Chương 4: Các kỹ thuật xây dựng hàm, sử dụng biến, hằng trong LTHDT

Huỳnh Quyết Thắng
Cao Tuấn Dũng
Bộ môn CNPM

Các thành phần tĩnh (static)

- Việc khai báo dữ liệu ở phạm vi toàn cục (global) có thể không đảm bảo an toàn hoặc gây xung đột
- Để khắc phục điều này thì ta khai báo dữ liệu dưới dạng Static
- Từ khoá static:
  - Các dữ liệu static chiếm các địa chỉ cố định và chỉ được tạo ra một lần, những lần tham chiếu sau sử dụng lại các dữ liệu đã được tạo ra này
- Mang tính cục bộ về khả năng sử dụng: đây có thể coi là một kỹ thuật quản lý định danh-biến/hàm

Các thành phần tĩnh

- Các biến địa phương khai báo cục bộ trong hàm:
  - Trong trường hợp các biến địa phương không khai báo là biến static thì mỗi lần gọi hàm chương trình dịch lại đăng ký tạo ra biến mới
  - Khi chúng ta khai báo các biến địa phương là các biến static thì chương trình dịch sẽ chỉ khởi tạo duy nhất một lần (ở lần gọi đầu tiên) biến địa phương này và thông qua con trỏ stack ở những lần gọi sau chỉ tham chiếu tới biến đã tạo ra này để sử dụng lại chúng mà không tạo ra biến mới
- Tạo một lần/tham chiếu nhiều lần/lưu giá trị của lần tham chiếu trước

Biến địa phương static

Biến địa phương static:

```
void f()
{ static int x=0;
  x++;
}
```

Lần gọi 1: f() 0

Lần gọi 2: f() 1

Biến địa phương không static

```
void f()
{ int x=0;
  x++;
}
```

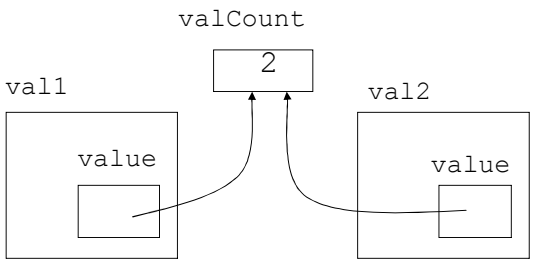
Lần gọi 1: f() 0

Lần gọi 2: f() 0

Thành phần dữ liệu tĩnh

- Tương tự giữa biến tĩnh và thành viên tĩnh
  - biến **static x** được khai báo trong hàm **f()**, một bản duy nhất tồn tại trong suốt quá trình chạy của chương trình.
  - dùng chung cho tất cả các lần chạy hàm **f()**,
  - bất kể hàm **f()** được gọi bao nhiêu lần
- Đối với class, **static** dùng để khai báo thành viên dữ liệu dùng chung cho mọi thể hiện của lớp.
  - một bản duy nhất tồn tại trong suốt quá trình chạy của chương trình,
  - dùng chung cho tất cả các thể hiện của lớp,
  - bất kể lớp đó có bao nhiêu thể hiện

Thành phần tĩnh: Chia sẻ giữa tất cả các đối tượng



Thành phần dữ liệu tĩnh

- **Định nghĩa lưu trữ cho các thành phần dữ liệu tĩnh của lớp**
  - Bắt buộc phải định nghĩa các thành phần dữ liệu tĩnh với từ khoá static
  - Khai báo đăng ký bộ nhớ để dành lưu trữ các dữ liệu thành phần tĩnh
  - Chỉ định nghĩa một lần
- Ví dụ nếu khai báo:

```
class A
 { static int i;
        .............
 };
 int A::i =1;
```


Đếm số đối tượng của một lớp (C++)

```
class MyClass {
public:
MyClass(); // Constructor
~MyClass(); // Destructor
void printCount(); // Output current value of count
private:
static int count; // static member to store
// number of instances of MyClass
};
```

Thành phần dữ liệu tĩnh Định nghĩa và khởi tạo

- Thành viên tĩnh được lưu trữ độc lập với các thể hiện của lớp, do đó, các thành viên tĩnh phải được định nghĩa:

  int MyClass::count;

- ta thường định nghĩa các thành viên tĩnh trong file chứa định nghĩa các phương thức
- nếu muốn khởi tạo giá trị cho thành viên tĩnh ta cho giá trị khởi tạo tại định nghĩa

