C++ Training Course

Pure virtual function and Abstract class

Lesson Objectives

- Understand about polymorphism in OOP
- Understand about abstract in OOP

Section 1

Polymorphism

Agenda

- Type of Polymorphism
- Overloading, Override
- Virtual function
- Virtual table

Type of Polymorphism các loại đa hình

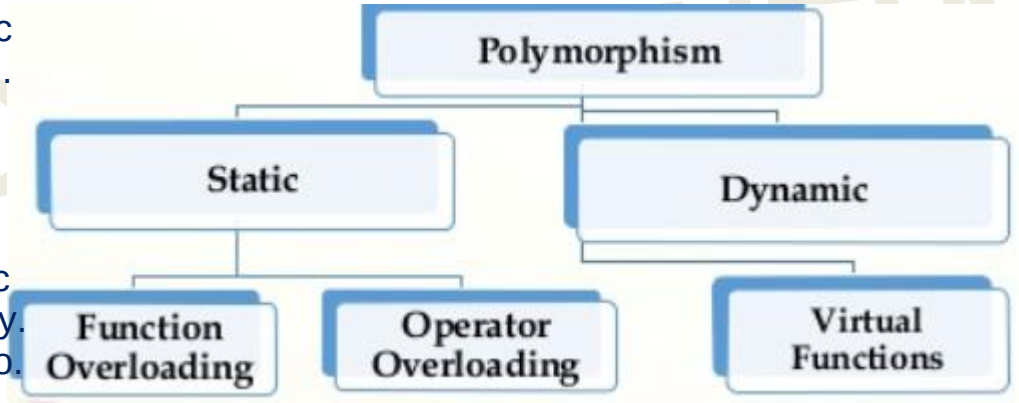
- *Static* polymorphism is also know as early binding and compile-time polymorphism. In static polymorphism memory will be allocated at compile-time.

▪ Đa hình tĩnh còn được gọi là ràng buộc sớm và đa hình tại thời điểm biên dịch. Trong đa hình tĩnh, bộ nhớ sẽ được cấp phát tại thời điểm biên dịch.

- *Dynamic* polymorphism is also know as late binding and run-time polymorphism. In dynamic memory will be allocated at run-time

▪ Đa hình động còn được gọi là ràng buộc muộn và đa hình tại thời điểm chạy. Trong đa hình động, bộ nhớ sẽ được cấp phát tại thời điểm chạy.

Đa hình tĩnh: Quyết định hàm nào sẽ được gọi được thực hiện tại thời điểm biên dịch. Thể hiện thông qua nạp chồng hàm và nạp chồng toán tử.
Đa hình động: Quyết định hàm nào sẽ được gọi được thực hiện tại thời điểm chạy. Thể hiện thông qua hàm ảo.



Method Overloading

- Whenever same method name is existing multiple times in the same class with different number of parameter or different order of parameter or different type of parameter is know as method overloading.

▪ Khi một phương thức có cùng tên tồn tại nhiều lần trong cùng một lớp nhưng có số lượng tham số khác nhau, thứ tự tham số khác nhau, hoặc kiểu tham số khác nhau, đó được gọi là nạp chồng phương thức.

```
int addInteger(int x, int y)
{
    return x + y;
}

double addDouble(double x, double y)
{
    return x + y;
}
```

Method Overriding

- Function overriding can’t be done within a class. For this we require a derived class and a base class. Ghi đè phương thức không thể thực hiện trong cùng một lớp. Để làm điều này, chúng ta cần một lớp dẫn xuất và một lớp cơ sở.
- Function that is redefined must have exactly the same declaration in both base and derived class, that mean same name, same parameter, same return type.

