Лекция 8. Виртуальные функции и указатель this

Казахский национальный университет имени аль-Фараби

Кафедра компьютерных наук

Объектно-ориентированное программирование

Лектор: старший преподаватель, PhD, Кенжебек Ержан Ғалымжанұлы

Цель лекции

Цель лекции

Изучить виртуальные функции, позднее связывание, абстрактные классы, виртуальные деструкторы и указатель this.

Ожидаемые результаты

1. Объяснять разницу между обычными и виртуальными функциями при доступе через указатель
2. Определять абстрактные классы и чистые виртуальные функции
3. Объявлять виртуальные деструкторы
4. Использовать указатель this

План лекции

- Доступ к обычным функциям через указатели
- Виртуальные функции
- Доступ к виртуальным методам через указатели
- Виртуальные деструкторы
- Указатель this

Про лекцию

- Эта лекция посвящена таким довольно слабо связанным между собой темам, как виртуальные функции, дружественные функции, статические функции, перегружаемые операции и методы, а также указатель this.
- Все это — прогрессивные особенности языка, однако далеко не везде их стоит использовать. Небольшие программы могут вполне обойтись и без них. Тем не менее они часто применяются и очень важны для больших, серьезных программ. В частности, **виртуальные функции необходимы при использовании полиморфизма** — одного из краеугольных камней ООП

Виртуальные функции

- **Виртуальный** означает **видимый**, **но не существующий в реальности**. Когда используются виртуальные функции, программа, которая, казалось бы, вызывает функцию одного класса, может в этот момент вызывать функцию совсем другого класса. А для чего вообще нужны виртуальные функции? Представьте, что имеется набор объектов разных классов, но вам хочется, чтобы они все были в одном массиве и вызывались с помощью одного и того же выражения

Доступ к обычным функциям через указатели (NotVirtual.cpp)

```cpp
// Доступ к обычным функциям через указатели
#include <iostream>
using namespace std;
//////////////////////////////////////////////
class Base // Базовый класс
 {
 public:
 void show() // Обычная функция
 { cout << "Base\n"; }
 };
//////////////////////////////////////////////
class Derv1 : public Base // Производный класс 1
 {
 public:
void show()
 { cout << "Derv1\n"; }
 };
//////////////////////////////////////////////
class Derv2 : public Base // Производный класс 2
 {
 public:
 void show()
 { cout << "Derv2\n"; }
 };
//////////////////////////////////////////////
int main()
{
 Derv1 dv1; // Объект производного класса 1
 Derv2 dv2; // Объект производного класса 2
 Base* ptr; // Указатель на базовый класс
 ptr = &dv1; // Адрес dv1 занести в указатель
 ptr->show(); // Выполнить show()
 ptr = &dv2; // Адрес dv2 занести в указатель
 ptr->show(); // Выполнить show()
 return 0;
}
```

Наш первый пример покажет, что бывает, когда базовый и производные классы содержат функции с одним и тем же именем, и к ним обращаются с помощью указателей, но без использования виртуальных функций


```cpp
// Доступ к обычным функциям через указатели
#include <iostream>
using namespace std;
//////////////////////////////////////////////
class Base // Базовый класс
 {
 public:
 void show() // Обычная функция
 { cout << "Base\n"; }
 };
//////////////////////////////////////////////
class Derv1 : public Base // Производный класс 1
 {
 public:
void show()
 { cout << "Derv1\n"; }
 };
//////////////////////////////////////////////
class Derv2 : public Base // Производный класс 2
 {
 public:
 void show()
 { cout << "Derv2\n"; }
 };
//////////////////////////////////////////////
int main()
{
 Derv1 dv1; // Объект производного класса 1
 Derv2 dv2; // Объект производного класса 2
 Base* ptr; // Указатель на базовый класс
 ptr = &dv1; // Адрес dv1 занести в указатель
 ptr->show(); // Выполнить show()
 ptr = &dv2; // Адрес dv2 занести в указатель
 ptr->show(); // Выполнить show()
 return 0;
}
```

Доступ к объектам через указатели

- ptr = &dv1; // Адрес объекта порожденного класса заносим в указатель базового
- Но постойте, а компилятор на нас не обидится за то, что мы присваиваем адрес объекта одного типа указателю на другой тип? Оказывается, наоборот — компилятор будет просто счастлив, потому что проверка типов отдыхает в этой ситуации. Мы скоро поймем, почему. **Дело в том, что указатели на объекты порожденных классов совместимы по типу с указателями на объекты базового класса.**
- Теперь хорошо бы понять, какая же, собственно, функция выполняется**?**

```
ptr->show();
ptr->show();
```

