trị được trỏ 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).

Bài 3.6 Các kiểu dữ liệu tự 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
```

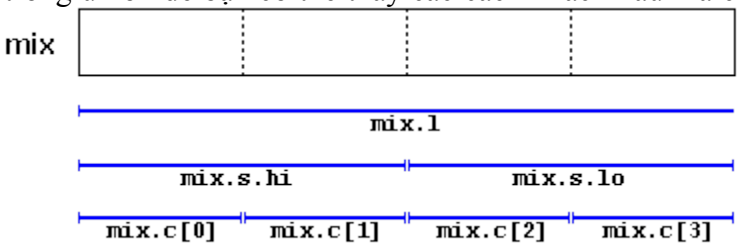
mỗi phần tử có một kiểu dữ liệu khác nhau. Nhưng vì tất cả chúng đều nằm cùng một chỗ trong bộ nhớ nên bất kì sự thay đổi nào đối với một phần tử sẽ ảnh hưởng tới tất cả các thành phần còn lại.

Một trong những công dụng của *union* là dùng để kết hợp một kiểu dữ liêu 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 tuỳ 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;
    int yens;
  } price;
} book;
```

union vô danh

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

Bài 4.1 Các lớp

Lớp là một phương thức logic để tổ chức dữ liệu và các hàm trong cùng một cấu trúc. Chúng được khai báo sử dụng từ khoá **class**, từ này có chức năng tương tự với từ khoá của C **struct** nhưng có khả năng gộp thêm các hàm thành viên.

Dạng thức của nó như sau:

```
class class_name {
  permission_label_1:
    member1;
  permission_label_2:
    member2;
  ...
  } object_name;
```

trong đó ***class_name*** là tên của lớp (kiểu người dùng tự định nghĩa) và trường mặc định ***object_name*** là một hay một vài tên đối tượng hợp lệ. Phần thân của khai báo chứa các **thành viên** của lớp, đó có thể là dữ liệu hay các hàm và có thể là các ***nhãn cho phép (permission labels)*** là một trong những từ khoá sau đây: **private:**, **public:** hoặc **protected:**.

- Các thành viên **private** của một lớp chỉ có thể được truy xuất từ các thành viên khác của lớp hoặc từ các lớp "*bạn bè*".
- Các thành viên **protected** có thể được truy xuất từ các thành viên trong cùng một lớp và các lớp bạn bè, thêm vào đó là từ các thành viên của các lớp thừa kế
- Cuối cùng, các thành viên **public** có thể được truy xuất từ bất kì chỗ nào mà lớp nhìn thấy.

Nếu chúng ta khai báo các thành viên của một lớp trước khi thêm vào các nhãn cho phép thì các thành viên đó được coi là **private**.

Ví dụ:

```
class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area (void);
  } rect;
```

Khai báo lớp **CRectangle** và một đối tượng có tên **rect** có kiểu là lớp **CRectangle**. Lớp này chứa bốn thành viên: hai viến có kiểu **int** (**x** và **y**) trong phần **private** (vì private là sự cho phép mặc định) và hai hàm trong phần **public**: **set_values()** và **area()**, ở đây chúng ta chỉ mới khai báo mẫu.

Hãy chú ý sự khác biệt giữa tên lớp và tên đối đối tượng: Trong ví dụ trước, **CRectangle** là tên lớp còn **rect** là tên một đối tượng có kiểu **CRectangle**.

Trong các phần tiếp theo của chương trình chúng ta có thể truy xuất đến các thành viên public của đối tượng **rect** như là đối với các hàm hay các biến thông thường bằng cách đặt tên của **đối tượng** rồi sau đó là một dấu chấm và tên thành viên của lớp (như chúng ta đã làm với các cấu trúc của C). Ví dụ:

```
rect.set_value                                    (3,4);
myarea = rect.area();
```

nhưng chúng ta không có khả năng truy xuất đến **x** hay **y** vì chúng là các thành viên private của lớp và chúng chỉ có thể được truy xuất từ các thành viên của cùng một lớp. Bối rối? Đây là ví dụ đầy đủ về lớp **CRectangle**:

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

class CRectangle {
```

```
area: 12
```

```
    int x, y;
  public:
    void set_values (int,int);
    int area (void) {return (x*y);}
};

void CRectangle::set_values (int a,
int b) {
  x = a;
  y = b;
}

int main () {
  CRectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
}
```

Một điều mới trong đoạn mã này là toán tử phạm vi **::** được dùng trong khai báo **set_values()**. Nó được sử dụng để khai báo ở bên ngoài các thành viên của một lớp. Chú ý rằng chúng ta đã định nghĩa đầy đủ hàm **area()** ngay bên trong lớp trong khi hàm **set_values()** mới chỉ được khai báo mẫu còn định nghĩa của nó nằm ở ngoài lớp. Trong phần khai báo ở ngoài này chúng ta bắt buộc phải dùng toán tử **::**.

Sự khác biệt duy nhất giữa việc khai báo đầy đủ một hàm bên trong lớp và việc chỉ khai báo mẫu là trong trường hợp thứ nhất hàm sẽ được tự động coi là *inline* bởi trình dịch, còn trong trường hợp thứ hai nó sẽ là một hàm thành viên bình thường.

Lý do khiến chúng ta khai báo **x** và **y** là các thành viên **private** vì chúng ta đã định nghĩa một hàm để thâótc với chúng (**set_values()**) và không có lý do gì để truy nhập trực tiếp đến các biến này. Có lẽ trong ví dụ rất đơn giản này bạn không thấy được một tiện ích lớn khi bảo vệ hai biến này nhưng trong các dự án lớn hơn nó có thể là rất quan trọng khi đảm bảo được rằng các giá trị đó không bị thay đổi một cách không mong muốn.

Một ích lợi nữa của lớp là chúng ta có thể khai báo một vài đối tượng khác nhau từ nó. Ví dụ, tiếp sau đây là ví dụ trước về lớp **CRectangle**, tôi chỉ thêm phần khai báo thêm đối tượng **rectb**.

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

class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area (void) {return (x*y);}
};

void CRectangle::set_values (int a,
int b) {
  x = a;
  y = b;
}

int main () {
```

```
rect area: 12
rectb area: 30
```

```
  CRectangle rect, rectb;
  rect.set_values (3,4);
  rectb.set_values (5,6);
     cout  <<  "rect  area:  "  <<
rect.area() << endl;
     cout  <<  "rectb  area:  "  <<
rectb.area() << endl;
}
```

Chú ý rằng lời gọi đến **rect.area()** không cho cùng kết quả với **rectb.area()** vì mỗi đối tượng của lớp **CRectangle** có các biến và các hàm của riêng nó

Trên đây là những khái niệm cơ bản về đối tượng và lập trình hướng đối tượng. Trong đối tượng các dữ liệu và các hàm là các thuộc tính thay vì trước đây đối tượng là các tham số của hàm trong lập trình cấu trúc. Trong bài này các phần tiếp sau chúng ta sẽ nói đến những lợi ích của phương thức này.

Constructors và destructors

Nói chung các đối tượng cần phải khởi tạo các biến hoặc cấp phát bộ nhớ động trong quá trình tạo ra chúng để có thể hoạt động tốt và tránh được việc trả về các giá trị không mong muốn. Ví dụ, điều gì sẽ xảy ra nếu chúng ta gọi hàm **area()** trước khi gọi hàm **set_values**?Có lẽ kết quả sẽ là một giá trị không xác định vì các thành viên **x** và **y** chưa được gán một giá trị cụ thể nào.

Để tránh điều này, một lớp cần có một hàm đặc biệt: một *constructor*, hàm này có thể được khai báo bằng cách đặt tên trùng với tên của lớp. Nó sẽ được gọi tự động khi một khai báo một đối tượng mới hoặc cấp phát một đối tượng có kiểu là lớp đó. Chúng ta thêm một *constructor* vào lớp **CRectangle**:

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

class CRectangle {
    int width, height;
  public:
    CRectangle (int,int);
        int  area  (void)  {return
(width*height);}
};

CRectangle::CRectangle  (int  a,  int
b) {
  width = a;
  height = b;
}

int main () {
  CRectangle rect (3,4);
  CRectangle rectb (5,6);
     cout  <<  "rect  area:  "  <<
rect.area() << endl;
     cout  <<  "rectb  area:  "  <<
rectb.area() << endl;
```

```
rect area: 12
rectb area: 30
```

```
}
```

Như bạn có thể thấy, kết quả của ví dụ này giống với ví dụ trước. Trong trường hợp này chúng ta chỉ thay thế hàm **set_values** bằng một hàm *constructor*. Hãy chú ý cách mà các tham số được truyền cho constructor khi một đối tượng được tạo ra:

```
CRectangle                    rect                    (3,4);
CRectangle rectb (5,6);
```

Bạn có thể thấy rằng constructor không có giá trị trả về, ngay cả kiểu **void** cũng không. Điều này luôn luôn phải như vậy.

Destructor làm các chức năng ngược lại. Nó sẽ được tự động gọi khi một đối tượng được giải phóng khỏi bộ nhớ hay phạm vi tồn tại của nó đã kết thúc (ví dụ như nếu nó được định nghĩa là một đối tượng cục bộ bên trong một hàm và khi hàm đó kết thúc thì phạm vi tồn tại của nó cũng hết) hoặc nó là một đối tượng đối tượng được cấp phát động và sẽ giải phóng bởi toán tử **delete**.

Destructor phải có cùng tên với tên lớp với dấu (**~**) ở đằng trước và nó không được trả về giá trị nào.

Destructor đặc biệt phù hợp khi mà một đối tượng cấp phát bộ nhớ động trong quá trình tồn tại của nó và trong thời điểm bị huỷ bỏ chúng ta muốn giải phóng bộ nhớ mà nó sử dụng.

