

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.

```
1 #include <iostream>
2 #include <string>
3
4 struct AdultTicket {
5     int cost;    // Стоимость билета.
6
7     AdultTicket() :
8         cost(10) // Взрослый билет стоит 10 фунтов.
9     { }
10 };
11
12 struct ChildTicket {
13     int cost;    // Стоимость билета.
14
15     ChildTicket() :
16         cost(5) // Детский билет стоит 5 фунтов.
17     { }
18 };
19
20 /*
21  В C++11 то же самое можно записать короче, а именно
22  struct AdultTicket {
23      int cost = 10;
24  };
25
26  struct ChildTicket {
27      int cost = 5;
28  };
29  Компилятор автоматически создаст соответствующие конструкторы.
30 */
31
```

```
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());
```

```
8 trainConductor.SellTicket(ChildTicket());
9 trainConductor.EmergencyBreak();
10 return 0;
11 }
```

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`.

```
1 #include <iostream>
2 #include <string>
3
4 /* Базовая структура для любого типа билета. */
5 struct Ticket {
6     int cost;           // Стоимость билета.
7     std::string type;  // Тип билета.
8
9     Ticket() :
10         cost(-1),
11         type("unknown")
12     { }
13 };
14
15 /* В C++11 можно написать короче:
16 struct Ticket {
17     int cost = -1;           // Значение по умолчанию для цены билета
18     std::string type = "unknown"; // Значение по умолчанию для типа билета
19 };
20 */
21
22 /* Конструкция ниже говорит о том, что структура AdultTicket является публичным
23 наследником (public) базовой структуры Ticket. При публичном наследовании public
24 члены базовой структуры (класса) становятся public членами производной структуры
25 (класса); protected члены базовой структуры (класса) доступны в производной структуре
26 (классе) как protected, а private члены базовой структуры (класса) недоступны
27 в производной(ом). */
28 struct AdultTicket : public Ticket {
29     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
```

```
81 { } // в порядке их объявления в классе. В данном примере их нет.
82
83 // В отличие от кондуктора автобуса, кондуктор поезда может дёрнуть стоп-кран.
84 void EmergencyBreak() const {
85     std::cout << "Train conductor uses the emergency break!" << std::endl;
86 }
87 };
```

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

```
1 int main() {
2     BusConductor busConductor;
3     TrainConductor trainConductor;
4     Conductor conductor;
5
6     busConductor.SellTicket(AdultTicket());
7     busConductor.SellTicket(ChildTicket());
8     trainConductor.SellTicket(AdultTicket());
9     trainConductor.SellTicket(ChildTicket());
10    trainConductor.EmergencyBreak();
11    conductor.SellTicket(AdultTicket());
12    conductor.SellTicket(ChildTicket());
13    return 0;
14 }
```

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

```
1 // Базовый класс
2 class Base {
3     public:
4     int baseElem;
5 };
6
7 // Производный класс
8 class Derived : public Base {
9     public:
10    int derivedElem;
11 };
12
13 void Function(Base* base, Derived* derived, Base& baseRef, Derived& derivedRef)
14 {
```

```
15  /* Указатели или ссылки на производные классы можно присваивать соответственно
16     указателям или ссылкам на базовые классы без явного приведения типа. Например, */
17  Base* base2 = derived;
18
19  // В получившемся объекте будут доступны все поля и методы базового класса.
20  base2->baseElem = 5; // Корректно, в результате derived->baseElem станет равным 5.
21
22  // Аналогично со ссылками
23  Base& baseRef2 = derivedRef;
24  baseRef2.baseElem = 3; // Корректно, derivedRef->baseElem станет равным 3.
25
26  /* Обратное преобразование требует явного приведения типа, то есть код
27  // Derived* derived2 = base; не скомпилируется. Вместо этого придётся писать */
28  Derived* derived2 = static_cast<Derived*>(base);
29
30  /* static_cast - операция статического приведения типа в C++, она не делает проверку,
31     был ли указатель (ссылка на) base изначально создан(а) как указатель (ссылка)
32     на тип Derived. В угловых скобках пишется тип данных, к которому хотим привести
33     значение, а в круглых - то, что хотим привести.
34     Примечание: проверку иерархии наследования и корректность приведения типа
35     в данном случае делает операция dynamic_cast. */
36  derived2->baseElem = 8; // Корректно, base->baseElem станет равным 8.
37
38  /* Однако следующий код может привести к падению программы, поскольку указатель
39     derived2 мог изначально быть создан как указатель на Base и не содержать
40     поля derivedElem.
41     // derived2->derivedElem = 8; Пример "плохого" кода. */
42
43  /* Аналогично, */
44  Derived& derivedRef2 = static_cast<Derived&>(baseRef);
45  derivedRef2.baseElem = 7; // Корректно, baseRef.baseElem станет равным 7.
46  // derivedRef2.derivedElem = 8; Пример "плохого" кода.
47
48  /* Если забыть поставить указатель или ссылку, то произойдёт полное поэлементное
49     копирование полей, унаследованных из базового класса в новый объект.
50     Для копирования будет использоваться функция operator=() класса Base,
51     созданная компилятором по умолчанию и выполняющая поэлементное копирование.
52     Такая операция называется срезкой (slicing). */
53  Base baseCopy = derivedRef; // Этот объект никак не связан с derivedRef.
54  baseCopy.baseElem = 1; // Никак не отразится на объекте derivedRef.
55  }
```

