Федеральное агентство по образованию

Государственное Образовательное Учреждение Высшего

Профессионального Образования

Уфимский Государственный Авиационный Технический Университет

Кафедра ВМиК

Лабораторная работа №5

По дисциплине “Объектно-ориентированное программирование”

Тема: Жизненный цикл объектов С++ и Виртуальность

Выполнила:

Студентка группы ПРО-229

Яппарова А.Р.

Проверил:

доцент кафедры ВМиК

Котельников В.А.

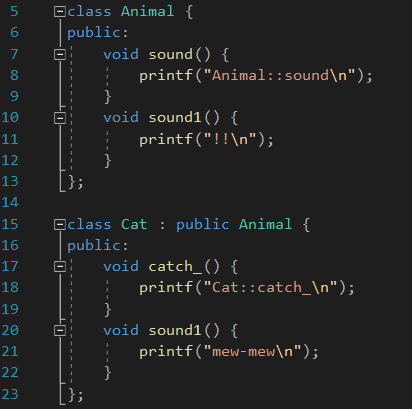
**Цель**: Определение и реализация тестовых классов, и написание программы, иллюстрирующей их использование.

**Задачи:**

* Определения
  + перекрываемых методов (показать на примере случай, когда вызывается перекрываемый метод, а когда наследуемый)
  + виртуальных методов, в т.ч. деструкторов (показать на примере, когда вызывается наследуемый виртуальный метод, а когда базовый)
* Реализация
  + проверки на принадлежность некоторому классу
  + безопасного приведения типов (dynamic\_cast)
  + безопасного приведения типов (вручную)
* Передача объектов как параметров в функции и возвращения объектов как результата из функции, контроль их жизненного цикла
* Умные указатели unique\_ptr и shared\_ptr, их влияние на жизненный цикл объектов

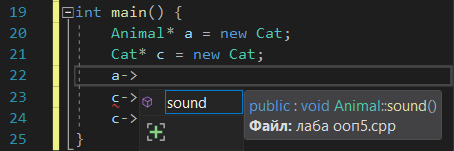
**Ход работы:**

1. Перекрываемые и наследуемые методы (лаба ооп5.cpp):



Метод sound(), наследуемый от класса Animal, а метод sound1(), перекрываемый метод. Создадим объекты класса Cat, лежащую в переменной класса предка(а) и лежащую в переменной своего собственного класса.

В переменной базового класса мы можем обратиться только к методу базового класса.

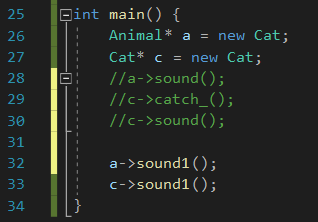


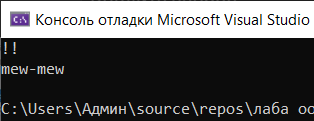
А в переменной с можно обратиться как к методам класса предка, так и к методам собственного класса.



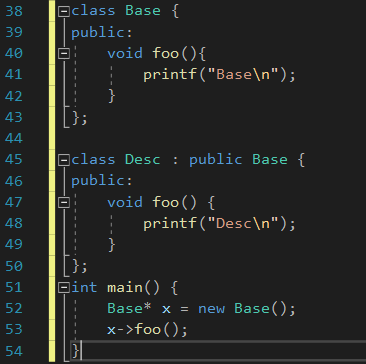
Перекрываемые методы (sound1()) в отличие от наследуемых перекрываются в классах потомках.

При вызове этого метода переменными разных классов будет совершаться разная работа.

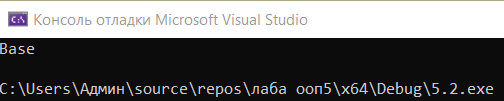




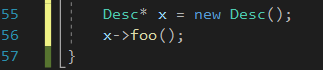
1. Виртуальные методы (5.2.cpp):

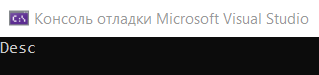


Создадим объект класса Base, лежащий в переменной собственного класса, и вызовем у него не виртуальный метод:



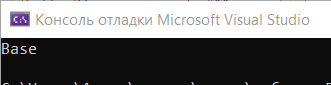
Создадим объект класса Desc, лежащий в переменной собственного класса, и вызовем у него не виртуальный метод:



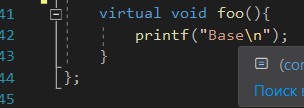


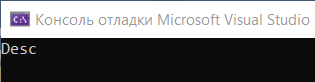
Создадим объект класса Desc, лежащий в переменной базового класса, и вызовем у него не виртуальный метод:





А если сделать виртуальным метод базового класса, то при таком же вызове этого метода получим:

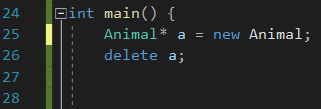




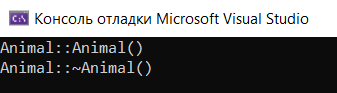
Виртуальный деструктор (5.5.cpp):



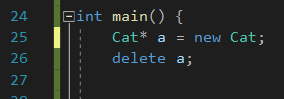
При вызове деструктора объекта базового класса, лежащего в переменной своего собственного класса



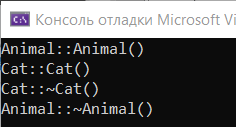
создастся конструктор Animal, затем его деструктор.



При вызове деструктора объекта класса наследника, лежащего в переменной своего собственного класса

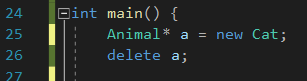


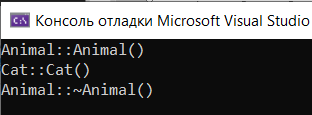
создастся конструктор Animal, затем конструктор Cat, потом деструкторы Cat и Animal соответственно.



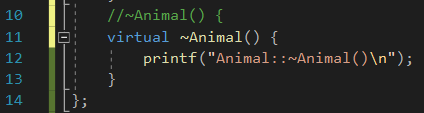
Если объекты лежат в переменных своего собственного класса, то им виртуальный деструктор не нужен.

При создании объекта класса Cat, помещенной в переменную базового класса попробуем вызвать не виртуальный деструктор у этой переменной.

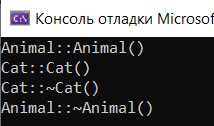




Деструктор класса Cat не вызвался, для того чтобы он все-таки вызвался нужен виртуальный деструктор:

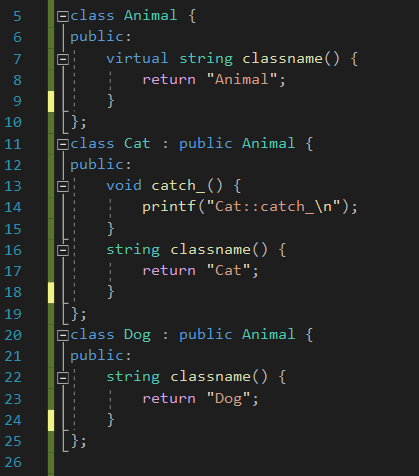


Тогда получим:

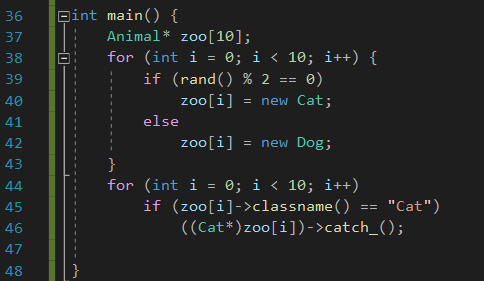


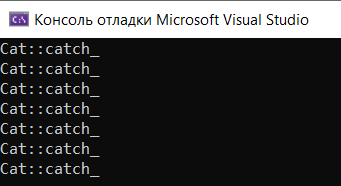
1. Реализация проверки на принадлежность некоторому классу (5.5.cpp):

В классе Animal создадим classname() и переопределим этот метод у всех его потомков.



Попробуем вызвать метод classname() у переменной базового класса zoo[i], в которой лежат объекты потомков, при условии, что эта переменная на выходе выдает “Cat”