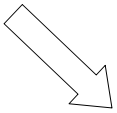
  int MyClass::count **= 0**;

Thành phần tĩnh: My Class

```
int MyClass::count = 0;
MyClass::MyClass() {
this->count++; // Increment the static count
}
MyClass::~MyClass() {
this->count--; // Decrement the static count
}
void MyClass::printCount() {
cout << "There are currently " << this->count
<< " instance(s) of MyClass.\n";
}
```

Sử dụng lớp MyClass

```
int main()
{
   MyClass* x = new MyClass;
   x->PrintCount();
   MyClass* y = new MyClass;
   x->PrintCount();
   y->PrintCount();
   delete x;
   y->PrintCount();
}
```



```
There are currently 1 instance(s) of MyClass.
There are currently 2 instance(s) of MyClass.
There are currently 2 instance(s) of MyClass.
There are currently 1 instance(s) of MyClass.
```


Đặc điểm của thành phần dữ liệu tĩnh

- Thuộc về lớp chứ không thuộc về bất cứ đối tượng nào, vì thế được sử dụng theo cú pháp: tên lớp :: tên biến
- Không thể sử dụng con trỏ this
- Chịu ảnh hưởng của các quy định về đóng gói dữ liệu: các từ khóa private, public, protected
- Các đối tượng của lớp (thông qua các hàm thành phần) có thể truy nhập và sử dụng các dữ liệu thành phần tĩnh
- Chỉ được cấp phát bộ nhớ cho các dữ liệu thành phần tĩnh một làn và nó là biến toàn cục trong phạm vi đang xét

Java: Định nghĩa bên trong lớp

static member: *one copy* per class; all instances share it

static method: can *only access* other *static members* (and local variables – if any)

`finalize()`: called when an instance is *garbage collected*

```
public class Employee {
    private String firstName;
    private String lastName;
    private static int count = 0;

    public Employee (String first, String last) {
        firstName = first;
        lastName = last;

        ++count;
    }

    protected void finalize() {
        --count;
    }

    public String getFirstName () { return firstName; }

    public String getLastName () { return lastName; }

    public static int getCount() { return count; }
}
```

Java: thành phần DL tĩnh

null reference: *narks* an object for *garbage collection*, but still `Employee.count == 2`

after garbage collection: `finalize()` has been called so `Employee.count == 1`

`e1.getCount() == e2.getCount() == Employee.getCount() == 1`

`e1.getCount() == e2.getCount() == Employee.getCount() == 2`

```
Employee e1 = new Employee("Susan", "Baker");

Employee e2 = new Employee("Bob", "Jones");

e1 = null;

System.gc();

e2 = null;
```

Phương thức tĩnh

- Từ khoá **static** còn được dùng cho các phương thức ➔ phương thức tĩnh
- Một phương thức tĩnh có thể được gọi một cách *độc lập* với mọi thể hiện của lớp
  - phương thức tĩnh không được dùng con trỏ (tham chiếu) this.
  - không thể sửa đổi các thành viên dữ liệu từ trong phương thức tĩnh.
  - có thể gọi phương thức tĩnh mà không cần tạo thể hiện nào của lớp - gọi thẳng bằng tên lớp

Phương thức tĩnh

- Hàm thành phần tĩnh chỉ có quyền truy nhập, xử lý dữ liệu của lớp (các dữ liệu thành phần tĩnh) mà không có quyền truy nhập và sử dụng các dữ liệu thành phần thông thường (tại sao?)
- Hàm thành phần tĩnh chịu ảnh hưởng của các quy định về đóng gói dữ liệu: private, public, protected

Ví dụ (C++)

```
class MyClass
{
public:
MyClass();
~MyClass();
static void printCount(); private:
static int count;
};
```

- Dùng tên lớp kèm theo toán tử phạm vi (::) để gọi phương thức tĩnh: MyClass::printCount();
- Hoặc có thể dùng đối tượng sẵn có để gọi phương thức tĩnh:

MyClass x;
x.printCount();

Phương thức tĩnh Java

```
class MyUtils {
    . . .
    //==================== mean
    public static double mean(int[] p) {
            int sum = 0;
      for (int i=0; i<p.length; i++) {
               sum += p[i];
            }
            return ((double)sum) / p.length;
    }
    . . .
}
…..
```

