ООП: Наследование и полиморфизм

Максим Бакиров
C++ - Разработчик в Яндекс

Проверка связи

Поставьте “+”, если меня видно и слышно

Если у вас нет звука:

- убедитесь, что на вашем устройстве и на колонках включен звук
- обновите страницу вебинара (или закройте страницу и заново присоединитесь к вебинару)
- откройте вебинар в другом браузере
- перезагрузите компьютер (ноутбук) и заново попытайтесь зайти

Максим Бакиров

О спикере:

- В C++ разработке с 2017 года
- С 2019 года работает в команде разработки Яндекс Браузера

Вспоминаем прошлое занятие

Вопрос: что такое ООП?

Вспоминаем прошлое занятие

Вопрос: что такое ООП?

Ответ: ООП - это объектно-ориентированное программирование

Вспоминаем прошлое занятие

Вопрос: что является краеугольным камнем в ООП?

Вспоминаем прошлое занятие

Вопрос: что является краеугольным камнем в ООП?

Ответ: классы и объекты - экземпляры классов

Вспоминаем прошлое занятие

Вопрос: о чём нам говорит принцип абстракции?

Вспоминаем прошлое занятие

Вопрос: о чём нам говорит принцип абстракции?

Ответ: о том, что при проектировании классов нам нужно выделять только те поля и методы, которые нужны для решения задачи

Вспоминаем прошлое занятие

Вопрос: о чём нам говорит принцип инкапсуляции?

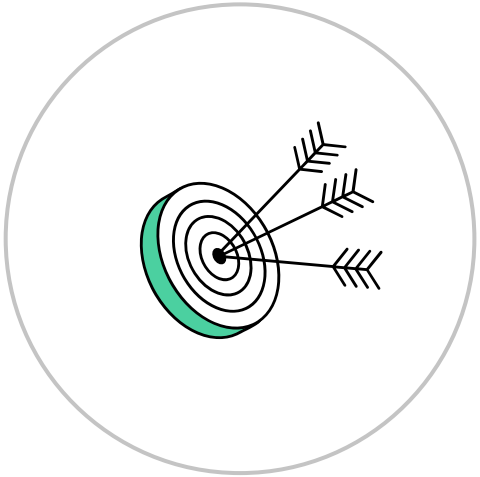
Вспоминаем прошлое занятие

Вопрос: о чём нам говорит принцип инкапсуляции?

Ответ: о том, что нужно строго контролировать, какие члены класса используются внешним кодом, и по возможности минимизировать их число

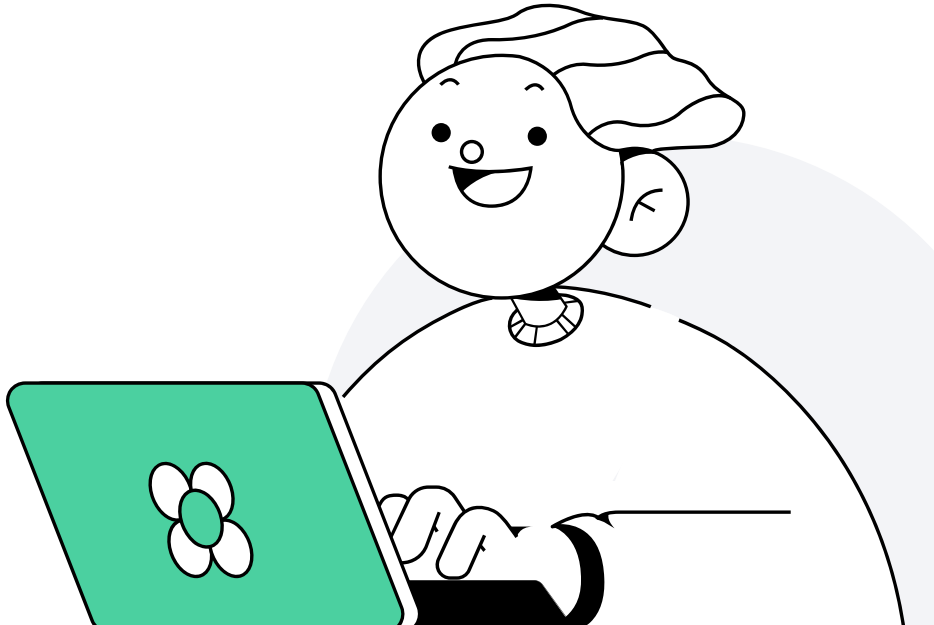
Цели занятия

- Разберёмся, что такое наследование
- Познакомимся с особенностями наследования конструкторов
- Узнаем, что такое множественное наследование
- Выясним, что такое полиморфизм



