# C++ Training Course

*Inheritance*

# Lesson Objectives

- Understand about inheritance (kế thừa) in C++
- Understand about constructors and initialization (khởi tạo)
- Understand about inheritance and access specifiers
- Understand about overriding and hiding functionality
- Understand about multiple inheritance

Cấu trúc của vtable:

Mỗi lớp có chứa ít nhất một hàm ảo sẽ có một vtable riêng.

Vtable là một mảng các con trỏ hàm, mỗi con trỏ trỏ đến một hàm ảo của lớp đó.

Tạo vtable:

Khi một lớp có chứa các hàm ảo được định nghĩa, trình biên dịch sẽ tạo một vtable cho lớp đó.Trong vtable, mỗi mục (entry) là một con trỏ trỏ đến định nghĩa của một hàm ảo của lớp.

Con trỏ vtable trong đối tượng:Mỗi đối tượng của một lớp có chứa các hàm ảo sẽ có một con trỏ ẩn trỏ đến vtable của lớp đó. Con trỏ này được gọi là vptr (virtual table pointer).

Vptr được khởi tạo tự động khi đối tượng được tạo ra và được thiết lập để trỏ đến vtable tương ứng của lớp.Gọi hàm ảo:Khi một hàm ảo được gọi thông qua con trỏ hoặc tham chiếu của lớp cơ sở, con trỏ vptr của đối tượng sẽ được sử dụng để tìm vtable.Sau đó, từ vtable, con trỏ hàm tương ứng với hàm ảo được gọi sẽ được sử dụng để thực thi hàm thực sự.

=> Các đối tượng lớp con đã chứa đầy đủ thông tin lớp cha (cấp phát bộ nhớ lúc khởi tạo và gọi constructor của lớp cha rồi) nên là Vtable đã được kế thừa trọn vẹn và phát triển, chúng không cần liên kết với nhau nữa

Section 1

# Inheritance in C++

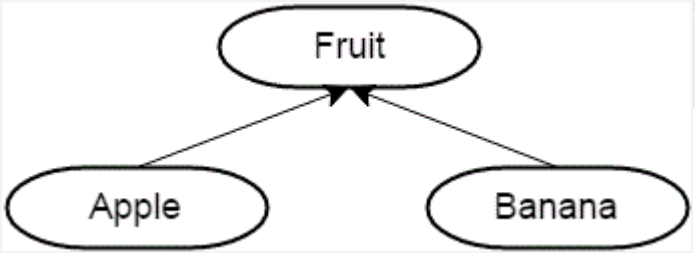
# Inheritance in C++. Agenda

- Introduction to inheritance
- Basic inheritance in C++
- Why inheritance is useful?

# Introduction to inheritance

- Inheritance
  - ✓ Model “is-a” relationship between two objects (mối quan hệ) Mô hình hóa mối quan hệ "là một" giữa hai đối tượng
  - ✓ Create new objects by directly (trực tiếp) acquiring (sử dụng) the attributes (thuộc tính) and behaviors (hành vi) of other objects and then extending or specializing them

    Tạo ra các đối tượng mới bằng cách trực tiếp sử dụng các thuộc tính và hành vi của các đối tượng khác và sau đó mở rộng hoặc đặc thù hóa chúng



