

LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG (C++)

Giảng viên:

Đặng Hoài Phương

Bộ môn:

Công nghệ phần mềm

Khoa:

Công nghệ Thông tin

Trường Đại học Bách Khoa

Đại học Đà Nẵng





GIỚI THIỆU

- Đặng Hoài Phương.
- E-mail: danghoaiphuongdn@gmail.com
- Tel: 0935578555
- Khoa Công nghệ thông tin, trường Đại học Bách khoa, Đại học Đà Nẵng.



CHƯƠNG 1

EXTERN C++



PROGRAMMING LANGUAGE

- Programming Language: là ngôn ngữ dùng để diễn tả thuật toán sao cho máy tính hiểu và thực hiện được.
- 2 loại:
 - Ngôn ngữ lập trình bậc thấp (low-level programming language):
 - Mã máy;
 - Hợp ngữ (Assembly).
 - Ngôn ngữ bậc cao (high-level programming language):
 - FORTRAN (1956);
 - PASCAL (1970), C (1972), C++ (1980), Java (1995), C# (2001), Python (1990), Swift (2014), Kotlin (2017), ...
 - **FRAMEWORK & SOFTWARE LIBRARIES**

- Là phương pháp xuất hiện đầu tiên.
- Giải quyết các bài toán nhỏ, tương đối đơn giản (lĩnh vực tính toán).

```
10      k=1
20      gosub 100
30      if y > 120 goto 60
40      k = k + 1
50      goto 20
60      print k, y
70      stop
100     y = 3*k*k + 7*k - 3
110     return
```

Lệnh nhảy đến vị trí bất kỳ trong chương trình

- Đặc trưng:
 - Đơn giản;
 - Đơn luồng;
 - Sử dụng biến toàn cục, lạm dụng lệnh GOTO;
 - Chỉ gồm một chương trình chính.



PROCEDURE-ORIENTED PROGRAMMING (POP)

- Tổ chức chương trình thành các **chương trình con**
 - PASCAL: thủ tục & hàm, C: hàm.
 - Chương trình hướng cấu trúc = cấu trúc dữ liệu + tập hợp hàm.
 - Trừu tượng hóa chức năng (Functional Astraction):
 - Không quan tâm đến cấu trúc hàm;
 - Chỉ cần biết kết quả thực hiện của hàm.
- Nền tảng của lập trình hướng cấu trúc.



PROCEDURE-ORIENTED PROGRAMMING (POP)

- Ví dụ:

```
int func(int j)
{
    return (3*j*j + 7*j-3);
}
int main()
{
    int k = 1
    while (func(k) < 120)
        k++;
    printf("%d\t%d\n", k, func(k));
    return(0);
}
```


NGÔN NGỮ C VÀ C++



- Ngôn ngữ C ra đời năm 1972.
- Phát triển thành C++ vào năm 1983.

- Một số từ khóa (keyword) mới đã được đưa vào C++ ngoài các từ khóa có trong C.
- Các chương trình sử dụng các tên trùng với các từ khóa cần phải thay đổi trước khi chương trình được dịch lại bằng C++.

asm	catch	class	delete	friend	inline
new	operator	private	protected	public	
template	this	throw	try	virtual	

- C:
 - Chú thích bằng `/* ... */` → dùng cho khối chú thích lớn gồm nhiều dòng.
- C++: giống như C
 - Đưa thêm chú thích bắt đầu bằng `//` → dùng cho các chú thích trên một dòng.
- Ví dụ:
 - `/* Đây là`
`chú thích trong C */`
`// Đây là chú thích trong C++`

- Quy tắc Pascal: viết hoa chữ cái đầu tiên của mỗi từ
 - Method, Interface, Enum, Events, Exception, Namespace, Property
- Quy tắc Camel: viết thường từ đầu tiên & chữ cái đầu tiên của từ kế tiếp
 - Bổ sung truy cập, Parameter

- 2 loại:
 - Kiểu dữ liệu gốc (Primitive Type);
 - Kiểu dữ liệu gốc modifier: sử dụng 1 trong các modifier sau:
 - signed
 - unsigned
 - short
 - long

Kiểu dữ liệu	Keyword	Byte
Boolean	bool	1
Ký tự	float	1
Số nguyên	int	4
Số thực	float	4
Số thực dạng double	double	8
Kiểu không có giá trị	void	



- Biến: vị trí bộ nhớ được dành riêng để lưu giá trị.
- 3 loại:
 - Biến giá trị;
 - Biến tham chiếu;
 - Biến con trỏ.
- Khai báo & Khởi tạo:
 - `int x;`
 - `x = 5;`
 - `int x = 5;`

- Phân loại theo phạm vi:
 - Biến cục bộ;
 - Biến toàn cục
 - ❖ Toán tử định phạm vi ::

```
#include <iostream>
using namespace std;
int x = 5;
int main()
{
    int x = 1;
    cout << x << ::x;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int x = 5;
int main()
{
    int y = 1;
    x = y;
    cout << x << y;
    return 0;
}
```

- **typedef**: tạo một tên mới cho một kiểu dữ liệu đang tồn tại:

typedef type name;

○ Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    typedef int SoNguyen;
    SoNguyen x = 5;
    cout << x;
    return 0;
}
```


- C: **(new_type) expression;**
- C++: vẫn sử dụng như trên.
 - Phép chuyển kiểu mới: **new_type (expression);**
→ Phép chuyển kiểu này có dạng như một hàm số chuyển kiểu đang được gọi.
 - ❖ **type_cast <new_type> (expression);**
 - type_cast: const_cast, static_cast, dynamic_cast, reinterpret_cast.



CONSTANT/LITERAL

- Có 2 cách định nghĩa:

- Sử dụng bộ tiền xử lý **#define**;

#define identifier value

- Sử dụng từ khóa **const**;

const type identifier = value;

type const identifier = value;

