Основы разработки на C++: жёлтый пояс

Неделя 5

Наследование и полиморфизм

Оглавление

Наследование и полиморфизм **2**

5.1 Наследование 2

5.1.1 Введение в наследование 2

5.1.2 Доступ к полям классов. Знакомство со списками инициализации 8

5.1.3 Порядок конструирования экземпляров классов 11

5.2 Полиморфизм 15

5.2.1 Унификация работы с классо-специфичным кодом. Постановка проблемы . 15

5.2.2 Решение проблемы с помощью виртуальных методов 18

5.2.3 Свойства виртуальных методов. Абстрактные классы 21

5.2.4 Виртуальные методы и передача объектов по ссылке 23

5.2.5 Хранение объектов разных типов в контейнере с помощью `shared_ptr` . . . 24

5.2.6 Задача о разборе арифметического выражения. Описание решения 25

5.2.7 Решение задачи частного примера 29

5.2.8 Описание и обзор общего решения задачи 30

Наследование и полиморфизм

Наследование

Введение в наследование

Наследование в языке C++ – это способ выразить связи между классами. Например, сказать, что один класс является подтипом другого класса, реализуя часть его функциональности, и может использоваться в связке вместе с ним. Посмотрим, как всё это работает на практическом примере. Мы хотим написать некий волшебный мир, в котором есть животные (кошки и собаки), которые могут есть фрукты – яблоки и апельсины, – а у фруктов есть некоторое количество здоровья. Значит, мы тогда реализуем все эти действия с помощью наших классов, пусть у нас кошки и собаки будут иметь метод «покушать», и когда они едят, будем выводить в консоль, что произошло.

```
#include <iostream>
using namespace std;
struct Apple {  // структура Яблоко
  int health = 10;
};
struct Orange {  // структура Апельсин
  int health = 5;
};
class Cat {  // создаём класс Кошка
public:
  void Meow() const {  // кошка может мяукать
    cout << "meow! ";
  }
  void Eat(const Apple& a) {  // кошка может есть яблоко
    cout << "Cat eats apple. " << a.health << "hp." << endl;
  }
};
```

```
int main() {
  Cat c;  // создали кошку
  c.Meow();  // помяукали
  Apple a;  // создали яблоко
  c.Eat(a);  // съели яблоко кошкой
  return 0;
}
// meow!
// Cat eats apple. 10hp.
```

Всё, кошка съела яблоко, в котором было 10 единиц здоровья.

А теперь попробуем скормить кошке апельсин. По идее она должна уметь есть фрукты, и апельсины тоже.

```
...
  Orange o;  // создали апельсин
  c.Eat(o);  // съели апельсин кошкой
  return 0;
}
// error: no matching function for call to 'Cat::Eat(Orange&)'
```

Появляется ошибка: нет метода «поесть» для апельсина. Определим его, продублировав метод яблока:

```
...// сразу под Meow() {}
  void Eat(const Apple& a) {  // кошка может есть яблоко
    cout << "Cat eats apple. " << a.health << "hp." << endl;
  }
  void Eat(const Orange& o) {  // кошка может есть апельсин
    cout << "Cat eats orange. " << o.health << "hp." << endl;
  }  // продублировали всё, заменив apple на orange
// meow!
// Cat eats orange. 5hp.
```

Всё, апельсин тоже съели. Всё бы ничего, но теперь добавим собаку, скопировав класс `Cat`:

```
class Dog {  // создаём класс Собака дублированием класса Cat с исправлениями
public:  // удалили мяуканье
  void Eat(const Apple& a) {
    cout << "Dog eats apple. " << a.health << "hp." << endl;
```

```
  }
  void Eat(const Orange& o) {
    cout << "Dog eats orange. " << o.health << "hp." << endl;
  }  // продублировали все, заменив apple на orange
};
...
int main() {
  Cat c;  // создали кошку
  c.Meow();
  Apple a;
  Orange o;
  c.Eat(o);
  Dog d;
  d.Eat(a);  // съели яблоко кошкой
  return 0;
}
// meow!
// Cat eats orange. 5hp.
// Dog eats apple. 10hp.
```

Всё работает. Но мы 4 раза продублировали функцию `Eat()`. Заметим, что объекты `Apple` и `Orange` отличаются только числом `hp`, но с ними одинаково работает функция `Eat()`. Если же нам придётся добавить ещё фрукты, то мы будем вынуждены копировать код для этого фрукта и в кошке, и в собаке. Это ужасно.

Создадим **отношение наследования** для классов яблоко и апельсин. Ведь они являются наследниками **базового класса** фрукт (`Fruit`). Яблоко и апельсин – наследники фрукта, и мы явно зададим публичное наследование:

```
struct Fruit {  // структура фрукты
  int health = 0;  // поле здоровья, пусть по умолчанию будет 0 hp
  string type = "fruit";  // добавляем тип того, что мы едим
};

struct Apple : public Fruit {  // указываем, что яблоко наследуется от фруктов
  Apple() {  // нам достаточно написать конструктор для яблока
    health = 10;  // яблоко видит поле health у фрукта, поэтому присваиваем 10
    type = "apple";  // явно указываем, что яблоко имеет тип яблока
  }  // т. к. наследование публичное, мы видим все поля Fruit в Apple
};
```

```
struct Orange : public Fruit {  // без отношения наследования : public Fruit
  Orange() {           // наш код не заработает
    health = 5;
    type = "orange";
  }
};
//
```

