NGÔN NGỮ LẬP TRÌNH

Bài 8: Con trỏ và mảng động

Giảng viên: Lý Anh Tuấn
Email: tuanla@tlu.edu.vn

Nội dung

1. Con trỏ
   - Các biến con trỏ
   - Quản lý bộ nhớ
2. Mảng động
   - Tạo và sử dụng
   - Phép tính con trỏ
3. Lớp, con trỏ, mảng động
   - Con trỏ *this*
   - Hàm hủy, hàm tạo sao chép

Giới thiệu

- Định nghĩa con trỏ
  - Địa chỉ bộ nhớ của một biến
- Nhắc lại: bộ nhớ được phân chia
  - Các vị trí bộ nhớ được đánh số
  - Địa chỉ được sử dụng làm tên của biến
- Chúng ta đã sử dụng con trỏ rồi
  - Tham số truyền tham chiếu
  - Địa chỉ của đối số thực sự được truyền

Biến con trỏ

- Các con trỏ được định kiểu
  - Có thể lưu trữ con trỏ trong biến
  - Không phải int, double mà là một con trỏ trỏ tới int, double, vân vân
- Ví dụ:
  double *p;
  - p được khai báo là một biến con trỏ trỏ tới double
  - Có thể lưu giữ các con trỏ trỏ tới các biến kiểu double

Khai báo biến con trỏ

- Con trỏ được khai báo giống các kiểu khác
  - Thêm "*" trước tên biến
  - Tạo ra con trỏ trỏ đến kiểu đó
- "*" phải được đặt trước mỗi biến
- int *p1, *p2, v1, v2;
  - p1, p2 lưu trữ con trỏ trỏ tới các biến int
  - v1, v2 là các biến nguyên nguyên bản

Địa chỉ và số

- Con trỏ là một địa chỉ
- Địa chỉ là một số nguyên
- Con trỏ không phải là một số nguyên
- C++ buộc các con trỏ được sử dụng làm địa chỉ
  - Không thể được sử dụng như số
  - Thậm chí nó “là một” số

Trỏ tới

- int *p1, *p2, v1, v2;
  p1 = &v1;
  - Thiết lập biến con trỏ p1 trỏ tới biến int v1
- Toán tử, &
  - Xác định địa chỉ của biến
- Các đọc:
  - “p1 bằng địa chỉ của v1”
  - Hoặc “p1 trỏ tới v1”

Trỏ tới

- int *p1, *p2, v1, v2;
  p1 = &v1;
- Có hai cách để tham chiếu đến v1:
  - Bằng bản thân biến v1:
    cout << v1;
  - Bằng con trỏ p1:
    cout << *p1;
- Toán tử khử tham chiếu, *
  - Biến con trỏ được khử tham chiếu
  - Nghĩa là: “Lấy dữ liệu mà p1 trỏ tới”

Ví dụ trỏ tới

- Xét:
  v1 = 0;
  p1 = &v1;
  *p1 = 42;
  cout << v1 << endl;
  cout << *p1 << endl;
- Sinh ra giá trị đầu ra
  42
  42
- p1 và v1 tham chiếu đến cùng một biến

Toán tử &

- Toán tử lấy địa chỉ
- Cũng được sử dụng để truyền tham biến
  - Không như nhau
  - Nhắc lại: tham số truyền tham biến truyền địa chỉ của tham số thực sự
- Hai trường hợp sử dụng toán tử liên quan chặt chẽ với nhau

