Наследование классов

М. А. Ложников

17 октября 2019 г.
Ревизия: 1

Это предварительная невыверенная версия. Читайте на свой страх и риск.

1 Постановка задачи

Реализуем классы кондукторов автобусов и поездов `BusConductor` и `TrainConductor` соответственно, которые продают взрослые `AdultTicket` и детские `ChildTicket` билеты.

```
#include <iostream>
#include <string>

struct AdultTicket {
  int cost;   // Стоимость билета.

  AdultTicket() :
    cost(10)  // Взрослый билет стоит 10 фунтов.
  { }
};

struct ChildTicket {
  int cost;   // Стоимость билета.

  ChildTicket() :
    cost(5)   // Детский билет стоит 5 фунтов.
  { }
};

/*
  В C++11 то же самое можно записать короче, а именно
  struct AdultTicket {
    int cost = 10;
  };

  struct ChildTicket {
    int cost = 5;
  };
  Компилятор автоматически создаст соответствующие конструкторы.
*/

```

```
32 class BusConductor {
33  public:
34   /* Метод для продажи взрослого билета. */
35   void SellTicket(const AdultTicket& ticket) const {
36     std::cout << "Bus conductor sold an adult ticket and harvested "
37               << ticket.cost << " pounds." << std::endl;
38   }
39
40   /* Метод для продажи детского билета. */
41   void SellTicket(const ChildTicket& ticket) const {
42     std::cout << "Bus conductor sold a child ticket and harvested "
43               << ticket.cost << " pounds." << std::endl;
44   }
45 };
46
47 class TrainConductor {
48  public:
49   /* Метод для продажи взрослого билета. */
50   void SellTicket(const AdultTicket& ticket) const {
51     std::cout << "Bus conductor sold an adult ticket and harvested "
52               << ticket.cost << " pounds." << std::endl;
53   }
54
55   /* Метод для продажи детского билета. */
56   void SellTicket(const ChildTicket& ticket) const {
57     std::cout << "Bus conductor sold a child ticket and harvested "
58               << ticket.cost << " pounds." << std::endl;
59   }
60
61   /* В отличие от кондуктора автобуса, кондуктор поезда может дёрнуть стоп-кран. */
62   void EmergencyBreak() const {
63     std::cout << "Train conductor uses the emergency break!" << std::endl;
64   }
65 };
```

Замечание 1.1 *Внимательный читатель, возможно, заметил, что при написании класса* `TrainConductor` *из-за копирования была допущена непреднамеренная ошибка: в методе* `SellTicket()` *вместо* `Train` *выведется* `Bus`*. Этот пример показывает, что следует избегать копирования и дублирования кода.*

Рассмотрим пример использования этих классов. В данном примере аргументы метода `SellTicket()` создаются при помощи стандартных конструкторов структур `AdultTicket` и `ChildTicket`.

```
1 int main() {
2   BusConductor busConductor;
3   TrainConductor trainConductor;
4
5   busConductor.SellTicket(AdultTicket());
6   busConductor.SellTicket(ChildTicket());
7   trainConductor.SellTicket(AdultTicket());
```

```
  trainConductor.SellTicket(ChildTicket());
  trainConductor.EmergencyBreak();
  return 0;
}
```

Недостатками предложенного подхода являются

- Избыточность кода. Метод `SellTicket` дублирован четыре раза.
- Сложность поддержки кода. Необходимо дописать много кода для добавления, например, билета для пенсионеров или кондуктора трамвая.
- Большая вероятность наличия ошибок, связанная с дублированием кода.

В следующем разделе мы рассмотрим подход, позволяющий избежать этих недостатков.

