

Class & Struct

Lập trình nâng cao

Kiểu dữ liệu có cấu trúc

Class / struct là cấu trúc cho phép định nghĩa các kiểu dữ liệu có cấu trúc: dữ liệu kèm theo các hàm xử lý dữ liệu đó. Ví dụ:

- Vector: Vectơ trong hệ toạ độ Đề-các: cặp toạ độ x và y , cùng các phép toán tổng, hiệu, tích có hướng, tích vô hướng...
- Circle: Hình tròn trong hình học: toạ độ tâm (x,y) và bán kính, các phép toán tính diện tích, tính chu vi, vẽ,...
- Student: Sinh viên trong ứng dụng quản lý đào tạo: tên, mã sinh viên, lớp, địa chỉ, ngày sinh...

Bài toán ví dụ

- Vector trong hệ tọa độ Đề-các: cặp tọa độ x và y , cùng các phép toán tổng, hiệu, tích có hướng, tích vô hướng...
- Viết một chương trình hỗ trợ tính tổng hai vector, in vector ra màn hình dạng (x,y) .
 - `add_vector()`: tính vector tổng của hai vector
 - `print_vector()`: in một vector ra màn hình

Quá nhiều tham số!

Cần 4 tham số cho 2
vector toán hạng

Cách 1

```
void add_vector(double x1, double y1, double x2, double y2,  
                double& x_sum, double& y_sum) {  
    x_sum = x1 + x2;    y_sum = y1 + y2;  
}
```

```
void print_vector(double x, double y) {  
    cout << "(" << x << "," << y << ")";  
}
```

```
int main() {  
    double xA = 1.2, xB = 2.0, yA = 0.4, yB = 1.6;  
    double xSum, ySum;  
    add_vector(xA, yA, xB, yB, xSum, ySum);  
    print_vector(xSum, ySum);  
    return 0;  
}
```

Không thể return 2 biến
đại diện cho vector tổng,
nên phải thêm 2 tham
biến

Đọc code khó mà hiểu đây là các vector

Cách tốt hơn

```
struct Vector {  
    double x;  
    double y;  
    ...  
};
```

```
Vector add(Vector v1, Vector v2) {  
    Vector sum;  
    sum.x = v1.x + v2.x;  
    sum.y = v1.y + v2.y;  
    return sum;  
}  
  
void print(Vector v) {  
    cout << "(" << v.x << ", " << v.y << ")";  
}
```

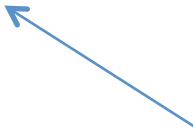
Ít tham số, dễ đọc. Do dữ liệu vector được đóng gói trong một cấu trúc

```
int main() {  
    Vector a(1.2, 0.4), b(2.0, 1.6);  
    Vector sum = add(a, b);  
    print(sum);  
    return 0;  
}
```

Code gọn,
dễ hiểu đây là các vector

Cách tốt hơn nữa

```
struct Vector {  
    double x;  
    double y;  
  
    Vector add(Vector other) {... }  
    void print() {...}  
};
```



Các hàm xử lý dữ liệu
cũng được đóng gói
kèm với dữ liệu

```
int main() {  
    Vector a(1.2, 0.4), b(2.0, 1.6);  
    Vector sum = a.add(b);  
    sum.print();  
    return 0;  
}
```

STRUCT VÀ CÁC BIẾN THÀNH VIÊN

Định nghĩa kiểu dữ liệu mới

```
struct Vector {  
    double x;  
    double y;  
};
```

định nghĩa kiểu Vector gồm:

- trường dữ liệu x
- trường dữ liệu y

Ý nghĩa:



```
Vector v1, v2;
```

biến v1, v2 thuộc kiểu Vector

Mỗi biến thuộc kiểu Vector có hai thành viên dữ liệu là x kiểu double và y kiểu double.

Sử dụng

```
struct Vector {  
    double x;  
    double y;  
};
```

định nghĩa kiểu Vector gồm:

- trường dữ liệu x
- trường dữ liệu y

```
Vector v;  
v.x = 1.0;  
v.y = 2.1;  
cout << v.x;
```

khai báo biến v kiểu Vector
gán giá trị cho trường x của biến v
gán giá trị cho trường y của biến v
lấy giá trị của x của y.

Ví dụ

```
struct Person {  
    string name;  
    string address;  
    int age;  
};
```

định nghĩa kiểu dữ liệu Person gồm:

- trường dữ liệu name
- trường dữ liệu address
- trường dữ liệu age

sử dụng

```
Person john;  
john.name = "John";  
john.address = "London";  
john.age = 20;
```

khai báo biến john kiểu Person

Point – tọa độ trong không gian 2D

```
struct Point {  
    double x;  
    double y;  
  
    Point(int _x, int _y) {...}  
};
```

Triangle – tam giác

```
struct Triangle {  
    Point a;  
    Point b;  
    Point c;  
    Triangle(int x1, int y1, int x2, int y2,  
             int x3, int y3)  
        :a(x1, y1), b(x2,y2), c(x3, y3)  
    {}  
};
```

Gọi constructor
Point(int x, int y) để
khởi tạo a, b, c

hoặc

```
struct Triangle {  
    Point a[3];  
};
```

Gọi constructor
Point() để khởi tạo a,
b, c

Khởi tạo các biến thành viên

```
struct Triangle {  
    Point a;  
    Point b;  
    Point c;  
    Triangle(int x1, int y1, int x2, int y2,  
              int x3, int y3)  
        :a(x1, y1), b(x2, y2), c(x3, y3)  
    {}  
  
    Triangle()  
    {}  
};
```

Gọi constructor
Point(int x, int y) để
khởi tạo a, b, c

Gọi ngầm constructor mặc định
Point() để khởi tạo a, b, c

Khởi tạo biến thành viên

```
struct Triangle {  
    Point a;  
    Point b;  
    Point c;  
};
```

Không khai báo constructor
Sẽ có constructor mặc định
Triangle()

– không tham số, nội dung rỗng
– Với nhiệm vụ ngầm gọi

constructor mặc định Point() để
khởi tạo a,b,c

Hoặc a[0], a[1], a[2].

```
struct Triangle {  
    Point a[3];  
};
```

struct làm tham số cho hàm

- Truyền bằng giá trị - pass by value

```
void print(Vector v) {  
    cout << "(" << v.x << "," << v.y << " )";  
}
```

- Truyền bằng tham chiếu - pass by reference

```
void print(Vector& v) {  
    cout << "(" << v.x << "," << v.y << " )";  
}
```

struct làm tham số cho hàm

- Truyền bằng con trỏ - pass by pointer

```
void print(Vector* pv) {  
    cout << "(" << pv->x << "," << pv->y << ")";  
}
```

Chú ý, khi truy nhập các trường từ con trỏ, phải dùng **toán tử mũi tên (->)** thay vì dấu chấm (.)

Struct và con trỏ

```
struct Vector {  
    double x;  
    double y;  
};
```

```
Vector v;  
v.x = 1.0;  
v.y = 2.1;  
cout << v.x;
```

```
Vector* pV = &v;
```

con trỏ tới v

```
pV->x = 1.0;
```

truy nhập v.x từ con trỏ

```
(*pV).y = 2.1;
```

truy nhập v.y từ biến v

```
cout << pV->x << ", " << (*pV).y;
```

Cú pháp truy nhập các trường

```
struct Vector {  
    double x;  
    double y;  
};  
  
Vector v;  
Vector* pV = &v;
```

Dùng **dấu chấm (.)** để truy nhập từ biến / ô nhớ struct:

v.x

(*pv).x

Dùng **mũi tên (->)** để truy nhập bằng con trỏ/địa chỉ:

pv->x

(&v) ->x

Cấp phát bộ nhớ động

- Giống hệt đối với các kiểu dữ liệu khác

```
int* p = new int;  
// sử dụng p ...  
delete p;
```

```
Vector* p = new Vector;  
// sử dụng p ...  
delete p;
```

```
int* arr = new int[10];  
// sử dụng arr ...  
delete [] arr;
```

```
Vector* arr = new Vector[10];  
// sử dụng arr ...  
delete [] arr;
```

Struct và phép gán

- Giống như các kiểu dữ liệu thông thường, phép gán được thực hiện khi:

- Phép gán

- ```
Vector v1 = v2;
```

- Truyền tham trị vào hàm

- ```
void print(Vector v) {...}
```

- ```
print(v1);
```

 //v1 được gán cho tham số v

- Trả về giá trị

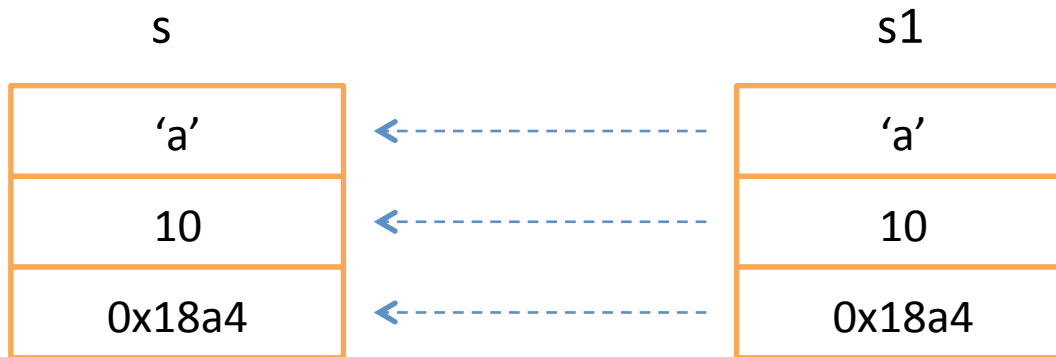
- ```
return v1;
```

 //v1 được gán cho biến nhận giá trị trả về

- Phép gán làm gì? copy từng trường vào biến đích – copy nông, chỉ sao chép giá trị.

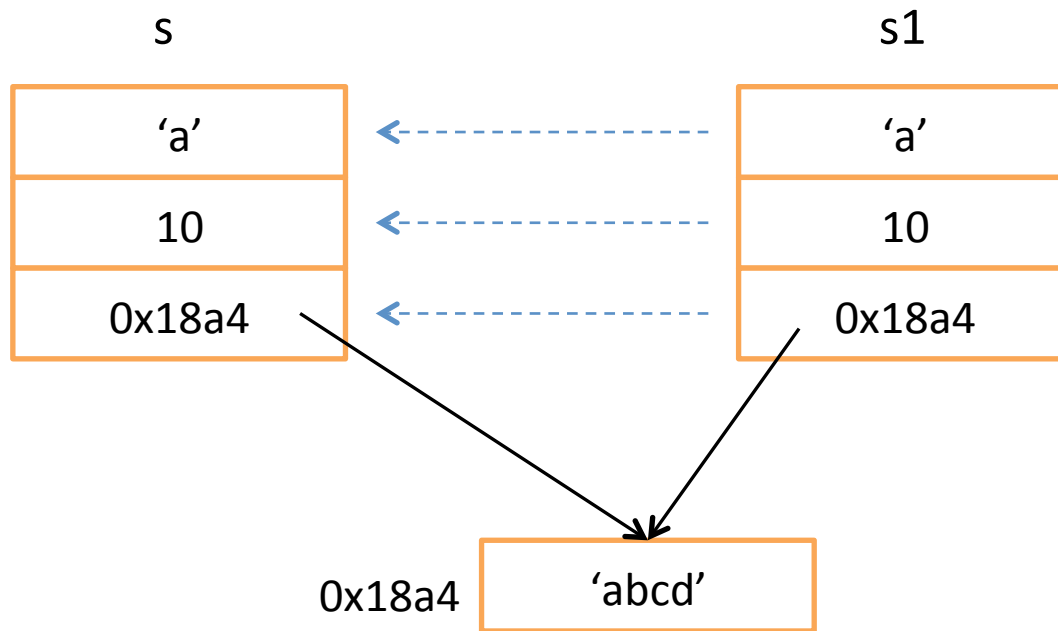
Copy nông – swallow copy

`s = s1;`



Copy nông – swallow copy

`s = s1;`



Đối với con trỏ, copy nông có nghĩa chỉ sao chép con trỏ, không sao chép nội dung nó trỏ tới

Best practice

- Khi muốn truyền struct vào hàm ở dạng chỉ đọc, nên truyền hằng tham chiếu thay vì tham trị
 - Không tốn công copy
 - Vẫn đảm bảo hàm không sửa đổi số.

```
void print(const Vector& v) { ... }
```

có hiệu ứng tương đương nhưng tốt hơn

```
void print(Vector v) { ... }
```

HÀM THÀNH VIÊN


```
struct Vector {  
    double x;  
    double y;  
};
```

Hàm xử lý nằm ngoài

```
void print(const Vector& v) {  
    cout << "(" << v.x << "," << v.y << ")";  
}
```

```
Vector add(const Vector& v1, const Vector& v2) {  
    Vector sum;  
    sum.x = v1.x + v2.x;  
    sum.y = v1.y + v2.y;  
    return sum;  
}
```

```
struct Vector {  
    double x;  
    double y;
```

Hàm thành viên

```
void print() {  
    cout << "(" << x << "," << y << " )";  
}
```

```
Vector add(const Vector& other) {  
    Vector sum;  
    sum.x = x + other.x;  
    sum.y = y + other.y;  
    return sum;  
}  
};
```

print() và add() là các hàm thành viên của struct nên có thể truy cập trực tiếp đến các biến thành viên x và y.

```
struct Vector {  
    double x;  
    double y;
```

Hàm thành viên

```
    void print() {  
        cout << "(" << x << "," << y << ")";  
    }  
    ...  
};
```

```
Vector v;  
v.print();
```

Hàm thông thường

```
void print(Vector& v) {  
    cout << "(" << v.x << "," << v.y << ")";  
}
```

```
Vector v;  
print(v);
```

```
struct Vector {
```

```
...
```

```
Vector add(Vector& other) {
```

```
    Vector sum;
```

```
    sum.x = x + other.x;
```

```
    sum.y = y + other.y;
```

```
    return sum;
```

```
}
```

```
};
```

Hàm thành viên

```
Vector v1, v2;
```

```
Vector s = v1.add(v2);
```

Hàm thông thường

```
Vector add(Vector& v1, Vector& v2) {
```

```
    Vector sum;
```

```
    sum.x = v1.x + v2.x;
```

```
    sum.y = v1.y + v2.y;
```

```
    return sum;
```

```
}
```

```
Vector v1, v2;
```

```
Vector s = add(v1, v2);
```

Cú pháp gọi hàm thành viên

```
struct Vector {  
    double x;  
    double y;  
    void print() {...}  
};  
  
Vector v;  
Vector* pV = &v;
```

Dùng **dấu chấm (.)** để truy nhập từ biến / ô nhớ struct:

v.print() **(*pv).print()**

Dùng **mũi tên (->)** để truy nhập bằng con trỏ/địa chỉ:

pv->print() **(&v)->print()**

(giống hệ truy nhập biến thành viên)

Hằng hàm thành viên

```
struct Vector {  
    ...  
    void print() {... }  
    Vector add(const Vector& other) { ... }  
};
```

Vấn đề: với `v` là `const Vector` (khai báo **`const Vector v;`**)
thì các lệnh sau bị lỗi biên dịch:

```
v.print();  
v.add(another_vector);
```

Lí do: `print()` và `add()` không đảm bảo với trình biên dịch rằng
chúng sẽ không sửa giá trị của biến struct mà nó là thành viên

Cần khai báo `print()` và `add()` là các **hằng hàm thành viên**

Hằng hàm thành viên

```
struct Vector {  
    double x;  
    double y;
```

print() đảm bảo không sửa giá trị của biến struct mà nó là thành viên

```
void print() const {  
    cout << "(" << x << ", " << y << ")";  
}
```

```
Vector add(const Vector& other) const {  
    Vector sum;  
    sum.x = x + other.x;  
    sum.y = y + other.y;  
    return sum;  
}
```

add() đảm bảo không sửa giá trị của biến struct mà nó là thành viên

```
};
```

Hằng biến chỉ có thể được dùng tại các vị trí const

```
struct Vector {  
    void print() const {...}  
    Vector add(Vector& other) const { ...}  
    void append(const Vector& tail) {...}  
};
```

```
const Vector v; Vector v2;  
v.print();           // ok vì print là hằng hàm thành  
viên  
v2.print();          //v2 không phải const nên không quan tâm  
v.add(v2);           // ok vì add là hằng hàm thành viên  
v2.add(v);           // lỗi vì tham số other không phải  
const  
v.append(v2);        // lỗi vì append không phải hằng hàm
```