```
#include <iostream>
using namespace std;
#define PI 3.14
int main()
{
    const int x = 5;
    int const y = 6;
    cout << PI << x << y;
    return 0;
}
```

- **Kiểu liệt kê:** Là kiểu dữ liệu mà tất cả các phần tử của nó (có ý nghĩa tương đương nhau) có giá trị là hằng số:
 - Được định nghĩa thông qua từ khóa **enum**;
 - Khai báo enum không yêu cầu cấp phát bộ nhớ, chỉ cấp phát bộ nhớ lúc tạo biến kiểu enum;
 - Mặc định các phần tử của enum có kiểu dữ liệu **int**;
 - Enum sẽ được ép kiểu ngầm định sang kiểu int, nhưng kiểu int phải ép kiểu tường minh sang enum;
 - Biến kiểu enum chỉ có thể được gán giá trị là một trong số các hằng đã khai báo bên trong kiểu enum đó, không thể sử dụng hằng của kiểu enum khác;
 - Phạm vi sử dụng của một khai báo enum cũng tương tự như phạm vi sử dụng khi khai báo biến.

```
#include <iostream>
using namespace std;
int main()
{
    enum Color
    {
        RED = 1,
        BLUE = 2,
        GREEN = 3,
        YELLOW = 4
    };
    Color c = GREEN;
    cout << c;
    return 0;
}
```

- Toán tử số học: **+**, **-**, *****, **/**, **%**, **++**, **--**
 - Toán tử quan hệ: **==**, **!=**, **>**, **<**, **>=**, **<=**
 - Toán tử logic: **&&**, **||**, **!**
 - Toán tử so sánh bit: **&**, **|**, **^**, **~**, **<<**, **>>**
 - Toán tử gán: **=**, **+=**, **-=**, ***=**, **/=**, **%=**, **<<=**, **>>=**, **&=**, **^=**, **|=**
 - Toán tử hỗn hợp: **sizeof**, **?x:y**, **,**, **&**, *****
- ❖ *Toán tử có thứ tự ưu tiên.*

- Thư viện:
 - Khai báo tệp tiêu đề:
 - `include <iostream.h>`
 - Sử dụng câu lệnh nhập, xuất dữ liệu trong C++ cần sử dụng thêm **using namespace std**:

```
#include <iostream>
using namespace std;
int main()
{
    //code
    return 0;
}
```

- Toán tử xuất:
 - C: ???
 - C++:
 - Cú pháp:
`cout << biểu_thức_1 << ... << biểu_thức_n;`
 - Trong đó **cout** được định nghĩa trước như một đối tượng biểu diễn cho thiết bị xuất chuẩn của C++ là màn hình;
 - **cout** được sử dụng kết hợp với toán tử chèn **<<** để hiển thị giá trị các biểu thức 1, 2, ..., n ra màn hình;
 - Sử dụng “\n” hoặc **endl** để xuống dòng mới.

- Toán tử xuất:
 - Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    int x = 5;
    cout << x << endl;
    cout << "x = " << x << endl;
    return 0;
}
```


- Toán tử nhập:
 - C: ???
 - C++:
 - Cú pháp:
`cin >> biến_1 >> ... >> biến_n;`
 - Toán tử **cin** được định nghĩa trước như một đối tượng biểu diễn cho thiết bị vào chuẩn của C++ là bàn phím;
 - **cin** được sử dụng kết hợp với toán tử trích **>>** để nhập dữ liệu từ bàn phím cho các biến 1, 2, ..., n.

- Xuất dữ liệu:
 - Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    int x;
    cout << "Input x = ";
    cin >> x;
    cout << "x = " << x << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    char x, int y;
    cout << "Input x = ";
    cin >> x;
    y = x;
    cout << x << y << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    char x, int y;
    cout << "Input y = ";
    cin >> y;
    x = y;
    cout << x << y << endl;
    return 0;
}
```

- Quy định số thực được hiển thị ra màn hình với p chữ số sau dấu chấm thập phân, sử dụng đồng thời các hàm sau:

```
cout << setiosflags(ios::showpoint) << setprecision(p);
```

- **setiosflags(ios::showpoint):** bật cờ hiệu showpoint(p).
- Định dạng trên sẽ có hiệu lực đối với tất cả các toán tử xuất tiếp theo cho đến khi gặp một câu lệnh định dạng mới.



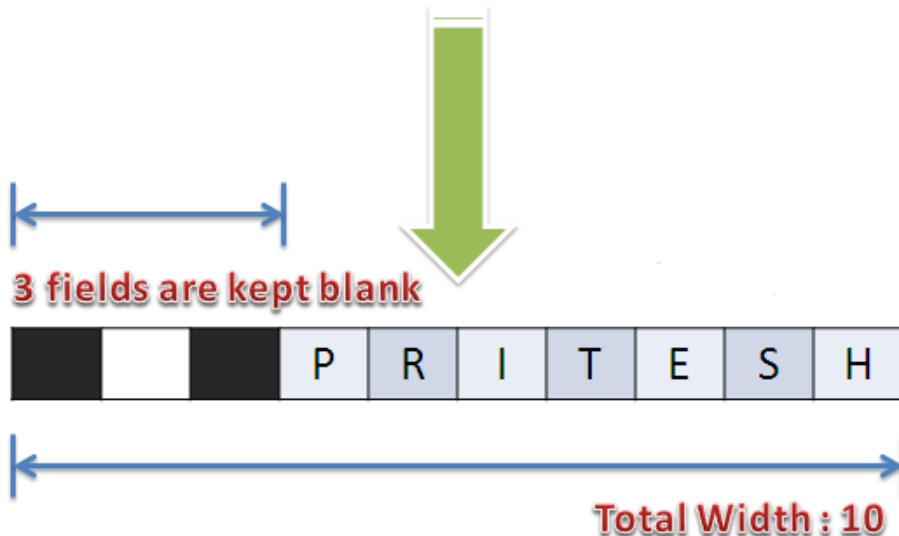
ĐỊNH DẠNG IN

- Ví dụ:

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    double x = 1234.5;
    //x = 1.234; x = 12.34; x = 123.4; x = 1234.5
    cout << setiosflags(ios::showpoint) << setprecision(3);
    cin >> x;
    return 0;
}
```