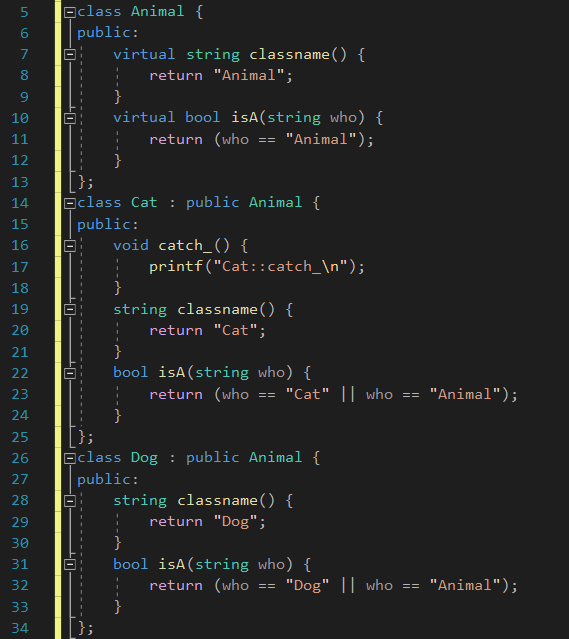


Мы это делали для того, чтобы выяснить у объекта кем он является, а потом небезопасно привели этот указатель в указатель на Cat и вызвали метод Catch\_.

Нарушается принцип инкапсуляции. Мы должны не запрашивать у этого объекта информацию, проводить вычисления и сделать вывод, а должны передать этому объекту всю необходимую информацию, чтобы он сам провел все необходимые вычисления.

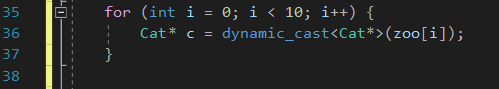
1. Реализация безопасного приведения типов (вручную) (5.4.cpp)

Все, что было выше — это небезопасное приведение типов, чтобы сделать его безопасным создадим в базовом классе метод isA, который будет сравнивать свой classname со строчкой, которую получили в параметр, и будет возвращать результат сравнения. Реализуем метод classname во всех наследниках. Теперь принцип инкапсуляции не нарушается.



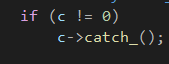
1. Реализация безопасного приведения типов (dynamic\_cast) (5.5.cpp):

Сейчас вместо isA существует другой безопасный способ приведения типов – dynamic\_cast. Это тот же самый isA, только dynamic\_cast не только проверяет, но и приводит.

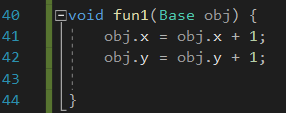


Если проверка справа от знака равенства прошла успешно, то в с будет тот же адрес что и в zoo[i], а если эта проверка не проходит, то в с окажется NULL.

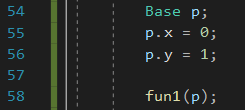
Нам остается добавить:

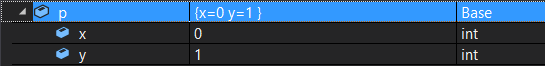


1. Передача объектов как параметров в функции и возвращения объектов как результата из функции, контроль их жизненного цикла (5.6.cpp):
2. Первая функции принимает во вход объект, тем самым создавая его копию

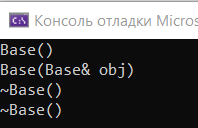


поэтому, при запуске функции, на выходе из нее значения p.x и p.y останутся неизменными

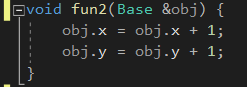


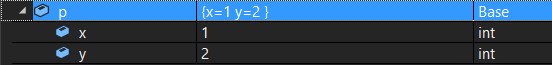


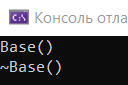
Вызовутся следующие конструкторы и деструкторы:



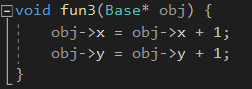
б) Функция 2 не создает копию объекта, а получает на вход адрес этого объекта

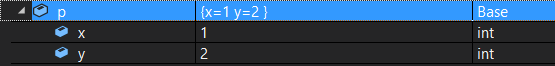


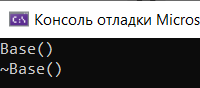




в) Функция 3 так же не создает копию объекта, в нее мы передаем не сам объект, а указатель на него

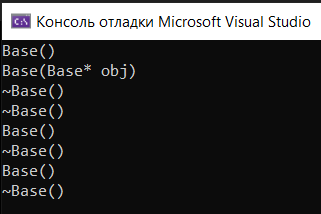
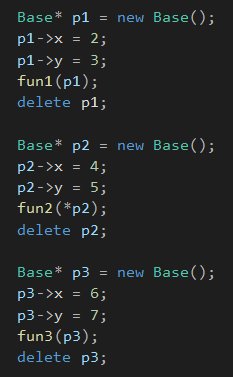


