

CHƯƠNG 2

Đối Tượng Và Lớp

Nội Dung

- Đối Tượng Và Lớp
- Quan hệ giữa các lớp
- Cài đặt lớp
- Lớp
- Các thành phần của lớp
- Thuộc tính truy xuất
- Khởi động và dọn dẹp
- Phương thức thiết lập và huỷ bỏ

Nội Dung

- Một số đặc điểm khác của lớp
- Thiết lập và huỷ bỏ đối tượng
- Phương thức thiết lập bản sao
- Giao diện và chi tiết cài đặt
- Một số nguyên tắc xây dựng lớp

Đối Tượng và Lớp

- Ta định nghĩa một đối tượng là một "cái gì đó" có ý nghĩa cho vấn đề ta quan tâm. Đối tượng phục vụ hai mục đích: Giúp hiểu rõ thế giới thực và cung cấp cơ sở cho việc cài đặt trong máy.
- Mỗi đối tượng có một *nét nhận dạng* để phân biệt nó với các đối tượng khác. Nét nhận dạng mang ý nghĩa các đối tượng được phân biệt với nhau do sự tồn tại vốn có của chúng chứ không phải các tính chất mà chúng có.

Đối Tượng và Lớp

- Các đối tượng có các đặc tính tương tự nhau được gom chung lại thành lớp đối tượng. Ví dụ *Người* là một lớp đối tượng. Một lớp đối tượng được đặc trưng bằng các *thuộc tính*, và các *hoạt động* (hành vi).
- Một *thuộc tính* (attribute) là một sự trừu tượng hoá các giá trị dữ liệu cho mỗi đối tượng trong lớp. *Tên*, *Tuổi*, *Cân nặng* là các thuộc tính của *Người*.

Đối Tượng và Lớp – Phương Thức

- Một *thao tác* (operation) là một hàm hay một phép biến đổi có thể áp dụng vào hay áp dụng bởi các đối tượng trong lớp.
- Cùng một thao tác có thể được áp dụng cho nhiều lớp đối tượng khác nhau, một thao tác như vậy được gọi là có tính *đa hình* (polymorphism).

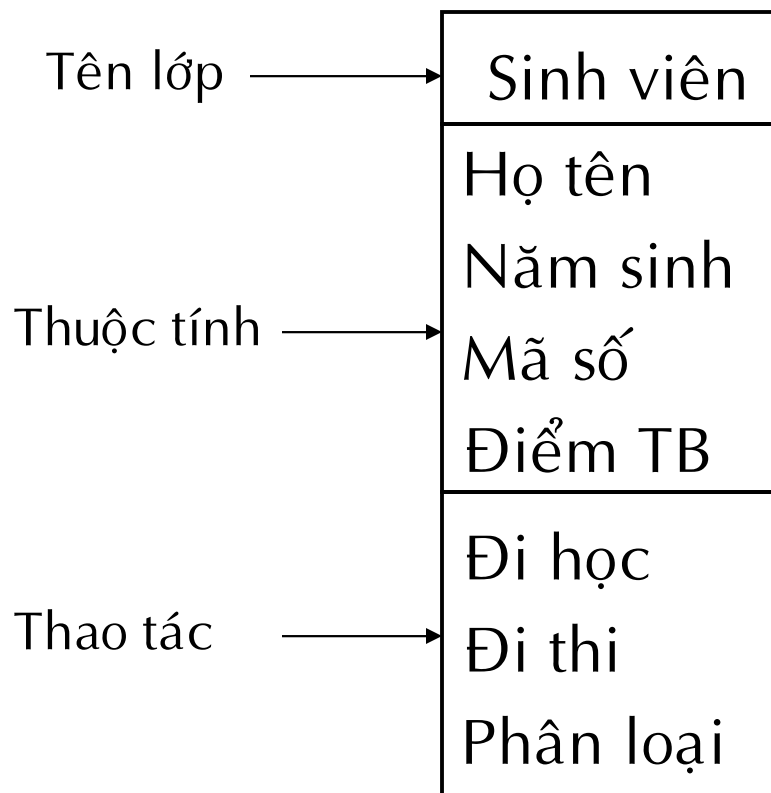
Đối Tượng và Lớp – Phương thức

- Mỗi thao tác trên mỗi lớp đối tượng cụ thể tương ứng với một cài đặt cụ thể khác nhau. Một cài đặt như vậy được gọi là một *phương thức* (method).
- Một đối tượng cụ thể thuộc một lớp được gọi là một *thể hiện* (instance) của lớp đó. John Smith, 25 tuổi, nặng 58kg, là một thể hiện của lớp người.

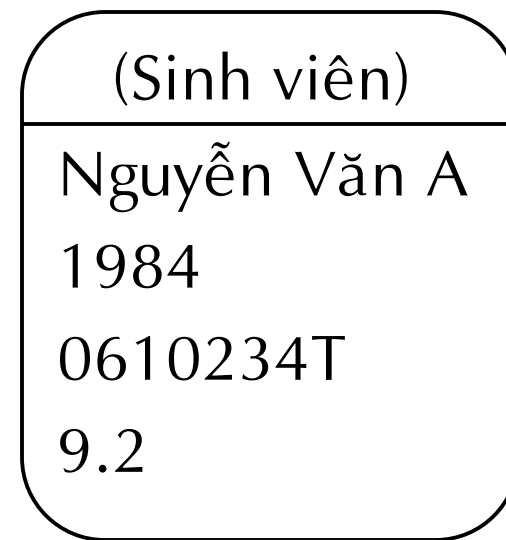
Sơ Đồ Đối Tượng

- Ta dùng *sơ đồ đối tượng* để mô tả các lớp đối tượng. Sơ đồ đối tượng bao gồm *sơ đồ lớp* và *sơ đồ thể hiện*.
- Sơ đồ lớp mô tả các lớp đối tượng trong hệ thống, một lớp đối tượng được diễn tả bằng một hình chữ nhật có 3 phần: phần đầu chỉ tên lớp, phần thứ hai mô tả các thuộc tính và phần thứ ba mô tả các thao tác của các đối tượng trong lớp đó.

Sơ đồ lớp và sơ đồ thể hiện



Sơ đồ lớp



Sơ đồ thể hiện

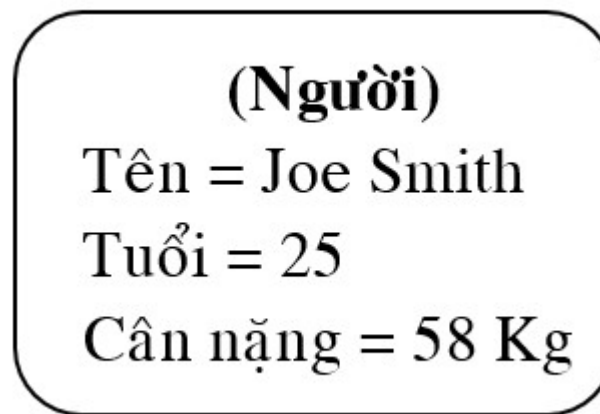
Sơ đồ lớp và sơ đồ thể hiện

Người
Tên
Tuổi
Cân nặng
Đổi việc làm
Đổi địa chỉ

Quốc gia
Tên
Dân số
Diện tích
Xuất khẩu
Nhập khẩu

Sơ đồ lớp

Sơ đồ lớp và sơ đồ thể hiện



Sơ đồ thể hiện

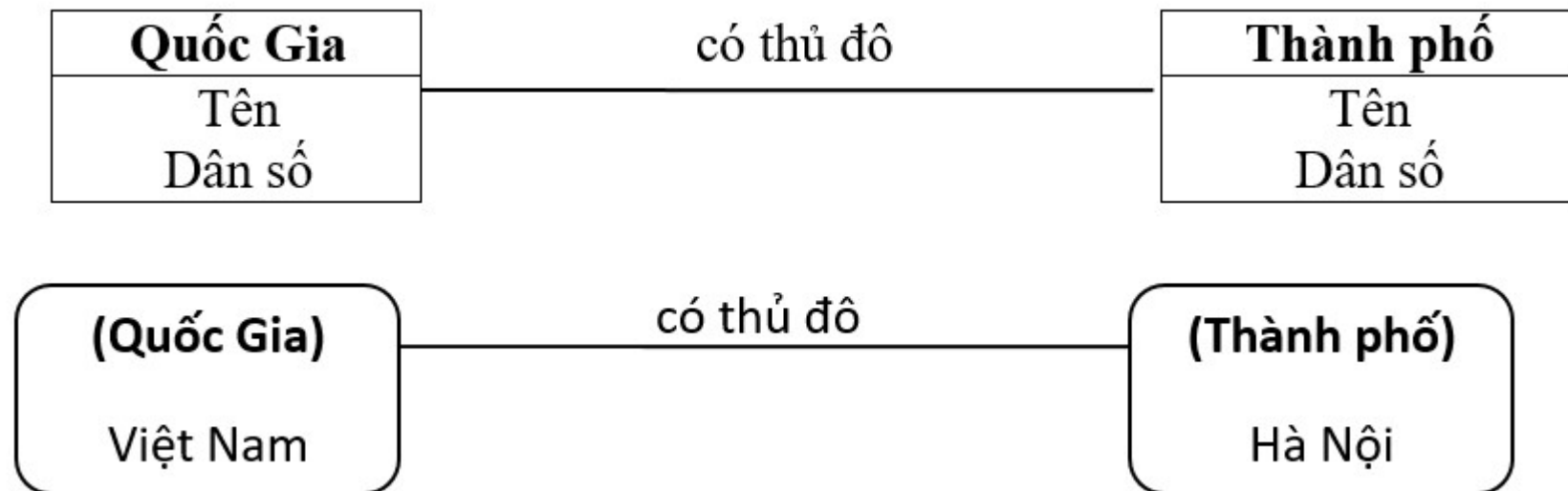
Quan hệ giữa các lớp

- Một mối *liên kết* (link) là một sự kết nối về mặt khái niệm giữa các đối tượng với nhau. Ví dụ Nguyễn Văn A *làm việc cho* Công ty X. Về mặt toán học, một mối liên kết là một bộ có thứ tự các đối tượng. Một mối liên kết là một thể hiện của một quan hệ.
- Một *quan hệ* (association) mô tả một nhóm các mối liên kết tương tự nhau về cấu trúc và về ngữ nghĩa. Ví dụ: Một người *làm việc cho* một Công ty.

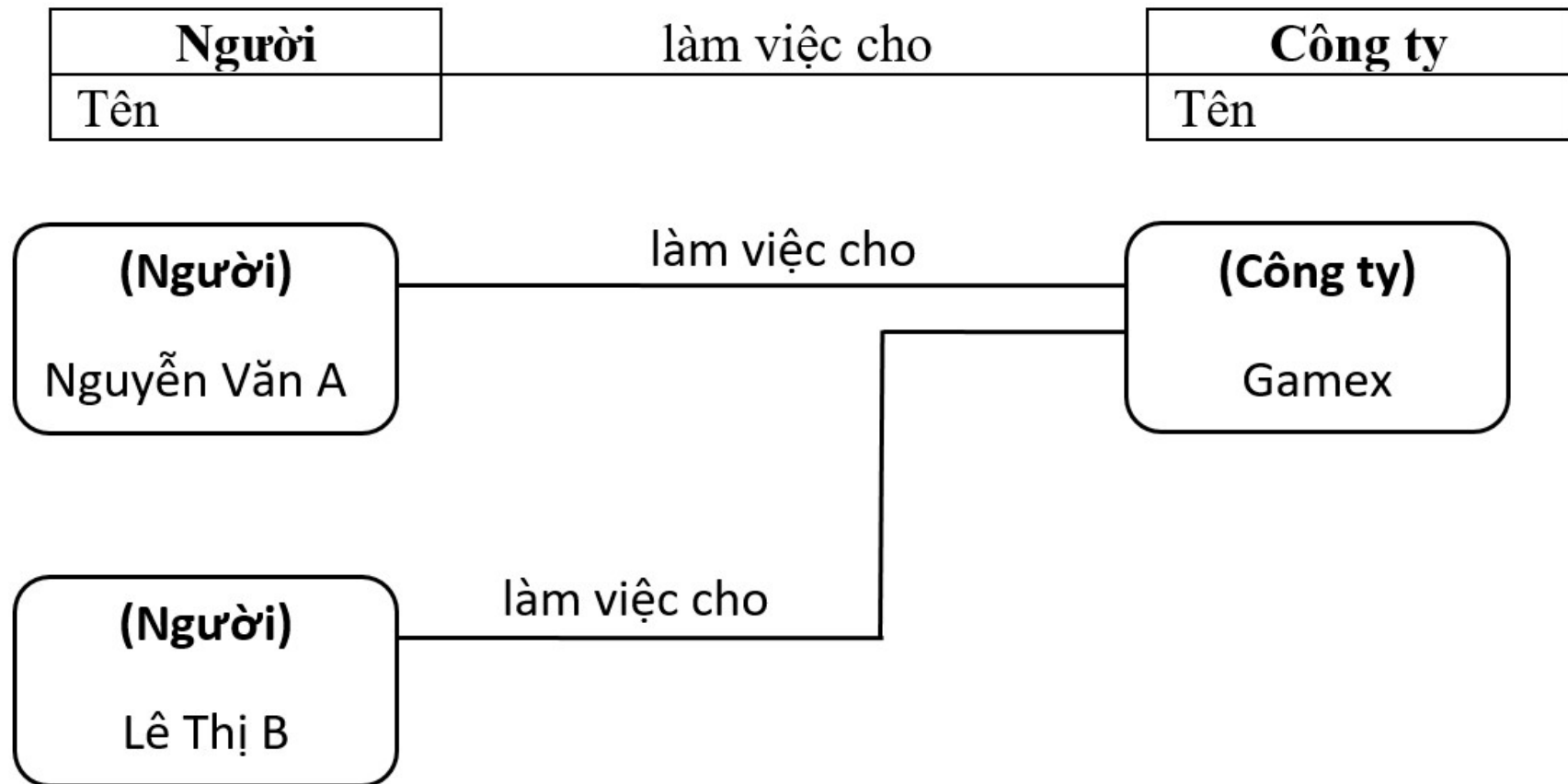
Quan hệ giữa các lớp

- Quan hệ có tính hai chiều. Một người làm việc cho một Công ty bao hàm ý nghĩa một công ty thuê một người.
- Quan hệ được mô hình bằng một *đường thẳng* nối hai lớp, trên đường thẳng có ghi nhãn là tên mối quan hệ. Một mối liên kết được vẽ bằng nối đường thẳng giữa hai thể hiện của các lớp đối tượng.