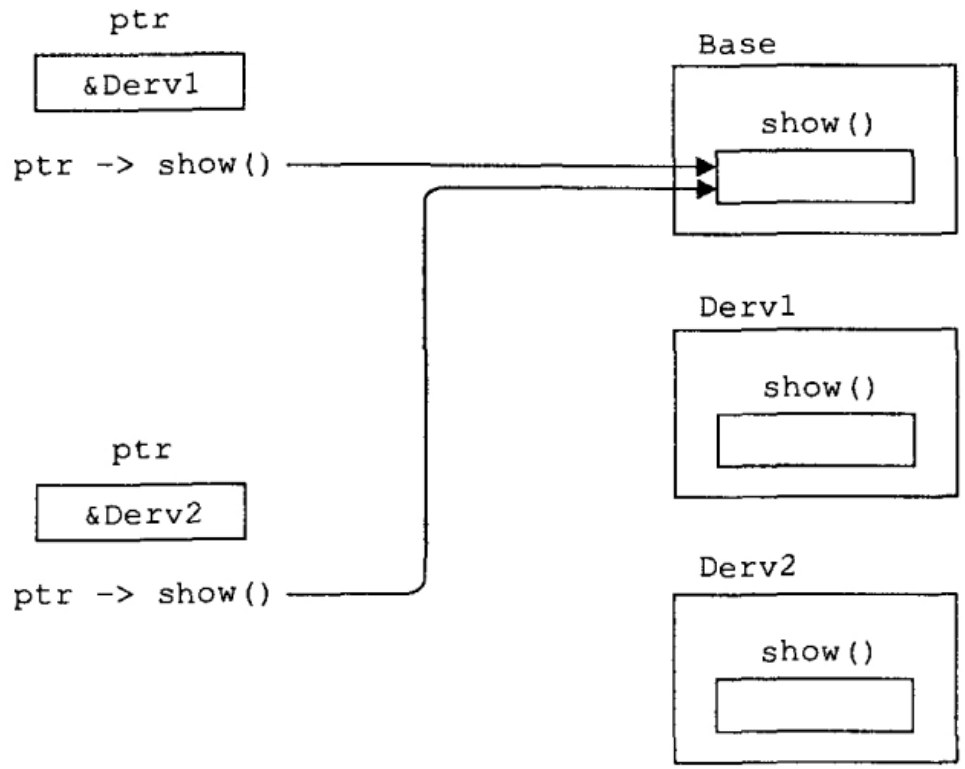
Результат выполнения программы дает простой ответ:

Base

Base

Как видите, всегда выполняется метод базового класса. Компилятор не смотрит на содержимое указателя ptr, а выбирает тот метод, который удовлетворяет типу указателя,

Доступ через указатель без использования вирутальных функций



с. 11.1. Доступ через указатель без использования виртуальных функций

Вывод

- Да, иногда именно это нам и нужно, но таким образом не решить поставленную в начале этой темы проблему доступа к объектам разных классов с помощью одного выражения.

Доступ к виртуальным методам через указатели (Virtual.cpp)

- Давайте сделаем одно маленькое изменение в нашей программе: поставим ключевое слово virtual перед объявлением функции show() в базовом классе.

```
1  // Доступ к виртуальным функциям через указатели
2  #include <iostream>
3  using namespace std;
4  ////////////////////////////////////////////
5  class Base // Базовый класс
6   {
7   public:
8   virtual void show() // виртуальная функция
9   { cout << "Base\n"; }
10  };
11 ////////////////////////////////////////////
12 class Derv1 : public Base // Производный класс 1
13  {
14  public:
15 void show()
16  { cout << "Derv1\n"; }
17  };
18 ////////////////////////////////////////////
19 class Derv2 : public Base // Производный класс 2
20  {
21  public:
22  void show()
23  { cout << "Derv2\n"; }
24  };
25 ////////////////////////////////////////////
26 int main()
27 {
28  Derv1 dv1; // Объект производного класса 1
29  Derv2 dv2; // Объект производного класса 2
30  Base* ptr; // Указатель на базовый класс
31  ptr = &dv1; // Адрес dv1 занести в указатель
32  ptr->show(); // Выполнить show()
33  ptr = &dv2; // Адрес dv2 занести в указатель
34  ptr->show(); // Выполнить show()
35  return 0;
36 }
```