- Để quy định độ rộng tối thiểu để hiển thị là k vị trí cho giá trị (nguyên, thực, chuỗi) ta dùng hàm: **setw(k); (#include <iomanip>).**
- Hàm này cần đặt trong toán tử xuất và nó chỉ có hiệu lực cho một giá trị được in gần nhất. Các giá trị in ra tiếp theo sẽ có độ rộng tối thiểu mặc định là 0.

```
cout << setw(10) << "Pritesh";
```



```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    cout << setw(5) << "KHOA" << "CNTT";
    return 0;
}
```

- Biến tham chiếu là một tên khác (bí danh) cho biến đã định nghĩa:

Kiểu_dữ_liệu &Biến_tham_chiếu = biến_hằng;

- Biến tham chiếu có đặc điểm là nó được sử dụng làm bí danh cho một biến (kiểu giá trị) nào đó và sử dụng vùng nhớ của biến này;
- Biến tham chiếu không được cung cấp vùng nhớ (dùng chung địa chỉ vùng nhớ với biến mà nó tham chiếu đến);
- Trong khai báo biến tham chiếu phải chỉ rõ tham chiếu đến biến nào;
- Biến tham chiếu có thể tham chiếu đến một phần tử mảng, nhưng không cho phép khai báo mảng tham chiếu.

```
#include <iostream>
using namespace std;
int main()
{
    int x = 5;
    int &y = x;
    y = 1;
    cout << x;
    x++;
    cout << y;
    return 0;
}
```

- Cú pháp:
const Kiểu_dữ_liệu &Hằng = Biến/Hằng;

- Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    const int a = 1;
    int b = 2;
    const int &x = a; //x++
    const int &y = b; //y++
    const int &z = 3; //z++
    cout << x << y << z;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    int x = 1;
    const int &y = x;
    x++; //y++;
    cout << x << y;
    return 0;
}
```


- Khai báo nguyên mẫu hàm & Định nghĩa hàm;
- Ví dụ:

```
#include <iostream>
using namespace std;
void Sum(int x, int y);
int main()
{
    int m = 1, n = 2;
    Sum(m, n);
    return 0;
}
void Sum(int x, int y)
{
    cout << x + y;
}
```

```
#include <iostream>
using namespace std;
void Sum(int x, int y)
{
    cout << x + y;
}
int main()
{
    int m = 1, n = 2;
    Sum(m, n);
    return 0;
}
```

- Truyền tham trị & tham chiếu (biến tham chiếu hoặc biến con trỏ):

```
#include <iostream>
using namespace std;
void Swap(int x, int y)
{
    int temp = x;
    x = y;
    y = temp;
}
int main()
{
    int m = 1, n = 2;
    Swap(m, n);
    cout << m << n;
    return 0;
}
```

```
#include <iostream>
using namespace std;
void Swap(int &x, int &y)
{
    int temp = x;
    x = y;
    y = temp;
}
int main()
{
    int m = 1, n = 2;
    Swap(m, n);
    cout << m << n;
    return 0;
}
```

- Đối số bình thường:

```
#include <iostream>
using namespace std;
void Show(int m)
{
    cout << m++;
}
int main()
{
    int x = 1;
    const int y = 1;
    Show(x);    //1
    Show(y);    //1
    return 0;
}
```

```
#include <iostream>
using namespace std;
void Show(int &m)
{
    cout << m++;
}
int main()
{
    int x = 1;
    const int y = 1;
    Show(x);    //1
    Show(y);    //compile error
    return 0;
}
```



HÀM VỚI ĐỐI SỐ HẲNG

- Đối số hằng: sử dụng khi không muốn thay đổi giá trị đối số truyền vào:

```
#include <iostream>
using namespace std;
void Show(const int m)
{
    cout << m; //cout << m++;
}
int main()
{
    int x = 1;
    const int y = 1;
    Show(x);    //1
    Show(y);    //1
    return 0;
}
```



HÀM VỚI ĐỐI SỐ HẲNG

- Đối số hằng tham chiếu:

```
#include <iostream>
using namespace std;
void Show(int &m)
{
    m++;
}
int main()
{
    int x = 1;
    const int y = 1;
    Show(x);    //2
    Show(y);    //compile error
    cout << x << y;
    return 0;
}
```