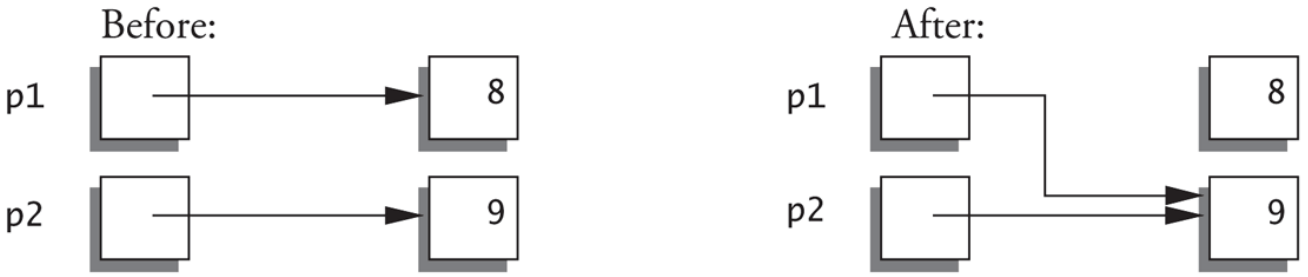
Phép gán con trỏ

- Biến con trỏ có thể được gán:
  int *p1, *p2;
  p2 = p1;
  - Gán một con trỏ cho một bằng một con trỏ khác
  - Làm cho p2 trỏ tới nơi p1 trỏ tới
- Không được nhầm lẫn với:
  *p1 = *p2;
  - Gán giá trị được trỏ tới bởi p1, cho giá trị được trỏ tới bởi p2

Phép gán con trỏ

Display 10.1 Uses of the Assignment Operator with Pointer Variables

```
p1 = p2;
```




```
*p1 = *p2;
```

Toán tử new

- Vì con trỏ có thể tham chiếu tới biến
  - Không thực sự cần có một định danh chuẩn
- Có thể cấp phát động biến
  - Toán tử new tạo ra biến
    - Không có định danh cho nó
    - Chỉ có một con trỏ
  - p1 = new int;
    - Tạo biến khuyết danh, và gán p1 trỏ đến nó
    - Có thể truy cập bằng *p1, sử dụng giống như biến nguyên bản

Ví dụ về thao tác con trỏ

Display 10.2 Basic Pointer Manipulations

```
1   //Program to demonstrate pointers and dynamic variables.
2   #include <iostream>
3   using std::cout;
4   using std::endl;


5   int main( )
6   {
7       int *p1, *p2;


8       p1 = new int;
9       *p1 = 42;
10      p2 = p1;
11      cout << "*p1 == " << *p1 << endl;
12      cout << "*p2 == " << *p2 << endl;


13      *p2 = 53;
14      cout << "*p1 == " << *p1 << endl;
15      cout << "*p2 == " << *p2 << endl;
```

Ví dụ về thao tác con trỏ

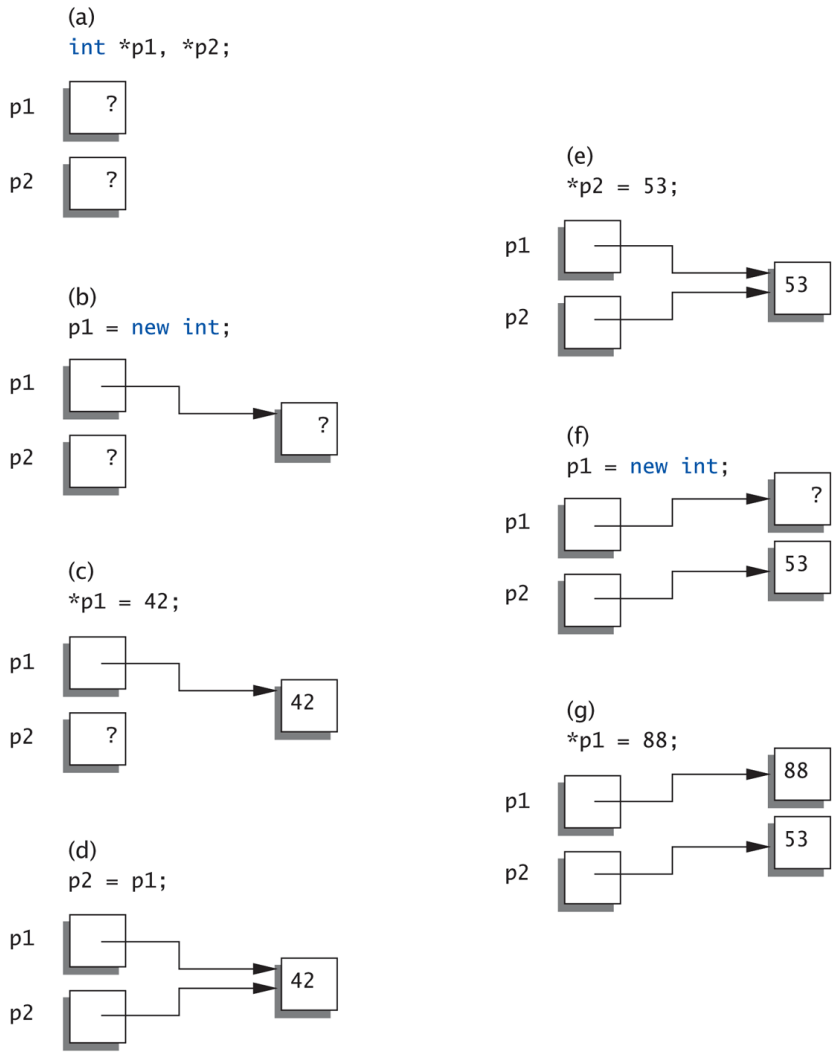
```
16        p1 = new int;
17        *p1 = 88;
18        cout << "*p1 == " << *p1 << endl;
19        cout << "*p2 == " << *p2 << endl;


20        cout << "Hope you got the point of this example!\n";
21        return 0;
22    }
```

