13.04.2024

Наследование
Абстрактные классы

Спикер: Ибрагимов Булат Ленарович

Fast Track в Телеком, 2024

НТР новые телеком решения

PHOENIX EDUCATION

МФТИ

ИБРАГИМОВ БУЛАТ ЛЕНАРОВИЧ

- Преподаватель в МФТИ. Проводит курсы по C++ и алгоритмам, структурам данных
- Научный сотрудник Института Искусственного Интеллекта (AIRI)
- Работал разработчиком-исследователем в Яндекс и Сбербанк

Наследование II

В предыдущей серии

Знаем, что указатели (ссылки) на базовый класс (`A`) могут ссылаться на объекты производного класса (`B`).

```
struct A {
  void f() { std::cout << "A::f()\n"; }
};

struct B : public A {
  void f() {std::cout << "B::f()\n"; }
};
```

```
B b;
A* b_ptr = &b;
b_ptr->f();  // A::f()
```

Но даже при совпадении сигнатур функций всегда вызывается метод базового класса...

Виртуальные функции

Виртуальные функции

Добавим `virtual` перед объявлением метода в классе `A` и, voila:

```
struct A {
  virtual void f() { std::cout << "A::f()\n"; }
};

struct B : public A {
  void f() {std::cout << "B::f()\n"; }  // <-- автоматически virtual!
};
```

```
B b;
A* b_ptr = &b;
b_ptr->f();  // B::f()
```

`virtual`: позднее связывание

`virtual` перед объявлением метода говорит, что решение о том, какую версию метода выбрать, должно быть принято *во время исполнения программы* (**позднее связывание**), а не на *этапе компиляции* (**раннее связывание**).

```
struct A {
    virtual void f() { std::cout << "A::f()\n"; }
    void g() { std::cout << "A::g()\n"; }
};

struct B : public A {
    void f() { std::cout << "B::f()\n"; }  // <-- автоматически virtual!
    void g() { std::cout << "B::g()\n"; }
};
```

```
A* ptr = new B;
ptr->g();  // во время компиляции "подставляется" вызов A::g
ptr->f();  // решение откладывается до момента исполнения программы
```

`virtual`: позднее связывание

```
A* ptr = new B;
ptr->g();  // во время компиляции "подставляется" вызов A::g
ptr->f();  // решение откладывается до момента исполнения программы
```

Зачем откладывать решение?

Неужели компилятор не может разобраться, что именно нужно подставлять?

`virtual`: позднее связывание

Решение откладывать необходимо, так как реальный тип объекта может быть неизвестен:

```
int x;
std::cin >> x;

A* ptr = (x == 0 ? new B : new A);
ptr->g();  // во время компиляции "подставляется" вызов A::g
ptr->f();  // решение откладывается до момента исполнения программы
```

Как вы понимаете, позднее связывание более затратно, чем раннее связывание, за счет дополнительных действий во время исполнения программы (выяснение истинного типа объекта).

Загадка от Жака Фреско

```
struct A {
  virtual void f();
  void g();
};

struct B : public A {
  void f();
  virtual void g();
};

struct C : public B {
  void f();
  void g() const;
};
```

```
A a = B();
B b = C();
C c = C();

a.f();  // ???
a.g();  // ???

b.f();  // ???
b.g();  // ???

c.f();  // ???
c.g();  // ???
```

```
A* apb = new B;
apb->f();  // ???
apb->g();  // ???

A* apc = new C;
apc->f();  // ???
apc->g();  // ???

B* bpc = new C;
bpc->f();  // ???
bpc->g();  // ???
```

Загадка от Жака Фреско

```
struct A {
  virtual void f();
  void g();
};

struct B : public A {
  void f();
  virtual void g();
};

struct C : public B {
  void f();
  void g() const;
};
```

```
A a = B();
B b = C();
C c = C();

a.f();  // A::f
a.g();  // A::g

b.f();  // B::f
b.g();  // B::g

c.f();  // C::f
c.g();  // C::g
```

```
A* apb = new B;
apb->f();  // ???
apb->g();  // ???

A* apc = new C;
apc->f();  // ???
apc->g();  // ???

B* bpc = new C;
bpc->f();  // ???
bpc->g();  // ???
```