```
#include <iostream>
using namespace std;
void Show(const int &m)
{
    cout << m++;    //compile error
}
void Display(const int &m)
{
    cout << m + 1;
}
int main()
{
    int x = 1;
    Show(x);
    Display(x);
    return 0;
}
```



HÀM VỚI ĐỐI SỐ MẶC ĐỊNH

- Đối số mặc định:
 - Định nghĩa các giá trị tham số mặc định cho các hàm;
 - Các đối số mặc định cần là các đối số cuối cùng tính từ trái qua phải;
 - Nếu chương trình sử dụng khai báo nguyên mẫu hàm thì các đối số mặc định cần được khởi gán trong nguyên mẫu hàm, không được khởi gán lại cho các đối số mặc định trong dòng đầu của định nghĩa hàm;
 - Khi xây dựng hàm, nếu không khai báo nguyên mẫu hàm thì các đối số mặc định được khởi gán trong dòng đầu của định nghĩa hàm;
 - Đối với các hàm có đối số mặc định thì lời gọi hàm cần viết theo quy định; các tham số vắng mặt trong lời gọi hàm tương ứng với các đối số mặc định cuối cùng (tính từ trái sang phải).



HÀM VỚI ĐỐI SỐ MẶC ĐỊNH

- Đối số mặc định:

```
#include <iostream>
using namespace std;
void Show(int x, int y = 1, int z = 2);
int main()
{
    Show();           //compile error
    Show(1);          //112
    Show(1, 2);        //122
    Show(1, 2, 3);     //123
    Show(1, , 1);      //compile error
    return 0;
}
void Show(int x, int y, int z)
{
    cout << x << y << z;
}
```

```
#include <iostream>
using namespace std;
void Show(int x, int y = 1, int z = 2)
{
    cout << x << y << z;
}
int main()
{
    Show();           //compile error
    Show(1);          //112
    Show(1, 2);        //122
    Show(1, 2, 3);     //123
    Show(1, , 1);      //compile error
    return 0;
}
```


HÀM TRẢ VỀ THAM CHIẾU

- Hàm trả về là một tham chiếu:

Kiểu &Tên_hàm(...)

{

//thân hàm

return <biến phạm vi toàn cục>;

}

- Biểu thức được trả lại trong câu lệnh **return** phải là biến toàn cục, khi đó mới có thể sử dụng được giá trị của hàm;
- Nếu trả về tham chiếu đến một biến cục bộ thì biến cục bộ này sẽ bị mất đi khi kết thúc thực hiện hàm. Do vậy tham chiếu của hàm sẽ không còn ý nghĩa nữa. Vì vậy, nếu hàm trả về là tham chiếu đến biến cục bộ thì biến cục bộ này **NÊN** khai báo **static**;
- Khi giá trị trả về của hàm là tham chiếu, ta có thể gặp các câu lệnh gán hơi khác thường, trong đó vế trái là một lời gọi hàm chứ không phải là tên của một biến.

HÀM TRẢ VỀ THAM CHIỀU

- Ví dụ:

```
#include <iostream>
using namespace std;
int x = 4;
int &Func()
{
    return x;
}
int main()
{
    cout << x;          //4
    cout << Func();     //4
    Func() = 8;
    cout << x;          //8
    system("pause");
    return 0;
}
```

```
#include <iostream>
using namespace std;
int x = 4;
int &Func()
{
    static int x = 5;
    return x;
}
int main()
{
    cout << x;          //4
    cout << Func();     //5
    Func() = 8;
    cout << x;          //4
    system("pause");
    return 0;
}
```

- Hàm trả về hằng tham chiếu:

```
#include <iostream>
using namespace std;
int &Func()
{
    static int x = 4;
    return x;
}
int main()
{
    cout << Func();
    cout << Func()++;
    return 0;
}
```

```
#include <iostream>
using namespace std;
const int &Func()
{
    static int x = 4;
    return x;
}
int main()
{
    cout << Func();
    cout << Func()++;    //compile error
    return 0;
}
```

- Hàm hằng:
 - `const int func(int value);`
 - `int func(int value) const;`

```
#include <iostream>
using namespace std;
const int Func()
{
    int x = 4;
    x++;
    return x;
}
int main()
{
    int x = Func();           //x = 4
    int y = Func()++;         //compile error
    return 0;
}
```

- Hàm nội tuyến (inline):
 - Hàm: làm chậm tốc độ thực hiện chương trình vì phải thực hiện một số thao tác có tính thủ tục khi gọi hàm:
 - Cấp phát vùng nhớ cho đối số và biến cục bộ;
 - Truyền dữ liệu của các tham số cho các đối;
 - Giải phóng vùng nhớ trước khi thoát khỏi hàm.
 - C++ cho khả năng khắc phục được nhược điểm trên bằng cách dùng hàm nội tuyến → dùng từ khóa inline trước khai báo nguyên mẫu hàm.
- ❖ Chú ý:
 - Hàm nội tuyến cần có từ khóa inline phải xuất hiện trước các lời gọi hàm;
 - Chỉ nên khai báo là hàm inline khi hàm có nội dung đơn giản;
 - Hàm đệ quy không thể là hàm inline.

- Ví dụ:

```
#include <iostream>
using namespace std;
inline int f(int a, int b);
int main()
{
    int s;
    s = f(5,6);
    cout << s;
    return 0;
}
int f(int a, int b)
{
    return a*b;
}
```

