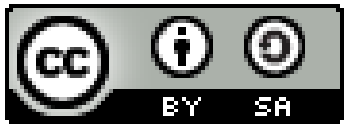
Отношения между объектами

и реализация отношений средствами C++

Это произведение доступно по лицензии
Creative Commons “Attribution-ShareAlike” (“Атрибуция — На тех же условиях”) 3.0 Непортированная.
http://creativecommons.org/licenses/by-sa/3.0/deed.ru



December 16, 2017

Отношения в терминах ООА

Наиболее часто используемые отношения между объектами (1)

Содержит:

- автомобиль содержит двигатель;
- студент содержит фамилию, номер зачетки;
- студент содержит желудок, сердце, скелет . . . ;
- электрическая схема содержит радиодетали.

Синоним (только с другой точки зрения) – “является частью”.

Использует:

- водитель управляет машиной;
- деканат принимает студента;
- графическая подсистема рисует эллипс.

Отношения в терминах OOA

Наиболее часто используемые отношения между объектами (2)

Является разновидностью:

- стриж является разновидностью птицы;
- велосипед является разновидностью транспорта;
- диод это полупроводниковый элемент;
- квадрат есть геометрическая фигура.

Реализован посредством:

- квадрат реализован с помощью прямоугольника;
- танк сделан из трактора;
- стек реализован посредством списка.

Отношение “Содержит”

Реализация отношения “Содержит” в C++ (1)

Отношение **“Содержит”** реализуется посредством **вложения**.
Пример: “студент содержит фамилию, номер зачетки”.

```
class Student {
  public:
    // ...
    const char* getName() const { return name; };
  private:
    char *name;
    int nz;
};
```

Как правило, *вложенные объекты* находятся в закрытой (**private**) части класса. Для доступа к таким объектам используется интерфейс того класса, который их содержит.

Отношение “Содержит”

Реализация отношения “Содержит” в C++(2)

Пример: “автомобиль содержит двигатель”.

```
class Car {
  public:
    Car( const char *mark);
    ~Car();
    // ...
    bool on() { return motor.start(); };
  private:
    Motor motor;
    // ...
};

int main()
{
  Car my_car( "LX-300" );
  my_car.on();
}
```

Отношение “Содержит”

Реализация отношения “Содержит” в C++(3)

Создание (инициализация) вложенных объектов происходит **до** создания содержащего объекта. Для передачи параметров конструктору вложенного объекта используется *список инициализации*. Порядок создания определяется порядком описания вложенных объектов в классе.

```
Car::Car( const char *mark )
    : motor("XV-5"),    //  это вызов
                        //  конструктора для motor
      title(mark)
{ //  <— здесь все вложенные объекты
  //  ... уже созданы
}
```

Отношение “Содержит”

Реализация отношения “Содержит” в C++(4)

Удаление вложенных объектов происходит **после** удаления содержащего объекта. Деструктор вложенных объектов вызывается автоматически. Порядок уничтожения противоположен порядку создания.

```
Car::~Car()
{
  //  ....
} //  <—– здесь будут автоматически вызваны
  //  деструкторы вложенных объектов.
```

Отношение “Использует”

Реализация отношения “Использует” в C++

Отношение **“Использует”** реализуется посредством **вызова функций** из интерфейса класса.

```
class Student {
  public:
    Student( const char *nm, int anz );
    ~Student();
    int exam();
    const char* getName() const { return name; };
  // ....
};

void f()
{
  Student a( "Vasya", 9103 ); //  создать
  a.exam();                   //  используем exam
  cout << a.getName() << endl; //  используем getName
} // <---   здесь вызывается (используется) деструктор
```

Отношение “Есть разновидность”
Реализация отношения “Есть разновидность” в C++ (1)

Отношение **“Есть разновидность”** реализуется посредством **открытого наследования**.

```
class A {
  // ...
};

class B : public A { //  обозначает: класс B
                     //  открыто наследует A.
  // ...
};
```

Ключевым моментом **открытого** наследования является следующее утверждение:
всё, что верно для класса A, верно и для класса B.
Класс B может дополнять класс A, но не может убирать что-либо из интерфейса или реализации A.
A – **базовый** класс для B. B – **производный** класс от A.