SAMPLE DIALOGUE

```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
Hope you got the point of this example!
```

Thao tác con trỏ: Giải thích ví dụ

Display 10.3 Explanation of Display 10.2



Toán tử new

- Tạo biến động mới
- Trả về con trỏ trỏ tới biến mới
- Nếu kiểu là kiểu lớp:
  - Gọi hàm tạo cho đối tượng mới
  - Có thể gọi hàm tạo với các đối số khởi tạo:
    MyClass *mcPtr;
    mcPtr = new MyClass(32.0, 17);
- Cũng có thể khởi tạo các kiểu không phải lớp:

```
int *n;
n = new int(17);        //Khởi tạo *n bằng 17
```

Con trỏ và Hàm

- Con trỏ là kiểu đầy đủ
  - Có thể được sử dụng giống như các kiểu khác
- Có thể là tham số hàm
- Có thể được trả về từ hàm
- Ví dụ:
  int* findOtherPointer(int* p);
  - Khai báo hàm này:
    - Có tham số con trỏ trỏ tới một tham số int
    - Trả về con trỏ trỏ tới một biến int

Quản lý bộ nhớ

- Heap
  - Còn được gọi là "freestore"
  - Dành chỗ cho các biến được cấp phát động
  - Tất cả các biến động mới chiếm vùng nhớ trong freestore
    - Nếu quá nhiều → có thể sử dụng tất cả bộ nhớ freestore
- Thao tác new tương lai sẽ thất bại nếu freestore đầy

Kiểm tra new thành công

- Các bộ biên dịch cũ:
  - Kiểm tra xem lời gọi tới new có trả về null hay không

```
int *p;
p = new int;
if (p == NULL)
{
    cout << “Loi:  Thieu bo nho.\n";
    exit(1);
}
```

  - Nếu new thành công, chương trình tiếp tục
- Các bộ biên dịch mới hơn:
  - Nếu thao tác new thất bại: chương trình tự động kết thúc và hiển thị thông báo lỗi

Toán tử delete

- Hủy cấp phát vùng nhớ động
  - Khi nó không còn cần thiết nữa
  - Trả lại vùng nhớ cho freestore
  - Ví dụ:

    ```
    int *p;
    p = new int(5);
    … //Mo so xu ly…
    delete p;
    ```

  - Hủy cấp phát vùng nhớ động được trỏ đến bởi con trỏ p

Con trỏ thừa

- delete p;
  - Hủy vùng nhớ động
  - Nhưng p vẫn trỏ đến đó
    - Được gọi là “con trỏ thừa”
  - Nếu sau đó p được khử tham chiếu (*p)
    - Không thể đoán được kết quả
- Tránh các con trỏ thừa
  - Gán con trỏ bằng NULL sau khi xóa:
    delete p;
    p = NULL;

Biến động và biến tự động

- Biến động
  - Được tạo bởi toán tử new
  - Được tạo và được hủy khi chương trình chạy
- Biến cục bộ
  - Được khai báo trong định nghĩa hàm
  - Không động
    - Được tạo khi hàm được gọi
    - Được hủy khi lời gọi hàm kết thúc
  - Thường được gọi là biến tự động