CONSTRUCTOR VÀ DESTRUCTOR

Khởi tạo các biến thành viên

```
Vector v1;  
v1.x = 1.0;  
v1.y = 2.1;  
Vector v2;  
v2.x = 1.3;  
v2.y = 2.2;
```

```
Student s;  
s.first_name = "John";  
s.last_name = "Smith";  
s.major = "cs";  
s.id = "15123456";
```

- Tốn nhiều dòng khởi tạo giá trị cho các biến thành viên? Thế này có hay hơn không?

```
Vector v1(1.0, 2.1);  
Vector v2(1.3, 2.2);  
Student s("1512345", "John", "Smith", "cs");
```

Constructor

- Là hàm thành viên đặc biệt có nhiệm vụ khởi tạo các biến thành viên.
 - Được gọi tự động khi khai báo hoặc cấp phát biến động
 - Trùng tên với tên struct
 - Không có kiểu trả về

```
struct Vector {  
    double x;  
    double y;  
    Vector(double _x, double _y) {  
        x = _x; y = _y;  
    }  
};
```

```
Vector v(1.0, 2.1);  
Vector* pv = new Vector(1.3, 2.2);
```

Ví dụ


```
struct Array {  
    int n;  
    int* arr;  
    Array(int _n) {  
        n = _n;  
        arr = new int[n];  
    }  
};
```

Sử dụng:

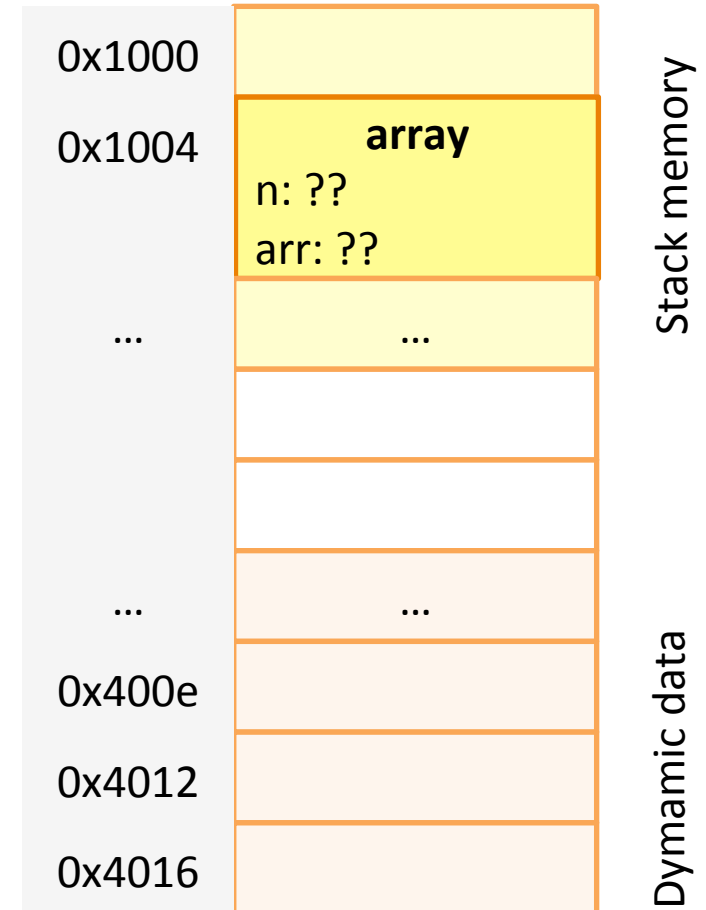
```
Array a(20) ;
```

```
Array* p = new Array(10) ;
```

Ví dụ

```
struct Array {  
    int n;  
    int* arr;  
     Array(int _n) {  
        n = _n;  
        arr = new int[n];  
    }  
};
```

```
Array array(2);
```

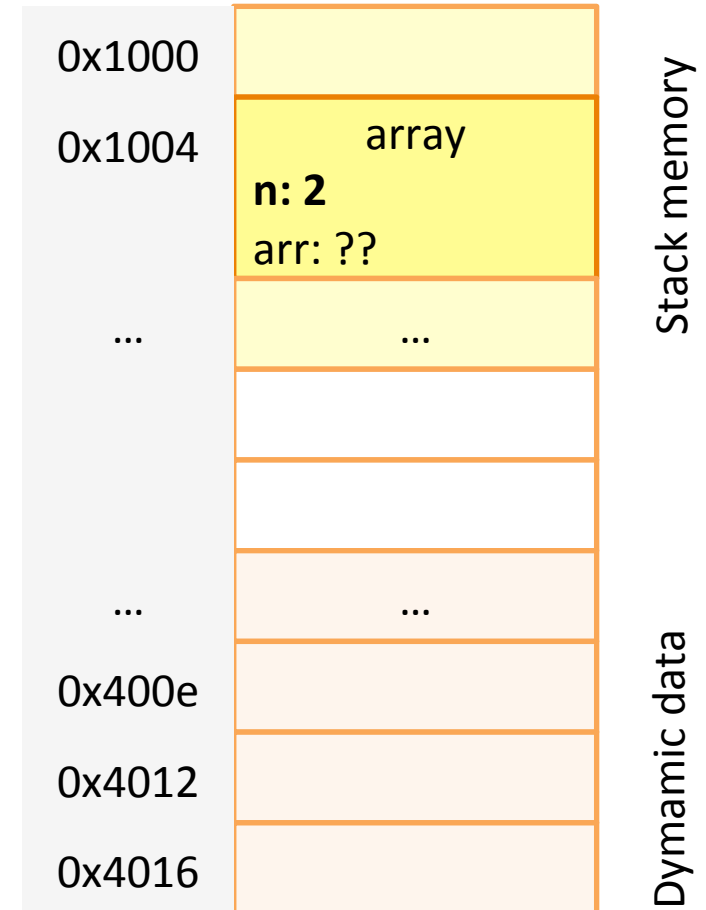


Ví dụ

```
struct Array {  
    int n;  
    int* arr;  
    Array(int _n) {  
        n = _n;  
        arr = new int[n];  
    }  
};
```

→

```
Array array(2);
```

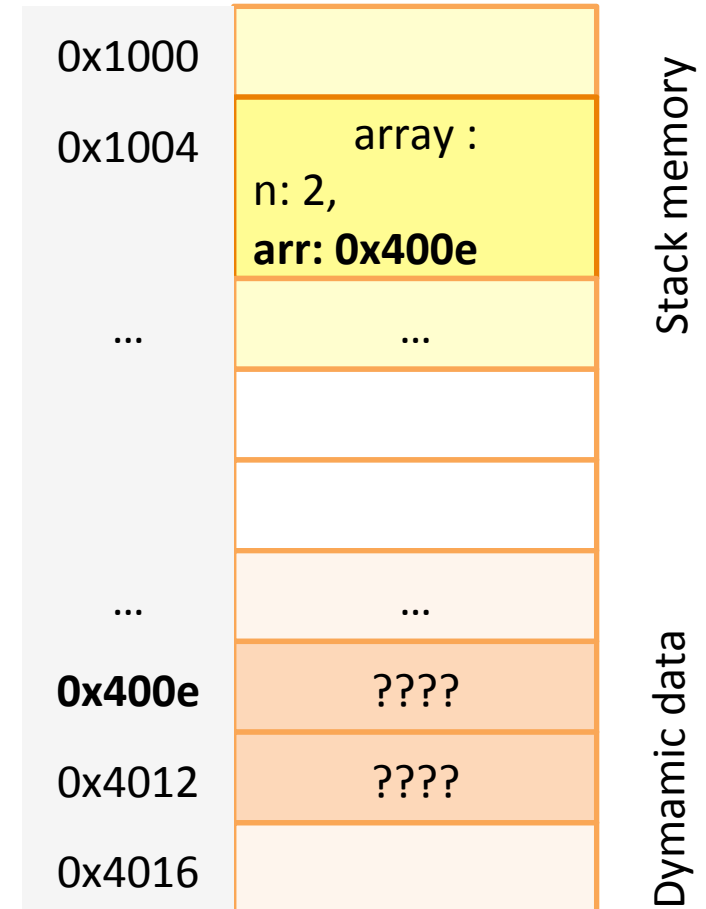


Ví dụ

```
struct Array {  
    int n;  
    int* arr;  
    Array(int _n) {  
        n = _n;  
        arr = new int[n];  
    }  
};
```

