

INHERITANCE and POLYMORPHISM

Instructor: <Name of Instructor>

© FPT Software

1

Latest updated by: HanhTT1

Agenda

- ❑ Inheritance
- ❑ Polymorphism



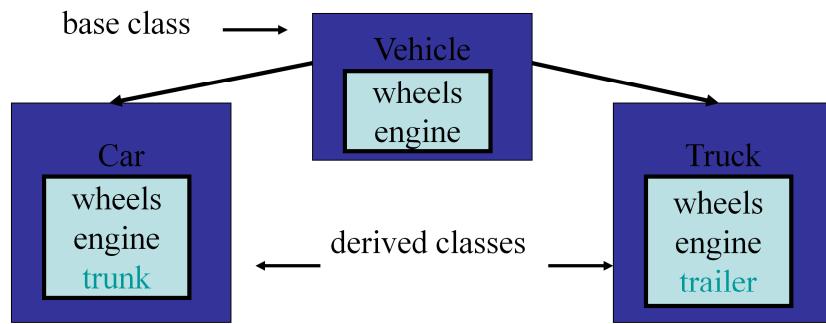
KẾ THỪA (Inheritance)

© FPT Software

3

Lớp cơ sở và lớp dẫn xuất

- Chúng ta có thể chia các lớp thành những lớp nhỏ hơn.
- Một lớp có thể được các lớp khác kế thừa các thuộc tính và phương thức
 - Lớp cơ sở (Base Class)
 - Lớp dẫn xuất (Derived Class)



© FPT Software

4

Base Class Date

```
class Date {  
    private:  
        int day, month, year; // member data  
    public:  
        Date(int d, int m, int y); // member functions  
        Date operator+(int days);  
        int operator-(Date &date);  
        bool LeapYear();  
        int DaysInMonth();  
        void Display();  
};
```

Derived Class DateTime

```
class DateTime : public Date {  
private:  
    int hours, minutes;      // bỗ sung thêm member data  
public:                   // bỗ sung thêm member functions  
    DateTime(int d, int m, int y, int h, int mi); // constructor  
    void SetTime(int h, int m);  
    void AddMinutes(int m);  
    void AddHours(int h);  
    void Display();          // overrides Date::Display()  
};
```

Derived Class DateTime ...

```
DateTime::DateTime(int d, int m, int y, int h, int mi)
    : Date(d,m,y) , hours(h) , minutes(mi)           // vien dan Date Constructor
{
}
void DateTime::Display() {
    Date::Display();                                // goi Display() cua lop Date
    cout << " " << hours << ":" << minutes;
}
void DateTime::AddHours(int h) {
    hours+=h;

    if (hours > 23) {
        hours-=24;
        *this++;                                     // su dung toan tu ++
    }
}
```

Derived Class DateTime ...

```
Date justdt(12,6,1998);           // constructor Date
DateTime dt(13,2,1997,20,15);     //constructor DateTime
dt = dt+5;
if (dt.LeapYear()) {
    cout << "Date is in a leapyear" << endl;
} else {
    cout << "Date is not in a leapyear" << endl;
}
dt.AddHours(2);
dt.Display();                     // Display() của lớp DateTime
justdt.Display();                // Display() của lớp Date
```

Derived Class Constructors

```
DateTime::DateTime(int d, int m, int y, int h, int m1) {  
    Date(d,m,y) ;  
    hours=h;  
    minutes=m1;  
}  
DateTime dt(12,4,1999,17,30);
```

- Chương trình dịch sẽ tạo ra một đối tượng kiểu DateTime và sau đó gọi hàm tạo DateTime() để khởi tạo nó.
- Hàm tạo DateTime() sẽ gọi hàm tạo Date() để khởi tạo day, month, year
- Hàm tạo DateTime khởi tạo hours, minutes của chính nó.

Overriding Member Functions

```
class Date {  
public:  
    void Display();  
};  
  
class DateTime : public Date {  
public:  
    void Display();           // overrides Date::Display()  
};
```

Overriding Member Function...