Định nghĩa kiểu con trỏ

- Có thể đặt tên các kiểu con trỏ
- Để có thể khai báo con trỏ giống như các biến khác
  - Loại bỏ đòi hỏi "*" trong khai báo con trỏ
- typedef int* IntPtr;
  - Định nghĩa một bí danh kiểu mới
  - Xét các khai báo:
    IntPtr p;
    int *p;
    - Hai khai báo này là tương đương

Lỗi thường gặp: Con trỏ truyền giá trị

- Ứng xử khó hiểu và rắc rối
  - Nếu hàm thay đổi tham số con trỏ của nó → thay đổi duy nhất là với bản sao cục bộ
- Xem ví dụ sau đây:

Ví dụ con trỏ truyền giá trị

Display 10.4 A Call-by-Value Pointer Parameter

```
1   //Program to demonstrate the way call-by-value parameters
2   //behave with pointer arguments.
3   #include <iostream>
4   using std::cout;
5   using std::cin;
6   using std::endl;

7   typedef int* IntPointer;

8   void sneaky(IntPointer temp);

9   int main( )
10  {
11      IntPointer p;

12      p = new int;
13      *p = 77;
14      cout << "Before call to function *p == "
15           << *p << endl;
```

Ví dụ con trỏ truyền giá trị

```
16        sneaky(p);

17        cout << "After call to function *p == "
18             << *p << endl;

19        return 0;
20   }
21   void sneaky(IntPointer temp)
22   {
23        *temp = 99;
24        cout << "Inside function call *temp == "
25             << *temp << endl;
26   }
```

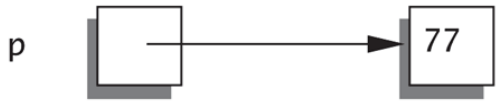
SAMPLE DIALOGUE

Before call to function *p == 77
Inside function call *temp == 99
After call to function *p == 99

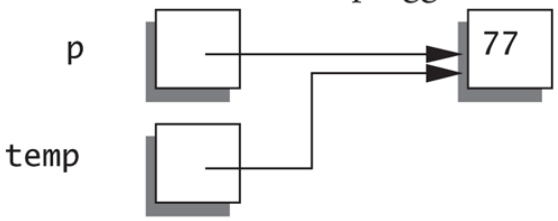
Con trỏ truyền giá trị

Display 10.5 **The Function Call `sneaky(p);`**

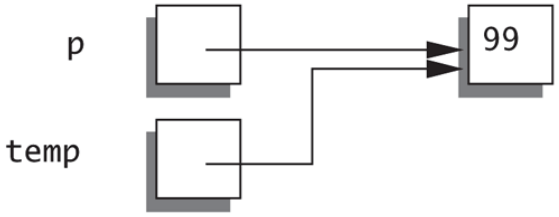
1. Before call to `sneaky`:



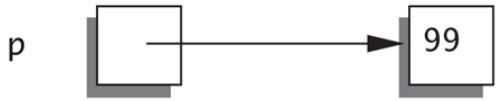
2. Value of `p` is plugged in for `temp`:



3. Change made to `*temp`:



4. After call to `sneaky`:



Mảng động

- Biến mảng
  - Biến con trỏ thực sự
- Mảng chuẩn
  - Kích thước cố định
- Mảng động
  - Kích thước không được chỉ ra ở thời điểm lập trình
  - Được quyết định khi chương trình chạy

Biến mảng

- Mảng được lưu trữ trong các địa chỉ bộ nhớ một cách tuần tự
  - Biến mảng tham chiếu đến biến được đánh chỉ số đầu tiên
  - Do vậy biến mảng là một kiểu biến con trỏ
- Ví dụ:
  int a[10];
  int * p;
  - Cả a và p đều là biến con trỏ

Biến mảng → Con trỏ

- Ví dụ:
  int a[10];
  typedef int* IntPtr;
  IntPtr p;
- a và p là các biến con trỏ
  - Có thể thực hiện phép gán:
    p = a; // Hợp lệ.
    - Bây giờ p trỏ tới chỗ a trỏ tới
  - a = p; // Không hợp lệ
    - Con trỏ mảng là một con trỏ hằng