Quan hệ giữa các lớp



Quan hệ giữa các lớp

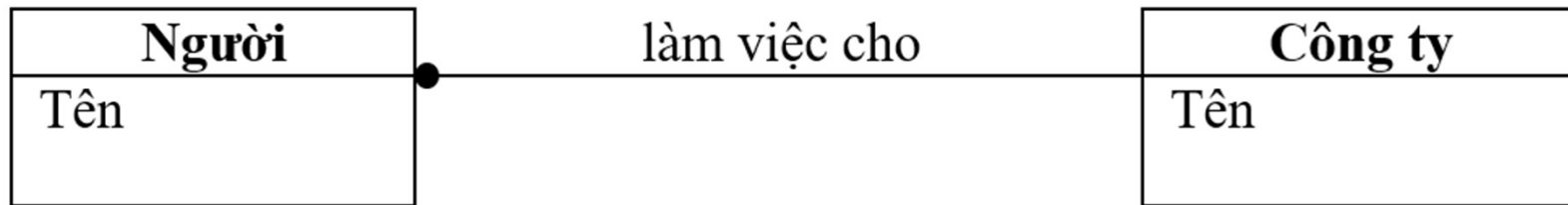


Quan hệ giữa các lớp – Bản số

- *Bản số*: Bản số qui định một thể hiện của một lớp có thể có quan hệ với bao nhiêu thể hiện của lớp khác. Dấu chấm tròn đặc để chỉ đối tượng ở bên phía nhiều của quan hệ, có thể kèm theo số.
 - ● : Từ 0 trở lên
 - ° : 0 hoặc 1
 - 2+ : Từ 2 trở lên
 - 3-5 : Từ 3 đến 5
 - 2,3,5 : 2 hoặc 3 hoặc 5

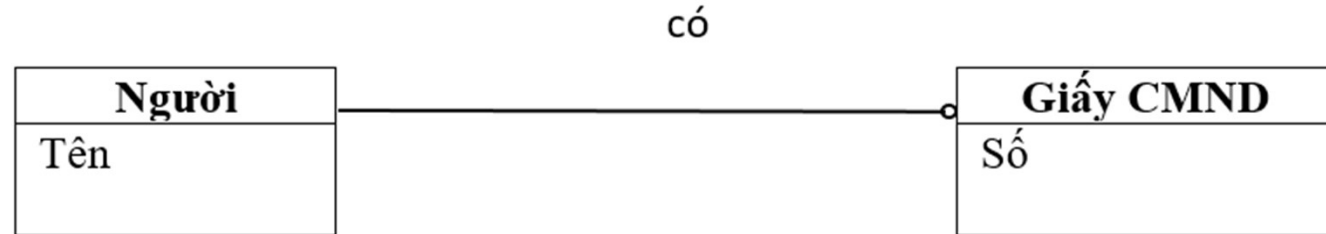
Bản số

- Quan hệ 1-nhiều:

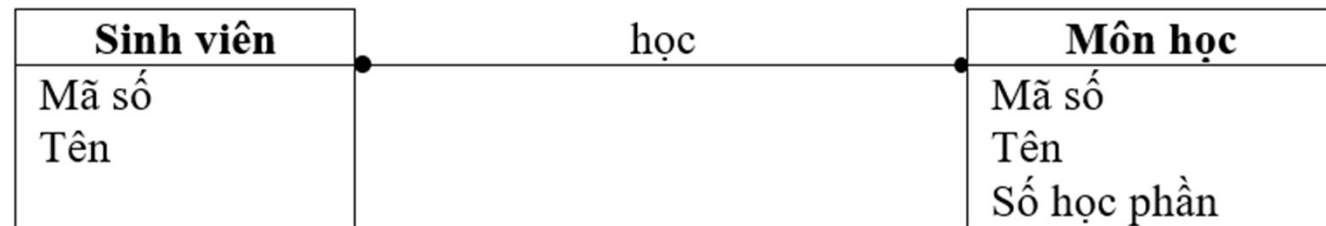


Bản số

Quan hệ 1-1(hoặc 0) :



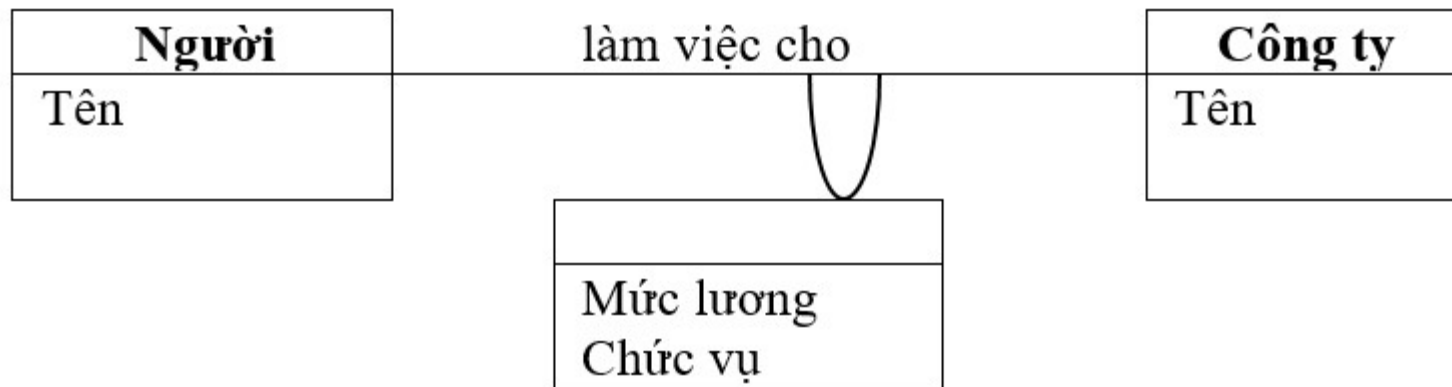
Quan hệ nhiều- nhiều



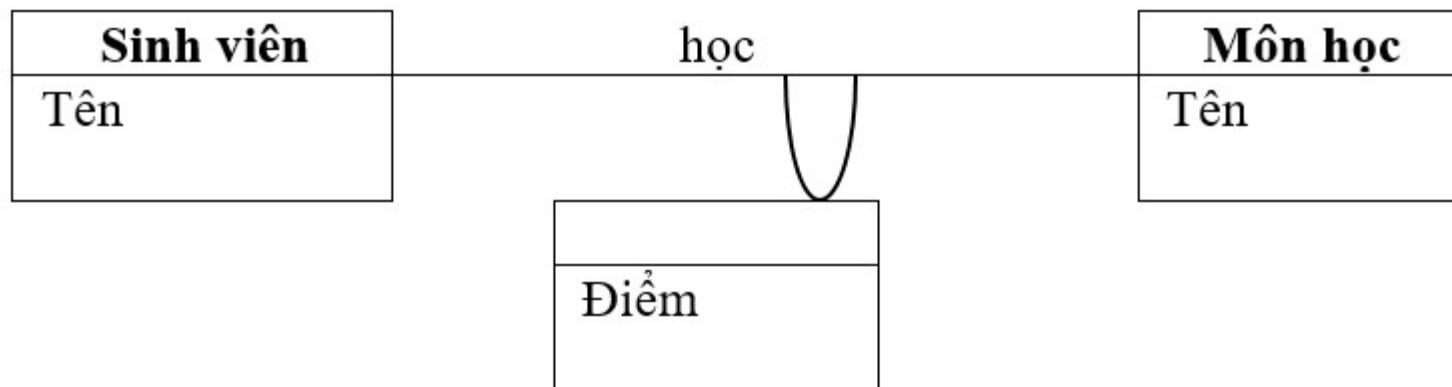
Quan hệ giữa các lớp

- *Thuộc tính liên kết*: Một thuộc tính liên kết (link attribute) là một tính chất của một mối quan hệ, thuộc tính liên kết được mô tả bằng một hình chữ nhật nối với quan hệ bằng một đường cong.

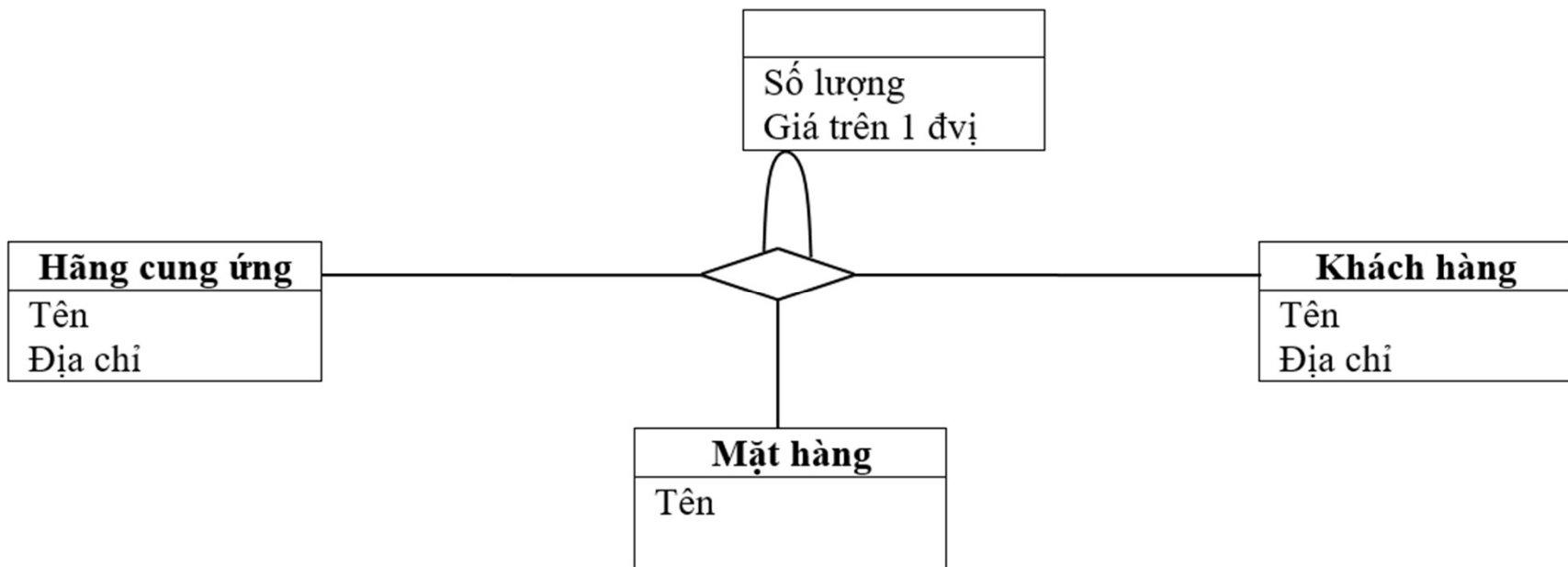
Thuộc tính liên kết



Quan hệ một người làm việc cho một công ty có mức lương và chức vụ



Thuộc tính liên kết



Quan hệ giữa các lớp

- *Tên vai trò*: Tên vai trò là danh hiệu được gắn vào một đầu của đường thẳng biểu diễn quan hệ để chỉ rõ vai trò của đối tượng tham gia với mỗi quan hệ.
- Thông thường tên vai trò không cần thiết khi mỗi quan hệ là giữa các lớp khác nhau. Trong trường hợp đó tên lớp đã chỉ rõ vai trò của đối tượng tham gia vào mỗi quan hệ.
- Tên vai trò có ý nghĩa khi mỗi quan hệ là giữa các đối tượng thuộc cùng một lớp.

