# Từ C đến C++

Lập trình hướng đối tượng

## Tài liệu đọc

- Eckel, Bruce. *Thinking in C++, 2nd Ed. Volume 1*.
  - Chapter 3: The C in C++
  - Chapter 8: Constants
    - Up to p. 352: Classes
  - Chapter 10: Name Control
    - Up to p. 423: Static members in C++
  - Chapter 13: Dynamic Object Creation
    - Up to p. 566: Overloading new & delete
  - Chapter 11: References and the Copy-Constructor
    - Up to p. 452: References in Functions

# Khác biệt đối với C

- Các khác biệt đối với C (ngoài các đặc điểm hướng đối tượng)
  - Chú thích
  - Các kiểu dữ liệu
  - Kiểm tra kiểu, đổi kiểu
  - Cảnh báo của trình biên dịch
  - Phạm vi và khai báo
  - Không gian tên
  - Hằng
  - Quản lý bộ nhớ
  - Tham chiếu



# Chú thích

- Bên cạnh chú thích kiểu C (nhiều dòng), C++ cho phép kiểu chú thích dòng đơn

C

```
/* This is a variable */
int x;
/* This is the variable
 * being given a value */
x = 5;
```

C++

```
// This is a variable
int x;
// This is the variable
// being given a value
x = 5;
```

# Chú thích

- C++ cho phép kiểu chú thích /* */ bao ngoài các chú thích dòng đơn.

C

```
/*
/* This is a variable */
int x;
/* This is the variable
 * being given a value */
x = 5;
*/
```

Chú thích có lỗi

C++

```
/*
// This is a variable
int x;
// This is the variable
// being given a value
x = 5;
*/
```



# Kiểu dữ liệu

- Kiểu giá trị Boolean: **`bool`**
  - Hai giá trị: **`true`** hoặc **`false`**
  - Các toán tử logic (**!**, **&&**, ...) lấy/tạo một giá trị **`bool`**
  - Các phép toán quan hệ (**==**, **<**, ...) tạo một giá trị **`bool`**
  - Các lệnh điều kiệu (**`if`**, **`while`**, ...) đòi hỏi một giá trị **`bool`**
- Để tương thích ngược với C, C++ ngầm chuyển từ **`int`** sang **`bool`** khi cần
  - Giá trị `0` → **`false`**
  - Giá trị khác `0` → **`true`**

# Kiểm tra kiểu dữ liệu

- C++ kiểm soát kiểu dữ liệu chặt chẽ hơn C
- C++ đòi hỏi hàm phải được khai báo trước khi sử dụng (mọi lời gọi hàm được kiểm tra khi biên dịch)
- C++ không cho phép gán giá trị nguyên cho các biến kiểu **enum**

```
enum Temperature {hot, cold};
enum Temperature t = 1; // Error in C++
```

- C++ không cho phép các con trỏ không kiểu (**void***) sử dụng trực tiếp tại bên phải lệnh gán hoặc một lệnh khởi tạo

```
void * vp;
int * ip = vp; // Error: Invalid conversion
```



# Đổi và ép kiểu dữ liệu

- C++ cho phép người dùng đổi kiểu dữ liệu một cách khá rộng rãi
- Trình biên dịch tự động thực hiện nhiều chuyển đổi dễ thấy:
  - Gán một giá trị thuộc kiểu số học này cho một biến thuộc kiểu khác
  - Các kiểu số học khác nhau cùng có trong các biểu thức
  - Truyền đối số cho các hàm
- Nếu hiểu rõ khi nào các chuyển đổi này xảy ra và trình biên dịch đang làm gì,ta có thể giải thích được các kết quả không mong đợi

# Đổi và ép kiểu dữ liệu

- Tự động chuyển đổi từ các đối tượng nhỏ thành các đối tượng lớn thì không có vấn đề gì, chiều ngược lại có thể có vấn đề
  - **short** → **long** (~16 bits → ~32 bits) không có vấn đề
  - **long** → **short** (~32 bits → ~16 bits) có thể mất dữ liệu
  - Khi chuyển từ các kiểu chấm động sang các kiểu nguyên có thể làm giảm độ chính xác của dữ liệu
- Trình biên dịch sẽ sinh cảnh báo (warning) đối với các chuyển đổi tự động có thể gây mất dữ liệu.



