



Learning OOP

A. Class and Object:

1. General

```
#include <iostream>

using namespace std;

class Student {
public:
    string name;
    int age;

    void display() {
        cout << "Name: " << name << endl;
        cout << "Age: " << age << endl;
    }
};

int main() {
    Student s;
    s.name = "Dat";
    s.age = 19;
    s.display();

    return 0;
}

==>
```

Name: Dat
Age: 19

1. Lớp (class) là nơi định nghĩa thông tin về các đối tượng, có thể hiểu lớp là 1 kiểu dữ liệu mà biến của kiểu dữ liệu này được coi là một đối tượng. Như **Student** là lớp dùng để mô tả thông tin về các đối tượng học sinh.
2. Đối tượng (object) là một thể hiện của lớp, bạn có thể hiểu đối tượng là một biến. Như trong ví dụ trên biến s là một đối tượng đại diện cho thông tin của một học sinh.
3. Thuộc tính (attribute) là các đặc điểm của các đối tượng, như **name** và **age** là hai thuộc tính của đối tượng học sinh.
4. Phương thức (method) là hành vi (hành động) của đối tượng, như ví dụ trên thì display() là một phương thức để giới thiệu bản thân của học sinh.
5. Truy xuất thuộc tính của đối tượng thông qua toán tử "."

2. Access Modifier (Phạm vi truy cập):

Phạm vi truy cập	Truy cập bên trong class	Truy cập bên trong lớp con	Truy cập bên ngoài class
private	✓	✗	✗
protected	✓	✓	✗
public	✓	✓	✓

▼ Phạm vi truy cập private:

Đây là phạm vi truy cập hạn chế nhất, tất cả các thuộc tính và phương thức sẽ chỉ được truy xuất từ bên trong lớp đó. Ví dụ nếu bạn truy xuất một thuộc tính hoặc phương thức được khai báo với từ khóa **private** từ bên ngoài thì chương trình sẽ báo lỗi.

```

1  #include <iostream>
2
3  using namespace std;
4
5  class Student {
6  public:
7      string name;
8  private:
9      void display() {
10         cout << "Name: " << name;
11     }
12 };
13
14 int main() {
15     Student s;
16     s.name = "Codelearn";
17     s.display();
18     return 0;
19 }

```



⇒ *Solution*: Thay đổi phạm vi truy cập của phương thức `display()` trong lớp `Student` từ `private` → `public`.

▼ Phạm vi truy cập *protected*:

Đây là phạm vi truy cập ít hạn chế hơn `private` ở chỗ các thuộc tính và phương thức có phạm vi truy cập này có thể được truy xuất từ một *lớp con*.

▼ Phạm vi truy cập *public*:

Đây là phạm vi truy cập rộng nhất, các thuộc tính và phương thức có phạm vi truy cập này có thể được truy xuất từ bất cứ đâu.

3. Constructor (Phương thức khởi tạo):

Phương thức này tự động được gọi khi bạn khởi tạo một đối tượng (với các phương thức thông thường thì cần phải dùng toán tử `.` để gọi tới). Constructor có đặc điểm là không được định nghĩa kiểu trả về và có tên trùng với tên lớp.

Có 3 loại:

- **Default constructor:** trong trường hợp người lập trình *không định nghĩa bất kỳ constructor nào* thì trình biên dịch sẽ **tự động tạo ra một constructor mặc định**.
- **Copy constructor:** được dùng để tạo ra 1 đối tượng từ 1 đối tượng sẵn có.
- **Full parameter constructor**

▼ a. Phương thức khởi tạo mặc định - default constructor:

```
#include <iostream>

using namespace std;

class Student {
public:
    Student() {
        cout << "Constructor called";
    }
};

int main() {
    Student s;
    return 0;
}

==> Constructor called
```

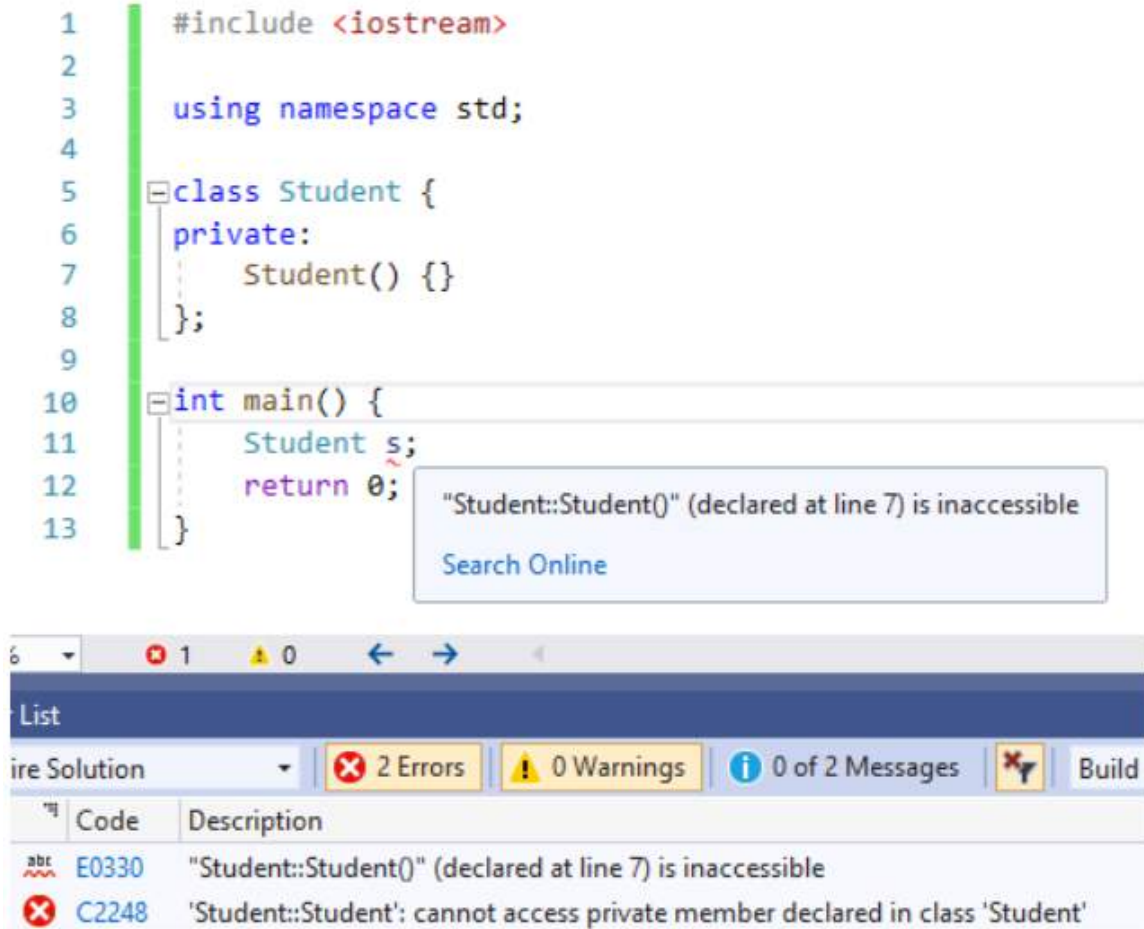
Nếu một lớp không được khai báo constructor

```
class Student {
};
```

thì chương trình sẽ tự động thêm vào một constructor mặc định (constructor rỗng) với phạm vi truy cập public bên trong lớp đó

```
class Student {
public:
    Student() {}
};
```

LƯU Ý: Nếu bạn dùng **private** hoặc **protected** để khai báo constructor thì bạn sẽ không thể khởi tạo được đối tượng của lớp này.