Tên vai trò

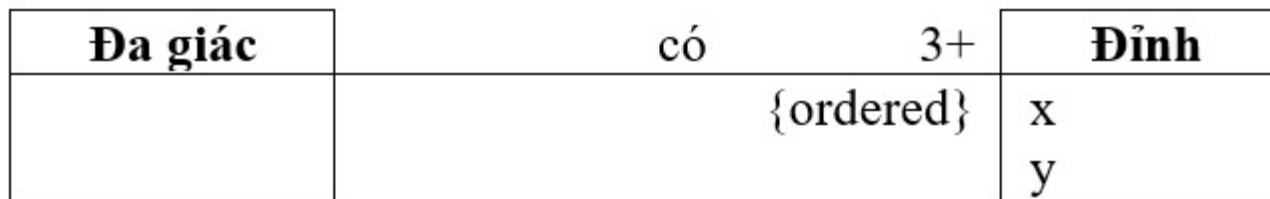
Người		thầy
Tên		dạy
Tuổi		
		trò

Quan hệ giữa các lớp

- *Sắp thứ tự*: Thông thường đối tượng ở phía nhiều của mỗi quan hệ không có thứ tự, và có thể quan điểm như một tập hợp.
- Đôi khi, các đối tượng có thứ tự rõ ràng, trong trường hợp này dùng từ khoá 'ordered' giữa hai dấu móc để biểu diễn tính thứ tự của các đối tượng.
- Sơ đồ sau chỉ mỗi quan hệ 'đa giác có đỉnh' với ràng buộc có tối thiểu 3 đỉnh và các đỉnh được sắp thứ tự.

Quan hệ giữa các lớp

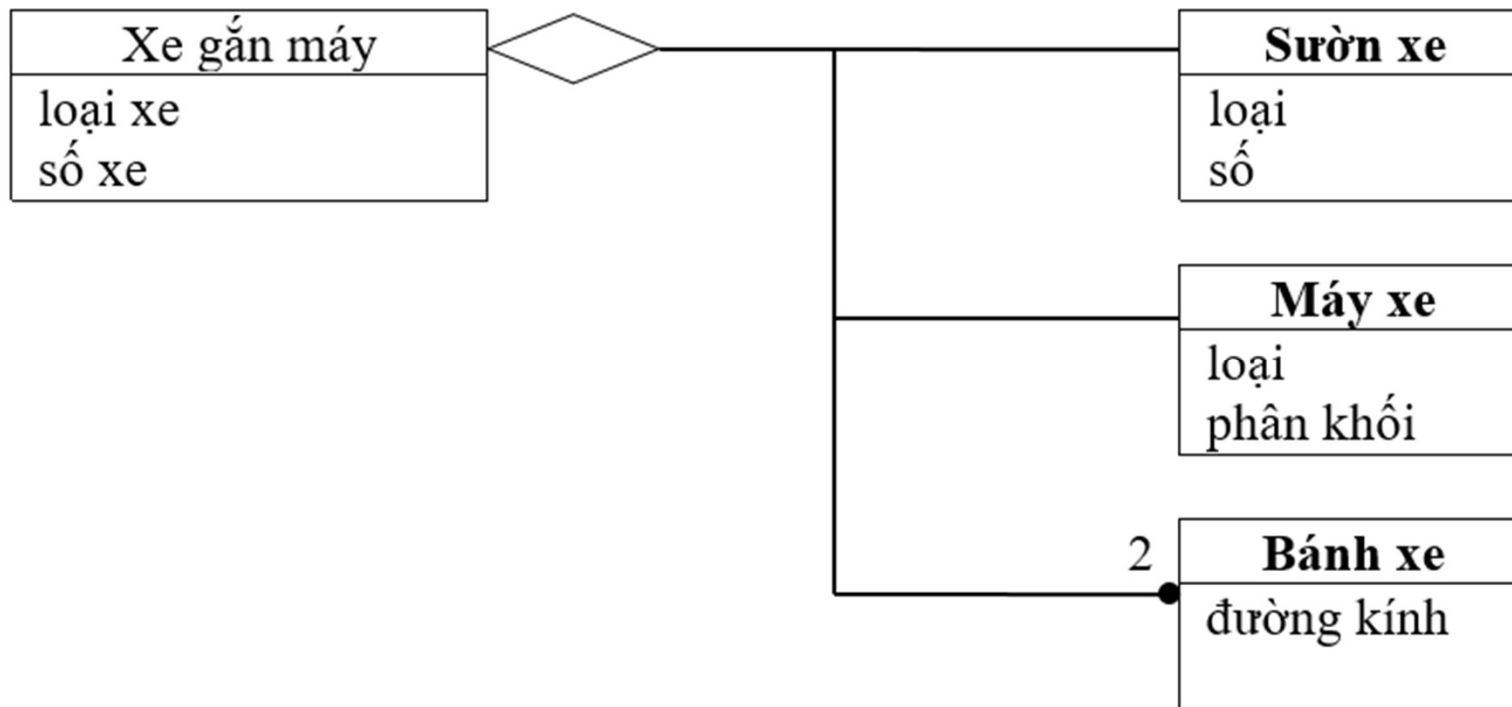
- Sơ đồ sau chỉ mối quan hệ ‘đa giác có đỉnh’ với ràng buộc có tối thiểu 3 đỉnh và các đỉnh được sắp thứ tự.



Quan hệ giữa các lớp

- *Quan hệ kết hợp*: Quan hệ kết hợp là một quan hệ đặc biệt, mô tả mỗi quan hệ "là sự kết hợp của" hoặc "là thành phần của". Mỗi quan hệ kết hợp được biểu diễn bằng một hình thoi nhỏ.

Quan hệ kết hợp



Cài đặt lớp trong một NNLT

- Lớp có thể được cài đặt trong một ngôn ngữ lập trình.
- Các ngôn ngữ lập trình hướng đối tượng hỗ trợ tốt cho việc cài đặt lớp nhờ có hỗ trợ cài đặt thuộc tính, phương thức và kế thừa.
- Ta sẽ minh họa cài đặt lớp bằng hai NNLT hướng đối tượng là C++ và/hoặc Objective C.

Cài đặt lớp trong một NNLT

- Nhắc lại: Một *kiểu dữ liệu* (trong một ngôn ngữ lập trình) là một biểu diễn cụ thể của một khái niệm có trong thực tế.
- Các ngôn ngữ lập trình cấp cao đều hỗ trợ các kiểu dữ liệu cơ bản, gọi là *kiểu có sẵn* (built-in data types), ví dụ trong C: int, long, float, double, char..., cho phép kiểm tra lúc biên dịch và phát sinh mã chương trình tối ưu. Các kiểu dữ liệu này cung cấp một giao diện tự nhiên độc lập với phần cài đặt.

Cài đặt lớp trong một NNLT

- Lớp là kiểu dữ liệu trừu tượng do *người sử dụng định nghĩa*, cho phép kết hợp dữ liệu, các phép toán, các hàm liên quan để tạo ra một đơn vị chương trình duy nhất. Các lớp này có đầy đủ ưu điểm và tiện lợi như các kiểu dữ liệu có sẵn.
- Lớp tách rời phần giao diện (chỉ liên quan với người sử dụng) và phần cài đặt lớp.

Cài đặt lớp trong một NNLT

- Lớp trong C/C++, Objective C có thể được cài đặt sử dụng từ khoá struct.
- Tuy nhiên, trong C++ ta dùng từ khoá class để cài đặt lớp sẽ tận dụng được các tính năng hướng đối tượng của C++.
- Objective C dùng cú pháp gần giống với Smalltalk để cài đặt lớp.

Cài đặt lớp - Ví dụ so sánh

- *Bài toán 1:* Xây dựng kiểu dữ liệu *phân số*, viết một ứng dụng cho phép thực hiện các phép toán số học trên hai phân số.
- *Bài toán 2:* Xây dựng kiểu dữ liệu *Stack* các số nguyên, viết hàm sử dụng Stack cho phép xuất một số nguyên trong hệ 16 (dưới dạng hexa). Viết ứng dụng minh họa.

Bài toán 1 - Cách tiếp cận cổ điển

```
struct PhanSo
```

```
{
```

```
    int tu, mau;
```

```
};
```

```
int uscln(int a, int b)
```

```
{
```

```
    if (b == 0) return abs(a);
```

```
    return uscln(b, a%b);
```

```
}
```

```
void PS_RutGon(PhanSo *a)
{
    int u = uscln(a->tu, a->mau);
    a->tu /= u;
    a->mau /= u;
    if (a->mau < 0)
    {
        a->tu = -a->tu;
        a->mau = -a->mau;
    }
}
```

```
void PS_Xuat(PhanSo a)
{
    printf("%d", a.tu);
    if (a.tu || a.mau != 1)
        printf("/%d", a.mau);
}
```

```
void PS_Nhap(PhanSo *a)
{
    scanf("%d%d", &a->tu, &a->mau);
}
```

```

PhanSo PS_Cong(PhanSo a, PhanSo b)
{
    PhanSo c;
    c.tu = a.tu*b.mau + a.mau*b.tu;
    c.mau = a.mau*b.mau;
    PS_RutGon(&c);
    return c;
}

```

```

PhanSo PS_Tru(PhanSo a, PhanSo b)
{
    PhanSo c;
    c.tu = a.tu*b.mau - a.mau*b.tu;
    c.mau = a.mau*b.mau;
    PS_RutGon(&c);
    return c;
}

```

```
PhanSo PS_Nhan(PhanSo a, PhanSo b)
{
    PhanSo c;
    c.tu = a.tu*b.tu;
    c.mau = a.mau*b.mau;
    PS_RutGon(&c);
    return c;
}
```

```
PhanSo PS_Chia(PhanSo a, PhanSo b)
{
    PhanSo c;
    c.tu = a.tu*b.mau;
    c.mau = a.mau*b.tu;
    PS_RutGon(&c);
    return c;
}
```

```

void main()
{
    PhanSo a = { 1,4 }, b = { 1,2 }, c;
    c = PS_Cong(a, b);
    PS_Xuat(c);
    puts("");
    c = PS_Tru(a, b);
    PS_Xuat(c);
    puts("");
    c = PS_Nhan(a, b);
    PS_Xuat(c);
    puts("");
    c = PS_Chia(a, b);
    PS_Xuat(c);
    puts("");
}

```

```
struct Fraction
{
    int numerator, denominator;
};

int gcd(int a, int b)
{
    if (b == 0) return abs(a);
    return gcd(b, a%b);
}
```

```
void fracReduce(Fraction *a)
{
    int u = gcd(a->numerator, a->denominator);
    a->numerator /= u;
    a->denominator /= u;
    if (a->denominator < 0)
    {
        a->numerator = -a->numerator;
        a->denominator = -a->denominator;
    }
}
```



```
void fracPrint(Fraction a)
{
    printf("%d", a.numerator);
    if (a.numerator || a.denominator != 1)
        printf("/%d", a.denominator);
}
```

```
void fracRead(Fraction *a)
{
    scanf("%d%d", &a->numerator, &a->denominator);
}
```

```

Fraction fracAdd(Fraction a, Fraction b)
{
    Fraction c;
    c.numerator = a.numerator*b.denominator +
a.denominator*b.numerator;
    c.denominator = a.denominator*b.denominator;
    fracReduce(&c);
    return c;
}

```

```

Fraction fracSub(Fraction a, Fraction b)
{
    Fraction c;
    c.numerator = a.numerator*b.denominator -
a.denominator*b.numerator;
    c.denominator = a.denominator*b.denominator;
    fracReduce(&c);
    return c;
}

```

```
Fraction fracMul(Fraction a, Fraction b)
{
    Fraction c;
    c.numerator = a.numerator*b.numerator;
    c.denominator = a.denominator*b.denominator;
    fracReduce(&c);
    return c;
}
```

```
Fraction fracDiv(Fraction a, Fraction b)
{
    Fraction c;
    c.numerator = a.numerator*b.denominator;
    c.denominator = a.denominator*b.numerator;
    fracReduce(&c);
    return c;
}
```

```

int main()
{
    Fraction a = { 1,4 }, b = { 1,2 }, c;
    c = fracAdd(a, b);
    fracPrint(c);
    puts("");
    c = fracSub(a, b);
    fracPrint(c);
    puts("");
    c = fracMul(a, b);
    fracPrint(c);
    puts("");
    c = fracDiv(a, b);
    fracPrint(c);
    puts("");
    return 0;
}

```

Kiểu Stack - Cách tiếp cận cổ điển

```
// stack.cpp
#pragma once

typedef int Item;

struct Stack
{
    Item *st, *top;
    int size;
};

void stackInit(Stack *ps, int sz);
void stackCleanUp(Stack *ps);
bool stackEmpty(Stack *ps);
bool stackFull(Stack *ps);
bool stackPush(Stack *ps, Item x);
bool stackPop(Stack *ps, Item *px);
```

Kiểu Stack - Cách tiếp cận cổ điển

```
// stack.cpp
#include "Stack.h"

void stackInit(Stack *ps, int sz)
{
    ps->top = ps->st = new Item[ps->size = sz];
}

void stackCleanUp(Stack *ps)
{
    delete[] ps->st;
}

bool stackEmpty(Stack *ps)
{
    return ps->top <= ps->st;
}
```

Kiểu Stack - Cách tiếp cận cổ điển

```
bool stackFull(Stack *ps)
{
    return ps->top - ps->st >= ps->size;
}
```

```
bool stackPush(Stack *ps, Item x)
{
    if (stackFull(ps)) return false;
    *ps->top++ = x;
    return true;
}
```

```
bool stackPop(Stack *ps, Item *px)
{
    if (stackEmpty(ps)) return false;
    *px = *--ps->top;
    return true;
}
```

Kiểu Stack - Cách tiếp cận cổ điển

```
#include <iostream>
using namespace std;
#include "stack.h"

void printHex(long n)
{
    static char hTab[] = "0123456789ABCDEF";
    Stack s;
    stackInit(&s, 8);
    int x;
    do {
        stackPush(&s, n % 16);
        n /= 16;
    } while (n);
    while (stackPop(&s, &x))
        cout << hTab[x];
    stackCleanup(&s);
}
```


Kiểu Stack - Cách tiếp cận cổ điển

```
int main()
{
    int a;
    cout << "Input a: ";
    cin >> a;
    printHex(a);
    cout << "\n";
    return 0;
}
```

Cài đặt lớp – Ví dụ so sánh

Nhận xét:

- Giải quyết được vấn đề.
- Khai báo cấu trúc dữ liệu nằm riêng, các hàm xử lý dữ liệu nằm riêng ở một nơi khác. Do đó *khó theo dõi quản lý khi hệ thống lớn*. Vì vậy khó bảo trì.
- Mọi thao tác đều có tham số đầu tiên là con trỏ đến đối tượng cần thao tác. Tư tưởng thể hiện ở đây là *hàm hay thủ tục đóng vai trò trọng tâm*. Đối tượng được gửi đến cho hàm xử lý.
- Trình tự sử dụng qua các bước: Khởi động, sử dụng thực sự, dọn dẹp.