```
#include <iostream>
using namespace std;
int f(int a, int b);
int main()
{
    int s;
    s = f(5,6);
    cout << s;
    return 0;
}
int f(int a, int b)
{
    return a*b;
}
```

- Khái niệm:
 - Con trỏ là biến dùng để chứa địa chỉ của biến khác;
 - Cùng kiểu dữ liệu với kiểu dữ liệu của biến mà nó trỏ tới.
- Cú pháp: **<kiểu dữ liệu> *<tên con trỏ>;**
- ❖ Lưu ý:
 - Có thể viết dấu ***** ngay sau kiểu dữ liệu;
 - Ví dụ: **int *a;** và **int* a;** là tương đương.
- Thao tác với con trỏ:
 - ***** là toán tử thâm nhập (dereferencing operator): ***p** là giá trị nội dung vùng nhớ con trỏ đang trỏ đến;
 - **&** là toán tử địa chỉ (address of operator): **&x** là địa chỉ của biến **x**; nếu **int *p = &x** thì **p ↔ x**.

- Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    int x = 10, y = 20;
    int *p1, *p2;
    p1 = &x; p2 = &y;
    cout << "x = " << x;           //x = 10
    cout << "y = " << y;           //y = 20
    cout << "*p1 = " << *p1;       //*p1 = 10
    cout << "*p2 = " << *p2;       //*p2 = 20
    *p1 = 50; *p2 = 90;
    cout << "x = " << x;           //x = 50
    cout << "y = " << y;           //y = 90
    cout << "*p1 = " << *p1;       //*p1 = 50
    cout << "*p2 = " << *p2;       //*p2 = 90
    *p1 = *p2;
    cout << "x = " << x;           //x = 90
    cout << "y = " << y;           //y = 90
    cout << "*p1 = " << *p1;       //*p1 = 90
    cout << "*p2 = " << *p2;       //*p2 = 90
    return 0;
}
```



CON TRỎ HẲNG & HẲNG CON TRỎ

- Lưu ý:

- **const int x = 1;** và **int const x = 1;** là như nhau;

- **Nếu int x = 1; thì**

- **const int *p = &x; (CON TRỎ HẲNG)**

- Không cho phép thay đổi giá trị vùng nhớ mà con trỏ đang trỏ đến thông qua con trỏ (*p).

- **int* const p = &x; (HẲNG CON TRỎ)**

- Không cho phép thay đổi vùng nhớ con trỏ đang trỏ tới, nhưng có thể thay đổi giá trị vùng nhớ đó thông qua con trỏ.

- Phép gán giữa các con trỏ:
 - Các con trỏ cùng kiểu có thể gán cho nhau thông qua phép gán và lấy địa chỉ con trỏ
$$<\text{tên con trỏ 1}> = <\text{tên con trỏ 2}>;$$
 - Lưu ý:
 - Bắt buộc phải dùng phép **lấy địa chỉ của biến** do con trỏ trỏ tới mà không được dùng phép **lấy giá trị của biến**;
 - Hai con trỏ phải cùng kiểu dữ liệu (nếu khác kiểu phải sử dụng các phương thức ép kiểu).

- Phép gán giữa các con trỏ:

- Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    int x = 1;
    int *p1, *p2;
    p1 = &x;
    cout << *p1;    //*p1 = 1
    *p1 += 2;
    cout << x;      //x = 3
    p2 = p1;
    *p2 += 3;
    cout << x;      //x = 6
    return 0;
}
```

- Sử dụng **typedef**:

```
#include <iostream>
using namespace std;
typedef int P;
typedef int* Q;
int main()
{
    int x = 1, y = 2;
    P *p1, *p2;
    p1 = &x; p2 = &y;
    cout << *p1 << *p2; //12
    Q p3, p4;
    p3 = &x; p4 = &y;
    cout << *p3 << *p4; //12
    P* p5, p6;
    p5 = &x;
    p6 = &y;    //compile error
    return 0;
}
```

- **const_cast**: thêm hoặc loại bỏ const khỏi một biến (là cách duy nhất để thay đổi tính hằng của biến).
- Ví dụ:

```
#include <iostream>
using namespace std;
void Show(char *str)
{
    cout << str;
}
int main()
{
    const char *str = "DUT";
    Show(str); //compile error
    Show(const_cast<char*>(str));
    return 0;
}
```

- Đối số con trỏ;
- Hàm trả về là tham chiếu.

```
#include <iostream>
using namespace std;
void Swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}
int main()
{
    int m = 1, n = 2;
    Swap(&m, &n);
    cout << m << n;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int &Func()
{
    int x = 5;
    return x;
}
int main()
{
    int *p = &Func();
    //cout << *p << *p;
    cout << *p;
    cout << *p;
    return 0;
}
```


- Khai báo & khởi tạo mảng:
 - `int A[5];`
 - `int A[5] = {1, 2, 3, 4, 5};`
 - `int A[] = {1, 2, 3, 4};`
 - `int A[2][3];`
 - `int A[2][3] = {{1, 2, 3}, {4, 5, 6}};`
 - `int A[2][3] = {1, 2, 3, 4, 5, 6};`
 - `int A[][3] = {{1, 2, 3}, {4, 5, 6}};`
 - ❖ Không được bỏ trống cột với mảng 2 chiều.
- Truy cập phần tử trong mảng:
 - `A[i]` – giá trị phần tử thứ `i`; `&A[i]` – địa chỉ phần tử thứ `i`;
 - `A[i][j]` – giá trị phần tử hàng `i`, cột `j`;
 - `&A[i]` – địa chỉ hàng `i`;
 - `&A[i][j]` – địa chỉ phần tử hàng `i`, cột `j`.

- Quan hệ giữa con trỏ và mảng 1 chiều:
 - Tên mảng được coi như một con trỏ tới phần tử đầu tiên của mảng:

```
int A[6] = {2,4,6,8,10,12};  
int *P;  
P = A; // P points to A
```



- Do tên mảng và con trỏ là tương đương, ta có thể dùng P như tên mảng. Ví dụ:

P[3] = 7; tương đương với A[3] = 7;

- Phép toán trên con trỏ và mảng 1 chiều:
 - Khi con trỏ trỏ đến mảng, thì các phép toán tăng hay giảm trên con trỏ sẽ tương ứng với phép dịch chuyển trên mảng;
 - ❖ Lưu ý: nếu **int *p = A;** thì **p++** và ***p++** là khác nhau
 - **p++** thao tác trên con trỏ: đưa con trỏ pa đến địa chỉ phần tử tiếp theo;
 - ***p++** là phép toán trên giá trị, tăng giá trị phần tử hiện tại của mảng.

- Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    int A[] = {1, 2, 3, 4, 5};
    int *p = A;
    p += 2;
    cout << *p; // *p = A[2] = 3
    p--;
    cout << *p; // *p = A[1] = 2
    *p++;
    cout << *p; // *p = A[1] = 3
    return 0;
}
```

- **Lưu ý:**

- **cin.get(a, n);** - nhập một chuỗi không quá n ký tự và lưu vào mảng một chiều a (kiểu char);
 - Toán tử nhập **cin>>** sẽ để lại ký tự chuyển dòng '**\n**' trong bộ đệm → làm trôi phương thức **cin.get**;
 - Để khắc phục tình trạng trên cần dùng phương thức **cin.ignore(1)** để bỏ qua một ký tự chuyển dòng.

- Con trỏ & mảng 2 chiều

- Địa chỉ ma trận A:

$$\mathbf{A} = \mathbf{A}[0] = *(\mathbf{A}+0) = \&\mathbf{A}[0][0];$$

- Địa chỉ của hàng thứ nhất:

$$\mathbf{A}[1] = *(\mathbf{A}+1) = \&\mathbf{A}[1][0];$$

- Địa chỉ của hàng thứ i:

$$\mathbf{A}[i] = *(\mathbf{A}+i) = \&\mathbf{A}[i][0];$$

- Địa chỉ phần tử:

$$\&\mathbf{A}[i][j] = (*(\mathbf{A}+i)) + j;$$

- Giá trị phần tử:

$$\mathbf{A}[i][j] = *((*(\mathbf{A}+i)) + j);$$

→ `int A[3][3];` tương đương `int (*A)[3];`.

- Con trỏ tới con trỏ:
 - **int A[3][3];** tương đương **int (*A)[3];**
 - **int A[3];** tương đương **int *A;**
 - Mảng 2 chiều có thể thay thế bằng một mảng các con trỏ tương đương một con trỏ trỏ đến con trỏ.
 - Ví dụ:

```
int A[3][3];  
int (*A)[3];  
int **A;
```


- Con trỏ & mảng 2 chiều
 - Bài tập: Chương trình sau gồm các hàm:
 - Nhập một ma trận thực cấp $m \times n$
 - In một ma trận thực dưới dạng bảng
 - Tìm phần tử lớn nhất và phần tử nhỏ nhất của dãy số thực;

Viết chương trình sẽ nhập một ma trận, in ma trận vừa nhập và in các phần tử lớn nhất và nhỏ nhất trên mỗi hàng của ma trận.



FUNCTION POINTER

- Mỗi hàm đều có 1 địa chỉ xác định trên bộ nhớ → có thể sử dụng con trỏ để trỏ đến địa chỉ của hàm.

```
#include <iostream>
using namespace std;
int Show(int x)
{
    return x;
}
int main()
{
    int n = 5;
    cout << Show(n);    //5
    cout << Show;      //Address of function Show
    return 0;
}
```

- ❖ Với compiler của Visual Studio (Window) thì kết quả đúng, với compiler của Java thì kết quả bằng 1.



FUNCTION POINTER

- Khai báo:
<kiểu dữ liệu trả về> (*<tên hàm>) ([<danh sách tham số>])
- Lưu ý:
 - Dấu “()” bao bọc tên hàm để thông báo đó là con trỏ hàm;
 - Nếu không có dấu “()” thì trình biên dịch sẽ hiểu ta đang khai báo một hàm có giá trị trả về là một con trỏ.
 - Ví dụ:

```
int (*Cal) (int a, int b)    //Khai báo con trỏ hàm  
int *Cal (int a, int b)    //Khai báo hàm trả về kiểu con trỏ
```
 - Đối số mặc định không áp dụng cho con trỏ hàm, vì đối số mặc định được cấp phát khi biên dịch, còn con trỏ hàm được sử dụng lúc thực thi.



FUNCTION POINTER

- Ví dụ:

```
#include <iostream>
using namespace std;
void Swap(int &x, int &y)
{
    int temp = x;
    x = y;
    y = temp;
}
int main()
{
    int m = 5, n = 10;
    void(*pSwap) (int &, int &) = Swap;
    (*pSwap)(m, n); //m = 10, n = 5
    return 0;
}
```



FUNCTION POINTER

- Sử dụng con trỏ hàm làm đối số:
 - Sử dụng khi cần gọi 1 hàm như là tham số của 1 hàm khác;
 - Ví dụ: `int (*Cal)(int a, int b);`
 - Thì có thể gọi các hàm có hai tham số kiểu `int` và trả về cũng kiểu `int`:
`int add(int a, int b);`
`int sub(int a, int b);`
 - Nhưng không được gọi các hàm khác kiểu tham số hoặc kiểu trả về:
`int add(float a, int b);`
`int add(int a);`
`char* sub(char* a, char* b);`



FUNCTION POINTER

- Sử dụng con trỏ hàm làm đối số:
 - Ví dụ: sắp xếp dữ liệu trong mảng số nguyên theo thứ tự tăng dần

```
#include <iostream>
#include <iomanip>
using namespace std;
void Show(int *p, int length)
{
    for (int i = 0; i < length; i++)
        cout << setw(3) << *(p + i);
}
void Swap(int &x, int &y)
{
    int temp = x;
    x = y;
    y = temp;
}

void SelectionSort(int *p, int length)
{
    for (int i = 0; i < length - 1; i++)
        for (int j = i + 1; j < length; j++)
            if (*(p + i) > *(p + j))
                Swap(*(p + i), *(p + j));
}

int main()
{
    int A[] = {1, 4, 2, 3, 6, 5, 8, 9, 7};
    int l = sizeof(A) / sizeof(int);
    SelectionSort(A, l);
    Show(A, l);
    return 0;
}
```

- Nhưng nếu muốn sắp xếp theo thứ tự giảm dần ???