▼ b. Phương thức khởi tạo sao chép - copy constructor:

- Cho phép khởi tạo một đối tượng của một lớp dựa trên việc sao chép một đối tượng sẵn có
- Được tự động sinh ra cho lớp và giúp sao chép từng thành phần thuộc tính (*member-wise copy*).
- Do quá trình sao chép là sao chép từng thành phần theo dạng từng bit (*bitwise copy*), nếu các thành phần dữ liệu là **các kiểu con trỏ** thì quá trình sao chép sẽ chỉ **sao chép địa chỉ của con trỏ chứ không thật sự sao chép vùng nhớ mà chúng quản lý** (*shallow copy*)

```
class Student
{
private:
    char* studentId;
    double gpa;
    char* fullname;
public:
    ...
    Student(const Student& s) {
```

```

// SHALLOW COPY
this->studentId = s.studentId;
this->gpa = s.gpa;
this->fullname = s.fullname;

// DEEP COPY
this->studentId = new char[strlen(s.studentId) + 1];
strcpy(this->studentId, s.studentId);
this->gpa = s.gpa;
this->fullname = new char[strlen(s.fullname) + 1];
strcpy(this->fullname, s.fullname)
}
};

```

Thành phần dữ liệu của lớp đối tượng có các biến con trỏ và chúng đang quản lý các vùng nhớ được cấp phát động, nếu có nhu cầu sao chép → *viết lại copy constructor*.

```
<ClassName>(const <ClassName>&)
```

▼ c. Phương thức khởi tạo có tham số - full parameter constructor:

```

#include <iostream>

using namespace std;

class Student {
private:
    string name;
    int age;
public:
    Student(const string& name, const int& age) {
        this->name = name;
        this->age = age;
    }

    void display() {
        cout << "Name: " << name << endl;
        cout << "Age: " << age << endl;
    }
};

int main() {
    Student s("Dat", 19);
    s.display();
    return 0;
}

==>

```

Name: Dat
Age: 19

4. Destructor (Hàm hủy):

Giống với constructor, destructor hay phương thức hủy (hàm hủy) cũng là một phương thức đặc biệt và **được chương trình tự động gọi tới**. Destructor thường được sử dụng để giải phóng bộ nhớ khi không dùng tới.

```
class Array {  
public:  
    int* arr;  
    int n;  
};
```

Array là lớp lưu thông tin về mảng các số nguyên được cấp phát động → cần phải giải phóng sau khi sử dụng đối tượng thuộc lớp này. Ví dụ nếu Array được sử dụng ở 100 năm thì phải gọi tới toán tử delete 100 lần → quá rườm rà, code bị lặp lại ⇒ Sử dụng destructor

```
#include <iostream>  
  
using namespace std;  
  
class Array {  
public:  
    int* arr;  
    int n;  
  
    ~Array() {  
        cout << "Destructor called" << endl;  
        delete[] arr;  
    }  
};  
  
void someFunc1() {  
    Array a;  
    a.n = 4;  
    a.arr = new int[a.n];  
    // do sth  
}  
  
void someFunc2() {  
    Array a;
```

```

    a.n = 4;
    a.arr = new int[a.n];
    // do sth
}

int main() {
    someFunc1();
    someFunc2();

    return 0;
}

==>
    Destructor called
    Destructor called

```

Phương thức `~Array()` được gọi, đồng nghĩa với việc bộ nhớ được cấp phát động đã được tự động giải phóng. Destructor của một đối tượng được gọi tới khi chương trình thoát khỏi phạm vi của nó, cụ thể hơn là 4 trường hợp:

- Khi kết thúc hàm chứa đối tượng đó.
- Khi kết thúc chương trình.
- Khi gọi tới toán tử delete.
- Khi thoát khỏi block code chứa đối tượng đó.

Constructor: Lớp cha trước, lớp con sau

Destructor: Lớp con trước, lớp cha sau

B. Static Variables and Methods:

1. Static:

Static Variables:

Biến tĩnh là biến được khai báo với từ khóa `static` và có đặc điểm là giá trị của nó được chia sẻ bởi tất cả các đối tượng trong chương trình.

```

#include <iostream>

using namespace std;

```



```

class Student {
private:
    string name;
    int age;
public:
    int numberOfStudents = 0;
    Student(const string& name, const int& age) {
        this->name = name;
        this->age = age;
    }ZX
};

int main() {
    Student s1("Dat", 19);
    Student s2("Tora", 19);
    s1.numberOfStudents = 2;
    s2.numberOfStudents = 3;
    cout << s1.numberOfStudents << " " << s2.numberOfStudents;
    return 0;
}

==>
    2 3

```

Có thể thấy giá trị của biến **numberOfStudents** ở hai đối tượng **s1** và **s2** là khác do đây là thuộc tính riêng của các đối tượng. Trong trường hợp trên nếu biến **numberOfStudents** là biến **static**:

```

#include <iostream>

using namespace std;

class Student {
private:
    string name;
    int age;
public:
    static int numberOfStudents;
    Student(const string& name, const int& age) {
        this->name = name;
        this->age = age;
    }
};

int Student::numberOfStudents = 0;

int main() {
    Student s1("Dat", 19);
    Student s2("Tora", 19);
    s1.numberOfStudents = 2;
    s2.numberOfStudents = 3;
    cout << s1.numberOfStudents << "\n" << s2.numberOfStudents;
    return 0;
}

```

```
=>
    3
    3
```

Có thể thấy giá trị của biến `numberOfStudents` ở hai đối tượng `s1` và `s2` là như nhau.

⇒ Do biến **static** có đặc điểm là **biến dùng chung** và **không thuộc một đối tượng nào trong chương trình** nên bạn **không thể khởi tạo giá trị** cho nó bên trong **constructor** của một lớp, thay vào đó **cần khởi tạo từ bên ngoài** (**int**

Student::numberOfStudents = 0;)

Các biến **static** thường được dùng để lưu các giá trị chung của các đối tượng hoặc các hằng số trong chương trình.

LƯU Ý: Các biến static không được coi là thuộc tính của một lớp, biến static chỉ có thể được truy cập bởi phương thức static.

Static Methods

Giống với biến tĩnh, phương thức tĩnh cũng được khai báo với từ khóa `static` và được sử dụng mà không cần phải khởi tạo đối tượng.

```
#include <iostream>

using namespace std;

class Math {
public:
    static int max(const int& a, const int& b) {
        return a > b ? a : b;
    }

    static int min(const int& a, const int& b) {
        return a < b ? a : b;
    }
};

int main() {
    cout << Math::max(2, 3) << endl;
    cout << Math::min(2, 3) << endl;

    return 0;
}
```

```
==>
```

```
3
```

```
2
```

Một số đặc điểm của phương thức tĩnh:

- Phương thức tĩnh có thể được gọi mà không cần phải khởi tạo đối tượng.
- Do không thuộc một đối tượng nào nên trong cùng một lớp, phương thức tĩnh chỉ có thể gọi tới phương thức tĩnh khác, không thể gọi tới phương thức không phải là static.
- Trong cùng một lớp, phương thức tĩnh không thể gọi tới các biến không phải là biến tĩnh.

