

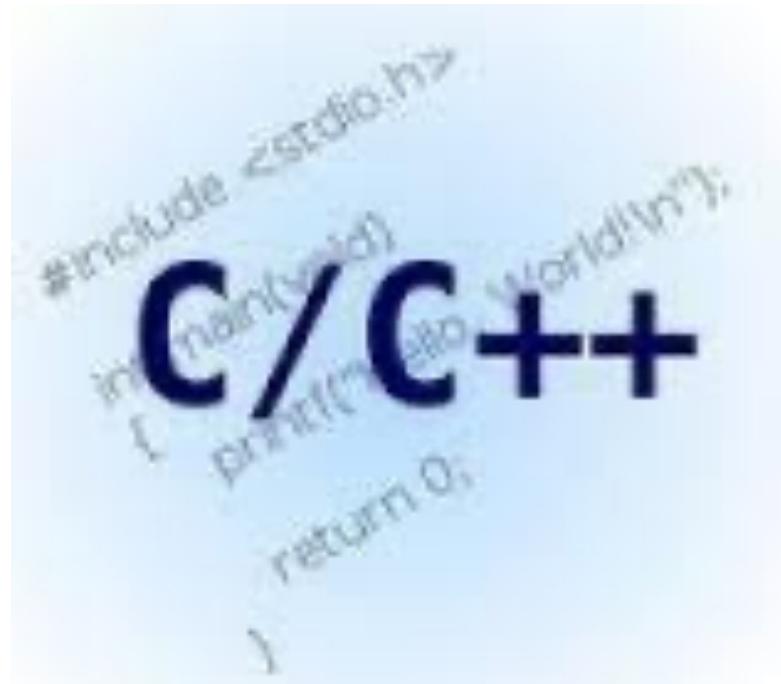
CHƯƠNG 1:

MỞ RỘNG CỦA

C++

TS. LÊ THỊ MỸ HẠNH
Bộ môn Công nghệ Phần mềm
Khoa Công Nghệ Thông Tin
Đại học Bách khoa – Đại học Đà Nẵng

Giới thiệu



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

Từ khóa

- Để bổ sung các tính năng mới vào C, một số từ khóa (keyword) mới đã được đưa vào C++ ngoài các từ khóa có trong C
- Các chương trình bằng C nào sử dụng các tên trùng với các từ khóa cần phải thay đổi trước khi chương trình được dịch lại bằng C++.
- Các từ khóa mới này là :

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

Chú thích

- chú thích trong C bằng /* ... */
- C++ đưa thêm chú thích bắt đầu bằng //.
 - kiểu chú thích /*...*/ được dùng cho các khối chú thích lớn gồm nhiều dòng,
 - còn kiểu // được dùng cho các chú thích trên một dòng.

Ví dụ: /* Đây là

chú thích trong C */

// Đây là chú thích trong C++

Biến

- Biến: vùng 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;
```

<Kiểu Dữ Liệu> <Tên biến>;

Hoặc:

<Kiểu Dữ Liệu> <Tên biến> = <Giá trị>;

Biến

- Trong C tất cả các câu lệnh khai báo biến, mảng cục bộ phải đặt tại đầu khối.
 - vị trí khai báo và vị trí sử dụng của biến có thể ở cách khá xa nhau, điều này gây khó khăn trong việc kiểm soát chương trình.
- C++ đã khắc phục nhược điểm này bằng cách cho phép các lệnh khai báo biến có thể đặt bất kỳ chỗ nào trong chương trình trước khi các biến được sử dụng.
 - Phạm vi hoạt động của các biến kiểu này là khối trong đó biến được khai báo.

Biến

■ 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;
}
```

Chuyển kiểu (CASTING)

- 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.

Khai báo hằng

- 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;
}
```

Nhập xuất dữ liệu

■ Trong C:

- `printf`
- `scanf`
- Khai báo thư viện: `# include <stdio.h>`

■ Trong C++:

- Biến dòng nhập `cin` và biến dòng xuất `cout`
- Khai báo thư viện: `#include <iostream.h>`
- Sử dụng câu lệnh nhập, xuất dữ liệu trong C++ cần sử dụng thêm `using namespace std`

Xuất dữ liệu

- 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.

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

Định dạng xuất

- 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);
```