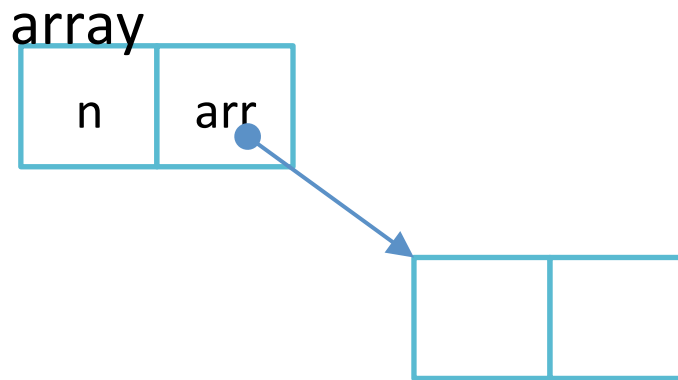


```
Array array(2);
```

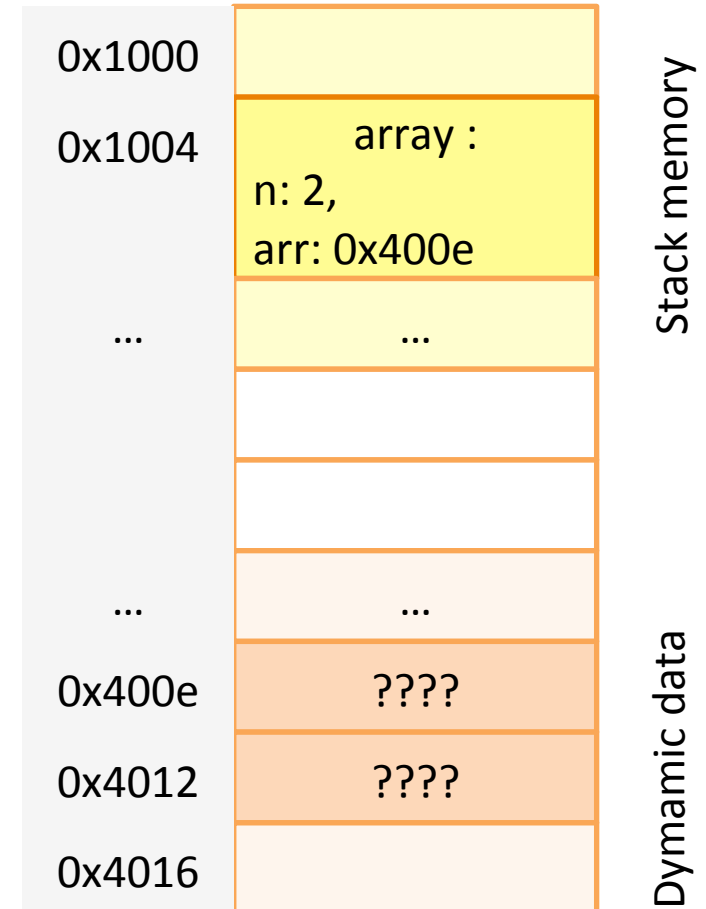


Ví dụ

```
struct Array {  
    int n;  
    int* arr;  
    Array(int _n) {  
        n = _n;  
        arr = new int[n];  
    }  
};
```



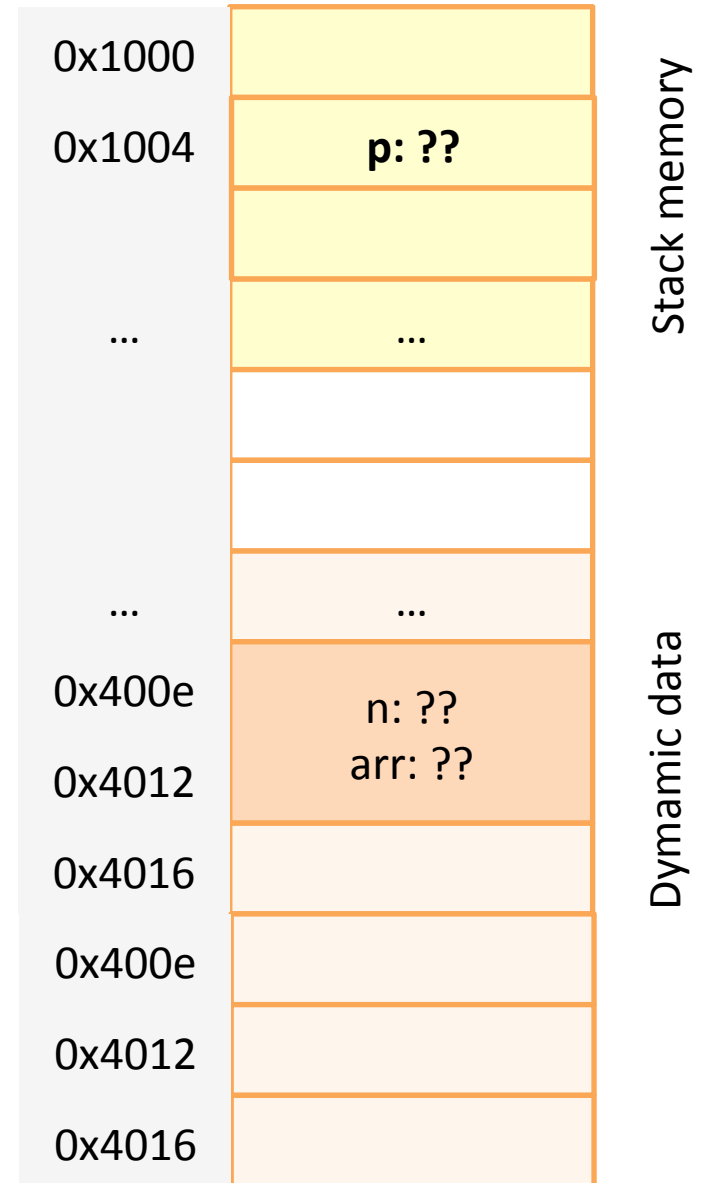
Array array(2);



Ví dụ

```
struct Array {  
    int n;  
    int* arr;  
    Array(int _n) {  
        n = _n;  
        arr = new int[n];  
    }  
};
```

```
Array* p  
= new Array(2);
```

