Информатика: семестр 3

Программирование на C++

Конспекты лекций

Лектор: Хирьянов Т.Ф.

2 января 2018 г.

Оглавление

Лекция 11

Наследование

11.1 Пример интерфейса – UnivMan

Представим, что у нас есть некий шаблон, описывающий общую схему кода. Пусть это будет, к примеру, *Unversity Man* – для краткости будет звать его **UnivMan**. Им попробуем описать человека из МФТИ. Для нашего интерфейса сделаем некоторые методы: **getName()**, **getPhoto()**, **getDepartment()**. Их тело мы писать не будем; эта работа ляжет на конкретную *реализацию* нашего шаблона.

Составим затем реализацию нашего шаблона: пусть, к примеру, мы захотим описать класс учителя вуза — это класс **Teacher**. Создав этот класс, мы возьмем на себя обязанность реализовать все методы, которые мы прописали в нашем 'шаблоне' **UnivMan**. При этом мы, помимо прочего, можем и дополнить наш 'дочерний' класс своими методами.

11.2 Определение наследования

Такая иерархия классов подводит нас к третьей парадигме ООП: **наследованию**. Наследование — это явление, когда один класс (потомок) перенимает все (строго говоря, почти все; о подробностях см. ниже) поля и методы другого класса (родителя).

Замечание. В нашем примере мы затребовали, чтобы класс **UnivMan** не содержал реализаций никаких функций. Такие классы называются *интерфейсами* – сами по себе эти штуки бесполезны, но они задают некий шаблон, по которому должны строиться его потомки (в примере это были **Teacher** и **Student**).

11.3 Иерархия наследования

Вопрос о том, какой класс сделать родителем, а какой – потомком, требует хорошей продуманности.

Если бы мы хотели в нашем примере добавить класс *старосты*, то его следовало бы сделать наследником класса *студентов* и просто добавить специфичные методы (например, функцию раздачи задавальников). НЕ стоит делать обычного *студента* наследником *старосты*, равно как и не стоит делать *старосту* прямым наследником **UnivMan** и потом по-новой реализовывать те же самые методы, что *студент* уже реализовал.

Хорошим правилом для построения схемы наследования дает критерий Барбары Лисков: "Дочерний класс должен удовлетворять всем интерфейсам родительского класса и не требовать ничего дополнительного."

11.4 Модификаторы доступа

Для полей и методов можно разграничить права доступа: кто сможет их видеть, а кто нет. Для этого существует 3 модификатора доступа:

1. **public** – доступно любым другим функциям (внутри класса, для потомков и остальных)
2. **protected** – доступно любым наследникам и самому классу.
3. **private** – доступно только самому классу.

Код 11.1: Демонстрация модификаторов доступа

```
1  class Base
2  {
3  public:
4      void b_pub() {}
5  private:
6      void b_priv() {}
7  protected:
8      void b_prot() {}
9  };
10
11 class Derivative: public Base // 'public' -- see descr.
12 {
13 public:
14     void d_pub() {
15         // We now have all methods from Base
16         b_pub();     // We are calling OWN b_pub(). It exists
17         b_prot();    // Child has access to protected section
18     }
19 };
20 int main() {
21     Base b;
22     b.b_pub();  // What's okay
```

```
23      Derivative d;
24      d.b_pub();   // OK
25  }
```

Обратите внимание на строку **11** – здесь слово **public** означает, что класс *наследуется* публично. Другие куски кода будут знать, что **Derivative** наследует **Base** и мы сможем через объекты класса **Derivative** доступаться до всего, что есть в **Base**. В таком случае тоже можно делать разные модификаторы:

1. **public** – все, кто имеет доступ к **Child** и **Parent**, будут знать о наследовании.
2. **protected** – только потомок и потомок потомка будут знать о наследовании. Поясним это. Путь наследование происходит по следующей схеме: **Parent** → **Child** → **Child-of-child**. При использовании **protected** в **Child** класс **Child-of-child** и только он будет знать о том, что **Child** унаследовал **Parent**. Потомки более высшего порядка и другие классы не будут знать о наследовании.
3. **private** – никто не будет знать о наследовании, кроме самого **Child**.

Подробнее об этом можно прочитать здесь (осторожно, английский!).

11.5 Override – переопределение методов

В дочернем классе мы можем не просто пользоваться методами родителя, а переопределять их. Это называется *переопределением* метода.