2 Решение проблемы с помощью механизма наследования

Попробуем вынести общие черты структур `AdultTicket` и `ChildTicket` (то есть поле `cost`) в некоторую базовую структуру `Ticket`, а общие черты классов `BusConductor` и `TrainConductor` (то есть метод `SellTicket`) в некоторый базовый класс `Conductor`.

```
#include <iostream>
#include <string>

/* Базовая структура для любого типа билета. */
struct Ticket {
  int cost;           // Стоимость билета.
  std::string type;   // Тип билета.

  Ticket() :
    cost(-1),
    type("unknown")
  { }
};

/* В C++11 можно написать короче:
  struct Ticket {
    int cost = -1;                 // Значение по умолчанию для цены билета
    std::string type = "unknown";  // Значение по умолчанию для типа билета
  };
*/

/* Конструкция ниже говорит о том, что структура AdultTicket является публичным
  наследником (public) базовой структуры Ticket. При публичном наследовании public
  члены базовой структуры (класса) становятся public членами производной структуры
  (класса); protected члены базовой структуры (класса) доступны в производной структуре
  (классе) как protected, а private члены базовой структуры (класса) недоступны
  в производной(ом). */
struct AdultTicket : public Ticket {
  AdultTicket() {
```

```
30      /* Значения полей базового класса (структуры) нельзя задавать в списке инициализации
31         производного. Поэтому задаём их в теле конструктора. */
32      cost = 10;
33      type = "adult";
34    }
35  };
36
37  /* Производная структура ChildTicket является наследницей базовой структуры Ticket. */
38  struct ChildTicket : public Ticket {
39    ChildTicket() {
40      /* В отличие от списка инициализации внутри тела конструктора производного класса
41         можно задать значения полей базового класса. */
42      cost = 5;
43      type = "child";
44    }
45  };
46
47  /* Базовый класс для кондукторов. */
48  class Conductor {
49   public:
50    Conductor(const std::string& newType = "Unknown") :
51      type(newType)
52    { }
53
54    /* Указатели и ссылки на производные классы можно приравнивать, соответственно
55       указателям и ссылкам на базовые классы. В получившейся переменной будут доступны
56       все поля и методы базового класса. Обратная операция требует явного приведения
57       типов и не всегда работает. Таким образом, в данную функцию можно передать
58       не только объекты типа Ticket, но и объекты типа AdultTicket и ChildTicket. */
59    void SellTicket(const Ticket& ticket) const {
60      std::cout << type << " conductor sold a(n) " << ticket.type
61                << " ticket and harvested " << ticket.cost << " pounds." << std::endl;
62    }
63
64   // protected переменные доступны в методах производного класса
65   protected:
66    std::string type; // Тип кондуктора
67  };
68
69  // Производный класс BusConductor является наследником базового класса Conductor
70  class BusConductor : public Conductor {
71   public:
72    BusConductor() :    // В списке инициализации первым вызывается конструктор базового
73      Conductor("Bus") // класса. Если нужен стандартный конструктор базового класса,
74    { }                // то его явно вызывать не нужно, он вызывается автоматически.
75  };
76
77  class TrainConductor : public Conductor {
78   public:
79    TrainConductor() :   // После вызова конструктора базового класса в списке
80      Conductor("Train") // инициализации должны следовать поля производного класса
```

```
  { }                    // в порядке их объявления в классе. В данном примере их нет.

  // В отличие от кондуктора автобуса, кондуктор поезда может дёрнуть стоп-кран.
  void EmergencyBreak() const {
    std::cout << "Train conductor uses the emergency break!" << std::endl;
  }
};
```

Замечание 2.1 *Если сделать поле* `type` *класса* `Conductor` *приватным (*`private`*), то оно будет недоступно в производных классах. Поля и методы с доступом* `protected` *доступны в самом классе и во всех его потомках, но недоступны извне класса и его потомков.*

Проиллюстрируем работу данных классов

```
int main() {
  BusConductor busConductor;
  TrainConductor trainConductor;
  Conductor conductor;

  busConductor.SellTicket(AdultTicket());
  busConductor.SellTicket(ChildTicket());
  trainConductor.SellTicket(AdultTicket());
  trainConductor.SellTicket(ChildTicket());
  trainConductor.EmergencyBreak();
  conductor.SellTicket(AdultTicket());
  conductor.SellTicket(ChildTicket());
  return 0;
}
```

В данном примере все три класса вызывают один и тот же метод `SellTicket` базового класса `Conductor`. Нам удалось убрать дублирование кода и структурировать программу и типы данных.