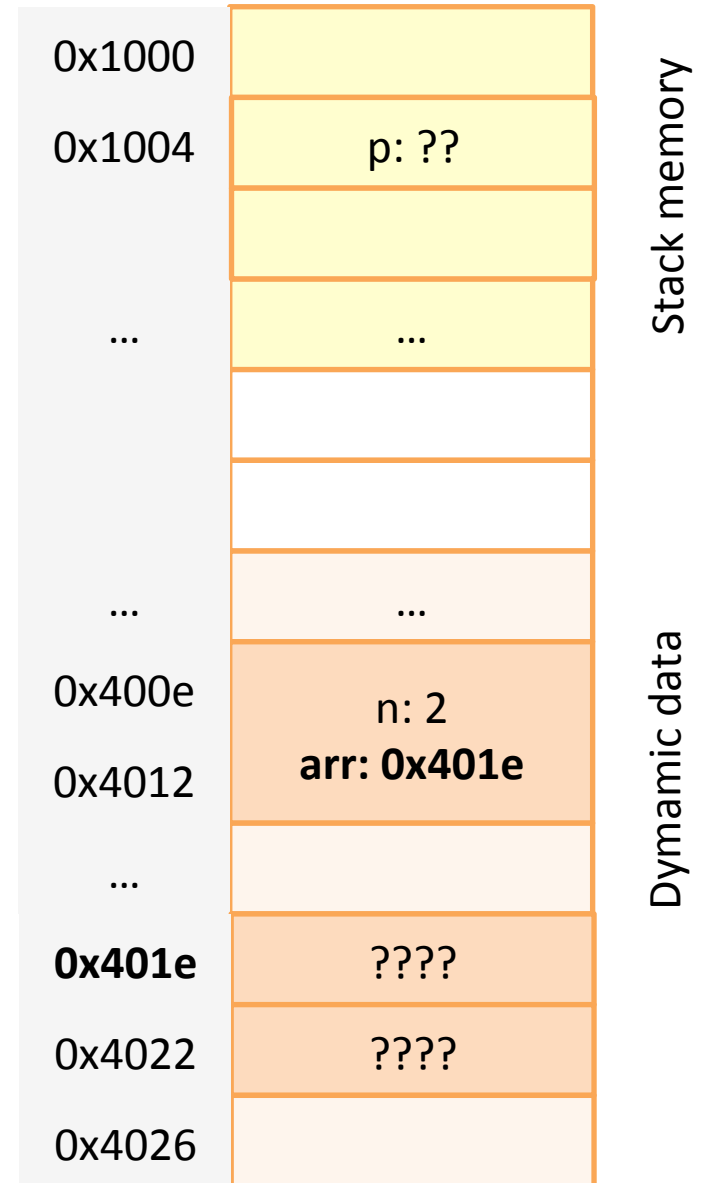


Ví dụ

```
struct Array {  
    int n;  
    int* arr;  
    Array(int _n) {  
        n = _n;  
        arr = new int[n];  
    }  
};
```



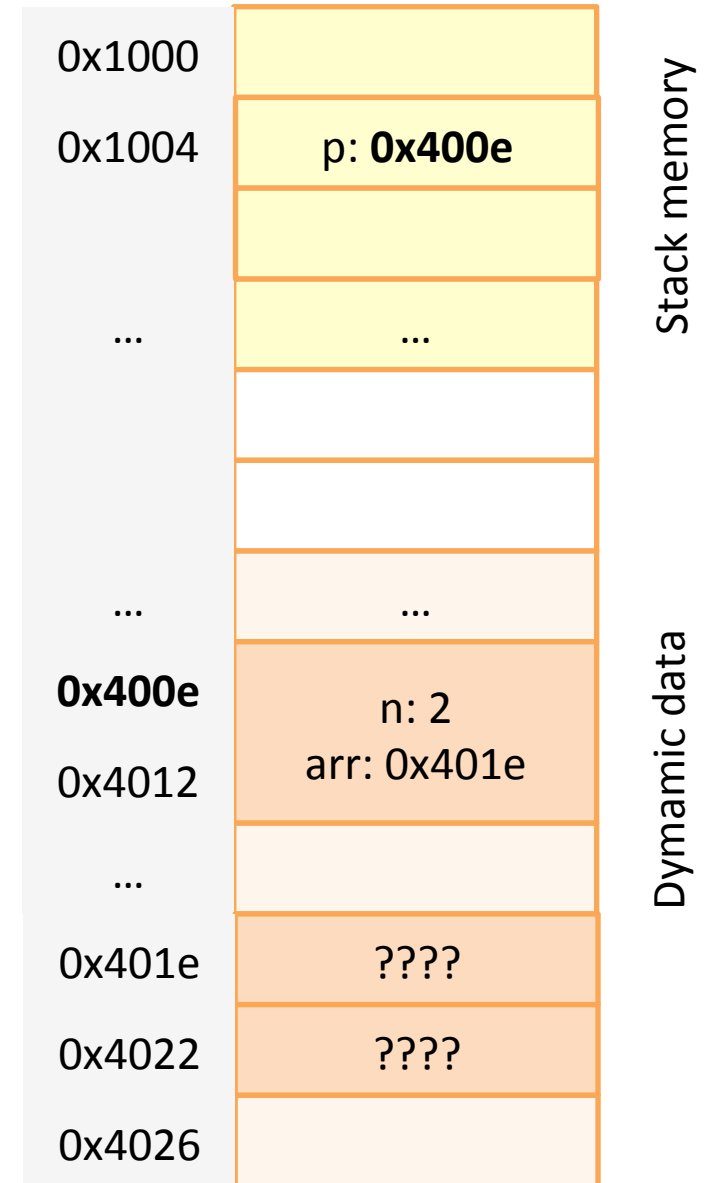
```
Array* p  
= new Array(2);
```



Ví dụ

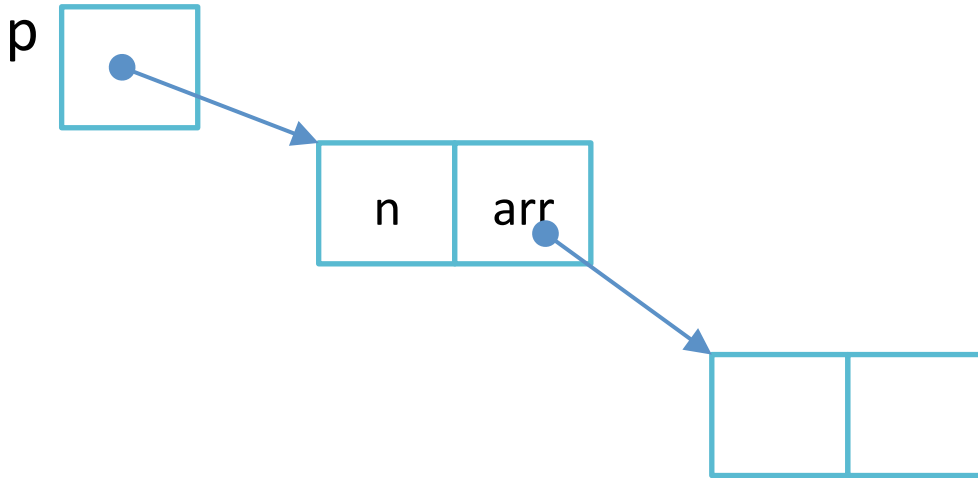
```
struct Array {  
    int n;  
    int* arr;  
    Array(int _n) {  
        n = _n;  
        arr = new int[n];  
    }  
};
```

```
Array* p  
= new Array(2);
```



Ví dụ

```
struct Array {  
    int n;  
    int* arr;  
    Array(int _n) {  
        n = _n;  
        arr = new int[n];  
    }  
};
```



```
Array* p  
= new Array(2);
```

0x1000		Stack memory
0x1004	p: 0x400e	
...	...	
		Dynamic data
...	...	
0x400e	n: 2	
0x4012	arr: 0x401e	
...		
0x401e	????	
0x4022	????	
0x4026		

Nhiều constructor

```
struct Array {  
    int n;  
    int* arr;
```

```
Array(int _n) {  
    n = _n;  
    arr = new int[n];  
}
```

```
Array(int _n, int default_value) {  
    n = _n;  
    arr = new int[n];  
    for (int i = ...) arr[i] = default_value;  
}
```

```
...  
};
```

```
Array a1 (10, 5) ; // gọi Array(int _n, int default_value)  
Array a2 (10) ;    // gọi Array(int _n)  
Array a3;          // gọi hàm Array()
```

Nhiều constructor

```
struct Array {  
    int n;  
    int* arr;
```

```
Array(int _n) {  
    n = _n;  
    arr = new int[n];  
}
```

```
Array(int _n, int default_value) {  
    n = _n;  
    arr = new int[n];  
    for (int i = ...) arr[i] = default_value;  
}
```

```
...  
};
```

```
Array a3;
```

```
// gọi hàm Array() -> lỗi nếu không có
```

Nhiều constructor

```
struct Array {  
    int n;  
    int* arr;
```

```
Array() { ... }
```



```
Array a3;
```

```
Array(int _n) {  
    n = _n;  
    arr = new int[n];  
}
```

```
Array(int _n, int default_value) {  
    n = _n;  
    arr = new int[n];  
    for (int i = ...) arr[i] = default_value;  
}
```

```
...  
};
```

