Ngôn ngữ lập trình C++

CHƯƠNG II. LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG TRONG C++

ThS. Phạm Đức Cường

cuongpd@ptit.edu.vn

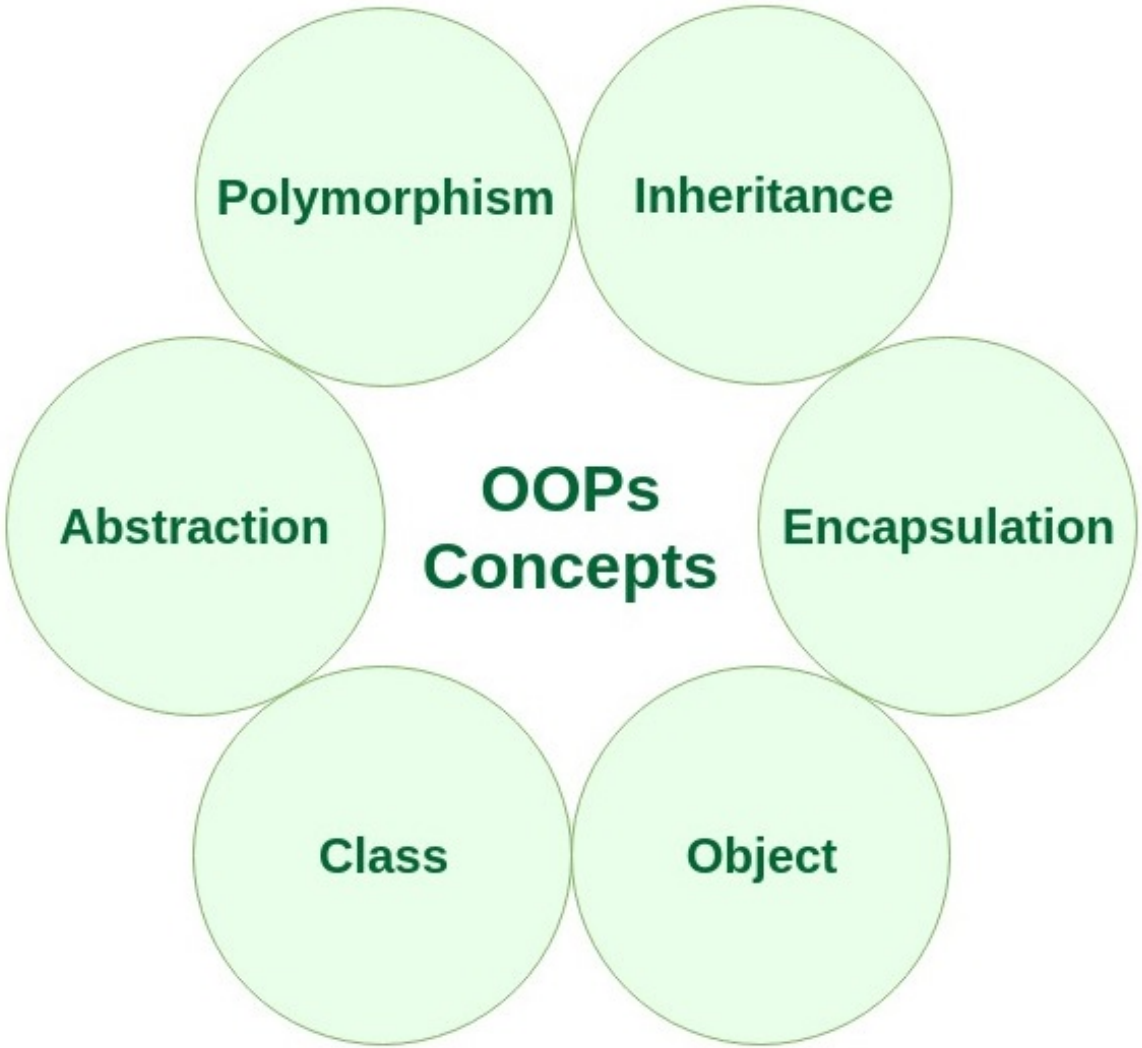
Posts and Telecommunications Institute of Technology

Học viện Công nghệ Bưu chính Viễn thông

2024

Nội dung chương

- **Phần 1:** OOP cơ bản
- **Phần 2:** OOP nâng cao



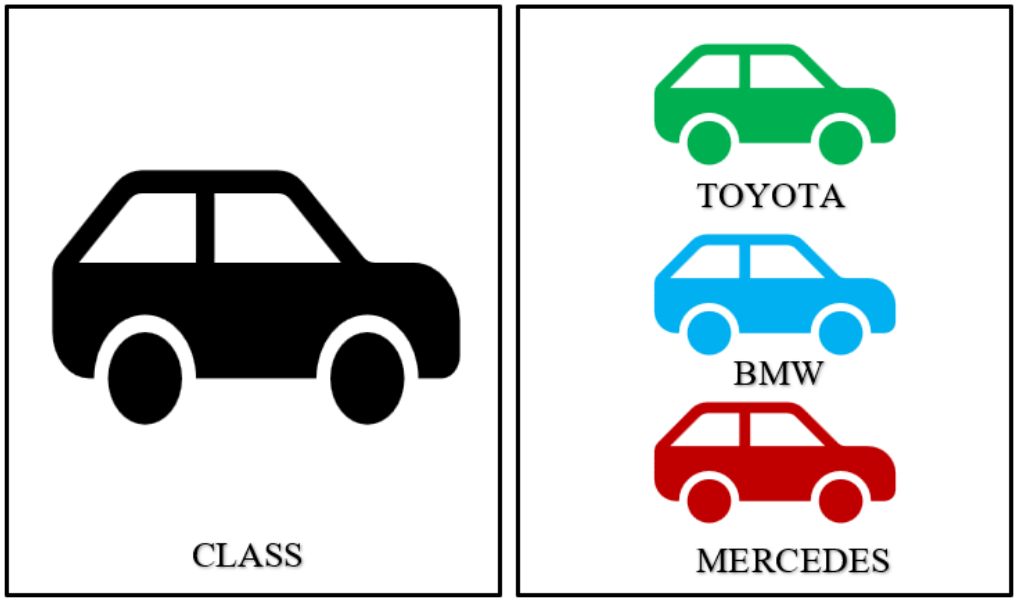
Hình 2: Khái niệm liên quan OOP

- Là khái niệm trung tâm của lập trình hướng dối tượng
- Trừu tượng hoá đối tượng trong thế giới thực thành đối tượng trong chương trình
- Thể hiện bằng từ khóa **class** trong C++, thuộc kiểu dữ liệu người dùng tự định nghĩa, chứa các biến thành viên (thuộc tính) và hàm thành viên (phương thức)
- Được coi là khuôn mẫu để tạo ra các đối tượng
- Cú pháp:

```
1 class ClassName {
2 access_specifier:
3     data_members;
4     member_functions;
5 };
```