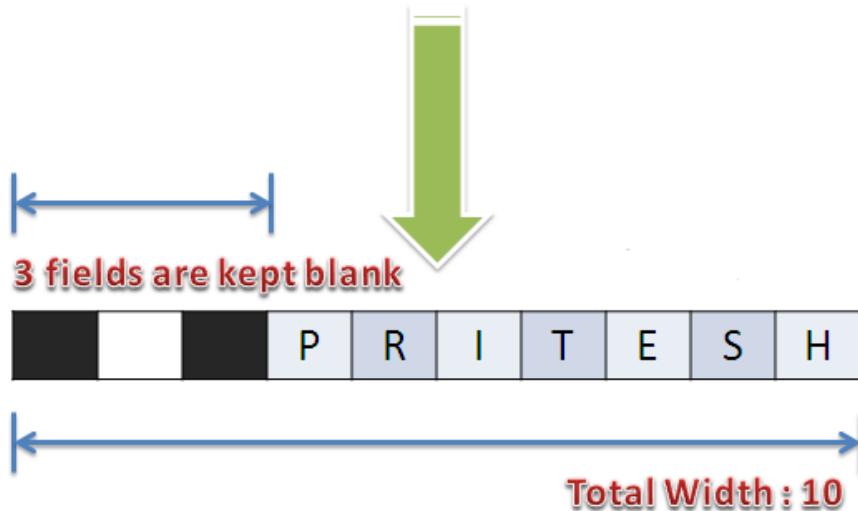
- setiosflags(ios::showpoint)**: bật cờ hiệu showpoint(p).
- Định dạng trên sẽ có hiệu lực đối với tất cả các toán tử xuất tiếp theo cho đến khi gặp một câu lệnh định dạng mới.

```
#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;
}
```

Định dạng xuất

- 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) ; (cần khai báo **#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;
}
```

Nhập dữ liệu

- 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.

```
#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;
}
```

Biến tham chiếu

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

Kiểu_dữ_liệu &Biến_tham_chiếu = biến;

- 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;
}
```

Hằng tham chiếu

■ Cú pháp:

const Kiểu_dữ_liệu &Hằng = Biến/Hằng;

■ Ví dụ:

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

```
#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;
}
```

Hàm

■ Khai báo nguyên mẫu hàm & Định nghĩa hàm

```
#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;
}
```

Hàm

- Truyền tham số: 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;
}
```

Hàm

■ Đối số

```
#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

- 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

■ 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

■ 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

■ 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

■ Hàm trả về là tham chiếu

- 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 gắp các câu lệnh gán hơi khác thường, trong đó về trái là một lời gọi hàm chứ không phải là tên của một biến.

```
Kiểu &Tên_hàm( . . . ) {  
    //thân hàm  
    return <biến_toàn_cục>;  
}
```

Hàm

■ Hàm trả là tham chiếu

```
#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

■ 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 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.

Hàm nội tuyến (inline)

```
#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;
}
```

Con trỏ

■ Khái niệm:

- Con trỏ là biến dùng để chứa địa chỉ của ô nhớ khác
- Cùng kiểu dữ liệu với kiểu dữ liệu của ô nhớ 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.

Con trỏ

```
#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

- Con trỏ hằng và hằng con trỏ
 - **const int x = 1;** và **int const x = 1;** là như nhau;

- Nếu **int x = 1;** thì

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

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

- **int* const p = &x;** (**HẰNG 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.

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
<ten con trỏ 1> = <ten 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).

Con trỏ

- 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;
}
```

Con trỏ

■ 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;
}
```

Con trỏ

■ **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;
}
```

Con trỏ và Mảng

- Khai báo & khởi tạo mảng:
 - int A[5];
 - intA[5] = {1, 2, 3, 4, 5};
 - intA[] = {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.

Con trỏ và Mảng

- 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;**

Con trỏ và Mảng

- 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.

Con trỏ và Mảng

■ Phép toán trên con trỏ và mảng 1 chiều:

□ 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;
}
```

Con trỏ và Mảng

■ Con trỏ & mảng 2 chiều

□ Địa chỉ ma trận A:

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

□ Địa chỉ của hàng thứ nhất:

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

□ Địa chỉ của hàng thứ i:

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

□ Địa chỉ phần tử:

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

□ Giá trị phần tử:

$$A[i][j] = *((*(A+i)) + j);$$

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

Con trỏ và Mảng

■ 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 trả hàm

- Mỗi hàm đều có 1 địa chỉ xác định trên bộ nhớ → có thể sử dụng con trả để trỏ đến địa chỉ của hàm.
 - Con trả đến các hàm
 - Con trả chứa địa chỉ của hàm
 - Tên hàm là địa chỉ bắt đầu của đoạn mã định nghĩa hàm
 - Con trả hàm có thể được
 - Truyền cho các hàm
 - Trả về từ các hàm
 - Được lưu trữ trong mảng
 - Được gán cho các con trả hàm khác

Con trả hàm

- 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 trả 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 trả.