```
// example on constructors and destructors
#include <iostream.h>

class CRectangle {
    int *width, *height;
  public:
    CRectangle (int,int);
    ~CRectangle ();
    int area (void) {return (*width * *height);}
};

CRectangle::CRectangle (int a, int b) {
  width = new int;
  height = new int;
  *width = a;
  *height = b;
}

CRectangle::~CRectangle () {
  delete width;
  delete height;
}

int main () {
    CRectangle rect (3,4), rectb (5,6);
    cout << "rect area: " << rect.area() << endl;
```

```
rect area: 12
rectb area: 30
```

```
    cout << "rectb area: " <<
rectb.area() << endl;
  return 0;
}
```

Quá tải các Constructors

Như bất kì hàm nào khác, một constructor có thể được quá tải bởi một vài hàm có cùng tên nhưng khác kiểu hay khác số tham số. Nhớ rằng ở một thời điểm trình dịch chỉ thực hiện một hàm phù hợp (xem phần 2.3, Hàm-II). Do vậy chỉ một hàm constructor phù hợp được gọi vào thời điểm một đối tượng lớp được khai báo.

Trong thực tế, khi khai báo một lớp mà chúng ta không chỉ định một hàm constructor nào thì trình dịch sẽ tự động tạo ra hai constructor quá tải ("*constructor mặc định*" và "*copy constructor*"). Ví dụ, đối với lớp:

```
class CExample {
  public:
    int a,b,c;
    void multiply (int n, int m) { a=n; b=m; c=a*b; };
  };
```

trình dịch sẽ tự động cho rằng lớp có hai constructor sau:

- **Constructor rỗng**
  Đây là một constructor không có tham số. Nó chẳng làm gì cả.

  ```
  CExample::CExample () { };
  ```

- **Copy constructor**
  Đây là một constructor có một tham số cùng kiểu với lớp. Nó thực hiện một việc là gán tất cả các biến thành viên không tĩnh (nonstatic) của lớp giá trị của biến tương ứng của đối tượng tham số.

  ```
  CExample::CExample (const CExample& rv) {
    a=rv.a;  b=rv.b;  c=rv.c;
  }
  ```

Cần phải nhấn mạnh rằng các constructor mặc định này chỉ tồn tại nếu không có constructor được khai báo.

Tất nhiên là bạn có thể quá tải constructor của nó để cung cấp các constructor khác nhau cho các mục đích khác nhau:

```
// overloading class constructors
#include <iostream.h>

class CRectangle {
    int width, height;
  public:
    CRectangle ();
    CRectangle (int,int);
        int area (void) {return
(width*height);}
};
```

```
rect area: 12
rectb area: 25
```

```
CRectangle::CRectangle () {
  width = 5;
  height = 5;
}

CRectangle::CRectangle (int a, int
b) {
  width = a;
  height = b;
}

int main () {
  CRectangle rect (3,4);
  CRectangle rectb;
    cout << "rect area: " <<
rect.area() << endl;
    cout << "rectb area: " <<
rectb.area() << endl;
}
```

Trong trường hợp này **rectb** được khai báo không dùng tham số, vì vậy nó được khởi tạo với *constructor* không có tham số, *constructor* này đặt **width** và **height** bằng **5**.

Chú ý rằng nếu chúng ta khai báo một đối tượng mới và không muốn truyền tham số cho nó thì không cần phải cặp ngoặc đơn **()**:

```
CRectangle rectb;   // đúng
CRectangle rectb(); // sai!
```

Con trỏ tới lớp

Tạo con trỏ trỏ tới các lớp là hoàn toàn hợp lệ, để có thể làm việc này chúng ta hiểu rằng một khi đã được khai báo, lớp trở thành một kiểu dữ liệu hợp lệ, vì vậy chúng ta có dùng tên lớp là kiểu cho con trỏ. Ví dụ:

```
CRectangle * prect;
```

là một con trỏ trỏ tới một đối tượng của lớp **CRectangle**.

Tương tự với cấu trúc, để tham chiếu trực tiếp tới một thành viên của một đối tượng được trỏ bởi một con trỏ bạn nên dùng toán tử **->**. Đây là một ví dụ:

```
// pointer to classes example
#include <iostream.h>

class CRectangle {
    int width, height;
  public:
    void set_values (int, int);
     int area (void) {return (width
* height);}
};
```

```
a              area:              2
*b             area:             12
*c              area:              2
d[0]            area:             30
d[1] area: 56
```

```
void CRectangle::set_values (int a,
int b) {
  width = a;
  height = b;
}

int main () {
  CRectangle a, *b, *c;
       CRectangle   *   d   =   new
CRectangle[2];
  b= new CRectangle;
  c= &a;
  a.set_values (1,2);
  b->set_values (3,4);
  d->set_values (5,6);
  d[1].set_values (7,8);
   cout << "a area: " << a.area() <<
endl;
   cout << "*b area: " << b->area()
<< endl;
   cout << "*c area: " << c->area()
<< endl;
     cout   <<   "d[0]   area:   "   <<
d[0].area() << endl;
     cout   <<   "d[1]   area:   "   <<
d[1].area() << endl;
  return 0;
}
```

Dưới đây là là bảng tóm tắt về các chức năng của các toán tử lớp (***, &, ., ->, [ ]**) và các con trỏ lớp xuất hiện trong ví dụ trên:

***x** *có thể đọc là:* trỏ bởi **x**
&x *có thể đọc là:* địa chỉ của **x**
x.y *có thể đọc là:* thành viên **y** của đối tượng **x**
(*x).y *có thể đọc là:* thàn viên **y** của đối tượng được trỏ bởi **x**
x->y *có thể được đọc là:* thành viên **y** của đối tượng được trỏ bởi **x**(tương đương với dòng trên)
x[0] *có thể đọc là:* đối tượng đầu tiên được trỏ bởi **x**
x[1] *có thể đọc là:* đối tượng thứ hai được trỏ bởi **x**
x[n] *có thể được đọc là:* đối tượng thứ (n+1)được trỏ bởi **x**

Các lớp được định nghĩa bằng từ khoá `struct`

Ngôn ngữ C++ đã mở rộng từ khoá **struct** của C làm cho nó cũng có các chức năng như từ khoá **class** ngoại trừ một điều là các thành viên của nó mặc định là **public** thay vì **private**.

Tuy nhiên, mặc dù cả **class** và **struct** gần như là tương đương trong C++, **struct** thường chỉ dùng các cấu trúc dữ liệu còn **class** thì dùng cho các lớp có cả các thủ tục và hàm thành viên.

Bài 4.2 Quá tải các toán tử

C++ cho phép sử dụng các toán tử chuẩn của ngôn ngữ giữa các lớp giống như với các kiểu dữ liệu cơ bản. Ví dụ:

```
int a, b, c;
a = b + c;
```

là hoàn toàn hợp lệ vì các biến ở đây đều có kiểu là các kiểu dữ liệu cơ bản. Tuy nhiên, việc chúng ta có thể thực hiện thao tác sau đây có vẻ không hiển nhiên lắm (thực tế là nó không hợp lệ):

```
struct { char product [50]; float price; } a, b, c;
a = b + c;
```

Phép gán một lớp (hay một cấu trúc) với một đối tượng cùng kiểu là được phép (copy constructor mặc định). Nhưng phép cộng sẽ gây ra lỗi vì nó được dùng với các kiểu dữ liệu không cơ bản.

Nhưng cần phải cám ơn khả năng quá tải toán tử của C++, chúng ta có thể làm cho các đối tượng kiểu như trên có thể chấp nhận các toán tử đó mà không làm thay đổi ý nghĩa của nó đối với các kiểu dữ liệu cơ bản. Dưới đây là danh sách tất cả các toán tử có thể được quá tải:

```
+     -     *     /     =     <     >     +=    -=    *=    /=    <<    >>
<<=   >>=   ==    !=    <=    >=    ++    --    %     &     ^     !     |
~     &=    ^=    |=    &&    ||    %=    []    ()    new   delete
```

Để làm quá tải một toán tử chúng ta chỉ cần viết một hàm thành viên của lớp có tên **operator** theo sau là toán tử chúng ta muốn làm quá tải. Mẫu như sau:

```
type operator sign (parameters);
```

Dưới đây là ví dụ về việc quá tải toán tử **+**. Chúng ta chuẩn bị tính tổng hai vector hai chiều **a(3,1)** và **b(1,2)**. Phép cộng giữa hai vector hai chiều chỉ đơn giản là cộng hai toạ độ x để lấy toạ độ kết quả x , cộng hai toạ độ y để lấy toạ độ kết quả y. Trong trường hợp này kết quả sẽ là **(3+1,1+2) = (4,3)**.

```
// vectors: ví dụ về quá tải toán tử
#include <iostream.h>

class CVector {
  public:
    int x,y;
    CVector () {};
    CVector (int,int);
    CVector operator + (CVector);
};

CVector::CVector (int a, int b) {
  x = a;
  y = b;
}

CVector CVector::operator+ (CVector param) {
```

```
4,3
```

```
  CVector temp;
  temp.x = x + param.x;
  temp.y = y + param.y;
  return (temp);
}

int main () {
  CVector a (3,1);
  CVector b (1,2);
  CVector c;
  c = a + b;
  cout << c.x << "," << c.y;
  return 0;
}
```

Nếu bạn thấy quá nhiều **CVector** hãy để ý rằng một số trong chúng tham chiếu đến tên lớp **CVector** còn số còn lại là tên các hàm (constructor và destructor). Đừng lẫn lộn

```
CVector (int, int);            // Hàm có tên Vector (constructor)
CVector operator+ (CVector);     // Hàm operator+   trả về kiểu
CVector
```

Hàm **operator+** của lớp **CVector** được dùng để quá tải toán tử số học +. Hàm này có thể được gọi bằng một trong các cách:

```
c              =              a              +              b;
c = a.operator+ (b);
```

Hãy chú ý rằng chúng ta đã thêm vào constructor rỗng (không có tham số) và chúng ta định nghĩa nó với một khối lệnh cũng rỗng nốt:

```
CVector () { };
```

điều này là cần thiết vì còn có một constructor khác,

```
CVector (int, int);
```

và vì vậy các *constructors mặc định* không tồn tại trong `CVector` nếu chúng ta không khai nó một cách rõ ràng. Khai báo sau đây sẽ là không hợp lệ:

```
CVector c;
```

Dù thế nào chăng nữa, tôi cần phải cảnh báo rằng một khối lệnh rỗng không nên để tạo một constructor vì nó không thoả mãn chức năng tối thiểu mà một constructor nên có, đó là việc khởi tạo tất cả các biến trong lớp. Trong trường hợp của chúng ta constructor này đã để các biến **x** và **y** là không xác định. Vì vậy một khai báo thích hợp hơn sẽ là một cái gì đó giống như thế này:

```
CVector () { x=0; y=0; };
```

để cho đơn giản tôi đã không viết vào trong ví dụ trên.

Cùng với việc tạo một constructor rỗng và một copy constructor, C++ còn mặc định định nghĩa **toán tử gán (=)** giữa hai lớp có cùng một kiểu. Nó copy toàn bộ nội dung của các thành

viên không tĩnh (non-static) của đối tượng bên phải phép gán cho đối tượng bên trái. Tất nhiên bạn có thể định nghĩa lại nó để thực hiện chức năng khác mà bạn muốn, ví dụ như chỉ copy một số thành viên nào đó của lớp.

Việc quá tải các toán tử không bắt buộc hoạt động của nó phải liên quan đến ý nghĩa thông thường của nó mặc dù điều này là nên làm. Ví dụ có vẻ không logic lắm nếu sử dụng toán tử + để trừ hai lớp hay toán tử **==** để điền số 0 vào một lớp mặc dù điều đó là hoàn toàn hợp lệ.

Mặc dù khai báo mẫu của hàm **operator+** là khá hiển nhiên vì nó lấy phần bên phải của toán tử làm tham số cho hàm `operator+`, các toán tử khác không phải cái nào cũng rõ ràng như thế. Ở đây chúng ta có một bảng tổng kết về việc các hàm **operator** phải được khai báo như thế nào (thay thế `@` bằng các toán tử tương ứng):

Biểu thức	Toán tử (@)	**Hàm thành viên**	**Hàm toàn cục**
`@a`	`+ - * & ! ~ + + --`	`A::operator@()`	`operator@(A)`
`a@`	`++ --`	`A::operator@(int)`	`operator@(A, int)`
`a@b`	`+ - * / % ^ & | < > == != <= >= << >> && || ,`	`A::operator@(B)`	`operator@(A, B)`
`a@b`	`= += -= *= /= %= ^= &= |= <<= >>= [ ]`	`A::operator@(B)`	-
`a(b, c...)`	`()`	`A::operator()(B, C...)`	-
`a->b`	`->`	`A::operator->()`	-

* trong đó **a** là một đối tượng của lớp **A**, **b** là một đối tượng của lớp **B** và **c** là một đối tượng của lớp **C**.

Như bạn có thể thấy trong bảng này, có hai cách để quá tải các toán tử của lớp: như là một hàm thành viên và như là một hàm toàn cục. Khác nhau giữa chúng không rõ ràng tuy nhiên tôi cần phải nhắc lại rằng các hàm không phải là thành viên của một lớp không thể truy xuất đến các thành viên là `private` hoặc `protected` của lớp trừ phi hàm toàn cục đó là *bạn* của lớp (thuật ngữ này sẽ được đề cập đến ở bài sau).

Từ khoá `this`

Từ khoá **this** ở bên trong một lớp đại diện cho đối tượng của lớp đó đang được thực hiện trong bộ nhớ. Nó là một con trỏ luôn có giá trị là địa chỉ của đối tượng.

Nó có thể được dùng để kiểm tra xem tham số được truyền cho một hàm thành viên có phải chính bản thân đối tượng hay không. Ví dụ:

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

class CDummy {
  public:
    int isitme (CDummy& param);
```

```
yes, &a is b
```

```
};

int CDummy::isitme (CDummy& param)
{
  if (&param == this) return 1;
  else return 0;
}

int main () {
  CDummy a;
  CDummy* b = &a;
  if ( b->isitme(a) )
    cout << "yes, &a is b";
  return 0;
}
```

Nó cũng thường được dùng trong hàm thành viên **operator=** mà trả về địa chỉ đối tượng (tránh việc sử dụng đối tượng tạm thời). Tiếp theo ví dụ về vector ở đầu bài chúng ta có thể viết một hàm **operator=** như sau:

```
CVector& CVector::operator= (const CVector& param)
{
  x=param.x;
  y=param.y;
  return *this;
}
```

Trong thực tế đây chính là đoạn mã được mặc định tạo ra nếu chúng ta không viết hàm thành viên **operator=**.

Các thành viên tĩnh

Một lớp có thể chứa các thành viên tĩnh, cả dữ liệu và các hàm.

Các dữ liệu tĩnh còn được gọi là "biến của lớp" vì nội dung của chúng không phụ thuộc vào một đối tượng nào. Chỉ có một giá trị duy nhất cho tất cả các đối tượng trong cùng một lớp.

Ví dụ, nó có thể được trong trường hợp bạn muốn có một biến chứa số đối tượng thuộc lớp đã được khai báo:

```
// static members in classes
#include <iostream.h>

class CDummy {
  public:
    static int n;
    CDummy () { n++; };
    ~CDummy () { n--; };
};

int CDummy::n=0;

int main () {
  CDummy a;
  CDummy b[5];
```

```
7
6
```

```
  CDummy * c = new CDummy;
  cout << a.n << endl;
  delete c;
  cout << CDummy::n << endl;
  return 0;
}
```

Trong thực tế, các thành viên tĩnh có cùng thuộc tính với các biến toàn cục. Vì vậy, để tránh việc chúng bị khai báo nhiều lần, theo chuẩn ANSI-C++ chúng ta chỉ được viết phần khai báo mẫu trong phần khai báo của lớp. Để có thể khởi tạo một thành viên tĩnh chúng ta phải viết một định nghĩa ở bên ngoài lớp, giống như ở trong ví dụ trước.

Vì nó là một biến duy nhất cho tất cả các đối tượng thuộc lớp, nó có thể được tham chiếu đến như là thành viên của bất kì đối tượng nào thuộc lớp hay thậm chí chính bản thân tên lớp (tất nhiên điều này chỉ hợp lệ với các thành viên **tĩnh**):

```
cout << a.n;
cout << CDummy::n;
```

Hai lời gọi trong ví dụ trên đều tham chiếu đến cùng một biến: biến tĩnh **n** trong lớp **CDummy**.

Một lần nữa, tôi phải nhắc lại rằng nó là một biến toàn cục (nghĩa là bạn có thể truy xuất nó từ bất kì đâu). Sự khác biệt duy nhất là bạn phải dùng nó với tên lớp.

Chúng ta cũng có thể tạo ra các hàm tĩnh giống như các biến tĩnh. Chúng cũng có ý nghĩa như đối với các biến tĩnh: đó là các hàm toàn cục có thể được gọi như thể là các đối tượng của một lớp. Chúng chỉ có thể truy xuất đến phần dữ liệu tĩnh cũng như không được phép sử dụng từ khoá **this** vì nó tạo tham chiếu đến một con trỏ đối tượng. Các hàm này thực tế không phải là thành viên của bất kì đối tượng nào mà thực chất là của lớp.

Bài 4.3 Quan hệ giữa các lớp

Các hàm bạn bè (từ khoá `friend`)

Trong bài trước chúng ta đã được biết rằng có ba mức bảo vệ khác nhau đối với các thành viên trong một lớp: **public**, **protected** và **private**. Đối với các thành viên *protected* và *private*, chúng không thể được truy xuất ở bên ngoài lớp mà chúng được khai báo. Tuy nhiên cái gì cũng có ngoại lệ, bằng cách sử dụng từ khoá *friend* trong một lớp chúng ta có thể cho phép một hàm bên ngoài truy xuất vào các thành viên **protected** và **private** trong một lớp.

Để có thể cho phép một hàm bên ngoài truy xuất vào các thành viên **private** và **protected** của một lớp chúng ta phải khai báo mẫu hàm đó với từ khoá **friend** bên trong phần khai báo của lớp. Trong ví dụ sau chúng ta khai báo hàm bạn bè **duplicate**:

```
// friend functions
#include <iostream.h>

class CRectangle {
    int width, height;
  public:
    void set_values (int, int);
     int area (void) {return (width
```

```
24
```

```
* height);}
       friend CRectangle duplicate
(CRectangle);
};

void CRectangle::set_values (int a,
int b) {
  width = a;
  height = b;
}

CRectangle duplicate (CRectangle
rectparam)
{
  CRectangle rectres;
               rectres.width       =
rectparam.width*2;
              rectres.height       =
rectparam.height*2;
  return (rectres);
}

int main () {
  CRectangle rect, rectb;
  rect.set_values (2,3);
  rectb = duplicate (rect);
  cout << rectb.area();
}
```

Ở bên trong hàm **duplicate**, chúng ta có thể truy xuất vào các thành viên **width** và **height** của các đối tượng khác nhau thuộc lớp **CRectangle**. Hãy chú ý rằng **duplicate()** không phải là thành viên của lớp **CRectangle**.

Nói chung việc sử dụng các hàm bạn bè không nằm trong phương thức lập trình hướng đối tượng, vì vậy tốt hơn là hãy sử dụng các thành viên của lớp bất cứ khi nào có thể. Như ở trong ví dụ trước, chúng ta hoàn toàn có thể tích hợp **duplicate()** vào bên trong lớp.

Các lớp bạn bè (friend)

Ngoài việc có thể khai báo các hàm bạn bè, chúng ta cũng có thể định nghĩa một lớp là bạn của một lớp khác. Việc này sẽ cho phép lớp thứ hai có thể truy xuất vào các thành viên **protected** and **private** của lớp thứ nhất:

```
// friend class
#include <iostream.h>

class CSquare;

class CRectangle {
    int width, height;
  public:
    int area (void)
      {return (width * height);}
    void convert (CSquare a);
};
```

```
16
```

```
class CSquare {
  private:
    int side;
  public:
    void set_side (int a)
      {side=a;}
    friend class CRectangle;
};

void  CRectangle::convert  (CSquare
a) {
  width = a.side;
  height = a.side;
}

int main () {
  CSquare sqr;
  CRectangle rect;
  sqr.set_side(4);
  rect.convert(sqr);
  cout << rect.area();
  return 0;
}
```

Trong ví dụ này chúng ta đã khai báo **CRectangle** là bạn của **CSquare** nên **CRectangle** có thể truy xuất vào các thành viên **protected** and **private** của **CSquare**, cụ thể hơn là **CSquare::side**, biến định nghĩa kích thước của hình vuông.