3 О связи между базовым и производным классами

```
// Базовый класс
class Base {
 public:
  int baseElem;
};

// Производный класс
class Derived : public Base {
 public:
  int derivedElem;
};

void Function(Base* base, Derived* derived, Base& baseRef, Derived& derivedRef)
{
```

```
    /* Указатели или ссылки на производные классы можно присваивать соответственно
       указателям или ссылкам на базовые классы без явного приведения типа. Например, */
    Base* base2 = derived;

    // В получившемся объекте будут доступны все поля и методы базового класса.
    base2->baseElem = 5; // Корректно, в результате derived->baseElem станер равным 5.

    // Аналогично со ссылками
    Base& baseRef2 = derivedRef;
    baseRef2.baseElem = 3; // Корректно, derivedRef->baseElem станер равным 3.

    /* Обратное преобразование требует явного приведения типа, то есть код
    // Derived* derived2 = base; не скомпилируется. Вместо этого придётся писать */
    Derived* derived2 = static_cast<Derived*>(base);

    /* static_cast - операция статического приведения типа в C++, она не делает проверку,
       был ли указатель (ссылка на) base изначально создан(а) как указатель (ссылка)
       на тип Derived. В угловых скобках пишется тип данных, к которому хотим привести
       значение, а в круглых - то, что хотим привести.
       Примечание: проверку иерархии наследования и корректность приведения типа
       в данном случае делает операция dynamic_cast. */
    derived2->baseElem = 8; // Корректно, base->baseElem станет равным 8.

    /* Однако следующий код может привести к падению программы, поскольку указатель
       derived2 мог изначально быть создан как указатель на Base и не содержать
       поля derivedElem.
    // derived2->derivedElem = 8; Пример "плохого" кода. */

    /* Аналогично, */
    Derived& derivedRef2 = static_cast<Derived&>(baseRef);
    derivedRef2.baseElem = 7; // Корректно, baseRef.baseElem станет равным 7.
    // derivedRef2.derivedElem = 8; Пример "плохого" кода.

    /* Если забыть поставить указатель или ссылку, то произойдёт полное поэлементное
       копирование полей, унаследованных из базового класса в новый объект.
       Для копирования будет использоваться функция operator=() класса Base,
       созданная компилятором по умолчанию и выполняющая поэлементное копирование.
       Такая операция называется срезкой (slicing). */
    Base baseCopy = derivedRef; // Этот объект никак не связан с derivedRef.
    baseCopy.baseElem = 1; // Никак не отразится на объекте derivedRef.
}
```

4 Контроль доступа в производных классах

Предположим, что в автобусе/поезде появились “менеджеры по продажам билетов”, которые повышают стоимость билета за свои услуги. Выразим это в функции `HarvestAndReap`:

```
void HarvestAndReap(Conductor& conductor, Ticket& ticket)
{
```

```
  ticket.cost += 2;
  conductor.SellTicket(conductor, ticket);
}
```

Чтобы защититься от такого случая можно объявить поле `cost` структуры `Ticket` как `protected`, тогда оно будет доступно только в методах базового и производного классов. Напомним, что по умолчанию элементы структур объявляются как `public`. Второй способ заключается в том, чтобы объявить цену и тип константными. Однако, он потребует модификацию структур `Ticket`, `AdultTicket` и `ChildTicket` поскольку константные поля структуры (класса) могут быть инициализированы только в списке инициализации в конструкторе или с помощью операции присваивания при объявлении поля (в C++11). Рассмотрим подробно этот способ.

```
struct Ticket {
  const int cost;
  const std::string type;

  Ticket(int newCost = -1, const std::string& newType = "unknown") :
    cost(newCost), // Задаём константы в списке инициализации конструктора
    type(newType)
  { }
};

struct AdultTicket : public Ticket {
  /* Нам необходимо инициализировать поля cost и type, которые были объявлены в базовом
     классе. Поскольку они объявлены как константы, то их инициализация возможна только
     в списке инициализации конструктора. Проблема заключается в том, что в списке
     инициализации производного класса нельзя явно инициализировать поля базового
     класса. На помощь приходит специальный конструктор базового класса. */
  AdultTicket() :        // Вызываем в списке инициализации конструктор базовой
    Ticket(10, "adult") // структуры, который инициализирует её поля.
  { }
};

struct ChildTicket : public Ticket {
  ChildTicket() :        // Нам нужен специальный конструктор базовой структуры, поэтому
    Ticket(5, "adult")  // его следует вызвать явно.
  { }
};
```

Замечание 4.1 *В данном примере для инициализации полей базового класса используется явный вызов конструктора базового класса в конструкторе производного класса. Данный способ инициализации полей, оказавшихся "по наследству" в производном классе является предпочтительным.*

Данная техника позволяет структурировать программу и защитить поля класса от случайных изменений.