- Đối tượng là thể hiện của một lớp
- Khi một lớp được định nghĩa, không có bộ nhớ nào được cấp phát, nhưng khi lớp được khởi tạo (tức là khi một đối tượng được tạo ra), bộ nhớ mới được cấp phát
- Cú pháp:

```
1 ClassName ObjectName;
```



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

class Car {
public:
    string brand;
    string model;
    int year;

    void printInfo() {
        cout << "Brand: " << brand << endl;
        cout << "Model: " << model << endl;
        cout << "Year: " << year << endl;
    }
};

int main() {
    Car car1;
    car1.brand = "Toyota";
    car1.model = "Camry";
    car1.year = 2024;
    car1.printInfo();
    return 0;
}
```

Mã nguồn 1: Ví dụ đối tượng trong C++

- Là các từ khoá dùng để kiểm soát quyền truy cập thành viên của lớp
- Chia thành ba loại:
  - **public:** có thể truy cập từ bên ngoài lớp
  - **private:** chỉ có thể truy cập từ bên trong lớp
  - **protected:** chỉ có thể truy cập từ bên trong lớp và các lớp dẫn xuất
- Mặc định, tất cả các thành viên của lớp đều là private (khác với struct - mặc định là public)

```
1 ClassName ObjectName;
```

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

class Car {
public:
    string brand;
    void setPrice(string c) {
        price = c;
    }
    string getPrice() {
        return price;
    }
private:
    string price;
};

int main() {
    Car car1;
    car1.brand = "Toyota"; // OK
    car1.price = "100000$"; // Error
    car1.setPrice("100000$"); // OK
    return 0;
}
```

Mã nguồn 2: Ví dụ access modifier

- Hai cách:
  - **Bên trong lớp:** viết trực tiếp trong lớp
  - **Bên ngoài lớp:** sử dụng toán tử phạm vi (::)
- Ví dụ:

```
1  class Car {
2  private:
3      string price;
4  public:
5      void setPrice(string c) {
6          price = c;
7      }
8      string getPrice();
9  };
10
11 string Car::getPrice() {
12     return price;
13 }
```

- Là phương thức đặc biệt được gọi tự động khi một đối tượng của lớp được tạo ra
- Sử dụng để khởi tạo giá trị cho các biến thành viên
- Cùng tên với lớp, không có kiểu trả về
- Có thể có hoặc không tham số
- Thường được khai báo public dù không bắt buộc
- **Constructor** cũng là hàm thành viên của lớp nên có thể được định nghĩa trong hoặc ngoài lớp
- Cú pháp:

```
1 class ClassName {
2 public:
3     ClassName();
4 };
```

- **Default Constructor:** không có tham số
- **Parameterized Constructor:** có tham số
- **Copy Constructor:** nhận tham số là một đối tượng cùng kiểu với lớp
- **Move Constructor:** nhận tham số là một đối tượng rvalue - ít dùng, tham khảo thêm Move constructor

- Là **constructor** không có tham số hoặc toàn bộ tham số đều có giá trị mặc định - trường hợp này có thể coi là **parameterized constructor**
- Nếu không có bất kì **constructor** nào được định nghĩa, trình biên dịch tự động tạo ra một **default constructor** khi biên dịch
- Nếu được định nghĩa, trình biên dịch không tạo ra **default constructor** khác mà tự động gọi **default constructor** đã được định nghĩa
- Ví dụ:

```
1  class Car {
2  public:
3      string brand;
4      string model;
5      int year;
6
7      Car(); // Compiler-generated default constructor
8      Car() { // User-defined default constructor
9          brand = "Toyota";
10         model = "Camry";
11         year = 2024;
12     }
13 };
```

- Là **constructor** có ít nhất một tham số
- Sử dụng để khởi tạo giá trị biến thành viên khi khởi tạo đối tượng
- Nếu được định nghĩa, trình biên dịch không tạo ra **default constructor** dù **default constructor** đã được định nghĩa hoặc không
- Ví dụ:

```
1  class Car {
2  public:
3      string brand;
4      string model;
5      int year;
6
7      Car(string b, string m, int y) {
8          brand = b;
9          model = m;
10         year = y;
11     }
12 };
13 Car car1("Toyota", "Camry", 2024);
14 Car car2 = Car("Honda", "Civic", 2024);
15 Car car3; // Error
```

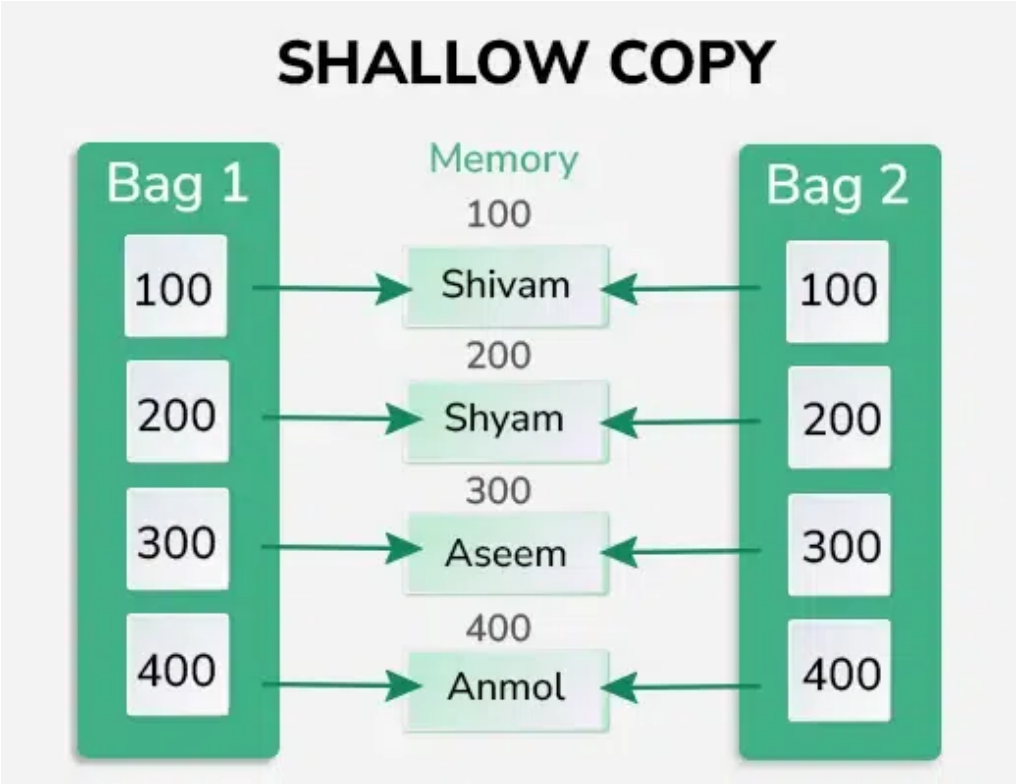
- **Parameterized constructor** có thể định nghĩa với tham số mặc định
- **Parameterized constructor** với toàn bộ tham số mặc định được coi là **default constructor**, có thể được gọi mà không cần truyền tham số, trình biên dịch sẽ báo lỗi nếu định nghĩa **default constructor** khác
- Ví dụ:

```
1  class Car {
2  public:
3      string brand;
4      string model;
5      int year;
6
7      Car (string b = "Toyota", string m = "Camry", int y = 2024)
       {
8          brand = b;
9          model = m;
10         year = y;
11     }
12 };
13 Car car1; // brand = "Toyota", model = "Camry", year = 2024
14 Car car2("Honda", "Civic"); // brand = "Honda", model = "Civic",
       year = 2024
```