// Lời gọi PT tĩnh bên trong lớp
class double avgAtt = mean(attendance);
// Lời gọi Phương thức tĩnh bên ngoài lớp
double avgAtt = MyUtils.mean(attendance);

Vì sao dùng phương thức tĩnh

- Với các phương thức không tương tác với các "thể hiện" của lớp nên khai báo static
- Phương thức mean trong ví dụ trước có thể không khai báo static tuy nhiên muốn gọi nó phải thông qua một đối tượng

Tham chiếu và copy constructor (C++)

- Tham chiếu được xem như là một bí danh (alias) của một biến hay một đối tượng.
- Sau khi khởi tạo một tham chiếu và gán cho nó tên của một đối tượng khác, tham chiếu hoạt động như chính đối tượng đã gán cho nó. Mọi thay đổi trên biến tham chiếu là thay đổi chính biến được tham chiếu tới.

  Khai báo và khởi tạo tham chiếu :

```
<type>  variable;
<type>  &reference = variable;
```

Tham chiếu

- Một tham chiếu có thể là một biến, tham số hình thức của hàm hay dùng làm giá trị trả về của một hàm.
- Khi sử dụng tham chiếu phải tuân theo những điều kiện sau:
  - Một tham chiếu phải được khởi tạo giá trị ngay khi nó được khai báo.
  - Sau khi khởi tạo tham chiếu đã gắn cho một biến nào đó thì ta không thể thay đổi để gắn tham chiếu tới một biến khác.
  - Không thể có tham chiếu với giá trị Null.
- Đây là lý do người ta còn sử dụng con trỏ sau khi đã có tham chiếu.


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

int y;
int& r = y;
const int& q = 12;  // (1)
int x = 0;          // (2)
int& a = x;         // (3)
int main() {
  cout << "x = " << x << ", a = " << a << endl;
  a++;
  cout << "x = " << x << ", a = " << a << endl;
}
```

Sử dụng tham chiếu trong hàm

- Truyền tham số cho hàm bằng tham chiếu.
  - Việc sử dụng tham chiếu trong khai báo tham số hình thức của hàm sẽ yêu cầu trình dịch truyền địa chỉ của biến cho hàm và hàm sẽ thao tác trực tiếp trên các biến đó.
  - Việc khởi tạo các tham số hình thức là tham chiếu được thực hiện tự động trong mỗi lời gọi hàm
  - Như vậy tham số được truyền cho hàm bằng tham chiếu phải là biến (trừ trường hợp có từ khoá Cosnt đứng trước khai báo tham số hình thức).

Hoán đổi nội dung hai biến (C++)

```
void  swap (int &x,int &y)
{   int  temp=x ;
    x=y;
    y=temp;
}
void main()
{
  int  a=1; b=2;
  swap (a,b);
}
```

Giá trị trả về của hàm là tham chiếu.

- *Trong trường hợp này định nghĩa hàm có dạng :*

```
<type>  & function(…)  {
        …   // thân hàm
        return <external variable > ;
}
```

- Biểu thức được trả lại trong câu lệnh return phải là tên của một biến xác định từ bên ngoài hàm.

Tham chiếu (Java)

- Trong Java, mọi đối tượng đều được sử dụng thông qua tham chiếu. Khi ta khai báo một biến ➔ tạo ra một tham chiếu.
- Java truyền tham số theo tham trị
- Lý do loại bỏ cơ chế truyền tham số theo tham chiếu:
  - Tạo ra những đoạn mã thay đổi thành phần dữ liệu từ bên ngoài đối tượng.

Tham chiếu và thể hiện (Java)

```
Time t1 = new Time();
Time t2 = new Time(2);
Time t3 = new Time(21, 34);
Time t4 = new Time(12, 25, 42);
Time t5 = t4;

t4.setHour(13);
```

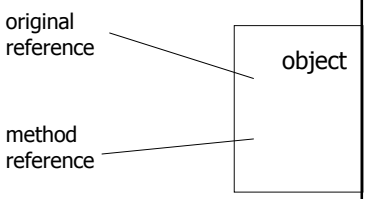
t1 = 00:00:00
t2 = 02:00:00
t3 = 21:34:00
t4 = 12:25:42
t5 = 12:25:42

References: **t4** and **t5** are *references* to the *same instance*

t4 = 13:25:42
t5 = 13:25:42