# Introduction to inheritance

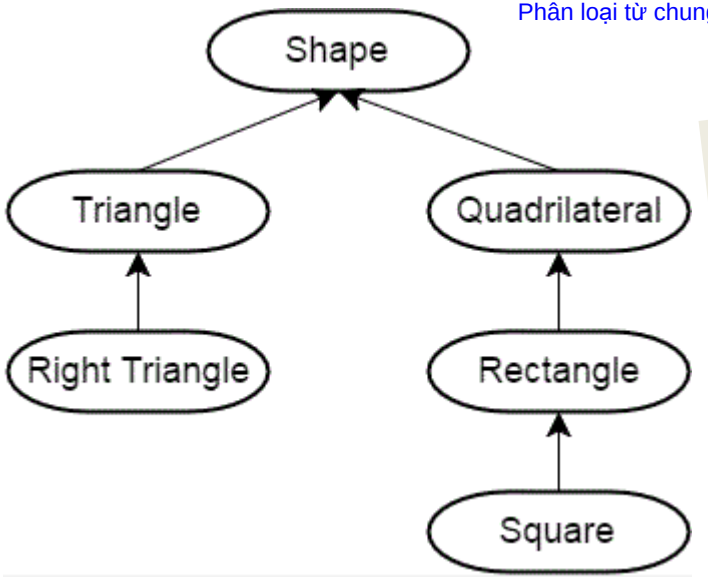
- Hierarchies: diagram that shows how various objects are related
  - ✓ Progression over time: 386 -> 486 -> Pentium
  - ✓ Categorize from general to specific: fruit -> apple -> red delicious

sơ đồ khác nhau liên quan

Sơ đồ cho thấy cách các đối tượng khác nhau liên quan đến nhau

Tiến triển theo thời gian: 386 -> 486 -> Pentium

Phân loại từ chung đến cụ thể: trái cây -> táo -> red delicious



# Basic inheritance in C++

- Inheritance in C++ takes place between classes Kế thừa trong C++ diễn ra giữa các lớp
  - ✓ Class being inherited: parent class, base class, or superclass Lớp được kế thừa: lớp cha, lớp cơ sở, hoặc lớp siêu
  - ✓ Class doing the inheriting: child class, derived class, or subclass Lớp thực hiện kế thừa: lớp con, lớp dẫn xuất, hoặc lớp phụ
- Child class inherits both behaviors and properties from the parent thuộc tính Lớp con kế thừa cả hành vi và thuộc tính từ lớp cha
- Child class can have its own members that are specific to that class Lớp con có thể có các thành viên riêng của nó, đặc trưng cho lớp đó

# Basic inheritance in C++

```
class Person
{
private:
   string m_name;
   int m_age;

public:
   Person(string name = "", int age = 0)
      : m_name(name)
      , m_age(age)
   {
   }

   string getName() { return m_name; }
   int getAge() { return m_age; }
};
```

```
class Employee : public Person
{
private:
   int m_salary;

public:
   Employee(string name = "", int age = 0, int salary = 0)
      : Person(name, age)
      , m_salary(salary)
   {
   }

   void printNameAndSalary()
   {
      cout << getName() << ": " << m_salary << endl;
   }
};
```

# Why inheritance is useful?

tái sử dụng

- Reusable: don't have to redefine the information from the base class in derived classes, only need add the additional functions or member variables => save effort

Có thể tái sử dụng: không cần phải định nghĩa lại thông tin từ lớp cơ sở trong các lớp dẫn xuất, chỉ cần thêm các hàm hoặc biến thành viên bổ sung => tiết kiệm công sức

bảo trì

- Easy to maintain: ever update or modify the base class, all of derived classes will automatically inherit the changes

Dễ bảo trì: khi cập nhật hoặc sửa đổi lớp cơ sở, tất cả các lớp dẫn xuất sẽ tự động kế thừa các thay đổi

OOP

- Object Oriented Programming Thinking: easy to design the system architecture from high level to low level

Tư duy lập trình hướng đối tượng: dễ dàng thiết kế kiến trúc hệ thống từ cấp cao đến cấp thấp