Phương thức được định nghĩa lại phải có khai báo chính xác giống nhau trong cả lớp cơ sở và lớp dẫn xuất, nghĩa là cùng tên, cùng tham số, cùng kiểu trả về.

```
class Base
{
public:
    virtual std::string_view getName() const { return "Base"; } // note addition of virtual keyword
};

class Derived: public Base
{
public:
    virtual std::string_view getName() const { return "Derived"; }
};

int main()
{
    Derived derived;
    Base &rBase{ derived };
    std::cout << "rBase is a " << rBase.getName() << '\n';

    return 0;
}
```

Final specifier từ khóa "Final"

- The final specifier can be used to tell the compiler to enforce this. If the user tries to override a function or class that has been specified as final, the compiler will give a compile error

Từ khóa final có thể được sử dụng để yêu cầu trình biên dịch thực thi điều này. Nếu người dùng cố gắng ghi đè một phương thức hoặc lớp đã được chỉ định là final, trình biên dịch sẽ báo lỗi biên dịch.

```
class A
{
public:
    virtual const char* getName() { return "A"; }
};

class B : public A
{
public:
    // note use of final specifier on following line -- that makes this function no longer overridable
    virtual const char* getName() override final { return "B"; } // okay, overrides A::getName()
};

class C : public B
{
public:
    virtual const char* getName() override { return "C"; } // compile error: overrides B::getName(), which is final
};
```

```
class A
{
public:
    virtual const char* getName() { return "A"; }
};

class B final : public A // note use of final specifier here
{
public:
    virtual const char* getName() override { return "B"; }
};

class C : public B // compile error: cannot inherit from final class
{
public:
    virtual const char* getName() override { return "C"; }
};
```

Overloading operator

- The process of making an operator to exhibit behavior in different instance is know as operator overloading.

thể hiện

Quá trình làm cho một toán tử thể hiện hành vi khác nhau trong các trường hợp khác nhau được gọi là nạp chồng toán tử.

- Only predefined operator can be overloaded

- Chỉ các toán tử được định nghĩa sẵn mới có thể được nạp chồng.

```
class Cents
{
private:
  int m_cents{};

public:
  Cents(int cents)
    : m_cents{ cents }
  {}

  // add Cents + Cents using a friend function
  friend Cents operator+(const Cents &c1, const Cents &c2);

  int getCents() const { return m_cents; }
};

// note: this function is not a member function!
Cents operator+(const Cents &c1, const Cents &c2)
{
  // use the Cents constructor and operator+(int, int)
  // we can access m_cents directly because this is a friend function
  return { c1.m_cents + c2.m_cents };
}
```

```
int main()
{
  Cents cents1{ 6 };
  Cents cents2{ 8 };
  Cents centsSum{ cents1 + cents2 };
  std::cout << "I have " << centsSum.getCents() << " cents.\n";

  return 0;
}
```

Virtual function

- A virtual function is a special type of function that, when called, resolves to the most-derived version of the function that exists between the base and derived class
  Một hàm ảo là một loại hàm đặc biệt mà khi được gọi, nó sẽ được giải quyết thành phiên bản dẫn xuất nhất của hàm đó tồn tại giữa lớp cơ sở và lớp dẫn xuất.
- A derived function is considered a match if it has the same signature (name, parameter types, and whether it is const) and return type as the base version of the function. Such functions are called overrides
  Một hàm dẫn xuất được coi là phù hợp nếu nó có cùng chữ ký (tên, các kiểu tham số, và liệu nó có là hàm const hay không) và kiểu trả về giống như phiên bản của hàm trong lớp cơ sở. Các hàm như vậy được gọi là hàm ghi đè (overrides).

```
class Base
{
public:
    virtual std::string_view getName() const { return "Base"; } // note addition of virtual keyword
};

class Derived: public Base
{
public:
    virtual std::string_view getName() const { return "Derived"; }
};
```

cơ chế của hàm ảo, Vtable? Vpointer ????