# Đổi và ép kiểu dữ liệu

- C++ cho phép người dùng ép kiểu một cách tường minh bằng nhiều cách
  - Ép kiểu kiểu C: `myInt = (int) myFloat;`
  - Ép kiểu kiểu hàm C++: `myInt = int(myFloat);`
- Để hạn chế ép kiểu quá mức và loại trừ các lỗi do ép kiểu, C++ cung cấp một cách mới sử dụng 4 loại ép kiểu tường minh
  - `static_cast`
  - `const_cast`
  - `reinterpret_cast`
  - `dynamic_cast`
- Cú pháp `myInt = static_cast<int>(myFloat)`

# Phạm vi và các Khai báo

- Trong C, các biến phải được định nghĩa tại đầu file hoặc tại bắt đầu của một khối **{…}**
- C++ cho phép khai báo sau và phạm vi của các biến được giới hạn chính xác hơn
  - Các khai báo có thể đặt tại các câu lệnh lặp for và các câu lệnh điều kiện
  - Phạm vi giới hạn bên trong vòng lặp hoặc khối điều kiện
- C++ còn bổ sung hai phạm vi mới:
  - Phạm vi không gian tên - *Namespace scope*
  - Phạm vi lớp - *Class scope*



# Namespace - Không gian tên

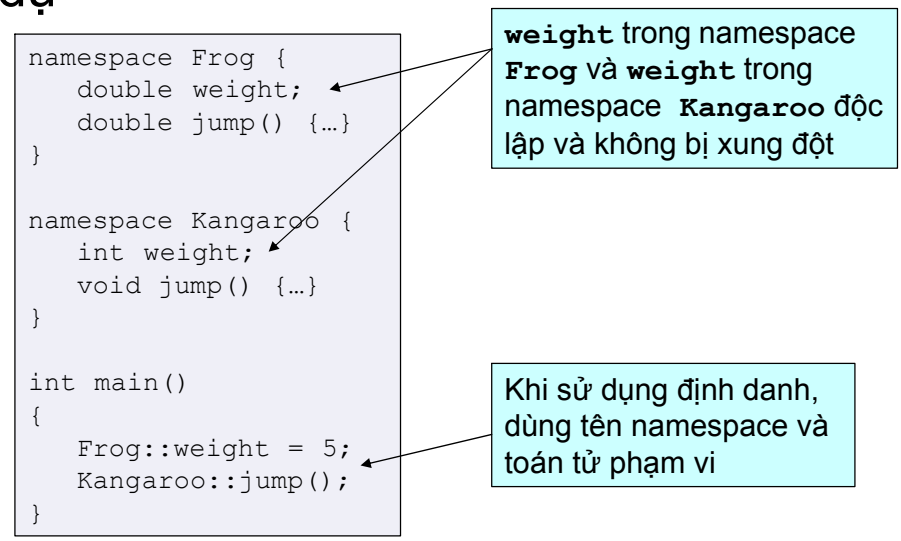
- Không gian tên được bổ sung vào C++ để biểu diễn cấu trúc logic và cung cấp khả năng quản lý phạm vi tốt hơn
- Không gian tên cung cấp một cơ chế tường minh để tạo các vùng khai báo
- Các tên khai báo trong một không gian tên
  - không xung đột với các tên được khai báo trong các không gian tên khác
  - Tránh xung đột tên biến, tên hàm
  - Nghiễm nhiên có thể được liên kết ra ngoài (external linkage)

```
namespace Frog {
   double weight;
   double jump() {…}
}

namespace Kangaroo {
   int weight;
   void jump() {…}
}
```

# Namespace

- Ví dụ






# Namespace

- C++ cung cấp hai cơ chế để đơn giản hóa việc sử dụng các namespaces: các khai báo using và các định hướng using.
- khai báo using (*using-declaration*) cho phép truy nhập một định danh cụ thể trong vùng khai báo tạm thời

```
using <namespace>::<name>;
```

  - Từ đây, ta có thể sử dụng tên mà không cần mỗi lần đều phải chỉ rõ namespace chứa nó.