Section 2

# Constructor and initialization

# Constructor and initialization. Agenda

chương trình

thứ tự

- Order of construction of derived class
- Steps to instantiate base class
- Steps to instantiate derived class

khởi tạo

# Order of construction of derived class

- C++ constructs derived classes in phases (giai đoạn)
  C++ xây dựng các lớp dẫn xuất theo các giai đoạn
  - ✓ Starting with the most-base class (top of inheritance tree)
    Bắt đầu với lớp cơ sở nhất (đỉnh của cây kế thừa)
  - ✓ Finishing with the most-child class (bottom of inheritance tree)
    Kết thúc với lớp con nhất (dưới cùng của cây kế thừa)
- As each class is constructed, the appropriate constructor from that class is called to initialize (khởi tạo) that part of the class (tương ứng)
  Khi mỗi lớp được xây dựng, hàm tạo thích hợp từ lớp đó sẽ được gọi để khởi tạo phần tương ứng của lớp đó

# Steps to instantiate base class

các bước để khởi tạo lớp cơ sở

- Memory for base is set aside — Bộ nhớ cho lớp cơ sở được dành riêng
- The appropriate Base constructor is called — Hàm tạo thích hợp của lớp cơ sở được gọi
- The initialization list initializes variables — Danh sách khởi tạo các biến được khởi tạo
- The body of the constructor executes — Phần thân của hàm tạo được thực thi
- Control is returned to the caller — Quyền kiểm soát được trả về cho người gọi

# Steps to instantiate derived class

- Memory for derived is set aside (Base & Derived portions) Bộ nhớ cho lớp dẫn xuất được dành riêng (bao gồm cả phần lớp cơ sở và lớp dẫn xuất), mỗi lớp dẫn xuất sẽ gọi đến 1 lớp cơ sở và cấp phát vùng nhớ mới riêng (kể cả 2 lớp dẫn xuất cùng 1 lớp cơ sở thì mỗi cái sẽ sở hữu 1 lớp cơ sở được khởi tạo riêng ở các địa chỉ khác nhau)
- The appropriate Derived constructor is called Hàm tạo thích hợp của lớp dẫn xuất được gọi
- The Base object is constructed first using the appropriate Base constructor. If no base constructor is specified, the default constructor will be used Đối tượng lớp cơ sở được khởi tạo trước bằng cách sử dụng hàm tạo thích hợp của lớp cơ sở. Nếu không chỉ định hàm tạo lớp cơ sở, hàm tạo mặc định sẽ được sử dụng
- The initialization list initializes variables Danh sách khởi tạo các biến được khởi tạo
- The body of the constructor executes Phần thân của hàm tạo được thực thi
- Control is returned to the caller Quyền kiểm soát được trả về cho người gọi

# Constructor and initialization. Example

```
class Base
{
private:
   int m_id;

public:
   Base(int id = 0)
      : m_id(id)
   {
      cout << "Base" << endl;
   }
};
```

```
class Derived: public Base
{
private:
   double m_cost;

public:
   Derived(int id = 0, double cost = 0.0)
      : Base(id)
      , m_cost(cost)
   {
      cout << "Derived" << endl;
   }
};
```

Section 3

# Inheritance and access specifiers

# Inheritance and access specifiers. Agenda

- Access specifiers in C++
- Kinds of inheritance

# Access specifiers in C++ Các bộ điều khiển truy cập trong C++

- Access specifiers to allow or restrict access to members of class
  hạn chế
  Các bộ điều khiển truy cập cho phép hoặc hạn chế truy cập đến các thành viên của lớp
- Three types of access
  - ✓ **public**: can be accessed by anybody có thể được truy cập bởi bất kỳ ai
  - ✓ **protected**: can be accessed by base members, friends, and derived classes có thể được truy cập bởi các thành viên lớp cơ sở, bạn bè và các lớp dẫn xuất
  - ✓ **private**: can only be accessed by base members and friends (but not derived classes) chỉ có thể được truy cập bởi các thành viên lớp cơ sở và bạn bè (nhưng không phải các lớp dẫn xuất)
- If not specify access specifier, default is private access