2. Overloading:

Nạp chồng phương thức (overload method)

Nếu một lớp có nhiều phương thức cùng tên nhưng khác nhau về kiểu dữ liệu hoặc số các tham số, thì đó là nạp chồng phương thức.

```
#include <iostream>

using namespace std;

class Math {
public:
    static int add(const int& a, const int& b) {
        return a + b;
    }

    static double add(const double& a, const double& b) {
        return a + b;
    }
};

int main() {
    cout << Math::add(2, 3) << endl;
    cout << Math::add(2.5, 4.5) << endl;

    return 0;
}

==>
5
7
```

Ngoài ra có thể nạp chồng phương thức bằng cách thay đổi tham số:

```
static int max(const int& a, const int& b) {  
    return a > b ? a : b;  
}  
  
static int max(const int& a, const int& b, const int& c) {  
    if(a >= b && a >= c) return a;  
    if(b >= c) return b;  
    return c;  
}
```

Ngoài ra các phương thức nạp chồng còn có thể gọi tới nhau:

```
static int max(const int& a, const int& b) {  
    return a > b ? a : b;  
}  
  
static int max(const int& a, const int& b, const int& c) {  
    return max(max(a, b), c);  
}
```

⇒ Nạp chồng phương thức giúp tránh được việc tạo ra các phương thức với tên gọi khác nhau, giúp cho code trở nên gọn gàng, dễ đọc hơn.

Nạp chồng toán tử (overload operator)

Là định nghĩa lại toán tử đã có trên kiểu dữ liệu người dùng tự định nghĩa để dễ dàng thể hiện các câu lệnh trong chương trình

```
class Fraction {  
    ...  
};  
  
Fraction f1(1, 2);  
Fraction f2(2, 3);  
Fraction res;  
  
// use function  
res = f1.add(f2);  
  
// use overload operator  
res = f1 + f2;
```

Các toán tử có thể được overload

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

Bảng các toán tử có thể overload được.

Vậy ta chỉ có một số toán tử sau không overload được:

- Toán tử . (chấm)
- Toán tử phạm vi ::
- Toán tử điều kiện ?:
- Toán tử sizeof

Một số lưu ý:

- Các toán tử một ngôi (-, ++) có thể đứng trước hoặc sau toán hạng
- Một số toán tử có thể làm toán tử một ngôi hoặc hai ngôi như toán tử *
- Toán tử chỉ mục ([...]) là toán tử hai ngôi
- Các từ khóa new và delete cũng được xem là toán tử nên có thể được overload

Có 2 loại là:

- **Hàm cục bộ (dùng phương thức của lớp)**

Tham số đầu tiên mặc định chính là toán hạng đầu tiên, đối với toán tử hai ngôi, ta chỉ cần truyền một tham số cho hàm, chính là toán hạng thứ hai.

```
class Fraction {
private:
    int nume;
    int deno;
public:
    Fraction() : nume(1), deno(2) {}

    Fraction operator + (const Fraction& f) {
        Fraction res;
        res.nume = this->nume * f.deno + this->deno * f.nume;
        res.deno = this->deno * f.deno;
        return res;
    }
};
```

```

    }
};

Fraction f1(1, 2);
Fraction f2(2, 3);
Fraction res = f1 + f2; // <=> res = f1.add(f2);

```

Do toán tử overload theo cách này là phương thức, được gọi từ một đối tượng, do đó **mặc định toán hạng đầu tiên phải là toán hạng có kiểu dữ liệu của lớp đó**, điều này cũng có nghĩa là bạn phải **đặt toán hạng có kiểu dữ liệu của lớp đó đầu tiên rồi mới đến toán hạng tiếp theo**.

Ví dụ:

```

Fraction res = f1 + f2; // DUNG <=> res = f1.add(f2)
Fraction res = f2 + f1; // SAI

```

Và đối với các kiểu dữ liệu có sẵn, ta không thể truy cập vào các lớp định nghĩa nên chúng, do đó ta không thể overload operator của chúng được → Ta sẽ sử dụng hàm toàn cục.

- **Hàm toàn cục (dùng hàm friend)**

Dùng hàm friend để có thể tự do lựa chọn thứ tự của toán hạng.

```

class Fraction {
    friend Fraction operator + (const Fraction& f1, const Fraction& f2);
};

Fraction operator + (const Fraction& f1, const Fraction& f2) {
    Fraction res;
    res.nume = f1.nume * f2.deno + f1.deno * f2.nume;
    res.deno = f1.deno * f2.deno;

    return res;
}

```

Muốn thay đổi thứ tự toán hạng → thay đổi thứ tự tham số

VÍ DỤ:

```

class Fraction
{
    int nume;
    int deno;
}

```

```

public:
    // default constructor
    Fraction() {
        this->nume = 0;
        this->deno = 1;
    }
    // copy constructor
    Fraction(const Fraction& obj) {
        nume = obj.nume;
        deno = obj.deno;
    }
    // full parameter constructor
    Fraction(int nume, int deno) {
        if (deno == 0) {
            Fraction();
        }

        if (deno < 0) {
            this->nume = -nume;
            this->deno = -deno;
        }
        else {
            this->nume = nume;
            this->deno = deno;
        }
    }
    // destructor
    // ~Fraction();

    void setNume(int nume);
    void setDeno(int deno);
    int getNume() {
        return nume;
    }
    int getDeno() {
        return deno;
    }
    int gcd(int a, int b);

    friend istream& operator >> (istream& in, Fraction& f);
    friend ostream& operator << (ostream& out, const Fraction& f);

    bool operator == (const Fraction& f);
    bool operator != (const Fraction& f);
    bool operator >= (const Fraction& f);
    bool operator <= (const Fraction& f);
    bool operator > (const Fraction& f);
    bool operator < (const Fraction& f);

    // += , -=, *=, /=, ++, --
    Fraction& operator+=(const Fraction& f);
    Fraction& operator-=(const Fraction& f);
    Fraction& operator*=(const Fraction& f);
    Fraction& operator/=(const Fraction& f);
    Fraction& operator++();
    Fraction operator++(int);
    Fraction& operator--();
    Fraction operator--(int);

```

```

// assignment operator
Fraction& operator = (const Fraction& f);
// fraction + fraction
Fraction operator + (const Fraction& f);
// fraction + int
Fraction operator + (const int& x);
// int + fraction
friend Fraction operator + (const int& x, const Fraction& f);
// fraction - fraction
Fraction operator - (const Fraction& f);
// fraction - int
Fraction operator - (const int& x);
// fraction * int
Fraction operator * (const Fraction& f);
// int * fraction
friend Fraction operator * (const int& x, const Fraction& f);
// fraction / int
Fraction operator / (const Fraction& f);

// Change fraction to integer
operator int() const {
    return nume / deno;
}

// Change fraction to float
operator float() const {
    return (float)nume / deno;
}
};

```

NẠP CHỒNG TOÁN TỬ NHẬP XUẤT

Để overload, ta sử dụng hàm toàn cục có hai tham số, tham số đầu tiên là một tham chiếu đến đối tượng kiểu istream (để nhập) hoặc ostream (để xuất), tham số thứ hai là tham chiếu đối tượng cần nhập, xuất, kiểu trả về của hàm chính là tham chiếu đến tham số đầu tiên của hàm (istream hoặc ostream).

- **Toán tử nhập:**

```

class Fraction {
private:
    friend istream& operator >> (istream& in, Fraction& f);
};

istream& operator >> (istream& in, Fraction& f) {
    cout << "Nume: ";
    in >> f.nume;
    cout << "Deno: ";
    in >> f.denp;

    return in;
}

```



```

}

Fraction f;
// thay vì gọi hàm
f.getInfo();

// sử dụng nạp chồng toán tử
cin >> f;

```

- **Toán tử xuất:**

```

class Fraction {
    friend ostream& operator << (ostream& out, const Fraction& f);
};

ostream& operator << (ostream& out, const Fraction& f) {
    if(f.nume == 0) {
        out << 0;
        return out;
    }
    if(f.deno == 1) {
        out << nume;
        return out;
    }
    out << f.nume << "/" << f.deno;
    return out;
}

Fraction f(3, 2);
cout << f;

```

Hạn chế của việc nạp chồng toán tử:

- Không thể tạo toán tử mới
- Không thể kết hợp các toán tử theo cách mà trước đó không được định nghĩa
- Không thay đổi được thứ tự ưu tiên toán tử
- Không thể tạo cú pháp mới cho toán tử
- Không thể định nghĩa lại một định nghĩa đã có của một toán tử

Một số ràng buộc của toán tử:

- Hầu hết các toán tử không ràng buộc ý nghĩa, ngoại trừ một số toán tử =, [], (), → thì được định nghĩa là hàm thành phần của lớp để toán hạng đầu tiên luôn nằm bên trái.