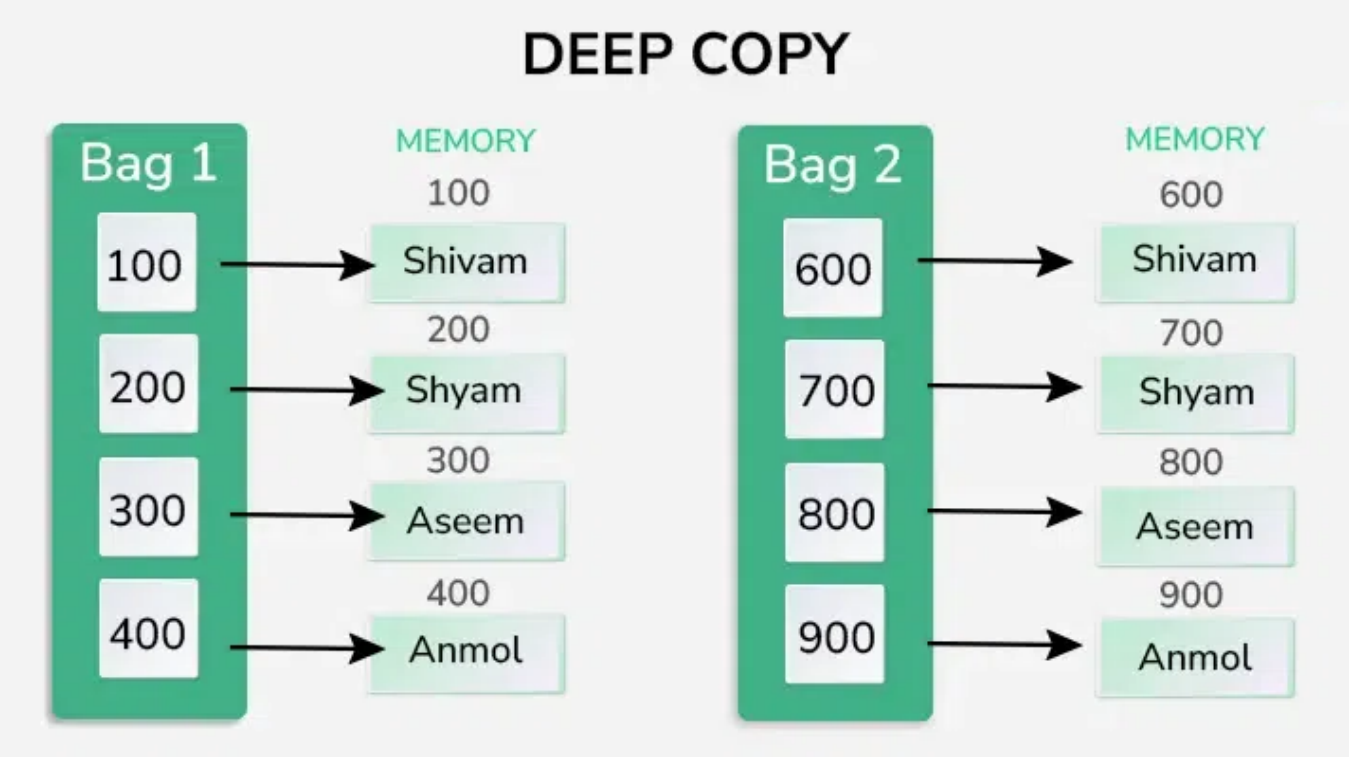
- Có duy nhất một tham số là đối tượng cùng kiểu với lớp
- Sử dụng để khởi tạo một đối tượng từ một đối tượng đã tồn tại
- Quá trình khởi tạo các thành viên của một đối tượng thông qua **copy constructor** được gọi là copy initialization (khởi tạo sao chép) hay member-wise initialization (khởi tạo từng thành viên)
- Nếu không được định nghĩa, trình biên dịch tự động tạo ra một **copy constructor** thực hiện gán giá trị của từng biến thành viên của đối tượng gốc cho đối tượng mới
- Cú pháp:

```
1 // const to avoid modification, can be omitted
2 ClassName(const ClassName &obj) {
3     member1 = obj.member1;
4     member2 = obj.member2;
5     ...
6 }
```

- **Copy constructor** do trình biên dịch tự động tạo ra không thực hiện đúng khi đối tượng chứa con trỏ hoặc tham chiếu do chỉ sao chép giá trị của con trỏ hoặc tham chiếu, không sao chép vùng nhớ mà con trỏ trỏ tới hoặc tham chiếu đến - **shallow copy**



- Cần tự định nghĩa **copy constructor** để thực hiện **deep copy** khi đối tượng chứa con trỏ hoặc tham chiếu, đảm bảo các con trỏ hoặc tham chiếu của đối tượng sao chép sẽ trỏ đến một bản sao mới của tài nguyên mà con trỏ hoặc tham chiếu của đối tượng gốc trỏ đến



Copy Constructor

```cpp
#include <iostream>

using namespace std;

class MyClass {
public:
    int* ptr;

    // Default Constructor
    MyClass(int val) {
        ptr = new int(val);  // Dynamically allocate memory
        cout << "Constructor called. Value = " << *ptr << endl;
    }

    // User-defined Copy Constructor (Deep Copy)
    MyClass(const MyClass& obj) {
        // Allocate new memory and copy the value
        ptr = new int(*obj.ptr);
        cout << "Copy constructor called. Value = " << *ptr << endl;
    }

    // Function to print the value
    void printValue() {
        cout << "Value = " << *ptr << endl;
    }
};

```

```
28 int main() {
29     MyClass obj1(10);
30     MyClass obj2 = obj1;
31 
32     cout << "Values after deep copy:" << endl;
33     obj1.printValue();
34     obj2.printValue();
35 
36     // Modify obj1's value to check independence of obj2
37     *obj1.ptr = 20;
38     cout << "\nValues after modifying obj1:" << endl;
39     obj1.printValue();
40     obj2.printValue();
41 }
```

