

## Môn LTHĐT

# Hướng dẫn thực hành tuần 2

---

### Mục đích

- Giới thiệu sơ lược về đối tượng và lớp đối tượng, cách cấp phát và giải phóng vùng nhớ trong C++, tìm hiểu một số khái niệm liên quan đến đối tượng: hàm dựng, hàm hủy.
- Tìm hiểu 3 vấn đề về con trỏ trong lập trình hướng đối tượng với hàm hủy, hàm dựng sao chép và toán tử gán.

### Nội dung

- Đối tượng và lớp đối tượng.
- Toán tử new và delete.
- Hàm dựng và hàm hủy.
- Hàm hủy và vấn đề con trỏ.
- Hàm dựng sao chép và vấn đề con trỏ.
- Toán tử gán và vấn đề con trỏ.

### Yêu cầu

Nắm vững những nội dung trình bày trong bài hướng dẫn thực hành số 1.

# 1. Đối tượng và lớp đối tượng

## Khái niệm

Đối tượng (*object*) là một thực thể phần mềm bao gồm **dữ liệu** và những **xử lý trên dữ liệu** đó. Thành phần dữ liệu được gọi là thuộc tính (*attribute*), thành phần xử lý được gọi là phương thức (*method*).

Thuộc tính ẩn chứa bên trong đối tượng mà những gì bên ngoài không thể truy xuất đến được. Phương thức là cách thức duy nhất để những gì bên ngoài đối tượng thực hiện những xử lý trên thuộc tính và thông qua những xử lý này thuộc tính của đối tượng thay đổi.

Tập hợp các đối tượng có cùng kiểu thuộc tính và phương thức tạo thành lớp đối tượng (class). Trong C++, lớp đối tượng được khai báo theo cấu trúc sau:

```
class <Tên lớp>
{
private:
    <Khai báo thành phần riêng>
public:
    <Khai báo thành phần công cộng>
};
```

Ví dụ xây dựng lớp đối tượng đơn giản:

```
class SampleClass
{
private:
    // Thuộc tính của lớp SampleClass.
    int    m_iVar1;
    float  m_fVar2;

public:
    // Phương thức của lớp SampleClass.
    void f1()
    {
        // Viết xử lý cho f1 ở đây.
    }

    void f2()
    {
        // Viết xử lý cho f2 ở đây.
    }
};
```

## Tầm vực truy cập

Tầm vực (scope) của các thành phần bên trong lớp đối tượng định nghĩa phạm vi hoạt động của những thành phần đó. Có 3 loại tầm vực dành cho các thành phần bên trong lớp đối tượng: public, private, và protected. Ở đây chúng ta tạm thời chỉ đề cập đến 2 tầm vực public và private.

Ý nghĩa của các tầm vực này như sau:

	Sử dụng bên trong lớp	Sử dụng bên ngoài lớp
Public	X	X
Private	X	

```
class SampleClass
{
private:
    int    m_iPrivateVar;

    void f1()
    {
        // Viet xu ly cho f1 o day.
    }

public:
    int    m_iPublicVar;

    void f2()
    {
        m_iPrivateVar = 1; // Thuoc tinh va phuong thuc private
        f1();              // co the truy cap trong pham vi lop.
    }
};

void main()
{
    SampleClass  obj;

    obj.m_iPrivateVar = 1; // Thuoc tinh va phuong thuc private
    obj.f1();              // khong the truy cap ngoai pham vi lop.

    obj.m_iPublic = 1;     // Thuoc tinh va phuong thuc public
    obj.f2();              // co the truy cap ngoai pham vi lop.
}
```

## Toán tử ::

Toán tử :: hay còn gọi là toán tử phân giải miền (scope resolution operator) được sử dụng để chỉ ra một cách tường minh một thuộc tính hoặc phương thức là thuộc một lớp đối tượng nào đó.

Ví dụ:

```
Employee::m_sName      // De cap den thuoc tinh m_sName cua lop Employee.
Circle::Draw()         // De cap den phuong thuc Draw() cua lop Circle.
```

Khi định nghĩa một phương thức bên ngoài lớp, chúng ta phải sử dụng toán tử phân giải miền để chỉ ra tường minh phương thức đó thuộc lớp nào.

