extra3 Проблема вызова виртуальных функций в конструкторах и деструкторах. Пример ошибки pure virtual function call. Проблема с аргументами по умолчанию в виртуальных функциях.

Вы не должны вызывать виртуальные функции во время работы конструкторов или деструкторов, потому что эти вызовы будут делать не то, что вы думаете, и результатами их работы вы будете недовольны.

Предположим, что имеется иерархия классов для моделирования биржевых транзакций, то есть поручений на покупку, на продажу и т. д. Важно, чтобы эти транзакции было легко проверить, поэтому каждый раз, когда создается новый объект транзакции, в протокол аудита должна вноситься соответствующая запись. Следующий подход к решению данной проблемы выглядит разумным:

```
1
2  class Transaction { // базовый класс для всех
3      public: // транзакций
4          Transaction();
5          virtual void logTransaction() const = 0; // выполняет зависящую от типа
6          // запись в протокол
7          ...
8  };
9
10 Transaction::Transaction() // реализация конструктора
11 { // базового класса
12     ...
13     logTransaction();
14 }
15
16 class BuyTransaction: public Transaction { // производный класс
17     public:
18         virtual void logTransaction() const = 0; // как протоколировать
19         // транзакции данного типа
20         ...
21 };
22
23 class SellTransaction: public Transaction { // производный класс
24 public:
25     virtual void logTransaction() const = 0; // как протоколировать
26     // транзакции данного типа
27     ...
28 };
```

Посмотрим, что произойдет при исполнении следующего кода: `BuyTransaction b;`

Ясно, что будет вызван конструктор BuyTransaction, но сначала должен быть вызван конструктор Transaction, потому что части объекта, принадлежащие базовому классу, конструируются прежде, чем части, принадлежащие производному классу. В последней строке конструктора Transaction вызывается виртуальная функция logTransaction, тут-то и начинаются сюрпризы. Здесь вызывается та версия logTransaction, которая определена в классе Transaction, а не в BuyTransaction, несмотря на то что тип создаваемого объекта – BuyTransaction. Во время конструирования базового класса не вызываются виртуальные функции, определенные в производном классе. Объект ведет себя так, как будто он принадлежит базовому типу. Короче говоря, во время конструирования базового класса виртуальных функций не существует.

Есть веская причина для столь, казалось бы, неожиданного поведения. Поскольку

конструкторы базовых классов вызываются раньше, чем конструкторы производных, то данные-члены производного класса еще не инициализированы во время работы конструктора базового класса. Это может стать причиной неопределенного поведения и близкого знакомства с отладчиком. Обращение к тем частям объекта, которые еще не были инициализированы, опасно, поэтому C++ не дает такой возможности.

Есть даже более фундаментальные причины. Пока над созданием объекта производного класса трудится конструктор базового класса, типом объекта является базовый класс. Не только виртуальные функции считают его таковым, но и все прочие механизмы языка, использующие информацию о типе во время исполнения (например, описанный в правиле 27 оператор dynamic_cast и оператор typeid). В нашем примере, пока работает конструктор Transaction, инициализируя базовую часть объекта BuyTransaction, этот объект относится к типу Transaction. Именно так его воспринимают все части C++, и в этом есть смысл: части объекта, относящиеся к BuyTransaction, еще не инициализированы, поэтому безопаснее считать, что их не существует вовсе. Объект не является объектом производного класса до тех пор, пока не начнется исполнение конструктора последнего.

То же относится и к деструкторам. Как только начинает исполнение деструктор производного класса, предполагается, что данные-члены, принадлежащие этому классу, не определены, поэтому C++ считает, что их больше не существует. При входе в деструктор базового класса наш объект становится объектом базового класса, и все части C++ – виртуальные функции, оператор dynamic_cast и т. п. – воспринимают его именно так.

Пример кода, вызывающего pure virtual function call:

```
#include <iostream>

class Base
{
 public:
  Base() { init(); }
  ~Base() {}

  virtual void log() = 0;

 private:
  void init() { log(); }
};

class Derived: public Base
{
 public:
  Derived() {}
  ~Derived() {}

  virtual void log() { std::cout << "Derived created" << std::endl; }
};

int main(int argc, char* argv[])
{
  Derived d;
  return 0;
}
```

Фулл по ссылке

беды с башкой (параметрами по умолчанию)

TL;DR

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

class A {
public:
    virtual void Foo (int n = 10) {
        cout << "A::Foo, n = " << n << endl;
    }
};

class B : public A {
public:
    virtual void Foo (int n = 20) {
        cout << "B::Foo, n = " << n << endl;
    }
};

int main() {
    A * pa = new B ();
    pa->Foo ();

    return 0;
}
```

выдаст `B::Foo, n = 10`, потому что компилятор строго следует стандарту языка, предписывающему подставить в код вызова функции значения параметров по умолчанию исходя из статического типа указателя, по которому осуществляется вызов виртуальной функции. Это A * в нашем случае, а значит значения параметра будет взято из декларации функции A::Foo.