Программирование на языке C++

Вводный курс

Александр Морозов
gelu.speculum@gmail.com

ИТМО, весенний семестр 2020

Содержание

Инкапсуляция

- сокрытие логически связанных подробностей в одном месте
- объединение данных и логики их обработки

Помогает разделять программу на слабо связанные компоненты.

Варианты:

- ООП
- пространства имён и модули (Erlang, Haskell)
- функции с локальным контекстом (Javascript, Scheme, Pascal)

Зачем нужно разделение?

- легче читать
- легче менять
- легче тестировать

Наследование

- возможность переиспользовать код
- возможность расширять функциональность
- возможность компоновать код

Статический и динамический полиморфизм

Статический:

- использование одного кода для разных типов
- на этапе компиляции полиморфизм раскрывается в обычный код
- на этапе исполнения не остаётся следов

Динамический:

- возможность работать с объектами разных типов через один интерфейс
- поведение выбирается на этапе исполнения
- рефлексия
- гетерогенные коллекции

Динамический полиморфизм в C++

- наследование классов
- ссылки или указатели
- приведение типов
- виртуальные методы

Наследование без полиморфизма

```
 1    class A
 2    {
 3        int m_a = 11;
 4    public:
 5        int compute() const
 6        { return m_a; }
 7    };
 8
 9    class B : public A
10    {
11        int m_b = 22;
12    public:
13        int compute() const
14        { return m_a + m_b; }
15    };
16
17    void process(const A & a)
18    { std::cout << a.compute() << std::endl; }
19
20    int main()
21    {
22        B b;
23        process(b);
24    }
```

Наследование с полиморфизмом

```
 1    class A
 2    {
 3        int m_a = 11;
 4    public:
 5        virtual int compute() const
 6        { return m_a; }
 7    };
 8
 9    class B : public A
10    {
11        int m_b = 22;
12    public:
13        int compute() const
14        { return m_a + m_b; }
15    };
16
17    void process(const A & a)
18    { std::cout << a.compute() << std::endl; }
19
20    int main()
21    {
22        B b;
23        process(b);
24    }
```

Содержание

Виртуальные методы

`virtual` *объявление метода*

```
1    class A
2    {
3        virtual void f();
4    };
5
6    void A::f() { ... }
```

Ограничения применения виртуальности

Не могут быть виртуальными:

- свободные функции
- статические методы
- шаблонные методы

Переопределение виртуальных методов

Если класс ниже по иерархии наследования определяет метод

- с тем же именем
- с тем же списком параметров
- с теми же cv-квалификаторами
- с теми же ссылочными квалификаторами

то этот метод тоже становится виртуальным и переопределяет исходный.

Особенности переопределения и вызова виртуальных методов

```
 1  class A
 2  {
 3  public:
 4      virtual void f();
 5      virtual int g();
 6  };
 7
 8  class B
 9  {
10  private:
11      void f();
12      int g() const; // warning
13  };
14
15  class C
16  {
17  public:
18      double g(); // error
19  };
20
21  void call(A & a)
22  {
23      a.f();
24      a.A::f();
25  }
26
27  int main()
28  {
29      B b;
30      call(b);
31  }
```

Скрытие и переопределение методов при наследовании

```
Base →··· →Derived

T Base::f(int);
```

	не виртуальный	виртуальный
T Derived::f(int)	скрывает	переопределяет
P Derived::f(int)	скрывает	переопределяет, если P ковариантен T
T Derived::f()	скрывает	скрывает
T Derived::f() const	скрывает	скрывает

Ковариантный тип

Y ковариантен X, если

- оба являются ссылкой или указателем на класс
- Y не более cv-квалифицирован, нежели X
- класс, на который сслыается/указывает X, должен быть однозначным и доступным предком класса, на который ссылается/указывает Y

Пример с ковариантным типом

```
 1    struct Base
 2    {
 3        const Base * clone()
 4        { return new Base(*this); }
 5    };
 6
 7    struct Derived : Base
 8    {
 9        Derived * clone()
10        { return new Derived(*this); }
11    };
12
13    int main()
14    {
15        Derived d;
16        auto c = d.clone(); // Derived *
17        Base & b = d;
18        auto cc = b.clone(); // const Base *
19    }
```

Пример без ковариантного типа

```
 1    struct Base
 2    {
 3        const Base * clone()
 4        { return new Base(*this); }
 5    };
 6
 7    struct Derived : Base
 8    {
 9        const Base * clone()
10        { return new Derived(*this); }
11    };
12
13    int main()
14    {
15        Derived d;
16        auto c = d.clone(); // const Base *
17        Base & b = d;
18        auto cc = b.clone(); // const Base *
19    }
```

override И final

Спецификаторы `override` и `final` можно использовать в конце объявления метода.

`override` – метод переопределяет унаследованный виртуальный метод.