Ví dụ hai cách định nghĩa phương thức sau là tương đương nhau:

<pre>class A { public:     int f(); };  int A::f() {     // Cai dat phuong thuc f(). }</pre>	<pre>class A { public:     int f()     {         // Cai dat phuong thuc f().     } };</pre>
--	---

Một lớp đối tượng trong C++ thường được lưu trữ trong 2 file: \*.h và \*.cpp. File \*.h chứa phần khai báo (declaration) của lớp đối tượng, trong khi file \*.cpp chứa phần định nghĩa.

## Con trỏ this

Con trỏ **this** được sử dụng trong phạm vi của lớp đối tượng. Nó đại diện cho con trỏ đang trỏ đến đối tượng gọi thực hiện phương thức đó.

Ví dụ:

```
#include "iostream.h"

class A
{
public:
    void ShowThis()
    {
        cout << this << endl;
    }
};

void main()
```

```
{
    A    obj1, obj2;

    obj1.ShowThis();
    obj2.ShowThis();
}
```

## Áp dụng

Để hiểu rõ hơn về đối tượng và lớp đối tượng, chúng ta xét ví dụ xây dựng lớp PhanSo.

**Bước 1:** vào VS, tạo project dạng Console Application (Visual C++).

**Bước 2:** thêm vào project file PhanSo.h và khai báo lớp PhanSo.

```
class PhanSo
{
private:
    int    m_iTuSo;
    int    m_iMauSo;

public:
    int LayTuSo();
    int LayMauSo();
    void DatTuSo(int iTuso);
    void DatMauSo(int iMauSo);
    PhanSo Cong(const PhanSo &a);
};
```

**Bước 3:** thêm vào project file PhanSo.cpp và định nghĩa các phương thức cho lớp PhanSo.

```
#include "iostream"
#include "PhanSo.h"
using namespace std;

int PhanSo::LayTuSo()
{
    return m_iTuSo;
}

int PhanSo::LayMauSo()
{
    return m_iMauSo;
}

void PhanSo::DatTuSo(int iTuso)
{
}
```

```

        m_iTuSo = iTuSo;
    }

    void PhanSo::DatMauSo(int iMauSo)
    {
        m_iMauSo = iMauSo;
    }

    PhanSo PhanSo::Cong(const PhanSo &a)
    {
        PhanSo      c;

        c.m_iTuSo = this->m_iTuSo * a.m_iMauSo + a.m_iTuSo * this->m_iMauSo;
        c.m_iMauSo = this->m_iMauSo * a.m_iMauSo;

        return c;
    }

```

**Bước 4:** thêm vào project file main.cpp và viết đoạn chương trình sử dụng lớp PhanSo vừa tạo.

```
#include "PhanSo.h"
```

```

void main()
{
    PhanSo      a, b, c;

    a.DatTuSo(1);
    a.DatMauSo(2);

    b.DatTuSo(1);
    b.DatMauSo(3);

    c = a.Cong(b);

    cout << "Tu so c = " << c.LayTuSo() << endl;
    cout << "Mau so c = " << c.LayMauSo() << endl;
}

```

**Bước 5:** biên dịch và chạy thử chương trình.

Trong lớp PhanSo ở trên, các thuộc tính m\_iTuSo và m\_iMauSo được khai báo trong phần “private” nên có tầm vực là private, vì vậy chúng ta không thể truy cập được chúng từ bên ngoài lớp. Trong khi đó, những phương thức được khai báo trong phần “public” nên có tầm vực là public, do đó chúng có thể được gọi từ bên ngoài lớp.

## 2. Toán tử new và delete

Trước đây trong C, việc cấp phát và giải phóng vùng nhớ được thực hiện thông qua các hàm trong thư viện “alloc.h”.

Ví dụ cấp phát vùng nhớ 10 byte dùng để chứa chuỗi ký tự:

```
unsigned char *s = (unsigned char *)malloc(10);
```

```
// ...
```

```
// ...
```

```
// ...
```

```
free(s);
```

Trong C++, việc cấp phát và giải phóng vùng nhớ được tích hợp trực tiếp vào bên trong ngôn ngữ thông qua 2 toán tử new và delete.

### **new**

Toán tử new dùng để cấp phát vùng nhớ cho một con trỏ thuộc một kiểu dữ liệu nào đó.