Virtual destructors

- Deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behaviour. To correct this situation, the base class should be defined with a virtual destructor

Xóa một đối tượng của lớp dẫn xuất bằng cách sử dụng con trỏ tới lớp cơ sở có hàm hủy không ảo dẫn đến hành vi không xác định. Để khắc phục tình huống này, lớp cơ sở nên được định nghĩa với một hàm hủy ảo

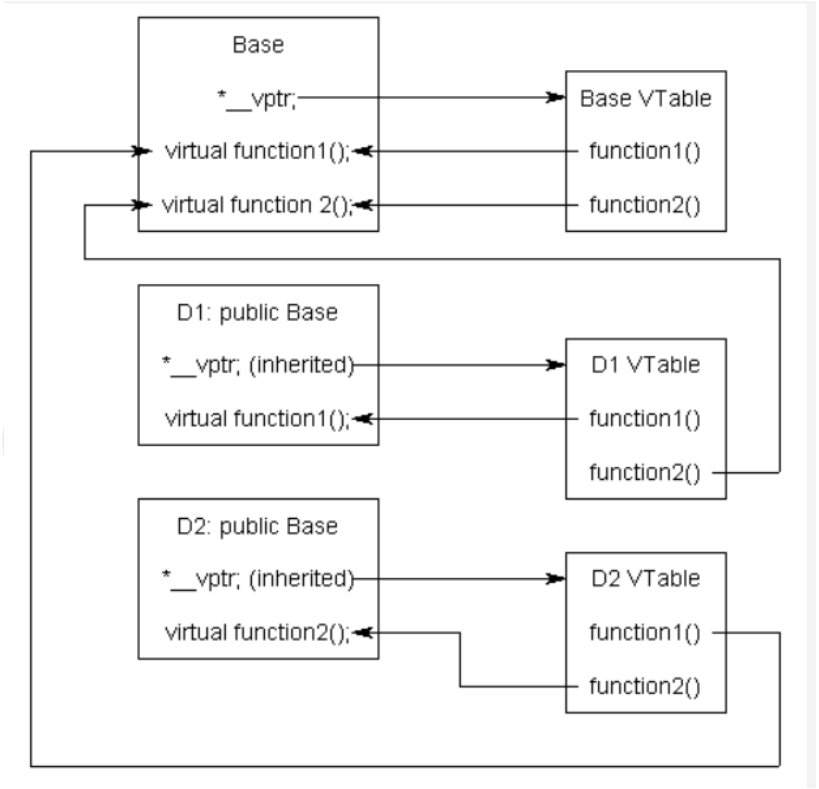
- Should we make all destructors virtual? The answer is whenever you are dealing with inheritance, you should make any explicit destructors virtual.

Chúng ta có nên làm cho tất cả các hàm hủy là ảo không? Câu trả lời là bất cứ khi nào bạn xử lý với kế thừa, bạn nên làm cho bất kỳ hàm hủy rõ ràng nào trở thành ảo.

có thể hiểu đây là đặc biệt với destructors // constructor không có hàm hủy ảo, không có hàm tạo ảo, không được private hàm tạo

```
#include <iostream>
class Base
{
public:
    virtual ~Base() // note: virtual
    {
        std::cout << "Calling ~Base()\n";
    }
};
```

Virtual table



Section 2

Abstraction

Agenda

- Pure virtual function
- Abstract class
- Object slicing

Pure virtual function

- Pure virtual function is virtual function with no definition. It start with virtual keyword and ends with = 0.

Hàm ảo thuần túy là một hàm ảo không có định nghĩa. Nó bắt đầu bằng từ khóa virtual và kết thúc bằng = 0.