- Ví dụ:

int (*Cal) (int a, int b) //Khai báo con trả hàm

int *Cal (int a, int b) //Khai báo hàm trả về kiểu con trả

- Đối số mặc định không áp dụng cho con trả hàm, vì đối số mặc định được cấp phát khi biên dịch, còn con trả hàm được sử dụng lúc thực thi.

Con trỏ hàm

■ Ví dụ

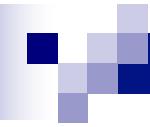
Swap(m,n)
Swap

```
#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;
}
```

Gọi thông
qua con
trỏ hàm

Con trả hàm

- 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);



Con trả hàm

■ Sử dụng con trả hàm làm đối số:

- Sử dụng khi cần gọi 1 hàm như là tham số của 1 hàm khác;
- Con trả hàm cũng là một biến con trả, do đó chúng ta có thể **sử dụng con trả hàm là tham số** của một hàm nào đó.
- Khi tham số của hàm là con trả hàm, đối số chính là địa chỉ của hàm.



Con trỏ hàm

■ Sử dụng con trỏ hàm làm đối số:

//Một số hàm tính đơn giản

```
int Cong(int a, int b) { return a + b; }
```

```
int Tru(int a, int b) { return a - b; }
```

```
int Nhan(int a, int b) { return a * b; }
```

//Định nghĩa kiểu con trỏ hàm PhepTinh

```
typedef int (*PhepTinh)(int, int);
```

//Định nghĩa hàm tính toán tổng quát

//với tham số là 2 số nguyên và con trỏ PhepTinh

```
int TinhToan(int a, int b, PhepTinh tinh) {
```

//Gọi hàm thông qua con trỏ hàm

```
    return tinh(a, b);
```

```
}
```

//Một số lời gọi hàm

```
int tong = TinhToan(5, 9, Cong);
```

```
int hieu = TinhToan(5, 9, Tru);
```

Con trả hàm

■ Sử dụng con trả 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 ???

Con trả hàm

- Sử dụng con trả 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

Con trỏ hàm

- 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)
```

Cấp phát động

- **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];**
- 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 = NULL.**

Cấp phát động

■ 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;
}
```

Cấp phát động

■ Giải phóng bộ nhớ trong C++:

- **delete biến** **contrô**
- **delete [] biến** **contrô**

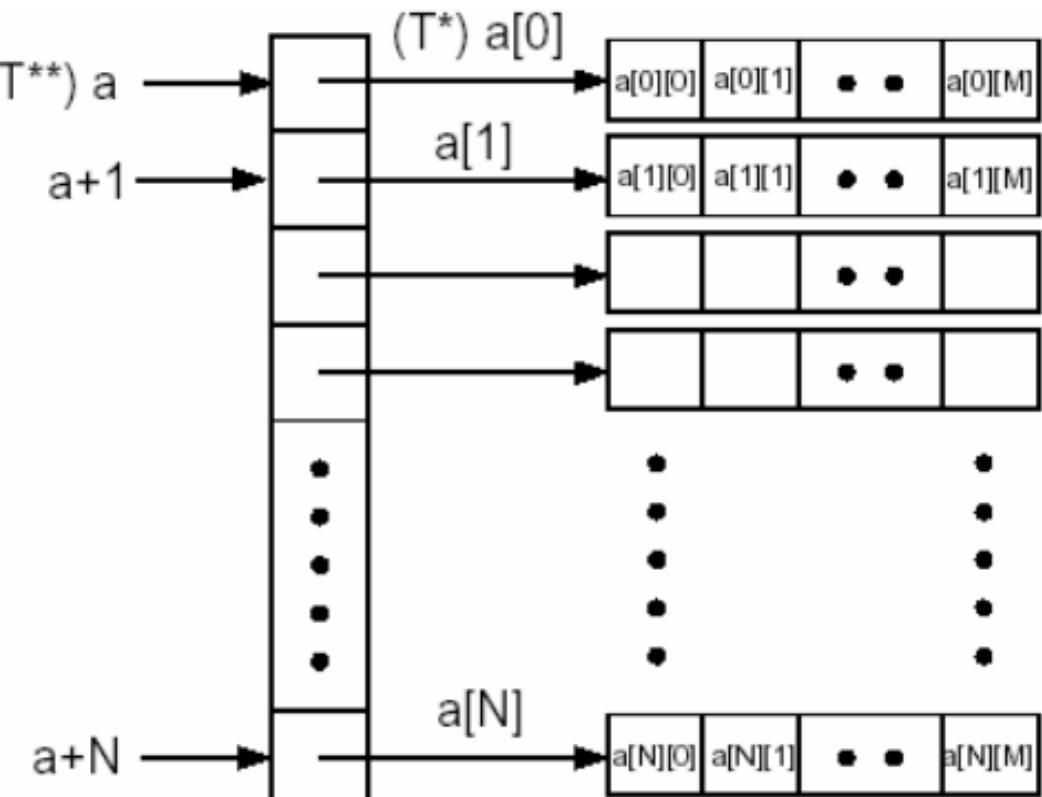
```
#include <iostream>
using namespace std;
int main()
{
    int *p1 = new int;
    *p1 = 1;
    delete p1;
    int n = 3;
    int *p2 = new int[n];
    for (int i = 0; i < n; i++)
        *(p2 + i) = i++;
    delete[] p2;
    return 0;
}
```

Cấp phát động

■ 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];
```



Cấp phát động

■ 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 trả tới
đầu mảng A

Huỷ không
hợp lệ

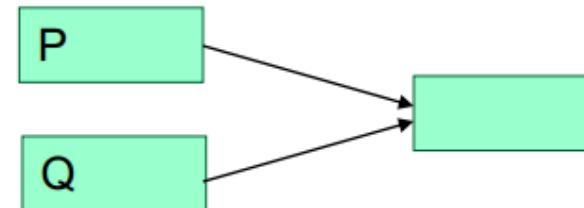
Kết quả phụ
thuộc trình
biên dịch

Cấp phát động

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

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

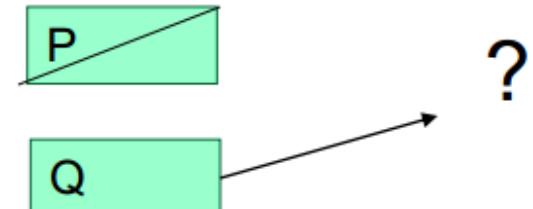
tạo



- Sau đó

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

Làm Q bị lạc



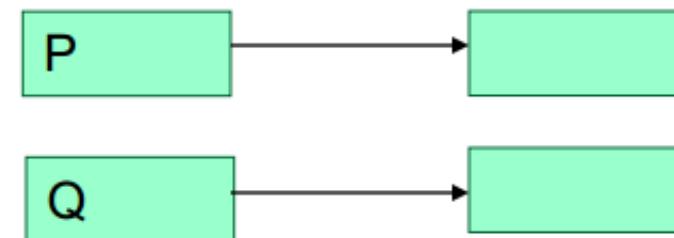
Cấp phát động

■ 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;
```

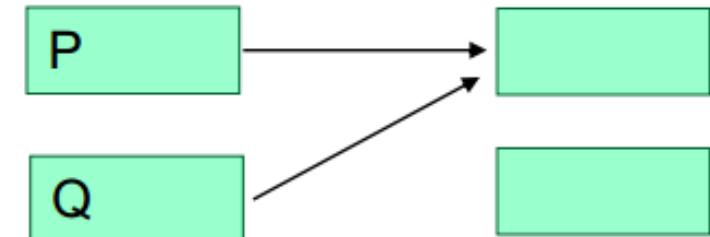
tạo



- Sau đó

```
Q = P;
```

Làm mất vùng
nhớ đã từng
được Q trỏ tới



Đa năng hóa (Overloading)

- Với ngôn ngữ C++, chúng ta có thể đa năng hóa các hàm và các toán tử (operator).
- Đa năng hóa là phương pháp cung cấp nhiều hơn một định nghĩa cho tên hàm đã cho trong cùng một phạm vi.
- Trình biên dịch sẽ lựa chọn phiên bản thích hợp của hàm hay toán tử dựa trên các tham số mà nó được gọi.
- Có hai hình thức đa năng hóa:
 - Đa năng hóa hàm
 - Đa năng hóa toán tử