FUNCTION POINTER

- Sử dụng con trỏ hàm làm đối số:
 - Thêm 2 hàm so sánh:

```
bool ascending(int left, int right)
{
    return left > right;
}
bool descending(int left, int right)
{
    return left < right;
}
```

- ascending: sắp xếp tăng dần
- Descending: sắp xếp giảm dần



FUNCTION POINTER

- Sử dụng con trỏ hàm làm đối số:

```
void SelectionSort(int *p, int length, bool CompFunc(int, int))
{
    for (int i = 0; i < length - 1; i++)
        for (int j = i + 1; j < length; j++)
            if (CompFunc(*(p + i), *(p + j)))
                Swap(*(p + i), *(p + j));
}

int main()
{
    int A[] = {1, 4, 2, 3, 6, 5, 8, 9, 7};
    int l = sizeof(A) / sizeof(int);
    SelectionSort(A, l, ascending);
    SelectionSort(A, l, descending);
    return 0;
}
```

- ❖ Lưu ý: Con trỏ hàm là đối số mặc định

```
void SelectionSort(int *p, int length,
                  bool CompFunc(int, int) = ascending)
```



STATIC & DYNAMIC ALLOWCATION

- **Cấp phát tĩnh (static allocation):** biến được cấp phát vùng nhớ khi biên dịch;
- **Cấp phát động (dynamic allocation):** biến được cấp phát vùng nhớ lúc thực thi:
 - Sử dụng từ khóa **new**;
 - Dùng biến con trỏ (**p**) để lưu trữ địa chỉ vùng nhớ:
 - Cấp phát bộ nhớ 1 biến: **p = new type**;
 - Cấp phát bộ nhớ 1 mảng: **p = new type[n]**;



STATIC & DYNAMIC ALLOWCATION

- Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    //Cấp phát tĩnh
    int x = 1; int *p1 = &x;
    //Cấp phát động
    int *p2 = new int; *p2 = 2;
    cout << x;          //1
    cout << *p1;         //1
    cout << *p2;         //2
    p1 = p2;
    p2 = new int; *p2 = 3;
    cout << *p1;         //2
    cout << *p2;         //3
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    //Cấp phát tĩnh
    int A[] = {1, 2, 3, 4}; int *p1 = A;
    //Cấp phát động
    int m = 5;
    int *p2 = new int[m];
    for (int i = 0; i < m; i++)
        *(p2 + i) = i++;
    return 0;
}
```



STATIC & DYNAMIC ALLOWCATION

- Kiểm tra vùng nhớ cấp phát có thành công hay không?
 - *Thành công*: con trỏ chứa địa chỉ đầu vùng nhớ được cấp phát; *Không thành công*: $p = \text{NULL}$.
 - Kiểm tra vùng nhớ cấp phát có thành công hay không bằng con trỏ hàm: `new_handler (new.h)`
 - Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    int *p = new int;
    if (p == NULL)
    {
        cout << "Error";
        exit(0);
    }
    return 0;
}
```