Kiểu Stack - Cách tiếp cận dùng hàm thành phần

```
// stack.h
#pragma once
typedef int Item;

struct Stack
{
    Item *st, *top;
    int size;
    void init(int sz){ top = st = new Item[size = sz];}
    void cleanUp(){ delete[] st; }
    bool empty() { return top <= st; }
    bool full() { return top - st >= size; }
    bool push(Item x);
    bool pop(Item *px);
};
```

Kiểu Stack - Cách tiếp cận dùng hàm thành phần

```
#include "Stack.h"
```

```
bool Stack::push(Item x)
{
    if (full()) return false;
    *top++ = x;
    return true;
}
```

```
bool Stack::pop(Item *px)
{
    if (empty()) return false;
    *px = *--top;
    return true;
}
```

Kiểu Stack - Cách tiếp cận dùng hàm thành phần

```
// stack.cpp, other include(s) ...  
#include "stack.h"
```

```
void printHex(long n)  
{  
    static char hTab[] = "0123456789ABCDEF";  
    Stack s;  
    s.init(8);  
    int x;  
    do {  
        s.push(n % 16);  
        n /= 16;  
    } while (n);  
    while (s.pop(&x))  
        cout << hTab[x];  
    s.cleanUp();  
}
```

```
int main()
{
    int a;
    cout << "Nhap a: ";
    cin >> a;
    printHex(a);
    cout << "\n";
    return 0;
}
```

Cài đặt lớp - Ví dụ so sánh

- *Nhận xét:*
- Giải quyết được vấn đề.
- Dữ liệu và các hàm xử lý dữ liệu được gom vào một chỗ bên trong cấu trúc. Do đó dễ theo dõi quản lý, dễ bảo trì nâng cấp.
- Các thao tác đều bớt đi một tham số so với cách tiếp cận cổ điển. Vì vậy việc lập trình gọn hơn. Tư tưởng thể hiện ở đây là *đối tượng đóng vai trò trọng tâm*. Đối tượng thực hiện thao tác trên chính nó.
- Trình tự sử dụng qua các bước: Khởi động, sử dụng thực sự, dọn dẹp.

Hàm thành phần

- Phương thức trong C++ còn được gọi là hàm thành phần.
- Hàm được khai báo trong lớp. Hàm thành phần có thể được định nghĩa bên trong hoặc bên ngoài lớp.
- Hàm thành phần có nghi thức giao tiếp giống với các hàm bình thường khác: có tên, danh sách tham số, giá trị trả về.
- Gọi hàm thành phần bằng phép toán dấu chấm (.) hoặc dấu mũi tên (->).

```
Stack s, *ps = &s;  
s.init(8); ps->push(10);
```


Ví dụ thêm: Kiểu Phân Số - Cách tiếp cận dùng hàm thành phần

```
struct Fraction
{
    int numerator, denominator;
    void reduce();
    void print();
    void input();
    Fraction add(Fraction b);
    Fraction sub(Fraction b);
    Fraction mul(Fraction b);
    Fraction div(Fraction b);
};
```

```
int gcd(int a, int b)
{
    if (b == 0) return abs(a);
    return gcd(b, a%b);
}
```

Listing cont...

```
void Fraction::reduce()
{
    int u = gcd(numerator, denominator);
    numerator /= u;
    denominator /= u;
    if (denominator < 0)
    {
        numerator = -numerator;
        denominator = -denominator;
    }
}
```

```
void Fraction::print()
```

Listing cont...

```
{
```

```
    printf("%d", numerator);
```

```
    if (numerator || denominator != 1)
```

```
        printf("/%d", denominator);
```

```
}
```

```
void Fraction::input()
```

```
{
```

```
    scanf_s("%d%d", &numerator,  
            &denominator);
```

```
}
```

Fraction Fraction::add(Fraction b) Listing cont...

```
{  
    Fraction c;  
    c.numerator = numerator*b.denominator +  
    denominator*b.numerator;  
    c.denominator = denominator*b.denominator;  
    c.reduce();  
    return c;  
}
```

```
Fraction Fraction::sub(Fraction b)  
{  
    Fraction c;  
    c.numerator = numerator*b.denominator -  
    denominator*b.numerator;  
    c.denominator = denominator*b.denominator;  
    c.reduce();  
    return c;  
}
```

Fraction Fraction::mul(Fraction b) Listing cont...

```
{  
    Fraction c;  
    c.numerator = numerator*b.numerator;  
    c.denominator = denominator*b.denominator;  
    c.reduce();  
    return c;  
}
```

Fraction Fraction::div(Fraction b)

```
{  
    Fraction c;  
    c.numerator = numerator*b.denominator;  
    c.denominator = denominator*b.numerator;  
    c.reduce();  
    return c;  
}
```

```
int main()
```

```
{
```

```
    Fraction a = { 1,4 }, b = { 1,2 }, c;
```

```
    c = a.add(b);
```

```
    c.print();
```

```
    puts("");
```

```
    c = a.sub(b);
```

```
    c.print();
```

```
    puts("");
```

```
    c = a.mul(b);
```

```
    c.print();
```

```
    puts("");
```

```
    c = a.div(b);
```

```
    c.print();
```

```
    puts("");
```

```
    return 0;
```

```
}
```

Listing cont...

Cài đặt lớp - Ví dụ so sánh

- *Nhận xét: (Nhắc lại)*
- Giải quyết được vấn đề.
- Dữ liệu và các hàm xử lý dữ liệu được gom vào một chỗ bên trong cấu trúc. Do đó dễ theo dõi quản lý, dễ bảo trì nâng cấp.
- Các thao tác đều bớt đi một tham số so với cách tiếp cận cổ điển. Vì vậy việc lập trình gọn hơn. Tư tưởng thể hiện ở đây là *đối tượng đóng vai trò trọng tâm*. Đối tượng thực hiện thao tác trên chính nó.
- Trình tự sử dụng qua các bước: Khởi động, sử dụng thực sự, (dọn dẹp).

Lớp

- Trong cách tiếp cận dùng struct và hàm thành phần, người sử dụng có toàn quyền truy xuất, thay đổi các thành phần dữ liệu của đối tượng thuộc cấu trúc.

```
Stack s;  
s.init(10);  
s.size = 100; // Nguy hiểm  
for (int i = 0; i < 20; i++) s.push(i);
```

```
Fraction a;  
a.numerator = 2;  
a.denominator = 0; // Oh dear! 2/0!!!
```


Lớp

- Vì vậy, ta không có sự an toàn dữ liệu. Lớp là một phương tiện để khắc phục nhược điểm trên.
- Lớp có được bằng cách thay từ khoá struct bằng từ khoá class (C++, Java, C#).
- Trong lớp mọi thành phần mặc nhiên đều là *riêng tư* (private) nghĩa là từ bên ngoài không được phép truy xuất. Do đó có sự an toàn dữ liệu.

Từ khoá class thay cho struct

```
class Stack
{
    Item *st, *top;
    int size;
    void init(int sz){ top = st = new
    Item[size = sz];}
    void cleanUp(){ delete[] st; }
    bool empty() { return top <= st; }
    bool full() { return top - st >= size; }
    bool push(Item x);
    bool pop(Item *px);
};
```

Lớp

- Mọi cố gắng truy xuất các thành phần của lớp sẽ bị báo lỗi biên dịch.

```
s.size = 100; // Bao sai
```

```
a.denominator = 0; // Bao sai
```

- Nghĩa là lỗi luận lý được trình biên dịch hỗ trợ phát hiện, giúp giảm rất nhiều bug khó tìm.

Thuộc tính truy xuất

- Tuy nhiên lớp như trên trở thành vô dụng vì các hàm thành phần cũng trở thành riêng tư và không thể được.

```
Stack s;
```

```
s.init(8);    // Bao sai
```

```
s.push(14);   // Bao sai
```

```
Fraction a;
```

```
a.setNumerator(2);    // Bao sai
```

```
a.setDenominator(14); // Bao sai
```

- Thuộc tính truy xuất có thể giải quyết vấn đề trên.
- Thuộc tính *truy xuất* của một thành phần nào của lớp là phần chương trình có thể nhìn thấy, truy xuất thành phần đó.

Thuộc tính truy xuất

- Thuộc tính *private*: Chỉ có các phương thức của lớp được phép truy xuất.
- Thuộc tính *public*: Có thể được truy xuất từ bất cứ nơi nào trong chương trình nguồn từ sau khai báo lớp.
- Phần *public* tạo nên phần *giao diện* của lớp.
- Nguyên tắc chung: Các thuộc tính dữ liệu của lớp có thuộc tính *private*, các phương thức để người sử dụng dùng có thuộc tính *public*.

Thuộc tính truy xuất

- Trong C++, thuộc tính mặc nhiên của lớp *private*, thuộc tính của struct là *public*.
- Ta có thể thay đổi thuộc tính truy xuất bằng nhãn *private* hoặc *public*. Sự thay đổi có tác dụng từ sau nhãn.

Thuộc tính truy xuất: Ví dụ

```
class Stack
```

```
{
```

```
    Item *st, *top;  
    int size;
```

Các thành phần private không được truy xuất từ bên ngoài

```
public:
```

```
    void init(int sz) { st = top = new  
        Item[size = sz]; }
```

```
    void cleanUp() { if (st) delete[] st; }
```

```
    bool full()const { return (top - st >= size); }
```

```
    bool empty() const { return (top <= st); }
```

```
    bool push(Item x);
```

```
    bool pop(Item *px);
```

```
};
```

Thuộc tính truy xuất: Ví dụ

```
class Fraction
```

```
{
```

```
    int numerator, denominator;  
    void reduce();
```

Các thành phần private
không được truy xuất
từ bên ngoài

```
public:
```

```
    void print();
```

```
    void input();
```

```
    Fraction add(Fraction b);
```

```
    Fraction sub(Fraction b);
```

```
    Fraction mul(Fraction b);
```

```
    Fraction div(Fraction b);
```

```
    void setNumerator(int n) { numerator = n; }
```

```
    void setDenominator(int d) { denominator = d; }
```

```
};
```


Thuộc tính truy xuất

- Các thành phần private *không thể* được truy xuất từ bên ngoài nhưng các thao tác public thì được.

```
Stack s;  
s.size = 100;           // Error  
s.init(8);              // Ok  
s.push(14);             // Ok  
Fraction a;  
a.denominator = 0;      // Error  
a.setNumerator(2);      // Ok  
a.setDenominator(14);   // Ok
```

Thuộc tính truy xuất

- Phạm vi truy xuất được sử dụng đúng sẽ cho phép ta kết luận: Nhìn vào lớp *thấy được mọi thao tác* trên lớp.
- Người dùng bình thường có thể khai thác hết các chức năng của lớp thông qua phần giao diện (public) của lớp.
- Người dùng cao cấp có thể thay đổi chi tiết cài đặt, cải tiến giải thuật cho các phương thức.

Cài đặt lớp trong Objective C

- Cú pháp cài đặt lớp trong Objective C khác với C++, gần giống với Smalltalk.
- Lớp trong Objective C có hai phần tách biệt, phần giao diện (interface) và phần cài đặt (implementation).
- Trong Objective C, thuộc tính dữ liệu định nghĩa trong phần cài đặt mặc nhiên là private, các thao tác trong phần giao diện mặc nhiên là public.