Таким образом с помощью публичного наследования мы получили доступ к полям, объявленным в нашем базовом классе. Теперь мы можем исправить наш код следующим образом:

```
class Cat {  // создаём класс Кошка
public:
  void Meow() const {
    cout << "meow! ";
  }
  void Eat(const Fruit& f) {  // кошка ест фрукты
    cout << "Cat eats " << f.type << ". "<< f.health << "hp.";
  }
};
...
int main() {
  Cat c;
  c.Meow();
  Orange o;
  c.Eat(o);
  return 0;
}
// meow! Cat eats orange. 5 hp.
```

Теперь для кошки есть всего один метод `Eat()` для всех фруктов. Аналогично делаем для собаки:

```
class Dog {
public:
  void Eat(const Fruit& f) {  // собака ест фрукты
    cout << "Dog eats " << f.type << ". "<< f.health  << "hp. ";
  }
};
...
int main() {
  Dog d;
```

```
  Orange o;
  d.Eat(o);
  return 0;
}
// Dog eats orange. 5hp.
```

Вот, мы уменьшили дублирование для фруктов. Добавим, например, ананас:

```
struct PineApple : public Fruit {
  PineApple() {
    health = 15;
    type = "papple";
  }
};
...
int main() {
  Dog d;
  PineApple p;
  d.Eat(p);  // теперь собака съест ананас
  return 0;
}
//Dog eats papple. 15hp.
```

Аналогичным образом уберём дублирование из собаки и кошки. На самом деле они должны наследоваться от базового класса Животные. Каждое животное умеет есть фрукты, но не каждое животное может мяукать (это умеет только кошка).

```
class Animal {
public:
  void Eat(const Fruit& f) {  // животное типа type ест фрукты
    cout <<type << " eats " << f.type << ". "<< f.health  << "hp. ";
  }
  string type = "animal";  // уберём дублирование в методе Eat
};
class Cat : public Animal {  // создаём отношение наследования
public:  // т. к. наследование публичное, Cat видит переменную type от Animal
  Cat() {
    type = "cat";
  }
  void Meow() const {
    cout << "meow! ";
```

```
  }  // удалили метод Eat() у кошки. Она животное, а животные умеют есть
};
```

С помощью публичного наследования мы получили доступ к методам и полям класса «животное». И поэтому теперь кошка может вызывать метод `Eat`, хотя этот метод не определён внутри «кошки», а определён внутри класса, от которого отнаследован класс «кошка». И теперь точно так же мы уберем дублирование для класса «собака».

```
class Dog : public Animal {
public:
  Dog() {
    type = "dog";
  }
};
```

Всё продолжает работать. Но теперь и собака и кошка могут есть, хотя в их описании это явно не указано. Кроме того, кошка может мяукать, а собака – нет.

Чтобы показать всю силу наследования, давайте создадим функцию «покормить», `DoMeal`. В неё мы будем передавать некое животное, некоторый фрукт, при этом сама функция не будет знать, что это за конкретное животное и что это за конкретный фрукт. Она просто вызовет метод «поесть» для этого фрукта. И, по идее, вывод в консоль измениться не должен.

```
...
void DoMeal(Animal& a, Fruit& f) {
  a.Eat(f);
}
int main() {
  Dog d;
  Cat c;
  Orange o;
  Apple a;
  DoMeal(d, a);  // эта функция ничего не знает ни о собаках, ни о кошках
  DoMeal(c, o);  // она знает только о классах базового типа
  return 0;
}
// dog eats apple. 10hp.
// cat eats orange. 5hp.
```

Замечу, что сейчас в классе Animal у нас строчка типа `type` доступна всем. То есть снаружи эту строчку тоже можно менять, когда вы используете данный класс. И, на самом деле, в какой-

то момент по ошибке вы можете внести какие-то изменения, которые вносить не хотелось бы. Например, после того как мы покормим собаку, добавлять к ней звёздочку.

```
void DoMeal(Animal& a, Fruit& f) {
  a.Eat(f);
  a.type += "*";
}...
  DoMeal(d, a);
  DoMeal(d, o);
  DoMeal(d, o);
// dog eats apple. 10hp.
// dog* eats apple. 10hp.
// dog** eats apple. 10hp.
```

О том, как этого избежать, в следующем видео.

Доступ к полям классов. Знакомство со списками инициализации

Зачем нужно наследование?

- Введение логической иерархии между классами (`Cat, Dog` ∈ `Animal`);
- Решение проблемы дублирования между однотипными структурами;
- Универсальные функции, работающие с базовыми классами-родителями (`DoMeal`).

Отношение публичного наследования объявляется так:

```
class TDerived : public TBase {...}
```

При таком публичном наследовании у класса-наследника появится в том числе доступ к полям и методам базового класса, от которого он унаследован.