Ví dụ:

```
int          *p = new int[10];    // Cấp phát vùng nhớ dùng để chứa 10 số nguyên
                                     // cho con trỏ p.
unsigned char *s = new char[10];  // Cấp phát vùng nhớ dùng để chứa 10 ký tự
                                     // cho con trỏ s.
PhanSo       *a = new PhanSo;     // Cấp phát vùng nhớ dùng để chứa
                                     // đối tượng PhanSo cho con trỏ a.
```

### **delete**

Toán tử delete dùng để giải phóng vùng nhớ đã **được cấp phát bằng toán tử new** cho một con trỏ nào đó.

Ví dụ:

```
delete []p;    // Giải phóng vùng nhớ đã cấp phát cho con trỏ p.
delete []s;    // Giải phóng vùng nhớ đã cấp phát cho con trỏ s.
delete a;      // Giải phóng vùng nhớ đã cấp phát cho con trỏ a.
```

### 3. Hàm dựng và hàm hủy

#### Hàm dựng

Hàm dựng (*constructor*) của một lớp đối tượng là một phương thức đặc biệt **được tự động gọi thực hiện** khi đối tượng thuộc lớp đó được tạo lập.

##### Các tính chất của hàm dựng:

- Được tự động gọi thực hiện khi đối tượng được tạo lập.
- Không có giá trị trả về (nhưng có thể có các tham số).
- Một lớp đối tượng có thể có nhiều hàm dựng khác nhau.
- Trong C++, hàm dựng có tên trùng với tên lớp.

Bất kỳ lớp đối tượng nào cũng có hàm dựng. Trong trường hợp chúng ta không khai báo hàm dựng nào cho lớp đối tượng, hàm dựng mặc định (*default constructor*) không tham số sẽ được tự động thêm vào.

Ví dụ sau thể hiện các tính chất của hàm dựng:

```
#include "iostream"
```

```
using namespace std;
```

```
class SampleClass
```

```
{
```

```
public:
```

```
    SampleClass(int x)
```

```
    {
```

```
        cout << "Khoi tao doi tuong voi tham so la so nguyen.";
```

```
    }
```

```
    SampleClass(int x, char *s)
```

```
    {
```

```
        cout << "Khoi tao doi tuong voi tham so la so nguyen va chuoai ky tu.";
```

```
    }
```

```
};
```

```
void main()
```

```
{
```

```
    SampleClass obj1(5); SampleClass
```

```
obj2(5, "hello"); SampleClass *p = new
```

```
SampleClass(2);
```

```
}
```

#### Hàm hủy

Hàm hủy (*destructor*) của một lớp đối tượng là một phương thức đặc biệt **được tự động gọi thực hiện** khi đối tượng thuộc lớp đó bị hủy đi.



### Các tính chất của hàm hủy:

- Được tự động gọi thực hiện khi đối tượng bị hủy đi.
- Không có giá trị trả về lần tham số.
- Một lớp đối tượng chỉ có duy nhất một hàm hủy.
- Trong C++, hàm hủy có tên trùng với tên lớp và thêm dấu "~" phía trước.

Ví dụ sau thể hiện các tính chất của hàm hủy:

```
#include "iostream"
using namespace std;

class SampleClass
{
public:
    virtual ~SampleClass()
    {
        cout << "Huy bo doi tuong.";
    }
};

void main()
{
    SampleClass obj;
    SampleClass *p = new SampleClass;

    delete p;
};
```

## 4. Hàm hủy và vấn đề con trỏ

Xét ví dụ xây dựng lớp mảng các số nguyên IntegerArray.

**Bước 1:** vào VS, tạo project dạng Console Application (Visual C++).

**Bước 2:** thêm vào project file IntegerArray.h và viết khai báo lớp IntegerArray như sau:

```
class IntegerArray
{
private:
    int *m_pElement;
    int m_iLength;
public:
    IntegerArray(int iLength);
    int GetLength();
    int & ElementAt(int iIndex);
    int & operator [](int iIndex);
};
```