`final` – метод переопределяет унаследованный виртуальный метод и не может быть переопределён в наследниках.

Пример использования override

```
 1    struct A
 2    {
 3        virtual void f(const X &);
 4        void g(int);
 5    };
 6
 7    struct B : A
 8    {
 9        void f(const X);
10    };
11
12    struct C : A
13    {
14        void f(const X &) const override;
15        void g(int) override;
16    };
17
18    void call(A & a, const X & x)
19    {
20        a.f(x);
21    }
```

Пример использования `final`

```
 1    struct A
 2    {
 3        virtual void f(const X &);
 4    };
 5
 6    struct B : A
 7    {
 8        void f(const X &) final;
 9    };
10
11    struct C : B
12    {
13        void f(const X &); // error
14    };
```

final в наследовании

```
 1    struct A
 2    {
 3    };
 4
 5    struct B final : A
 6    {
 7    };
 8
 9    struct C : B // error
10    {
11    };
```

Полиморфизм и деструкторы

```
 1    class Base
 2    {
 3    public:
 4        virtual int compute()
 5        { return 0; }
 6    };
 7
 8    class Derived : public Base
 9    {
10        std::vector<int> m_data;
11
12        int compute()
13        { return std::accumulate(m_data.begin(), m_data.end(), 0); }
14    public:
15        Derived(const std::size_t size)
16            : m_data(size)
17        { std::iota(m_data.begin(), m_data.end(), 1); }
18    };
19
20    int main()
21    {
22        Base * b = new Derived(100);
23        int res = b->compute();
24        delete b;
25        return res;
26    }
```

Корректное полиморфное удаление

```
 1    class Base
 2    {
 3    public:
 4        virtual ~Base() = default;
 5    };
 6
 7    class Derived : public Base
 8    { ... };
 9
10    int main()
11    {
12        Base * b = new Derived;
13        delete b; // ~Derived() is called
14    }
```

Чисто виртуальный метод и абстрактные классы

```
 1    struct A
 2    {
 3        virtual void f() = 0;
 4    };
 5
 6    struct B : A
 7    {
 8        void f();
 9    };
10
11    int main()
12    {
13        A * a = new B; // OK
14        A * aa = new A; // error
15    }
```

Больше чистой виртуальности

```
1   struct A
2   {
3       virtual void f() = 0;
4   }
5
6   void A::f() { ... }
7
8   struct B
9   {
10      void f()
11      { A::f(); }
12  };
13
14  struct C
15  {
16      virtual ~C() = 0;
17  };
18
19  C::~C() { ... } // mandatory
20
21  struct D : C
22  {
23      ~D(); // or else D is abstract
24  };
25
26  struct E
27  {
28      virtual void g() { ... }
29  };
30
31  struct F : E
32  {
33      void g() = 0;
34  };
```

Параметры по умолчанию в виртуальных методах

```
 1    int m = 111;
 2
 3    struct A
 4    {
 5        virtual int f(int k = m) { return k; }
 6    };
 7
 8    struct B : A
 9    {
10        int f(int k = -13) { return k; }
11    };
12
13    int get_n(A & a)
14    {
15        return a.f();
16    }
17
18    int main()
19    {
20        B b;
21        return get_n(b);
22    }
```

Рекомендации по использованию виртуальности

- интерфейс (в базовом классе) делать не виртуальным
- переопределяемую реализацию оформлять приватными виртуальными методами базового класса
- методы интерфейса вызывают нужные виртуальные методы реализации
- если предполагается полиморфное создание/удаление - сделать виртуальный деструктор в базовом классе
- иначе рассмотреть возможность защищённого не виртуального деструктора

Содержание

Приведение типов по иерархии наследования

Можно приводить указатели и ссылки на классы вверх и вниз по иерархии наследования.

- Неявное приведение вверх: **Derived *** $\rightarrow$**Base ***
- Явное приведение вниз: **Base *** $\rightarrow$**Derived ***
- Типо-безопасное приведение по иерархии наследования

При таком приведении сам объект **не меняется** – меняется лишь указатель или ссылка для доступа к нему.

Пример иерархии наследования

```
1   struct LeftRoot
2   {
3       virtual void print() const
4       { std::cout << "LeftRoot" << std::endl; }
5   };
6
7   struct LeftMiddle : LeftRoot
8   {
9       void print() const
10      { std::cout << "LeftMiddle" << std::endl; }
11  };
12
13  struct LeftLeaf : LeftMiddle
14  {
15      void print() const
16      { std::cout << "LeftLeaf" << std::endl; }
17  };
18
19  struct RightRoot
20  {
21      virtual void another_print() const
22      { std::cout << "RightRoot" << std::endl; }
23  };
24
25  struct Middle : LeftMiddle, RightRoot
26  {
27      void print() const
28      { std::cout << "Middle" << std::endl; }
29      void another_print() const
30      { std::cout << "Middle" << std::endl; }
31  };
32
33  struct RightLeaf : Middle
34  {
35      void print() const
36      { std::cout << "RightLeaf" << std::endl; }
37      void another_print() const
38      { std::cout << "RightLeaf" << std::endl; }
39  };
```