Загадка от Жака Фреско

```
struct A {
  virtual void f();
  void g();
};

struct B : public A {
  void f();
  virtual void g();
};

struct C : public B {
  void f();
  void g() const;
};
```

```
A a = B();
B b = C();
C c = C();

a.f();  // A::f
a.g();  // A::g

b.f();  // B::f
b.g();  // B::g

c.f();  // C::f
c.g();  // C::g
```

```
A* apb = new B;
apb->f();  // B::f
apb->g();  // A::g

A* apc = new C;
apc->f();  // C::f
apc->g();  // A::g

B* bpc = new C;
bpc->f();  // C::f
bpc->g();  // B::g
```

Вызов виртуальной функции из метода

"Виртуальность" работает и внутри методов:

```
struct A {
  void PrintName() const { std::cout << Name() << '\n'; }
  virtual const char* Name() const { return "A"; }
};

struct B : public A {
  const char* Name() const { return "B"; }
};
```

```
A* ptr = new B;
ptr->PrintName();  // B
```

Потому что `Name()` в `A::PrintName()` <=> `this->Name()`

Вызов виртуальной функции из метода

А в конструкторах и деструкторах "виртуальность" не работает!

```
struct A {
  virtual void f() { std::cout << "A "; }
  A()  { f(); }
  ~A() { f(); }
};

struct B : public A {
  void f() override { std::cout << "B "; }
  B()  { f(); }
  ~B() { f(); }
};

B b;
```

```
A B B A
```

Динамический полиморфизм

Напоминание:

- **Полиморфизм** - свойство системы, позволяющее использовать различные реализации в рамках одного интерфейса.
- **Статический полиморфизм** - вид полиморфизма, при котором выбор реализации осуществляется на этапе компиляции (перегрузка функций, шаблоны, перегрузка операций и т.д)
- **Динамический полиморфизм** - вид полиморфизма, при котором выбор реализации осуществляется во время выполнения программы.
- Основной механизм реализации динамического полиморфизма в C++ - **виртуальные функции**.
- Поэтому наследование с применением виртуальных функций называют **полиморфным**

Пример применения

```
struct Animal {
  virtual void Voice() const { ... }
};

struct Cat : Animal {
  void Voice() const { std::cout << "Meow\n"; }
};

struct Dog : Animal {
  void Voice() const { std::cout << "Woof\n"; }
};

struct Fox : Animal {
  void Voice() const { std::cout << "???\n"; }
};

std::array<Animal*, 10> animals;
// ... (fill animals)
for (int i = 0; i < 10; ++i) {
    animals[i]->Voice();
}
```

Виртуальный деструктор

В чем проблема деструктора?

```
class Stack { /* ... */ };

class StackMax : public Stack { /***/ };
```

```
Stack* stack_ptr = new StackMax;
// ...
delete stack_ptr;
```

Проблема

```
Stack* stack_ptr = new StackMax;
// ...
delete stack_ptr;  // <-- UB
```

- Тип `stack_ptr` - `Stack*`.
- Следовательно `delete` вызывает деструктор `Stack`, но не `StackMax`!
- То есть, `StackMax` уничтожен некорректно, а это пропуск в чудесный мир *Undefined Behaviour*
- На практике, скорее всего, вы столкнетесь с утечкой памяти (если производный класс выделял динамическую память или содержит поля с нетривиальными деструкторами)

Виртуальный деструктор

Проблема в том, что версия деструктора выбиралась на этапе компиляции. Чтобы отложить это решение до момента фактического исполнения кода, необходимо объявить деструктор виртуальным:

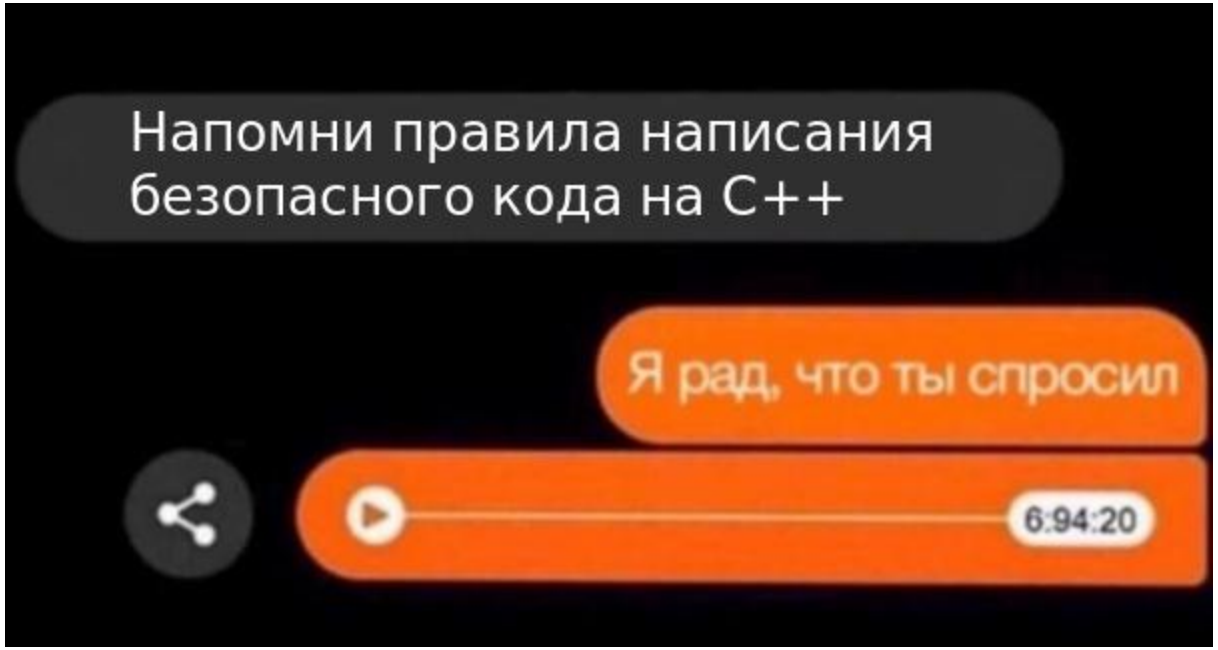
```
class Stack {
  // ...
  virtual ~Stack();
  // ...
};

class StackMax : public Stack { /* ... */ };
```

```
Stack* stack_ptr = new StackMax;
// ...
delete stack_ptr;  // Ok: деструктор выбирается во время исполнения
```

Мораль

Для полиморфных классов **необходимо** писать виртуальный деструктор



`override` и `final`

Проблема

Для переопределения виртуальной функции в производном классе необходимо (почти) в точности сохранить тип функции. В противном случае компилятор будет считать это определением нового метода (не связанного с предыдущим).

```
struct A {
  virtual void f(int);
  virtual void g() const;
};

struct B : public A {
  void f(long);
  void g();
};
```

```
A* ptr = new B;
ptr->f(0);  // A::f(int) : int != long
ptr->g();   // A::g()    : потерян const
```

`override`

Чтобы убедиться, что вы действительно переопределяете виртуальную функцию базового класса, а не создаете новую, можно написать `override` после типа функции.

```
struct A {
  virtual void f(int);
  virtual void g() const;
};

struct B : public A {
  void f(long) override;
  void g() override;
};
```

Теперь компилятор выдаст CE в случае несоответствия типов функций:

```
error: 'void B::f(long int)' marked 'override', but does not override
error: 'void B::g()' marked 'override', but does not override
```

Ковариантные возвращаемые типы

Тип виртуальной функции и ее переопределения должны быть равны с точностью до *ковариантных* возвращаемых типов.