```
public void tricky(Point arg1, Point arg2)
{
  arg1.x = 100;
  arg1.y = 100;
  Point temp = arg1;
  arg1 = arg2;
  arg2 = temp;
}
public static void main(String [] args)
{
  Point pnt1 = new Point(0,0);
  Point pnt2 = new Point(0,0);
  System.out.println("X: " + pnt1.x + " Y: " +pnt1.y);
  System.out.println("X: " + pnt2.x + " Y: " +pnt2.y);
  System.out.println(" ");
  tricky(pnt1,pnt2);
  System.out.println("X: " + pnt1.x + " Y:" + pnt1.y);
  System.out.println("X: " + pnt2.x + " Y: " +pnt2.y);
}
```

X: 0 Y: 0
X: 0 Y: 0
X: 100 Y: 100
X: 0 Y: 0



Hàm thiết lập sao chép (copy constructor)

- Trong C++ ta có thể khai báo một biến và gán cho nó giá trị của một biến cùng kiểu đã khai báo trước đó, hoặc có thể khai báo một đối tượng và gán cho nó nội dung của một đối tượng cùng lớp đã có sẵn.

  Ví dụ: int p; int x = p;
- Khi một đối tượng được khai báo thì một hàm thiết lập tương ứng của lớp sẽ được gọi. Hàm thiết lập được gọi khi khai báo và khởi tạo nội dung một đối tượng thông qua một đối tượng khác gọi là hàm thiết lập sao chép.

Hàm thiết lập sao chép

- Nhiệm vụ của hàm thiết lập sao chép là tạo đối tượng và sao chép nội dung từ một đối tượng đã có sang đối tượng vừa được tạo ra.
- Dạng khai báo của hàm thiết lập là :

  <Name> (<type> &) ;

  hoặc <Name> (cosnt <type> &) ;
- Từ khoá cosnt trong khai báo tham số hình thức nhằm ngăn cấm mọi thay đổi nội dung của tham số truyền cho hàm.
- Ta cũng có thể tạo ra đối tượng mới giống đối tượng cũ một số đặc điểm, không hoàn toàn như phép gán. Đây là phương thức thiết lập có tham số là tham chiếu đến đối tượng thuộc chính lớp này.

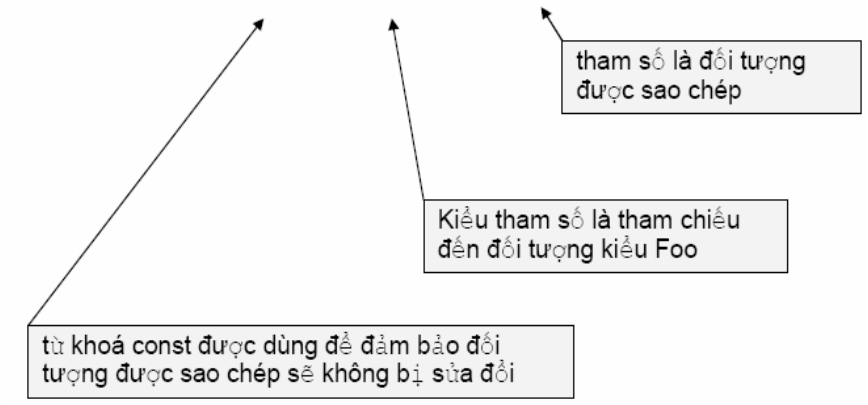
Hàm thiết lập sao chép

```
MyClass x(5);
MyClass y = x; hoặc MyClass y(x);
```

- C++ cung cấp sẵn một copy constructor, nó chỉ đơn giản copy từng thành viên dữ liệu từ đối tượng cũ sang đối tượng mới.
- Tuy nhiên, trong nhiều trường hợp, ta cần thực hiện các công việc khởi tạo khác trong copy constructor
  - Thí dụ: lấy giá trị cho một ID duy nhất từ đâu đó, hoặc thực hiện sao chép "sâu" (chẳng hạn khi một trong các thành viên là con trỏ giữ bộ nhớ cấp phát động)
- Trong trường hợp đó, ta có thể định nghĩa lại copy constructor

Khai báo điển hình

```
Foo(const Foo& existingFoo);
```



```
class Person {
    public:
    Person(const char *name0="", int
    age0=0);
        Person(const Person &p);
                void print();
    private:
        char name[30];
        int age;
};
```

Sử dụng tường minh hàm thiết lập sao chép:

```
Person person("Matti", 20);
Person twinBrother(person);
```

```
Person::Person(const
    Person &p) {
    strcpy(name, p.name);
    age = p.age;
}
```

Sử dụng không tường minh:

```
void f(Person p);
void main(void)  {
    Person person("Matti", 20);
    f(person);
}
```


Hàm thiết lập sao chép

- Chú ý vấn đề rò rỉ bộ nhớ khi viết code cho hàm tạo sao chép
- Trong Java, không có khái niệm copy constructor.

Hàm Inline

- Khi một định nghĩa hàm có chứa từ "inline" thì hàm đó sẽ không được biên dịch như một đoạn chương trình riêng có thể được gọi. Thay vào đó nó được chen thẳng vào những chỗ mà hàm này được gọi. Ví dụ:

```
inline int plusOne(int x) { return ++x; }
```

- Các hàm được định nghĩa trong thân của một lớp được tự động trở thành các hàm inline. Tuy nhiên bạn có thể làm cho một hàm của một lớp trở thành inline mà không cần định nghĩa nó trong thân lớp bằng cách đặt từ "inline" vào định nghĩa hàm.

Hàm Inline

- Khi bạn tạo một lời gọi tới một hàm inline, đầu tiên trình biên dịch phải kiểm tra chắc chắn rằng lời gọi đó được tạo ra một cách đúng đắn. Nếu tất cả các thông tin về kiểu hàm hợp với ngữ cảnh của lời gọi thì mã inline sẽ được thay thế trực tiếp vào chỗ gọi hàm.
- Từ đó ta thấy rằng một hàm inline phải được định nghĩa trước khi nó được sử dụng

Hàm Inline

- Nếu hàm inline chứa các lệnh điều khiển chương trình phức tạp ví dụ như các cấu trúc lặp, rẽ nhánh thì copiler sẽ bỏ qua tính inline của hàm. Ta chỉ nên dùng hàm inline để chứa các lệnh gán, biểu thức và lệnh gọi hàm đơn giản.
- Compiler có thể bỏ qua từ khóa inline nếu như nó thấy cần thiết. Ví dụ như trong chương trình của ta có quá nhiều lời gọi tới các hàm inline thì compiler sẽ bỏ qua tính inline của hàm xì thiếu bộ nhớ hoặc nếu các hàm inline dài, và các hàm đệ quy thì không thể là inline.
- So sánh hàm Inline và hàm thường?


```
class CStr
{
  char *pData;
  int nLength;
     …
  public:
       …
  //implicit inline function
    char *get_Data(void) {return pData; }
    int getlength(void);
     …
};
  //explicit inline function
  inline void CStr::getlength(void) {
     return nLength;
  }
```

Hàm Inline

- Tổng kết về hàm inline:
  - Ưu điểm: việc sử dụng hàm inline có tác dụng tiết kiệm được thời gian không phải thực hiện các xử lý đầu vào khi gọi hàm như: đẩy đối số vào stack, tạo một lời gọi, sau đó khi trở về thì phải giải toả các tham số khỏi stack. Trong nhiều trường hợp mã của nó nhỏ hơn so với việc nếu nó được cấp phát trên ngăn xếp.
  - Nhược điểm: làm cho chương trình lớn hơn. Việc sử dụng nhiều lời gọi tới hàm inline và các hàm inline dài sẽ làm cho chương trình bị phình to.

Kỹ thuật chồng hàm trong LTHDT

- Ý tưởng của nguyên lý chồng hàm: cho phép đặt tên hàm trùng nhau để mô tả bản chất công việc, nhưng các đối số hoặc kiểu dữ liệu trả về từ hàm là khác nhau
- Căn cứ vào số lượng hoặc kiểu dữ liệu của các giá trị truyền cho đối số HĐH sẽ chọn ra hàm phù hợp nhất để thực hiện trong trường hợp chồng hàm.
- Nếu như **không chọn được** hoặc **chọn được >1 (hai hàm trở lên**) như vậy thì sẽ báo lỗi.

Kỹ thuật chồng hàm trong LTHDT

- Phân loại kỹ thuật chồng hàm:
  - Chồng hàm dựa trên các đối số: số lượng và kiểu dữ liệu
  - Ví dụ: void f (int);
    void f (int, float);
    void f ();
  - Đặc điểm: Chấp nhận ở tất cả các ngôn ngữ lập trình hướng đối tượng

Kỹ thuật chồng hàm trong LTHDT

- Phân loại kỹ thuật chồng hàm:
  - Chồng hàm dựa trên kiểu dữ liệu trả về từ hàm. Ví dụ:
    void f(int);
    int f(int);
    float f(int);
  - Đặc điểm: Khó thực hiện chương trình dich, không chấp nhận trong các ngôn ngữ lập trình hướng đối tượng như C++

Chồng hàm

- Các điểm cần lưu ý về chồng hàm:
  - Các hàm được xem xét là chồng hàm nếu như chúng phải có cùng phạm vi (cùng trong một lớp, hoặc một mô-đun)
  - Các hàm phải có cùng tên hàm
  - Chỉ nên sử dụng khi các hàm có cùng mục đích, chức năng
- Chồng hàm thường được gặp nhất khi xây dựng các hàm thiết lập cho lớp

Java: định nghĩa chồng phương thức