Cài đặt lớp trong Objective C

```
@interface ClassName: ParentClassName
    propertyDeclarations;
    methodDeclarations;
@end
```

```
@implementation ClassName
{
    memberDeclarations;
}
methodDefinitions;
@end
```

```
// Fraction.h
```

```
#import <Foundation/Foundation.h>
```

```
@interface Fraction : NSObject  
+ (int)gcd: (int)a and: (int)b;  
- (void)setNumerator: (int)n;  
- (void)setDenominator: (int)d;  
- (void)set: (int)n over: (int)d;  
- (void)print;  
- (Fraction *)add: (Fraction *)b;  
- (Fraction *)sub: (Fraction *)b;  
- (Fraction *)mul: (Fraction *)b;  
- (Fraction *)div: (Fraction *)b;  
@end
```

```
// Fraction.m
```

```
#import "Fraction.h"
```

```
@implementation Fraction
```

```
{
```

```
    int numerator, denominator;
```

```
}
```

```
+(int)gcd: (int)a and: (int)b {
```

```
    if (b == 0)
```

```
        return a;
```

```
    return[Fraction gcd: b and: a % b];
```

```
}
```

```
-(void)setNumerator: (int)n {  
    numerator = n;  
}  
  
-(void)setDenominator: (int)d {  
    denominator = d;  
}  
  
-(void)set: (int)n over: (int)d {  
    int u = [Fraction gcd: n and: d];  
    numerator = n / u;  
    denominator = d / u;  
    if (denominator < 0) {  
        numerator = -numerator;  
        denominator = -denominator;  
    }  
}
```

```
- (int) numerator {  
    return numerator;  
}  
  
- (int) denominator {  
    return denominator;  
}  
  
- (void) print {  
    if (numerator != 0 || denominator != 1)  
        NSLog(@"%i / %i", numerator, denominator);  
    else  
        NSLog(@"%i", numerator);  
}
```



```
-(Fraction *)add: (Fraction *)b {  
    Fraction *c = [Fraction alloc];  
    c = [c init];  
    [c set: numerator * b.denominator + denominator *  
    b.numerator over: denominator * b.denominator];  
    return c;  
}  
  
-(Fraction *)sub: (Fraction *)b {  
    Fraction *c = [Fraction alloc];  
    c = [c init];  
    [c set: numerator * b.denominator - denominator *  
    b.numerator over: denominator * b.denominator];  
    return c;  
}
```

```
-(Fraction *)mul: (Fraction *)b {  
    Fraction *c = [[Fraction alloc] init];  
    [c set: numerator * b.numerator over:  
    denominator * b.denominator];  
    return c;  
}  
  
-(Fraction *)div: (Fraction *)b {  
    Fraction *c = [[Fraction alloc] init];  
    [c set: numerator * b.denominator over:  
    denominator * b.numerator];  
    return c;  
}  
  
@end
```

```
// main.m
#import <Foundation/Foundation.h>
#import "Fraction.h"

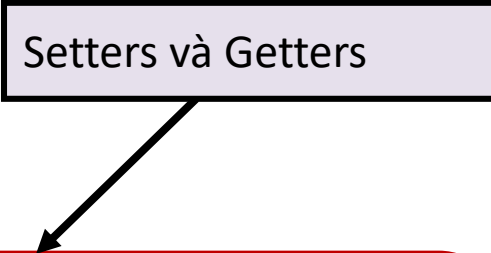
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Fraction *a = [Fraction alloc];
        a = [a init];
        [a set: 1 over: 3];
        Fraction *b = [[Fraction alloc] init];
        [b set: 5 over: 6];
        Fraction *c = [a add: b];
        [c print];
        [[a sub: b] print];
        [[a mul: b] print];
        [[a div: b] print];
    }
    return 0;
}
```

Setters và Getters

- Thiết kế lớp chuẩn thì các thành phần dữ liệu sẽ có thuộc tính truy xuất là private nên không thể truy xuất trực tiếp.
- Để có thể thao tác trên các dữ liệu ta dùng các phương thức *đưa vào* và *lấy ra*, gọi là *setter* và *getter*.
- Trong lớp Stack, phương thức đưa vào là push và phương thức lấy ra là pop.

Setter & Getter

```
class Fraction
{
    int numerator, denominator;
    void reduce();
    public:
    void print();
    void input();
    Fraction add(Fraction b);
    Fraction sub(Fraction b);
    Fraction mul(Fraction b);
    Fraction div(Fraction b);
    void setNumerator(int n) { numerator = n; }
    void setDenominator(int d) { denominator = d; }
    void set(int n, int c);
    int getNumerator() { return numerator; }
    int getDenominator() { return denominator; }
};
```



Setters và Getters

Setters & Getters

```
@interface Fraction : NSObject

+ (int)gcd: (int)a and: (int)b;
- (void)setNumerator: (int)n;
- (void)setDenominator: (int)d;
- (void)set: (int)n over: (int)d;
- (void)print;
- (Fraction *)add: (Fraction *)b;
- (Fraction *)sub: (Fraction *)b;
- (Fraction *)mul: (Fraction *)b;
- (Fraction *)div: (Fraction *)b;
@end
```

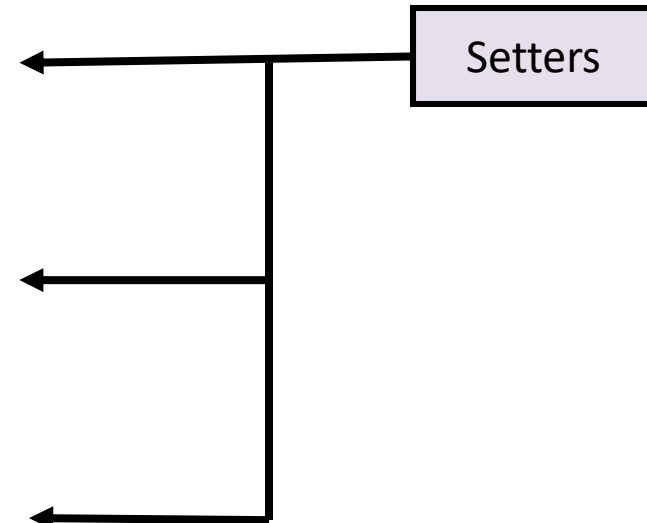
Setters

Setters & Getters

```
@implementation Fraction
//...
-(void)setNumerator: (int)n {
    numerator = n;
}

-(void)setDenominator: (int)d {
    denominator = d;
}

-(void)set: (int)n over: (int)d {
    int u = [Fraction gcd: n and: d];
    numerator = n / u;
    denominator = d / u;
    if (denominator < 0) {
        numerator = -numerator;
        denominator = -denominator;
    }
}
```



Setters & Getters

```
@implementation Fraction
```

```
{  
    int numerator, denominator;  
}
```

```
-(void)setNumerator: (int)n {  
    numerator = n;  
}
```

```
-(void)setDenominator: (int)d {  
    denominator = d;  
}
```

```
-(void)set: (int)n over: (int)d {  
    int u = [Fraction gcd: n and: d];  
    numerator = n / u;  
    denominator = d / u;  
    if (denominator < 0) {  
        numerator = -numerator;  
        denominator = -denominator;  
    }  
}
```

Setters

Setters & Getters

//...

```
-(int)numerator {  
    return numerator;  
}  
  
-(int)denominator {  
    return denominator;  
}
```

Getters



//...

@end

Tự động phát sinh setters & getters trong Objective C

- **Objective C** có cơ chế cho phép phát sinh các phương thức setter(s) và getter(s) một cách tự động.
- Ta dùng từ khoá **@property** trong phần giao diện và từ khoá **@synthesize** trong phần cài đặt.

setters & getters

```
@interface Fraction : NSObject
```

```
@property int numerator, denominator;
```

```
+(int)gcd: (int)a and : (int)b;
```

```
// -(void) setNumerator: (int) n;
```

```
// -(void) setDenominator: (int) d;
```

```
-(void)set: (int)n over : (int)d;
```

```
-(void)print;
```

```
-(Fraction *)add: (Fraction *)b;
```

```
-(Fraction *)sub: (Fraction *)b;
```

```
-(Fraction *)mul: (Fraction *)b;
```

```
-(Fraction *)div: (Fraction *)b;
```

```
@end
```

Tự động phát
sinh setters

No need

setters & getters

@implementation Fraction

@synthesize numerator, denominator;

```
/*
{
    int numerator, denominator;
}
-(void) setNumerator: (int) n {
    numerator = n;
}

-(void) setDenominator: (int) d {
    denominator = d;
}
-(int) numerator {
    return numerator;
}

-(int) denominator {
    return denominator;
} */
```

Đồng bộ với
@property

No need

Khởi động và dọn dẹp

Để đối tượng sẵn sàng ở trạng thái hoạt động, ta cần bảo đảm:

1. Đối tượng được khởi động.
2. Đối tượng không bị khởi động dư.

Đối tượng sau khi được sử dụng, ta cần bảo đảm:

1. Tài nguyên cấp cho đối tượng được giải phóng và
2. Tài nguyên này được giải phóng đúng 1 lần.

Người sử dụng có thể *quên khởi động* hoặc *quên dọn dẹp*.

Khởi động và dọn dẹp

- Ta có thể khởi động và dọn dẹp đối tượng một cách tự động nhờ phương thức thiết lập và huỷ bỏ
- Phương thức thiết lập và huỷ bỏ được xây dựng nhằm mục đích khắc phục lỗi quên khởi động đối tượng hoặc khởi động dư. Việc quên khởi động đối tượng thường gây ra những lỗi rất khó tìm.

Khởi động và dọn dẹp

- *Phương thức thiết lập* là hàm thành phần đặc biệt được tự động gọi đến mỗi khi một đối tượng thuộc lớp được tạo ra. Người ta thường lợi dụng đặc tính trên để khởi động đối tượng.
- Trong C++, phương thức thiết lập có tên trùng với tên lớp để phân biệt nó với các hàm thành phần khác.
- Có thể có nhiều phiên bản khác nhau của phương thức thiết lập

Khởi động và dọn dẹp

- *Phương thức huỷ bỏ* là hàm thành phần đặc biệt được tự động gọi đến mỗi khi một đối tượng bị huỷ đi. Người ta thường lợi dụng đặc tính trên để dọn dẹp đối tượng.
- Trong C++, Phương thức huỷ bỏ bắt đầu bằng dấu ngã (~) theo sau bởi tên lớp để phân biệt nó với các hàm thành phần khác.
- Chỉ có thể có tối đa một phương thức huỷ bỏ.
- Với PTTL & HB, ta đưa việc sử dụng đối tượng thành qui trình 1 bước thay vì 3 bước.


```

class Stack
{
    Item *st, *top;
    int size;
    void init(int sz){
        top = st = new Item[size = sz]; }
    void cleanUp(){ delete[] st; }
public:
    Stack(int sz = 20) { init(sz); }
    ~Stack() { cleanUp(); }
    bool empty() { return top <= st; }
    bool full() { return top - st >= size; }
    bool push(Item x);
    bool pop(Item *px);
};

```

Tự động khởi động
nếu quên khởi động

Sử dụng PT Thiết Lập & Huỷ bỏ

- Với phương thức thiết lập ta có thể “*dạy*” trình biên dịch:
 1. *Báo lỗi* nếu người sử dụng quên khởi động, hoặc
 2. *Khởi động tự động* nếu người sử dụng quên khởi động.
- Ta cũng có thể bảo đảm đối tượng không bị khởi động dư.
- Với phương thức huỷ bỏ, ta có cơ chế *dọn dẹp tự động*.
- Với PTTL và PTHB, ta có thể rút gọn qui trình 3 bước trở thành qui trình 1 bước.

```

class Stack
{
    Item *st, *top;
    int size;
    void init(int sz){
        top = st = new Item[size = sz]; }
    void cleanUp(){ delete[] st; }
public:
    Stack(int sz = 20) { init(sz); }
    ~Stack() { cleanUp(); }
    bool empty() { return top <= st; }
    bool full() { return top - st >= size; }
    bool push(Item x);
    bool pop(Item *px);
};

```

Tự động khởi động
nếu quên khởi động

```

void XuatHe16(long n)
{
    static char hTab[] = "0123456789ABCDEF";
    Stack s;
    int x;
    do {
        s.push(n % 16);
        n /= 16;
    } while (n);
    while (s.pop(&x))
        cout << hTab[x];
}

```

Qui trình sử dụng 1 bước:
Khai báo và sử dụng

```

class Stack
{
    Item *st, *top;
    int size;
    void init(int sz){
        top = st = new Item[size = sz]; }
    void cleanUp(){ delete[] st; }
public:
    Stack(int sz) { init(sz); }
    ~Stack() { cleanUp(); }
    bool empty() { return top <= st; }
    bool full() { return top - st >= size; }
    bool push(Item x);
    bool pop(Item *px);
};

```

Báo sai nếu quên
khởi động

Khởi động trong Objective C

- Ta có thể khởi động đối tượng trong Objective C bằng cách định nghĩa lại phương thức `init` của lớp cơ sở `NSObject`.

```
-(instancetype)init {  
    self = [super init];  
    if (self) {  
        numerator = 0;  
        denominator = 1; // or [self set: 0 over: 1];  
    }  
    return self;  
}
```

Tự khởi động 0/1
nếu quên khởi động

- Khởi động sẽ được nhắc lại ở chương sau.

```
int main(int argc, const char * argv[]) {  
@autoreleasepool {  
    Fraction *a = [[Fraction alloc] init];  
    [a set : 1 over : 3];  
    Fraction *b = [[Fraction alloc] init];  
    [b set : 5 over : 6];  
    Fraction *c = [[Fraction alloc] init];  
    [a print];  
    [b print];  
    [c print];  
}  
return 0;  
}
```

Khởi động tự động,
dọn dẹp tự động

```
void XuatHe16(long n)
{
    static char hTab[] = "0123456789ABCDEF";
    Stack s;
    int x;
    do {
        s.push(n % 16);
        n /= 16;
    } while (n);
    while (s.pop(&x))
        cout << hTab[x];
}
```

Trình tự sử dụng: Tạo đối tượng và sử dụng thực sự

Các đặc tính khác của lớp

- Tự tham chiếu
- Thuộc tính lớp và phương thức lớp (thành phần tĩnh)
- Hàm thành phần hằng (C++)
- Hàm bạn (C++)

Tự tham chiếu

- Là tham số ngầm định của phương thức trả về đến đối tượng gọi thao tác. Nhờ đó phương thức biết được nó đang thao tác trên đối tượng nào.
- Khi một đối tượng gọi một thao tác, địa chỉ của đối tượng được gửi đi một cách ngầm định, trong C++ với tên **this**, trong Objective C là **self**.
- Tên các thành phần của đối tượng trong một phương thức được hiểu là của đối tượng có địa chỉ **this** hoặc **self**.