Отношение “Есть разновидность”

Пример: “курица есть разновидность птицы”.

```
class Bird {
  public:
    Bird( const char *nm );
    ~Bird();
    void fly();   //  птицы летают
    void eat();   //  и едят
  // ...
};
class Hen : public Bird {
  public:
    Hen( const char *nm, int e2d );
    ~Hen();
    void fly();      //  курица летает по-своему
    //  ест как обычные птицы – не определяем свою
    //  функцию eat – используем из базового
    int makeEgg(); //  + несет яйца
  // ...
};
```

Отношение “Есть разновидность”
Конструкторы производных классов

Базовая часть производного класса создается **до** производного. Параметры конструктору передаются в списке инициализации.

```
Hen::Hen( const char *nm, int e2d )
    :egg2day( a2d ),
     Bird( nm )
{
  // ....
}

Hen r( "Ryaba", 2 );
```

Сначала инициализируются базовые объекты (в порядке описания), потом – вложенные объекты, и только потом работает конструктор наследника (производного класса).

Отношение “Есть разновидность”
Деструкторы производных классов

Базовая часть производного класса уничтожается **после** уничтожения производного. Деструктор базового класса вызывается **автоматически** после деструктора производного.

```
Hen::~Hen()
{
  //  ....
} //  <- здесь вызываются деструкторы для
  //  Bird, egg2day

int f()
{
  Hen r( "Ryaba", 2 );
  r.fly();
} //  <- здесь вызывается деструктор Hen
```

Сначала уничтожатся производная часть объекта, потом – вложенные объекты, последняя – базовая часть. Порядок уничтожения противоположен порядку создания.

Отношение “Есть разновидность”
Доступ к членам базового класса (1)

Наследники не имеют доступа к закрытой (private) части базового класса. Что бы разрешить доступ наследникам, но запретить всем остальным, используется модификатор **protected** (защищённая часть класса).

```
class A {

  public:    //  <— открытая часть
    void f();    //  могут использовать все

  protected:   //  <— защищённая часть
    double p;    //  могут использовать ф-ции
                 //  A и наследники

  private:   //  <— закрытая часть
    char x;      //  могут использовать только
                 //  функции, описанные в A
};
```

Отношение “Есть разновидность”
Доступ к членам базового класса (2)

```
class Bird {
  public:
    void fly();   //  могут использовать все
  protected:
    double weight; //  могут использовать
                   //  Bird и наследники
  private:
    char *name;   //  могут использовать
                  //  функции, описанные в Bird
  // ...
};
class Hen : public Bird { /* ... */ };
void Hen::fly()
{
  Bird::fly();      //  функция из базового класса (public)
  weight -= 0.01;   //  можно – protected
  name = "xx";      //  НЕЛЬЗЯ – private
}
```

Отношение “Есть разновидность”
Преобразования

Так как открытое наследование обозначает “есть разновидность”, то:
1. Объекты открытого производного типа можно использовать там, где требуются объекты базового (срезка).

```
void g( Bird x ) { /* ... */ };
void f()
{
  Hen r( "Galina_Blanka", 5 );
  g( r );
}
```

2. Указатели и ссылки на объекты открытого производного типа можно преобразовывать в указатели на объекты базового.

```
Bird *p = &r; // можно
p->fly();
Hen *t = p;   // а так нельзя!
```

Отношение “Есть разновидность”

Необходимость полиморфного поведения

При использовании обычных функций-членов и преобразования указателей и ссылок из производного класса в базовый объекты теряют свою классовую индивидуальность: всё работает как базовый класс.

```
Bird *m[3]; // массив из 3х указателей на Bird
m[0] = new Hen( "Fat_hen", 10 );
m[1] = new Martlet( "Fast" ); // стриж
m[2] = new Falcon( "Mad_eye" ); // сокол

for( int i=0; i<3; ++i )
  m[i]->fly(); // <— все летают как просто птицы
for( int i=0; i<3; ++i )
  delete m[i]; // <— все умирают как просто птицы
```

Это не то поведение, какое требуется от иерархии классов птиц. Необходимо **полиморфное** поведение – что бы несмотря на формальный тип указателя, вызывалась функция-член для настоящего типа.