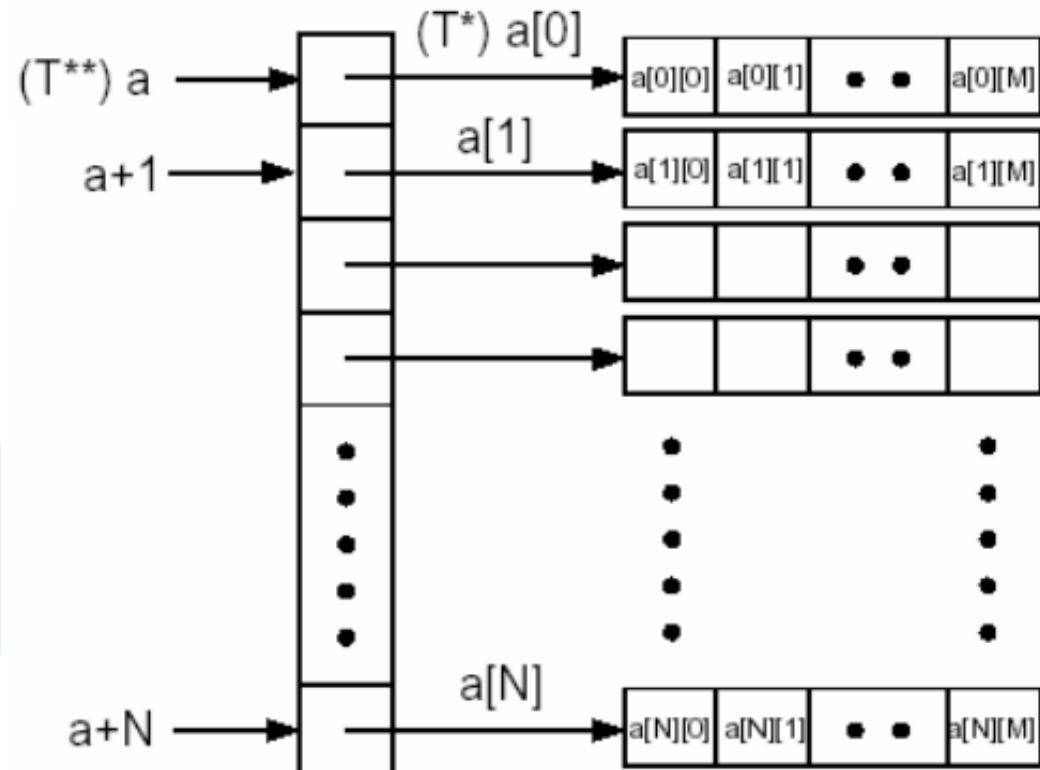


STATIC & DYNAMIC ALLOWCATION

- Giải phóng bộ nhớ trong C++:

```
#include <iostream>
using namespace std;
int main()
{
    int *p1 = new int;
    *p1 = 1;
    delete p1;
    int n = 3;
    int *p2 = new int[n];
    for (int i = 0; i < n; i++)
        *(p2 + i) = i++;
    delete[] p2;
    return 0;
}
```

- Cấp phát động mảng đa chiều:
 - Cấp phát động mảng hai chiều $(N+1)(M+1)$ gồm các đối tượng IQ:



```

IQ **a = new (IQ*) [N+1];
for (int i=0; i<N+1; i++)
    a[i] = new IQ[M+1];
  
```

- Huỷ mảng động bất hợp lệ:

```
#include <iostream>
int main ()
{
    int* A = new int[6];
    // dynamically allocate array
    A[0] = 0; A[1] = 1; A[2] = 2;
    A[3] = 3; A[4] = 4; A[5] = 5;
    int *p = A + 2;
    cout << "A[1] = " << A[1] << endl;
    delete [] p; // illegal!!!
    // results depend on particular compiler
    cout << "A[1] = " << A[1] << endl;
}
```

P không trở tới
đầu mảng A

Huỷ không
hợp lệ

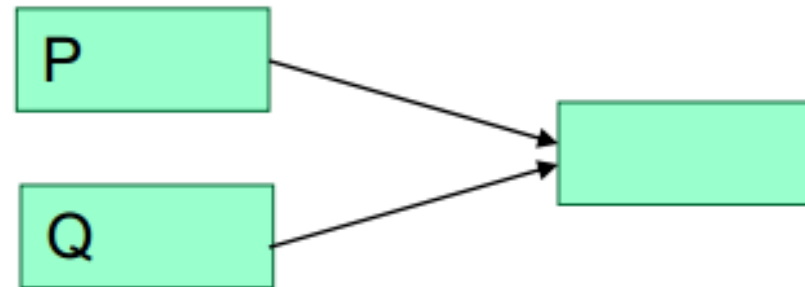
Kết quả phụ
thuộc trình
biên dịch

DANGLING POINTER

- Con trỏ lạc: khi delete P, ta cần chú ý không xóa vùng bộ nhớ mà một con trỏ Q khác đang trỏ tới:

```
int *P;  
int *Q;  
P = new int;  
Q = P;
```

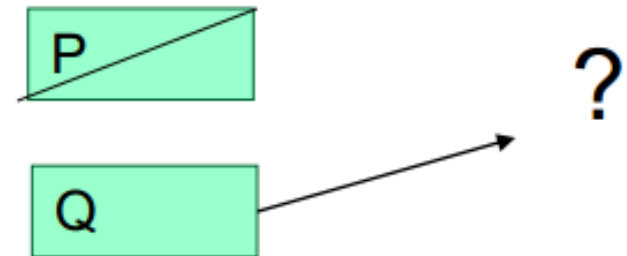
tạo



- Sau đó:

```
delete P;  
P = NULL;
```

Làm Q bị lạc

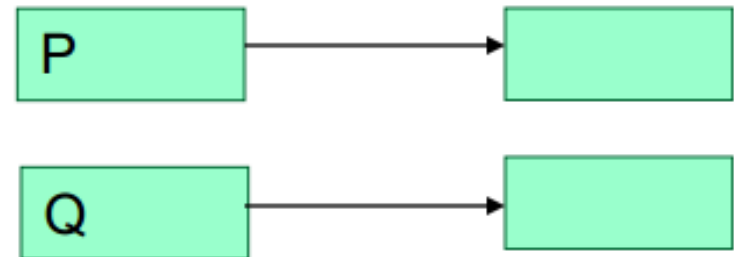


- Rò rỉ bộ nhớ:

- Một vấn đề liên quan: *mất* mọi con trỏ đến một vùng bộ nhớ được cấp phát. Khi đó, vùng bộ nhớ đó bị mất dấu, không thể trả lại cho heap được:

```
int *P;  
int *Q;  
P = new int;  
Q = new int;
```

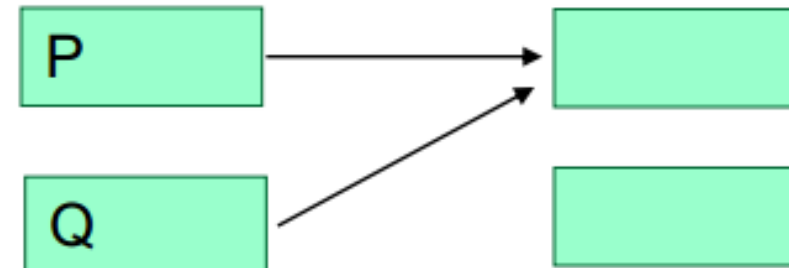
tạo



- Sau đó:

```
Q = P;
```

Làm mất vùng
nhớ đã từng
được Q trỏ tới



Thank You !