# Namespace

- Ví dụ sử dụng khai báo using

```
namespace Frog {
   double weight;
   double jump() {…}
}

int main()
{
   using Frog::weight;
   int weight; // Error (already declared locally)
   weight = 5; // Sets Frog::weight to 5
}
```

Khai báo using cho định danh **weight** trong namespace **Frog.**

Từ đây, **weight** được hiểu là **Frog::weight**



# Namespace

- Khai báo using dành cho 1 tên, định hướng using cho phép truy nhập mọi định danh trong namespace

```
using namespace <namespace>;
```

- Định hướng using thường được đặt tại mức toàn cục.

```
#include <iostream>
using namespace std;
…
```

# Quản lý bộ nhớ

- Cấp phát bộ nhớ động trong C trông rối rắm và dễ lỗi
  - `myObj* obj = (myObj*)malloc(sizeof(myObj));`
- Các nhà thiết kế C++ thấy rằng:
  - Một ngôn ngữ sử dụng class sẽ hay phải sử dụng bộ nhớ động.
  - Không có lý do gì để tách cấp phát bộ nhớ động ra khỏi việc khởi tạo đối tượng (hay tách thu hồi bộ nhớ động ra khỏi việc hủy đối tượng)
- **`malloc`** và **`free`** đã được thay bằng **`new`** và **`delete`**



# Quản lý bộ nhớ

- Lợi thế của **new** so với **malloc**:
  - Không cần chỉ ra lượng bộ nhớ cần cấp phát
  - Không cần đổi kiểu
  - Không cần dùng lệnh if để kiểm tra xem bộ nhớ đã hết chưa
  - Nếu bộ nhớ đang được cấp cho một đối tượng, hàm khởi tạo (constructor) của đối tượng sẽ được gọi tự động (tương tự, **delete** sẽ tự động gọi hàm hủy (destructor) của đối tượng)
- Ví dụ:

```
myObj* obj = new myObj;        //một đối tượng
delete obj;

myObj* obj = new myObj[10];       // mảng đối tượng
delete[] obj;
```

# Tham chiếu – Reference

- Tham chiếu tới một đối tượng là một biệt danh tới đối tượng đó
- Có thể coi mọi thao tác trên tham chiếu đều được thực hiện trên chính đối tượng nguồn.

```
int x = 5;
int& y = x;
cout << "x = " << x << " y = " << y << ".\n"; // x = 5 y = 5
x = x + 1;
cout << "x = " << x << " y = " << y << ".\n"; // x = 6 y = 6
y = y + 1;
cout << "x = " << x << " y = " << y << ".\n"; // x = 7 y = 7

int *p = &y;
*p = 9;
cout << "x = " << x << " y = " << y << ".\n"; // x = 9 y = 9
```



# Tham chiếu – Reference

- Tham chiếu có thể được dùng độc lập nhưng thường hay được dùng làm tham số cho hàm
- C truyền mọi đối số cho hàm bằng giá trị (truyền trị - call by value)
  - Khi cần, ta có thể truyền một con trỏ tới đối tượng (chính nó cũng được truyền bằng giá trị)
  - `void myFunction(myObj* obj) {…}`
- C++ cho phép các đối số hàm được truyền bằng tham chiếu (call by reference)
  - `void myFunction(myObj& obj) {…}`
  - **`myObj&`** có nghĩa "tham chiếu tới **`myObj`**"
  - đối với các đối số là các đối tượng lớn, truyền bằng tham chiếu đỡ tốn kém hơn truyền bằng giá trị (do chỉ truyền địa chỉ bộ nhớ)