5 Очерёдность конструкторов и деструкторов

Рассмотрим следующий пример:

```cpp
#include <iostream>
#include <string>

/* Базовый класс. Он выводит информацию в момент вызова своего конструктора и
   деструктора. */
class Base {
 public:
  Base() {
    std::cout << "Undefined base class is constructed!" << std::endl;
  }

  /* Специальный конструктор умеет задавать id для класса. */
  Base(const std::string& newId) : // Конструктор
      id(newId) {
    std::cout << "Base '" << id << "' is constructed!" << std::endl;
  }

  ~Base() { // Деструктор
    std::cout << "Base '" << id << "' is destructed!" << std::endl;
  }
 protected:
  std::string id;
};

class Derived : public Base { // Производный класс
 public:
 /* Поскольку стандартный конструктор базового класса вызывается автоматически,
    то нет нужды вызывать его явно в списке инициализации конструктора производного
    класса. */
  Derived() :
      b1("Field 1 of unknown class"), // Порядок соответствует порядку объявления
      b2("Field 2 of unknown class") {
    std::cout << "Undefined derived class is constructed!" << std::endl;
  }

  /* Здесь необходимо вызвать специальный конструктор базового класса для того, чтобы
     задать поле id базового класса. Вызов соответствующего конструктора делается
     в списке инициализации производного класса. Этот вызов должен находиться
     по порядку раньше всех остальных элементов в списке инициализации. */
  Derived(const std::string& newId) :
      Base(newId), // Вызываем конструктор базового класса в списке инициализации
      b1("Field 1 of derived '" + newId + "'"),
      b2("Field 2 of derived '" + newId + "'") {
    std::cout << "Derived class '" << id << "' is constructed!" << std::endl;
  }

  ~Derived() {
    std::cout << "Derived class '" << id << "' is destructed!" << std::endl;
  }

```

```
51  private:  // Поля класса инициализируются строго в порядке их объявления в классе.
52    Base b1; // Их порядок в списке инициализации должен строго соответствовать.
53    Base b2; // При нарушении порядка в списке инициализации могут возникать ошибки.
54  };
55
56  int main() {
57    Base base("Class 0");
58    Derived derivedDefined("Class 1");
59    Derived derivedUndefined;
60
61    return 0;
62  }
```

Программа создаст классы в порядке их объявления в функции `main()` и уничтожит их при завершении функции `main()` в порядке, обратном порядку объявления. При создании объекта производного класса сначала вызовется конструктор базового класса, потом инициализируются поля производного класса, а затем вызывается конструктор производного класса. При уничтожении объекта производного класса сначала сработает деструктор производного класса, потом деструкторы его полей (если есть), а затем деструктор базового класса.

Внутри класса его поля создаются в порядке объявления в классе, именно поэтому в списке инициализации их нужно указывать в точно таком же порядке. В противном случае могут возникнуть ошибки, например, если значение одного из полей зависит от другого поля, которое должно было быть ранее инициализировано. При уничтожении объекта его поля уничтожаются в порядке, обратном порядку объявления.

Таким образом, программа выведет на экран

```
Base 'Class 0' is constructed!
Base 'Class 1' is constructed!
Base 'Field 1 of derived 'Class 1'' is constructed!
Base 'Field 2 of derived 'Class 1'' is constructed!
Derived class 'Class 1' is constructed!
Undefined base class is constructed!
Base 'Field 1 of unknown class' is constructed!
Base 'Field 2 of unknown class' is constructed!
Undefined derived class is constructed!
Derived class '' is destructed!
Base 'Field 2 of unknown class' is destructed!
Base 'Field 1 of unknown class' is destructed!
Base '' is destructed!
Derived class 'Class 1' is destructed!
Base 'Field 2 of derived 'Class 1'' is destructed!
Base 'Field 1 of derived 'Class 1'' is destructed!
Base 'Class 1' is destructed!
Base 'Class 0' is destructed!
```

6 Дополнительная информация

Наследование бывает не только публичным (`public`), но и защищённым (`protected`) или приватным (`private`). Кроме того, наследование бывает ещё и виртуальным (`virtual` используется в комбинации с первыми тремя ключевыми словами). Об этом можно прочитать, например, здесь [3].

Материалы данного занятия подготовлены под впечатлением от курса Яндекса [4]. Автор выражает глубокую благодарность создателям курса.

Список литературы

[1] *Бьерн Страуструп* Язык программирования C++. М:Бином. 2011.

[2] `https://en.cppreference.com`

[3] `https://en.cppreference.com/w/cpp/language/derived_class`

[4] `https://www.coursera.org/specializations/c-plus-plus-modern-development`