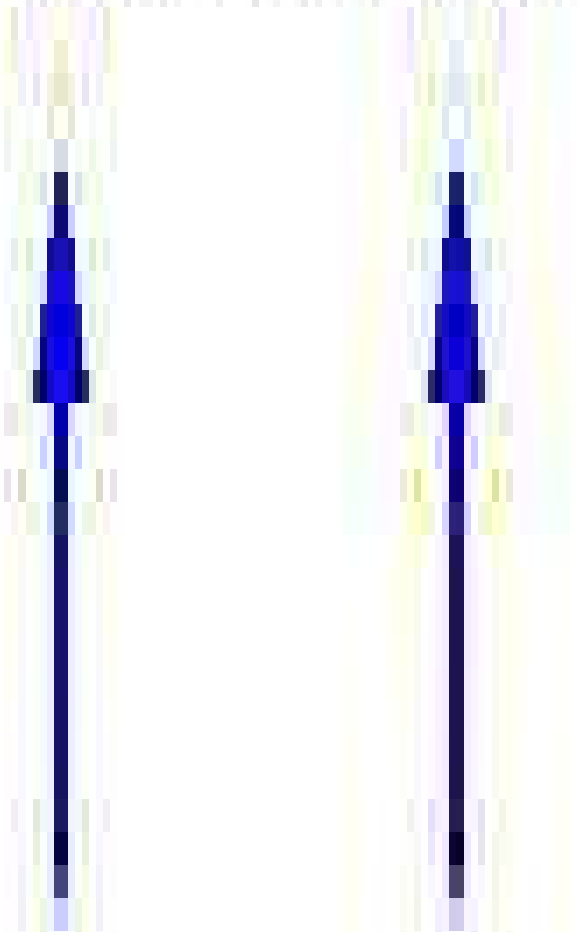
```
int addition (int a, int b)
```



```
z = addition ( 5 , 3 );
```

Giáo trình C++

Khuyết Danh

Chào mừng các bạn đón đọc đầu sách từ dự án sách cho thiết bị di động
Nguồn: http://vnthuquan.net
Phát hành: Nguyễn Kim Vỹ.

Mục lục

Khuyết Danh

Giáo trình C++

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 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
```

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` 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

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

Khuyết Danh

Giáo trình C++

1/ Cơ bản về C++

Bài 1.2

Các biến, kiểu và hằng số

Khuyết Danh

Giáo trình C++

1/ Cơ bản về C++

Bài 1.3

Các toán tử

Khuyết Danh

Giáo trình C++

1/ Cơ bản về C++

Bài 1.4

Giao tiếp với console

Khuyết Danh

Giáo trình C++

2/ Các cấu trúc điều khiển và hàm

Bài 2.1

Các cấu trúc điều khiển

Khuyết Danh

Giáo trình C++

2/ Các cấu trúc điều khiển và hàm

Bài 2.2

Hàm (I)

function open() {return false;}

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 ; return 0;}

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

Trong trường hợp này chúng ta tạo ra hàm **subtraction**. Chức năng của hàm này là lấy hiệu của hai tham số rồi trả về kết quả.
Tuy nhiên, nếu phân tích hàm **main** các bạn sẽ thấy chương trình đã vài lần gọi đến hàm **subtraction**. Tôi đã sử dụng vài cách gọi khác nhau để các bạn thấy các cách khác nhau mà một hàm có thể được gọi.
Để có hiểu cặn kẽ ví dụ này bạn cần nhớ rằng một lời gọi đến một hàm có thể hoàn toàn được thay thế bởi giá trị của nó. Ví dụ trong lệnh gọi hàm đầu tiên :

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

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

```
z = 5;
```

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

Tương tự như vậy

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

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

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

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

Còn với lệnh

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

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

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

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

chúng ta có thể viết:

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

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

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

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

type name (argument1, argument2 ...) statement

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

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

// void function example#include <iostream.h> void dummyfunction (void){ cout << "I m a function!";} int main (){ dummyfunction (); return 0;}

```
I m a function!
```

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

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