- Nếu đã overload toán tử rồi thì phải làm đầy đủ. Ví dụ overload +, -, *, / thì overload luôn cả +=, -=, *=, /=...
- Luôn tôn trọng ý nghĩa của toán tử gốc
- Cố gắng tái sử dụng mã nguồn một cách tối đa (ví dụ như đảo thứ tự toán hạng)

C. Encapsulation, Inheritance, Polymorphism, Abstraction:

1. Encapsulation (Tính đóng gói):

Là kỹ thuật giúp che giấu được những thông tin bên trong đối tượng → Giúp hạn chế các lỗi khi phát triển chương trình.

```
class Student {
private:
    string name;
    int age;
    double gpa;
public:
    string getName() {
        return name;
    }

    void setName(const string& name) {
        this->name = name;
    }

    int getAge() {
        return age;
    }

    void setAge(const int& age) {
        this->age = age;
    }

    double getGpa() {
        return gpa;
    }

    void setGpa(const double& gpa) {
        this->gpa = gpa;
    }
};
```

Để đảm bảo tính đóng gói → cài đặt phạm vi truy cập của các thuộc tính là **private** và truy xuất tới các thuộc tính này thông qua các phương thức **public** (gọi là các *setter, getter*)

Các lợi ích chính mà tính đóng gói đem lại:

- Hạn chế được các truy xuất không hợp lệ tới các thuộc tính của đối tượng.
- Giúp cho trạng thái của các đối tượng luôn đúng.
- Giúp ẩn đi những thông tin không cần thiết về đối tượng.
- Cho phép thay đổi cấu trúc bên trong lớp mà không ảnh hưởng tới lớp khác.

2. Inheritance (Tính kế thừa):

Kế thừa trong OOP chính là thừa hưởng lại những thuộc tính và phương thức của một lớp. Tính kế thừa giúp:

- code không bị lặp lại
- tái sử dụng lại code
- tăng khả năng mở rộng của chương trình

```
#include <iostream>

using namespace std;

class Person {
private:
    string name;
    int age;
public:
    string getName() {
        return name;
    }

    void setName(const string& name) {
        this->name = name;
    }

    int getAge() {
        return age;
    }

    void setAge(const int& age) {
        this->age = age;
    }
};
```

```

class Student : public Person {
private:
    double gpa;
public:
    double getGpa() {
        return gpa;
    }

    void setGpa(const double& gpa) {
        this->gpa = gpa;
    }
};

int main() {
    Student s;
    s.setName("Dat");
    s.setAge(19);
    s.setGpa(9.5);
    cout << "Name: " << s.getName() << endl;
    cout << "Age: " << s.getAge() << endl;
    cout << "Gpa: " << s.getGpa() << endl;

    return 0;
}

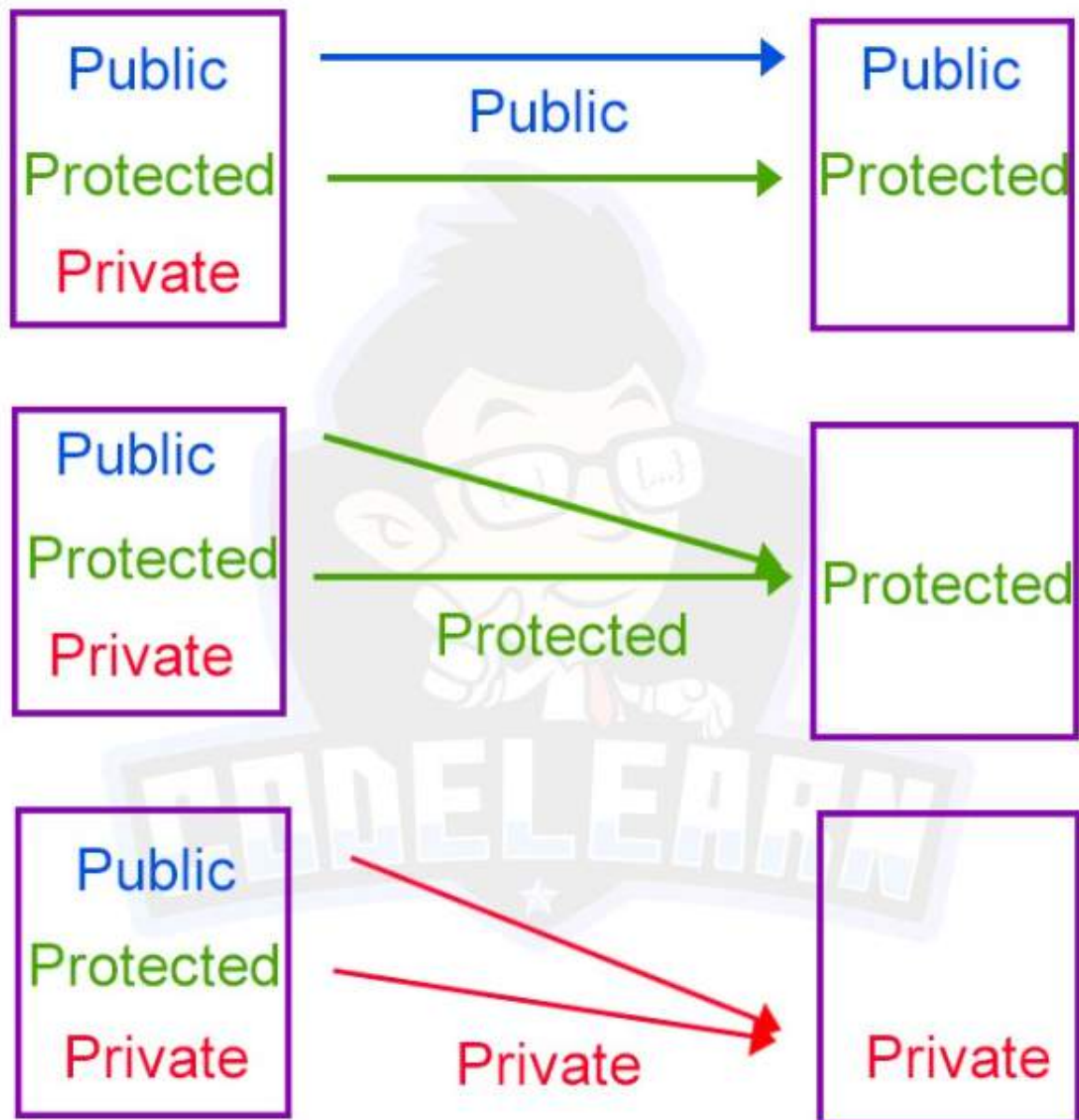
==>
    Name: Dat
    Age: 19
    Gpa: 9.5

```

LƯU Ý: Các thuộc tính và phương thức có phạm vi truy cập private sẽ không thể được truy cập từ lớp con và khi làm thực tế nên tách lớp cha và lớp con ra từng file riêng.

a. Thay đổi phạm vi truy cập của thuộc tính và phương thức được kế thừa:

C++ cho phép thay đổi phạm vi truy cập của các thuộc tính và phương thức được kế thừa bằng cách kế thừa với từ khóa **protected**, **private**.