На выходе имеем:

Derv1

Derv2

Доступ через указатель к виртуальным функциям

- Теперь, как видите, выполняются методы производных классов, а не базового. Мы изменили содержимое ptr с адреса из класса Derv1 на адрес из класса Derv2, и изменилось реальное выполнение show().
- Значит, один и тот же вызов ptr->show() ставит на выполнение разные функции в зависимости от содержимого ptr. **Компилятор выбирает функцию, удовлетворяющую тому, что занесено в указатель, а не типу указателя, как было в предыдущей программе**

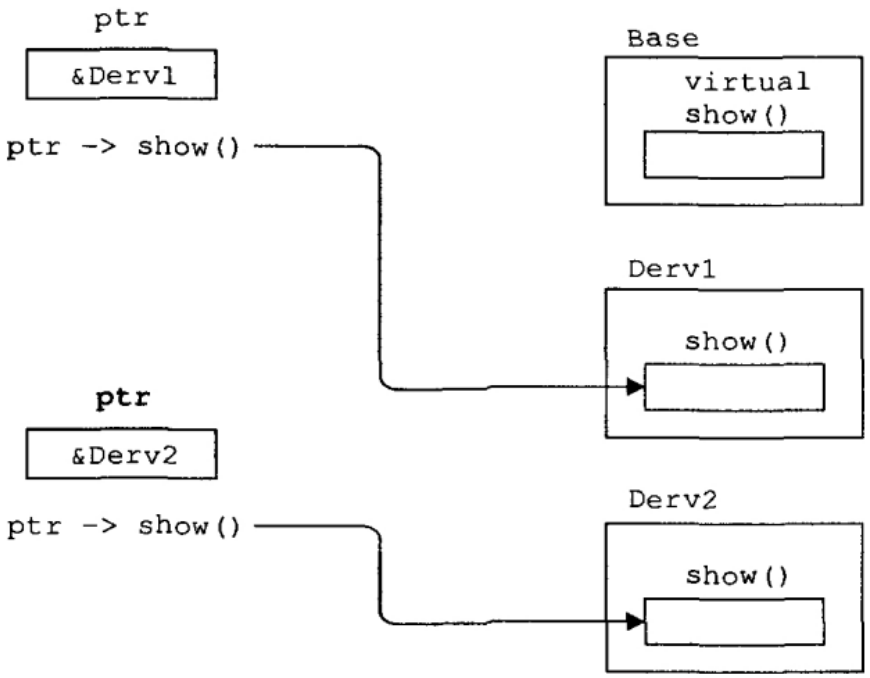


Рис. 11.2. Доступ через указатель к виртуальным функциям

Позднее связывание

- Любознательный читатель может удивиться, как же компилятор узнает, какую именно функцию ему компилировать? В программе **NotVirtual.cpp** у компилятора нет проблем с выражением ptr->show(); Он всегда компилирует вызов функции show() из базового класса. Однако в программе VIRT компилятор не знает, к какому классу относится содержимое ptr. Ведь это может быть адрес объекта как класса Derv1, так и класса Derv2.
- На самом деле компилятор не очень понимает, что ему делать, поэтому откладывает принятие решения до фактического запуска программы. А когда программа уже поставлена на выполнение, когда известно, на что указывает ptr, тогда будет запущена соответствующая версия draw. Такой подход называется поздним связыванием или динамическим связыванием. (Выбор функций в обычном порядке, во время компиляции, называется ранним связыванием или статическим связыванием.) Позднее связывание требует больше ресурсов, но дает выигрыш в возможностях и гибкости.

Абстрактные классы

- Базовый класс, объекты которого никогда не будут реализованы, называется абстрактным классом. Такой класс может существовать с единственной целью — быть родительским по отношению к производным классам, объекты которых будут реализованы. Еще он может служить звеном для создания иерархической структуры классов.
- Как нам объяснить людям, использующим созданную нами структуру классов, что объекты родительского класса не предназначены для реализации? Можно, конечно, заявить об этом в документации, но это никак не защитит наш базозый класс от использования не по назначению. Надо защитить его программно. Для этого достаточно ввести в класс хотя бы одну чистую виртуальную функцию. Чистая виртуальная функция — это функция, после объявления которой добавлено выражение = 0.