- Khi có các hàm cùng tên và cùng danh sách tham số trong base class và derived class thì hàm trong derived class sẽ được thực hiện.

```
DateTime dt(12,3,2001,12,30);  
dt.Display(); // Display() của DateTime
```

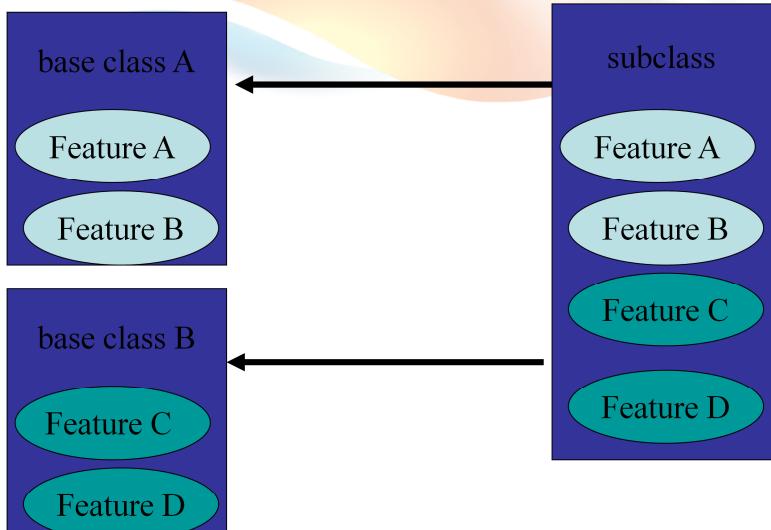
- Ta có thể xác định rõ cần sử dụng member function của lớp nào.

```
void DateTime::Display() {  
    Date::Display();  
    cout << " " << hours << ":" << minutes;  
}
```

Truy nhập vào các thành phần của lớp cơ sở

- Từ khoá (Access Specifier) private, protected và public xác định khả năng truy nhập tới các thành phần của lớp cơ sở.
 - private: chỉ được truy nhập trong lớp cơ sở
 - protected: chỉ được truy nhập trong lớp cơ sở và lớp dẫn xuất của nó
 - public: được truy nhập ở cả bên ngoài phần định nghĩa lớp.

Đa thừa kế



© FPT Software

13

Đa thừa kế ...

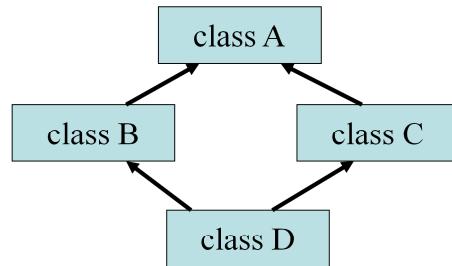
```
class Date {  
    private:  
        int day, month, year;  
    ...  
};  
class Time {  
    private:  
        int hours, minutes;  
    ...  
};  
class DateTime : public Date, public Time {  
public:  
    DateTime(int d, int m, int y, int h, int mi) : Date(d,m,y), Time(h, mi)  
    {};  
    ...  
};
```

Nhập nhằng trong đa thừa kế

```
class Date {  
    void add(int days);  
};  
class Time {  
    void add(int minutes);  
};  
DateTime dt(13,2,1998,23,10);  
dt.add(3);      // báo lỗi  
dt.Date::add(4); // sử dụng của lớp Date  
dt.Time::add(5); // sử dụng của lớp Time
```

Nhập nhằng trong đa thừa kế

```
class A { public: void F(); };
class B : public A { ... };
class C : public A { ... };
class D : public B, public C {};
D d;
d.F(); // không dịch
```





ĐA HÌNH

(Polymorphism)

© FPT Software

17

Đa hình

- Đa hình là một hàm có nhiều hình thức thể hiện khác nhau tùy từng hoàn cảnh cụ thể
- Đa hình là một đặc trưng của ngôn ngữ lập trình hướng đối tượng
- Phân loại:
 - Đa hình tĩnh
 - Đa hình động

Đa hình tĩnh

- Overload một phương thức, tức là tạo ra nhiều phương thức có cùng tên nhưng khác nhau về danh sách tham số
- Overrid một phương thức, tức là tạo ra một phương thức trong lớp dẫn xuất có cùng prototype với một phương thức trong lớp cơ sở.