Đa năng hóa hàm (Function Overloading)

- Trong ngôn ngữ C cũng như mọi ngôn ngữ máy tính khác, mỗi hàm đều phải có một tên phân biệt.
 - Như trong ngôn ngữ C, vì cần thiết phải có tên phân biệt nên C phải có hàm riêng cho mỗi kiểu dữ liệu số,
 - do vậy có tới ba hàm khác nhau để trả về trị tuyệt đối của một tham số :
- int abs(int i);
long labs(long l);
double fabs(double d);
- Tất cả các hàm này đều cùng thực hiện một chức năng nên chúng ta thấy điều này nghịch lý khi phải có ba tên khác nhau.
- C++ giải quyết điều này bằng cách cho phép chúng ta tạo ra các hàm khác nhau có cùng một tên. Đây chính là đa năng hóa hàm.
- Như vậy, trong C++ chúng ta có thể định nghĩa lại các hàm trả về trị tuyệt đối để thay thế các hàm trên như sau :

```
int Myabs(int i);  
long Myabs(long l);  
double Myabs(double d);
```

Đa năng hóa hàm (Function Overloading)

Trình biên dịch dựa vào sự khác nhau về số các tham số, kiểu của các tham số để có thể xác định chính xác phiên bản cài đặt nào của hàm MyAbs() thích hợp với một lệnh gọi hàm được cho,

```
MyAbs (-7); //Gọi hàm int MyAbs (int)  
MyAbs (-71); //Gọi hàm long MyAbs (long)  
MyAbs (-7.5); //Gọi hàm double MyAbs (double)
```

Quá trình tìm hàm đa năng hóa :

- nếu tìm thấy một phiên bản định nghĩa nào đó của một hàm được đa năng hóa mà có kiểu dữ liệu các tham số của nó trùng với kiểu các tham số đã gởi tới trong lệnh gọi hàm thì phiên bản hàm đó sẽ được gọi.
- Nếu không trình biên dịch C++ sẽ gọi đến phiên bản nào cho phép chuyển kiểu dễ dàng nhất.

```
MyAbs ('c'); //Gọi int MyAbs (int)  
MyAbs (2.34f); //Gọi double MyAbs (double)
```

Bất kỳ hai hàm nào trong tập các hàm đa năng phải có các tham số khác nhau.

Đa năng hóa toán tử (Operators overloading)

Trong C, khi tạo ra một kiểu dữ liệu mới, để thực hiện các thao tác liên quan đến kiểu dữ liệu đó thường thông qua các hàm

Ví dụ:

```
typedef struct Complex{  
    double Real;  
    double Imaginary;  
};  
Complex SetComplex(double R,double I);  
Complex AddComplex(Complex C1,Complex C2);  
Complex SubComplex(Complex C1,Complex C2);  
=> C3 = AddComplex(C1,C2); //Hơi bất tiện !!!  
      C4 = SubComplex(C1,C2);
```

Điều này trở nên không thoải mái vì thực chất thao tác cộng và trừ là các toán tử chứ không phải là hàm.

Đa năng hóa toán tử (Operators overloading)

Để khắc phục yếu điểm này, trong C++ cho phép chúng ta có thể định nghĩa lại chức năng của các toán tử đã có sẵn một cách tiện lợi và tự nhiên hơn rất nhiều.

Điều này gọi là đa năng hóa toán tử..

Ví dụ:

```
Complex operator + (Complex C1,Complex C2);  
Complex operator - (Complex C1,Complex C2);  
=>      C3 = C1 + C2;  
        C4 = C1 - C2;
```

Như vậy trong C++, các phép toán trên các giá trị kiểu số phức được thực hiện bằng các toán tử toán học chuẩn chứ không phải bằng các tên hàm như trong C.

Chẳng hạn chúng ta có lệnh sau:

```
C4 = AddComplex (C3, SubComplex (C1,C2)) ;
```

thì ở trong C++, chúng ta có lệnh tương ứng như sau:

```
C4 = C3 + C1 - C2;
```

Đa năng hóa toán tử (Operators overloading)

Cú pháp:

```
data_type operator oper_sym( parameters ) {  
    .....  
}
```

Trong đó:

data_type: Kiểu trả về.

operator_symbol: Ký hiệu của toán tử.

parameters: Các tham số (nếu có).