**Bước 3:** thêm vào project file IntegerArray.cpp và viết cài đặt lớp IntegerArray như sau:

```
#include "iostream"
#include "IntegerArray.h"
using namespace std;

IntegerArray::IntegerArray(int iLength)
{
    if (iLength < 0)
    {
        cout << "Loi: chieu dai mang la so am.";
        return;
    }
    m_iLength = iLength;
    m_pElement = new int[m_iLength];
}

int IntegerArray::GetLength()
{
    return m_iLength;
}

int & IntegerArray::ElementAt(int iIndex)
{
    if (iIndex < 0 || iIndex >= m_iLength)
        cout << "Loi: truy xuất phần tử ngoài phạm vi mảng.";
    return m_pElement[iIndex];
}

int & IntegerArray::operator [] (int iIndex)
{
    return ElementAt(iIndex);
}
```

**Bước 4:** thêm vào project file main.cpp và viết đoạn chương trình sử dụng lớp IntegerArray vừa tạo như sau:

```
#include "IntegerArray.h"
void main()
{
    IntegerArray a(3);
    a[0] = 0;
    a[1] = 1;
    a[2] = 2;

    a.ElementAt(1) = 3;
    a.ElementAt(2) = 4;

    for (int i = 0; i < a.GetLength(); i++)
        cout << a[i] << endl;
}
```

**Bước 5:** biên dịch và chạy thử chương trình.

Trong đoạn chương trình trên, sau khi kết thúc hàm `main()`, đối tượng `a` bị hủy, vùng nhớ mà `m_pElement` và `m_iLength` chiếm chỗ được giải phóng. Nhưng vùng nhớ cấp phát cho `m_pElement` lại không được giải phóng. Điều này gây ra rò rỉ bộ nhớ (*memory leak*), đặc biệt nghiêm trọng trong trường hợp vùng nhớ cấp phát cho `m_pElement` lớn. Để giải quyết vấn đề này, chúng ta phải viết lệnh giải phóng vùng nhớ cấp phát cho `m_pElement` trong hàm hủy của lớp `IntegerArray`.

Trong file `IntegerArray.h` thêm vào khai báo hàm hủy cho lớp `IntegerArray` như sau:  
`virtual ~IntegerArray();`

Trong file `IntegerArray.cpp` viết cài đặt cho hàm hủy như sau:  
`IntegerArray::~IntegerArray()`

```
{
    if (m_pElement != NULL)
        delete []m_pElement;
}
```

*Như vậy, khi lớp đối tượng có thuộc tính kiểu con trỏ và cấp phát vùng nhớ thì phải xây dựng hàm hủy cho lớp đối tượng đó để giải phóng vùng nhớ đã cấp phát.*

## 5. Hàm dựng sao chép và vấn đề con trỏ

### Khái niệm

Hàm dựng sao chép (*copy constructor*) là hàm dựng khởi tạo đối tượng dựa trên một đối tượng có cùng kiểu.

#### Các tính chất của hàm dựng sao chép:

- Hàm dựng sao chép có đầy đủ tính chất của một hàm dựng thông thường.
- Thường được dùng để tạo bản sao của đối tượng.
- Trong C++, hàm dựng sao chép của lớp `A` được khai báo như sau:

```
class A
{
public:
    A(const A &obj);
};
```

Bất kỳ lớp đối tượng nào cũng có hàm dựng sao chép. Trong trường hợp chúng ta không khai báo hàm dựng sao chép cho lớp đối tượng, hàm dựng sao chép mặc định (*default copy constructor*) sẽ được tự động được thêm vào.

### Ví dụ

Để hiểu rõ hơn về hàm dựng sao chép, chúng ta chỉnh sửa lại hàm `main()` của ví dụ lớp

IntegerArray bên trên như sau:

```
void main()
{
    IntegerArray a(3);

    a[0] = 0;
    a[1] = 1;
    a[2] = 2;

    a.ElementAt(1) = 3;
    a.ElementAt(2) = 4;

    IntegerArray b(a); // Tao doi tuong b la ban sao cua doi tuong a.

    for (int i = 0; i < b.GetLength(); i++)
        cout << b[i] << endl;
}
```

Biên dịch và chạy thử chương trình chúng ta nhận được thông báo lỗi ngay tại hàm hủy của lớp IntegerArray. **Chuyện gì đã xảy ra???**