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:

```
1 void HarvestAndReap(Conductor& conductor, Ticket& ticket)
2 {
```

```
3 ticket.cost += 2;
4 conductor.SellTicket(conductor, ticket);
5 }
```

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`. Вторым способом 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.

```
1 struct Ticket {
2     const int cost;
3     const std::string type;
4
5     Ticket(int newCost = -1, const std::string& newType = "unknown") :
6         cost(newCost), // Задаём константы в списке инициализации конструктора
7         type(newType)
8     { }
9 };
10
11 struct AdultTicket : public Ticket {
12     /* Нам необходимо инициализировать поля cost и type, которые были объявлены в базовом
13        классе. Поскольку они объявлены как константы, то их инициализация возможна только
14        в списке инициализации конструктора. Проблема заключается в том, что в списке
15        инициализации производного класса нельзя явно инициализировать поля базового
16        класса. На помощь приходит специальный конструктор базового класса. */
17     AdultTicket() : // Вызываем в списке инициализации конструктор базовой
18         Ticket(10, "adult") // структуры, который инициализирует её поля.
19     { }
20 };
21
22 struct ChildTicket : public Ticket {
23     ChildTicket() : // Нам нужен специальный конструктор базовой структуры, поэтому
24         Ticket(5, "adult") // его следует вызвать явно.
25     { }
26 };
```

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:

```
1 #include <iostream>
2 #include <string>
3
4 /* Базовый класс. Он выводит информацию в момент вызова своего конструктора и
5 деструктора. */
6 class Base {
7     public:
8     Base() {
9         std::cout << "Undefined base class is constructed!" << std::endl;
10    }
11
12    /* Специальный конструктор умеет задавать id для класса. */
13    Base(const std::string& newId) : // Конструктор
14        id(newId) {
15        std::cout << "Base '" << id << "' is constructed!" << std::endl;
16    }
17
18    ~Base() { // Деструктор
19        std::cout << "Base '" << id << "' is destructed!" << std::endl;
20    }
21    protected:
22        std::string id;
23 };
24
25 class Derived : public Base { // Производный класс
26     public:
27     /* Поскольку стандартный конструктор базового класса вызывается автоматически,
28     то нет нужды вызывать его явно в списке инициализации конструктора производного
29     класса. */
30     Derived() :
31         b1("Field 1 of unknow class"), // Порядок соответствует порядку объявления
32         b2("Field 2 of unknow class") {
33         std::cout << "Undefined derived class is constructed!" << std::endl;
34     }
35
36     /* Здесь необходимо вызвать специальный конструктор базового класса для того, чтобы
37     задать поле id базового класса. Вызов соответствующего конструктора делается
38     в списке инициализации производного класса. Этот вызов должен находиться
39     по порядку раньше всех остальных элементов в списке инициализации. */
40     Derived(const std::string& newId) :
41         Base(newId), // Вызываем конструктор базового класса в списке инициализации
42         b1("Field 1 of derived '" + newId + "'"),
43         b2("Field 2 of derived '" + newId + "'") {
44         std::cout << "Derived class '" << id << "' is constructed!" << std::endl;
45     }
46
47     ~Derived() {
48         std::cout << "Derived class '" << id << "' is destructed!" << std::endl;
49     }
50 }
```



```
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>