Mã nguồn 3: Ví dụ copy constructor

- Toán tử gán mặc định với đối tượng cũng thực hiện **shallow copy**, nếu muốn thực hiện **deep copy** cần nạp chồng toán tử gán
- Copy constructor và toán tử gán dễ nhầm lẫn với nhau, cần phân biệt rõ ràng
- Ví dụ:

```
1 MyClass t1, t2;
2 MyClass t3 = t1; // Copy constructor
3 MyClass t4(t1); // Copy constructor
4 t2 = t1; // Assignment operator
```

Hàm huỷ - Destructor

- Là phương thức đặc biệt được gọi tự động khi một đối tượng của lớp sắp bị huỷ
- Sử dụng để huỷ các đối tượng do **constructor** tạo ra, giải phóng bộ nhớ đã cấp phát
- Không thể định nghĩa nhiều hơn một **destructor** trong một lớp
- Tự động gọi khi một đối tượng ra khỏi phạm vi của nó hoặc, khi hàm **delete** được gọi hoặc kết thúc chương trình
- Các đối tượng bị huỷ theo thứ tự ngược với thứ tự được tạo ra

- Cùng tên với lớp, không có kiểu trả về, không có tham số và có dấu $\sim$ ở trước
- **Destructor** cũng là hàm thành viên của lớp nên có thể được định nghĩa trong hoặc ngoài lớp
- **Destructor** được tạo tự động nếu không định nghĩa, không thể giải phóng bộ nhớ động được cấp phát bằng **new** nếu không định nghĩa
- Cú pháp:

```
1 class ClassName {
2 public:
3     ~ClassName();
4 };
```

- Gói gọn dữ liệu và thông tin vào trong một đơn vị duy nhất
- Hai đặc tính quan trọng:
  - **Ẩn Thông tin - Data hiding:** ẩn chi tiết triển khai nội bộ của lớp, chỉ cung cấp giao diện công khai
  - **Bảo vệ dữ liệu - Data protection:** bảo vệ trạng thái nội bộ của đối tượng, không cho phép truy cập trực tiếp từ bên ngoài

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

class Calculator {
private:
    int a;
    int b;
public:
    int half(int input) {
        a = input;
        b = a / 2;
        return b;
    }
};

int main() {
    int n;
    cin >> n;
    Calculator c;
    int ans = c.half(n);
    cout << ans << endl;

    return 0;
}
```

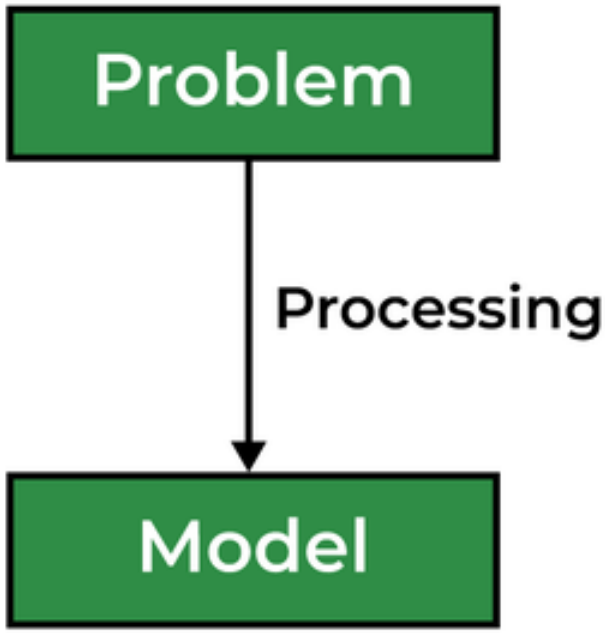
Mã nguồn 4: Ví dụ encapsulation

- **Đặc điểm của encapsulation:**
  - Phải dùng đối tượng để truy cập hàm của lớp
  - Hàm trong lớp phải sử dụng biến thành viên của lớp để được coi là encapsulation
  - Không có hàm sử dụng biến thành viên thì không phải là encapsulation
  - Cải thiện khả năng đọc, bảo trì, và bảo mật
  - Kiểm soát việc sửa đổi các thành viên dữ liệu

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

class Encapsulation {
private:
    // Data hidden from outside world
    int x;
public:
    // Function to set value of x
    void set(int a) {
        x = a;
    }

    // Function to return value of x
    int get() {
        return x;
    }
};

int main() {
    Encapsulation obj;
    obj.set(5);
    cout << obj.get();

    return 0;
}
```

Mã nguồn 5: Ví dụ encapsulation

- Cung cấp thông tin cần thiết cho bên ngoài và bỏ qua các chi tiết hoặc cách triển khai không cần thiết
- Các loại trừu tượng:
  - **Trừu tượng dữ liệu - Data abstraction**
  - **Trừu tượng điều khiển - Control abstraction**

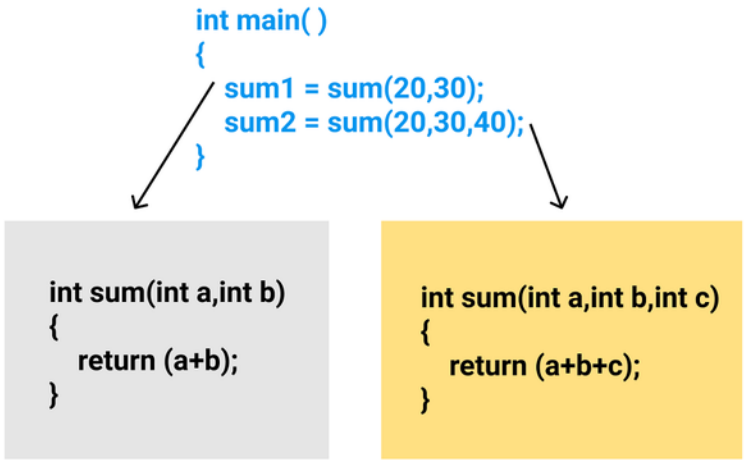


```cpp
#include <iostream>
using namespace std;

class Abstraction {
private:
    int a, b;

public:
    void set(int x, int y) {
        a = x;
        b = y;
    }

    void display() {
        cout << "a = " << a << endl;
        cout << "b = " << b << endl;
    }
};

int main() {
    Abstraction obj;
    obj.set(10, 20);
    obj.display();

    return 0;
}
```

Mã nguồn 6: Ví dụ abstraction

- **Ưu điểm của abstraction:**
  - Giúp người dùng tránh viết mã cấp thấp
  - Tránh sự trùng lặp mã và tăng khả năng tái sử dụng
  - Có thể thay đổi cách triển khai bên trong lớp mà không ảnh hưởng đến người dùng
  - Tăng cường bảo mật cho ứng dụng bằng cách chỉ cung cấp những chi tiết quan trọng
  - Giảm độ phức tạp và sự dư thừa của mã, từ đó tăng tính dễ đọc
  - Có thể thêm các tính năng mới hoặc thay đổi hệ thống mà ít ảnh hưởng đến mã hiện có