# Tham chiếu – Reference

- Ví dụ: tham chiếu làm đối số cho hàm

```
int f(int &i) { ++i; return i; }
int main() {
   int j = 7;   cout << f(j) << endl; cout << j <<
  endl;
}
```

- Biến i là một biến địa phương của hàm **`f`**. **`i`** thuộc kiểu tham chiếu **`int`** và được tạo khi **`f`** được gọi.
- Trong lời gọi **`f(j)`**, **`i`** được tạo tương tự như trong lệnh **`int &i = j;`**
- Do đó trong hàm **`f`**, **`i`** sẽ là một tên khác của biến **`j`** và sẽ luôn như vậy trong suốt thời gian tồn tại của **`i`**



# Tham chiếu – Reference

- Tham chiếu *phải* được khởi tạo
  - `int& x; // Error`
- Giá trị của tham chiếu không được thay đổi sau khi đã khởi tạo
  - không thể "chiếu" lại một tham chiếu tới đối tượng khác
  - chú ý phân biệt giữa khởi tạo tham chiếu và gán trị cho tham chiếu
- Truy nhập tới tham chiếu chính là truy nhập tới đối tượng nguồn
  - áp dụng cho cả toán tử & và phép gán
    - **cout << "The address of x is: " << &x << ".\n";**
    - **cout << "The address of y is: " << &y << ".\n";**
    - Output:
    - The address of x is 0x0056dc13.
    - The address of y is 0x0056dc13.

# Tham chiếu – Reference

- Vậy tại sao dùng tham chiếu thay cho con trỏ?
- Tham chiếu sạch hơn, không dễ gây lỗi như con trỏ, đặc biệt khi dùng trong hàm
  - tham chiếu đảm bảo không chiếu tới null
- Sử dụng tham chiếu trong nguyên mẫu hàm giúp cho việc gọi hàm dễ hiểu hơn
  - không cần dùng toán tử địa chỉ

```
void func1(int *pi) { (*pi)++; }
void func2(int &ri) { ri++; }

int main() {
   int i = 1;

   func1(&i); // call using address of i
   func2(i);  // call using i
}
```



# Const

- Trong C, hằng được định nghĩa bằng định hướng tiền xử lý **`#define`**

```
#define PI 3.14
```

  - Biên dịch chậm hơn (trình tiền xử lý tìm và thay thế)
  - Trình debug không biết đến các tên hằng
  - Sử dụng **#define** không gắn được kiểu dữ liệu với giá trị hằng (v.d. '**15**' là **int** hay **float?**)
- Const của ANSI-C ít dùng hơn và có nghĩa hơi khác:
  - ANSI-C **const** không được trình biên dịch chấp nhận là hằng
    - Không dùng để khai báo mảng được
    - In C++, **const** values can be used when declaring arrays:

# Const

Hằng của ANSI-C và C++ có các quy tắc phạm vi khác nhau

- Các giá trị **`#define`** có phạm vi file (do trình tiền xử lý chỉ thực hiện tìm và thay thế tại file đó)
- **`const`** của ANSI-C được coi là các biến có giá trị không đổi và có phạm vi chương trình
  - Do đó ta không thể có các biến trùng tên tại hai file khác nhau
- Trong C++, các định danh **`const`** tuân theo các quy tắc phạm vi như các biến khác
  - Do đó, chúng có thể có phạm vi toàn cục, phạm vi không gian tên, hoặc phạm vi khối



# const và con trỏ

- Ba loại:

```
1.  const int * pi;     // con trỏ tới hằng
2.  int * const ri = &i; // hằng con trỏ
3.  const int * const ri = &i; //hằng con trỏ tới hằng
```