```
public class Ship4 {
public double x=0.0, y=0.0, speed=1.0, direction=0.0;
public String name;

public Ship4(double x, double y, double speed, double direction, String name)
    {
     this.x = x;
     this.y = y;
     this.speed = speed;
     this.direction = direction;
     this.name = name;
}
public Ship4(String name) {
    this.name = name;
}
private double degreesToRadians(double degrees) {
    return(degrees * Math.PI / 180.0);
}
...
```

Java: định nghĩa chồng phương thức

```
    public void move() {
      move(1);
    }
    public void move(int steps) {
        double angle = degreesToRadians(direction);
        x = x + (double)steps * speed * Math.cos(angle);
        y = y + (double)steps * speed * Math.sin(angle);
    }
    public void printLocation() {
        System.out.println(name + " is at ("
        + x + "," + y + ").");
    }
}
```


Sử dụng phương thức định nghĩa chồng

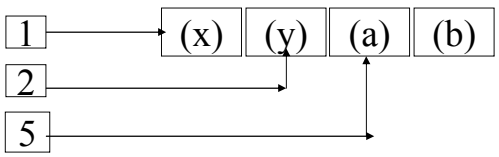
```
public class Test4 {
   public static void main(String[] args) {
        Ship4 s1 = new Ship4("Ship1");
        Ship4 s2 = new Ship4(0.0, 0.0, 2.0, 135.0, "Ship2");
        s1.move();
        s2.move(3);
        s1.printLocation();
        s2.printLocation();
   }
}
```

Hàm có đối số mặc định (C++)

- Khai báo hàm void hamf (int x, float y=1.0)
  - ý nghĩa: đối số x là đối số không có giá trị mặc định, đối số y là đối số có giá trị mặc định
  - có thể có hai dạng gọi hàm func:
    - func(10); đối số x nhận giá trị x=10 và đối số y nhận giá trị y=1.0 (giá trị mặc định)
    - func(10, 5.0); đối số x nhận giá trị x=10 và đối số y nhận giá trị y=5.0 (giá trị truyền vào)
- Hàm với các đối số mặc định cho phép they đổi dạng của hàm khi truyền các giá trị/biến cho các đối số

Hàm có đối số mặc định (C++)

- Nguyên tắc khi khai báo hàm với đối số mặc định:
  - Để đảm bảo chương trình dịch xác định đúng giá trị/biến truyền cho các đối số chúng ta cần phải giữ nguyên tắc:
    - Các đối số không có giá trị mặc định được xếp lên đầu danh sách của các đối số của hàm
    - Các đối số có giá trị mặc định được xếp xuống cuối danh sách của các đối số của hàm
- void f (int x, int y, int a=0, float b=1.0)
- Gọi hàm:
- f(1, 2, 5)



Hàm có đối số mặc định (C++)

```
int getSum(int, int=0, int=0);// OK
int getSum(int, int=0, int); //wrong!
```

- Khi một đối số bị bỏ qua trong lời gọi hàm, tất cả đối số sau nó cũng phải bị bỏ qua.