Nhiều constructor

```
struct Array {  
    int n;  
    int* arr;
```

```
Array(int _n = 5) {  
    n = _n;  
    arr = new int[n];  
}
```

← `Array a3;`

```
Array(int _n, int default_value) {  
    n = _n;  
    arr = new int[n];  
    for (int i = ...) arr[i] = default_value;  
}
```

```
...  
};
```


Hủy biến struct cấp phát động

```
struct Array {  
    int n;  
    int* arr;  
    Array(int _n) {  
        n = _n;  
        arr = new int[n];  
    }  
};
```

```
Array* p = new Array(2) ;
```

```
... .
```

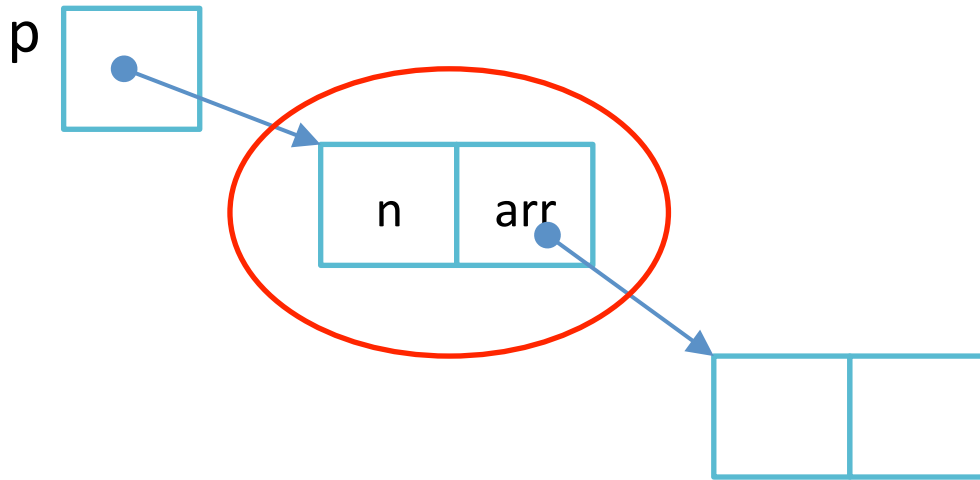
```
delete p;    // hủy struct mà p trỏ tới
```

```

struct Array {
    int n;
    int* arr;
    Array(int _n) {
        n = _n;
        arr = new int[n];
    }
};

```

Mặc định

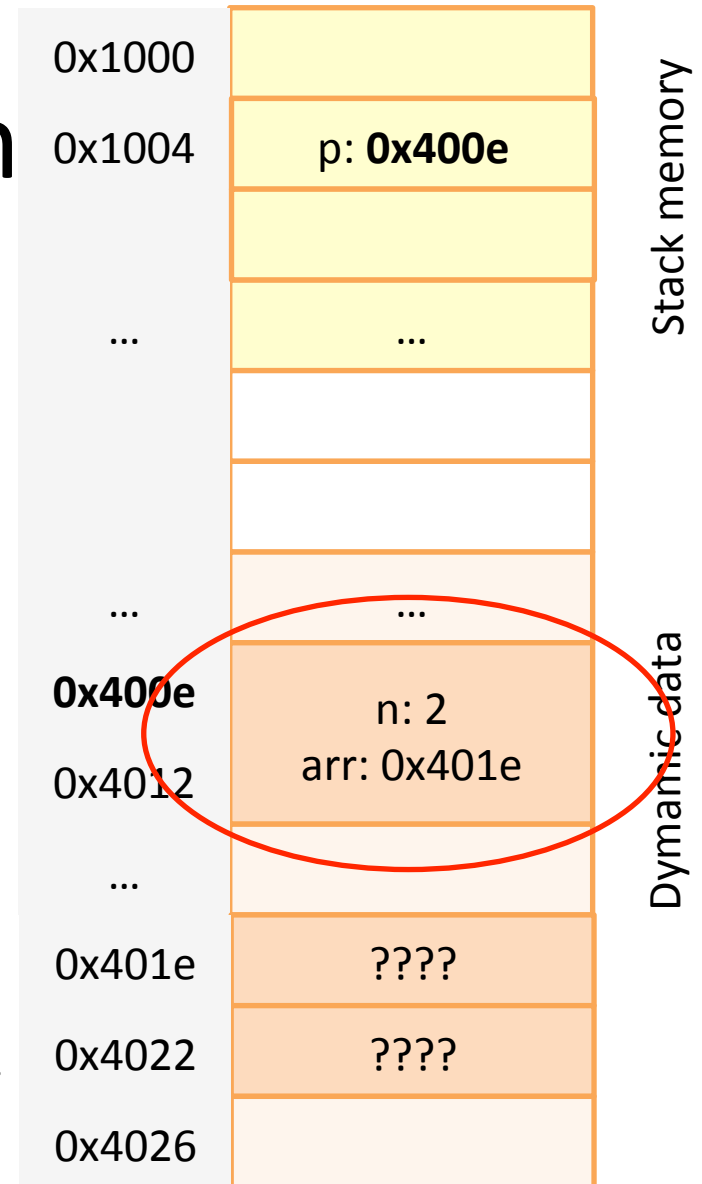


```

Array* p = new Array(2);
...

```

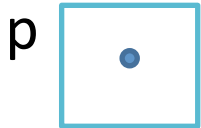
delete p; // hủy struct mà p trở tới



```

struct Array {
    int n;
    int* arr;
    Array(int _n) {
        n = _n;
        arr = new int[n];
    }
};

```



Mặc định

Dữ liệu mờ côi –
thất thoát bộ nhớ



```

Array* p = new Array(2);

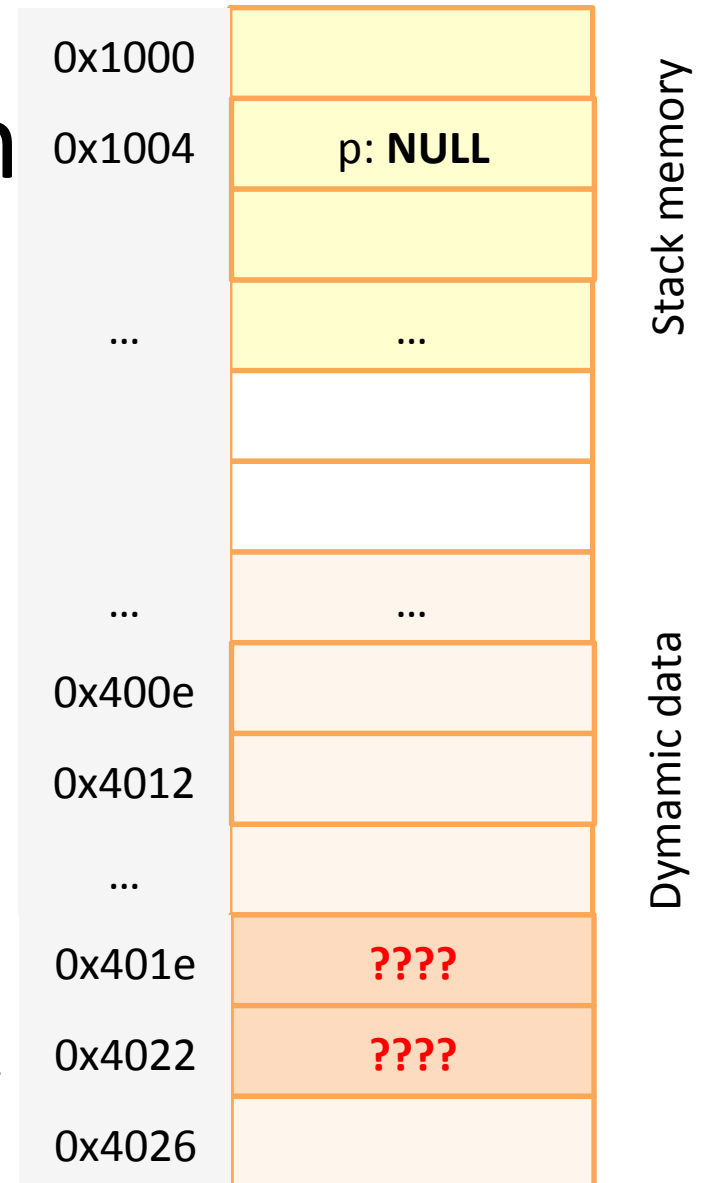
```