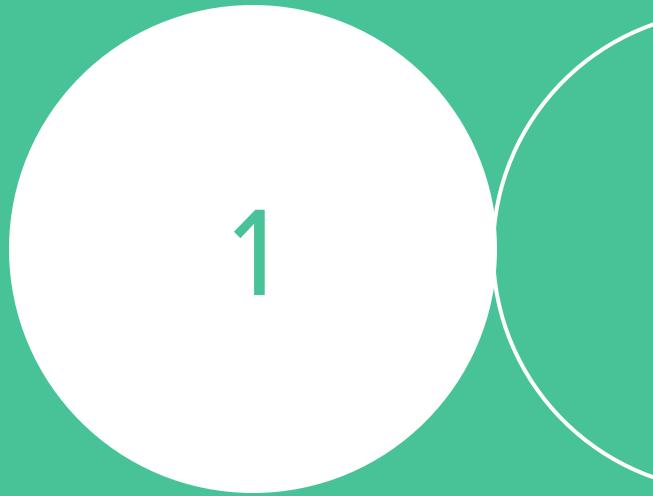
План занятия

1. Наследование
2. Множественное наследование
3. Полиморфизм
4. Итоги
5. Домашнее задание



*Нажми на нужный раздел для перехода

Наследование



Наследование

Наследование - это даже не совсем *принцип*, ему скорее подойдёт название *механизм*.

Наследование - один из самых важных и широко используемых механизмов в ООП

Про наследование сразу будем говорить в рамках языка C++

Что такое наследование

Наследование - это понятие, которое относится к классам (и структурам - в дальнейшем всё, что касается классов, касается и структур тоже)

Один класс может **наследовать** другой класс - в таком случае у нас есть:

- **класс-родитель** (суперкласс, базовый класс и родительский класс)
- **класс-потомок** (класс-наследник, подкласс, производный класс)

Что такое наследование

Когда один класс наследует другой - он наследует все его **поля** и **методы**

Это означает, что все поля и методы, объявленные в классе-родителе появляются и в классе-наследнике без дополнительных телодвижений. При этом мы можем добавлять новые поля и методы в класс-наследник

Синтаксис наследования

Для того, чтобы унаследовать один класс от другого, нам нужно в объявлении класса-потомка после его названия поставить двоеточие и указать название класса-родителя с модификатором доступа (пока это будет public, разберёмся позднее)

```
class Parent
{ ... };

class Child : public Parent
{ ... };
```

Разбираем механизм наследования

После того, как мы унаследовали класс Child от класса Parent, объекты класса Child получили поля и методы класса Parent

```
class Parent
{
public:
    int field1;
    void method1() { }
};
class Child: public Parent
{ };
int main(int argc, char** argv)
{
    Child child;
    child.field1 = 20; // можно обратиться к полю field1 у объекта типа Child
    child.method1(); // можно вызвать метод method1 у объекта типа Child
}
```

Разбираем механизм наследования

Также мы можем обращаться к унаследованным полям и методам внутри класса-наследника

```
class Parent
{
public:
    int field1;
    void method1() { }
};
class Child: public Parent
{
public:
    void method2()
    {
        field1 = 30;
        method1();
    }
};
```

Модификаторы доступа

Посмотрите на код и скажите, где будет ошибка, а где код выполнится?

```
class Parent
{
public:
    void method1()
    {
        public_method(); // ???
        protected_method(); // ???
        private_method(); // ???
    }
    void public_method() { }
protected:
    void protected_method() { }
private:
    void private_method() { }
};
```

```
class Child: public Parent
{
public:
    void method2()
    {
        public_method(); // ???
        protected_method(); // ???
        private_method(); // ???
    }
};
int main(int argc, char** argv)
{
    Child child;
    child.public_method(); // ???
    child.protected_method(); // ???
    child.private_method(); // ???
}
```