Ví dụ: Overloading Function

```
class Mammal {  
public:  
    void Move() {cout << "Mamal moves 1 step";}  
    void Move(int d) {cout << "Mamal moves <<d<< steps";}  
};  
  
int main() {  
    Mamal m;  
    m.Speak();  
    m.Speak(10);  
}
```

Ví dụ: Overriding Function

```
class Mammal
{
    public:
        void Speak() {cout << "Mamal";}
};

class Dog : public Mammal
{
    public:
        void Speak() {cout << "Dog";}
};

int main()
{
    Mamal m;
    Dog d;
    m.Speak();           // Mamal
    d.Speak();           // Dog
}
```

© FPT Software

21

Ẩn phương thức của lớp cơ sở

- Nếu lớp cơ sở có một phương thức bị chồng và lớp dẫn xuất lại override phương thức này, thì phương thức của lớp dẫn xuất sẽ ẩn tất cả các phương thức của lớp cơ sở có cùng tên với nó.

Ví dụ: Án phương thức của lớp cơ sở

```
class Mammal {  
    public:  
        void Move() {cout << "Mamal moves 1 step";}  
        void Move(int d) {cout << "Mamal moves <<d<< steps";}  
};  
class Dog : public Mammal {  
    public:  
        void Move() {cout << "Dog moves 1 step";}  
};  
int main() {  
    Mamal m;  
    Dog d;  
    m.Move();  
    m.Move(10);  
    d.Move();  
    d.Move(10); // Lỗi  
}
```

Đa hình động

- Được thể hiện thông qua hàm ảo
- Từ khoá *virtual* xác định hàm thành phần của lớp cơ sở sẽ bị override bởi lớp dẫn xuất.
- Khi lớp cơ sở định nghĩa các hàm ảo thì C++ sẽ tìm kiếm hàm đó trong lớp dẫn xuất trước, sau đó mới đến lớp cơ sở.

```
#include <iostream.h>
class base {
public:
    void a(void) { cout << "base::a called\n"; }
    virtual void b(void){ cout << "base::b called\n"; }
    virtual void c(void) { cout << "base::c called\n"; }
};
class derived: public base {
public:
    void a(void) { cout << "derived::a called\n"; }
    void b(void) { cout << "derived::b called\n"; }
};
void do_base(base& a_base) {
    cout << "Call functions in the base class\n";
    a_base.a();
    a_base.b();
    a_base.c();
}
main() {
    derived a_derived;
    cout << "Calling functions in the derived class\n";
    a_derived.a();
    a_derived.b();
    a_derived.c();
    do_base(a_derived);
}
```

Ví dụ: Hàm ảo

```
void do_base(base& a_base) {  
    cout << "Call functions in the base class\n";  
    a_base.a();  
    a_base.b();  
    a_base.c();  
}  
main() {  
    derived a_derived;  
    cout << "Calling functions in the derived class\n";  
    a_derived.a();  
    a_derived.b();  
    a_derived.c();  
    do_base(a_derived);  
    return (0);  
}
```

Khi nào sử dụng hàm ảo?

- Lớp Parent và Child cùng có phương thức f
- Khai báo một con trỏ thuộc kiểu của lớp Parent
 - Parent* p;
- Con trỏ này trỏ đến đối tượng của lớp Child
 - p = new Child;
- Sau đó, thực hiện lời gọi
 - p->f;
- Kết quả: f của lớp Parent sẽ được viễn dẫn
- Nếu f được khai báo là hàm ảo trong lớp Parent thì f của lớp Child sẽ được viễn dẫn.

Ví dụ: Không sử dụng hàm ảo

```
class Mammal {  
    public:  
        void Move() {cout << "Mammal moves 1 step";}  
};  
class Dog : public Mammal {  
    public:  
        void Move() {cout << "Dog moves 1 step";}  
};  
int main() {  
    Mamal* p = new Dog();  
    p->Move(); // "Mammal moves 1 step"  
}
```

Ví dụ: Sử dụng hàm ảo

```
class Mammal {  
    public:  
        void virtual Move() {cout << "Mammal moves 1 step";}  
};  
class Dog : public Mammal {  
    public:  
        void Move() {cout << "Dog moves 1 step";}  
};  
int main() {  
    Mamal* p = new Dog();  
    p->Move(); // "Dog moves 1 step"  
}
```

Hàm thuần ảo

- Nếu khai báo hàm ảo như sau:
virtual void send_it(void) = 0;
 - “=0” có nghĩa là hàm thuần ảo (pure virtual function). Tức là, nó sẽ không được gọi một cách trực tiếp.
 - Hàm thuần ảo **phải được** overload trong subclass
 - Hàm ảo **có thể được** overload trong subclass