  Nếu không chỉ định bộ điều khiển truy cập, mặc định là truy cập private

# Access specifiers in C++

```
class Base {
public:
   int m_public;       // can be accessed by anybody
protected:
   int m_protected;  // can be accessed by Base members, friends, and derived classes
private:
   int m_private;      // can only be accessed by Base members and friends (but not derived classes)
};

class Derived: public Base {
public:
   Derived() {
      m_public = 1;        // allowed: can access public base members from derived class
      m_protected = 2;   // allowed: can access protected base members from derived class
      m_private = 3;       // not allowed: can not access private base members from derived class
   }
};

int main() {
   Base base;
   base.m_public = 1;        // allowed: can access public members from outside class
   base.m_protected = 2;  // not allowed: can not access protected members from outside class
   base.m_private = 3;       // not allowed: can not access private members from outside class
}
```

# Kinds of inheritance

- Three different ways to inherit Có ba cách khác nhau để kế thừa
  - ✓ public
  - ✓ protected
  - ✓ Private
- If not specify inheritance type, default is private inheritance
  Nếu không chỉ định loại kế thừa, mặc định là kế thừa private
- When members are inherited, their access specifier may be changed (in the derived class only) depending on type of inheritance Khi các thành viên được kế thừa, bộ điều khiển truy cập của chúng có thể bị thay đổi (chỉ trong lớp dẫn xuất) tùy thuộc vào loại kế thừa

# Kinds of inheritance

| Access specifier in base class | Access specifier when inherited publicly | Access specifier when inherited protectedly | Access specifier when inherited privately |
|---|---|---|---|
| Public | Public | Protected | Private |
| Protected | Protected | Protected | Private |
| Private | Inaccessible | Inaccessible | Inaccessible |

```
class Base {
private:    int privateMember;
public:
int getPrivateMember() {        return privateMember; // Cho phép truy cập privateMember thông qua hàm public    }
void setPrivateMember(int value) {        privateMember = value; // Cho phép thiết lập giá trị privateMember thông qua hàm public    }};
class Derived : public Base
{public:
void accessPrivateMember() {
// privateMember = 10; // Lỗi: không thể truy cập trực tiếp        setPrivateMember(10);
 / OK: truy cập thông qua hàm thành viên public
int value = getPrivateMember(); // OK: truy cập thông qua hàm thành viên public    }};
```

Section 4

# Overriding and hiding functionality

# Overriding and hiding functionality. Agenda

- Overriding behavior
- Hiding inherited functionality

# Overriding behavior

- Calling a base class function gọi một hàm ở lớp cơ
  - ✓ When a member function is called with a derived class object, the compiler first looks to see if that member exists in the derived class. If not, it begins walking up the inheritance chain and checking whether the member has been defined in any of the parent classes. It uses the first one it finds

tồn tại

chuỗi kế thừa

Khi một hàm thành viên được gọi bằng đối tượng của lớp dẫn xuất, trình biên dịch sẽ kiểm tra xem hàm thành viên đó có tồn tại trong lớp dẫn xuất hay không. Nếu không, nó sẽ bắt đầu đi lên chuỗi kế thừa và kiểm tra xem hàm thành viên có được định nghĩa trong bất kỳ lớp cha nào hay không. Nó sẽ sử dụng hàm đầu tiên mà nó tìm thấy.

- Redefining behaviors Định nghĩa lại hành vi
  - ✓ If we had defined member function in the Derived class, it would have been used instead. This means that we can make functions work differently with our derived classes by redefining them in the derived class

Nếu chúng ta đã định nghĩa hàm thành viên trong lớp dẫn xuất, hàm đó sẽ được sử dụng. Điều này có nghĩa là chúng ta có thể làm cho các hàm hoạt động khác nhau với các lớp dẫn xuất bằng cách định nghĩa lại chúng trong lớp dẫn xuất.