b. Kế thừa phương thức khởi tạo:

1. Lớp con **không được thừa hưởng** các thuộc tính và phương thức **private** từ lớp cha:

Các lớp con chỉ có thể thừa hưởng được các thuộc tính và phương thức có phạm vi truy cập **public**, **protected**. Nếu bạn truy xuất tới thuộc tính **private** của lớp cha từ lớp con thì chương trình sẽ báo lỗi


```

using namespace std;

class Person {
private:
    string name;
public:
    void setName(string name) {
        this->name = name;
    }
    string getName() {
        return name;
    }
};

class Student : public Person {
public:
    Student() {
        name = "Default";
    }
};

```



 (field) std::string Person::name
[Search Online](#)
 member "Person::name" (declared at line 7) is inaccessible
[Search Online](#)

Muốn truy xuất được tới các thuộc tính private của lớp cha thì phải thông qua các setter và getter của lớp cha

```

#include<iostream>

using namespace std;

class Person {
private:
    string name;
public:
    void setName(string name) {
        this->name = name;
    }
    string getName() {
        return name;
    }
};

class Student : public Person {
public:
    Student() {
        setName("Default");
    }
};

```

2. Constructor của lớp con luôn gọi tới constructor của lớp cha:

```

#include <iostream>

using namespace std;

class Person {
public:
    Person() {
        cout << "Person constructor" << endl;
    }
};

class Student : public Person {
public:
    Student() {
        cout << "Student constructor" << endl;
    }
};

int main() {
    Student s;
}

```

```

    return 0;
}

==>
    Person constructor
    Student constructor

```

Constructor mặc định (constructor không tham số) của lớp cha đã được gọi cùng với constructor của lớp con. Do đó, khi lớp cha không có constructor mặc định mà lớp con không chỉ rõ cần gọi tới constructor nào của lớp cha thì chương trình sẽ báo lỗi

c. Ghi đè phương thức:

Trong kế thừa, khi lớp con khai báo phương thức có tên trùng với phương thức ở lớp cha thì phương thức của lớp cha sẽ bị thay thế bởi phương thức ở lớp con.

```

#include <iostream>

using namespace std;

class SuperClass {
public:
    void display() {
        cout << "SuperClass" << endl;
    }
};

class SubClas : public SuperClass {
public:
    void display() {
        cout << "SubClass" << endl;
    }
};

int main() {
    SubClass s;
    s.display();
    return 0;
}

==>
    SubClass

```

Trong trường hợp phương thức của lớp cha bị ghi đè thì bạn vẫn có thể gọi tới nó bằng toán tử "::":

```

class SubClas : public SuperClass {
public:

```



```

void display() {
    SuperClass::display();
    cout << "SubClass" << endl;
}
};

==>
SuperClass
SubClass

```

d. Up-casting và Down-casting:

1. Up-casting:

Khi biến con trở của lớp cha trở tới đối tượng của lớp con.

```

#include <iostream>

using namespace std;

class Animal {
public:
    void sound() {
        cout << "some sound" << '\n';
    }
};

class Cat : public Animal {
public:
    void sound() {
        cout << "meow meow" << '\n';
    }
};

class Dog : public Animal {
public:
    void sound() {
        cout << "woof woof" << '\n';
    }
};

int main() {

    Cat cat;
    Animal* animal_01 = &cat;
    // khi biến của lớp cha tham chiếu tới đối tượng của lớp con
    // biến này chỉ có thể gọi tới các thuộc tính có ở lớp cha
    animal_01->sound();

    Dog dog;
    Animal* animal_02 = &dog;
    animal_02->sound();
}

```

```

    return 0;
}

==>
    some sound
    some sound

```

Đối tượng thuộc lớp Cat và lớp Dog đã được gán cho biến con trỏ thuộc lớp Animal (đối tượng lớp con bị chuyển kiểu về đối tượng thuộc lớp cha) → **up-casting**.

*LƯU Ý: Khi biến của lớp cha tham chiếu tới đối tượng của lớp con thì biến này chỉ có thể gọi tới các thuộc tính và phương thức **có ở lớp cha**.*

2. Down-casting:

Ngược với up-casting là chuyển từ lớp con sang trỏ lớp cha, thì down-casting chuyển từ lớp cha sang con trỏ lớp con.

Down-casting có đặc điểm là có thể gọi tới các phương thức **chỉ có ở lớp con** mà không có ở lớp cha.

```

#include <iostream>

using namespace std;

class Animal {
public:
    void sound() {
        cout << "some sound" << '\n';
    }
};

class Dog : public Animal {
public:
    void sound() {
        cout << "woof woof" << '\n';
    }

    void play() {
        cout << "The dog is playing" << endl;
    }
};

int main() {
    Animal animal;
    Dog* d = (Dog*)&animal;
    // có thể gọi được các thuộc tính có ở lớp con nhưng ko có ở lớp cha
    // => lý do phải ép kiểu

```

```

        d->sound();
        d->play();
        return 0;
    }

==>
    woof woof
    The dog is playing

```

e. Virtual function:

```

#include <iostream>

using namespace std;

class Animal {
public:
    void sound() {
        cout << "some sound" << '\n';
    }
};

class Cat : public Animal {
public:
    void sound() {
        cout << "meow meow" << '\n';
    }
};

class Dog : public Animal {
public:
    void sound() {
        cout << "woof woof" << '\n';
    }
};

int main() {

    Cat cat;
    Animal* animal_01 = &cat;
    // khi biến của lớp cha tham chiếu tới đối tượng của lớp con
    // biến này chỉ có thể gọi tới các thuộc tính có ở lớp cha
    animal_01->sound();

    Dog dog;
    Animal* animal_02 = &dog;
    animal_02->sound();

    return 0;
}

==>

```

```
some sound
some sound
```

Tuy `animal_01` và `animal_02` trỏ tới đối tượng `cat` và `dog` nhưng phương thức được gọi lại là nằm trong lớp `Animal`. Do khi biên dịch, compiler không thể xác định được đối tượng mà con trỏ `animal` đang trỏ tới là đối tượng thuộc lớp nào, dẫn tới mặc định các phương thức được gọi sẽ là phương thức ở lớp cha. Để gọi tới các phương thức ở lớp con thì cần thêm từ khóa **virtual** vào phương thức ở lớp cha để **chỉ rõ cho compiler biết rằng phương thức cần được gọi** sẽ được xác định **tại thời điểm runtime**.

```
#include <iostream>

using namespace std;

class Animal {
public:
    virtual void sound() {
        cout << "some sound" << '\n';
    }
};

class Cat : public Animal {
public:
    void sound() {
        cout << "meow meow" << '\n';
    }
};

class Dog : public Animal {
public:
    void sound() {
        cout << "woof woof" << '\n';
    }
};

int main() {

    Cat cat;
    Animal* animal_01 = &cat;
    // khi biến của lớp cha tham chiếu tới đối tượng của lớp con
    // biến này chỉ có thể gọi tới các thuộc tính có ở lớp cha
    animal_01->sound();

    Dog dog;
    Animal* animal_02 = &dog;
    animal_02->sound();

    return 0;
}
```

```
==>
    meow meow
    woof woof
```

f. Virtual destructor:

```
#include <iostream>

using namespace std;

class Base {
public:
    ~Base() {
        cout << "Base's destructor" << endl;
    }
};

class Derived : public Base {
public:
    ~Derived() {
        cout << "Derived's destructor" << endl;
    }
};

int main() {
    Base* n = new Derived();
    delete n;
    return 0;
}

==>
    Base's destructor
```