Перейдём к проблеме непреднамеренной модификации полей классов в наших программах. Напомню проблему: у нас была функция «покормить», в неё мы добавляли звёздочку к типу, в результате двух кормёжек переменная `type` внутри объекта класса «собаки» менялась. Рассмотрим, как от этого можно защититься.

```
void DoMeal(const Animal& a, Fruit& f) {
  a.Eat(f);  // передаём константный тип, но тогда мы не сможем его изменять
  a.type += "*";
}  // этот способ не используем из-за его локальности
```

Способ 1: можно сделать переменную в базовом классе защищённой (написав ключевое слово `protected`). В данном случае, когда мы пишем ключевое слово `protected`, никто извне не может обратиться к переменной типа `type`, в данном случае, у класса `animal`, у объектов класса `animal` в рантайме. Но при этом к объекту типа `type` имеют доступ наследники класса `animal`, то есть, в данном случае, когда мы будем создавать экземпляр класса «кошки», когда мы будем создавать этот объект в его конструкторе, будет доступ в переменную `type`.

Скомпилируем:

```
class Animal {
public:
  void Eat(const Fruit& f) {  // животное типа type ест фрукты
    cout <<type << " eats " << f.type << ". "<< f.health  << "hp. ";
  }
protected:  // все переменные и методы ниже будут защищены (не видны снаружи)
  string type = "animal";  // уберём дублирование в методе Eat
};
```

И теперь компилятор будет жаловаться, пока мы не уберём изменение поля `type` в `DoMeal`. Заметим, что наследники могут обращаться к полям и менять их, но уже в себе.

Способ 2: объявление поля константным внутри базового класса. Это тоже хорошее, рабочее решение. Но далее мы видим, что где-то почему-то убирается константность. То есть читаем ошибку, видим, что в данном случае мы пытаемся записать новое значение в константную переменную. Но мы ведь сейчас находимся в конструкторе. Почему так происходит? Здесь надо сказать одну очень важную вещь: в языке C++ все объекты всегда имеют две стадии: либо объект сейчас создается, вот в данный текущий момент происходит его создание, он только-только появляется в памяти, его поля инициализируются, но он ещё до конца не создан. А вторая стадия – это момент, когда уже объект создан, то есть под него выделена память, там всё проинициализировано, как надо, и мы с ним работаем.

```
class Animal {
public:
  void Eat(const Fruit& f) {
    cout << type << " eats " << f.type << ". " << f.health  << "hp. ";
  }
```

```
  const string type = "animal";
};
class Cat : public Animal {
public:
  Cat() {  // на этом этапе переменные внутри класса уже проинициализированы
    type = "cat";  // тут ошибка из-за константности поля
  }
  void Meow() const {
    cout << "meow! ";
  }
};
...  // закомментируем все последующие обращения к type и увидим:
// animal eats apple. 10hp.
```

Видим, что изначально `type` в `Cat` инициализируется значением `animal`, а мы потом пытаемся его переприсвоить на `Cat`. Создадим конструктор класса животное:

```
class Animal {
public:
  Animal(const string& t = "animal") {
    type = t;  // опять пытаемся в константную строку что-то записывать
  }
  void Eat(const Fruit& f) {
    cout << type << " eats " << f.type << ". " << f.health << "hp. ";
  }
  const string type;
};
// passing 'const string'....
```

Видим, что `type` проинициализирована ещё до фигурных скобок. Воспользуемся синтаксисом списков инициализации:

```
class Animal {
public:
  Animal(const string& t = "animal")
  : type(t) {  // хотим проинициализировать type значением t
  }
  void Eat(const Fruit& f) {
    cout << type << " eats " << f.type << ". " << f.health << "hp. ";
  }
  const string type;
```

```
};
// animal eats apple. 10hp.
```

Мы смогли проинициализировать переменную `type` нужным нам значением `t` с помощью синтаксиса списков инициализации. Теперь нам надо из класса-наследника как-то передать свой собственный тип. Для этого нам надо вызвать конструктор базового класса, в данном случае `animal`, с другим аргументом.

```
class Cat : public Animal {
public:
  Cat() : Animal("cat") {  // передаём в Animal нужное нам значение
  }  // которое запишется в константный type
  void Meow() const {
    cout << "meow! ";
  }
};  // далее при любой попытке модификации type, компилятор нам не позволит это сделать
// cat eats apple. 10hp.
```

Таким образом, мы можем защитить поля классов двумя способами:

1. Объявить поле в секции `protected`. Тогда к нему доступ будут иметь только наследники;
2. Объявить поле константным. Его нужно будет проинициализировать при создании объекта и его нельзя будет менять на протяжении жизни.

Синтаксис списков инициализации выглядит так:

```
MyClass(int v1, int v2)  // в конструкторе класса после объявления аргументов
  : Var1(v1)  // пишем инициализируемые поля классов и значения, которые запишем
  , Var2(v2)
{
  ...
}
```

Порядок конструирования экземпляров классов

Будем работать с упрощённой версией кода:

```cpp
#include <iostream>

using namespace std;

struct Log {  // структура данных логер
  Log(string name) : n(name) {
    cout << "+ " << n << endl;  // пишет + и свой id, когда создаётся
  };
  ~Log() {  // деструктор
    cout << "- " << n << endl;  // пишет - и свой id, когда удаляется
  }
  string n;
};

struct Fruit {
  string type = "fruit";
  Log l = Log("Fruit");  // заведём логер для фруктов
};

struct Apple : public Fruit {
  Apple() {
    type = "apple";
  }
  Log l = Log("Apple");  // теперь логер для яблока
};

int main() {
  Apple a;
  return 0;
}

// + Fruit
// + Apple
// - Apple
// - Fruit
```

Видим, что логер был создан сначала внутри `Fruit`, затем внутри яблока, затем удалён внутри яблока и только потом внутри `Fruit`. Это может говорить о том, что если у нас есть некоторая иерархия классов, то при создании объекта, принадлежащего данной иерархии, всегда будет сначала вызываться конструктор базового класса, затем класса-наследника и так далее по це-

почке. При этом, когда объекты данных классов будут удаляться, сначала всегда вызывается объект самого последнего класса наследника и так далее по цепочке вверх, пока мы не дойдем до деструктора самого базового класса.