```
sum = getSum(num1, num2);  // OK
sum = getSum(num1, , num3);// wrong!
```

Ngôn ngữ Java không hỗ trợ đặc tính này

Khi nào sử dụng ?

- Sử dụng hàm với các đối số có giá trị mặc định:
  - Thông thường công việc mà hàm đó thực hiện không thay đổi bản chất hay giải thuật thực hiện. Các đối số nhận các giá trị mặc định hay truyền vào chỉ làm thay đổi kết quả mà không thay đổi ý nghĩa công việc.
  - Trường hợp thứ hai nên sử dụng hàm có đối số giá trị mặc định: công việc trong hàm mang tính chất mở rộng trong những trường hợp đối số nhận những giá trị truyền vào

Khi nào sử dụng ?

- Ví dụ (1): void f (int x, int y, int a=0, float b=1.0)
  - Công việc thực hiên trong f phụ thuộc vào 4 đối số x, y, a, b nhưng thông thường a=0 và b=1 tuy nhiên trong một số trường hợp a và b có thể nhận những giá trị khác
- Ví dụ (2): void f (int x, int y)
  - bình thường hàm f phụ thuộc vào hai giá trị đối số x,y. Bây giờ vì lý do phát triển mở rộng f phụ thuộc vào 3 đối số f(int x, int y, int a).
  - Làm thế nào có thể định nghĩa lại f mà trong chương trình những lời gọi cũ khi f có hai đối số không bị ảnh hưởng. Lời giải khai báo đối số a đối số với giá trị mặc định:
  - f (int x, int y, int a=0);

Hằng trong LTHDT

- Nguyên lý về hằng trong LTHDT thể hiện các đặc điểm và tư tưởng lập trình: những gì có thể thay đổi và những gì không được thay đổi và khi nào nên sử dụng chúng
- Trong các ngôn ngữ lập trình hướng đối tượng như có các từ khoá mang ý nghĩa khác nhau để sử dụng trong những trường hợp khai báo hằng số:
  - const (C++)
  - final (java)

Con trỏ hằng

- Con trỏ tới một hằng:
  - *const int * pi*
  - Trong trường hợp này chúng ta khai báo rằng pi là một con trỏ và giá trị mà pi trỏ tới là một giá trị không đổi, không được phép sử dụng pi để thay đổi giá trị này
  - *int a =10; int b=20;*
  - *pi=&a; //Lệnh đúng*
  - *pi=&b; //Lệnh đúng*
  - **pi = 100; //Lệnh sai*

Hằng con trỏ

- Hằng con trỏ
  - *int const *pi*
  - Trong trường hợp này chúng ta khai báo rằng pi là một con trỏ và là hằng tức là pi trỏ tới một địa chỉ không đổi, không được phép sử dụng pi để thay đổi địa chỉ mà pi trỏ tới
  - *int a =10; int b=20; int const *pi = &a;*
  - *pi=&b; //Lệnh sai*
- **pi = 100; //Lệnh đúng*

Sử dụng hằng trong hàm

- Truyền các đối số bằng từ khoá const
  - Khi khai báo hàm sử dụng các từ khóa const để khai báo các đối số trong hàm, chúng ta dã quy định luôn là trong thân hàm không thể sử dụng các lện thay đổi giá trị của các đối số này

```
void f(const int n)
 { .....
    n++; // Lỗi vì đã  thay đổi giá trị của n
   ....
}
```


Giá trị trả về từ hàm là hằng

- Tương tự như trường hợp truyền các đối số, chúng ta cũng có thể khai báo hàm có giá trị trả về là hằng, tức là bên trong thân hàm không được phép thay đổi biến nằm ở lệnh return
- Trên thực tế đối với các dữ liệu cơ bản giá trị trả về từ hàm không có ý nghĩa

```
#include <iostream>
const int f(int &i) {  i++;  return (++i); }
void main()
{  int d=1;  d=f(d);  cout<<d; }
```

Hằng trong lớp

- Thành phần dữ liệu là hằng
- Các hàm thành phần có khai báo const sau danh sách tham số:
  - Không được quyền thay đổi thành phần dữ liệu của đối tượng trong lớp. Thường dùng cho các phương thức Get.
  - Làm việc trên các hằng đối tượng

```
class Class {
public:
    Class( int a0, int b0 ) ;
    void mf1() const;
    void mf2();
private:
    int a, b;
};

void gf1 ( const Class &c ) ;
void gf2 ( Class &c ) ;
```

```
void main(void) {
Class c1(1, 2) ;
const Class c2(10, 20) ;
c1.mf1 ( ) ;   // OK
c1.mf2 ( ) ;   // OK
gf1 ( c1 ) ;   // OK
gf2 ( c1 ) ;   // OK
c2.mf1 ( ) ;   // OK
c2.mf2 ( ) ;   // Syntax error
gf1 ( c2 ) ;   // OK
gf2 ( c2 ) ;   // Syntax error
}
```