Модификаторы доступа

Модификатор доступа protected даёт доступ к члену в классе и его наследниках

```
class Parent
{
public:
    void method1()
    {
        public_method(); // правильно
        protected_method(); // правильно
        private_method(); // правильно
    }
    void public_method() { }
protected:
    void protected_method() { }
private:
    void private_method() { }
};
```

```
class Child: public Parent
{
public:
    void method2()
    {
        public_method(); // правильно
        protected_method(); // правильно
        private_method(); // ошибка!
    }
};
int main(int argc, char** argv)
{
    Child child;
    child.public_method(); // правильно
    child.protected_method(); // ошибка!
    child.private_method(); // ошибка!
}
```

Модификаторы доступа при наследовании

Разберём, что означает модификатор доступа, указываемый перед классом-родителем при наследовании

Этот модификатор доступа контролирует видимость **унаследованных** членов:

- public: видимость унаследованных членов **не изменяется**
- protected: все унаследованные члены с видимостью public **меняют** свою видимость на protected, остальные без изменений
- private: все унаследованные члены (с видимостью public и protected) **меняют** свою видимость на private (для класса-наследника)

Если модификатор доступа при наследовании не указан, то по умолчанию он является private

Модификаторы доступа при наследовании

```
class Parent
{
public:
    void public_method() { }
protected:
    void protected_method() { }
private:
    void private_method() { }
};

class Child: public Parent
{
public:
    void method1()
    {
        public_method(); // ???
        protected_method(); // ???
        private_method(); // ???
    }
};
```

```
class GrandChild: public Child
{
public:
    void method2()
    {
        public_method(); // ???
        protected_method(); // ???
        private_method(); // ???
    }
};
int main(int argc, char** argv)
{
    Child child;
    child.public_method(); // ???
    child.protected_method(); // ???
    child.private_method(); // ???
    GrandChild grand_child;
    grand_child.public_method(); // ???
    grand_child.protected_method(); // ???
    grand_child.private_method(); // ???
}
```

Модификаторы доступа при наследовании

```
class Parent
{
public:
    void public_method() { }
protected:
    void protected_method() { }
private:
    void private_method() { }
};

class Child: public Parent
{
public:
    void method1()
    {
        public_method(); // правильно
        protected_method(); // правильно
        private_method(); // ошибка!
    }
};
```

```
class GrandChild: public Child
{
public:
    void method2()
    {
        public_method(); // правильно
        protected_method(); // правильно
        private_method(); // ошибка!
    }
};
int main(int argc, char** argv)
{
    Child child;
    child.public_method(); // правильно
    child.protected_method(); // ошибка!
    child.private_method(); // ошибка!
    GrandChild grand_child;
    grand_child.public_method(); // правильно
    grand_child.protected_method(); // ошибка!
    grand_child.private_method(); // ошибка!
}
```

Модификаторы доступа при наследовании

```
class Parent
{
public:
    void public_method() { }
protected:
    void protected_method() { }
private:
    void private_method() { }
};

class Child: protected Parent
{
public:
    void method1()
    {
        public_method(); // ???
        protected_method(); // ???
        private_method(); // ???
    }
};
```

```
class GrandChild: public Child
{
public:
    void method2()
    {
        public_method(); // ???
        protected_method(); // ???
        private_method(); // ???
    }
};
int main(int argc, char** argv)
{
    Child child;
    child.public_method(); // ???
    child.protected_method(); // ???
    child.private_method(); // ???
    GrandChild grand_child;
    grand_child.public_method(); // ???
    grand_child.protected_method(); // ???
    grand_child.private_method(); // ???
}
```

Модификаторы доступа при наследовании

```
class Parent
{
public:
    void public_method() { }
protected:
    void protected_method() { }
private:
    void private_method() { }
};

class Child: protected Parent
{
public:
    void method1()
    {
        public_method(); // правильно
        protected_method(); // правильно
        private_method(); // ошибка!
    }
};
```

```
class GrandChild: public Child
{
public:
    void method2()
    {
        public_method(); // правильно
        protected_method(); // правильно
        private_method(); // ошибка!
    }
};
int main(int argc, char** argv)
{
    Child child;
    child.public_method(); // ошибка!
    child.protected_method(); // ошибка!
    child.private_method(); // ошибка!
    GrandChild grand_child;
    grand_child.public_method(); // ошибка!
    grand_child.protected_method(); // ошибка!
    grand_child.private_method(); // ошибка!
}
```