Теперь зарефрактором наш код:

```
struct Fruit {
  Fruit(const string& t)
    :l(t + " (Fruit)") {}  // инициализируем логер строкой + именем базового класса
  const Log l;  // сделали логер константным
};
// т. к. яблоко унаследовано от Fruit, то перед Apple вызывается конструктор Fruit
struct Apple : public Fruit {
  Apple() : Fruit("apple") {
  }
  Log l = Log("Apple");  // теперь логер для яблока
};
// + apple (Fruit)
// + Apple
// - Apple
// - apple (Fruit)
```

Теперь сделаем яблокам уникальные идентификаторы:

```
struct Apple : public Fruit {
  Apple(const string& t) : Fruit(t), l(t) {  // добавили индентификатор и логер
  }
  const Log l;
};

int main() {
  Apple a1 = Apple("a1");
  Apple a2 = Apple("a2");
  return 0;
}
// + a1 (Fruit) // + a1 // + a2 (Fruit) // + a2
// - a2 // - a2 (Fruit) // - a1 // - a1 (Fruit)
```

Значит, сначала создали объект `a1`, потом объект `a2`. Когда для них начали вызываться деструкторы, произошло все ровно в обратном порядке. Значит, когда мы создаём переменные обычные, конструирование и удаление объектов работает точно так же. **Кто создаётся раньше, тот уда-**

ляется позже. Создадим более сложный класс Яблочного дерева, который будет содержать в себе много яблок:

```
class AppleTree {
public:
  AppleTree()
    : a1("a1")
    , a2("a2") {
  }
  Log l = Log("AppleTree");
  Apple a1;
  Apple a2;
}

int main() {
  AppleTree at;
  return 0;
}

// + AppleTree
// + a1 (Fruit) // + a1 // + a2 (Fruit) // + a2
// - a2 // - a2 (Fruit) // - a1 // - a1 (Fruit)
// - AppleTrees
```

Вопрос: от чего зависит порядок инициализации? Даже поменяв строчки местами:

```
: a2("a2")
, a1("a1") {  // выскочит пара warning-ов
```

Всё равно порядок создания не изменится. Список инициализации не меняет порядок создания. Он просто указывает значения, которыми мы их инициализируем. Реально же порядок зависит только от того, как они перечислены внутри класса.

Хотя это всего лишь предупреждение, инициализация не в том порядке может нам помешать:

```
class AppleTree {
public:
  AppleTree(const string& t)
    : type(t)  // инициализируем переменную type первой
    , a1(type + "a1")  // и тут её активно используем
    , a2(type + "a2") {
  }
```

```
  Apple a1;
  Apple a2;
  string type;  // а здесь эта переменная объявлена последней
};

int main() {
  AppleTree at("AppleTree");
  return 0;
}
// terminate called after throwing an instance of std::bad_alloc
```

Видим ошибки или какой-то бред. Поскольку мы инициализировали `type` последним и использовали его перед этим, мы обратились к неизвестной памяти. Если изменить порядок, то всё будет работать.

Полиморфизм

Унификация работы с классо-специфичным кодом. Постановка проблемы

Наследование нам помогало с дедупликацией повторяющегося кода в различных классах. Соответственно, дедупликацию мы исключали за счёт создания базового класса и вынесения общего кода в методы этого класса. Восстановим код из лекции:

```
#include <iostream>
using namespace std;

class Animal {
public:
  Animal(const string& type) : type_(type) {}
  void Eat(const string& fruit) {
    cout << type_ << " eats " << fruit << endl;
  }
private:
  const string type_;
};
```

```
class Cat : public Animal {
public:
  Cat() : Animal("cat") {}
  void Meow() const {
    cout << "Meow!" << endl;
  }
};

class Dog : public Animal {
public:
  Dog() : Animal("dog") {}
  void Bark() const {
    cout << "Whaf!" << endl;
  }
};

int main() {
  Cat c;
  Dog d;
  c.Eat("apple");
  d.Eat("orange");
  return 0;
}
// cat eats apple
// dog eats orange
```

Мы научились делать общий код для разных классов. Давайте теперь рассмотрим не общий код, а специфичный для каждого класса. Давайте напишем функции, которые позволяют нам единообразно просить животных издавать разные звуки. Соответственно, самый простой вариант – это написать отдельную функцию для кошки и для собаки:

```
void MakeSound(const Cat& c) {
  c.Meow();
}
void MakeSound(const Dog& d) {
  d.Bark();
}
int main() {
  ...
  MakeSound(c);
  MakeSound(d);
```

```
}
// Meow!
// Whaf!
```

Снова возникает проблема:

- Если мы имеем X различных классов: `Cat`, `Dog`, `Horse`, ...
- И у нас Y методов для каждого класса: голос, действие, ...
- Получим в итоге $X * Y$ функций!