Lớp trừu tượng (abstract class)

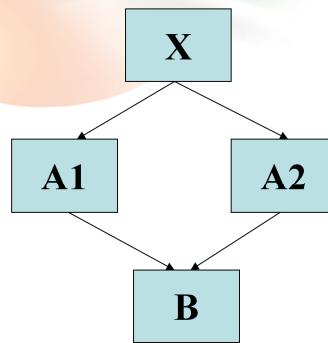
- Hàm thuần ảo được khai báo trong lớp sẽ làm cho lớp đó trở thành lớp cơ sở trừu tượng.
- Lớp cơ sở trừu tượng là lớp cơ sở không có đối tượng nào và chỉ sử dụng để cho các lớp khác kế thừa.
- Một lớp chỉ đóng vai trò là lớp cơ sở cho các lớp khác và không có đối tượng cụ thể của nó được tạo ra thì gọi là lớp trừu tượng.

Ví dụ: hàm thuần ảo

```
class Mammal {  
    public:  
        virtual void Move() = 0;  
};  
class Dog : public Mammal {  
    public:  
        void Move() {cout << "Dog moves 1 step";}  
};  
void main() {  
    Dog p;  
    p.Move();           // "Dog moves 1 step"  
    Mammal m;          // "Lỗi"  
    m.Move();  
}
```

Lớp cơ sở ảo

Nhập nhằng trong đa kế thừa



- Lớp B sẽ có hai bản sao của tất cả các thành phần từ lớp X.
- Khi gọi đến một trong những thành phần này từ lớp B, chương trình dịch sẽ thông báo lỗi.

Giải quyết xung đột

- Gọi tường minh
 - Ví dụ lớp X có phương thức x được thừa kế
 - Lời gọi x từ một đối tượng của lớp B
B b;
b.A1 :: x;
b.A2 :: x;
- Sử dụng lớp cơ sở ảo

Lớp cơ sở ảo

- Lớp cơ sở ảo đảm bảo trong lớp dẫn xuất chỉ tạo ra một bản sao của các thành phần được thừa kế từ lớp cơ sở.

```
#include <iostream.h>
class Automobile
{
    private:
        char *Manufacturer;
        double Price;
    public:
        Automobile(){}
        void GetPrice()
        {
            cin >> Price;
        }
};
class Car : public virtual Automobile
{
    private:
        int Price;
    public:
        Car(){}
        //...
};
class SportsVehicle : public virtual Automobile
{
    private:
        int Price;
    public:
        SportsVehicle(){}
        //...
};
class SportsCar : public Car, public SportsVehicle
{};
void main()
{
    SportsCar SC;
    SC.GetPrice();
}
```

Lớp cơ sở ảo

- Hàm tạo của lớp cơ sở chỉ được gọi trong hàm tạo của lớp dẫn xuất trực tiếp từ nó.
- Hàm tạo của lớp cơ sở ảo thì được gọi ở tất cả các lớp dẫn xuất nó.
- Quy tắc như sau:
 - Hàm tạo của lớp cơ sở ảo được gọi đầu tiên
 - Tiếp theo đó là hàm tạo của các lớp dẫn xuất trực tiếp
 - ...

```
class Temporary : public Employee
{
    int DaysWorked;
public:
    Temporary(){}
    Temporary(char *EName, int Days) : Employee(EName)
    {
        DaysWorked = Days;
    }
};
class Clerk : public Employee
{
    char *Location;
public:
    Clerk(){}
    Clerk(char *EName, char *City) : Employee(EName)
    {
        Location = City;
    }
};
class TempClerk : public Temporary, public Clerk
{
    int Salary;
public:
    TempClerk(){}
    TempClerk(char *EName, char *City, int Days, int Sal)
        : Clerk(EName, City), Temporary(EName, Days)
    {
        Salary = Sal;
    }
}
```

```
Class Temporary : virtual public Employee
{
    int DaysWorked;
public:
    Temporary(){}
    Temporary(char *EName, int Days) : Employee(EName)
    {
        DaysWorked = Days;
    }
};

class Clerk : virtual public Employee
{
    char *Location;
public:
    Clerk(){}
    Clerk(char *EName, char *City) : Employee(EName)
    {
        Location = City;
    }
};

class TempClerk : public Temporary, public Clerk
{
    int Salary;
public:
    TempClerk(){}
    TempClerk(char *EName, char *City, int Days, int Sal)
        : Employee(EName), Clerk(EName, City), Temporary(EName, Days)
    {
        Salary = Sal;
    }
};

void main()
{
    TempClerk TC("ALLEN", "LA", 22, 2000);
}
```