Phân tích chương trình chúng ta thấy rằng lớp IntegerArray không khai báo hàm dựng sao chép, do đó hàm dựng sao chép mặc định sẽ được tự động thêm vào. Hàm dựng sao chép mặc định tạo bản sao từ đối tượng bằng cách sao chép từng thuộc tính của đối tượng đó, nhưng lại không sao chép vùng nhớ cấp phát cho những thuộc tính kiểu con trỏ. Như vậy m\_pElement của đối tượng b và m\_pElement của đối tượng a sẽ sử dụng chung một vùng nhớ. Khi kết thúc hàm main(), đối tượng b bị hủy trước, hàm hủy của b được gọi để hủy vùng nhớ cấp phát cho m\_pElement của b. Kế đó đối tượng a bị hủy, hàm hủy của a cũng được gọi để hủy vùng nhớ cấp phát cho m\_pElement của a. Nhưng bấy giờ vùng nhớ này đã bị hủy trước đó. Chính điều này gây ra lỗi chương trình.

Hơn nữa việc 2 mảng sử dụng chung một vùng nhớ để lưu các phần tử rõ ràng là sẽ gây ra những sai lệch về mặt ngữ nghĩa của chương trình.

Để giải quyết vấn đề này, chúng ta phải xây dựng hàm tạo sao chép cho lớp IntegerArray. Trong file IntegerArray.h, thêm vào khai báo hàm dựng sao chép cho lớp IntegerArray như sau:

```
IntegerArray(const IntegerArray &obj);
```

Trong file IntegerArray.cpp viết cài đặt cho hàm dựng sao chép

```
IntegerArray::IntegerArray(const IntegerArray &obj)
{
    m_iLength = obj.m_iLength;
    m_pElement = new int[m_iLength];

    // Sao chép vùng nhớ đã cấp phát cho m_pElement của obj.
    for (int i = 0; i < m_iLength; i++)
        m_pElement[i] = obj.m_pElement[i];
}
```

Như vậy, khi lớp đối tượng có thuộc tính kiểu con trỏ và cấp phát vùng nhớ thì phải xây dựng hàm dựng sao chép cho lớp đối tượng đó để sao chép vùng nhớ đã cấp phát.

## 6. Toán tử gán và vấn đề con trỏ

Vẫn sử dụng lớp IntegerArray xây dựng ở trên, chúng ta chỉnh sửa lại hàm main() như sau:

```
void main()
{
    IntegerArray a(3);

    a[0] = 0;
    a[1] = 1;
    a[2] = 2;

    a.ElementAt(1) = 3;
    a.ElementAt(2) = 4;

    IntegerArray b(3);

    b = a; // Tao ban sao tu doi tuong a bang toan tu gan.

    for (int i = 0; i < b.GetLength(); i++)
        cout << b[i] << endl;
}
```

Biên dịch và chạy thử chương trình chúng ta nhận được thông báo lỗi ngay tại hàm hủy của lớp IntegerArray. **Chuyện gì đã xảy ra???**

Phân tích toán tử gán chúng ta thấy rằng khi gán đối tượng x bằng đối tượng y (x, y cùng kiểu) thì tất cả các thuộc tính của x sẽ được gán giá trị bằng các thuộc tính tương ứng của y. Điều này có nghĩa là trong ví dụ trên m\_pElement của đối tượng b sẽ được gán giá trị bằng m\_pElement của đối tượng a. Như vậy m\_pElement của b và m\_pElement của a sẽ cùng trỏ đến một vùng nhớ. Đây chính là vấn đề chúng ta đã gặp phải khi đề cập đến hàm tạo sao chép ở trên.

Để giải quyết vấn đề này, chúng ta phải định nghĩa lại toán tử gán cho lớp IntegerArray. Trong file IntegerArray.h, thêm vào khai báo toán tử gán cho lớp IntegerArray như sau:

```
IntegerArray & operator =(const IntegerArray &obj);
```

Trong file IntegerArray.cpp viết cài đặt cho hàm dựng sao chép

```
IntegerArray & IntegerArray::operator =(const IntegerArray &obj)
{
    m_iLength = obj.m_iLength;
    m_pElement = new int[m_iLength];
```

```
// Sao chép vùng nhớ đã cấp phát cho m_pElement của obj.
for (int i = 0; i < m_iLength; i++)

    m_pElement[i] = obj.m_pElement[i];

return *this;
}
```

***Như vậy, khi lớp đối tượng có thuộc tính kiểu con trỏ và cấp phát vùng nhớ thì phải định nghĩa lại toán tử gán cho lớp đối tượng đó để sao chép vùng nhớ đã cấp phát.***