Модификаторы доступа при наследовании

```
class Parent
{
public:
    void public_method() { }
protected:
    void protected_method() { }
private:
    void private_method() { }
};

class Child: private Parent
{
public:
    void method1()
    {
        public_method(); // ???
        protected_method(); // ???
        private_method(); // ???
    }
};
```

```
class GrandChild: public Child
{
public:
    void method2()
    {
        public_method(); // ???
        protected_method(); // ???
        private_method(); // ???
    }
};
int main(int argc, char** argv)
{
    Child child;
    child.public_method(); // ???
    child.protected_method(); // ???
    child.private_method(); // ???
    GrandChild grand_child;
    grand_child.public_method(); // ???
    grand_child.protected_method(); // ???
    grand_child.private_method(); // ???
}
```

Модификаторы доступа при наследовании

```
class Parent
{
public:
    void public_method() { }
protected:
    void protected_method() { }
private:
    void private_method() { }
};

class Child: private Parent
{
public:
    void method1()
    {
        public_method(); // правильно
        protected_method(); // правильно
        private_method(); // ошибка!
    }
};
```

```
class GrandChild: public Child
{
public:
    void method2()
    {
        public_method(); // ошибка!
        protected_method(); // ошибка!
        private_method(); // ошибка!
    }
};
int main(int argc, char** argv)
{
    Child child;
    child.public_method(); // ошибка!
    child.protected_method(); // ошибка!
    child.private_method(); // ошибка!
    GrandChild grand_child;
    grand_child.public_method(); // ошибка!
    grand_child.protected_method(); // ошибка!
    grand_child.private_method(); // ошибка!
}
```

Наследование конструкторов

В отличие от методов и полей, конструкторы в C++ автоматически **не наследуются**

Однако при создании объекта класса-потомка вызывается как конструктор класса-потомка, так и конструктор класса-родителя, причём в строго определённом порядке

Вопрос: Как вы думаете, в каком порядке? Напишите в чат

Наследование конструкторов

Вопрос: Как вы думаете, в каком порядке? Напишите в чат
Ответ: Сначала вызывается конструктор родителя, потом конструктор наследника

```
class Parent
{
public:
    Parent()
    {
        std::cout << "Создаю родителя\n";
    }
};

class Child: public Parent
{
public:
    Child()
    {
        std::cout << "Создаю потомка\n";
    }
};
```

```
int main(int argc, char** argv)
{
    Parent parent; // Создаю родителя

    Child child; // Создаю родителя
                 Создаю потомка
}
```

Наследование конструкторов

Вопрос: Как вы думаете, что будет, если в родительском классе отсутствует конструктор без параметров?

```
class Parent
{
public:
    Parent(int a)
    {
        std::cout << "Создаю родителя " << a <<
" \n";
    }
};
class Child: public Parent
{
public:
    Child()
    {
        std::cout << "Создаю потомка\n";
    }
};
```

```
int main(int argc, char** argv)
{
    Parent parent(3); // Создаю родителя 3

    Child child; // ???
                    Создаю потомка
}
```

Наследование конструкторов

Ответ: Код не скомпилируется

```
class Parent
{
public:
    Parent(int a)
    {
        std::cout << "Создаю родителя " << a
<< " \n";
    }
};
class Child: public Parent
{
public:
    Child() // Ошибка!
    {
        std::cout << "Создаю потомка\n";
    }
};
```

```
int main(int argc, char** argv)
{
    Parent parent(3); // Создаю родителя 3

    Child child; // Ошибка!
                 Создаю потомка
}
```

Наследование конструкторов

Почему возникла ошибка?

Дело в том, что по умолчанию конструктор класса-наследника пытается вызвать конструктор класса-родителя **без параметров**. Если у класса-родителя такой есть - а это происходит в случае, если программист его объявил сам или не объявил вообще никакого конструктора - то всё в порядке

Но в предыдущем примере объявили в классе-родителе только один конструктор с целочисленным параметром, а это значит, что конструктор без параметров у класса-родителя **отсутствует**

Выбираем конструктор родителя

Как сделать так, чтобы мы могли выбрать, какой конструктор класса-родителя будет вызван перед конструктором класса-потомка?