Nếu không có từ khóa **virtual** thì phương thức được gọi tới sẽ luôn là phương thức ở lớp cha → Khi gọi toán tử **delete** thì destructor được gọi sẽ là destructor ở lớp cha ⇒ Destructor của lớp con không được gọi tới → dẫn đến nguy hiểm do tài nguyên của đối tượng thuộc lớp con đã không được giải phóng → rò rỉ bộ nhớ ⇒ Khai báo destructor của lớp cha với từ khóa **virtual**.

```
#include <iostream>

using namespace std;

class Base {
public:
    virtual ~Base() {
```

```

        cout << "Base's destructor" << endl;
    }
};

class Derived : public Base {
public:
    ~Derived() {
        cout << "Derived's destructor" << endl;
    }
};

int main() {
    Base* n = new Derived();
    delete n;
    return 0;
}

==>
    Derived's destructor
    Base's destructor

```

3. Polymorphism and Abstraction:

a. Polymorphism:

- Đa hình với nạp chồng phương thức
- Đa hình với ghi đè phương thức
- Đa hình thông qua các đối tượng đa hình (polymorphic objects: up-casting)

Variable Hiding

Variable Hiding xảy ra khi lớp con khai báo thuộc tính có tên giống tên thuộc tính ở lớp cha, lúc này thuộc tính của lớp cha sẽ **không bị lớp con ghi đè mà bị lớp con ẩn đi**.

```

#include <iostream>

using namespace std;

class SuperClass {
public:
    int x = 10;

    virtual void display() {
        cout << x << endl;
    }
}

```

```

    }
};

class SubClass : public SuperClass {
public:
    int x = 20;

    void display() {
        cout << x << endl;
    }
};

int main() {
    SuperClass* s = new SubClass();
    cout << s->x << endl;
    s->display();
    return 0;
}

==>
    10
    20

```

b. Abstraction:

Mục tiêu chính của tính trừu tượng là ẩn đi những thông tin không cần thiết để chỉ tập trung vào những tính năng của đối tượng. Tính chất này giúp ta trọng tâm hơn vào những tính năng thay vì phải quan tâm tới cách mà nó được thực hiện.

Lớp trừu tượng (abstract class):

1. Lớp trừu tượng là lớp có ít nhất một phương thức trừu tượng. Phương thức trừu tượng là các phương thức **virtual** được gán giá trị **= 0**

```

class Animal {
public:
    virtual void sound() = 0;
};

```

2. Không thể khởi tạo được đối tượng của lớp trừu tượng mà chỉ có thể khởi tạo được đối tượng của lớp con

```

#include <iostream>

using namespace std;

class Animal {
public:
    virtual void sound() = 0;
};

```

```

class Cat : public Animal {
public:
    void sound() {
        cout << "meow meow" << endl;
    }
};

int main() {
    Animal* a = new Cat();
    a->sound();
    return 0;
}

==>
    meow meow

```

Chương trình không báo lỗi và chạy được. Nhưng nếu thay **Animal* a = new Cat()** thành **Animal a;** thì chương trình sẽ báo lỗi → *do không thể khởi tạo được đối tượng của lớp trừu tượng.*

3. Nếu một lớp được kế thừa từ lớp trừu tượng thì lớp đó phải ghi đè tất cả các phương thức trừu tượng

D. Relationships between objects:

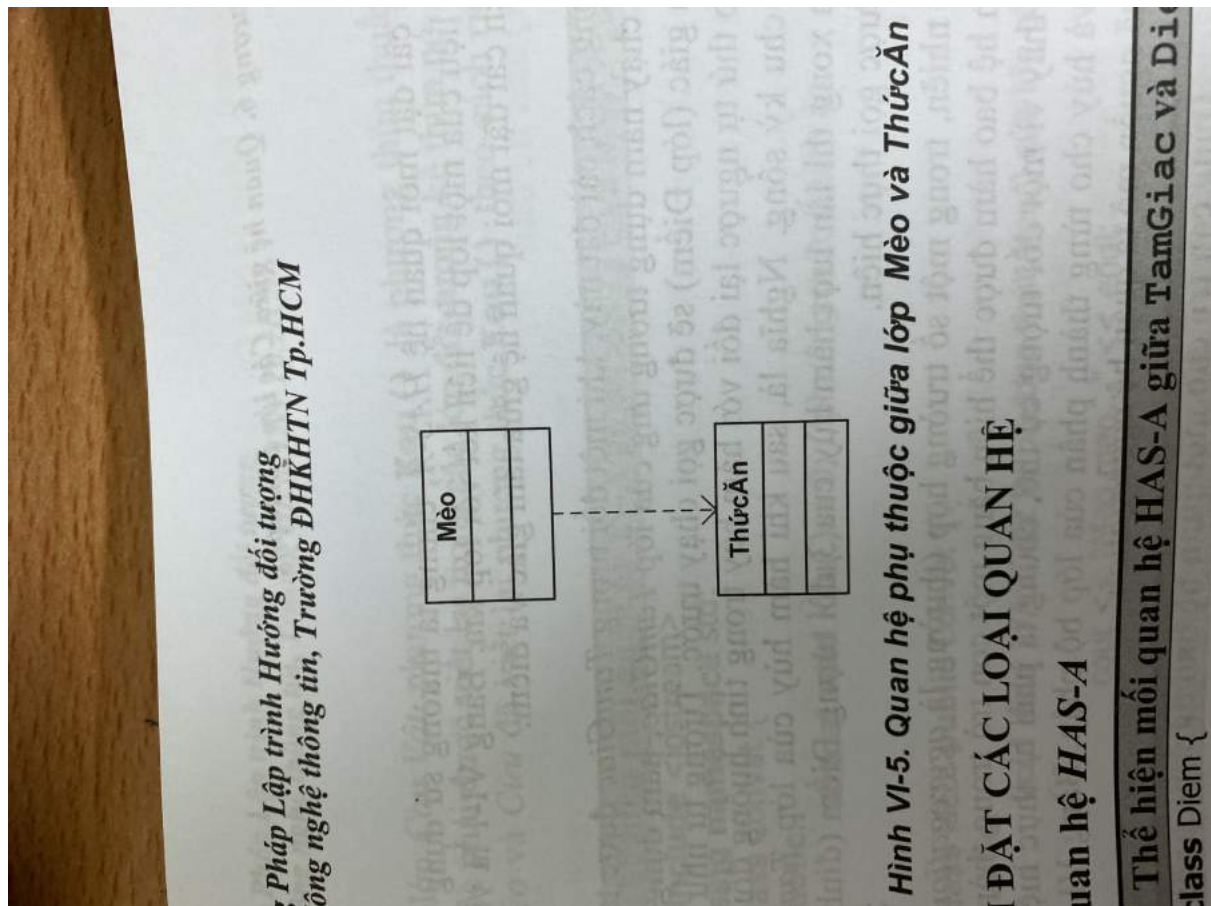
1. Quan hệ phụ thuộc:

Là quan hệ mà đối tượng của lớp này sử dụng đối tượng của lớp kia. Nếu trong **ClassA** có **sử dụng đối tượng** của **ClassB** (có thể là các thuộc tính, các tham số, các biến cục bộ,...) thì lớp **ClassA** có quan hệ phụ thuộc với lớp **ClassB**.

Quan hệ phụ thuộc cho biết khi có sự thay đổi trên lớp chính thì có thể gây ra sự thay đổi trên lớp phụ thuộc. Tuy nhiên, lớp chính hoàn toàn độc lập đối với các thay đổi từ lớp phụ thuộc.