Что же можно сделать? Первая же идея, которая приходит нам в голову – это воспользоваться методикой из предыдущей лекции. Давайте мы объединим функцию `MakeSound` и унесём её в базовый класс.

```
class Animal {
public:
  Animal(const string& type) : type_(type) {}
  void Eat(const string& fruit) {
    cout << type_ << " eats " << fruit << endl;
  }
  void Voice() const {  // приходится делать громоздкую конструкцию,
    if (type_ == "cat") {  // поскольку animal не знает про котов
      cout << "Meow!" << endl;
    } else if (type_ == "dog") {
      cout << "Whaf!" << endl;
    }
  }
private:
  const string type_;
};  // удаляем функции Bark и Meow
...
void MakeSound(const Animal& a) {
  a.Voice();
}

int main() { ...
  MakeSound(c);
  MakeSound(d);
}
```

Вроде исправили, но остались минусы, и основной заключается в том, что различное поведение классов-потомков необходимо переносить в базовый класс. Дополним класс животных, добавив попугая:

```
...
  } else if (type_ == "dog") {
    cout << "Whaf!" << endl;
  } else if (type == "parrot") {  // попугай называет себя (по имени) хорошим
    cout << name_ << " is good!" << endl;  // попытаемся сделать это тут
  }
...
class Parrot : public Animal {
public:
  Parrot(const string& name) : Animal("parrot"), name_(name) {}
private:
  const string& name_;  // имя попугая делаем приватным
};
```

Ловим ошибку: в базовом классе неизвестно приватное поле `name`. Таким образом мы не можем перенести функцию `Voice` из класса `Parrot` в родительский класс `Animal`.

Решение проблемы с помощью виртуальных методов

У нас не получилось объединить различные специфичные методы разных классов в рамках одного метода в базовом классе (мы не смогли использовать приватное поле одного из классов).

Вынесем логику определённых классов обратно в эти классы:

```
#include <iostream>
using namespace std;

class Animal {
public:
  Animal(const string& type) : type_(type) {}
  void Eat(const string& fruit) {
    cout << type_ << " eats " << fruit << endl;
  }
private:
  const string type_;
```

```
};

class Cat : public Animal {
public:
  Cat() : Animal("cat") {}
  void Voice() const {
    cout << "Meow!" << endl;
  }
};

class Dog : public Animal {
public:
  Dog() : Animal("dog") {}
  void Voice() const {
    cout << "Whaf!" << endl;
  }
};

class Parrot : public Animal {
public:
  Parrot(const string& name) : Animal("parrot"), name_(name) {}
  void Voice() const {
    cout << name_ << " is good!" << endl;
  }
private:
  const string& name_;
};

void MakeSound(const Animal& a) {
  a.Voice();
}

int main() {
  Cat c;
  Dog d;
  MakeSound(c);
  MakeSound(d);
  return 0;
}
// error: 'const class Animal' has no member named 'Voice'
```

Для избавления от ошибки просто объявим этот метод в `animal`, но там он ничего не будет делать:

```
class Animal {
public:
  Animal(const string& type) : type_(type) {}
  void Eat(const string& fruit) {
    cout << type_ << " eats " << fruit << endl;
  }
  void Voice() const {};  // вот тут
private:
  const string type_;
};
//
```

Всё корректно скомпилировалось, но никто не издаёт звуков. Нам нужно, чтобы базовый класс узнал о методах в классах-потомках. Это делается добавлением ключевого слова **`virtual`** к методу в базовом классе.

```
  ...
  virtual void Voice() const {};  // в Animal
// Meow!
// Whaf!
```

То есть мы сказали базовому классу, что в классах-потомках может быть своя реализация метода `Voice` с той же сигнатурой (возвращаемый тип и константность должны совпадать). И теперь, когда мы вызываем метод `Voice()` у базового класса, то будет вызываться этот метод у нужного класса-потомка.

Теперь добавим попугая:

```
int main() {
  Parrot p("Kesha");
  MakeSound(p);
  return 0;
}
// Kesha is good
```

Всё работает и для попугая. Теперь допустим, что мы захотели изменить название `Voice()` в `Animal` на `Sound()`.

```
  virtual void Sound() const {};  // в Animal
```

```
// error: 'const class Animal' has no member named 'Voice'
```

Ошибка возникает в функции `MakeSound()`. Поменяем название и здесь:

```
void MakeSound(const Animal& a) {
  a.Sound();
}
//
```

Программа собирается, но ничего не выводит. Это происходит, потому что в потомках остался метод `Voice()`, но базовый класс ничего про него не знает.

Вернём обратно название `Voice()` везде. И в каждом потомке напишем:

```
  void Voice() const override {
    ...
  }
...
```

Всё снова работает, как мы ожидаем. Если же мы сейчас повторим те же действия по переименованию в `Sound()` в базовом классе и в `MakeSound()`. Теперь нам выдаётся ошибка:

```
// ... marked 'override', but does not override
```

Компилятор подсказывает, что в базовом классе метод `Voice` неизвестен, и в этом случае мы не совершим ошибки по случайному переименованию функций.

Таким образом, мы узнали про два новых ключевых слова:

- **`virtual`** добавляется к методам в базовом классе. Позволяет вызывать методы производных классов через ссылку на базовый класс;
- **`override`** добавляется к методам в производных классах (классах-потомках). Требует объявления метода в базовом классе с такой же сигнатурой.