Отношение “Есть разновидность”

Виртуальные функции

В C++ для реализации полиморфного поведения объектов используются **виртуальные функции**. Функция-член становиться виртуальной при её описании в классе с ключевым словом **virtual**.

```
class Bird {
  public:
  virtual ~Bird();
  virtual void fly();
  void eat();
};
class Hen : public Bird{
  public:
  virtual ~Hen();
  void fly();  //  “virtual” повторять не обязательно
};
Bird *p = new Hen( "Polya", 1 );
p->fly(); //  fly для класса Hen
delete p; //  деструктор для Hen
```

Отношение “Есть разновидность”

Реализация виртуальных функций (1)

Для реализации возможности полиморфного поведения объект должен содержать информацию о том, какие функции следует вызывать при вызове через указатель. Это достигается следующим образом:

1. Для каждого класса, с котором есть виртуальные функции, создаётся компилятором таблица указателей на эти функции.
2. В каждом объекте класса с виртуальными функциями существует невидимый для пользователя указатель (vptr). Он указывает на таблицу виртуальных функций для данного класса. Заполняется при создании объекта.
3. При вызове виртуальной функции с использованием указателя или ссылки на объект, компилятор вызывает функцию, адрес которой находится в таблице виртуальных функций.

Отношение “Есть разновидность”
Реализация виртуальных функций (2)

Эта технология называется “позднее связывание”, то есть конкретная функция выбирается во время выполнения программы. Невиртуальные функции, а также функции, для которых тип объекта точно известен, выбираются на этапе компиляции (раннее связывание).

```
Hen a( "SuperHen", 100 );
a.fly(); // раннее (статическое) связывание. a – точно Hen

Bird *p = &a;
p->fly(); // позднее (динамическое) связывание:
          // p может указывать на любого
          // открытого наследника Bird
```

Отношение “Есть разновидность”

Правила для виртуальных функций

При использовании виртуальных функций есть несколько правил:

- Так как адрес виртуальной функции содержится в объекте, а на объект указывает указатель **this**, то виртуальными могут быть только функции-члены и деструктор.
- Если в классе есть хотя бы одна виртуальная функция, то объекты этого класса планируется использовать полиморфно. Следовательно, в таком классе должен быть виртуальный деструктор.
- Аргументы у виртуальных функций в базовом классе и наследнике должны совпадать. В противном случае произойдет **сокрытие**, а не замещение. (См. “using”)

Невиртуальные функции-члены являются инвариантом относительно наследования. Переопределять их в производных классах не рекомендуется.

Отношение “Есть разновидность”

Наследование интерфейса и реализации

Если функция не определена в производном классе, то происходит её наследование из базового. При этом наследуются:

- интерфейс функции, то есть то, что такая функция есть, и у неё есть определённые аргументы (`fly()` – все птицы летают);
- реализация функции, то есть то действие, которое выполняет функция (`fly() { /* действия */ }` – если не сказано иначе, птицы летают так).

Часто бывают случаи, когда надо наследовать интерфейс, но нельзя наследовать реализацию. Например, если мы напишем класс Ostrich (страус), и забудем определить функцию fly, то страус будет летать как большинство птиц.

Отношение “Есть разновидность”

Чистые виртуальные функции и абстрактные классы

Что бы наследовать интерфейс, но не наследовать реализацию, используются **чистые виртуальные функции**. При описании такой функции после параметров добавляется “=0”;

```
class Bird {
  public:
  virtual void fly() = 0;   //  чистая виртуальная
  //  ... функция
};
```

Класс в котором есть хотя бы одна чистая виртуальная функция, является **абстрактным классом**. Объекты абстрактных классов создавать нельзя – эти классы описывают абстрактное понятие (например: птицы). Если наследник переопределит чистую виртуальную функцию, то он становится **конкретным классом**, и объекты этого типа создавать можно.

Отношение “Есть разновидность”

Чистые виртуальные функции и абстрактные классы (2)

```
class Bird {  // абстрактный класс
  public:
    virtual void fly() = 0;  // чистая виртуальная
};
void Bird::fly() { /* может быть тело */ }

class Sparrow : public Bird { // конкретный класс
  public:
    virtual void fly() { Bird::fly(); };
};

class Hen : public Bird { // конкретный класс
  public:
    virtual void fly() { slow_fly(); };
};

class Ostrich : public Bird { // абстрактный
  public:
};
```

Отношение “Реализован посредством”

Реализация отношения “Реализован посредством” в C++

Отношение **“Реализован посредством”** реализуется посредством **закрытого наследования** или **вложения**.

```
class Rectangle {
  public:
    Rectangle( int aw, int ah );
    int width() const { return w; };
    int height() const { return h; };
  protected:
    int w, h;
};

class Square : private Rectangle {
  public:
    Square( int w ) : Rectangle( w, w );
    int width() const
      { return Rectangle::width() };
}
```