- Syntax: virtual void display() = 0

▪ Cú pháp: virtual void display() = 0

Abstract class

- Abstract class is a class which contains at least one pure virtual function in it
  - Lớp trừu tượng là một lớp chứa ít nhất một hàm ảo thuần túy.
- Abstract class is used to provide an Interface for its sub class
  - Lớp trừu tượng được sử dụng để cung cấp một giao diện cho các lớp con của nó.
- Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class
  - Các lớp kế thừa từ một lớp trừu tượng phải cung cấp định nghĩa cho các hàm ảo thuần túy, nếu không chúng cũng sẽ trở thành lớp trừu tượng.
- A class with at least one pure virtual function or abstract function is called abstract class
  - Một lớp có ít nhất một hàm ảo thuần túy hoặc hàm trừu tượng được gọi là lớp trừu tượng.
- Pure virtual function is also know as abstract function
  - Hàm ảo thuần túy cũng được biết đến là hàm trừu tượng.
- We can't create object of abstract class so we need to create sub class of abstract class to use that function
  - Chúng ta không thể tạo đối tượng của lớp trừu tượng, vì vậy chúng ta cần tạo lớp con của lớp trừu tượng để sử dụng các hàm đó.

Object slicing

- Object slicing occurs when you assign derived class objects using base class variables. The result of this is that the derived class part of an object can be “sliced off” and lost.

Việc object slicing xảy ra khi bạn gán đối tượng của lớp dẫn xuất bằng biến của lớp cơ sở. Kết quả của việc này là phần của đối tượng thuộc lớp dẫn xuất có thể bị "cắt đi" và mất đi.

```
class A {
   int foo;
};

class B : public A {
   int bar;
};
```

```
B b;

A a = b;
```

Dynamic casting

- Object slicing occurs when you assign derived class objects using base class variables. The result of this is that the derived class part of an object can be “sliced off” and lost.

```
class A {
   int foo;
};

class B : public A {
   int bar;
};
```

```
B b;

A a = b;
```

References

- https://www.learncpp.com/

Static Cast (static_cast): // trả về lỗi lúc biên dịch
Dùng để chuyển đổi kiểu tường minh và an toàn trong phạm vi các kiểu dữ liệu có liên quan (related types),
ví dụ như từ con trỏ lớp cơ sở sang con trỏ lớp dẫn xuất, hoặc giữa các kiểu dữ liệu nguyên thủy như int sang double.
Ví dụ: double d = 3.14; int i = static_cast<int>(d);

Dynamic Cast (dynamic_cast): // trả về lỗi lúc runtime
Dùng để chuyển đổi giữa các lớp trong kế thừa đa hình (polymorphic classes).Kiểm tra tính hợp lệ của việc chuyển đổi và trả về nullptr nếu không thể thực hiện.Sử dụng phổ biến trong việc chuyển đổi con trỏ hoặc tham chiếu của lớp cơ sở sang lớp dẫn xuất và ngược lại.
Ví dụ: Derived* d = dynamic_cast<Derived*>(base_ptr); if (d) { // làm gì đó với d }

Const Cast (const_cast):
Dùng để loại bỏ hoặc thêm const hoặc volatile cho một biến.Thường được sử dụng để giả lập việc thay đổi giá trị của biến không const hoặc volatile khi cần thiết.Ví dụ: const int* p = # int* q = const_cast<int*>(p); *q = 5;

Reinterpret Cast (reinterpret_cast):
Dùng để chuyển đổi giữa các kiểu không liên quan nhau, chẳng hạn như từ con trỏ sang số nguyên hoặc ngược lại.Không an toàn và không nên sử dụng nếu không cần thiết, do có thể dẫn đến lỗi logic hoặc không xác định.
Ví dụ: int i = 10; double* dp = reinterpret_cast<double*>(&i);

Functional Cast (type(value) hoặc type{value}):
Được sử dụng tương tự như static_cast, nhưng có thể xem xét cho phép việc chuyển đổi ngầm định và tường minh.Được khuyến khích sử dụng thay thế cho các loại cast cú pháp khác, vì nó dễ đọc và hiểu hơn.

Lesson Summary

- Polymorphism, Abstraction
- Early binding, late binding
- Virtual function and pure virtual function
- Virtual table
- Object slicing

Thank you