...

```

delete p; // hủy struct mà p trở tới

```



Destructor

- Hàm thành viên đặc biệt cho phép lập trình viên tự dọn dẹp các biến thành viên được cấp phát động
 - Được gọi tự động bởi các lệnh delete
 - Không có kiểu trả về, không tham số
 - Tên trùng với tên struct và thêm dấu ngã (~) ở đầu

```
Array* p = new Array(2);
```

```
... .
```

```
delete p; // lệnh này gọi hàm destructor ~Array().  
           //Ta sẽ dùng ~Array() để dọn dẹp mảng động arr
```

Destructor

```
struct Array {  
    int n;  
    int* arr;  
  
    Array(int _n) {  
        n = _n;  
        arr = new int[n];  
    }  
  
    ~Array() {  
        delete [] arr;  
    }  
  
    ...  
}
```

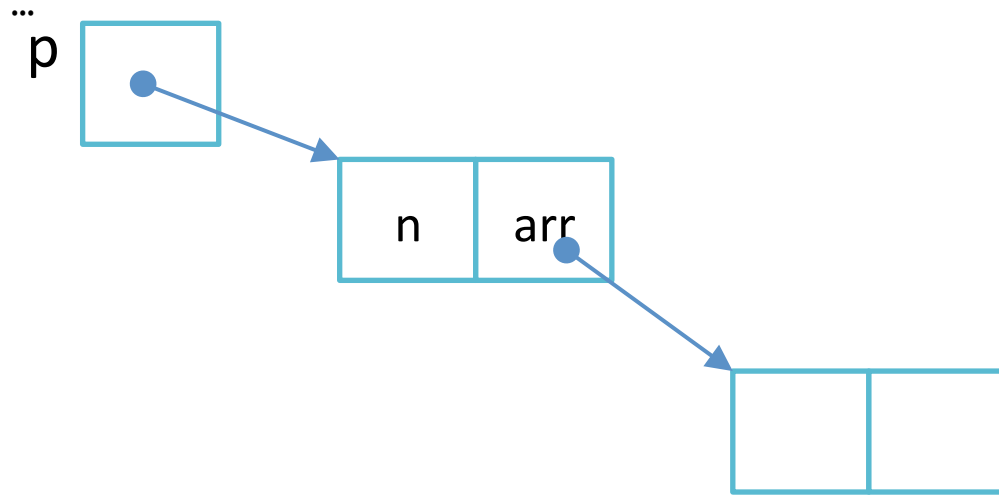
Nhiệm vụ của destructor:
giải phóng tất cả các biến
thành viên đã được cấp
phát động

```

struct Array {
    Array(int _n) {
        n = _n;
        arr = new int[n];
    }
    ~Array() {
        delete [] arr;
    }
}

```

Dùng destructor



```
Array* p = new Array(2);
```

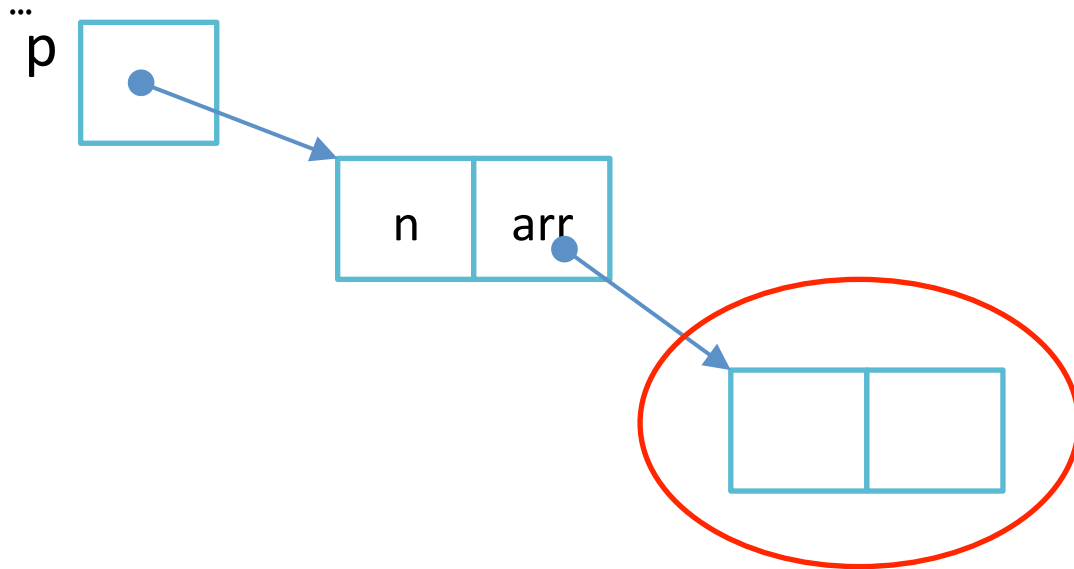
...

➔ **delete p;** // hủy struct mà p trở tới

0x1000		Stack memory
0x1004	p: 0x400e	
...	...	
		Dynamic data
...	...	
0x400e	n: 2 arr: 0x401e	
0x4012		
...		
0x401e	????	
0x4022	????	
0x4026		

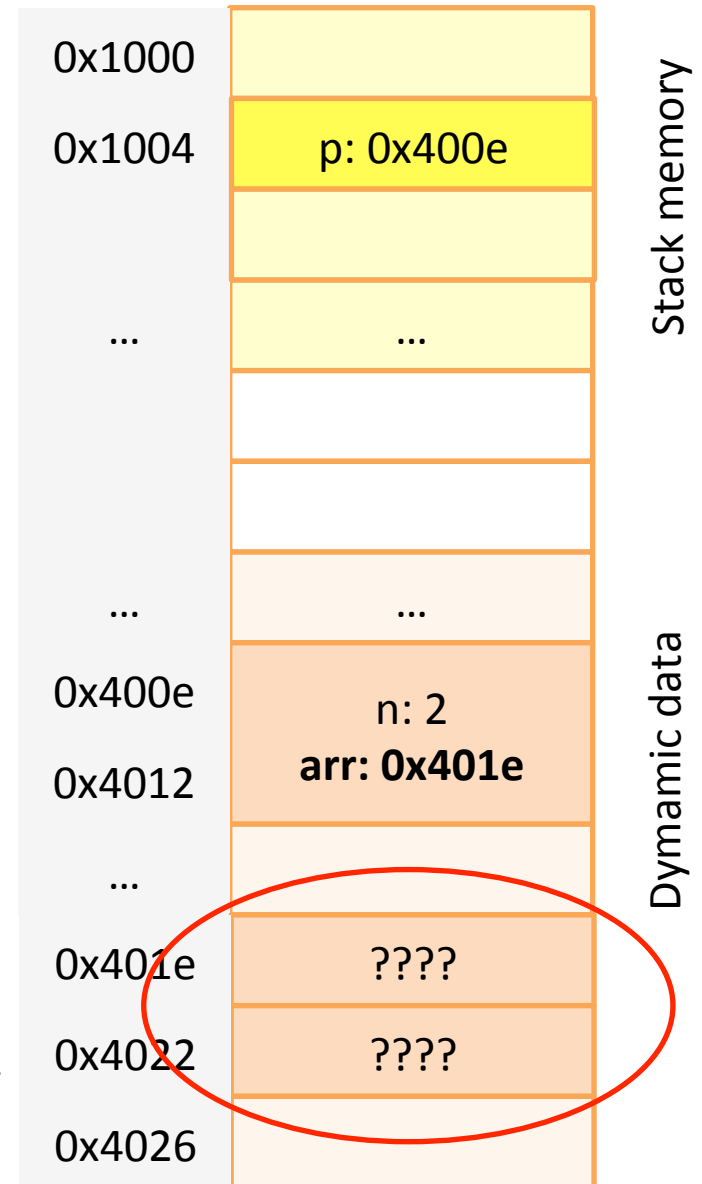
Dùng destructor

```
struct Array {  
    Array(int _n) {  
        n = _n;  
        arr = new int[n];  
    }  
    ~Array() {  
        delete [] arr;  
    }  
};
```