- Cho phép một lớp có khả năng định nghĩa ra nhiều phương thức có cùng tên, nhưng khác nhau về tham số truyền vào
- Các loại đa hình:
  - **Nạp chồng toán tử - Operator overloading**
  - **Nạp chồng hàm - Function overloading**



- **Nạp chồng toán tử - Operator overloading** là cách thức cho phép một toán tử thực hiện nhiều chức năng khác nhau tùy thuộc vào loại dữ liệu của toán hạng
- Các hàm toán tử giống như các hàm bình thường, khác biệt ở tên của hàm toán tử luôn bắt đầu bằng từ khóa toán tử, tiếp theo là ký hiệu của toán tử, và các hàm toán tử được gọi khi toán tử tương ứng được sử dụng

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

class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i = 0) {
        real = r;
        imag = i;
    }
    Complex operator+(Complex const& obj) {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
    void print() {
        cout << real << " + i" << imag << endl;
    }
};

int main() {
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; c3.print();
    return 0;
}
```

- Hầu hết toán tử có thể được nạp chồng, ngoại trừ:
  - **sizeof**
  - **typeid**
  - **::**
  - **.**
  - **.***
  - **?:**
- Nạp chồng toán tử hoạt động khi ít nhất một trong các toán hạng của toán tử là đối tượng của lớp đã được nạp chồng
- Toán tử nạp chồng không thể thay đổi số lượng toán hạng, nói cách khác, số lượng tham số của toán tử không thể thay đổi
- Toán tử nạp chồng không thể thay đổi ưu tiên của toán tử
- Toán tử gán luôn được nạp chồng mặc định

- Toán tử ép kiểu có thể được nạp chồng để chuyển đổi một kiểu dữ liệu thành một kiểu dữ liệu khác
- Ví dụ:

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

class Fraction {
private:
    int num, den;
public:
    Fraction(int n, int d) {
        num = n;
        den = d;
    }
    operator float() const {
        return float(num) / float(den);
    }
};

int main() {
    Fraction f(2, 5);
    float val = f;
    cout << val << '\n';
    return 0;
}
```

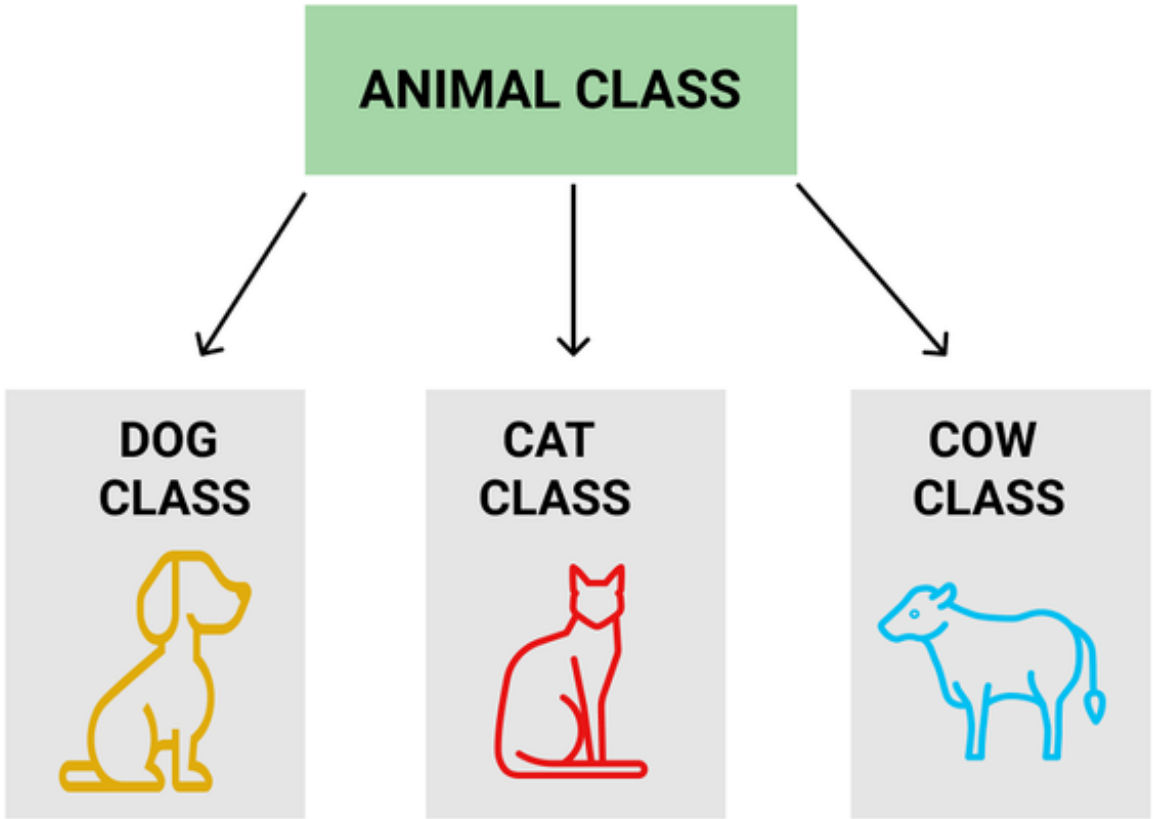
- Mọi constructor nếu có thể nhận duy nhất 1 tham số được gọi là conversion constructor (hàm tạo chuyển đổi), giúp chuyển đổi đối tượng ngầm định sang kiểu kiểu dữ liệu của tham số

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

class Point {
private:
    int x, y;
public:
    Point(int i = 0, int j = 0) {
        x = i;
        y = j;
    }
    void print() {
        cout << "x = " << x << ", y = " << y << endl;
    }
};

int main() {
    Point t(20, 20); t.print();
    t = 30; t.print();
    return 0;
}
```

- **Nạp chồng hàm - Function overloading** là cách thức cho phép một lớp hay chương trình có nhiều hàm cùng tên nhưng khác nhau về tham số truyền vào
- Ví dụ:

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

void add(int a, int b) {
  cout << "sum = " << (a + b);
}
void add(double a, double b) {
    cout << endl << "sum = " << (a + b);
}
void add(int a, int b, int c) {
  cout << "sum = " << (a + b + c);
}

int main() {
    add(10, 2);
    add(5.3, 6.2);
    add(10, 20, 30);
    return 0;
}
```

- Khả năng sử dụng các thuộc tính và phương thức của lớp khác mà không cần phải viết lại gọi là **kế thừa**



- Cú pháp:

```
class DerivedClass : accessModifier BaseClass {
    // Derived class members
};
```

- Ví dụ:

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

class Parent {
public:
    int id_p;
    void printID_p() {
        cout << "Base ID: " << id_p << endl;} };
class Child : public Parent {
public:
    int id_c;
    void printID_c() {
        cout << "Child ID: " << id_c << endl;} };

int main() {
    Child obj1;
    obj1.id_p = 7; obj1.printID_p();
    obj1.id_c = 91; obj1.printID_c();
    return 0;
}
```

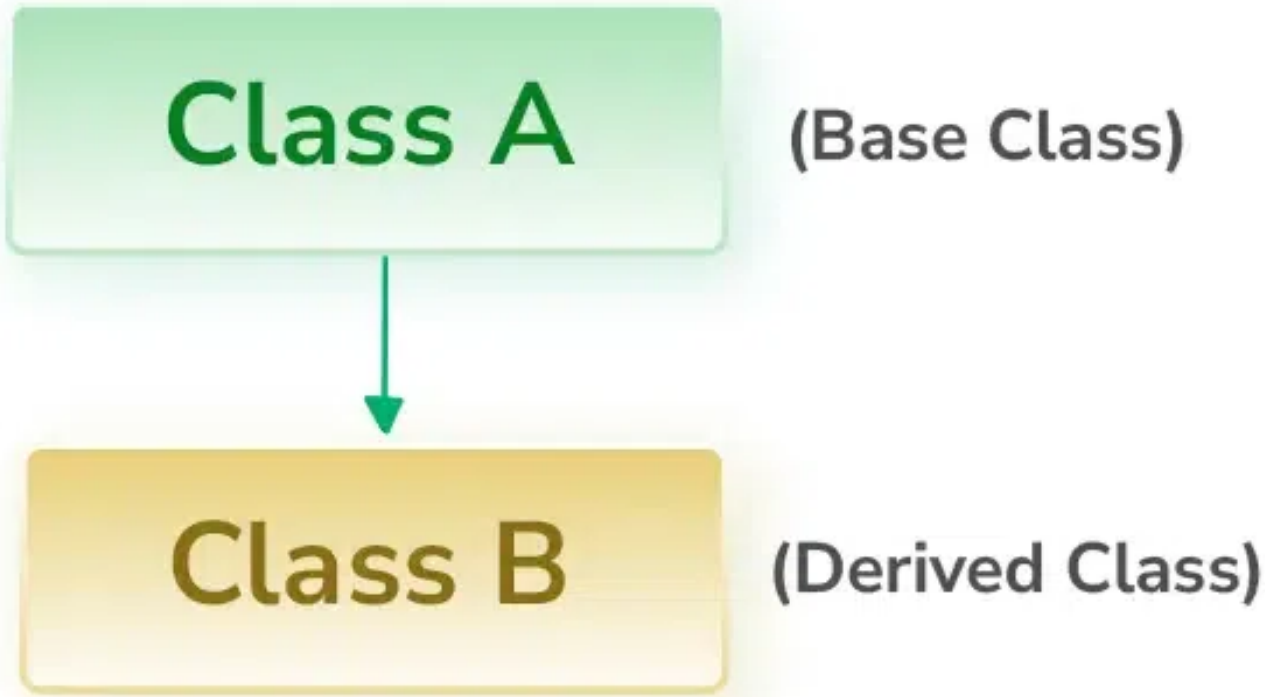
- Ba chế độ kế thừa:
  - **public**
  - **protected**
  - **private** - chế độ mặc định khi không được chỉ định

	MODE OF INHERITANCE		
Base class member access specifier	**Public**	**Protected**	**Private**
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not Accessible (Hidden)	Not Accessible (Hidden)	Not Accessible (Hidden)

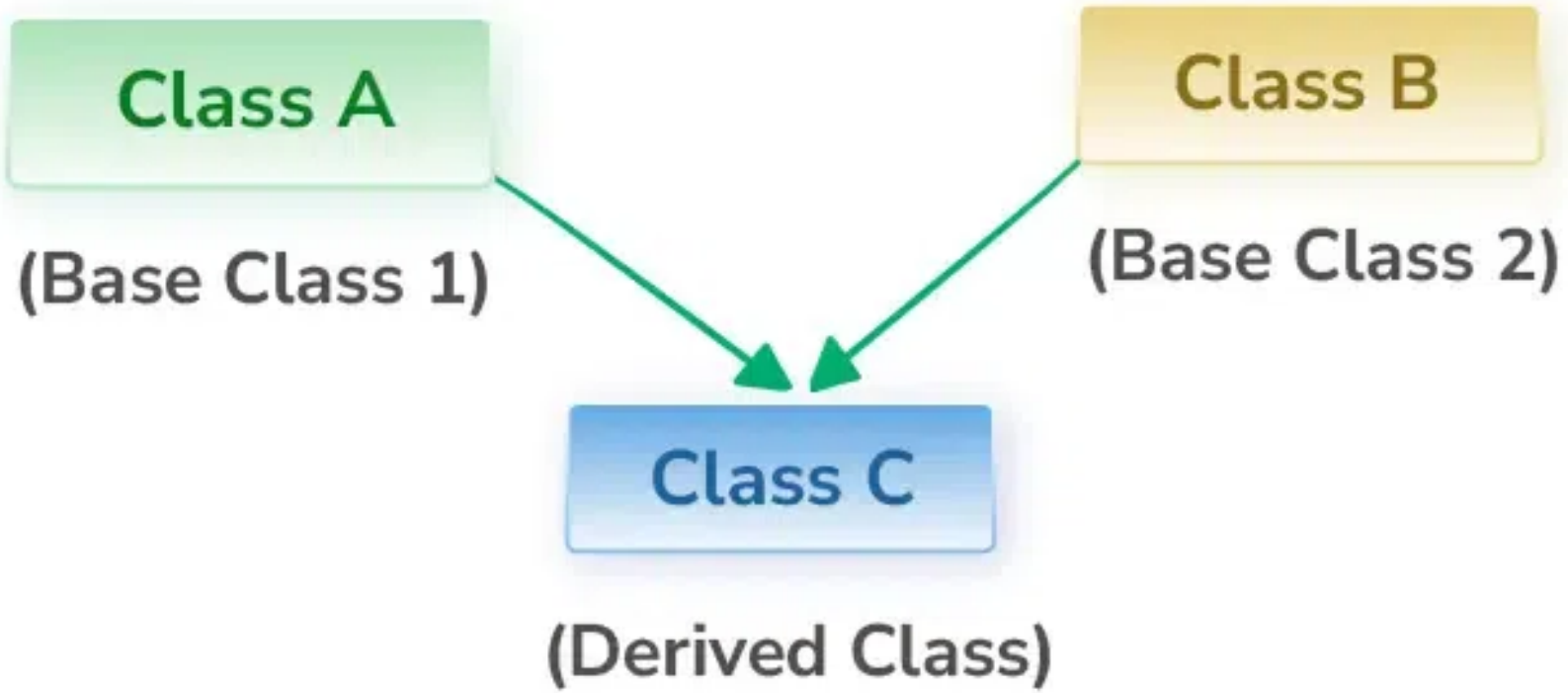
```cpp
class A {
public:
    int x;
protected:
    int y;
private:
    int z;
};