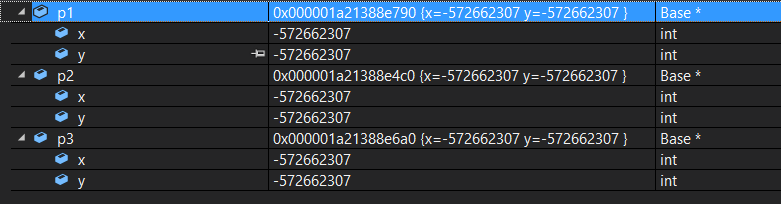




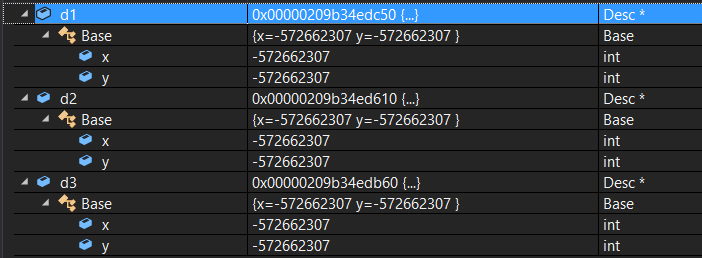
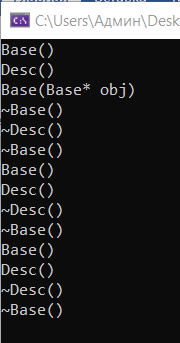
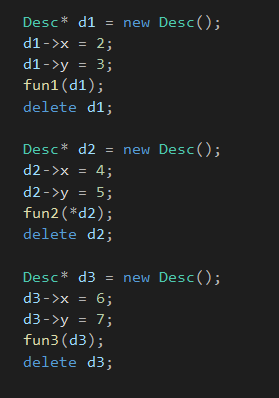
При создании объектов динамически:

Создается объект класса Base и присваивается переменной указателю на Base

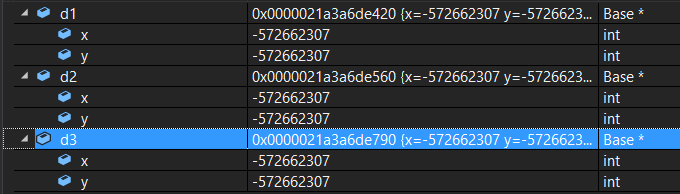
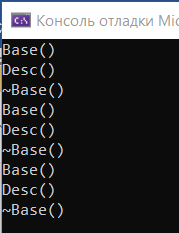
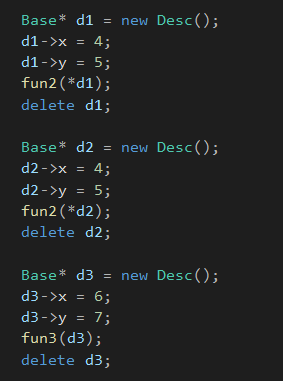




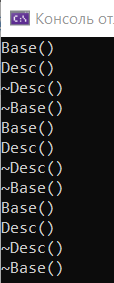
Создается объект класса Desc (наследника класса Base) и присваивается переменной указателю на свой собственный класс:



Создается объект класса Desc и присваивается переменной указателю на базовый класс Base:



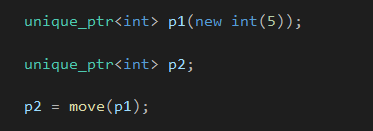
Если деструктор в базовом классе сделать виртуальным, то вызовутся деструкторы наследника:

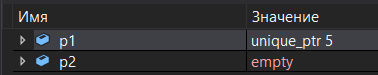


1. Умные указатели unique\_ptr и shared\_ptr, их влияние на жизненный цикл объектов (5.7.cpp):

Unique\_ptr – уникальный указатель:

Не позволяет одному указателю передать другой указатель, для смены владельца данных используется функция move







Вместо move можно использовать собственный метод указателя swap.

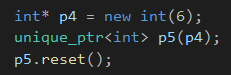


Есть другие методы unique\_ptr:



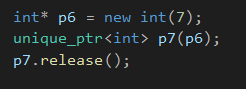
Делает умный указатель обычным

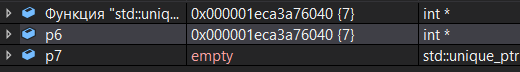
Когда вызывается метод reset p5 затирает данные, на которые указывает, следовательно становится пустым



Метод release:



Данные остаются в памяти, но указатель их забывает

Shared\_ptr – общий указатель

Позволяет нескольким указателям владеть одним и тем же объектом