Разные варианты приведения по иерархии

	Неявное	static_cast	dynamic_cast
LeftLeaf →LeftRoot	✓	✓	✓
RightLeaf →LeftRoot	✓	✓	✓
RightLeaf →RightRoot	✓	✓	✓
LeftRoot →LeftLeaf	✗	✓(!)	✓
Middle →LeftRoot →LeftLeaf	✗	✓	✗
RightLeft →LeftRoot →RightRoot	✗	✗	✓
LeftRoot →...	✗	✗	✓

Варианты использования `dynamic_cast`

```
X x = dynamic_cast<X>(y);
```

- `X` ≡ `decltype(y)`
- `X` ≡ `const decltype(y)`
- `y == nullptr` →`x == nullptr`
- `X` – однозначный, доступный предок `decltype(y)`
- `X = void *`, `decltype(y)` – полиморфный
- `decltype(y)` – полиморфный, `X` – ссылка или указатель на класс
  - `X` – указывает/ссылается на однозначного, публично унаследованного потомка `y`
  - `y` – публичный предок объекта класса `Z`, а `X` – другой однозначный, доступный предок `Z`
  - иначе,
    - `X` – указатель, `x == nullptr`
    - `X` – ссылка, `std::bad_cast` исключение

RTTI

Run-time type information

- `dynamic_cast`
- `typeid`

Множественное наследование

```
 1  struct A
 2  {
 3      int a;
 4
 5      A(int a_ = 0) : a(a_) {}
 6  };
 7
 8  struct B : A
 9  {
10      B(int a_) : A(a_) {}
11
12      int b_get_a() const
13      { return a; }
14  };
15
16  struct C : A
17  {
18      C(int a_) : A(a_) {}
19
20      int c_get_a() const
21      { return a; }
22  };
23
24  struct D : B, C
25  {
26      D(int b, int c) : B(b), C(c) {}
27  };
28
29  int main()
30  {
31      D d(11, 22);
32      std::cout << d.a << std::endl; // error
33      std::cout << d.b_get_a() << std::endl; // OK
34      std::cout << d.c_get_a() << std::endl; // OK
35  }
```

Множественное наследование и абстрактные классы

```
 1    struct Interface
 2    {
 3        virtual void f() = 0;
 4        virtual void g() = 0;
 5    };
 6
 7    struct A : Interface
 8    {
 9        void f();
10    };
11
12    struct B : Interface
13    {
14        void g();
15    };
16
17    struct C : A, B
18    {};
19
20    int main()
21    {
22        C c; // error: C is abstract
23    }
```

Виртуальное наследование

При указании наследования используется ключевое слово `virtual`

- в каждом конкретном типе-наследнике виртуальный предок встречается как подобъект ровно один раз
- конкретный тип-наследник создаёт подобъекты виртуальных предков (даже если в иерархии наследования между ними другие классы), причём до создания подобъектов обычных предков
- наследник должен иметь доступ к конструктору виртуального предка
- в списке инициализации конструктора наследника можно явно вызвать конструктор виртуального предка
- операция копирования конкретного наследника должна учитывать эти особенности
- для приведения типов по иерархии наследования с виртуальным наследованием необходимо использовать `dynamic_cast`

Решение проблемы неоднозначности

```
 1    struct A
 2    {
 3        int a;
 4
 5        A(int a_) : a(a_) {}
 6    };
 7
 8    struct B : virtual A
 9    {
10        B(int a_) : A(a_) {}
11
12        int b_get_a() const
13        { return a; }
14    };
15
16    struct C : virtual A
17    {
18        C(int a_) : A(a_) {}
19
20        int c_get_a() const
21        { return a; }
22    };
23
24    struct D : B, C
25    {
26        D(int b, int c) : A(b+c), B(b), C(c) {}
27    };
28
29    int main()
30    {
31        D d(11, 22);
32        std::cout << d.a << std::endl; // OK
33        std::cout << d.b_get_a() << std::endl; // OK
34        std::cout << d.c_get_a() << std::endl; // OK
35    }
```

Решение проблемы частичной реализации интерфейса и объединения этих реализаций

```
 1    struct Interface
 2    {
 3        virtual void f() = 0;
 4        virtual void g() = 0;
 5    };
 6
 7    struct A : virtual Interface
 8    {
 9        void f()
10        { g(); }
11    };
12
13    struct B : virtual Interface
14    {
15        void g();
16    };
17
18    struct C : A, B
19    {};
20
21    int main()
22    {
23        C c; // OK
24        c.f(); // call B::g()
25    }
```