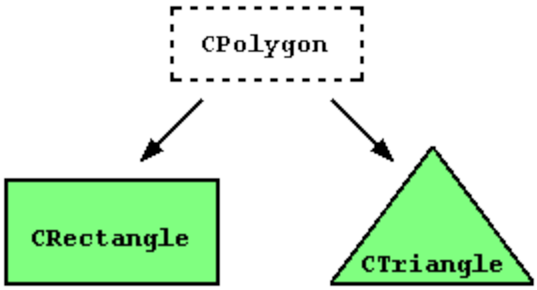
Bạn có thể thấy một điều mới lạ trong chương trình, đó là phần khai báo mẫu rỗng của lớp **CSquare**, điều này là cần thiết vì bên trong phần khai báo của **CRectangle** chúng ta tham chiếu đến **CSquare** (như là một tham số trong **convert()**). Phần định nghĩa đầy đủ của **CSquare** được viết ở sau.

Chú ý rằng tình bạn giữa hai lớp có thể không như nhau nếu chúng ta không chỉ định. Trong ví dụ này **CRectangle** được coi là bạn của **CSquare** nhưng đối với **CRectangle** thì không. Bởi vậy **CRectangle** có thể truy xuất vào các thành viên **protected** và **private** của **CSquare** nhưng điều ngược lại là không đúng. Tuy nhiên chẳng có gì cấm đoán chúng ta khai báo **CSquare** là bạn của **CRectangle**.

Sự thừa kế giữa các lớp

Một trong những tính năng quan trọng của lớp là sự thừa kế. Nó cho phép chúng ta tạo một đối tượng xuất pháp từ một đối tượng khác. Ví dụ, giả sử chúng ta muốn khai báo một loạt các lớp mô tả các đa giác như là **CRectangle** hay **CTriangle**. Cả hai đều có những đặc tính chung, ví dụ như là chiều cao và đáy.

Điều này có thể được biểu diễn bằng lớp **CPolygon** mà từ đó chúng ta có thể thừa kế hai lớp, đó là **CRectangle** và **CTriangle**.



Lớp **CPolygon** sẽ chứa các thành viên chung đối với mọi đa giác. Trong trường hợp của chúng ta: chiều rộng và chiều cao.

Các lớp xuất phát từ các lớp khác được thừa hưởng tất cả các thành viên nhìn thấy được của lớp. Điều này có nghĩa là một lớp cơ sở có thành viên **A** và chúng ta tạo thêm một lớp xuất phát từ nó với một thành viên mới là **B**, lớp được thừa kế sẽ có cả **A** và **B**.

Để có thể thừa kế một lớp từ một lớp khác, chúng ta sử dụng toán tử **:** (dấu hai chấm) trong phần khai báo của lớp con:

```
class derived_class_name: public base_class_name;
```

trong đó *derived_class_name* là tên của lớp con (lớp được thừa kế) và *base_class_name* là tên của lớp cơ sở. **public** có thể được thay thế bởi **protected** hoặc **private**, nó xác định quyền truy xuất đối với các thành viên được thừa kế như chúng ta sẽ thấy ở ví dụ này:

```
// derived classes
#include <iostream.h>

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
  };

class CRectangle: public CPolygon {
  public:
    int area (void)
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area (void)
        { return (width * height /
2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << endl;
```

```
20
10
```

```
  cout << trgl.area() << endl;
  return 0;
}
```

Như bạn có thể thấy, các đối tượng của lớp **CRectangle** và **CTriangle** chứa tất cả các thành viên của **CPolygon**, đó là **width**, **height** và **set_values()**.

Từ khoá **protected** tương tự với **private**, sự khác biệt duy nhất chỉ xảy ra khi thừa kế các lớp. Khi chúng ta thừa kế một lớp, các thành viên **protected** của lớp cơ sở có thể được dùng bởi các thành viên khác của lớp được thừa kế còn các thành viên **private** thì không. Vì chúng ta muốn rằng **width** và **height** có thể được tính toán bởi các thành viên của các lớp được thừa kế **CRectangle** và **CTriangle** chứ không chỉ bởi các thành viên của **CPolygon**, chúng ta đã sử dụng từ khoá **protected** thay vì **private**.

Chúng ta có thể tổng kết lại các kiểu truy xuất khác nhau tuỳ theo ai truy xuất chúng:

Truy xuất	public	protected	private
Các thành viên trong cùng lớp	có	có	có
Các thành viên của các lớp thừa kế	có	có	không
Không phải là thành viên	có	không	không

trong đó "không phải là thành viên" đại diện cho bất kì sự tham chiếu nào từ bên ngoài lớp, ví dụ như là từ **main()**, từ một lớp khác hay từ bất kì hàm nào.

Trong ví dụ của chúng ta, các thành viên được thừa kế bởi **CRectangle** và **CTriangle** cũng tuân theo các quyền truy xuất như đối với lớp cơ sở **CPolygon**:

```
CPolygon::width           // protected access
CRectangle::width         // protected access

CPolygon::set_values()    // public access
CRectangle::set_values()  // public access
```

Có điều này vì chúng ta đã thừa kế một lớp từ một lớp khác với quyền truy xuất **public**, hãy nhớ:

```
class CRectangle: public CPolygon;
```

từ khoá **public** đại diện cho mức độ bảo về tối thiểu mà các thành viên được thừa kế của lớp cơ sở (**CPolygon**) phải có được trong lớp mới (**CRectangle**). Mức độ này đối với các thành viên được thừa kế có thể được thay đổi nếu thay vì dùng **public** chúng ta sử dụng **protected** hay **private**, ví dụ, giả sử rằng **daughter** là một lớp được thừa kế từ **mother**, chúng định nghĩa như thế này:

```
class daughter: protected mother;
```

điều này sẽ thiết lập **protected** là mức độ truy xuất tối thiểu cho các thành viên của **daughter** được thừa kế từ lớp cơ sở. Có nghĩa là tất cả các thành viên **public** trong **mother** sẽ trở thành **protected** trong **daughter**. Tất nhiên, điều này không cản trở **daughter** có thể có các thành viên **public** của riêng nó. Mức độ tối thiểu này chỉ áp dụng cho các thành viên được thừa kế từ **mother**.

Nếu không có mức truy xuất nào được chỉ định, **`private`** được dùng với các lớp được tạo ra với từ khoá **`class`** còn **`public`** được dùng với các cấu trúc.

Những gì được thừa kế từ lớp cơ sở?

Về nguyên tắc tất cả các thành viên của lớp đều được thừa kế trừ:

- **Constructor và destructor**
- **Thành viên `operator=()`**
- **Bạn bè**

Mặc dù constructor và destructor của lớp cơ sở không được thừa kế, constructor mặc định (constructor không có tham số) và destructor của lớp cơ sở luôn luôn được gọi khi một đối tượng của lớp được thừa kế được tạo lập hay phá huỷ.

Nếu lớp cơ sở không có constructor mặc định hay bạn muốn một constructor đã quá tải được gọi khi một đối tượng mới của lớp được thừa kế được tạo lập, bạn có thể chỉ định nó ở mỗi định nghĩa của constructor trong lớp được thừa kế:

```
derived_class_name (parameters) : base_class_name
(parameters) {}
```

Ví dụ:

```
// constructors and derivated classes
#include <iostream.h>

class mother {
  public:
    mother ()
          { cout << "mother: no parameters\n"; }
    mother (int a)
          { cout << "mother: int parameter\n"; }
};

class daughter : public mother {
  public:
    daughter (int a)
         { cout << "daughter: int parameter\n\n"; }
};

class son : public mother {
  public:
    son (int a) : mother (a)
           { cout << "son: int parameter\n\n"; }
};

int main () {
  daughter cynthia (1);
  son daniel(1);
```

```
mother:         no         parameters
daughter:        int        parameter

mother:         int        parameter
son: int parameter
```

```
  return 0;
}
```

Hãy quan sát sự khác biệt giữa việc constructor của **mother** được gọi khi khi một đối tượng **daughter** mới được tạo lập và khi một đối tượng **son** được tạo lập. Sở dĩ có sự khác biệt này là do phần khai báo constructor của **daughter** và **son**:

```
daughter (int a)          // không có gì được chỉ định: gọi
constructor mặc định
son (int a) : mother (a)  // constructor được chỉ định: gọi
cái này
```

Đa thừa kế

Trong C++ chúng ta có thể tạo lập một lớp thừa kế các trường và các phương thức từ nhiều hơn một lớp. Điều này có thể thực hiện bằng cách tách các lớp cơ sở khác nhau bằng dấu phẩy trong phần khai báo của lớp được thừa kế. Ví dụ, nếu chúng ta có một lớp dùng để in ra màn hình (**COutput**) và chúng ta muốn các lớp của chúng ta **CRectangle** và **CTriangle** cũng thừa kế các thành viên của nó cùng với các thành viên của **CPolygon**, chúng ta có thể viết:

```
class CRectangle: public CPolygon, public COutput {
class CTriangle: public CPolygon, public COutput {
```

Đây là ví dụ đầy đủ:

```
// đa thừa kế
#include <iostream.h>

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
  };

class COutput {
  public:
    void output (int i);
  };

void COutput::output (int i) {
  cout << i << endl;
  }

class CRectangle: public CPolygon,
public COutput {
  public:
    int area (void)
      { return (width * height); }
  };

class CTriangle: public CPolygon,
```

```
20
10
```

```
public COutput {
  public:
    int area (void)
        { return (width * height /
2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  rect.output (rect.area());
  trgl.output (trgl.area());
  return 0;
}
```

Bài 4.4 Các thành viên ảo. Đa hình.