Свойства виртуальных методов. Абстрактные классы

Методы, помеченные словом **`virtual`**, называются **виртуальными**. Рассмотрим их свойства. Добавим к получившемуся после наших изменений коду класс Лошадь, в котором не будем

объявлять метод `Sound()`:

```
class Horse : public Animal {
public:
  Horse() : Animal("horse") {}
};
...
int main() { ...
  Horse h;
  MakeSound(h);
  return 0;
}
//
```

Для лошади ничего не вывелось. Посмотрим, какой метод `Sound()` вызывается для лошади. Изменим в `Animal` метод `Sound`:

```
virtual void Sound() const {
  cout << type_ << " is silent " << endl;  // по умолчанию будет говорить так
}
// horse is silent
```

Таким образом, в данном отношении виртуальные методы не отличаются от обычных: если метод не объявлен в классе-потомке, то он будет вызван из базового класса.

Рассмотрим другое полезное свойство виртуальных методов: допустим, мы хотим, чтобы у каждого потомка был по-своему реализован метод `Sound()`. Пока под это условие не подходит `Horse`. Будем считать, что мы просто забыли его реализовать и хотим, чтобы компилятор подсказывал нам, что нам надо реализовать. Потребуем обязательной реализации в классах-потомках:

```
class Animal {
public:
  Animal(const string& type) : type_(type) {}
  void Eat(const string& fruit) {
    cout << type_ << " eats " << fruit << endl;
  }
  virtual void Sound() const = 0;  // это не присваивание, а формальный синтаксис
private:
  const string type_;
};
// cannot declare variable 'h' to be of abstract type 'Horse'
```

Мы сказали виртуальному методу `Sound()`, что он является абстрактным (чисто виртуальным), т.е. требует обязательной реализации в каждом классе-потомке. Класс `Horse` считается абстрактным, потому что в нем нет реализации абстрактного метода `Sound()`, и мы не можем создавать объекты абстрактных классов.

Виртуальные методы и передача объектов по ссылке

Заметим, что мы передавали объекты классов-потомков по ссылке на базовый класс.

Попробуем передать по значению:

```
void MakeSound(const Animal a) {  // теперь без &
  a.Voice();
}
```

Программа не собралась, потому что компилятор считает, что мы хотим преобразовать этот объект в тип базового класса. Но мы не можем создавать объекты базового класса, потому что он является абстрактным (из-за виртуального метода `Voice()` в базовом классе). Уберём абстрактность метода `Voice()`:

```
class Animal {
public:
  Animal(const string& type) : type_(type) {}
  void Eat(const string& fruit) {
    cout << type_ << " eats " << fruit << endl;
  }
  virtual void Voice() const  {
    cout << type_ << " is silent" << endl;
  }
  ...
// cat is silent
```

Видим, что при передаче по значению вызываются методы базового класса, а не класса-потомка. При передаче по значению мы теряем всю информацию о классе-потомке, и у нас сохраняется информация только о базовом классе, в котором `Voice()` печатает `is silent`. То есть при передаче по значению мы теряем преимущества виртуальных методов.

Хранение объектов разных типов в контейнере с помощью shared_ptr

Вернёмся к функции main():

```
int main() {
  Cat c;
  Dog d;
  Parrot p("Kesha");
  Horse h;

  MakeSound(c);
  MakeSound(d);
  MakeSound(p);  // 4 очень похожих вызова
  MakeSound(h);
  return 0;
}
```

Хочется сделать вызовы универсальными с помощью контейнеров. Мы могли бы сложить всех животных в контейнер и, например, в цикле for пробегаться по ним. Все объекты мы можем объединить ссылкой на базовый класс:

```
#include <memory>  // заголовок с shared_ptr
...
int main() {
  // vector<Animals&> animals; вектор из ссылок создать не получится
  // но есть другой тип из базовых библиотек, с помощью которого можно это сделать:
  shared_ptr<Animal> a;

  a = make_shared<Cat>();  // в угловых скобках указываем реальный тип
  a->Sound();  // похожим на итераторы образом вызываем метод Sound()
  return 0;
}
// Meow!
```

Видим, что вызвался метод Sound для Cat.

Самое полезное свойство shared_ptr, которое мы использовали сейчас: в качестве типа объекта, в который оборачивается shared_ptr можем указать просто Animal, и при этом shared_ptr будет вести себя как ссылка (мы сложим в него объекты производных классов и сможем обращаться к интерфейсу базового класса). Второе свойство: у shared_ptr интерфейс, похожий на итераторы (с ними просто работать).

```
int main() {
  shared_ptr<Animal> a;
  a = make_shared<Cat>();
  vector<shared_ptr<Animal>> animals;  // векторы животных из shared_ptr
  for (auto a : animals) {  // передаем по значению, а можно и const auto& a
    MakeSound(*a);  // получаем объект, который обёрнут в shared_ptr
  }
  return 0;
}
// Meow!
```

Всё работает, теперь сделаем то же для вектора животных.

```
int main() {
  shared_ptr<Animal> a;
  a = make_shared<Cat>();
  vector<shared_ptr><Animal>> animals = {
    make_shared<Cat>(),
    make_shared<Dog>(),
    make_shared<Parrot>("Kesha"),  // имя передаётся в конструктор попугая
  }
  for (auto a : animals) {
    MakeSound(*a);
  }
  return 0;
}
// Meow! Whaf! Kesha is good!
```

С помощью `shared_ptr` мы можем помещать различные типы объектов производных классов в контейнеры и обходить их с помощью циклов, как и обычные типы.