class B : public A {
    // x is public
    // y is protected
    // z is not accessible from B
};
class C : protected A {
    // x is protected
    // y is protected
    // z is not accessible from C
};
class D : private A { // same as class D : A
    // x is private
    // y is private
    // z is not accessible from D
};
```

Mã nguồn 7: Ví dụ các chế độ kế thừa

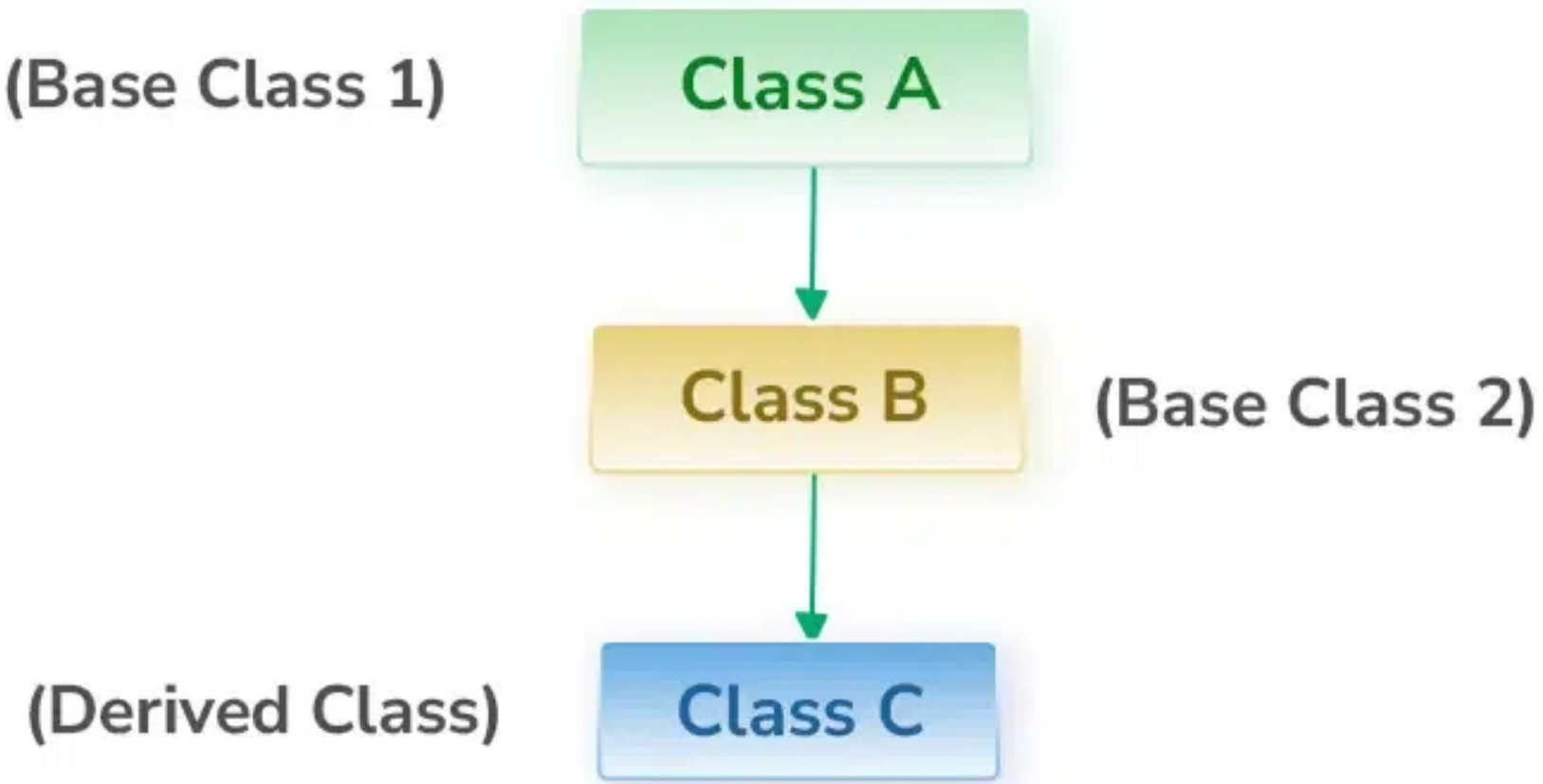
- Năm loại kế thừa:
  - **Single inheritance**
  - **Multilevel inheritance**
  - **Multiple inheritance**
  - **Hierarchical inheritance**
  - **Hybrid inheritance**



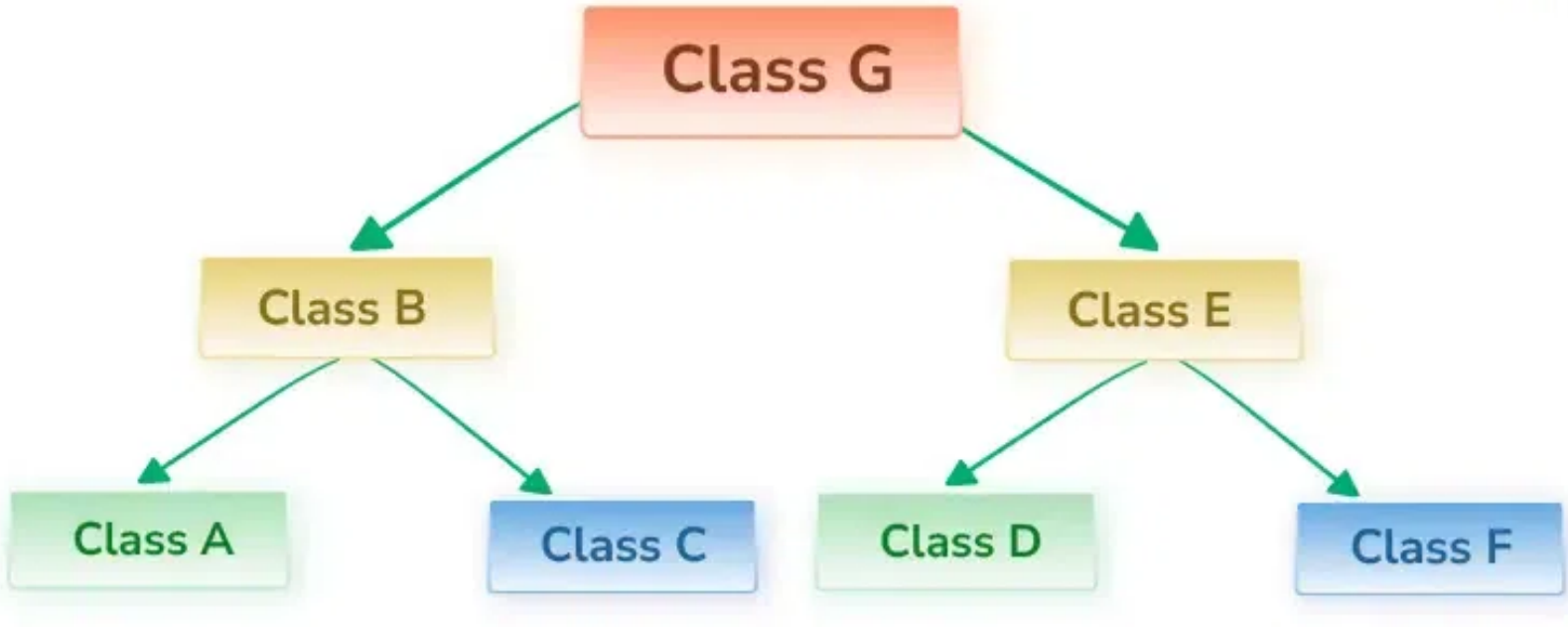
Hình 3: Single inheritance



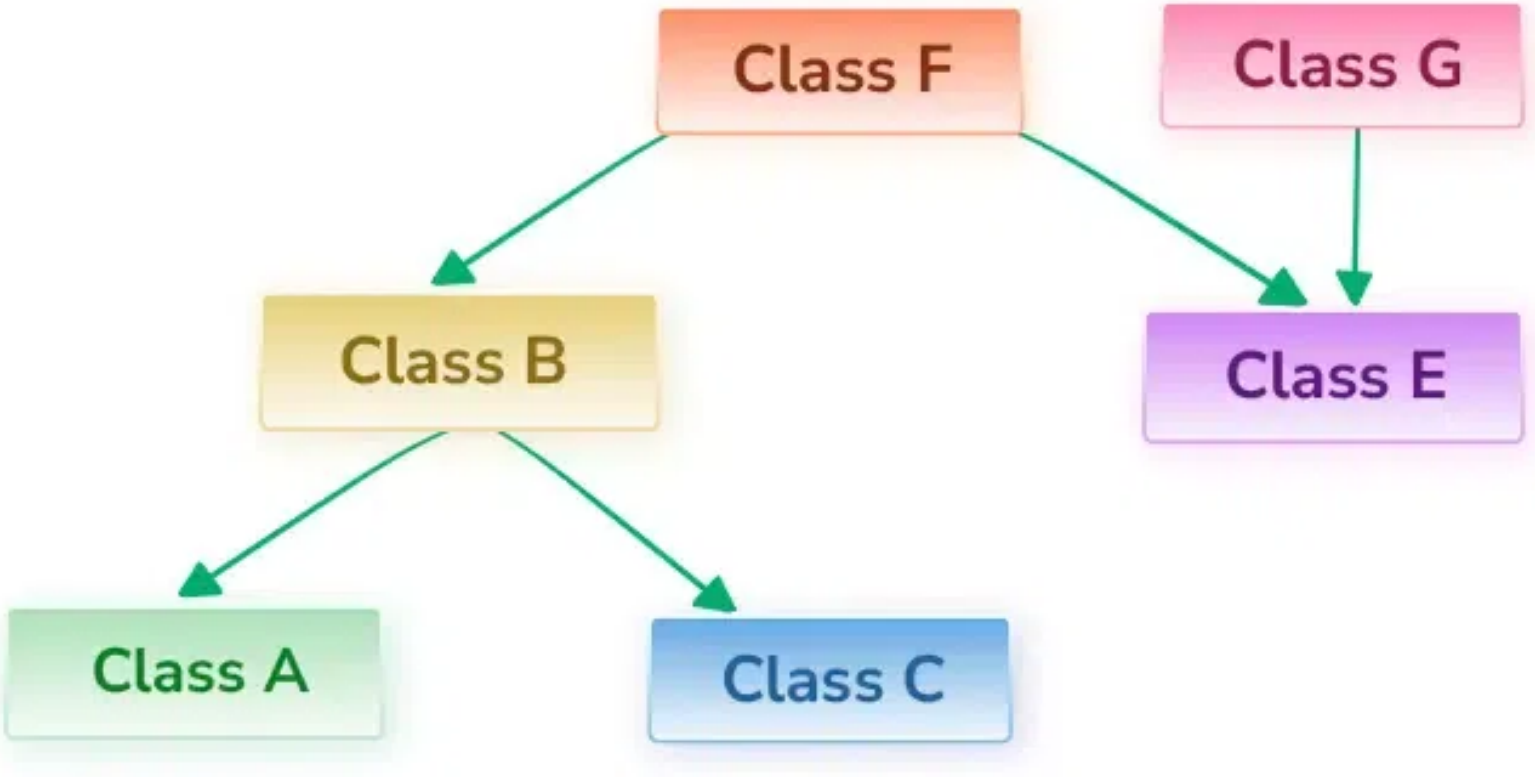
Hình 4: Multilevel inheritance



Hình 5: Multiple inheritance



Hình 6: Hierarchical inheritance



Hình 7: Hybrid inheritance

- **Constructor** của lớp dẫn xuất được gọi theo thứ tự từ lớp cơ sở đến lớp dẫn xuất
- **Destructor** của lớp dẫn xuất được gọi theo thứ tự ngược lại từ lớp dẫn xuất đến lớp cơ sở
- Có thể gọi **constructor** và **destructor** của lớp cơ sở từ lớp dẫn xuất

- Có thể định nghĩa lại hàm thành viên của lớp cơ sở trong lớp dẫn xuất
- Ví dụ:

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

class Parent {
public:
    void Parent_Print() {
        cout << "Base Function" << endl;
    }
};

class Child : public Parent {
public:
    void Parent_Print() {
        cout << "Derived Function" << endl;
    }
};

int main() {
    Child Child_Derived;
    Child_Derived.Parent_Print(); // Derived Function
    return 0;
}
```

- **this** là một con trỏ đặc biệt tồn tại trong mọi hàm thành viên của một lớp, trỏ đến đối tượng gọi hàm
- Tự động có sẵn các thành viên của lớp
- Là con trỏ hằng, không thể trỏ đến một đối tượng khác
- Thường dùng để phân biệt giữa các biến thành viên và các tham số của hàm hoặc trả về đối tượng trong các hàm thành viên

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

class MyClass {
private:
    int x;
public:
    MyClass(int x) {
        this->x = x;
    }
    MyClass& setX(int x) {
        this->x = x;
        return *this;
    }
    void print() {
        cout << "x = " << this->x << endl;
    }
};

int main() {
    MyClass obj(10);
    obj.setX(20).print();

    return 0;
}
```

Mã nguồn 8: Ví dụ con trỏ this