Сокрытие и перегрузка виртуальных функций

Если в классе-наследнике объявлена функция, с тем же именем, что и в базовом, но другими аргументами, то происходит не перегрузка, а **замещение** функции.

```
class T;
class A {
  public:
    virtual ~A();
    virtual void f(const T &x);
};
class B : public A
{
  public:
    virtual void f(int x); //  это совсем другая f
};
B b; T t;
b.f( t ); //  ошибка, f(const T &x) не видна
```

```
warning: 'virtual void A::f(const T&)' was hidden
```

Сокрытие и перегрузка виртуальных функций

Для разрешения перегрузки можно явно ввести в область видимости имя из базового класса с помощью объявления **using**.

```
class T;
class A {
  public:
    virtual ~A();
    virtual void f(const T &x);
};
class B : public A
{
  public:
    using A::f;
    virtual void f(int x); //  это перегруженная f
};
B b; T t;
b.f( t ); //  вызывается f из A
```

Сокрытие и перегрузка виртуальных функций

В C++11 с помощью ключевого слова “ **override** ” можно потребовать, что бы данная виртуальная функция не скрывала функции из базовых классов. Словом “ **final**” помечаются функции, которые нельзя переопределять.

```
class T;
class A {
  public:
    virtual ~A();
    virtual void f(const T &x);
    virtual void g(void) const;
    virtual void fi(void) const;
};
class B : public A
{
  public:
    virtual void f(int x) override; //  будет ошибка
    virtual void g(void) const override; // Ok
    virtual void fi(void) const final override;
};
```

Множественное наследование

Общий синтаксис

Класс может иметь нескольких предков.

```
class A {
};

class B {
};

class C {
};

class D: public A, public B, private C {
};
```

В этом примере класс D есть A, есть B, реализован посредством C.

Множественное наследование

Пример 1 - базовый класс

```
class WarUnit { //  базовый класс
  public:
    WarUnit( const char *name );
    virtual ~WarUnit();
    virtual int attack( WarUnit *target ) = 0;
  protected:
    double health;
    double x, y, z, alpha, phi, theta;
    //  ......
};

int WarUnit::attack( WarUnit *target ) {
{
  return aim_to_target( target );
}
```

Множественное наследование

Пример 1 - наследники

```
//  солдат есть боевая единица
class Soldier : public WarUnit {
  public:
    Soldier( const char *name, double hitrate );
    ~Soldier();
    virtual int attack( WarUnit *target ) override;
};

int Soldier::attack( WarUnit *target )
{
  WarUnit::attack( target );
  return use_AK47( target );
}

class Tank : public WarUnit { .... };

class SAM : public WarUnit { .... };
```

Множественное наследование

Пример 1 - гибрид танка и зенитки. Версия 1.

```
class TankSAM : public Tank, public SAM {
  public:
    TankSAM( const char *name, double calibr1,
             double calibr2 );
    ~TankSAM();
};

TankSAM::TankSAM( const char *name, double calibr1,
                  double calibr2 )
   : Tank( name, calibr1 ), SAM( name, calibr2 )
{}

TankSAM a( "Super-Tank", 125, 40 );
a.attack( &enemy ); //  неопределённость
//  Tank::attack или SAM::attack?

a.Tank::attack( &enemy ); //  можно,
//  но потеряли полиморфное поведение
```

Множественное наследование

Пример 1 - гибрид танка и зенитки. Версия 2.

```
class TankSAM : public Tank, public SAM {
  public:
    TankSAM( const char *name, double calibr1,
             double calibr2 );
    virtual int attack( WarUnit *target ) override;
};

int TankSAM::attack( WarUnit *target )
{
  if( target->is_AirUnit() )
    return SAM::attack( target );
  return Tank::attack( target );
}

TankSAM a( "Super-Tank", 125, 40 );
a.attack( &enemy ); //  можно, сохранили полиморфное
//  поведение, но есть дублирование общих данных
```

Множественное наследование

Пример 1 - гибрид... Виртуальное наследование. Версия 3.

```
class Tank : virtual public WarUnit { .... };
class SAM  : virtual public WarUnit { .... };
class TankSAM : public Tank, public SAM {
  public:
    TankSAM( const char *name, double calibr1,
             double calibr2 );
    virtual int attack( WarUnit *target ) override;
};
TankSAM::TankSAM( const char *name, double calibr1,
                  double calibr2 )
   : WarUnit(name),
   Tank( name, calibr1 ), SAM( name, calibr2 )
{}

TankSAM a( "Super-Tank", 125, 40 );
a.attack( &enemy ); //  можно, сохранили полиморфное
//  поведение, нет дублирования общих данных +
//  новые проблемы
```