Quan hệ phụ thuộc thường xuất hiện trong các trường hợp sau:

- Lớp phụ thuộc sử dụng một biến toàn cục là lớp chính.
- Lớp phụ thuộc sử dụng lớp chính là một tham số trong các hành động/phương thức của nó.
- Lớp phụ thuộc sử dụng biến cục bộ là lớp chính.
- Lớp phụ thuộc gửi thông điệp đến cho lớp chính.



```

class ClassA {
public:
    void myMethod(ClassB b) {
        b.doSth();
    }
};
  
```

Trong trường hợp trên, ClassA sẽ phụ thuộc vào lớp ClassB, nếu ClassB thay đổi thì ClassA sẽ phải thay đổi theo

2. Quan hệ kết hợp (Association: HAS-A, PART-OF):

Quan hệ kết hợp xảy ra khi một đối tượng có thuộc tính là một đối tượng khác.

Lớp A có chứa đối tượng của lớp B (hay nói cách khác, lớp B là một bộ phận của lớp A).

```

class Manager {
private:
    string name;
  
```

```

public:
    Manager() {
    }
    Manager(const string& name) {
        this->name = name;
    }
    string getName() {
        return name;
    }
};

class Employee {
private:
    string name;
    Manager manager;
public:
    Employee(const string& name, const Manager& manager) {
        this->name = name;
        this->manager = manager;
    }
    string getManagerName() {
        return manager.getName();
    }
    string getName() {
        return name;
    }
};

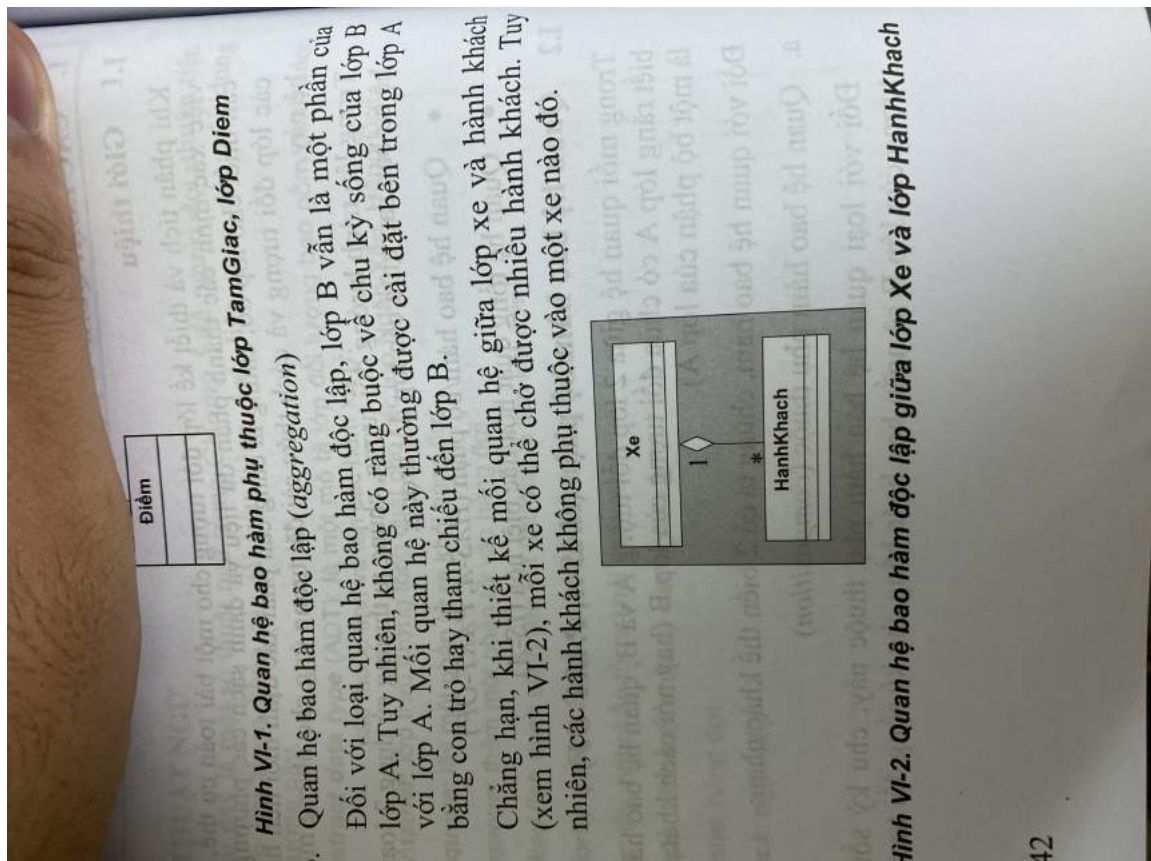
```

Lớp **Employee** có thuộc tính là đối tượng thuộc lớp **Manager**, do đó hai lớp này có quan hệ kết hợp. Quan hệ kết hợp được chia làm 2 loại: *quan hệ bao hàm độc lập (Aggregation)* và *quan hệ hợp thành (Composition)*.

a. **Quan hệ bao hàm độc lập (Aggregation):**

Quan hệ bao hàm độc lập xảy ra khi một đối tượng có thuộc tính là một đối tượng khác và **2 đối tượng này có thể tồn tại độc lập**.

Lớp B vẫn là một phần của lớp A. Tuy nhiên, không có ràng buộc về chu kỳ sống của lớp B với lớp A. Mỗi quan hệ này ***thường được cài đặt bên trong lớp A bằng con trỏ tham chiếu đến lớp B***.



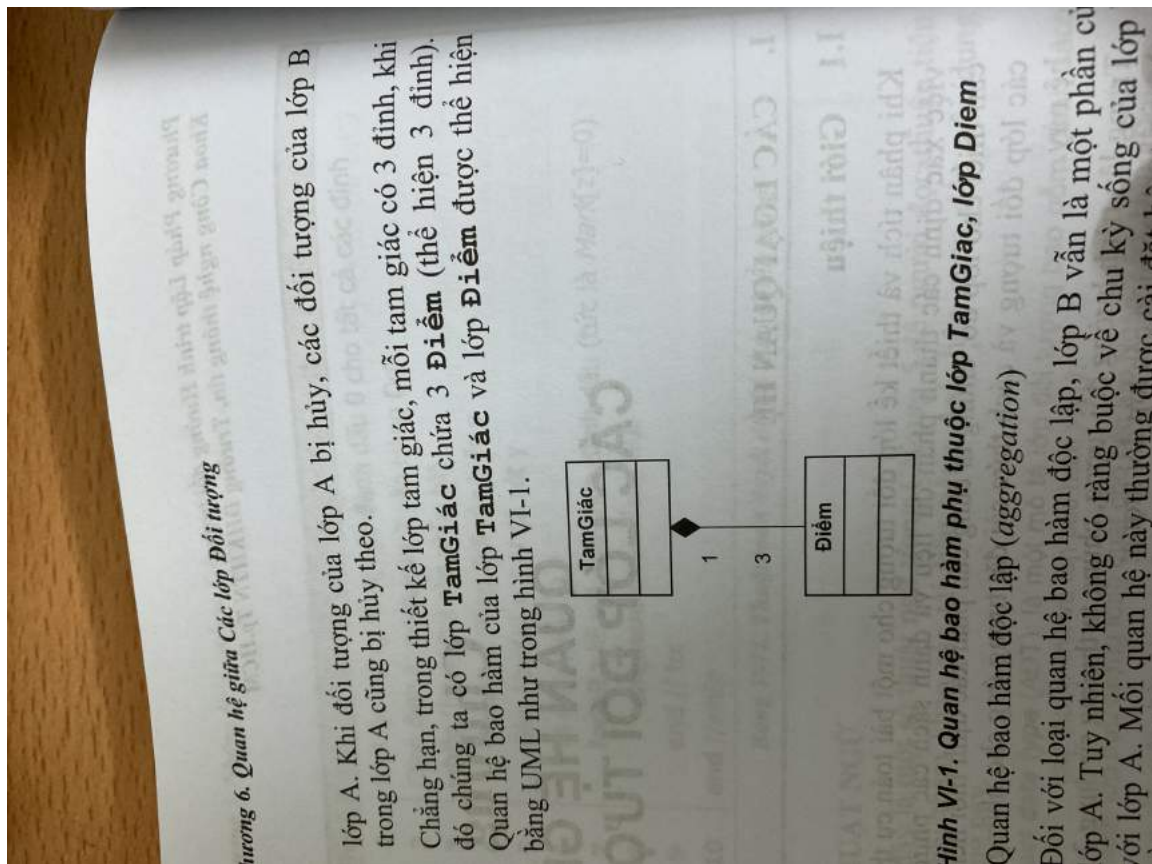
```
class ClassA {
private:
    ClassB b;
public:
    void setB(ClassB b) {
        this->b = b;
    }
};
```