Для этого нужно сразу после имени конструктора и круглых скобок конструктора в классе-потомке поставить двоеточие и “вызвать” нужный конструктор класса-родителя через его название и скобки, а в скобки подставить значение параметра или параметров. Или можем оставить скобки пустыми - тогда будет вызван конструктор без параметров, что является ситуацией по умолчанию:

```
class Child : public Parent
{
    Child() : Parent([<параметры>])
    { ... }
}
```

Выбираем конструктор родителя

Устраним ошибку, подставив значение по умолчанию (например, 5)

```
class Parent
{
public:
    Parent(int a)
    {
        std::cout << "Создаю родителя " << a
<< " \n";
    }
};
class Child: public Parent
{
public:
    Child() : Parent(5)
    {
        std::cout << "Создаю потомка\n";
    }
};
```

```
int main(int argc, char** argv)
{
    Parent parent(3); // Создаю родителя 3

    Child child; // Создаю родителя 5
                 Создаю потомка
}
```

Выбираем конструктор родителя

Можно также передавать параметр из конструктора наследника в конструктор родителя

```
class Parent
{
public:
    Parent(int a)
    {
        std::cout << "Создаю родителя " << a
<< " \n";
    }
};
class Child: public Parent
{
public:
    Child(int a) : Parent(a)
    {
        std::cout << "Создаю потомка\n";
    }
};
```

```
int main(int argc, char** argv)
{
    Parent parent(3); // Создаю родителя 3

    Child child(8); // Создаю родителя 8
                    Создаю потомка
}
```

Вызываем свой же конструктор

Более того, мы можем указать, что один наш конструктор должен перед своим исполнением вызвать другой наш конструктор!

Вопрос: Как вы думаете, как это сделать? Напишите в чат

Вызываем свой же конструктор

Более того, мы можем указать, что один наш конструктор должен перед своим исполнением вызвать другой наш конструктор!

Вопрос: Как вы думаете, как это сделать? Напишите в чат

Ответ: Для этого после двоеточия вместо названия класса-родителя надо написать название своего класса и указать нужные параметры

```
class Child : public Parent
{
    Child() : Child([<параметры>]) // вызываем конструктор Child с параметрами
    { ... }
    Child([<параметры>]) : Parent([<параметры>]) // вызываем конструктор Parent с
                                                 параметрами
    { ... }
}
```

Вызываем свой же конструктор

Как вы думаете, что будет выведено на консоль? Напишите в чат

```
class Parent
{
public:
    Parent(int a)
    {
        std::cout << "Создаю
родителя " << a << " \n";
    }
};
```

```
class Child: public Parent
{
public:
    Child() : Child(5)
    {
        std::cout << "Создаю потомка\n";
    }
    Child(int a) : Parent(a)
    {
        std::cout << "Создаю потомка " << a << " \n";
    }
};
int main(int argc, char** argv)
{
    Parent parent(3); // Создаю родителя 3
    Child child_empty; // ???
    Child child_8(8); // ???
}
```

Вызываем свой же конструктор

```
class Parent
{
public:
    Parent(int a)
    {
        std::cout << "Создаю
родителя " << a << " \n";
    }
};
```

```
class Child: public Parent
{
public:
    Child() : Child(5)
    {
        std::cout << "Создаю потомка\n";
    }
    Child(int a) : Parent(a)
    {
        std::cout << "Создаю потомка " << a << " \n";
    }
};
int main(int argc, char** argv)
{
    Parent parent(3); // Создаю родителя 3
    Child child_empty; // Создаю родителя 5
                          Создаю потомка 5
                          Создаю потомка
    Child child_8(8); // Создаю родителя 8
                         Создаю потомка 8
}
```

Множественное наследование

2

Множественное наследование

В C++ нам доступно **множественное наследование**

Множественное наследование - это когда один класс имеет сразу несколько родительских классов. Тогда класс-потомок наследует поля и методы всех родительских классов

```
class Parent1 { ... };
class Parent2 { ... };
...
class ParentN { ... };

class Child : [<модиф.>] Parent1, [<модиф.>] Parent2, ... , [<модиф.>] ParentN { ... };
```

Множественное наследование. Пример

```
class Parent1
{
public:
    void method1() { }
};
class Parent2
{
public:
    void method2() { }
};
class Parent3
{
public:
    void method3() { }
};
```

```
class Child: public Parent1, public Parent2, public Parent3
{
};
int main(int argc, char** argv)
{
    Child child;
    child.method1(); // правильно
    child.method2(); // правильно
    child.method3(); // правильно
}
```

Как вы думаете

какие потенциальные проблемы кроются в множественном наследовании?