Để có thể hiểu được phần này bạn cần hiểu rõ về cách sử dụng **con trỏ** và **thừa kế giữa các lớp**. Nếu có vài biểu thức nào có vẻ lạ lùng với bạn, bạn có thể xem lại các phần sau:

```
int a::b(c) {};     // Các lớp (Bài 4.1)
a->b                // Con trỏ và đối tượng (Bài 4.2)
class a: public b; // Quan hệ giữa các lớp(Bài 4.3)
```

Con trỏ tới lớp cơ sở

Một trong những lợi thế lớn của việc thừa kế các lớp là **một con trỏ trỏ tới một lớp được thừa kế là tương thích về kiểu với một con trỏ trỏ tới lớp cơ sở của nó**. Bài này sẽ đề cập đầy đủ đến việc tận dụng tính năng mạnh mẽ này của C++. Ví dụ, chúng ta sẽ viết lại chương trình của chúng ta về hình chữ nhật và hình tam giác trong chương trước để xem xét tính năng này:

```
// con trỏ tới lớp cơ sở
#include <iostream.h>

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
  };

class CRectangle: public CPolygon {
  public:
    int area (void)
      { return (width * height); }
  };

class CTriangle: public CPolygon {
```

```
20
10
```

```
  public:
    int area (void)
        { return (width * height /
2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << endl;
  cout << sqre.area() << endl;
  return 0;
}
```

Hàm **main** tạo hai con trỏ trỏ tới hai đối tượng của lớp **CPolygon**, đó là ***ppoly1** và ***ppoly2**. Chúng được gán cho địa chỉ của **rect** và **trgl**, đây là các đối tượng thuộc lớp thừa kế từ **CPolygon** nên đó là những phép gán hợp lệ.

Sự hạn chế duy nhất khi sử dụng ***ppoly1** và ***ppoly2** thay vì **rect** và **trgl** là cả ***ppoly1** và ***ppoly2** đều có kiểu là **CPolygon*** và vì vậy chúng ta chỉ có thể tham chiếu đến các thành viên mà **CRectangle** và **CTriangle** được thừa kế từ **CPolygon**. Vì nguyên nhân đó chúng ta không thể gọi đến thành viên **area()** khi dùng ***ppoly1** và ***ppoly2**.

Để các con trỏ đó có thể truy xuất đến **area()** như là một thành viên hợp lệ, cần phải khai báo thành viên này trong lớp cơ sở chứ không chỉ trong các lớp thừa kế.

Các thành viên ảo

Nếu muốn khai báo một phần tử trong một lớp mà chúng ta muốn định nghĩa lại nó trong các lớp thừa kế thì chúng ta phải đặt trước nó từ khoá **virtual** để việc sử dụng con trỏ tới các đối tượng thuộc lớp này là thích hợp.

Hãy xem ví dụ sau:

```
// các thành viên ảo
#include <iostream.h>

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area (void)
      { return (0); }
  };

class CRectangle: public CPolygon {
  public:
```

```
20
10
0
```

```
    int area (void)
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area (void)
        { return (width * height /
2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon poly;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  CPolygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << endl;
  cout << ppoly2->area() << endl;
  cout << ppoly3->area() << endl;
  return 0;
}
```

Bây giờ cả ba lớp (**CPolygon**, **CRectangle** và **CTriangle**) đều có cùng các thành viên: **width**, **height**, **set_values()** và **area()**.

area() được định nghĩa là **virtual** vì nó sẽ được định nghĩa lại trong các lớp thừa kế. Bạn có thể kiểm tra lại rằng nếu bạn bỏ từ khoá đó và thực hiện chương trình thì kết quả sẽ là **0** cho cả 3 đa giác thay vì **20,10,0**. Nguyên nhân là do thay vì gọi hàm **area()** tương ứng với mỗi đối tượng (**CRectangle::area()**, **CTriangle::area()** và **CPolygon::area()**), **CPolygon::area()** sẽ được gọi cho tất cả thông qua một con trỏ tới **CPolygon**.

Trừu tượng hoá lớp cơ sở

Các lớp trừu trượng là một cái gì đó rất giống với lớp lớp **CPolygon** trong ví dụ trước của chúng ta. Sự khác biệt duy nhất là trong ví dụ đó chúng ta đã định nghĩa hàm **area()** cho các đối tượng thuộc lớp **CPolygon** (giống như đối tượng **poly**), trong khi ở trong một lớp trừu tượng cơ sở chúng ta có thể bỏ qua việc định nghĩa hàm này bằng cách thêm **=0**(bằng không) vào phần khai báo hàm.

Lớp **CPolygon** có thể được định nghĩa như sau:

```
// abstract class CPoligon
class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area (void) =0;
```

```
};
```

Hãy chú ý cách chúng ta thêm **=0** vào **virtual int area (void)** thay vì định nghĩa đầy đủ cho hàm. Kiểu hàm này có tên là là *pure virtual function* (hàm ảo thuần tuý) và tất cả các lớp chứa bất kì một hàm ảo thuần tuý nào đều được coi là lớp trừu tượng cơ sở.

Sự khác biệt lớn của một lớp trừu tượng cơ sở là không thể tạo được các đối tượng thuộc lớp. Nhưng chúng ta có thể tạo các con trỏ trỏ đến chúng. Vì vậy một khai báo như sau:

```
CPolygon poly;
```

sẽ là không hợp lệ cho lớp trừu tượng cơ sở được khai báo ở trên. Tuy nhiên con trỏ:

```
CPolygon * ppoly1;
CPolygon * ppoly2
```

là hoàn toàn hợp lệ. Có điều này vì hàm trừu tượng thuần tuý mà nó có không được định nghĩa và không thể toạ được một đối tượng nếu như chưa định nghĩa tất cả các thành viên của nó. Tuy nhiên một con trỏ trỏ tới một đối tượng thuộc lớp thừa kế mà hàm này đã được định nghĩa là hoàn toàn hợp lệ.

Dưới đây chúng ta có một ví dụ đầy đủ:

```
// các thành viên ảo.
#include <iostream.h>

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area (void) =0;
  };

class CRectangle: public CPolygon {
  public:
    int area (void)
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area (void)
        { return (width * height /
2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << ppoly1->area() << endl;
```

```
20
10
```

```
  cout << ppoly2->area() << endl;
  return 0;
}
```

Nếu bạn xem lại chương trình bạn sẽ thấy rằng chúng ta tham chiếu đến các đối tượng thuộc các lớp khác nhau nhưng chỉ sử dụng một kiểu con trỏ duy nhất. Điều này là cực kì hữu dụng, bây giờ chúng ta có thể tạo một hàm thành viên của **CPolygon** có khả năng in ra màn hình kết quả của hàm **area()** mà không phụ thuộc vào lớp được thừa kế là lớp nào.

```
// ejemplo miembros virtuales
#include <iostream.h>

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area (void) =0;
    void printarea (void)
         { cout << this->area() <<
endl; }
  };

class CRectangle: public CPolygon {
  public:
    int area (void)
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area (void)
        { return (width * height /
2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  return 0;
}
```

```
20
10
```

Hãy nhớ rằng **this** biểu diễn một con trỏ trỏ đến đối tượng đang được thực hiện.

Các lớp trừu tượng và các thành viên ảo cung cấp cho C++ tính năng đa hình khiến cho việc lập trình hướng đối tượng trở thành một công cụ hữu dụng. Tất nhiên chúng ta đã thấy cách đơn giản nhất để sử dụng những tính năng này, nhưng hãy tưởng tượng nếu những tính năng này được áp dụng cho các mảng các đối tượng hay các đối tượng được cấp phát thông qua bộ nhớ động.

Bài 5.1 Templates

Các mẫu hàm

Các mẫu cho phép tạo các hàm có thể chấp nhận bất kì kiểu dữ liệu nào làm tham số và trả về giá trị mà không phải làm quá tải hàm với tất cả các kiểu dữ liệu có thể. Khai báo mẫu của nó có thể là một trong hai kiểu sau:

```
template <class indetifier> function_declaration;
template <typename indetifier> function_declaration;
```

sự khác biệt duy nhất giữa hai kiểu khai báo mẫu này là việc sử dụng từ khoá **class** hay **typename**, sự khác nhau giữa chúng là không rõ ràng vì cả hai đều có cùng một ý nghĩa và đều cho một kết quả như nhau.

Ví dụ, để tạo một hàm mẫu trả về giá trị lớn hơn của hai đối tượng chúng ta có thể sử dụng:

```
template <class GenericType>
GenericType GetMax (GenericType a, GenericType b) {
 return (a>b?a:b);
}
```

Ở dòng đầu tiên, chúng ta đã tạo một mẫu cho một kiểu dữ liệu tổng quát với tên **GenericType**. Vì vậy trong hàm sau đó, **GenericType** trở thành một kiểu dữ liệu hợp lệ và nó được sử dụng như là một kiểu dữ liệu cho hai tham số **a**, **b** và giá trị trả về của hàm **GetMax**.