Biến mảng → Con trỏ

- Biến mảng
  int a[10];
- Con trỏ hằng
  - Kiểu “const int *”
  - Mảng đã được cấp phát bộ nhớ
  - Biến là phải luôn luôn trỏ đến đó và không thể thay đổi
- Đối lập với con trỏ nguyên bản
  - Có thể (và thường xuyên) thay đổi

Mảng động

- Các hạn chế của mảng
  - Trước hết phải chỉ rõ kích thước
  - Có thể không biết cho đến khi chương trình chạy
- Phải ước lược kích thước lớn nhất cần thiết
  - Đôi khi có thể, đôi khi không
  - Lãng phí bộ nhớ
- Mảng động
  - Có thể tăng thêm và co lại khi cần

Tạo mảng động

- Sử dụng toán tử new
  - Cấp phát động bằng biến con trỏ
  - Xử lý giống như các mảng chuẩn
- Ví dụ:

```
typedef double * DoublePtr;
DoublePtr d;
d = new double[10];  //Kich thuoc trong ngoac vuong
```

  - Tạo biến mảng cấp phát động d, có mười phần tử, kiểu cơ sở là double

Xóa mảng động

- Được cấp phát động ở thời điểm chạy
  - Do vậy nên được hủy ở thời điểm chạy
- Ví dụ:

```
d = new double[10];
… //Xu ly
delete [] d;
```

  - Hủy cấp phát tất cả bộ nhớ của mảng động
  - Dấu ngoặc vuông chỉ ra đó là mảng
  - Nhắc lại: d vẫn tiếp tục trỏ đến đó
    - Nên gán d = NULL;

Hàm trả về một mảng

- Kiểu mảng không được phép là kiểu trả về của hàm
- Ví dụ:
  int [] someFunction(); // Khong hop le
- Thay vì vậy trả về con trỏ trỏ đến kiểu cơ sở mảng:
  int* someFunction(); // Hop le

Phép toán con trỏ

- Có thể thực hiện phép toán trên các con trỏ
  - Phép toán địa chỉ
- Ví dụ:

```
typedef double* DoublePtr;
DoublePtr d;
d = new double[10];
```

  - d chứa địa chỉ của d[0]
  - d + 1 là địa chỉ của d[1]
  - d + 2 là địa chỉ của d[2]

Vận hành mảng theo cách khác

- Sử dụng phép toán con trỏ
- Duyệt mảng không cần chỉ số

```
for (int i = 0; i < arraySize; i++)
        cout << *(d + I) << " " ;
```

- Tương đương với:

```
for (int i = 0; i < arraySize; i++)
        cout << d[I] << " " ;
```

- Chỉ được phép cộng/trừ với con trỏ
  - Không được phép nhân, chia
- Có thể sử dụng ++ và -- với con trỏ

Mảng động nhiều chiều

- Mảng nhiều chiều là mảng của các mảng
- Định nghĩa kiểu giúp biểu diễn nó:
  typedef int* IntArrayPtr;
  IntArrayPtr *m = new IntArrayPtr[3];
  - Tạo mảng ba con trỏ
  - Mỗi con trỏ được cấp phát mảng 4 số kiểu int
- for (int i = 0; i < 3; i++)
  m[i] = new int[4];
  - Kết quả thu được là mảng động 3x4

Trở lại với lớp

- Toán tử ->
  - Ký hiệu viết tắt
- Kết hợp toán tử hủy tham chiếu, *, và toán tử dấu chấm
- Chỉ ra thành viên của lớp được trỏ đến bởi con trỏ có sẵn:
- Ví dụ:

```
MyClass *p;
p = new MyClass;
p->grade = "A";   Tương đương với:
(*p).grade = "A";
```

Con trỏ this

- Định nghĩa hàm thành viên có thể cần tham chiếu tới đối tượng gọi
- Sử dụng con trỏ this được định nghĩa trước
  - Tự động trỏ tới đối tượng gọi

```
Class Simple
{
public:
    void showStuff() const;
private:
    int stuff;
};
```