Strategy Pattern in C++

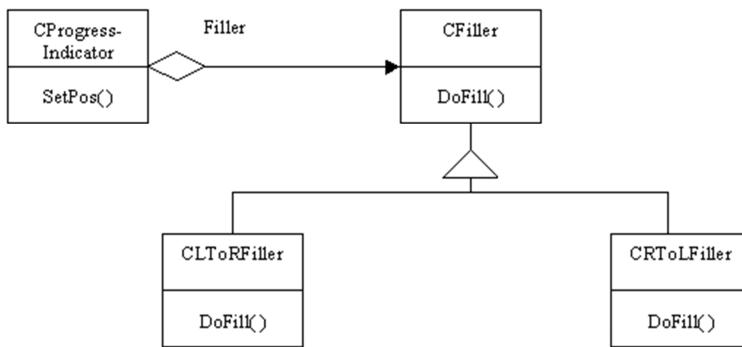
- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

Strategy pattern là chiến lược chọn giải thuật tốt nhất tùy vào ngữ cảnh (context) để xử lý vấn đề nếu có nhiều giải thuật để lựa chọn xử lý.

Có nhiều giải thuật để xử lý một vấn đề, mỗi giải thuật được đóng gói vào một class phù hợp. Strategy pattern sẽ thực hiện lựa chọn các giải thuật khác nhau một cách độc lập tùy theo ngữ cảnh để xử lý

Strategy Pattern in C++

- **Sample:** It is usually a rectangular window that is gradually filled, from left to right, with the highlight color as the operation progresses. The range and the current position are used to determine the percentage of the progress indicator to fill with the highlight color.



© FPT Software

41

Có 2 cách để giải quyết được vấn đề:

- Fill từ trái qua phải
 - Fill từ phải qua trái
- Tùy vào vị trí ban đầu.

`CProgressIndicator` là context class

`CFiller` là strategy object.

Strategy Pattern in C++

- Classify:
 - Strategy object as a required parameter to the context
 - Strategy object as an optional parameter to the context
 - Strategy as a template class parameter to the context

Strategy Pattern in C++

- Strategy object as a required parameter to the context:
 - In this approach, the progress indicator (Context) takes a filler (Strategy) object as a parameter in its constructor and maintains a reference to it. The progress indicator delegates the request to the filler object when SetPos method is called.

Strategy Pattern in C++

```
// Forward declaration for CFiller class
class CFiller;

// Class declaration for CProgressIndicator
class CProgressIndicator
{
    // Method declarations
public:
    CProgressIndicator(CFiller *);
    INT SetPos(INT);
    INT SetFiller(CFiller *);
    ...
    ...
    // Data members
protected:
    CFiller * m_pFiller;
};

// CProgressIndicator - Implementation
CProgressIndicator ::CProgressIndicator(CFiller * pFiller)
{
    // Validate pFiller
    ASSERT(pFiller != NULL);
    m_pFiller = pFiller;
}
```

CProgressIndicator is Context
CFiller is Strategy object

© FPT Software

44

Strategy Pattern in C++

```
INT CProgressIndicator ::SetPos(INT nPos)
{
    // Some initialization code before forwarding the request to filler object
    ...
    ...
    // Request forwarding to filler object
    INT nStatus = m_pFiller->DoFill(...);
    ...
    ...
    return nStatus;
}

INT * CProgressIndicator ::SetFiller(CFiller * pFiller)
{
    // Validate pFiller
    ASSERT(pFiller != NULL);
    // Set new filler object
    m_pFiller = pFiller;
    return 0;
}
```

Strategy Pattern in C++

- Strategy object as an optional parameter to the Context
 - This approach is similar to the first approach, but the filler object (Strategy) is taken as an optional parameter when progress indicator (Context) is created. The progress indicator creates a default filler object (Left to Right filler), if the application did not specify the filler object during construction