- Adding to existing functionality

Thêm vào chức năng hiện có

  - ✓ Sometimes we don't want to completely replace a base class function, but instead want to add additional functionality to it. In this case, we can call base function by form: <base_class>::<function_name>()

thay thế

thay vào đó

Đôi khi chúng ta không muốn hoàn toàn thay thế một hàm của lớp cơ sở, mà thay vào đó muốn thêm chức năng bổ sung vào nó. Trong trường hợp này, chúng ta có thể gọi hàm cơ sở bằng cú pháp: <tên_lớp_cơ_sở>::<tên_hàm>()

# Overriding behavior

```
class Base
{
public:
   void print()
   {
      cout << "Base" << endl;
   }
};

class Derived : public Base
{
public:
   void print()
   {
      Base::print();
      cout << "Derived" << endl;
   }
};
```

```
int main()
{
   Base base;
   base.print();

   Derived derived;
   derived.print();

   return 0;
}
```

***việc thêm chức năng ở đây gần như là :***
***ghi đè hàm = inport hàm của class cha <tên class cha>::<tên hàm> + chức năng bổ sung***

# Hiding inherited functionality

- Changing an inherited member's access level

Thay đổi mức truy cập của một thành viên được kế thừa

***chỉ có thể thay đổi lv access của đối tượng hoặc phương thức mà class con có thể truy cập (VD: class cha protected sang class con public hoặc private; class cha public sang class con private hoặc protected; nếu class cha là private thì class con không thể truy cập để thay đổi access lv)***

```
class Base
{
private:
    int m_value;

public:
    Base(int value)
        : m_value(value)
    {
    }

protected:
    void printValue() { std::cout << m_value; }
};
```

```
class Derived: public Base
{
public:
    Derived(int value)
        : Base(value)
    {
    }

    using Base::printValue;
};
```

# Hiding inherited functionality

- Hiding functionality

```
class Base
{
public:
    int m_value;

    Base(int value)
        : m_value(value)
    {
    }

    int getValue() { return m_value; }
};
```

```
class Derived : public Base
{
private:
    using Base::m_value;

public:
    Derived(int value)
        : Base(value)
    {
    }

    // mark this function as inaccessible
    int getValue() = delete;
};
```



***phân biệt:***
***int getValue() = delete; // xóa đi 1 hàm; class được kế thừa từ đó về sau sẽ không được kế thừa hàm ẩn này***
***int getValue() = 0; // khai báo hàm thuần ảo, kế thừa từ đó về sau cần định nghĩa lại hàm này***
***int getValue(){}; // khai báo hàm không làm gì, kế thừa từ đó về sau có thể gọi hàm này nhưng hàm không làm gì***

Section 5

# Multiple inheritance

# Multiple inheritance. Agenda

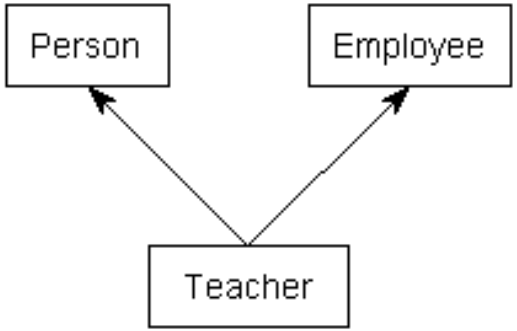
- Multiple inheritance in C++
- Problems with multiple inheritance
- Multiple inheritance discussion

# Multiple inheritance in C++



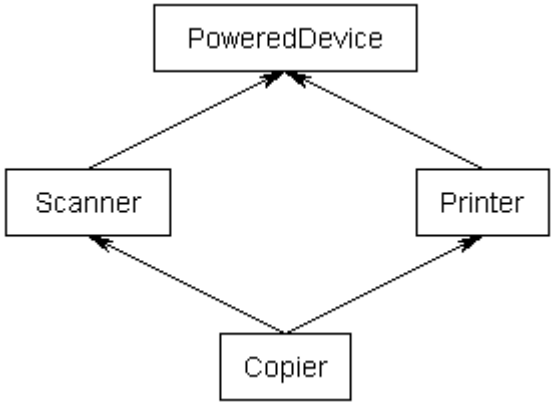
- C++ provides the ability to do multiple inheritance that enables a derived class to inherit members from more than one parent