```
dummyfunction ();
```

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.

Khuyết Danh

Giáo trình C++

2/ Các cấu trúc điều khiển và hàm

Bài 2.3

Hàm (II)

Khuyết Danh

Giáo trình C++

3/ Dữ liệu nâng cao

Bài 3.1

Mảng

Khuyết Danh

Giáo trình C++

3/ Dữ liệu nâng cao

Bài 3.2

Xâu kí tự

function open() {return false;}

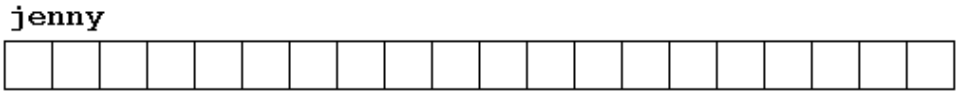
Trong tất cả các chương trình chúng ta đã thấy cho đến giờ, chúng ta chỉ sử dụng các biến kiểu số, chỉ dùng để biểu diễn các số. Nhưng bên cạnh các biến kiểu số còn có các xâu kí tự, chúng cho phép chúng ta biểu diễn các chuỗi kí tự như là các từ, câu, đoạn văn bản... Cho đến giờ chúng ta mới chỉ dùng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

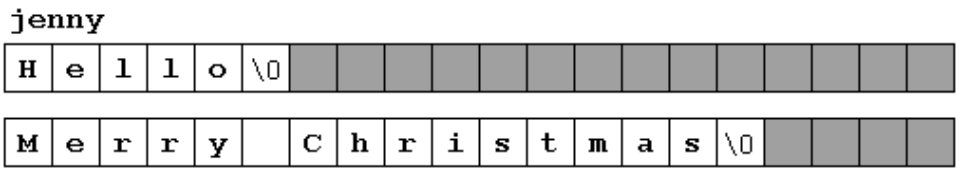
```
char jenny [20];
```

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;
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[]`**

Khuyết Danh

Giáo trình C++

3/ Dữ liệu nâng cao

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 (&).

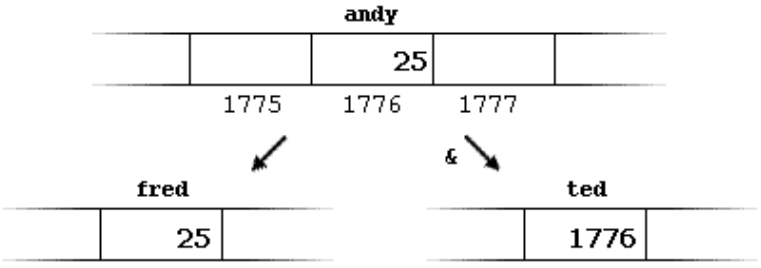
Vào thời điểm mà chúng ta khai báo một biến thì nó phải được lưu trữ trong một vị trí cụ thể trong bộ nhớ. Nói chung chúng ta không quyết định nơi nào biến đó được đặt - thật may mắn rằng điều đó đã được làm tự động bởi trình biên dịch và hệ điều hành, nhưng một khi hệ điều hành đã gán một địa chỉ cho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

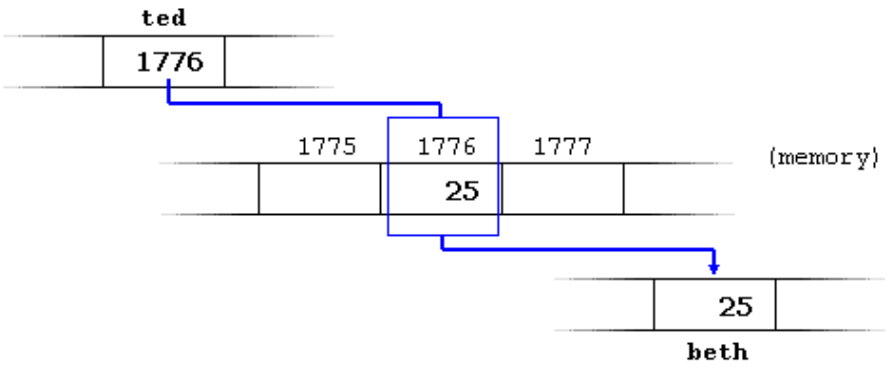
được gọi là **con trỏ**. Trong C++ con trỏ có rất nhiều ưu điểm và chúng được sử dụng rất thường xuyên, Tiếp theo chúng ta sẽ thấy các biến kiểu này được khai báo như thế nào.

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 đặt trước tên biến con trỏ một dấu sao (*) - ở đây có thể được dịch là "**giá trị được trỏ bởi**". Vì vậy, nếu chúng ta viết:

```
beth = *ted;
```

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



Bạn phải phân biệt được rằng **ted** có giá trị **1776**, nhưng ***ted** (với một dấu sao đằng trước) 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, không phải là kiểu của bản thân con trỏ. Ví dụ:

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

đó là ba khai báo của con trỏ. Mỗi biến 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 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;
}
```

value1==10 / value2==20

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

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

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

Con trỏ và mảng.

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

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

lệnh sau sẽ hợp lệ:

```
p = numbers;
```

Ở đây **p** và **numbers** là tương đương và chúng có 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;
```

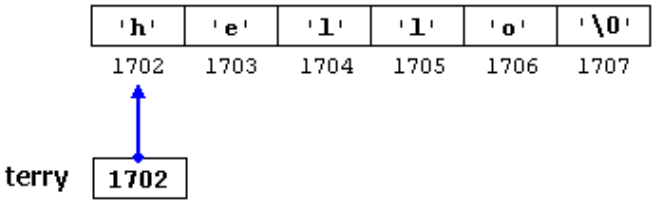
Trong một phép gán con trỏ chúng ta phải luôn luôn gán địa chỉ mà nó trỏ tới chứ không phải là giá trị mà nó trỏ tới. Bạn cần phải nhớ rằng khi khai báo một biến con trỏ, dấu sao (`*`) được dùng để chỉ ra nó là một con trỏ, và hoàn toàn khác với toán tử tham chiếu. Đó là hai toán tử khác nhau mặc dù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

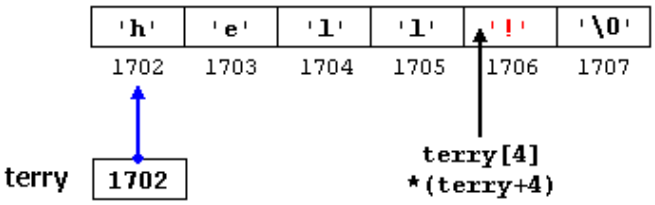


cần phải nhắc lại rằng **terry** mang giá trị **1702** chứ không phải là **h** hay **"hello"**.

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++;
```

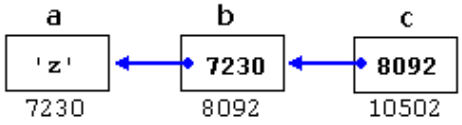
Như đã nói trong các bài trước, tôi khuyên các bạn nên dùng các cặp ngoặc đơn để tránh những kết quả không mong muốn.

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
#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;
    }
```

6, 10, 13

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

Con trỏ hàm

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

```
// pointer to functions
#include <iostream.h>
int addition (int a, int b)
{ return (a+b); }
int subtraction (int a, int b)
{ return (a-b); }
int (*minus)(int,int) = subtraction;
int operation (int x, int y, int (*functocall)(int,int))
{
int g;
g = (*functocall)(x,y);
return (g);
}
int main ()
{
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;
```

Khuyết Danh

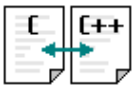
Giáo trình C++

3/ Dữ liệu nâng cao

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

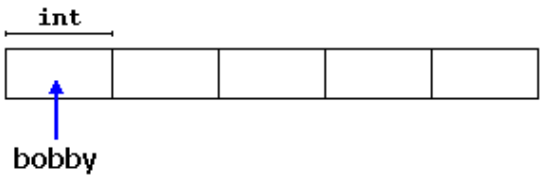
```
pointer = new type [elements]
```

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***.