При этом есть и свои тонкости. Рассмотрим их на примере:

Код 11.2: Тонкости вызова переопределенных функций

```
1   class Base
2   {
3   public:
4     void b_pub() {}
5   private:
6     void b_priv() {}
7   protected:
8     void b_prot() {}
9   };
10
11  class Derivative: public Base
12  {
13  public:
14      void d_pub() {
15          b_pub();
16          b_prot();
17      }
```

```
18 };
19
20 void foo(Base& x) {
21     x.b_pub(); // x is considered to be from Base
22     // Equal to:
23     x.Base::b_pub();
24 }
25
26 int main()
27 {
28     Derivative d;
29     foo(d);  // Will call b_pub from Base!
30 }
```

Метод **foo** (строка **20**) 'видит' **x** как объект класса **Base**. И метод **b_pub** он будет тоже вызывать из этого класса, даже если мы **foo** на вход подадим экземпляр класса-наследника.

Заметьте, что функцию мы могли бы и вызывать через обращение к классу как к пространству имен (строка **23**).

11.6 Списки инициализации

С конструкторами бывают проблемы. Рассмотрим пример:

Код 11.3: Тонкости конструкторов

```
1  #include<iostream>
2
3  class Student
4  {
5  public:
6      Student(int x, std::string s)
7      {
8          age = x;
9          name = s;
10     }
11
12     void aging()
13     {
14         age++;
15         std::cout << "I am " << age << " years old\n";
16     }
17 protected:
18     int age;
19     std::string name;
20 };
21
```

```
22 class ElderStudent: public Student
23 {
24 private:
25     int groupID;
26 public:
27     ElderStudent(int x, std::string s, int group_id) : Student(x, s)
28     {
29         groupID = group_id;
30     }
31 };
32
33 int main() {
34     return 0;
35 }
```

На строке **27** конструктор старосты. Ему нужно вызвать конструктор родителя. Почему же?

При создании объекта типа **ElderStudent** мы по факту берем класс **Student** и обертываем его тем, что мы написали в наследнике. Инициализация класса **Student** происходит через *конструктор* класса **Student**. Таким образом, за счет наследования этого класса в **ElderStudent** мы заставляем программу выполнить конструктор родительского класса (**Student**). Но вызывать конструктор напрямую нельзя. Конечно, можно понадеяться, что компилятор сделает все тихо и в неявном виде, но что будет, если наш *староста* в конструкторе будет получать не те переменные, что получал в конструкторе обычный *студент*? А если у *студента* нет конструктора по умолчанию, какой же тогда выбрать? Компилятор в этом случае не разберется и единственный способ разрулить ситуацию – использовать *списки инициализации*. Это сделано после символа ':'.

Магия списка инициализации заключается в том, что значение у нас будет инициализирован **Student** с параметрами **x** и **s** (их мы передали *старосте* в конструктор) еще до входа в само тело конструктора. Теперь у нас все хорошо: мы имеем инициализированного студента и уже готовы его отнаследовать и переопределять (что и сделано уже в *теле* конструктора).

Из вышесказанного вытекает *важный вывод*: при создании объекта будут вызваны конструкторы **всех** родительских классов, начиная от самого старшего в иерархии.

О порядке в списке инициализации

(Т.Ф. не успел этого рассказать на лекции) При использовании списка инициализации можно инициализировать несколько переменных (а не только **Student(x, s)**, как у нас в примере). Важно то, что порядок инициализации не будет зависеть от порядка перечисления через запятую в нашем конструкторе, а будет целиком определяться порядком в исходном классе. На нашем примере этого нельзя увидеть, потому

что мы инициализируем лишь одну переменную, однако это будет заметно в следующем коде:

Код 11.4: Маленький пример к порядку инициализации

```
1 class A {
2 public:
3     int age;
4     int mark;
5     A: age(5), mark(age) {}
6     // Absolutely the same as:
7     A: mark(age), age(5) {}
8 };
```

Порядок инициализации будет одинаков. Сначала в **age** уйдет 5, а затем в **mark** уйдет **age** (то есть 5). Кстати, в строке **5** видим еще один способ присвоить переменную **age** – **age(5)**. Для **age** вызывается конструктор с параметром 5, от этого в **age** уйдет значение 5 (что логично). (Для тех, кто шарит: на примитивах этот трюк тоже работает. В C++ им для таких целей специально добавили конструктор)