Con trỏ this

```
bool Stack::push(Item x)
{
    if (full()) // if (this->full())
        return false;
    *top++ = x; // *this->top++ = x;
    return true;
}
```

- Ta thường không cần tường minh sử dụng từ khoá **this**.

Từ khoá self

```
-(Fraction *)addIn: (Fraction *)b {  
    numerator = numerator * b.denominator +  
    denominator * b.numerator;  
    denominator *= b.denominator;  
    [self reduce]; // use self to call reduce  
    return self;  
}
```

Hoặc

```
-(Fraction *)addIn: (Fraction *)b {  
    // use self to call set  
    [self set: numerator * b.denominator +  
    denominator * b.numerator over:  
    denominator * b.denominator];  
    return self;  
}
```

Phương thức hằng

```
inline char *__strdup(const char *s) {  
    return strcpy(new char[strlen(s) + 1], s);  
}  
  
class String {  
    char *p;  
public:  
    String(const char *s) { p = __strdup(s); }  
    String(const String &s) { p = __strdup(s.p); }  
    ~String() { if (p) delete [] p; }  
    int getLength() { return strlen(p); }  
    void print();  
    void set(const char *s);  
    void input();  
    void concat(const String &b);  
};
```

```
int main()
{
    String b("This is a string");
    b.print();
    cout << "\n";
    b.set("a variable string");
    b.print();
    cout << "\n";

    String mySchool("Dai Hoc Tu Nhien");
    mySchool.print();
    cout << "\n";
    mySchool.set("Tieu Hoc Tu Nhien"); // Oh dear!
    mySchool.print();
    cout << "\n";

    return 0;
}
```

```

int main()
{
    String b("This is a string");
    b.print();
    cout << "\n";
    b.set("a variable string");
    b.print();
    cout << "\n";

    const String mySchool("Dai Hoc Tu Nhien");
    mySchool.print();
    cout << "\n";
    mySchool.set("Tieu Hoc Tu Nhien"); // Error, good!
    mySchool.print();
    // Error, Oh dear!
    cout << "\n";

    return 0;
}

```

Phương thức hằng

- Phương thức hằng hay hàm thành phần hằng là phương thức có thể áp dụng được cho các đối tượng hằng.
- Ta qui định một phương thức là hằng bằng cách thêm từ khoá `const` vào cuối khai báo của nó.
- Ta khai báo phương thức là hằng khi nó không thay đổi các thành phần dữ liệu của đối tượng.

Phương thức hằng

```
inline char *__strdup(const char *s) {  
    return strcpy(new char[strlen(s) + 1], s);  
}  
  
class String {  
    char *p;  
public:  
    String(const char *s) { p = __strdup(s); }  
    String(const String &s) { p = __strdup(s.p); }  
    ~String() { if (p) delete p; }  
    int getLength() const { return strlen(p); }  
    void print() const;  
    void set(const char *s);  
    void input();  
    void concat(const String &b);  
};
```

```

int main()
{
    String b("This is a string");
    b.print();
    cout << "\n";
    b.set("a variable string");
    b.print();
    cout << "\n";

    const String mySchool("Dai Hoc Tu Nhien");
    mySchool.print();
    cout << "\n";
    mySchool.set("Tieu Hoc Tu Nhien"); // Error, good!
    mySchool.print(); // Ok, good!
    cout << "\n";

    return 0;
}

```

Hàm bạn

- Nguyên tắc chung khi thao tác trên lớp là thông qua các hàm thành phần (setters, getters). Tuy nhiên có những trường hợp ngoại lệ, khi hàm phải thao tác trên hai lớp.
- Hàm bạn của một lớp là hàm được khai báo ở bên ngoài nhưng được phép truy xuất các thành phần riêng tư của lớp.

Hàm bạn

- Ta làm một hàm trở thành hàm bạn của lớp bằng cách đưa khai báo của hàm đó vào trong lớp, thêm từ khoá friend ở đầu.
- Ta dùng hàm bạn trong trường hợp hàm phải là hàm toàn cục nhưng có liên quan mật thiết với lớp, hoặc là hàm thành phần của một lớp khác.

Nhân ma trận, không dùng hàm bạn

```
const int N = 4;
class Vector
{
    double a[N];
public:
    double Get(int i) const { return a[i]; }
    void Set(int i, double x) { a[i] = x; }
};

class Matrix
{
    double a[N][N];
public:
    double Get(int i, int j) const { return a[i][j]; }
    void Set(int i, int j, double x) { a[i][j] = x; }
    //...
};
```

Nhân ma trận, không dùng hàm bạn

```
Vector Multiply(const Matrix &m, const
Vector &v)
{
    Vector r;
    for (int i = 0; i < N; i++)
    {
        r.Set(i, 0);
        for (int j = 0; j < N; j++)
            r.Set(i, r.Get(i) + m.Get(i,
j)*v.Get(j));
    }
    return r;
}
```

Nhân ma trận, dùng hàm bạn

```
class Matrix;
class Vector
{
    double a[N];
public:
    double Get(int i) const { return a[i]; }
    void Set(int i, double x) { a[i] = x; }
    friend Vector Multiply(const Matrix &m, const Vector &v);
};
```

```
class Matrix
{
    double a[N][N];
public:
    double Get(int i, int j) const { return a[i][j]; }
    void Set(int i, int j, double x) { a[i][j] = x; }
    friend Vector Multiply(const Matrix &m, const Vector &v);
};
```

```
Vector Multiply(const Matrix &m, const  
Vector &v)
```

Nhân ma trận, dùng hàm bạn

```
{  
    Vector r;  
    for (int i = 0; i < N; i++)  
    {  
        r.a[i] = 0;  
        for (int j = 0; j < N; j++)  
            r.a[i] += m.a[i][j] * v.a[j];  
    }  
    return r;  
}
```


Thuộc tính lớp và phương thức lớp

- Thuộc tính lớp là thuộc tính dùng chung cho mọi đối tượng thuộc lớp.
- Phương thức lớp là phương thức có thể hoạt động không thuộc tính dữ liệu của lớp, nghĩa là nó không cần đối tượng.
- Ta dùng phương thức lớp thay cho hàm toàn cục.
- Trong C++, từ khoá static được dùng để khai báo thuộc tính hay phương thức là của lớp.

Thuộc tính và phương thức lớp

```
class CDate
{
    static int dayTab[][13];
    int day, month, year;
public:
    static bool LeapYear(int y) { return y %
400 == 0 || y % 4 == 0 && y % 100 != 0; }
    static int DayOfMonth(int m, int y);
    static bool ValidDate(int d, int m, int y);
    void Input();
};
```

Thuộc tính và phương thức lớp

```
class CDate
{
    static int dayTab[][13];
    int day, month, year;
public:
    static bool LeapYear(int y) { return y %
400 == 0 || y % 4 == 0 && y % 100 != 0; }
    static int DayOfMonth(int m, int y);
    static bool ValidDate(int d, int m, int y);
    void Input();
};
```

Thuộc tính và phương thức lớp

```
int CDate::dayTab[][13] =  
{  
    { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },  
    { 0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }  
};
```

```
int CDate::DayOfMonth(int m, int y)  
{  
    return dayTab[LeapYear(y)][m];  
}
```

```
bool betw(int x, int a, int b)  
{  
    return x >= a && x <= b;  
}
```

Thuộc tính và phương thức lớp

```
bool CDate::ValidDate(int d, int m, int y)
{
    return betw(m, 1, 12) && betw(d, 1, DayOfMonth(m,
y));
}
void CDate::Input()
{
    int d, m, y;
    cin >> d >> m >> y;
    while (!ValidDate(d, m, y))
    {
        cout << "Please enter a valid date: ";
        cin >> d >> m >> y;
    }
    day = d; month = m; year = y;
}
```

Thuộc tính và phương thức lớp

```
class Stack
{
    Item *st, *top;
    int size;
    void init(int sz);
    void cleanUp() { if (st) delete[] st; }
public:
    Stack(int sz = 20) { init(sz); }
    ~Stack() { CleanUp(); }
    static Stack *create(int sz);
    bool full()const { return (top - st >= size); }
    bool empty() const { return (top <= st); }
    bool push(Item x);
    bool pop(Item *px);
};
```

Thuộc tính và phương thức lớp

```
Stack *Stack::create(int sz)
{
    Stack *ps = new Stack(sz);
    if (!ps->st)
    {
        delete ps;
        return NULL;
    }
    return ps;
}
```

Phương thức lớp Stack::create được dùng thay cho phương thức thiết lập

```
void main()
{
    Stack *ps = new Stack(50000); // ko biet tao duoc stack khong
    Stack *pr = Stack::create(50000);
    if (!pr)
    { cout << "Khong the tao stack"; return; }
    else
        pr->push(5); // ...
}
```

Thuộc tính và phương thức lớp

```
#import <Foundation/Foundation.h>
```

```
@interface CDate : NSObject
+(BOOL)leapYear : (int)y;
+(int)dayOfMonth: (int)m andYear :
(int)y;
+(BOOL)validDay: (int)d andMonth : (int)m
andYear : (int)y;
-(void)input;
-(void)print;

@end
```

Phương thức lớp

Thuộc tính và phương thức lớp

```
#import "CDate.h"
```

```
@implementation Cdate
```

```
{
```

```
    int day, month, year;
```

```
}
```

```
static const int dayTab[][13] = {
```

```
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
```

```
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
```

```
};
```

```
BOOL betw(int x, int a, int b) {
```

```
    return a <= x && x <= b;
```

```
}
```

```
+(BOOL)leapYear:(int)y {
```

```
    return (y % 400 == 0) || (y % 4 == 0 && y % 100 != 0);
```

```
}
```

Thuộc tính lớp

Thuộc tính và phương thức lớp

```
+(int)dayOfMonth : (int)m andYear : (int)y {  
    BOOL b = [CDate leapYear : (y)];  
    return dayTab[b][m];  
}  
  
+(BOOL)validDay : (int)d andMonth : (int)m  
andYear : (int)y {  
    return betw(d, 1, [CDate dayOfMonth : m  
andYear : y]);  
}  
  
-(void)print {  
    NSLog(@"%02d / % 02d / %d: ", day, month,  
year);  
}
```

```
-(void)input {  
    int d, m, y;  
    scanf("%d%d%d", &d, &m, &y);  
    while (![CDate validDay : d andMonth : m  
andYear : y]) {  
        NSLog(@"Enter valid date(d m y : ");  
        scanf("%d%d%d", &d, &m, &y);  
    }  
    day = d;  
    month = m;  
    year = y;  
}
```

@end

```
#import <Foundation/Foundation.h>
#import "cdate.h"

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        CDate *d = [[CDate alloc] init];
        NSLog(@"Enter valid date : ");
        [d input];
        [d print];
    }
    return 0;
}
```

Thiết lập và huỷ bỏ đối tượng

- Ta cần kiểm soát khi nào phương thức thiết lập được gọi, khi nào phương thức huỷ bỏ được gọi.
- Khi nào đối tượng thiết lập được gọi? Khi đối tượng được tạo ra.
- Khi nào phương thức huỷ bỏ được gọi? Khi đối tượng bị huỷ đi.
- Thời gian từ khi đối tượng được tạo ra đến khi nó bị huỷ đi được gọi là thời gian sống.

Thời gian sống

- Thời gian sống của đối tượng khác nhau tùy thuộc cách đối tượng được lưu trữ.
 1. Đối tượng tự động.
 2. Đối tượng tĩnh
 3. Đối tượng trong vùng nhớ tự do
 4. Đối tượng tạm thời
 5. Đối tượng thành phần

Đối tượng tự động

- Đối tượng tự động (automatic objects) là đối tượng được tự động sinh ra và tự động bị huỷ đi. Đối tượng tự động có thể là đối tượng địa phương, là tham số truyền bằng giá trị hoặc là đối tượng giá trị trả về.
- Đối tượng địa phương
 - Được khai báo, định nghĩa bên trong một khối.
 - Được tự động sinh ra khi chương trình thực hiện ngang dòng lệnh chứa định nghĩa và bị huỷ đi sau khi chương trình hoàn tất khối chứa định nghĩa đó.

Đối tượng tự động

- Khi khởi động một đối tượng bằng một đối tượng cùng kiểu, cơ chế tạo đối tượng mặc nhiên là sao chép từng thành phần, do đó đối tượng được khởi động có khả năng sẽ chia sẻ tài nguyên với đối tượng nguồn.
- Khi 2 đối tượng này bị huỷ đi, phần *tài nguyên chung sẽ bị giải phóng 2 lần!*.