GenericType thực sự không biểu diễn một kiểu dữ liệu cụ thể nào, chúng ta có thể gọi hàm với bất kì kiểu dữ liệu hợp lệ nào. Kiểu dữ liệu này sẽ đáp ứng như là *pattern(mẫu)* và sẽ thay thế **GenericType** bên trong hàm. Cách thức để gọi một lớp mẫu với một kiểu dữ liệu mẫu như sau:

```
function <pattern> (parameters);
```

Ví dụ, để gọi hàm **GetMax** và so sánh hai giá trị nguyên kiểu **int** chúng ta có thể viết:

```
int x,y;
GetMax <int> (x,y);
```

Ok, dưới đây là ví dụ đầy đủ:

```
// mẫu hàm
#include <iostream.h>

template <class T>
T GetMax (T a, T b) {
  T result;
```

```
6
10
```

```
  result = (a>b)? a : b;
  return (result);
}

int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax<int>(i,j);
  n=GetMax<long>(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

(Trong trường hợp này chúng ta gọi kiểu dữ liệu tổng quát là **T** thay vì **GenericType** vì nó ngắn hơn, thêm vào đó nó là một trong những tên phổ biến nhất được dùng cho mẫu mặc dù chúng ta có thể sử dụng bất cứ tên hợp lệ nào).

Ở ví dụ trên chúng ta đã sử dụng cùng một hàm **GetMax()** với các tham số kiểu **int** và **long** chỉ với một phiên bản duy nhất của hàm. Như vậy có thể nói chúng ta đã viết một mẫu hàm và gọi nó bằng hai kiểu khác nhau.

Như bạn có thể thấy, bên trong mẫu hàm **GetMax()** kiểu **T** có thể được dùng để khai báo các đối tượng mới:

```
T result;
```

Trong trường hợp cụ thể này kiểu dữ liệu tổng quát **T** được sử dụng như là tham số chao hàm **GetMax**, trình biên dịch có thể tự động tìm thấy kiểu dữ liệu nào phải truyền cho nó mà không cần bạn phải chỉ định **<int>** hay **<long>**. Bởi vậy chúng ta có thể viết:

```
int                                                    i,j;
GetMax (i,j);
```

Cách này được sử dụng phổ biến hơn và cũng cho kết quả như vậy:

```
// mẫu hàm II
#include <iostream.h>

template <class T>
T GetMax (T a, T b) {
  return (a>b?a:b);
}

int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax(i,j);
  n=GetMax(l,m);
  cout << z << endl;
  cout << n << endl;
  return 0;
}
```

```
6
10
```

Chú ý trong trường hợp này, bên trong hàm **main()** chúng ta đã gọi hàm mẫu **GetMax()** mà không chỉ định rõ kiểu dữ liệu ở giữa hai ngoặc nhọn **<>**. Trình biên dịch sẽ tự động xác định kiểu dữ liệu nào là cần thiết chô mỗi lời gọi.

Vì hàm mẫu của chúng ta chỉ dùng một kiểu dữ liệu (**class T**) và cả hai tham số của nó đều có cùng một kiểu, chúng ta không thể gọi hàm mẫu với tham số là hai đối tượng có kiểu khác nhau:

```
int i;
long l;
k = GetMax (i,l);
```

Chúng ta cũng có thể khiến cho hàm cho hàm mẫu chấp nhận nhiều hơn một kiểu dữ liệu tổng quát. Ví dụ:

```
template <class T, class U>
T GetMin (T a, U b) {
  return (a<b?a:b);
}
```

Trong trường hợp này, hàm mẫu **GetMin()** chấp nhận hai tham số có kiểu khác nhau và trả về một đối tượng có cùng kiểu với tham số đầu tiên (**T**). Ví dụ, sau khi khai báo như trên chúng ta có thể gọi hàm như sau:

```
int i,j;
long l;
i = GetMin<int,long> (j,l);
```

hoặc thậm chí:

```
i = GetMin (j,l);
```

Các lớp mẫu

Chúng ta cũng có thể viết các lớp mẫu, một lớp mà có các thành viên dựa trên các kiểu dữ liệu tổng quát không cần phải được định nghĩa ở thời điểm tạo lớp. Ví dụ

```
template <class T>
class pair {
    T values [2];
  public:
    pair (T first, T second)
    {
      values[0]=first; values[1]=second;
    }
};
```

Lớp mà chúng ta vừa định nghĩa dùng để lưu trữ hai phần tử có kiểu bất kì. Ví dụ, nếu chúng ta muốn khai báo một đối tượng thuộc lớp này để lưu trữ hai giá trị nguyên kiểu **int** có giá trị là **115** and **36** chúng ta có thể viết:

```
pair<int> myobject (115, 36);
```

lớp này cũng có thể được dùng để lưu trữ bất kì kiểu dữ liệu nào khác:

```
pair<float> myfloats (3.0, 2.18);
```

Hàm thành viên duy nhất được định nghĩa *inline* bên trong định nghĩa của lớp, tuy nhiên nếu chúng ta định nghĩa một hàm thành viên bên ngoài lớp chúng ta luôn phải đặt trước dòng định nghĩa tiền tố **`template <... >`**.

```
// mẫu lớp
#include <iostream.h>

template <class T>
class pair {
    T value1, value2;
  public:
    pair (T first, T second)
                        {value1=first;
value2=second;}
    T getmax ();
};

template <class T>
T pair<T>::getmax ()
{
  T retval;
   retval = value1>value2? value1 :
value2;
  return retval;
}

int main () {
  pair <int> myobject (100, 75);
  cout << myobject.getmax();
  return 0;
}
```

```
100
```

chú ý cách bắt đầu định nghĩa hàm **`getmax`** :

```
template <class T>
T pair<T>::getmax ()
```

Tất cả các **`T`** xuất hiện ở đó đều là cần thiết. Mỗi khi bạn khai báo một hàm thành viên bạn sẽ phải theo một khuôn mẫu tương tự như thế.

Chuyên môn hoá mẫu

Việc chuyên môn hoá mẫu cho phép một mẫu tạo ra những bản thực thi đặc biệt khi làm việc với một loại dữ liệu xác định nào đó. Ví dụ, giả sử rằng lớp mẫu **`pair`** có một hàm dùng để trả về phần dư trong phép chia giữa hai trường ở trong, nhưng chúng ta chỉ muốn nó làm việc khi kiểu dữ liệu là **`int`** còn những kiểu còn lại thì hàm luôn trả về **`0`**. Nó có thể được làm theo cách sau:

```
// Chuyên môn hoá mẫu
#include <iostream.h>

template <class T>
class pair {
    T value1, value2;
  public:
    pair (T first, T second)
                        {value1=first;
```

```
25
0
```

```
value2=second;}
    T module () {return 0;}
};

template <>
class pair <int> {
    int value1, value2;
  public:
    pair (int first, int second)
                      {value1=first;
value2=second;}
    int module ();
};

template <>
int pair<int>::module() {
  return value1%value2;
}

int main () {
  pair <int> myints (100,75);
        pair    <float>    myfloats
(100.0,75.0);
  cout << myints.module() << '\n';
    cout  <<  myfloats.module()  <<
'\n';
  return 0;
}
```

Như bạn có thể thấy sự chuyên môn hoá được định nghĩa theo cách sau:

template <> class *class_name* **<***type***>**

Sự chuyên môn hoá này là một phần của một mẫu, vì vậy chúng ta phải bắt đầu phần khai báo với **template <>**. Và rõ ràng rằng đó là sự chuyên môn hoá cho một kiểu dữ liệu cụ thể nên chúng ta không thể dùng một kiểu dữ liệu tổng quát, cặp ngoặc nhọn **<>** cũng phải để trống. Sau phần tên lớp chúng ta phải viết thêm tên kiểu dữ liệu muốn dùng ở giữa cặp ngoặc nhọn **<>**.

Khi chúng ta chuyên biệt hoá một kiểu dữ liệu cho một mẫu chúng ta cũng phải định nghĩa tất cả các thành viên tương xứng với sự chuyên môn hoá đó (nếu bạn thấy chưa rõ lắm, hãy xem lại ví dụ trên trong đó chúng ta đã phải viết lại constructor cho chính nó mặc dù cái này hoàn toàn giống như constructor ở trong lớp tổng quát. Nguyên nhân là do không có thành viên nào được "thừa kế" từ lớp tổng quát cho lớp chuyên môn hoá.

Các giá trị tham số cho mẫu

Bên cạnh các tham số của mẫu là kiểu dữ liệu (được đứng trước bởi từ khoá **class** hay **typename**), các hàm mẫu và các lớp mẫu còn có thể có các tham số khác không phải là kiểu mà chúng có thể là các giá trị hằng. Trong ví dụ dưới đây lớp mẫu được dùng để lưu trữ mảng:

```
// array template
#include <iostream.h>
```

```
100
3.1416
```

```
template <class T, int N>
class array {
    T memblock [N];
  public:
    setmember (int x, T value);
    T getmember (int x);
};

template <class T, int N>
array<T,N>::setmember   (int   x,   T
value) {
  memblock[x]=value;
}

template <class T, int N>
T array<T,N>::getmember (int x) {
  return memblock[x];
}

int main () {
  array <int,5> myints;
  array <float,5> myfloats;
  myints.setmember (0,100);
  myfloats.setmember (3.0,3.1416);
    cout  <<  myints.getmember(0)  <<
'\n';
   cout  <<  myfloats.getmember(3)  <<
'\n';
  return 0;
}
```

Chúng ta cũng có thể thiết lập các giá trị mặc định cho bất kì tham số mẫu nào giống như với các tham số của hàm bình thường.

Ví dụ:

```
template <class T>                  // Trường hợp phổ biến nhất: một
tham số lớp.
template <class T, class U>     // Hai tham số lớp.
template <class T, int N>       // Một lớp và một giá trị nguyên.
template <class T = char>       // Một tham số lớp với giá trị mặc
định.
template <int Tfunc (int)>      // Tham số là một hàm.
```

Mẫu và các dự án

Theo quan điểm của trình biên dịch, các mẫu không phải là các hàm hay lớp thông thường. Chúng được dịch theo nhu cầu. Điều đó có nghĩa là mã của một hàm mẫu không được dịch cho đến khi cần dùng đến. Với các trình dịch ở thời điểm thực hiện bài viết, trình dịch sẽ tạo ra từ mẫu một hàm cụ thể cho kiểu dữ liệu được yêu cầu.

Khi các dự án phát triển lên, nó thường chi mã của một chương trình thành nhiều file khác nhau. Trong những trường hợp đó, nói chúng phần giao tiếp và phần thực hiện là tách biệt. Lấy ví dụ một thư viện hàm nào đó, phần giao tiếp thường gồm tất cả những khai báo mẫu của các hàm có thể được gọi, chúng thường được khai báo trong một "header file"với phần mở

rộng **.h**, và phần thực hiện (phần định nghĩa cả các hàm đó) được đặt trong các file mã C++ riêng.

Đối với mẫu, chúng ta có một điều bắt buộc khi thực hiện các dự án nhiều file: phần thực hiện (phần định nghĩa) của một lớp mẫu hay hàm mẫu **phải** nằm trong cùng một file với phần khai báo. Điều này có nghĩa là chúng ta không thể tách phần giao tiếp thành một file riêng mà phải gộp cả phần giao tiếp và phần thực thi vào bất kì fiel nào sử dụng mẫu.

Trở lại với trường hợp thư viện hàm, nếu chúng ta muốn tạo một thư viện hàm mẫu, thay vì việc tạo một header file (**.h**) chúng ta nên tạo một "template file" chứa cả phần giao tiếp và phần thực thi của các hàm mẫu (không có một quy định nào về phần mở rộng của những file này, bạn thích dùng thế nào cũng được hoặc cứ để là **.h** cho tiện). Một điều cuối cùng nữa là việc một "template file" được dùng nhiều lần trong một dự án sẽ không gây ra lỗi liên kết vì chúng được dịch theo nhu cầu và những trình biên dịch hỗ trợ templates sẽ biết phải làm thế nào để không sinh mã lặp trong các trường hợp đó.

Bài 5.1 Namespaces

Namespaces cho phép chúng ta gộp một nhóm các lớp, các đối tượng toàn cục và các hàm dưới một cái tên. Nói một cách cụ thể hơn, chúng dùng để chia phạm vi toàn cụ thành những phạm vi nhỏ hơn với tên gọi *namespaces*.

Khuông mẫu để sử dụng *namespaces* là:

```
namespace identifier
{
  namespace-body
}
```

Trong đó *identifier* là bất kì một tên hợp lệ nào và *namespace-body* là một tập hợp những lớp, đối tượng và hàm được gộp trong *namespace*. Ví dụ:

```
namespace general
{
  int a, b;
}
```

Trong trường hợp này, **a** và **b** là những biến bình thường được tích hợp bên trong *namespace* **general**. Để có thể truy xuất vào các biến này từ bên ngoài namespace chúng ta phải sử dụng toán tử **::**. Ví dụ, để truy xuất vào các biến đó chúng ta viết:

```
general::a
general::b
```

Namespace đặc biệt hữu dụng trong trường hợp có thể có một đối tượng toàn cục hoặc một hàm có cùng tên với một cái khác, gây ra lỗi định nghĩa lại. Ví dụ:

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

namespace first
{
  int var = 5;
}

namespace second
```

```
5
3.1416
```

```
{
  double var = 3.1416;
}

int main () {
  cout << first::var << endl;
  cout << second::var << endl;
  return 0;
}
```

Trong ví dụ này có hai biến toàn cục cùng có tên **var**, một được định nghĩa trong *namespace* **first** và cái còn lại nằm trong **second**. Chương trình vẫn chạy ngon, cám ơn *namespaces*.

using namespace

Chỉ thị **using** theo sau là **namespace** dùng để kết hợp mức truy xuất hiện thời với một *namespace* cụ thể để các đối tượng và hàm thuộc *namespace* có thể được truy xuất trực tiếp như thể chúng được khai báo toàn cục. Cách sử dụng như sau:

using namespace *identifier*;

Ví dụ:

```
// using namespace example
#include <iostream.h>

namespace first
{
  int var = 5;
}

namespace second
{
  double var = 3.1416;
}

int main () {
  using namespace second;
  cout << var << endl;
  cout << (var*2) << endl;
  return 0;
}
```

```
3.1416
6.2832
```

Trong trường hợp này chúng ta có thể sử dụng **var** mà không phải đặt trước nó bất kì toán tử phạm vi nào.

Bạn phải để ý một điều rằng câu lệnh **using namespace** chỉ có tác dụng trong khối lệnh mà nó được khai báo hoặc trong toàn bộ chương trình nếu nó được dùng trong phạm vi toàn cục. Ví dụ, nếu chúng ta định đầu tiên sử dụng một đối tượng thuộc một *namespace* và sau đó sử dụng một đối tượng thuộc một *namespace* khác chúng ta có thể làm như sau:

```
// using namespace example
#include <iostream.h>

namespace first
{
```

```
5
3.1416
```

```
  int var = 5;
}

namespace second
{
  double var = 3.1416;
}

int main () {
  {
    using namespace first;
    cout << var << endl;
  }
  {
    using namespace second;
    cout << var << endl;
  }
  return 0;
}
```

Định nghĩa bí danh

Chúng ta cũng có thể định nghĩa những tên thay thế cho các *namespaces* đã được khai báo. Cách thức để làm việc này như sau:

```
namespace new_name = current_name ;
```

Namespace std

Một trong những ví dụ tốt nhất mà chúng ta có thể tìm thấy về *namespaces* chính là bản thân thư viện chuẩn của C++. Theo chuẩn ANSI C++, tất cả định nghĩa của các lớp, đối tượng và hàm của thư viện chuẩn đều được định nghĩa trong *namespace* **std**.

Bạn có thể thấy rằng chúng ta đã bỏ qua luật này trong suốt tutorial này. Tôi đã quyết định làm vậy vì luật này cũng mới như chuẩn ANSI (1997) và nhiều trình biên dịch cũ không tương thích với nó.

Hầu hết các trình biên dịch, thậm chí cả những cái tuân theo chuẩn ANSI, cho phép sử dụng các file header truyền thống (như là **iostream.h**, **stdlib.h**), những cái mà chúng ta trong suốt tutorial này. Tuy nhiên, chuẩn ANSI đã hoàn toàn thiết kế lại những thư viện này để tận dụng lợi thế của tính năng templates và để tuân theo luật phải khai báo tất cả các hàm và biến trong namespace **std**.

Chuẩn ANSI đã chỉ định những tên mới cho những file "header" này, cơ bản là dùng cùng tên với các file của chuẩn C++ nhưng không có phần mở rộng **.h**. Ví dụ, **iostream.h** trở thành **iostream**.

Nếu chúng ta sử dụng các file include của chuẩn ANSI-C++ chúng ta phải luôn nhớ rằng tất cả các hàm, lớp và đối tượng sẽ được khai báo trong **std**. Ví dụ:

```
// ANSI-C++ compliant hello world
#include <iostream>
```

```
Hello world in ANSI-C++
```

```
int main () {
    std::cout  <<  "Hello  world  in
ANSI-C++\n";
  return 0;
}
```

Mặc dù vậy chúng ta nên sử dụng **`using namespace`** để khỏi phải viết toán tử `::` khi tam chiếu đến các đối tượng chuẩn:

```
// ANSI-C++ compliant hello world
(II)
#include <iostream>
using namespace std;

int main () {
   cout << "Hello world in ANSI-C+
+\n";
  return 0;
}
```

```
Hello world in ANSI-C++
```

Bài 5.1 Exception handling

Trong suốt quá trình phát triển một chương trình, có thể có một số trường hợp mà một số đoạn mã chạy sai do truy xuất đến những tài nguyên không tồn tại hay vượt ra ngoài khoảng mong muốn...

Những loại tình huống bất thường này được nằm trong cái được gọi là exceptions và C++ đã vừa tích hợp ba toán tử mới để xử lý những tình huống này: **try**, **throw** và **catch**.

Dạng thức sử dụng như sau:

```
try {
  // đoạn mã cần thử
  throw exception;
}
catch (type  exception)
{
  // đoạn được thực hiện trong trường hợp có lỗi
}
```

Nguyên tắc hoạt động:
- Đoạn mã nằm trong khối **`try`** được thực hiện một cách bình thường. Trong trường hợp có lỗi xảy ra, đoạn mã này phải sử dụng từ khoá **`throw`** và một tham số để báo lỗi. Kiểu tham số này mô tả chi tiết hoá lỗi và có thể là bất kì kiểu hợp lệ nào.
- Nếu có lỗi xảy ra, nếu lệnh **`throw`** đã được thực hiện bên trong khối **`try`**, khối **`catch`** sẽ được thực hiện và nhận tham số được truyền bởi **`throw`**.

Ví dụ:

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

int main () {
  char myarray[10];
  try
  {
    for (int n=0; n<=10; n++)
    {
           if  (n>9)  throw  "Out  of
range";
      myarray[n]='z';
    }
  }
  catch (char * str)
  {
    cout << "Exception: " << str <<
endl;
  }
  return 0;
}
```

```
Exception: Out of range
```

Trong ví dụ này, nếu bên trong vòng lặp mà **n** lớn hơn 9 thì một lỗi sẽ được thông báo vì **myarray[n]** trong trường hợp đó có thể trỏ đến địa chỉ ô nhớ không tin cậy. Khi **throw** được thực hiện, khối **try** ngay lập tức kết thúc và mọi đối tượng được tạo bên trong khối **try** bị phá huỷ. Sau đó, quyền điều khiển được chuyển cho khối **catch** tương ứng (chỉ được thực hiện trong những tình huống như thế này). Cuối cùng chương trình tiếp tục ngay sau khối, trong trường hợp này: **return 0;**.

Cú pháp được sử dụng bởi **throw** tương tự với **return**: Chỉ có một tham số và không cần đặt nó nằm trong cặp ngoặc đơn.

Khối **catch** phải nằm ngay sau khối **try** mà không được có đoạn mã nào nằm giữa chúng. Tham số mà **catch** chấp nhận có thể là bất kì kiểu dữ liệu hợp lệ nào. Hơn nữa, **catch** có thể được quá tải để có thể chấp nhận nhiều kiểu dữ liệu khác nhau. Trong trường hợp này khối **catch** được thực hiện là khối phù hợp với kiểu của tham số được gửi đến bởi **throw**:

```
//   exceptions:   multiple   catch
blocks
#include <iostream.h>

int main () {
  try
  {
    char * mystring;
    mystring = new char [10];
      if  (mystring  ==  NULL)  throw
"Allocation failure";
    for (int n=0; n<=100; n++)
    {
      if (n>9) throw n;
      mystring[n]='z';
    }
  }
  catch (int i)
```

```
Exception: index 10 is out of range
```

```
  {
    cout << "Exception: ";
     cout << "index " << i << " is
out of range" << endl;
  }
  catch (char * str)
  {
     cout << "Exception: " << str <<
endl;
  }
  return 0;
}
```

Ở đây có thể có hai trường hợp xảy ra:

1. Khối dữ liệu 10 kí tự không thể được cấp phát (gần như là chẳng bao giờ xảy ra nhưng không có nghĩa là không thể): lỗi này sẽ bị chặn bởi **`catch (to char * str)`**.
2. Chỉ số cực đại của **`mystring`** đã bị vượt quá: lỗi này sẽ bị chặn bởi **`catch (int i)`**, since parameter is an integer number.

Chúng ta có thể định nghĩa một khối **`catch`** để chặn tất cả các exceptions mà không phụ thuộc vào kiểu được dùng để gọi **`throw`**. Để làm việc này chúng ta phải viết dấu ba chấm thay vì kiểu và tên số tham số:

```
try {
  // code here
}
catch (...) {
  cout << "Exception occurred";
}
```

Còn có thể lồng các khối **`try-catch`** vào các khối **`try`** khác. Trong trường hợp này, một khối **`catch`** bên trong có thể chuyển tiếp exception nhận được cho khối bên ngoài, để làm việc này chúng ta sử dụng biểu thức **`throw;`** không có tham số. Ví dụ:

```
try {
  try {
      // code here
  }
  catch (int n) {
      throw;
  }
}
catch (...) {
  cout << "Exception occurred";
}
```

Exception không bị chặn

Nếu một exception không bị chặn bởi bất kì lệnh **`catch`** nào vì không có lệnh nào có kiểu phù hợp, hàm đặc biệt **`terminate`** sẽ được gọi.

Hàm này đã được định nghĩa sẵn để chấm dứt chương trình ngay lập tức và hiển thịc thông báo lỗi "Abnormal termination". Dạng thức của nó như sau:

```
void terminate();
```

Những exceptions chuẩn

Một số hàm thuộc thư viện C++ chuẩn gửi các exceptions mà chúng ta có thể chặn nếu chúng ta sử dụng một khối **try**. Những exceptions này được gửi đi với kiểu tham số là một lớp thừa kế từ **std::exception**. Lớp này (std::exception) được định nghĩa trong file header C++ chuẩn **<exception>** và được dùng làm mẫu cho hệ thống phân cấp các exception chuẩn:

```
exception
  bad_alloc       (gửi bởi new)
  bad_cast        (gửi bởi dynamic_cast khi thất bại với một kiểu
                  tham chiếu)
  bad_exception   (được gửi khi một exception không phù hợp với
                  lệnh catch nào)
  bad_typeid      (gửi bởi typeid)
  logic_error
  domain_error
  ...
invalid_argument
  length_error
  out_of_range
  runtime_error
  overflow_error
  range_error
  underflow_error
  ...
ios_base::failure (gửi bởi ios::clear)
```

Bởi vì đây là một hệ thống phân lớp có thứ bậc, nếu bạn sử dụng một khối **catch** để chặn bất kì một exception nào nằm trong hệ thông này bằng cách sử dụng tham số biến (thêm một dấu & vào phía trước tên của tham số) bạn sẽ chặn được tất cả các exception thừa kế (luật thừa kế trong C++)

Ví dụ dưới đây chặn một exception có kiểu **bad_typeid** (được thừa kế từ **exception**), lỗi này được tạo ra khi muốn biết kiểu của một con trỏ null.

```
// Những exception chuẩn

#include <iostream.h>
#include <exception>
#include <typeinfo>

class A {virtual f() {}; };

int main () {
  try {
    A * a = NULL;
    typeid (*a);
  }
  catch (std::exception& e)
  {
        cout  <<  "Exception:  "  <<
```

```
Exception: Attempted typeid of NULL
pointer
```

```
e.what();
  }
  return 0;
}
```

Bạn có thể sử dụng các lớp của hệ thống phân cấp các exception chuẩn này báo những lỗi của mình hoặc thừa kế những lớp mới từ chúng.

Bài 5.4 Chuyển đổi kiểu nâng cao

Cho đến nay, để có thể chuyển đổi kiểu một đối tượng đơn giản sang kiểu khác chúng ta đã sử dụng toán tử chuyển đổi kiểu truyền thống. Ví dụ, để chuyển một số dấu phẩy động có kiểu **double** sang dạng số nguyên có kiểu **int** chúng ta sử dụng:

```
int i;
double d;
i = (int) d;
```

hoặc

```
i = int (d);
```

Điều này làm việc tốt đối với các kiểu cơ bản đã có định nghĩa các cacchs chuyển đổi cơ bản, tuy nhiên những toán tử này cũng có thể được áp dụng bừa bãi với các lớp và con trỏ tới các lớp. Bởi vậy, hoàn toàn hợp lệ khi viết như sau:

```
// class type-casting
#include <iostream.h>

class CDummy {
    int i;
};

class CAddition {
        int x,y;
  public:
        CAddition (int a, int b)
{ x=a; y=b; }
        int result() { return x+y;}
};

int main () {
  CDummy d;
  CAddition * padd;
  padd = (CAddition*) &d;
  cout << padd->result();
  return 0;
}
```

Mặc dù chương trình trên là hợp lệ trong C++ (thực tế là nó sẽ được dịch mà không có bất kì một lỗi hay warning nào đối với hầu hết các trình dịch) nhưng nó là một đoạn mã không tốt lắm vì chúng ta sử dụng hàm **result**, đó là một thành viên của **CAddition**, **padd** không phải là một đối tượng, nó chỉ là một con trỏ được chúng ta gán cho địa chỉ của một đối tượng

không có quan hệ. Khi truy xuất đến thành viên **result**, chương trình sẽ tạo ra lỗi run-time hoặc chỉ là một kết quả không mong muốn.

Để có thể điều khiển việc chuyển đổi kiểu giữa các lớp, chuẩn ANSI-C++ đã định nghĩa bốn toán tử chuyển đổi kiểu mới: **reinterpret_cast**, **static_cast**, **dynamic_cast** và **const_cast**. Tất cả chúng đều có cùng dạng thức khi sử dụng:

```
reinterpret_cast <new_type> (expression)
    dynamic_cast <new_type> (expression)
     static_cast <new_type> (expression)
      const_cast <new_type> (expression)
```

Trong đó *new_type* kiểu mà *expression* phải được chuyển đổi thành. Để tạo ra sự tương tự dễ hiểu với các toán tử chuyển đổi truyền thống các biểu thức này có nghĩa là:

```
(new_type) expression
new_type (expression)
```

reinterpret_cast

reinterpret_cast chuyển đổi một con trỏ sang bất kì kiểu con trỏ nào khác. Nó cũng cho phép chuyển đổi từ con trỏ sang dạng số nguyên và ngược lại.

Toán tử này có thể chuyển đổi con trỏ giữa các lớp không có quan hệ với nhau. Kết quả của toán tử này là một bản copy giá trị của con trỏ này sang con trỏ kia. Dữ liệu được trỏ đến không hề được kiểm tra hay chuyển đổi.

Trong trường hợp chuyển đổi giữa con trỏ và số nguyên, cách chuyển nội dung của nó phụ thuộc vào hệ thống.

```
class A {};
class B {};
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

reinterpret_cast đối xử với tất cả các con trỏ giống như các toán tử chuyển đổi truyền thống.

static_cast

static_cast cho phép thực hiện bất kì phép chuyển đổi nào

static_cast allows to perform any casting that can be implicitly performed as well as also the inverse cast (even if this is not allowed implicitly).

Applied to pointers to classes, that is to say that it allows to cast a pointer of a derived class to its base class (this is a valid conversion that can be implicitly performed) and can also perform the inverse: cast a base class to its derivated class.

In this last case the base class that is being casted is not checked to determine wether this is a complete class of the destination type or not.

```
class                          Base                          {};
class         Derived:          public          Base         {};
Base         *          a          =          new          Base;
Derived * b = static_cast<Derived*>(a);
```

static_cast, aside from manipulating pointers to classes, can also be used to perform conversions explicitly defined in classes, as well as to perform standard conversions between fundamental types:

```
double                                                d=3.14159265;
int i = static_cast<int>(d);
```

dynamic_cast

dynamic_cast is exclusively used with pointers and references to objects. It allows any type-casting that can be implicitly performed as well as the inverse one when used with polymorphic classes, however, unlike **static_cast**, **dynamic_cast** checks, in this last case, if the operation is valid. That is to say, it checks if the casting is going to return a valid complete object of the requested type.

Checking is performed during run-time execution. If the pointer being casted is not a pointer to a valid complete object of the requested type, the value returned is a **NULL** pointer.

```
class      Base      {      virtual      dummy(){};      };
class      Derived      :      public      Base      {      };

Base*          b1          =          new          Derived;
Base*           b2           =           new           Base;
Derived*  d1  =  dynamic_cast<Derived*>(b1);     // succeeds
Derived*  d2  =  dynamic_cast<Derived*>(b2);     // fails:
returns NULL
```

If the type-casting is performed to a reference type and this casting is not possible an *exception* of type **bad_cast** is thrown:

```
class      Base      {      virtual      dummy(){};      };
class      Derived      :      public      Base      {      };

Base*          b1          =          new          Derived;
Base*           b2           =           new           Base;
Derived  d1  =  dynamic_cast<Derived&*>(b1);     // succeeds
Derived  d2  =  dynamic_cast<Derived&*>(b2);     // fails:
exception thrown
```

const_cast

This type of casting manipulates the *const* attribute of the passed object, either to be set or removed:

```
class                          C                          {};
const       C       *       a       =       new       C;
C * b = const_cast<C*> (a);
```

Neither of the other three new **cast** operators can modify the constness of an object.

typeid

ANSI-C++ also defines a new operator called **typeid** that allows to check the type of an expression:

typeid (*expression***)**

this operator returns a refernece to a constant object of type **type_info** that is defined in standard header file **<typeinfo>**. This returned value can be compared with another using operators == and != or can serve to obtain a string of characters representing the data type or class name by using its **name()** method.

```
// typeid, typeinfo
#include <iostream.h>
#include <typeinfo>

class CDummy { };

int main () {
  CDummy* a,b;
  if (typeid(a) != typeid(b))
  {
       cout << "a and b are of
different types:\n";
         cout << "a is: " <<
typeid(a).name() << '\n';
         cout << "b is: " <<
typeid(b).name() << '\n';
  }
  return 0;
}
```

```
a and b are of different types:
a     is:     class     CDummy     *
b is: class CDummy
```