Типы называются ковариантными, если они являются указателями или ссылками и ссылаются на родственные классы (предок-потомок).

Короче говоря, если виртуальная функция возвращает указатель/ссылку на класс, то при переопределении можно вернуть указатель/ссылку на производный класс.

```
struct A {
  // ...
  virtual A* Clone() const { return new A(*this); }
};

struct B : public A {
  // ...
  B* Clone() const override { return new B(*this); }  // Ok: B* и A* - ковариантные
};
```

`final`

Чтобы запретить дальнейшее переопределение виртуального метода можно пометить его словом `final`:

```
struct A {
  virtual void f();
};

struct B : public A {
  void f() final;
};

struct C : public B {
  void f() override;  // CE
};
```

```
error: virtual function 'void C::f() const' overriding final function
```

`final`

- Финальной можно пометить только виртуальную функцию:

```
struct A {
  void f();
};

struct B : public A {
  void f() final;  // CE: f не виртуальная!
};
```

- Можно объявить виртуальную функцию и сразу сделать ее финальной:

```
struct A {
  virtual void f() final;
};
```

Это имеет смысл, если хочется запретить наследникам определять функции с тем же именем и типом.

final

`final` можно использовать и при определении класса, чтобы запретить от него наследоваться:

```
struct A {
// ...
};

struct B final : public A {
// ...
};

struct C : public B {};  // CE
```

```
error: cannot derive from 'final' base 'B' in derived type 'C'
```

Чисто виртуальные функции и абстрактные классы

Проблема

Допустим, вы решили написать свой мессенджер. Необходимо поддержать кучу видов сообщений (текстовые, видео, стикеры, голосовые)

```
class Message {
  // ...
  void Send(Chat* chat_ptr) const;
  virtual void Display() const;  // должна быть переопределена для всех типов
};

class TextMessage : public Message {
  // ...
  void Display() const override;  // отображение текстового сообщения
};

class StickerMessage : public Message {
  // ...
  void Display() const override;  // отображение стикера
};

// ... другие классы
```

Проблема

```
class Message {
  // ...
  void Send(Chat* chat_ptr) const;
  virtual void Display() const;  // должна быть переопределена для всех типов
};
```

Философские вопросы:

1. Имеет ли смысл создание объекта `Message` (без типа)?
2. Что должен делать `Display` в `Message`?
3. Как "заставить" наследников реализовывать `Display`?

Учитываем, что:

- Нельзя оставить метод без реализации (ошибка линковки)
- Наследник может "забыть" реализовать `Display` (тогда будет вызван `Message::Display`)

Чисто виртуальные функции

Для решения проблемы необходимо указать, что `Message` не является полноценным типом, а представляет из себя лишь *интерфейс*, то есть набор методов характерных для каждого сообщения.

Для этого необходимо объявить методы интерфеса *чисто виртуальными*:

```
class IMessage {
  // ...
  virtual void Display() const = 0;
};
```

Чисто виртуальные функции

- Чисто виртуальные функции (за исключением чисто виртуальных деструкторов) можно оставлять без реализации

```
class IMessage {
  // ...
  virtual void Display() const = 0;  // Ok
  virtual ~IMessage() = 0;  // LE: необходима реализация
}
```

- Чисто виртуальные функции могут быть реализованы только вне класса:

```
class IMessage {
  // ...
};

IMessage::~IMessage() {
  MessageCleanup();
}
```

Абстрактный класс

Абстрактный класс - класс с *хотя бы одним* чисто виртуальным методом

```
class IMessage {
  // ...
  virtual void Display() const = 0;
};
```

