Программирование и алгоритмизация

Санкт-Петербургский государственный политехнический университет

03 марта 2015

Отношения между классами

Бывают следующих типов:

- является (is a). Открытое public-наследование;
- содержит (has a). Нет наследования, объект содержит объект другого класса;
- подобен (as a). Закрытое (private или protected) наследование.

Для корректного построения иерархии, программист должен представлять различия между типами взаимоотношений и применять их исключительно по назначению.

Виды наследования

- одиночное;
- множественное

Одиночное наследование

```
class A //базовый класс
{ //список членов класса A
};

class B : public A
{ //производный класс
};
```

Такое объявление говорит компилятору, что класс B включает в себя весь класс A, как составную часть. В зависимости от спецификаторов доступа методы класса B имеют право обращаться к членам класса A или нет.

Спецификаторы при наследовании

Спецификатор при наследовании, позволяет задать для базового класса более жесткие ограничения на свойства видимости переменных.

Пример наследования спецификаторов

```
class A
{ private:   int a;
protected: int b;
public: int c;
};
class B : public A //если private, protected?
{
public: B() {
a = 0;
b = 0;
c = 0; }//???
};
int main()
{ B b;
b.a = 1; //???
b.b = 2; //???
b.c = 3; //???  }
```

Порядок вызова конструкторов

```
class A{};
class B : public A {};
class C : public B {};
C::C()
{ //вызов конструктора класса B
//тело конструктора C
}
B::B()
{ //вызов конструктора класса A
//тело конструктора B
}
A::A()
{
//тело конструктора A
}
```

Порядок вызова деструкторов

```
class A{};
class B : public A {};
class C : public B {};
C::~C()
{
//тело деструктора C
}//вызов деструктора класса B
B::B()
{
//тело деструктора B
}//вызов деструктора класса A
A::A()
{ //тело деструктора A }
```

При вызове оператора delete, вначале вызываются деструкторы, а потом освобождается динамичская память.

Иерархия животных

```
//animal.h
class Animal
{
protected:
int m_age;
char *m_name;
public:
Animal(int age, const char *name);
};
//dog.h
class Dog : public Animal
{
bool m_hasMaster;
char *m_masterName;
public:
Dog(int age, bool master, const char *name, const
    char *mName); };
```

Передача параметров конструктору базового класса

```
//animal.cpp
Animal::Animal(int age, char *name)
{
}

//dog.cpp
Dog::Dog(int age, bool master, const char *name,
    const char *masterName) : Animal(age, name)
{
m_hasMaster = master;
m_masterName = new char [strlen(masterName) + 1];
strcpy(m_masterName, masterName);
}
```

Передача параметров конструктору копирования

```
//animal.cpp
Animal::Animal(const Animal &a)
{
m_age = a.m_age;
m_name = new char [strlen(a.m_name) + 1];;
strcpy(m_name, a.m_name);
}

//dog.cpp
Dog::Dog(const Dog &d) : Animal(d)
{
m_hasMaster = d.m_hasMaster;
m_masterName = new char [strlen(d.masterName) + 1];
strcpy(m_masterName, d.m_masterName);
}
```

Правила ООП

Открытое наследование в C++ моделирует следующее утверждение:

- производный класс — разновидность базового класса;
- все что применимо к базовому классу, должно быть применимо к производному классу;
- везде, где может быть использован объект A, может быть использован объект B, поскольку объект B содержит базовую часть A.

Правила открытого наследования

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

void f1(A, A); //A &, A*
void f2(B, B); //B &, B*
int main()
{
A a; B b;
f1( a, b ); //???
f2( a, b ); //???
}
```

Раннее связывание

```
class Animal
{
public:
void Voice() const { cout << "???"; }
};
class Dog : public Animal
{
void Voice() const { cout << "Gav"; }
};
class Cat : public Animal
{
void Voice() const { cout << "Miau"; }
};
```

Раннее связывание

```
void F(const Animal *p)
{ p->Voice(); }

int main()
{
Animal a;
Dog b;
Cat c;

F(&a); //???
F(&b); //???
F(&c); //???

}
```

Раннее связывание. Исправление

```
class Animal
{
AnimalType type;
};
void F(const Animal *p)
{
if(type == CAT)
static_cast<Cat*>(p)->Voice();
else if(type == DOG)
static_cast<Dog*>(p)->Voice();
else p->Voice();
}
```

Механизм вызова виртуальных функций

- Если в объявлении класса появляется виртульная функция, то компилятор формирует таблицу виртуальных функций.
- Каждый объект в дополнении к стандартным полям хранения данных содержит поля для хранения указателя на таблицу виртуальныйх функций — vfptr;

Механизм позднего связывания

```
class Animal
{
public:
virtual void Voice() const { cout << "???"; }
};

class Dog : public Animal
{
virtual void Voice() const { cout << "Gav"; }
};

class Cat : public Animal
{
virtual void Voice() const { cout << "Miau"; }
};
```

Позднее связывание

```
void F(const Animal *p)
{ p->Voice(); }

int main()
{
Animal a;
Dog b;
Cat c;

F(&a); //???
F(&b); //???
F(&c); //???

}
```

Виртуальные деструкторы

```
class Animal
{ public:
/* virtual */ ~Animal();
};
class Dog: public Animal
{ public:
~Dog();
};
int main()
{
Animal *zoo[] = { new Animal(), new Dog() };
for(int i = 0; i < sizeof(zoo)/sizeof(Animal*); i
    ++)
{ zoo[i]->Voice();
delete zoo[i]; }
}
```

Спецификатор разрешения области видимости

```
class A
{
public:
void F(int);
};
class B : public A
{
public:
void F(int, int);
void F(int);
};
int main()
{
B b;
b.F(1, 2); //???
b.F(3); //???
b.A::F(3); //???
}
```

Спецификатор разрешения области видимости

Спецификатор разрешения области видимости отменяет полиморфизм

```
class A
{ public:
virtual void VF();
};
class B : public A
{ public:
virtual void VF();
};
int main()
{ A *p = new B;
p->VF(); //???
p->A::VF(); //???
}
```

Использование виртуальных и не виртуальных методов

```
class A
{ public: void General() { VF(); }
private: virtual void VF() { /* ... */ }
};
class B: public A
{ virtual void VF() { /* ... */ } };
int main()
{
A* pA = new A;
A* pB = new B;
pA->General();
pB->General();
pA->A::General();
pB->A::General();
pA->A::VF();
pB->B::VF();
}
```

Чисто виртуальные функции и абстрактные классы

```
class A
{
public:
virtual void VF() = 0;
};
```

Делает функцию чисто виртуальной. Чисто виртуальный метод может, но не обязан иметь тело.

Абстрактный класс

- класс содержащий хотя бы одну чисто виртуальную (pure virtual) функцию, называется абстрактным;
- компилятор НЕ ПОЗВОЛЯЕТ создавать объеты такого класса;
- абстрактный класс используется только в качестве базового класса при наследовании;
- компилятор следит, чтобы в производных классах такой метод был реализован;
- можно использовать указатель на абстрактный класс.

Абстрактный класс

```
class Animal
{ public:
void Voice() const = 0;
};
class Dog : public Animal
{ void Voice() const { cout << "Gav"; } };
class Cat : public Animal
{ void Voice() const { cout << "Miau"; } };
int main()
{
Animal animal; //???
Animal *zoo[] = { new Dog, new Cat, new Animal };
    //???
Animal *zoo2[] = { new Dog, new Cat };
//FIXME
}
```