Strategy Pattern in C++

```
// CProgressIndicator - Implementation
CProgressIndicator ::CProgressIndicator(CFiller * pFiller)
{
    // Check and create filler object
    if(pFiller == NULL)
    {
        // Create a default Left to Right filler object
        m_pFiller = new CLToRFiller;
        m_bCreated = TRUE;
    }
    else
    {
        m_pFiller = pFiller;
        m_bCreated = FALSE;
    }
}

CProgressIndicator::~CProgressIndicator()
{
    // Delete filler object, only if it is created by the progress indicator
    if(m_bCreated == TRUE)
    {
        delete m_pFiller;
    }
}
```

Default object

© FPT Software

47

Strategy Pattern in C++

```
INT CProgressIndicator ::SetPos(INT nPos)
{
    // Some initialization code before forwarding the request to CFiller object
    ASSERT(m_pFiller != NULL);
    ...
    ...
    // Request forwarding to CFiller object
    INT nStatus = m_pFiller->DoFill(...);
    ...
    ...
    return nStatus;
}

INT * CProgressIndicator ::SetFiller(CFiller * pFiller)
{
    // Validate Filler object
    ASSERT(pFiller != NULL);
    // Delete filler object, only if it is created by the progress indicator
    if(m_bCreated == TRUE)
    {
        delete m_pFiller;
        m_bCreated = FALSE;
    }
    // Set new filler object
    m_pFiller = pFiller;
    return 0;
}
```

© FPT Software

48

Strategy Pattern in C++

- Strategy as a template class parameter:
 - If there is no need to change the filler class (Strategy) at run time, it can be passed as a template parameter to the progress indicator (Context) class at compile time.

Strategy Pattern in C++

```
template <class TFiller> class CProgressIndicator
{
    // Method declarations
public:
    INT SetPos(INT);
    ...
    ...
    // Data members
protected:
    TFiller m_theFiller;
};

// CProgressIndicator - Implementation

INT CProgressIndicator ::SetPos(INT nPos)
{
    // Some initialization code before forwarding the request to CFiller
    // object
    ...
    ...
    // Request forwarding to CFiller object
    INT nStatus = m_theFiller.DoFill(...);
    ...
    ...
    return nStatus;
}

// Application code using CProgressIndicator

CProgressIndicator<CLToRFiller> LtoRFillerObj;
```

Fillter is used as Template

© FPT Software

50

Composite Pattern in C++

- the **composite pattern** is a partitioning design pattern. The composite pattern describes that a group of objects are to be treated in the same way as a single instance of an object.
- Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

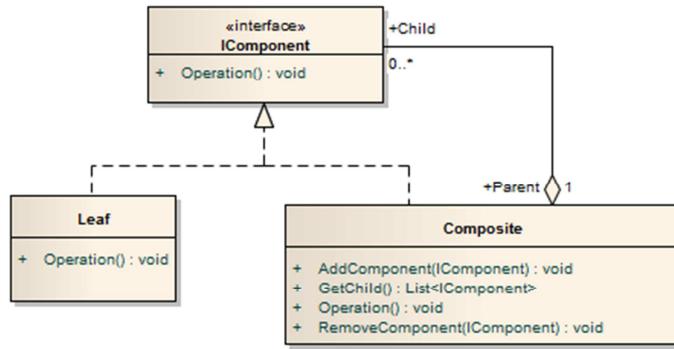
Composite design pattern là một cấu trúc cây và yêu cầu mỗi phần tử trong cấu trúc cây đó phải thực hiện một nhiệm vụ.

Composite design pattern phân loại các phần tử trong cây bao gồm composite và leaf.

Một composite có thể có các yếu tố khác bên dưới nó

Một leaf thì không có yếu tố nào khác dưới nó.

Composite Pattern in C++



- ★ IComponent that is interface, Composite class and Leaf class must be implement.
- ★ Method Operation() is common method

Composite Pattern in C++

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

// 2. Create an "interface" (lowest common denominator)
class Component
{
public:
    virtual void traverse() = 0;
};

class Leaf: public Component
{
    // 1. Scalar class  3. "isa" relationship
    int value;
public:
    Leaf(int val)
    {
        value = val;
    }
    void traverse()
    {
        cout << value << ' ';
    }
};
```

© FPT Software

53

Composite Pattern in C++