Задача о разборе арифметического выражения. Описание решения

Поставим задачу написать парсер арифметических выражений:

- Входные данные: Выражение, состоящее из операций `+`, `-`, `*`, цифр и переменной `x`. Её значение нам дано;

- Выходные данные: значение выражения при введённом значении переменной.

Пример работы:

```
// Enter expression:
// 1+2-3*x+x
// Enter x:
// 5
// Expression value -7. Enter x:
// 1
// Expression value 1...
```

Теперь попробуем написать наш разбиратель:

```
#include <iostream>
#include <memory>
#include <stack>
#include <vector>

using namespace std;

int main() {
  string tokens;
  cout << "Enter expression: ";
  cin >> tokens;  // запрашиваем у пользователя выражение
  int x = 0;
  auto expr = Parse(tokens, x);  // функция разбора выражения для всех x
  cout << "Enter x: ";
  while (cin >> x) {  // пока пользователь будет вводить значения, вычисляем
    cout << "Expression value: " << expr->Evaluate() << endl;
    cout << "Enter x: ";
  }
  return 0;
}
```

Теперь реализуем функции `Parse` и `Evaluate`. Мы пока не знаем ничего о типе переменной `expr`.

```
struct Node {  // тип выводимых значений
public:
  int Evaluate() {
    return 0;
```

```
  }
};
Node Parse(const string& tokens, int& x)  // должна возвращать объект типа Node
  // у которого есть метод Evaluate, вычисляющий выражение
```

Чтобы понять, как разбирать арифметическое выражение, сделаем это вручную на примере:

```
int main() {
  string tokens;
  string tokens = "5+7-x*x+x";  // -x^2 + x + 12
  cout << "Enter x: ";
  while (cin >> x) {  // пока пользователь будет вводить значения, вычисляем
    cout << "Expression value: " << expr->Evaluate() << endl;
    cout << "Enter x: ";
  }
  return 0;
}
```

Для простоты разбора будем использовать только числа из одной цифры, а также не будет скобок, пробелов и ненужных символов. И будем считать, что выражение всегда корректно. Разделим выражение на типы ввода:

```
struct Node {  // тип выводимых значений -- это базовый класс для переменных и цифр
public:
  virtual int Evaluate() {
    return 0;
  }  // сделали метод виртуальным
};  // т. к. для каждого класса-потомка надо описывать, что делать в Evaluate

class Digit : public Node {
public:  // класс цифра, который будет всегда возвращать значение самой цифры
  Digit(int d) : d_(d) {
  }
  int Evaluate() override {
    return d_;
  }
private:
  const int d_;
}

class Variable : public Node {  // переменная x
```

```
public:
  Variable(const int& x) : x_(x) {
  }  // сохраняем константную ссылку
  int Evaluate() override {  // при вычислении x возвращаем его значение
    return x_;
  }
private:
  const int& x_;
}  // остались операции. Заметим, что у них разный приоритет. Умножение раньше всего

class Operation {  // для сохранения операции надо сохранять левый и правый операнды
public:  // передаём символ операции, левое и правое слагаемое
  Operation(char op, shared_ptr<Node> left, shared_ptr<Node> right) :
    op_(op), left_(left), right_(right) {
  }
  int Evaluate() {
    // вычисление операций оставим на потом
  }
private:
  const char op_;
  shared_ptr<Node> left_, right_;
};
```

Заметим, что в выражении `5 + 7 * 3` для операции `+` левое слагаемое – это число 5, а правое – это результат умножения 7 и 3. При вычислении значения операции нужно сначала вычислить оба слагаемых, а потом произвести саму операцию и вернуть значение.

```
class Operation : public Node {  // унаследуем от Node
public:
  Operation(char op, shared_ptr<Node> left, shared_ptr<Node> right) :
    op_(op), left_(left), right_(right) {
  }
  int Evaluate() override {  // операция унаследована от Node
    if (_op == '*') {
      return _left->Evaluate() * _right->Evaluate();
    } else if (_op == '+') {
      return _left->Evaluate() + _right->Evaluate();
    } else if (_op == '-') {
      return _left->Evaluate() - _right->Evaluate();
    }
    return 0;  // если операция не распознана, но для корректной строки не дойдём
```

```
  }
...
```