▪ Trong C++, cung cấp khả năng đa kế thừa cho phép một lớp dẫn xuất kế thừa các thành viên từ nhiều hơn một lớp cha



# Problems with multiple inheritance

- Ambiguity can result when multiple base classes contain a function with the same name
- Diamond problem




Vấn đề của đa kế thừa
- Sự mơ hồ có thể xảy ra khi nhiều lớp cơ sở chứa một hàm có cùng tên
- Vấn đề kim cương (Diamond problem)

con trỏ của lớp cha trỏ đến đối tượng là lớp con:
VD: Shape *shape = new Triangle();
tham chiếu lớp cha đến đối tượng của lớp con:
VD: void renderShape(const Shape& shape) {
    shape.draw(); // Gọi phương thức draw() của đối tượng cụ thể
}

int main() {
    Triangle triangle;
    renderShape(triangle); // Triangle được tham chiếu dưới dạng Shape
    return 0;
}
Container lưu trữ con trỏ hoặc tham chiếu của lớp cha:
VD:    std::vector<Shape*> shapes;
    shapes.push_back(new Triangle());
    shapes.push_back(new Circle());

là các ứng dụng phổ biến nhất của tính đa hình, giúp tăng tính linh hoạt của mã nguồn, tuy nhiên có 1 hạn chế là chỉ có thể truy cập các đặc tính public hoặc virtual đã có ở lớp cha

# Multiple inheritance discussion

- Most of the problems that can be solved using multiple inheritance can be solved using single inheritance as well
  - Hầu hết các vấn đề có thể được giải quyết bằng đa kế thừa cũng có thể được giải quyết bằng kế thừa đơn
- Many object-oriented languages (Smalltalk, PHP, …) do not support multiple inheritance
  - Nhiều ngôn ngữ hướng đối tượng (Smalltalk, PHP, ...) không hỗ trợ đa kế thừa
- Many relatively modern languages (Java, C#, …) restrict classes to single inheritance of normal classes, but allow multiple inheritance of interface classes
  - Nhiều ngôn ngữ tương đối hiện đại (Java, C#, ...) hạn chế các lớp chỉ kế thừa từ một lớp thông thường, nhưng cho phép đa kế thừa từ các lớp giao diện
- Multiple inheritance makes the language too complex, and ultimately causes more problems than it fixes
  - Đa kế thừa làm cho ngôn ngữ trở nên quá phức tạp và cuối cùng gây ra nhiều vấn đề hơn là giải quyết được

**=> *Avoid multiple inheritance unless alternatives lead to more complexity***

=> Tránh đa kế thừa trừ khi các phương án thay thế dẫn đến sự phức tạp hơn

# References

- https://www.learncpp.com/cpp-tutorial/111-introduction-to-inheritance/
- https://www.learncpp.com/cpp-tutorial/112-basic-inheritance-in-c/
- https://www.learncpp.com/cpp-tutorial/113-order-of-construction-of-derived-classes/
- https://www.learncpp.com/cpp-tutorial/114-constructors-and-initialization-of-derived-classes/
- https://www.learncpp.com/cpp-tutorial/115-inheritance-and-access-specifiers/
- https://www.learncpp.com/cpp-tutorial/11-6-adding-new-functionality-to-a-derived-class/
- https://www.learncpp.com/cpp-tutorial/11-6a-calling-inherited-functions-and-overriding-behavior/
- https://www.learncpp.com/cpp-tutorial/11-6b-hiding-inherited-functionality/
- https://www.learncpp.com/cpp-tutorial/117-multiple-inheritance/

# Lesson Summary

- Inheritance in C++
- Constructors and initialization
- Inheritance and access specifiers
- Overriding and hiding functionality
- Multiple inheritance

# Thank you