Объектно-ориентированное программирование

2023

Наследование

В наследственной иерархии общая часть структуры и поведения сосредоточена в наиболее общем суперклассе.

По этой причине говорят о наследовании, как об иерархии *обобщение-специализация.*

Суперклассы при этом отражают наиболее общие, а подклассы - более специализированные абстракции, в которых члены суперкласса могут быть дополнены, модифицированы и даже скрыты.

Классификация животного мира

Конструкторы при наследовании

```
class employee {
// ...
public:
// ...
employee(char* n, int d);
};
class manager : public employee {
// ...
public:
// ...
manager(char* n, int i, int d);
};
```

```
manager::manager(char* n, int l, int d)
: employee(n,d), // вызов родителя
level(l), // аналогично level=l
group(0)
{
}
```

Деструкторы при наследовании

C++ вызывает деструктор для текущего типа и для всех его родителей.

Последовательность вызова конструкторов и деструкторов

Конструкторы вызываются начиная от родителя к наследнику.

Деструкторы вызываются начиная от наследника к родителю.

```
class A {}
class B : A {}
class C: B{}
```

Конструкторы:

`A, B, C`

Деструкторы:

`~C, ~B,~A`

Ссылка на родителя

```
class Parent {
public:
Parent(void);
~Parent(void);
void Foo(void);
};
class Child : public Parent {
public:
Child(void);
~Child(void);
void Foo(void);
};
```

```
void Parent::Foo(void)
{
std::cout << "Parent\n";
}

void Child::Foo(void)
{
Parent::Foo();
std::cout << "Child\n";
}
```

Примеры

01_FirstTryOnInheritance
02_ProtectedMembers
03_BaseClassAccessSpecifiersADemo
04_ResurectingMembersBackInContext
05_ConstructorsWithInheritance
06_InheritingBaseConstructors
07_InheritanceAndDestructors
08_ReusedSymbolsInInheritance

Модификаторы функций

Ключевое слово **virtual** опционально и поэтому немного затрудняло чтение кода, заставляя вечно возвращаться в вершину иерархии наследования, чтобы посмотреть объявлен ли виртуальным тот или иной метод.

Типовые ошибки: Изменение сигнатуры метода в наследнике.

Модификаторы функций

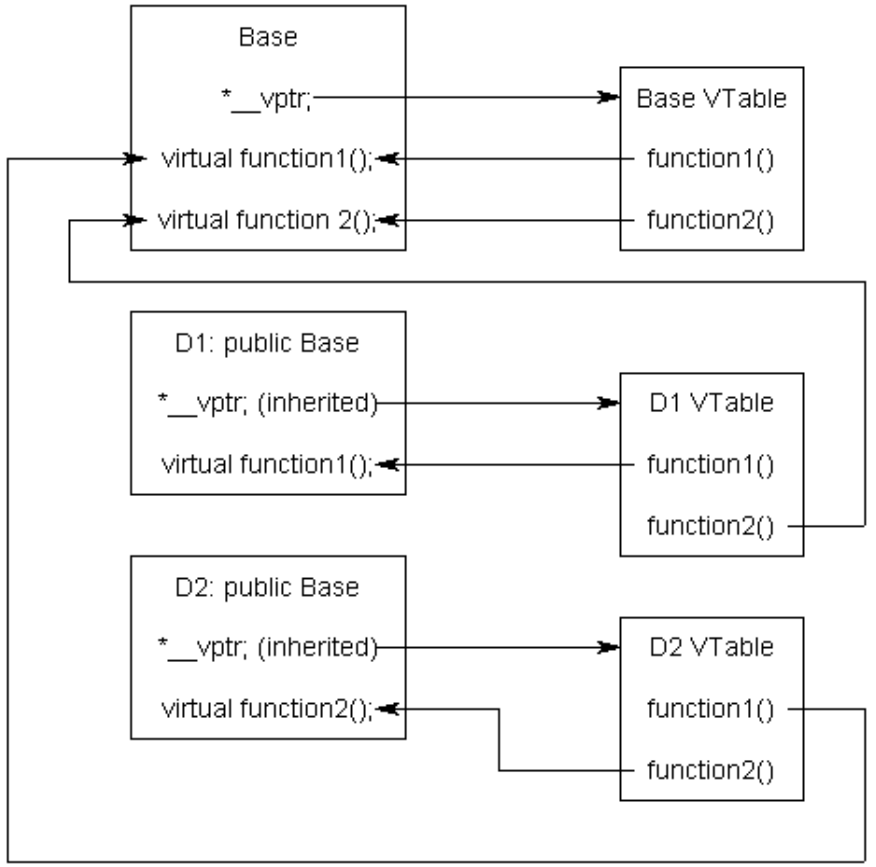
Модификатор **override** позволяет указать компилятору, что мы хотим переопределить виртуальный метод. Если мы ошиблись в описании сигнатуры метода – то компилятор выдаст нам ошибку. Этот модификатор влияет только на проверки в момент компиляции.