Напишите в чат

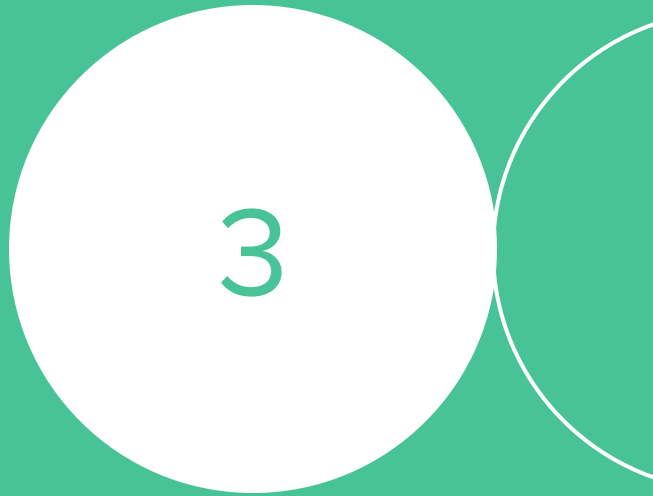
Множественное наследование. Проблемы

1. Проблема с множественным наследованием возникает, если в классах-родителях окажутся методы с одинаковой сигнатурой. Какой из этих методов будет вызван в классе-потомке?

2. Также существует проблема ромбовидного наследования - когда дочерний класс наследует два родительских, а родительские наследуются от одного и того же класса “дедушки”. В таком случае возникает сразу несколько непростых проблем

Поэтому множественное наследование используется редко, а во многих ООП-языках и вовсе запрещено

Полиморфизм



Полиформизм

это способность, имея объект родительского класса, вызывать из него методы его потомков

Прежде, чем мы познакомимся с тем,
как это делается, мы должны узнать ещё кое-что
интересное о классах-наследниках

Приведение типов

Ранее мы приводили тип float к типу double
Для такой комбинации типов доступно неявное приведение - то есть мы можем просто присвоить переменной типа double значение типа float

Для базовых и производных классов работает похожий механизм -
мы можем присвоить переменной типа **указателя на объект родительского класса** значение типа **указателя на объект производного класса** - то есть мы можем привести объект производного класса к базовому типу

Однако у такой переменной будут доступны только те члены, которые объявлены в родительском классе - члены, объявленные в производном классе, будут недоступны

Приведение типов. Пример

```
class Parent
{
public:
    int p_field;
    Parent() { p_field = 1; }
};
class Child: public Parent
{
public:
    int c_field;
    Child()
    {
        p_field = 2;
        c_field = 3;
    }
};
```

```
int main(int argc, char** argv)
{
    Parent parent;
    std::cout << parent.p_field << std::endl; // ???
    std::cout << parent.c_field << std::endl; // ???
    Child child;
    std::cout << child.p_field << std::endl; // ???
    std::cout << child.c_field << std::endl; // ???
    Parent* par_child = &child;
    std::cout << par_child->p_field << std::endl; // ???
    std::cout << par_child->c_field << std::endl; // ???
}
```

Приведение типов. Пример

```
class Parent
{
public:
    int p_field;
    Parent() { p_field = 1; }
};
class Child: public Parent
{
public:
    int c_field;
    Child()
    {
        p_field = 2;
        c_field = 3;
    }
};
```

```
int main(int argc, char** argv)
{
    Parent parent;
    std::cout << parent.p_field << std::endl; // 1
    std::cout << parent.c_field << std::endl; // Ошибка!
    Child child;
    std::cout << child.p_field << std::endl; // 2
    std::cout << child.c_field << std::endl; // 3
    Parent* par_child = &child;
    std::cout << par_child->p_field << std::endl; // 2
    std::cout << par_child->c_field << std::endl; // Ошибка!
}
```

Как вы думаете

что будет, если в базовом и производном классе будут методы с одинаковой сигнатурой?