```
function open() {return false;}
```

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);
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";
}
```

Enter title: Alien
Enter year: 1979

My favourite movie is:

```
 2001 A Space Odyssey (1968)
And yours:
 Alien (1979)
```

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: ";
    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";
                }
```

```
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)

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);
    pmovie->year = atoi (buffer);
    cout << "\nYou have entered:\n";
    cout << pmovie->title;
    cout << " (" << pmovie->year << ")\n";
    return 0;
}
```

```
Enter title: Matrix
Enter year: 1999

You have entered:
Matrix (1999)
```

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). Bạn sẽ có thêm các thông tin chi tiết trong phần 4.1, Lớp.

Khuyết Danh

Giáo trình C++

3/ Dữ liệu nâng cao

Bài 3.6

Các kiểu dữ liệu tự định nghĩa

function open() {return false;}

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
union vô danh

```
    struct {
 char title[50];
char author[50];
     union {
  float dollars;
    int yens;
     } price;
     } book;

    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 đựoc 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`.

Khuyết Danh

Giáo trình C++

4/ Lập trình hướng đối tượng

Bài 4.1

Các lớp

Khuyết Danh

Giáo trình C++

4/ Lập trình hướng đối tượng

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) {
  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;
}
```

4,3

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);
};
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;
}
```

yes, &a is b

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];
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.

Khuyết Danh

Giáo trình C++

4/ Lập trình hướng đối tượng

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 * 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);
};
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;
}
```

16

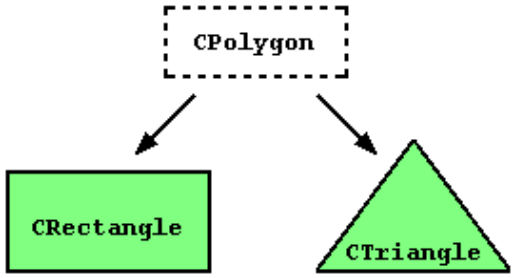
Trong ví dụ này chúng ta đã khai báo **CRectangle** là bạn của **CSquare** nên **CRectangle** có thể truy xuất vào các thành viên **protected** and **private** của **CSquare**, cụ thể hơn là **CSquare::side**, biến định nghĩa kích thước của hình vuông.

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

20

10

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);

return 0;
}
```

mother: no parameters
daughter: int parameter

mother: int parameter
son: int parameter

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, 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;
}
```

20

10

Khuyết Danh

Giáo trình C++

4/ Lập trình hướng đối tượng

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 {
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;
            }
```

```
20
10
```

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:
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;
    }
```

```
20
10
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;
cout << ppoly2->area() << endl;
          return 0;
              }
```

```
20
10
```

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.

Khuyết Danh

Giáo trình C++

5/ Phần nâng cao

Bài 5.1

Templates

Khuyết Danh

Giáo trình C++

5/ Phần nâng cao

Bài 5.2

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
{
double var = 3.1416;
}
int main () {
cout << first::var << endl;
cout << second::var << endl;
return 0;
}
```

```
5
3.1416
```

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
{
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;
}
```

```
5
3.1416
```

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:

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

Hello world in ANSI-C++

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++

Tên của các file C cũng có một số thay đổi. Bạn có thể tìm thêm thông tin về tên mới của các file header chuẩn trong tài liệu Các file header chuẩn.

Khuyết Danh

Giáo trình C++

Bài 5.3

Exception handling

Exception handling là một tính năng mới được giới thiệu bởi chuẩn ANSI-C++. Nếu bạn sử dụng một trình biên dịch C++ không tương thích với chuẩn ANSI C++ thì bạn không thể sử dụng tính năng này.

Lời cuối: Cám ơn bạn đã theo dõi hết cuốn truyện.
Nguồn: http://vnthuquan.net
Phát hành: Nguyễn Kim Vỹ.

Nguồn:
Được bạn: mickey đưa lên
vào ngày: 18 tháng 8 năm 2004