Чистая виртуальная функция

```cpp
// Чистая виртуальная функция
#include <iostream>
using namespace std;
class Base // базовый класс
{
public:
virtual void show() = 0; // чистая виртуальная функция
};
class Derv1 : public Base // порожденный класс 1
{
public:
void show()
{ cout << "Derv1\n"; }
};
//////////////////////////////////////////////
class Derv2 : public Base // порожденный класс 2
{
public:
void show()
{ cout << "Derv2\n"; }
};
//////////////////////////////////////////////
int main()
{
// Base bad; // невозможно создать объект
// из абстрактного класса
Base* arr[2]; // массив указателей на
// базовый класс
Derv1 dv1; // Объект производного класса 1
Derv2 dv2; // Объект производного класса 2

arr[0] = &dv1; // Занести адрес dv1 в массив
arr[1] = &dv2; // Занести адрес dv2 в массив
arr[0]->show(); // Выполнить функцию show()
arr[1]->show(); // над обоими объектами
return 0;
}
```

Конструкция = 0 — это просто способ сообщить компилятору, что функция будет чистой виртуальной. Если теперь в main() попытаться создать объект класса Base, то компилятор будет недоволен тем, что объект абстрактного класса пытаются реализовать. Он выдаст даже имя чистой виртуальной функции, которая делает класс абстрактным. Помните, что хотя это только объявление, но определение функции show() базового класса не является обязательным. Впрочем, если вам надо его написать, это можно сделать

Результат программы VirtualPure.cpp

- Между прочим, мы внесли еще одно, не связанное с предыдущими, изменение в VIRTPURE: теперь адреса методов хранятся в массиве указателей и доступны как элементы этого массива.
- Обработка этого, впрочем, ничем не отличается от использования единственного указателя. VIRTPURE выдает результат, не отличающийся от VIRT:

Derv1

Derv2

Виртуальный деструктор

```
class Base { public: ~Base() // невиртуальный деструктор
// virtual ~Base() // виртуальный деструктор
 {
  cout << "Base удален\n";
 }
};
```

Указатель this

- Каждый нестатический метод получает скрытый указатель this на объект, для которого он вызван
- Через this можно обратиться к полю, скрытому параметром с тем же именем
- return *this позволяет строить цепочки вызовов
- В константных методах this имеет тип const T*

```
class Counter {
  int n = 0;
public:
  Counter& add(int n) {
    this->n += n;  // поле += параметр
    return *this;  // для цепочки
  }
  void show() const { cout << n; }
};

// Counter c;
// c.add(1).add(2).show();  // 3
```

Выводы

1. Без ключевого слова virtual вызов через указатель на базовый класс всегда выполняет метод базового класса.
2. Виртуальная функция выбирается по типу реального объекта, а не по типу указателя (позднее связывание).
3. Абстрактный класс содержит хотя бы одну чистую виртуальную функцию (= 0); создавать его объекты нельзя.
4. Деструктор базового класса следует делать виртуальным, чтобы при удалении через указатель вызывался деструктор производного класса.
5. Указатель this ссылается на объект, для которого вызван метод.

Контрольные вопросы

1. Что произойдёт при вызове ptr->show() через указатель на базовый класс, если функция не виртуальна?
2. Как ключевое слово virtual меняет поведение программы?
3. Что такое позднее связывание?
4. Что такое чистая виртуальная функция и абстрактный класс?
5. Зачем нужен виртуальный деструктор? К чему приводит его отсутствие?
6. Что такое указатель this и для чего он используется?

Список литературы

Основная литература

1. Лафоре Р. Объектно-ориентированное программирование в C++. 4-е изд. — СПб.: Питер, 2004.
2. Страуструп Б. Язык программирования C++. Специальное издание. — М.: Бином, 2011.
3. Липпман С., Лажойе Ж., Му Б. Язык программирования C++. Базовый курс. 5-е изд. — М.: Вильямс, 2014.

Дополнительная литература

4. Майерс С. Эффективный и современный C++. — М.: Вильямс, 2016.
5. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приёмы объектно-ориентированного проектирования. Паттерны проектирования. — СПб.: Питер, 2007.
6. Шилдт Г. Полный справочник по C++. 4-е изд. — М.: Вильямс, 2006.

Электронные ресурсы

7. C++ Core Guidelines: isocpp.github.io/CppCoreGuidelines
8. Справочник по C++ и стандартной библиотеке: en.cppreference.com

- Спасибо за внимание!