- Нельзя создавать объекты абстрактного класса
- Но можно создавать ссылки и указатели на абстрактный класс

```
IMessage msg;  // CE
IMessage* ptr = new TextMessage;
IMessage& ref = *ptr;
```

```
error: cannot declare variable 'msg' to be of abstract type 'IMessage'
```

Абстрактный класс

- Если наследник не реализует чисто виртуальную функцию, то он сам становится абстрактным:

```
class IMessage {
  // ...
  virtual void Display() const = 0;
};

class TextMessage : public IMessage {
  // ...
  void Display() const override { /* ... */ }
};

class StickerMessage : public IMessage {
  // ... (Display не переопределяется => класс абстрактный)
};
```

```
TextMessage text;         // Ok
StickerMessage sticker;   // CE
```

Абстрактный класс

- Вызов чисто виртуального метода в конструкторе/деструкторе абстрактного класса приводит к undefined behaviour

```
class IMessage {
  // ...
  virtual void Display() const = 0;

  IMessage() {
    // ...
    Display();  // UB
  }
};
```

- Абстрактные классы используются для определения интерфейсов для создания семейства классов с одинаковыми свойствами и методами

Как это работает: таблица виртуальных функций

`static_cast`

Пусть `C` унаследован от `B`, а `B` унаследован от `A`.

Можно ли преобразовывать указатели в рамках действующей иерархии? - Да!

Например, с помощью `static_cast`:

```
A* ptr = new B;
auto c_ptr = static_cast<C*>(ptr);    // "вниз" по иерархии
auto b_ptr = static_cast<B*>(c_ptr);  // "вверх" по иерархии
```

В чем проблема?

`static_cast`

Пусть `C` унаследован от `B`, а `B` унаследован от `A`.

Можно ли преобразовывать указатели в рамках действующей иерархии? - Да!
Например, с помощью `static_cast`:

```
A* ptr = new B;
auto c_ptr = static_cast<C*>(ptr);    // "вниз" по иерархии
auto b_ptr = static_cast<B*>(c_ptr);  // "вверх" по иерархии
```

Но `static_cast` не принимает во внимание реальный тип объекта под указателем! Он и не может этого сделать, так как `static_cast` проверяет допустимость преобразований на этапе компиляции.

`dynamic_cast`

Позволяет преобразовывать указатели/ссылки на **полиморфный** класс в указатели/ссылки на предка или потомка.

В отличие от `static_cast` проверяет возможность преобразования во время исполнения, а не во время компиляции.

Синтаксис: `dynamic_cast<[тип]>([выражение]`, где

- `[выражение]` возвращает указатель или ссылку на полиморфный класс
- `[тип]` - указатель или ссылка на потомка или предка.

Если преобразование некорректно (реальный тип объекта не совпадает с запрошенным), то возвращает `nullptr` для указателей или бросает `std::bad_cast` для ссылок.

`dynamic_cast`: примеры

```
struct A {};

struct B : public A {
  virtual void f();
};

struct C : public B {};
struct D : public C {};
```

```
B* ptr = new C;
auto c_ptr = dynamic_cast<C*>(ptr);  // OK
auto a_ptr = dynamic_cast<A*>(ptr);  // OK

dynamic_cast<C*>(a_ptr);  // CE: (A - не полиморфный!)
dynamic_cast<D*>(ptr);    // OK: == nullptr
dynamic_cast<C&>(*ptr);   // OK: == *c_ptr
dynamic_cast<D&>(*ptr);   // RE: std::bad_cast
```

Резюме

- Виртуальные функции позволяют использовать механизм полиморфного наследования: реализация метода будет выбираться на основе объекта под указателем (ссылкой), а не на основе типа указателя (ссылки)
- Стоит помнить о необходимости виртуального деструктора при полиморфном наследовании
- Используйте `override` при переопределении виртуальных функций
- Чисто виртуальные функции делают класс абстрактным. Такие функции должны быть переопределены в наследниках (иначе наследник тоже будет абстрактным)