- Các khái niệm cần ghi nhớ:
- Nếu một đối tượng là hằng
  - Không thể sửa đối tượng đó
  - Chỉ có con trỏ tới hằng mới được dùng để trỏ tới hằng, con trỏ thường không dùng được
- Nếu **`PI`** được khai báo là con trỏ tới hằng:
  - Có thể thay đổi **`PI`**, nhưng ***`PI`** không thể bị thay đổi.
  - **`PI`** có thể trỏ đến hằng hoặc biến thường

# const và con trỏ

- Khi sử dụng một con trỏ, có hai đối tượng có liên quan: chính con trỏ đó và đối tượng nó trỏ tới
- Cú pháp cho con trỏ tới hằng và hằng con trỏ rất dễ nhầm lẫn
- Quy tắc: trong lệnh khai báo, từ khoá const *bên trái* dấu * có nghĩa đối tượng được trỏ tới là hằng, từ khoá const *bên phải* dấu * có nghĩa con trỏ là hằng
- Cách dễ hơn: đọc các khai báo từ phải sang trái
  - `char c = 'Y';` // c là char
  - `char *const cpc = &c;` //cpc là hằng con trỏ tới char
  - `const char *pcc;` // pcc là con trỏ tới hằng char
  - `const char *const cpcc = &c;` // cpcc là hằng con trỏ tới hằng char



# hằng tham chiếu làm tham số hàm

Có hai lý do để truyền tham biến, nhưng nếu hàm được truyền không cần sửa đối tượng được truyền thì ta nên khai báo tham biến là **`const`**. Có hai ích lợi:

1. Nếu trong hàm, ta lỡ sửa đổi tham số thì trình biên dịch sẽ bắt lỗi.
   - ngăn chặn được một số lỗi lập trình viên có thể phạm
2. Ta có thể truyền các đối số là hằng hoặc không phải hằng cho hàm có hằng tham biến
   - Ngược lại, đối với các hàm có tham biến không phải là hằng, ta không thể truyền hằng làm đối số.

# hằng tham chiếu làm tham số hàm

- Ví dụ

Lỗi: Lo là hằng nên không được sửa đổi

```
void non_constRef(LargeObj &Lo) { Lo.height +=10; }          // Fine

    void constRef(const LargeObj &Lo) { Lo.height +=10; }  // Error
    !!!
```

```
void non_constRef(LargeObj &Lo) { cout Lo.height; }

void constRef(const LargeObj &Lo) { cout Lo.height; }

int main {
   LargeObj dinosaur;   const LargeObj rocket;

   non_constRef(dinosaur);
   constRef(dinosaur);
   non_constRef(rocket);           // Error
   constRef(rocket);
}
```

Lỗi: rocket là hằng, nên không thể làm đối số không phải hằng



# const: Hàm thành viên

Đối với hàm thành viên không sửa dữ liệu của đối tượng chủ, ta nên khai báo hàm đó là hằng hàm

- Đối với các đối tượng được khai báo là hằng, C++ chỉ cho phép gọi các hàm thành viên là hằng mà không cho phép gọi các hàm thành viên không phải là hằng của đối tượng đó.

```
class Date
{
   int year, month, day;
public:
   int getDay() const { return day; }
   int getMonth() const { return month; }
   void addYear(int y)            // Non-const function
};
```

# Const - Tóm tắt

Nên khai báo hằng đối với:

- Các đối tượng mà ta không định sửa đổi

```
const double PI = 3.14;
const Date openDate(18,8,2003);
```

- Các tham số của hàm mà ta không định cho hàm đó sửa đổi

```
void printHeight(const LargeObj &LO)
   { cout << LO.height; }
```

- Các hàm thành viên không thay đổi đối tượng chủ

```
int Date::getDay() const { return day; }
```

Lưu ý: các yêu cầu trên áp dụng cho tất cả các bài tập, bài thi của môn học.