Các toán tử được đa năng hóa sẽ được lựa chọn bởi trình biên dịch:

- khi gặp một toán tử làm việc trên các kiểu không phải là kiểu có sẵn, trình biên dịch sẽ tìm một hàm định nghĩa của toán tử nào đó có các tham số đối sánh với các toán hạng để dùng.

Đa năng hóa toán tử (Operators overloading)

Các giới hạn của đa năng hóa toán tử:

- Chúng ta không thể định nghĩa các toán tử mới.
- Hầu hết các toán tử của C++ đều có thể được đa năng hóa.
- Các toán tử sau không được đa năng hóa là :
 - `::`: Toán tử định phạm vi.
 - `.*`: Truy cập đến con trỏ là trường của struct hay thành viên của class.
 - `.`: Truy cập đến trường của struct hay thành viên của class.
 - `?:`: Toán tử điều kiện
 - `sizeof`
- và chúng ta cũng không thể đa năng hóa bất kỳ ký hiệu tiền xử lý nào.
- Chúng ta không thể thay đổi thứ tự ưu tiên của một toán tử hay không thể thay đổi số các toán hạng của nó.
- Chúng ta không thể thay đổi ý nghĩa của các toán tử khi áp dụng cho các kiểu có sẵn.
- Đa năng hóa các toán tử không thể có các tham số có giá trị mặc định.

Các toán tử có thể đa năng hóa:

`+ - * / % ^ ! = < > += -=
^= &= |= << >> <<= <= >= && || ++ --
() [] new delete & | ~ *= /= %= >>= == != , ->*`

Bài tập

■ Bài tập 1.1:

Viết chương trình cho phép thực hiện các thao tác trên kiểu struct **phân số**:

- Hàm toán tử Nhập, xuất phân số (>>, <<)
- Nghịch đảo, rút gọn phân số.
- Hàm toán tử +, -, *, /, <, >, ==, != giữa hai phân số.

Bài tập

■ Bài tập 1.2:

Viết chương trình cho phép thực hiện các thao tác trên kiểu struct **số phức**:

- Hàm toán tử Nhập, xuất số phức (>>, <<).
- Hàm tính module số phức.
- Hàm toán tử +, -, *, /, <, >, ==, != giữa hai số phức.

Bài tập

■ Bài tập 1.3:

Viết chương trình cho phép thực hiện các thao tác trên kiểu struct **đơn thức**:

- Hàm toán tử nhập, xuất đơn thức.
- Tính giá trị, đạo hàm, nguyên hàm đơn thức.
- Hàm toán tử +, -, *, /, <, >, ==, != giữa hai đơn thức cùng bậc.

Bài tập

■ Bài tập 1.4:

Viết chương trình cho phép thực hiện các thao tác trên kiểu struct Date gồm ngày, tháng, năm:

- Hàm toán tử nhập, xuất 01 ngày Date (>>, <<)
- Tính xác định thứ trong tuần.
- Hàm toán tử ++, -- để tăng, giảm 01 ngày; toán tử <, >, ==, != giữa hai ngày.

Bài tập

■ Bài tập 1.5:

Viết chương trình cho phép thực hiện các thao tác trên kiểu **mảng**:

- Hàm nhập, xuất mảng.
- Lấy kích thước mảng.
- Lấy phần tử tại vị trí nào đó.
- Sắp xếp tăng, giảm (theo các phương pháp sắp xếp: Chọn trực tiếp, Chèn trực tiếp, Nối bọt BubbleSort, QuickSort, HeapSort, ShellSort, RadixSort)
- Tìm phần tử nào đó trong mảng (tuần tự, nhị phân, nội suy).

Bài tập

■ Bài tập 1.6:

Thông tin một học sinh bao gồm:

- Họ tên.
- Điểm văn, toán.

Viết chương trình cho phép thực hiện các thao tác trên kiểu struct **học sinh** để quản lý danh sách học sinh.

- Nhập, xuất thông tin học sinh.
- Tính điểm trung bình của mỗi học sinh.
- Xếp loại theo tiêu chí:
 - Giỏi (≥ 8.0), Khá (≥ 7.0).
 - Trung bình (≥ 5.0), Yếu (< 5).