Виртуальные функции

```
class Base{
        virtual void function1();
        virtual void function2();
};

class D1 : public Base {
        void function1() override;
};

class D2 : public Base {
        void function2() override;
}
```



О виртуальных функциях

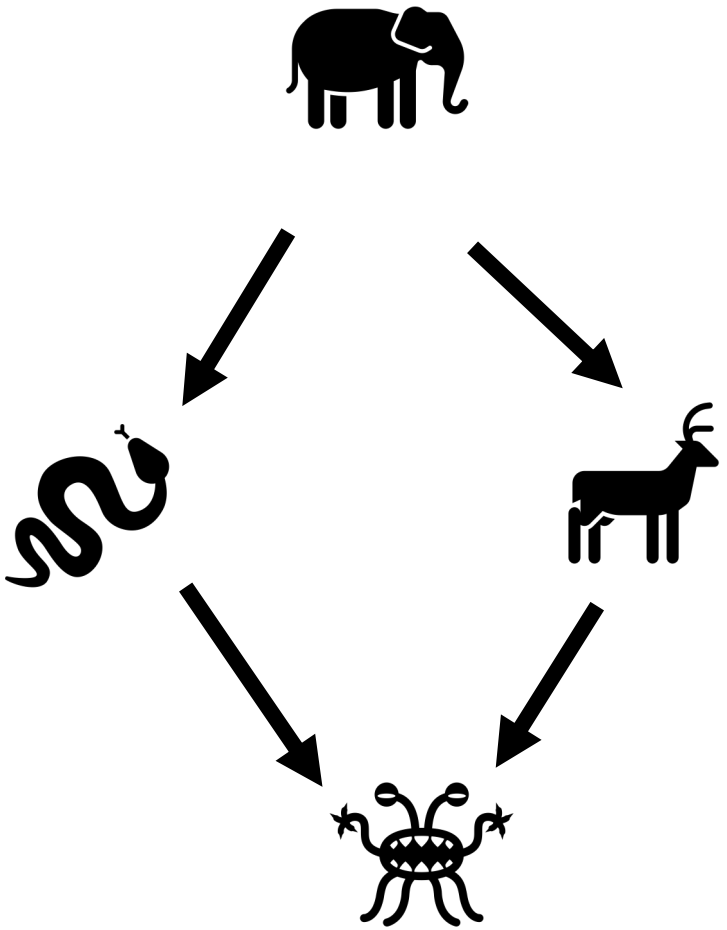
1. Виртуальную функцию можно использовать, даже если нет производных классов от ее класса.
2. В производном же классе не обязательно переопределять виртуальную функцию, если она там не нужна.
3. При построении производного класса надо определять только те функции, которые в нем действительно нужны.

Абстрактные классы

```
class Item
{
public:
        virtual const char * GetMyName() = 0;
};
```

Объект абстрактного класса нельзя создать!

Множественное наследование



Модификатор final

Модификатор **final**, указывающий что производный класс не должен переопределять виртуальный метод.

Работает только с модификатором **virtual**. Т.е. Создавать «копию» функции в классе-наследнике с помощью этой техники запретить нельзя.

Применяется, в случае если нужно запретить дальнейшее переопределение метода в дальнейших наследниках наследника (очевидно, что в родительском классе такой модификатор ставить бессмысленно).

Примеры

- 09_PolymorphismWithVirtualFunctions
- 10_PolymorphicObjectsStoredInCollections
- 11_Override
- 12_InheritanceAndPolymorphismWithStaticMembers
- 13_Final
- 14_PolymorphicFunctionsAndAccessSpecifiers
- 15_VirtualFunctionsWithDefaultArguments
- 16_DynamicCasts
- 17_TypeIdOperator
- 18_PureVirtualFunctionsAndAbstractClasses

Перегрузка операций

OPERATOR

Перегрузка операций

Почему операция **std::cin >> file_text** имеет смысл?