- Hai cách để hàm thành viên truy cập:

```
cout << stuff;
cout << this->stuff;
```

Nạp chồng toán tử gán

- Toán tử gán trả về tham chiếu
  - Do vậy cho phép chuỗi phép gán
  - Ví dụ, a = b = c;
  - Gán a và b bằng c
- Toán tử phải trả về “kiểu tương tự” với kiểu của biến phía tay trái
  - Để cho phép chuỗi làm việc
  - Con trỏ this sẽ giúp thực hiện việc này

Nạp chồng toán tử gán

- Toán tử gán phải là thành viên của lớp
  - Nó có một tham số
  - Toán hạng bên trái là đối tượng gọi
    s1 = s2;
    - Có thể hiểu là: s1.=(s2);
- s1 = s2 = s3;
  - Đòi hỏi (s1 = s2) = s3;
  - Do vậy (s1 = s2) phải trả về đối tượng thuộc kiểu của s1
    - Và truyền nó cho “=s3”;

Định nghĩa toán tử = nạp chồng

- Ví dụ StringClass:

```
StringClass& StringClass::operator=(const StringClass& rtSide)
{
        if (this == &rtSide)            // neu ve trai bang ve phai
                return *this;
        else
        {
                capacity = rtSide.capacity;
                length = rtSide.length;
                delete [] a;
                a = new char[capacity];
                for (int i = 0; i < length; i++)
                        a[i] = rtSide.a[i];
                return *this;
        }
}
```

Sao chép nông và sâu

- Sao chép nông
  - Phép gán chỉ sao chép nội dung của các biến thành viên
  - Phép gán mặc định và hàm tạo sao chép mặc định
- Sao chép sâu
  - Khi liên quan tới con trỏ và cấp phát động
  - Phải hủy tham chiếu biến con trỏ để thực hiện sao chép dữ liệu
  - Hãy tự nạp chồng toán tử gán và hàm tạo sao chép nếu gặp trường hợp này!

Hàm hủy

- Các biến cấp phát động
  - Không biến mất nếu không được delete
- Nếu con trỏ là dữ liệu thành viên private
  - Chúng cấp phát động dữ liệu thực
    - Trong hàm tạo
  - Phải có cách nào đó để giải phóng vùng nhớ khi đối tượng bị hủy
- Câu trả lời: Viết hàm hủy

Hàm hủy

- Ngược lại với hàm tạo
  - Được gọi tự động khi đối tượng ra ngoài phạm vi hoạt động
  - Phiên bản mặc định chỉ xóa các biến thường, không xóa các biến động
- Định nghĩa như hàm tạo, thêm dấu ngã ~

```
MyClass::~MyClass()
{
//Thuc hien cong viec don dep
}
```

Hàm tạo sao chép

- Tự động gọi khi:

  1. Khai báo đối tượng thuộc lớp đồng thời khởi tạo nó bằng đối tượng khác
  2. Khi hàm trả về đối tượng thuộc lớp
  3. Khi đối số có kiểu của lớp được truyền giá trị vào hàm

- Cần bản sao tạm thời của đối tượng
  - Hàm tạo sao chép sinh ra nó
- Hàm tạo sao chép mặc định
  - Giống phép gán mặc định, nó chỉ sao chép trực tiếp các dữ liệu thành viên
- Có dữ liệu con trỏ → hãy tự viết hàm tạo sao chép

Tóm tắt

- Con trỏ là địa chỉ vùng nhớ
  - Cung cấp cách tham chiếu gián tiếp tới biến
- Biến động
  - Được tạo và hủy khi chạy chương trình
- Freestore
  - Vùng nhớ cho biến động
- Mảng cấp phát động
  - Có kích thước được xác định khi chương trình chạy

Tóm tắt

- Hàm hủy
  - Là hàm thành viên đặc biệt của lớp
  - Tự động hủy đối tượng
- Hàm tạo sao chép
  - Là hàm thành viên một đối số
  - Được gọi tự động khi cần bản sao tạm thời
- Toán tử gán
  - Cần được nạp chồng dưới dạng hàm thành viên
  - Trả về tham chiếu để có thể gọi theo chuỗi