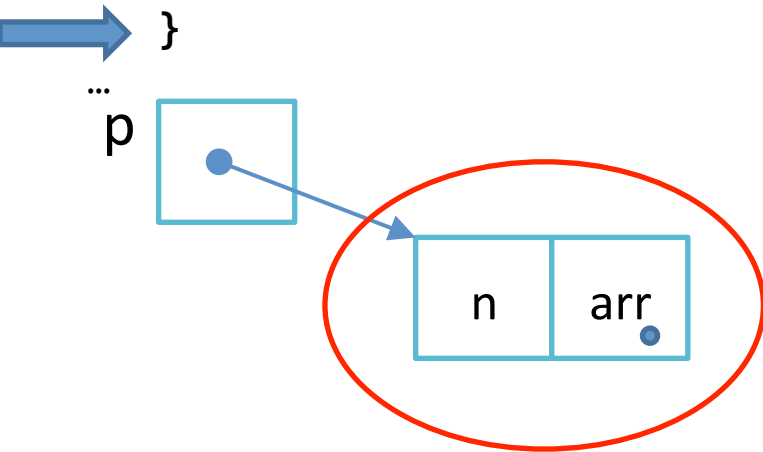
```
Array* p = new Array(2);  
... .
```

delete p; // hủy struct mà p trở tới



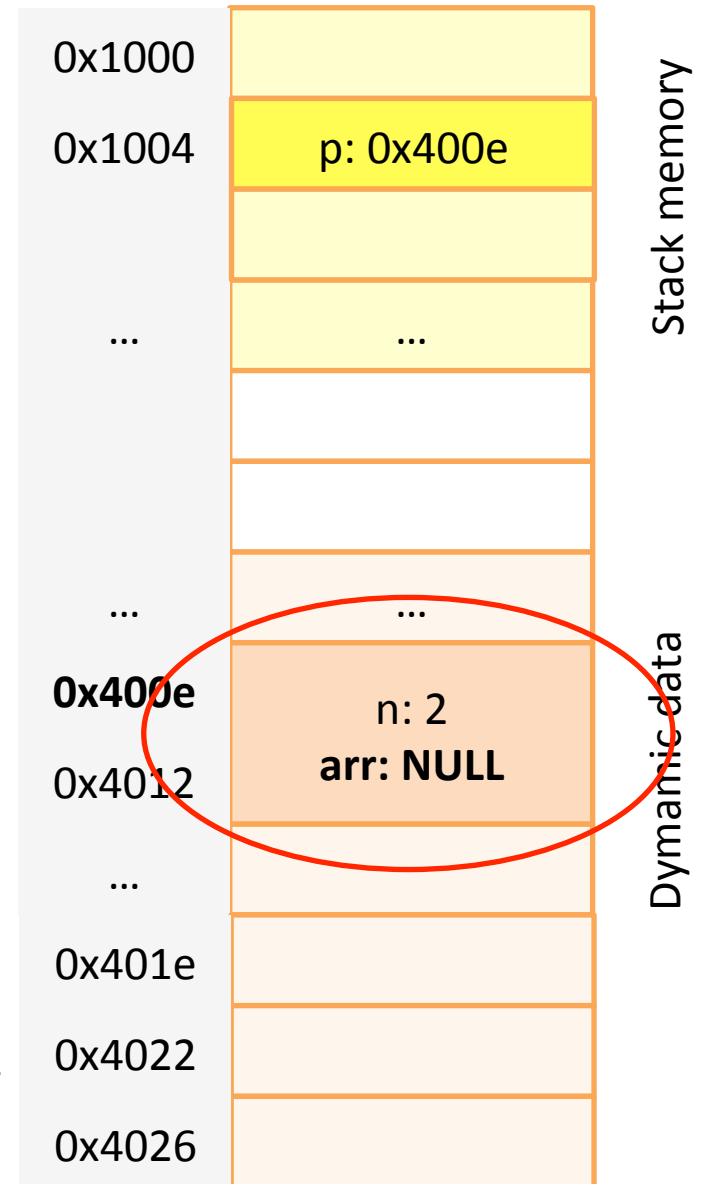
Dùng destructor

```
struct Array {  
    Array(int _n) {  
        n = _n;  
        arr = new int[n];  
    }  
    ~Array() {  
        delete [] arr;  
    }  
};
```



```
Array* p = new Array(2);  
... .
```

delete p; // hủy struct mà p trở tới

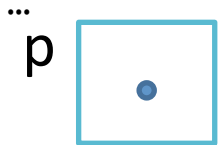



```

struct Array {
    Array(int _n) {
        n = _n;
        arr = new int[n];
    }
    ~Array() {
        delete [] arr;
    }
}

```

Hủy

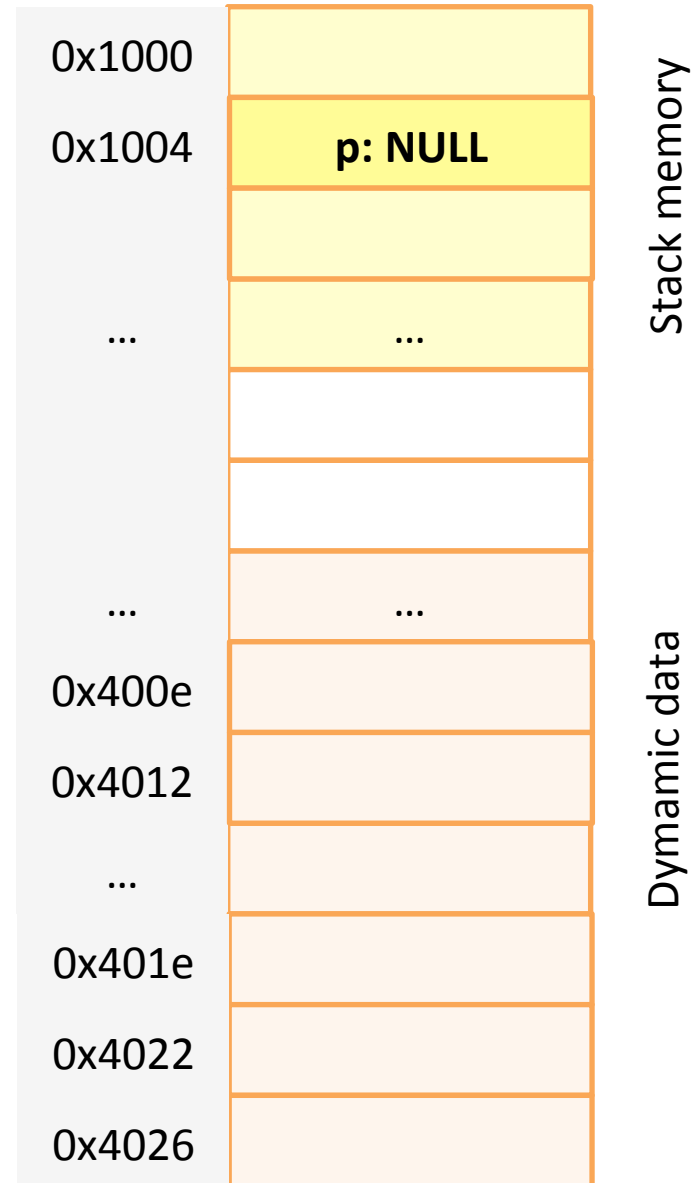


Đã dọn dẹp sạch sẽ

```
Array* p = new Array(2);
```

...

➔ `delete p; // hủy struct mà p trở tới`



Struct để mô hình String

- Array of char
- Length
- Khởi tạo: String s (“Hello”);
- Muốn s.length == 5
- Muốn s.print() -> in “Hello”

```
String s(“Hi”);  
Cout << s.length;  
s.print();
```

```
String* p = new String(“abc”);  
p->print();  
p->length;  
delete p;
```

```
struct String {  
    char* arr;  
    int length;
```

```
String(const char* _s) {  
    length = strlen(_s);  
    arr = new char[length];  
    strncpy(arr, s, length);  
}  
~String() {  
    delete [] arr;  
}  
void print() {  
    cout << arr;  
}
```

...