Đối tượng địa phương

```
inline char *__strdup(const char *s) {  
    return strcpy(new char[strlen(s) + 1], s);  
}
```

```
class String {  
    char *p;  
public:  
    String(const char *s) { p = __strdup(s); }  
    ~String() { if (p) delete [] p; }  
    int getLength() const { return strlen(p); }  
    void print();  
    void set(const char *s);  
    void input();  
    void concat(const String &b);  
};
```

Đối tượng địa phương

```
#include <iostream>
using namespace std;
#include "string.h"
int main()
{
    String a("This is a string");
    a.print();
    cout << "\n";
    String b = a; // String b(a);
    b.print();
    cout << "\n";
    return 0;
}
```

Phương thức huỷ bỏ được gọi cho cả a và b, phần tài nguyên chung bị giải phóng 2 lần

output

new 00000106C60DFE40

a: Dai Hoc Tu Nhlen

b: Dai Hoc Tu Nhlen

Dai Hoc Tu Nhlen

delete 00000106C60DFE40

delete 00000106C60DFE40

Đối tượng là tham trị

```
class String {  
    char *p;  
public:  
    String(const char *s) { p = __strdup(s); }  
    ~String() { if (p) delete p; }  
    int getLength() const { return strlen(p); }  
    void print();  
    void set(const char *s);  
    void input();  
    void concat(const String &b);  
};
```

Đối tượng là tham trị

- Đối tượng là tham số hàm, truyền bằng giá trị thì tham số hình thức là bản sao của tham số thực sự, nên có nội dung vật lý giống tham số thực sự do cơ chế sao chép từng thành phần.
- Khi kết thúc hàm, đối tượng tham trị bị huỷ, phương thức huỷ bỏ được gọi, phần *tài nguyên chung sẽ bị giải phóng* trong khi tham số thực sự vẫn tồn tại!.

Đối tượng là tham trị

```
class String
{
    char *p;
public:
    String(const char *s) { p = __strdup(s);
    }
    ~String() { if (p) { delete[] p; }
    }
    int getLength() const { return strlen(p);
    }
    void print() const;
    void input();
    int compare(String s) const;
};
```

```

int String::compare(String s) const
{
    return strcmp(p, s.p);
}
void main()
{
    String a("Dai Hoc Tu Nhlen");
    String b("Dai Hoc Bach Khoa");
    cout << "a = "; a.print(); cout << "\n";
    cout << "b = "; b.print(); cout << "\n";
    int c = a.compare(b);
    cout << (c > 0 ? "a > b" : c == 0 ? "a = b" :
    "a < b") << "\n";
}

```

Đối tượng là tham trị

Phương thức huỷ bỏ được gọi bao nhiêu lần?

Output

```
new 0000029BA7DCF930
new 0000029BA7DD0290
a = Dai Hoc Tu Nhlen
b = Dai Hoc Bach Khoa
delete 0000029BA7DD0290
a > b
delete 0000029BA7DD0290
```


Đối tượng giá trị trả về

```
class String
{
    char *p;
public:
    String(const char *s = "Alibaba") { p =
        __strdup(s);}
    ~String() { if (p) { delete[] p;}
    int getLength() const { return strlen(p); }
    int compare(String s) const;
    String upCase() const;
};
```

Đối tượng giá trị trả về

```
String String::upCase() const
{
    String r = *this;
    strupr(r.p);
    return r;
}
```

```
void main()
{
    String a("Dai Hoc Tu Nhlen");
    cout << "a = "; a.print(); cout << "\n";
    String A;
    A = a.upCase();
    cout << "a = "; a.print(); cout << "\n";
    cout << "A = "; A.print(); cout << "\n";
}
```

Output

[illegible]

Đối tượng giá trị trả về

- Đối tượng giá trị trả về là bản sao của biểu thức trả về, do đó có sự chia sẻ tài nguyên (*sai*).
- Trong ví dụ trên, tài nguyên còn bị chia sẻ do phép gán.
- Có thể thay phép gán bằng khởi động
`String A a.toUpperCase();`
Nhưng vẫn còn lỗi chia sẻ tài nguyên do đối tượng giá trị trả về.

Phương thức thiết lập bản sao

- Các lỗi sai nêu trên gây ra do sao chép đối tượng, ta có thể khắc phục bằng cách “dạy” trình biên dịch sao chép đối tượng một cách luận lý thay vì sao chép từng thành phần theo nghĩa vật lý. Điều đó được thực hiện nhờ phương thức thiết lập bản sao.
- Phương thức thiết lập bản sao là phương thức thiết lập có tham số là đối tượng cùng kiểu, dùng để sao chép đối tượng.

Phương thức thiết lập bản sao

- Phương thức thiết lập bản sao thực hiện sao chép theo nghĩa logic, thông thường là tạo nên tài nguyên mới (sao chép sâu).
- Phương thức thiết lập bản sao cũng có thể chia sẻ tài nguyên cho các đối tượng (sao chép nông). Trong trường hợp này, cần có cơ chế để kiểm soát việc huỷ bỏ đối tượng.

Phương thức thiết lập bản sao

- Tham số của phương thức thiết lập bản sao bắt buộc là tham chiếu.
- Phương thức thiết lập bản sao có thể được dùng để sao chép nông, tài nguyên vẫn được chia sẻ nhưng có một biến đếm để kiểm soát.

Lưu ý:

- Nếu đối tượng không có tài nguyên riêng thì không cần phương thức thiết lập bản sao.
- Khi truyền tham số là đối tượng thuộc lớp có phương thức thiết lập bản sao, ta nên truyền bằng tham chiếu và có thêm từ khoá **const**.

```

class String
{
    char *p;
public:
    String(const char *s = "Alibaba") { p =
        __strdup(s); }
    String(const String &s) { p = __strdup(s.p); }
    ~String() { if (p) { cout << "delete " << (void
        *)p << "\n"; delete[] p; } }
    int getLength() const { return strlen(p); }
    void print() const;
    void input();
    void set(const char *s);
    void concat(const String &b);
    int compare(String s) const;
    String upCase() const;
};

```

Copy
constructor




```
inline int String::compare(String s)
const
{
    return strcmp(p, s.p);
}
```

```
inline String String::upCase() const
{
    String r = *this;
    strupr(r.p);
    return r;
}
```

0.

```
void main()
{
    String a("Dai Hoc Tu Nhlen");
    String b("Dai Hoc Bach Khoa");
    String aa = a;
    cout << "a: "; a.print(); cout << "\n";
    cout << "b: "; b.print(); cout << "\n";
    cout << "aa: "; aa.print(); cout << "\n";
    int c = a.compare(b);
    cout << "b: "; b.print(); cout << "\n";
    cout << (c > 0 ? "a > b" : c == 0 ? "a = b" : "a
    < b") << "\n";
    cout << "a = "; a.print(); cout << "\n";
    String A = a.upCase();
    cout << "a = "; a.print(); cout << "\n";
    cout << "A = "; A.print(); cout << "\n";
}
```

Output

```
a: Dai Hoc Tu Nhlen
b: Dai Hoc Bach Khoa
aa: Dai Hoc Tu Nhlen
delete 000001D7BFE46BC0
b: Dai Hoc Bach Khoa
a > b
a = Dai Hoc Tu Nhlen
delete 000001D7BFE47250
a = Dai Hoc Tu Nhlen
A = DAI HOC TU NHIEN
delete 000001D7BFE46A80
delete 000001D7BFE46B20
delete 000001D7BFE475C0
delete 000001D7BFE469E0
```

Sao chép nông và sao chép sâu

- Dùng phương thức thiết lập bản sao như trên, trong đó đối tượng mới có tài nguyên riêng *là sao chép sâu*.
- Ta có thể *sao chép nông* bằng cách chia sẻ tài nguyên và dùng một *biến đếm* để kiểm soát số thể hiện các đối tượng có chia sẻ cùng tài nguyên.
- Khi một đối tượng thay đổi nội dung, nó phải được tách ra khỏi các đối tượng dùng chung tài nguyên, nói cách khác, nó phải có tài nguyên riêng (như sao chép sâu).

```

class StringRep
{
    friend class String;
    char *p;
    int n;
    StringRep(const char *s) { p = strdup(s); n = 1; }
    ~StringRep() { cout << "delete " << (void *)p << "\n";
        delete [] p; }
};

class String {
    StringRep *rep;
public:
    String(const char *s = "Alibaba"){ rep = new StringRep(s); }
    String(const String &s) { rep = s.rep; rep->n++; }
    ~String();
    void print() const { cout << rep->p; }
    int compare(String s) const;
    String upCase() const;
    void toUpper();
};

```

```

String::~~String()
{
    if (--rep->n <= 0)
        delete rep;
}

int String::compare(String s) const
{
    return strcmp(rep->p, s.rep->p);
}

String String::upCase() const
{
    String r(rep->p);
    strupr(r.rep->p);
    return r;
}

```

```

int main()
{
    String a("Dai Hoc Tu Nhlen");
    String b("Dai Hoc Bach Khoa");
    String aa = a;
    cout << "a: "; a.print(); cout << "\n";
    cout << "b: "; b.print(); cout << "\n";
    cout << "aa: "; aa.print(); cout << "\n";
    int c = a.compare(b);
    cout << "b: "; b.print(); cout << "\n";
    cout << (c > 0 ? "a > b" : c == 0 ? "a = b" : "a < b")
    << "\n";
    cout << "a = "; a.print(); cout << "\n";
    String A = a.upCase();
    cout << "a = "; a.print(); cout << "\n";
    cout << "A = "; A.print(); cout << "\n";
    return 0;
}

```

```
a: Dai Hoc Tu Nhlen
b: Dai Hoc Bach Khoa
aa: Dai Hoc Tu Nhlen
b: Dai Hoc Bach Khoa
a > b
a = Dai Hoc Tu Nhlen
a = Dai Hoc Tu Nhlen
A = DAI HOC TU NHIEN
delete 00000146B7A90750
delete 00000146B7A901B0
delete 00000146B7A90390
Press any key to continue . . .
```


Đối tượng tĩnh

Trong C++

- Đối tượng *tĩnh* có thể là đối tượng được định nghĩa toàn cục, được định nghĩa có thêm từ khoá static (toàn cục hoặc địa phương).
- Đối tượng toàn cục hoặc tĩnh toàn cục được tạo ra khi *bắt đầu chương trình* và bị huỷ đi khi *kết thúc chương trình*.
- Đối tượng tĩnh địa phương được tạo ra khi chương trình thực hiện đoạn mã chứa định nghĩa đối tượng lần đầu tiên và bị huỷ đi khi *kết thúc chương trình*.

Đối tượng tĩnh

Trình tự thực hiện chương trình gồm:

1. Gọi phương thức thiết lập cho các đối tượng toàn cục
2. Thực hiện hàm `main()`
3. Gọi phương thức huỷ bỏ cho các đối tượng toàn cục

Đối tượng tĩnh

Hãy sửa đoạn chương trình sau:

```
void main()  
{  
    cout << "Hello, world.\n";  
}
```

Để có xuất liệu:

Entering a C++ program saying...

Hello, world.

And then exiting...

Yêu cầu không thay đổi hàm main() dưới bất kỳ hình thức nào.

Đối tượng tĩnh

```
void main()
{
    cout << "Hello, world.\n";
}

class Dummy
{
public:
    Dummy() { cout << "Entering a C++ program
saying...\n"; }
    ~Dummy() { cout << "And then
exitting..."; }
};
```

Đối tượng là thành phần của lớp

- Đối tượng có thể là thành phần của đối tượng khác, khi một đối tượng thuộc lớp “lớn” được tạo ra, các thành phần của nó cũng được tạo ra. Phương thức thiết lập (nếu có) sẽ được tự động gọi cho các đối tượng thành phần.
- Nếu đối tượng thành phần phải được cung cấp tham số khi thiết lập thì đối tượng kết hợp (đối tượng lớn) phải có phương thức thiết lập để cung cấp tham số thiết lập cho các đối tượng thành phần.

