

Con trỏ trong C++

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.

1. 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:

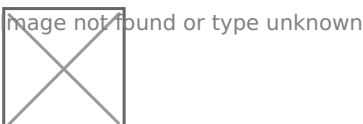
```
ted = &andy;
```

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.

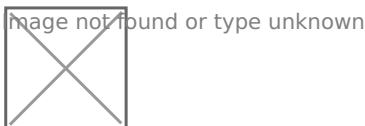
Những biến lưu trữ địa chỉ của một biến khác (như `ted` ở trong ví dụ trước) được gọi là **con trỏ**. Trong C++ con trỏ có rất nhiều ưu điểm và chúng được sử dụng rất thường xuyên, Tiếp theo chúng ta sẽ thấy các biến kiểu này được khai báo như thế nào.

2. 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
```

3. 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;
```

```
}
```

Kết quả:

```
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;
}
```

Kết quả:

```
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.

4. Con trỏ và mảng

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

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

lệnh sau sẽ hợp lệ:

```
p = numbers;
```

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

```
numbers = p;
```

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

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

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

Kết quả:

```
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 trở đến. Cặp ngoặc vuông này được coi như là toán tử offset và ý nghĩa của chúng không đổi khi được dùng với biến con trỏ. Ví dụ, hai biểu thức sau đây:

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

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

5. Khởi tạo con trỏ

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

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

là tương đương với:

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

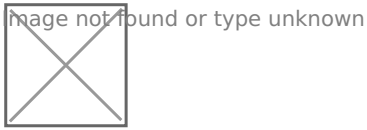
Trong một phép gán con trỏ chúng ta phải luôn luôn gán địa chỉ mà nó trỏ tới chứ không phải là giá trị mà nó trỏ tới. Bạn cần phải nhớ rằng khi khai báo một biến con trỏ, dấu sao (*) được dùng để chỉ ra nó là một con trỏ, và hoàn toàn khác với toán tử tham chiếu. Đó là hai toán tử khác nhau mặc dù chúng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

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

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

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

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

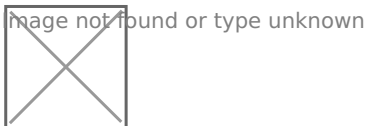


cần phải nhắc lại rằng terry mang giá trị 1702 chứ không phải là 'h' hay "hello".

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

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

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



6. 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.

7. 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'
```

8. Con trỏ không kiểu

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

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

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

```

Kết quả:

```
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.

9. 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;
}
```

Kết quả:

8

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

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

Revision #1

Created 5 October 2019 15:18:21 by Laptrinh.vn

Updated 5 October 2019 15:40:38 by Laptrinh.vn