```
class Composite: public Component
{
    // 1. Vector class  3. "isa" relationship
    vector < Component * > children; // 4. "container" coupled to the interface
public:
    // 4. "container" class coupled to the interface
    void add(Component *ele)
    {
        children.push_back(ele);
    }
    void traverse()
    {
        for (int i = 0; i < children.size(); i++)
        // 5. Use polymorphism to delegate to children
        children[i]->traverse();
    }
};
```

Composite Pattern in C++

```
int main()
{
    Composite containers[4];

    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 3; j++)
            containers[i].add(new Leaf(i *3+j));

    for (i = 1; i < 4; i++)
        containers[0].add(&(containers[i]));

    for (i = 0; i < 4; i++)
    {
        containers[i].traverse();
        cout << endl;
    }
}
```

Factory Pattern in C++

- Basically a Factory consists of an interface class which is common to all of the implementation classes that the factory will create. Then you have the factory class which is usually a singleton class that spawns instances of these implementation classes.

Factory pattern dùng để giải quyết vấn đề tạo một đối tượng mà không cần thiết chỉ ra một cách chính xác lớp nào sẽ được tạo.

Factory pattern giải quyết vấn đề này bằng cách định nghĩa một phương thức cho việc tạo đối tượng, và các lớp con thừa kế có thể override để chỉ rõ đối tượng nào sẽ được tạo.

Factory Pattern in C++

- Implement Abstract Interface Class

```
class IAnimal
{
public:
    virtual int GetNumberOfLegs() const = 0;
    virtual void Speak() = 0;
    virtual void Free() = 0;
};

typedef IAnimal* ( __stdcall *CreateAnimalFn ) (void);
```

Factory Pattern in C++

- Implementing classes that implement the IAnimal interface

```
// IAnimal implementations
class Cat : public IAnimal
{
public:
    int GetNumberOfLegs() const { return 4; }
    void Speak() { cout << "Meow" << endl; }
    void Free() { delete this; }

    static IAnimal * __stdcall Create() { return new Cat(); }
};

class Dog : public IAnimal
{
public:
    int GetNumberOfLegs() const { return 4; }
    void Speak() { cout << "Woof" << endl; }
    void Free() { delete this; }

    static IAnimal * __stdcall Create() { return new Dog(); }
};
```

Factory Pattern in C++

□ Factory Class Declaration

```
// Factory for creating instances of IAnimal
class AnimalFactory
{
private:
    AnimalFactory();
    AnimalFactory(const AnimalFactory &) { }
    AnimalFactory &operator=(const AnimalFactory &) { return *this; }

    typedef map<string, CreateAnimalFn> FactoryMap;
    FactoryMap m_FactoryMap;
public:
    ~AnimalFactory() { m_FactoryMap.clear(); }

    static AnimalFactory *Get()
    {
        static AnimalFactory instance;
        return &instance;
    }

    void Register(const string &animalName, CreateAnimalFn pfnCreate);
    IAnimal *CreateAnimal(const string &animalName);
};
```

© FPT Software

59

Factory Pattern in C++

□ Factory Class Implementation

```
AnimalFactory::AnimalFactory()
{
    Register("Cat", &Cat::Create);
    Register("Dog", &Dog::Create);
}

void AnimalFactory::Register(const string &animalName, CreateAnimalFn pfnCreate)
{
    m_FactoryMap[animalName] = pfnCreate;
}

IAnimal *AnimalFactory::CreateAnimal(const string &animalName)
{
    FactoryMap::iterator it = m_FactoryMap.find(animalName);
    if( it != m_FactoryMap.end() )
        return it->second();
    return NULL;
}
```

Factory Pattern in C++

```
int main( int argc, char **argv )
{
    IAnimal *pAnimal = NULL;
    string animalName;

    while( pAnimal == NULL )
    {
        cout << "Type the name of an animal or 'q' to quit: ";
        cin >> animalName;

        if( animalName == "q" )
            break;

        IAnimal *pAnimal = AnimalFactory::Get()->CreateAnimal(animalName);
        if( pAnimal )
        {
            cout << "Your animal has " << pAnimal->GetNumberOfLegs() << " legs." << endl;
            cout << "Your animal says: ";
            pAnimal->Speak();
        }
        else
        {
            cout << "That animal doesn't exist in the farm! Choose another!" << endl;
        }
        if( pAnimal )
            pAnimal->Free();
        pAnimal = NULL;
        animalName.clear();
    }
    return 0;
}
```

© FPT Software

61



© FPT Software

62