В C++ существуют механизмы, которые позволяют сопоставлять арифметический и другие операции, такие как побитовый сдвиг обычным функциям!

Это позволяет лучше описывать типы. Мы можем описать не просто класс, но и операции с объектами этого класса.

Предупреждение

- Это механизм, при неумелом использовании которого можно полностью запутать код.
- Непонятный код – причина сложных ошибок!
- Перегруженные операции помогают определить «свойства» созданного вами класса, но не алгоритма работы с классами!

Перегрузка операций

Можно описать функции, для описания следующих операций:

+ - * / % ^ & | ~ !

= < > += -= *= /= %= ^= &=

|= << >> >>= <<= == != <= >= &&

|| ++ -- ->* , -> [] () new delete

Нельзя изменить приоритеты этих операций, равно как и синтаксические правила для выражений. Так, нельзя определить унарную операцию % , также как и бинарную операцию !.

Синтаксис

```
type operator operator-symbol ( parameter-list )
```

Ключевое слово operator позволяет перегружать операции. Например:

- Перегрузка унарных операторов:
  - **ret-type operator** op **(** arg **)**
  - где **ret-type** и op соответствуют описанию для функций-членов операторов, а arg — аргумент типа класса, с которым необходимо выполнить операцию.
- Перегрузка бинарных операторов
  - **ret-type operator** op**(arg1, arg2)**
  - где *ret-type* и op — элементы, описанные для функций операторов членов, а arg1 и arg2 — аргументы. Хотя бы один из аргументов **должен принадлежать типу класса**.

Префиксные и постфиксные операторы ++ и --

Операторы инкремента и декремента относятся к особой категории, поскольку имеется два варианта каждого из них:

- преинкрементный и постинкрементный операторы;
- предекрементный и постдекрементный операторы.

При написании функций перегруженных операторов полезно реализовать отдельные версии для префиксной и постфиксной форм этих операторов. Для различения двух вариантов используется следующее правило: **префиксная** форма оператора объявляется точно так же, как и любой другой унарный оператор; в **постфиксной** форме принимается дополнительный аргумент типа **int**.

Пример:

```
friend Point& operator++( Point& ) // Prefix increment
friend Point& operator++( Point&, int ) // Postfix increment
friend Point& operator--( Point& ) // Prefix decrement
friend Point& operator--( Point&, int ) // Postfix decrement
```

Примеры

21_AdditionOperatorAsMember
22_AdditionOperatorAsNonMember
23_SubscriptOperatorReading
24_SubscriptOperatorReadingWritting
25_SubscriptOperatorForCollectionTypes
26_StreamInsertionOperator
27_StreamExtractionOperator
28_OtherArithmeticOperators
29_CompoundOperators_ReusingOperators
30_CustomTypeConversions
31_ImplicitConversionsWithOverloadedBinaryOperators
32_UnaryPrefixIncrementOperatorAsMember
33_UnaryPrefixIncrementOperatorAsNonMember
34_UnaryPostfixIncrementOperator
35_UnaryPrefixPostfixDecrementOperator
36_CopyAssignmentOperator
37_CopyAssignmentOperatorForOtherTypes
38_TypeConversionsRecap
39_Functors

Лабораторная работа №3

Разработать классы согласно варианту задания, классы должны наследоваться от базового класса Figure. Фигуры являются фигурами вращения.

- Все классы должны поддерживать набор общих методов:
- Вычисление геометрического центра фигуры вращения;
- Вывод в стандартный поток вывода std::cout координат вершин фигуры через перегрузку оператора << для std::ostream;
- Чтение из стандартного потока данных фигур через перегрузку оператора >> для std::istream
- Вычисление площади фигуры через перегрузку оператора приведения к типу double;

Создать программу, которая позволяет:

- Вводить из стандартного ввода std::cin фигуры, согласно варианту задания.
- Сохранять созданные фигуры в динамический массив (по аналогии с предыдущей лабораторной работой Array) указатели на фигуру (Figure*)
- Фигуры должны иметь переопределенные операции копирования (=), перемещения (=) и сравнения (==)
- Вызывать для всего массива общие функции (1-3 см. выше).Т.е. распечатывать для каждой фигуры в массиве геометрический центр и площадь.
- Необходимо уметь вычислять общую площадь фигур в массиве.
- Удалять из массива фигуру по индексу;

Спасибо!

НА СЕГОДНЯ ВСЕ