Напишите в чат

Приведение типов. Пример

```
class Parent
{
public:
    void method()
    {
        std::cout << "Родитель";
    }
};
class Child: public Parent
{
public:
    void method()
    {
        std::cout << "Потомок";
    }
};
```

```
int main(int argc, char** argv)
{
    Parent parent;
    parent.method(); // Родитель
    Child child;
    child.method(); // Потомок
    Parent* par_child = &child;
    par_child->method(); // Родитель
}
```

Виртуальные методы - это методы, помеченные ключевым словом **virtual** - оно указывается до типа возвращаемого значения

Такие методы могут быть **переопределены** в производном классе - это значит, что при вызове метода, объявленного в родительском классе и переопределенного в дочернем классе, у объекта дочернего класса с помощью указателя на родительский класс - будет вызвана реализация из производного класса

Виртуальные методы

Переопределенный метод обычно помечается ключевым словом **override**, которое указывается после сигнатуры метода и до его тела

Использовать его необязательно, однако его наличие позволяет удостовериться в том, что вы переопределяете виртуальный метод - если это не так, будет сгенерирована ошибка компиляции

Виртуальные методы. Пример

```
class Parent
{
public:
    virtual void method()
    {
        std::cout << "Родитель";
    }
};
class Child: public Parent
{
public:
    void method() override
    {
        std::cout << "Потомок";
    }
};
```

```
int main(int argc, char** argv)
{
    Parent parent;
    parent.method(); // ???
    Child child;
    child.method(); // ???
    Parent* par_child = &child;
    par_child->method(); // ???
}
```

Виртуальные методы. Пример

```
class Parent
{
public:
    virtual void method()
    {
        std::cout << "Родитель";
    }
};
class Child: public Parent
{
public:
    void method() override
    {
        std::cout << "Потомок";
    }
};
```

```
int main(int argc, char** argv)
{
    Parent parent;
    parent.method(); // Родитель
    Child child;
    child.method(); // Потомок
    Parent* par_child = &child;
    par_child->method(); // Потомок
}
```

Вызов базовой реализации

Мы можем вызвать базовую реализацию - похоже на вызов конструктора предка

```
class Parent
{
public:
    virtual void method()
    {
        std::cout << "Родитель";
    }
};
class Child: public Parent
{
public:
    void method() override
    {
        std::cout << "Потомок\n";
        Parent::method();
    }
};
```

```
int main(int argc, char** argv)
{
    Parent parent;
    parent.method(); // Родитель
    Child child;
    child.method(); // Потомок
                    Родитель
    Parent* par_child = &child;
    par_child->method(); // Потомок
                         Родитель
}
```

Полиморфизм

С помощью наследования и виртуальных функций достигается полиморфизм. Это очень полезно для создания **полиморфных функций** - в таких функциях одним из аргументов является указатель на объект базового класса

Один раз написав такую функцию, в дальнейшем вы можете создавать новые классы на основе базового, переопределять там методы и вызывать вашу функцию, передавая туда указатели на объекты вашего нового класса (приведенные к базовому) - и функция сможет работать с ними!

Итоги

Итоги занятия

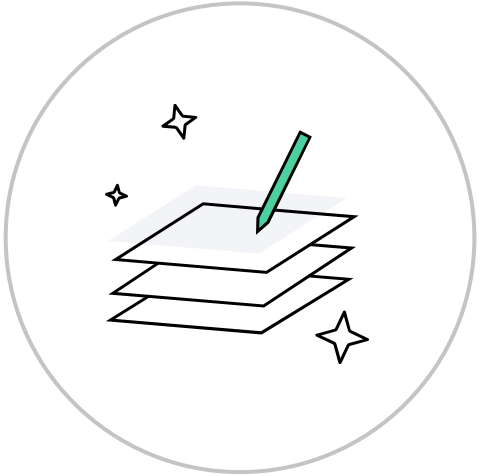
Сегодня мы

1. Разобрались, что такое наследование
2. Познакомились с особенностями наследования конструкторов
3. Узнали, что такое множественное наследование
4. Выяснили, что такое полиморфизм

Домашнее задание

Давайте посмотрим ваше домашнее задание.

1. Вопросы по домашней работе задавайте в чате группы
2. Задачи можно сдавать по частям
3. Зачёт по домашней работе ставят после того, как приняты все задачи



Дополнительные материалы

- Наследование
- Полиморфизм

Задавайте вопросы и пишите отзыв о лекции

Максим Бакиров
C++ - разработчик в Яндекс