Đối tượng là thành phần của lớp

- Cú pháp để khởi động đối tượng thành phần trong C++ là dùng dấu hai chấm (:) theo sau bởi tên thành phần và tham số khởi động.
- Khi đối tượng kết hợp bị huỷ đi thì các đối tượng thành phần của nó cũng bị huỷ đi, nghĩa là phương thức huỷ bỏ sẽ được gọi cho các đối tượng thành phần, sau khi phương thức huỷ bỏ của đối tượng kết hợp được gọi.
- Trong Objective C, đối tượng thành phần không phải kiểu cơ bản bắt buộc là *con trỏ*.

```
class Point
{
    double x, y;
public:
    Point(double xx, double yy) { x = xx; y = yy; }
    // ...
};
```

```
class Triangle
{
    Point A, B, C;
public:
    void draw() const;
    // ...
};
```

```
Triangle t; // Error
```

```

class String
{
    char *p;
public:
    String(const char *s) { p = __strdup(s); }
    String(const String &s) { p = __strdup(s.p); }
    ~String() { if (p) delete[] p; }
    //...
};
class Student
{
    String MaSo;
    String HoTen;
    int NamSinh;
public:
};
Student s; // Error

```


Khởi động đối
tượng thành phần

```
class Point
{
    double x, y;
public:
    Point(double xx, double yy) { x = xx; y = yy; }
    // ...
};
class Triangle
{
    Point A, B, C;
public:
    Triangle(double xA, double yA, double xB, double yB,
double xC, double yC): A(xA, yA), B(xB, yB), C(xC, yC)
    {}
    void draw() const;
    // ...
};

Triangle t(100, 100, 200, 400, 300, 300); // Ok
```

```

class String
{
    char *p;
public:
    String(const char *s) { p = __strdup(s); }
    String(const String &s) { p = __strdup(s.p); }
    ~String() { if (p) delete[] p; }
    //...
};
class Student
{
    String id;
    String name;
    int birthYear;
public:
    Student(const char *n, const char *i, int y): name(n),
        id(i) { birthYear = y; }
};
Student s("To Van Ve", "20182145", 2000); // Ok

```

Đối tượng là thành phần của lớp

- Cú pháp để khởi động đối tượng thành phần trong C++ dùng dấu hai chấm (:) cũng được dùng cho đối tượng thành phần thuộc kiểu cơ bản.

```
class Point
{
    double x, y;
public:
    Point(double xx, double yy):x(xx), y(yy) { }
    // ...
};
```

Đối tượng là thành phần của lớp

```
class Student
{
    String id;
    String name;
    int birthYear;
public:
    Student(const char *n, const char *i, int
y): name(n), id(i), birthYear(y) { }
};

Student s("Vo Van Ve", "20182145", 2000); // Ok
```

Đối tượng thành phần – Huỷ bỏ

- Khi đối tượng kết hợp bị huỷ bỏ, các đối tượng thành phần của nó cũng bị huỷ bỏ. Phương thức huỷ bỏ cho các đối tượng thành phần sẽ được gọi một cách tự động.

Đối tượng hành phần – huỷ bỏ

```
class Student
{
    String id;
    String name;
    int birthYear;
    int nSubjects;
    double *aPoints;
public:
    Student(const char *n, const char *i, int y,
            int nS, double *d) : name(n), id(i),
            birthYear(y), nSubjects(nS) {
        memcpy(aPoints = new double[nSubjects], d,
            nSubjects * sizeof(double));
    }
    ~Student() { delete[] aPoints; }
};
```

Đối tượng là thành phần của mảng

- Khi một mảng được tạo ra, các phần tử của nó cũng được tạo ra, do đó phương thức thiết lập sẽ được gọi cho từng phần tử một.
- Vì không thể cung cấp tham số khởi động cho tất cả các phần tử của mảng nên khi khai báo mảng, mỗi đối tượng trong mảng phải có khả năng tự khởi động, nghĩa là có thể được thiết lập không cần tham số.

Đối tượng là thành phần của mảng

- Đối tượng có khả năng tự khởi động trong các trường hợp sau:
 1. Lớp không có phương thức thiết lập.
 2. Lớp có phương thức thiết lập không tham số.
 3. Lớp có phương thức thiết lập mà mọi tham số đều có giá trị mặc nhiên.

Đối tượng là thành phần của mảng

```
class Point {  
    double x, y;  
public:  
    Point(double xx, double yy) { x = xx; y = yy; }  
    // ...  
};  
Point ap[5]; // Error
```

```
class String {  
    char *p;  
public:  
    String(const char *s) { p = __strdup(s); }  
    String(const String &s) { p = __strdup(s.p); }  
    ~String() { if (p) delete[] p; }  
    //...  
};  
String as[3]; // Error
```

Đối tượng là thành phần của mảng

```
class Student
```

```
{
```

```
    String id;
```

```
    String name;
```

```
    int birthYear;
```

```
public:
```

```
    Student(const char *n, const char *i, int  
y): name(n), id(i), birthYear(y) { }
```

```
    // ...
```

```
};
```

```
Student aS[4]; // Error
```

Đối tượng là thành phần của mảng

```
class Point {
    double x, y;
public:
    Point(double xx, double yy) { x = xx; y = yy; }
    Point() : x(0), y(0) {}
    // ...
};
Point ap[5]; // Ok

class String {
    char *p;
public:
    String(const char *s = "Alibaba") { p = __strdup(s); }
    String(const String &s) { p = __strdup(s.p); }
    ~String() { if (p) delete[] p; }
    //...
};
String as[3]; // Ok
```

Đối tượng là thành phần của mảng

```
class Student
{
    String id;
    String name;
    int birthYear;
public:
    Student(const char *n = , const char *i,
            int y): name(n), id(i), birthYear(y) { }
    // ...
};
```

```
Student aS[4]; // Error
```

```
class Student
{
    String id;
    String name;
    int birthYear;
public:
    Student(const char *n = "Aladin", const
char *i = "20182132", int y = 2000):
        name(n), id(i), birthYear(y) { }
    // ...
};

Student aS[4]; // Ok
```

```
class Student
{
    String id;
    String name;
    int birthYear;
public:
    Student(const char *n = "Aladin", const
    char *i = "20182132", int y = 2000):
        name(n), id(i), birthYear(y) { }
    // ...
};

Student aS[4]; // Ok
```

PTTL không tham số

```
class Student
{
    String id;
    String name;
    int birthYear;
public:
    Student(const char *n, const char *i, int y):
        name(n), id(i), birthYear(y) { }
    Student() : name("Aladin"), id("20182132"),
        birthYear(2000) { }

    // ...
};

Student aS[4]; // Ok
```

Đối tượng được cấp phát động

- Đối tượng được cấp phát động là đối tượng được tạo ra bằng phép toán **new** và bị huỷ đi bằng phép toán **delete**
- Phép toán **new** cấp đối tượng trong vùng heap (hay vùng free store) và gọi phương thức thiết lập cho đối tượng được cấp.
- Dùng **new** có thể cấp một đối tượng và dùng **delete** để huỷ một đối tượng.
- Dùng **new** và **delete** cũng có thể cấp nhiều đối tượng và huỷ nhiều đối tượng.

Đối tượng được cấp phát động

```
class Point {  
    double x, y;  
public:  
    Point(double xx, double yy) { x = xx; y = yy; }  
    // ...  
};
```

```
class String {  
    char *p;  
public:  
    String(const char *s) { p = __strdup(s); }  
    String(const String &s) { p = __strdup(s.p); }  
    ~String() { if (p) delete[] p; }  
    //...  
};
```

Cấp và huỷ 1 đối tượng

```
int *pi = new int;  
int *pj = new int(15);  
Point *pd = new Point(20, 40);  
String *pa = new String("Nguyen Van A");  
//...  
//...
```

```
delete pa;  
delete pd;  
delete pj;  
delete pi;
```

Đối tượng được cấp phát động

```
class Point {  
    double x, y;  
public:  
    Point(double xx, double yy) { x = xx; y = yy; }  
    Point() : x(0), y(0) {}  
    // ...  
};
```

```
class String {  
    char *p;  
public:  
    String(const char *s = "Alibaba") { p =  
        __strdup(s); }  
    String(const String &s) { p = __strdup(s.p); }  
    ~String() { if (p) delete[] p; }  
    //...  
};
```

Cấp và huỷ nhiều đối tượng

```
// Cac doi tuong phai co  
// kha nang tu khoi dong
```

```
int *pai = new int[10];  
Point *pad = new Point[5]; // Ok  
String *pas = new String[5]; // Ok  
// ...  
// ...
```

```
delete[] pas;  
delete[] pad;  
delete[] pai;
```

Giao diện và chi tiết cài đặt

- Lớp có hai phần tách rời, một là phần **giao diện** khai báo trong phần public để người sử dụng “thấy” và sử dụng, và hai là **chi tiết cài đặt** bao gồm dữ liệu khai báo trong phần **private** của lớp và chi tiết mã hoá các hàm thành phần, vô hình đối với người dùng.
- Ta có thể **thay đổi uyển chuyển chi tiết cài đặt**, nghĩa là có thể thay đổi tổ chức dữ liệu của lớp, cũng như có thể thay đổi chi tiết thực hiện các phương thức (do sự thay đổi tổ chức dữ liệu hoặc để cải tiến giải thuật). Nhưng phải bảo đảm **không thay đổi phần giao diện** để không ảnh hưởng đến người sử dụng, và do đó không làm đổ vỡ kiến trúc của hệ thống.

Giao diện và chi tiết cài đặt

Ví dụ:

- Lớp Time biểu diễn khái niệm thời điểm, có thể được cài đặt bằng ba thuộc tính dữ liệu biểu diễn giờ phút giây hoặc bằng một thuộc tính dữ liệu là tổng số giây tính từ 0g.
- Lớp Stack biểu diễn cấu trúc dữ liệu hoạt động theo nguyên tắc LIFO có thể được cài đặt bằng mảng hoặc danh sách liên kết

Time: hour, minute, second

```
class Time
{
    int hour, minute, second;
    static bool valid(int h, int m, int s);
public:
    Time(int h = 0, int m = 0, int s = 0) { set(h, m, s); }
    void set(int h, int m, int s);
    int getHour() const { return hour; }
    int getMinute() const { return minute; }
    int getSecond() const { return second; }
    void input();
    void print() const;
    void increase();
    void decrease();
};
```

```

const long SECONDS_OF_DAY = 3600L * 24;
class Time
{
    int totalSeconds;
    static bool valid(int h, int m, int s);
public:
    Time(int h = 0, int m = 0, int s = 0) { set(h, m, s); }
    void set(int h, int m, int s);
    int getHour() const { return totalSeconds / 3600; }
    int getMinute() const { return (totalSeconds % 3600) / 60; }
    int getSecond() const { return totalSeconds % 60; }
    void input();
    void print() const;
    void increase();
    void decrease();
};

```

Time: totalSeconds

Các nguyên tắc xây dựng lớp

- Khi điều ta nghĩ đến là một khái niệm riêng rẽ, xây dựng lớp biểu diễn khái niệm đó. Ví dụ lớp Sinh Viên, lớp Tam Giác, lớp Người, lớp Phân Số...
- Khi điều ta nghĩ đến là một thực thể cụ thể riêng biệt, tạo đối tượng thuộc lớp. Ví dụ đối tượng Sinh viên “Nguyen Van A” (và các thuộc tính khác như mã số, năm sinh...).

Các nguyên tắc xây dựng lớp

- Lớp là biểu diễn cụ thể của một khái niệm, vì vậy lớp luôn luôn là *danh từ*.
- Các thuộc tính của lớp là các thành phần dữ liệu, nên chúng luôn luôn là *danh từ*.
- Các hàm thành phần là các thao tác chỉ rõ hoạt động của lớp nên các hàm này là *động từ*.
- Các thuộc tính dữ liệu phải vừa đủ để mô tả khái niệm, *không dư*, không thiếu.

Các nguyên tắc xây dựng lớp

- Các thuộc tính có thể được suy diễn từ những thuộc tính khác thì dùng phương thức để thực hiện tính toán. Chu vi, diện tích tam giác, hình ellipse là thuộc tính suy diễn.

```
class CEllipse // Sai
{
    Point T;
    double rx, ry;
    double area, perimeter;
public:
};
```

```
class CEllipse // Đúng
{
    Point T;
    double rx, ry;
public:
    double getArea() const;
    double getPerimeter() const;
};
```

Các nguyên tắc xây dựng lớp

- Cá biệt có thể có một số thuộc tính suy diễn đòi hỏi nhiều tài nguyên hoặc thời gian để thực hiện tính toán, ta có thể khai báo là dữ liệu thành phần. Ví dụ tuổi thọ trung bình của một quốc gia.

```
class Country
{
    long population;
    double area;
    double avgLongevity;
    //...
public:
    double getAvgLongevity() const;
    //...
};
```

Các nguyên tắc xây dựng lớp

- Chi tiết cài đặt, bao gồm dữ liệu và phần mã hoá các hàm thành phần có thể thay đổi uyển chuyển nhưng phần giao diện, nghĩa là phần khai báo các hàm thành phần cần phải cố định để không ảnh hưởng đến người sử dụng. Tuy nhiên nên cố gắng cài đặt dữ liệu một cách tự nhiên theo đúng khái niệm.

Các nguyên tắc xây dựng lớp

- Dữ liệu thành phần nên được kết hợp thay vì phân rã.

```
// Nên
class CEllipse
{
    Point T;
    double rx, ry;
public:
    // ...
};
```

```
// Không Nên
class CEllipse
{
    double xT, yT;
    double rx, ry;
public:
    // ...
};
```

Các nguyên tắc xây dựng lớp

- Trong mọi lớp, định nghĩa một hoặc nhiều phương thức để khởi động đối tượng.
- Nên có phương thức thiết lập có khả năng tự khởi động không cần tham số.
- Nếu đối tượng tài nguyên luận lý thì phải có phương thức thiết lập, phương thức thiết lập *bản sao* để khởi động đối tượng bằng đối tượng cùng kiểu và có phương thức huỷ bỏ để dọn dẹp tài nguyên. (Ngoài ra còn phải có phép gán, chương tiếp theo, C++).

Các nguyên tắc xây dựng lớp

- Ngược lại, đối tượng đơn giản không cần tài nguyên riêng thì không cần phương thức thiết lập bản sao và cũng không cần phương thức huỷ bỏ.

Chương 2 - Đối tượng và lớp

Q&A