Решение задачи частного примера

Вспомним, как у нас вообще вычисляются арифметические выражения. Во-первых, они вычисляются слева направо для операций с равным приоритетом. Операции с большим приоритетом вычисляются в первую очередь. То есть для нашего выражения сначала мы должны вычислить операцию `x * x`. Затем у нас остаются три равноправные операции: это сложение, вычитание и сложение. (в выражении `5 + 7 - x * x + x`). И мы должны их вычислить слева направо. Опишем множители:

```
int main() {
  string tokens = "5+7-x*x+x";
  shared_ptr<Node> var1 = make_shared<Variable>(x);  // делаем операцию умножения
  shared_ptr<Node> var2 = make_shared<Variable>(x);
  shared_ptr<Node> mul1 = make_shared<Operation>('*', var1, var2);
  shared_ptr<Node> dig1 = make_shared<Digit>(5);  // делаем операцию сложения
  shared_ptr<Node> plus1 = make_shared<Operation>('+', dig1, dig2);
  shared_ptr<Node> dig2 = make_shared<Digit>(7);
  shared_ptr<Node> var3 = make_shared<Variable>(x);
  shared_ptr<Node> minus1 = make_shared<Operation>('-', plus1, mul1);
  shared_ptr<Node> plus2 =  make_shared<Operation>('+', minus1, var3);
  // итоговый результат будет в самом правом и самом неприоритетном операторе, plus2
  cout << "Enter x: ";
  while (cin >> x) {  // пока пользователь будет вводить значения, вычисляем
    cout << plus2->Evaluate() << endl;
    cout << "Enter x: ";
  }
  return 0;
}
```

При значении `x = 1` выведет `12`. Это правда. При `x = 5` получим: `-8` и т. д.

Описание и обзор общего решения задачи

Решим исходную задачу: выражение мы должны, на самом деле, вводить также с консоли – оно должно быть произвольное – с точностью до тех условий, которые мы указали ранее. И мы должны его уметь преобразовывать в нужный нам формат не вручную, а с помощью алгоритма, который бы эффективно по нему проходился и сам составлял необходимый набор объектов, переменных, операций.

Возьмем заранее заготовленный код (находится в материалах к видеолекции) и разберём его:

```
#include <iostream>
#include <memory>
#include <stack>
#include <vector>
using namespace std;
// структура классов у нас остаётся той же
struct Node {  // базовый класс
  virtual int Evaluate() const = 0;
};

struct Value : public Node {  // класс для значения. Ранее был Digit
  Value(char digit) : _value(digit - '0') {
  }
  int Evaluate() const override {
    return _value;
  }
  private:
    const uint8_t _value;
};

struct Variable : public Node {  // класс для переменной x
  Variable(const int &x) : _x(x) {
  }
  int Evaluate() const override {
    return _x;
  }
  private:
    const int &_x;
};
// алгоритм называется shunting-yard, алгоритм сортировочных станций.
struct Op : public Node {  // класс для операций
```

```
  Op(char value)
    : precedence([value] {
    if (value == '*') {
      return 2;
    } else {
      return 1;
    }
  }()),
  _op(value) {
  }
  const uint8_t precedence;
  int Evaluate() const override {
    if (_op == '*') {
      return _left->Evaluate() * _right->Evaluate();
    } else if (_op == '+') {
      return _left->Evaluate() + _right->Evaluate();
    } else if (_op == '-') {
      return _left->Evaluate() - _right->Evaluate();
    }
    return 0;
  }
// передаём левый и правый операнды отдельными методами. удобнее для произвольного
  void SetLeft(shared_ptr<Node> node) {
    _left = node;
  }
  void SetRight(shared_ptr<Node> node) {
    _right = node;
  }
  private:
    const char _op;
    shared_ptr<const Node> _left, _right;
};

template <class Iterator>
shared_ptr<Node> Parse(Iterator token, Iterator end, const int &x) {
  // функцию Parse сделали шаблонной, т. е. для любых контейнеров из символов
  if (token == end) {
    return make_shared<Value>('0');
  }
  stack<shared_ptr<Node>> values;
  stack<shared_ptr<Op>> ops;
```

```
  auto PopOps = [&](int precedence) {  // реализация алгоритма сортировочных станций
    while (!ops.empty() && ops.top()->precedence >= precedence) {
      auto value1 = values.top();
      values.pop();
      auto value2 = values.top();
      values.pop();
      auto op = ops.top();
      ops.pop();
      op->SetRight(value1);
      op->SetLeft(value2);
      values.push(op);
    }
  };
  while (token != end) {
    const auto &value = *token;
    if (value >= '0' && value <= '9') {
      values.push(make_shared<Value>(value));
    } else if (value == 'x') {
      values.push(make_shared<Variable>(x));
    } else if (value == '*') {
      PopOps(2);
      ops.push(make_shared<Op>(value));
    } else if (value == '+' || value == '-') {
      PopOps(1);
      ops.push(make_shared<Op>(value));
    }
    ++token;
  }
  while (!ops.empty()) {
    PopOps(0);
  }
  return values.top();
}

int main() {
  string tokens;
  cout << "Enter expression: ";
  getline(cin, tokens);  // считываем выражение
  int x = 0;
  auto node = Parse(tokens.begin(), tokens.end(), x);  // парсим выражение
    // по итераторам начала и конца строки
```

```
  cout << "Enter x: ";
  while (cin >> x) {
    cout << "Expression value: " << node->Evaluate() << endl;
    cout << "Enter x: ";
  }
  return 0;
}
```

Введём различные переменные `x` для выражения `5 + 7 - x * x + x` и получим: для `x = 0` ответ `12`, для `1` – `12`, для `2` – `10`. Всё верно. Введём другое выражение и убедимся в корректности. Кроме того, `x` может быть отрицательным.

Таким образом, с помощью полученных нами знаний о типе `shared_ptr` и виртуальных методах мы смогли реализовать довольно непростой алгоритм и решить относительно сложную задачу по разбору выражений и вычислению их с использованием внешней переменной.