Đối tượng của ClassB tồn tại độc lập với đối tượng của ClassA (ClassA không tạo ra đối tượng của ClassB)

b. Quan hệ bao hàm phụ thuộc (Composition):

Quan hệ bao hàm phụ thuộc xảy ra khi đối tượng của ClassB là 1 phần trong đối tượng của ClassA, khi đối tượng của ClassA bị hủy thì đối tượng của ClassB cũng bị hủy theo.

Chu kỳ sống của đối tượng lớp B sẽ ngắn hơn hoặc bằng chu kỳ sống của đối tượng lớp A. **Khi đối tượng của lớp A bị hủy, các đối tượng của lớp B trong lớp A cũng bị hủy theo**



```

class ClassA {
private:
    ClassB *b;
public:
    ClassA() {
        b = new ClassB();
    }
};
  
```

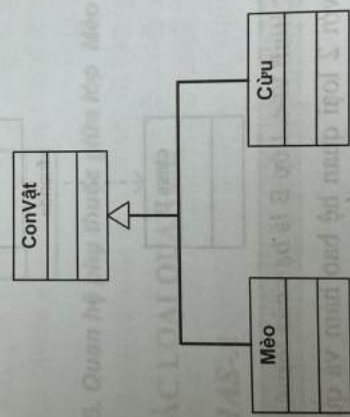
3. Quan hệ tổng quát hóa/đặc biệt hóa (IS-A):

Được sử dụng cho mối liên hệ giữa 2 lớp đối tượng A và B trong trường hợp lớp này là một trường hợp tổng quát hay đặc biệt của lớp kia

1.3 Quan hệ tổng quát hóa/đặc biệt hóa (IS-A)

Đây là quan hệ đã được trình bày và minh họa khá kỹ trong Chương 4. Quan hệ tổng quát hóa/đặc biệt hóa được sử dụng cho mối liên hệ giữa 2 lớp đối tượng A và B trong trường hợp lớp này là một trường hợp tổng quát hay đặc biệt của lớp kia.

Chẳng hạn, lớp *ConVật* là trường hợp tổng quát hóa của lớp *Mèo*, lớp *Cừu* (xem hình VI-3).



Hình VI-3. Quan hệ kế thừa giữa lớp *ConVật* và *Mèo*, *Cừu*

Trong ví dụ trên, chúng ta xem “*Con Mèo* là một *Con Vật*” hay “*Con Cừu* là một *Con Vật*” nên chúng sẽ thừa hưởng tất cả các tính chất như một *Con Vật*.

MỘT SỐ LOẠI QUAN HỆ KHÁC

2 loại quan hệ chính thường được sử dụng ở trên, chính là

4. Quan hệ bạn bè (friend):

Quan hệ bạn bè là quan hệ cho phép lớp bạn của mình được quyền truy xuất đến toàn bộ nội dung (dữ liệu hay phương thức) của mình, kể cả các thành phần thuộc *private*.

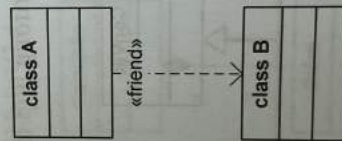
Quan hệ bạn là quan hệ tĩnh giữa hai lớp đối tượng, có thể kiểm tra vào thời điểm biên dịch mã nguồn.

Hàm friend

Chương 6. Quan hệ giữa Các lớp Đối tượng

Quan hệ bạn là quan hệ tính giữa hai lớp đối tượng, có thể kiểm tra vào thời điểm biên dịch mã nguồn.

Khi lớp B là bạn của lớp A: thì lớp B có quyền truy xuất và sử dụng tất cả các thành phần (kể cả thành phần *private*) của lớp A. Hình VI-4 minh họa mô tả UML của quan hệ này.



Hình VI-4. Lớp B là bạn của lớp A

Ghi chú. Khác với 2 loại quan hệ bao hàm và quan hệ đặc biệt hóa, quan hệ bạn bè không tự hủy đối tượng của lớp bạn khi lớp này kết thúc chu kỳ sống.

2 Quan hệ phụ thuộc

Quan hệ phụ thuộc cho biết khi có sự thay đổi trên lớp...

E. Exception Handling:

Ví dụ 1:

```
// IndexOutOfRangeException.h
#include <iostream>
#include <exception>
#include <string>
#include <sstream>

using namespace std;

class IndexOutOfRangeException : public exception {
private:
    int idx;
    int capacity;
public:
    IndexOutOfRangeException(const int& idx, const int& capacity) {
        this->idx = idx;
        this->capacity = capacity;
    }
    const char* what() const throw() {
        stringstream ss;
        ss << "Exception: Index " << idx << " is out of range [0, " << capacity << ")";
        string s = ss.str();
    }
}
```

```

        // return s.c_str();
        return _strdup(s.c_str());
    }
};

```

```

#include "IndexOutOfRangeException.h"

int main() {
    int n = 0;
    cout << "Enter size of array: ";
    cin >> n;
    Array a(n);
    cin >> a;
    cout << "Original array: " << a << endl;
    int idx = 0;
    cout << "Enter index to get value: ";
    cin >> idx;
    try {
        if (!a.isValidIndex(idx)) {
            throw IndexOutOfRangeException(idx, a.getSize());
        }
        cout << "Value at index " << idx << " is " << a.getValAt(idx) << endl;
    }
    catch (IndexOutOfRangeException& e) {
        cout << e.what() << endl;
    }
    return 0;
}

```

Ví dụ 2:

```

// FileNotFoundException.h
#include <iostream>
#include <exception>
#include <string>
#include <sstream>

using namespace std;

class FileNotFoundException : public exception {
private:
    string url;
    string fileName;
public:
    FileNotFoundException(const string& url, const string& fileName) {
        this->url = url;
        this->fileName = fileName;
    }
    const char* what() const throw() {
        stringstream ss;
        ss << "Exception: File " << fileName << " not found" << endl;
        ss << "Details: " << url << " is invalid";
    }
}

```

```

        string s = ss.str();
        // return s.c_str();
        return _strdup(s.c_str());
    }
};

```

```

#include "FileNotFoundException.h"
#include <fstream>

int main() {
    ifstream ifs;
    string url;
    cin.ignore();
    cout << "Enter file url: ";
    getline(cin, url);
    string fileName = url.substr(url.rfind('\\') + 1);
    ifs.open(url);
    try {
        if (!ifs.is_open()) {
            throw FileNotFoundException(url, fileName);
        }
        cout << "File " << fileName << " is opened" << endl;
        string msg = "";
        getline(ifs, msg);
        cout << "Read from file: " << msg << endl;
        ifs.close();
    }
    catch (FileNotFoundException& e) {
        cout << e.what() << endl;
    }
    return 0;
}

```