

Статические члены класса. Дружественные функции и классы.

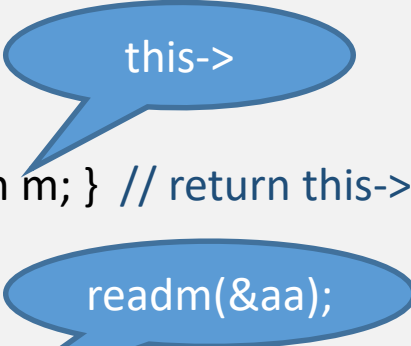
Лекция + практика

Скрытый указатель *this

Указатель ***this** — это скрытый константный указатель, содержащий адрес объекта, который вызывает метод класса.

С его помощью метод класса определяет, с данными какого объекта ему предстоит работать.

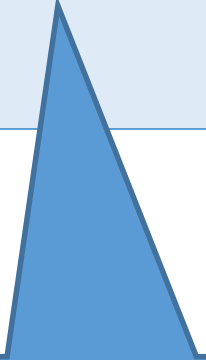
```
class Example {  
    int m;  
public:  
    int readm() { return m; } // return this->m  
};  
void f() {  
    Example aa, bb;  
    int a = aa.readm(); // this указывает на aa  
    int b = bb.readm(); // this указывает на bb  
}
```



Другое практическое применение **this** - с его помощью можно возвращать текущий объект класса

Указатель **this** может быть использован только для **нестатического метода**.

```
class Something  
{  
    private:  
        int data;  
    public:  
        Something(int data)  
        {  
            this->data = data;  
        }  
};
```



Явное указание указателя ***this** (параметр с тем же именем, что и переменная-член)

Несколько замечаний о классах.

Когда классы становятся больше и сложнее, наличие всех методов внутри тела класса может затруднить его управление и работу с ним.


Определения методов следует организовывать вне тела самого класса. Для этого определите методы класса, как если бы они были обычными функциями, но в качестве префикса добавьте к имени функции имя класса с оператором разрешения области видимости (::).

```
class Date
{
private:
    int m_day;
    int m_month;
    int m_year;

public:
    Date(int day, int month, int year);
    void SetDate(int day, int month, int year);
};
```

```
// Конструктор класса Date
Date::Date(int day, int month, int year)
{
    SetDate(day, month, year);
}

// Метод класса Date
void Date::SetDate(int day, int month, int year)
{
    m_day = day;
    m_month = month;
    m_year = year;
}
```



Несколько замечаний о классах.

Определения классов могут быть помещены в заголовочные файлы для облегчения их повторного использования в нескольких файлах или проектах. Обычно, определение класса помещается в заголовочный файл с тем же именем, что у класса, а методы, определенные вне тела класса, помещаются в файл `.cpp` с тем же именем, что у класса.

Классы — это пользовательские типы данных, которые *освобождаются от определения только в одном месте*. Поэтому класс, определенный в заголовочном файле, можно свободно подключать в другие файлы.

Параметры по умолчанию для методов должны быть объявлены в теле класса (в заголовочном файле), где они будут видны всем, кто подключает этот заголовочный файл с классом.

Методы, определенные внутри тела класса, считаются неявно встроенными. Встроенные функции освобождаются от правила одного определения. А это означает, что проблем с определением простых методов (таких как функции доступа) внутри самого класса возникать не должно.

Методы, определенные вне тела класса, рассматриваются, как обычные функции, и подчиняются правилу одного определения, поэтому эти функции должны быть определены в файле `.cpp`, а не внутри `.h`. Единственным исключением являются шаблоны функций.

Классы и const

Объекты классов можно сделать константными (используя ключевое слово **const**). Инициализация выполняется через конструкторы классов:

```
const Date date1;           // инициализация через конструктор по умолчанию
const Date date2(12, 11, 2018); // инициализация через конструктор с параметрами
const Date date3 { 12, 11, 2018 }; // инициализация через конструктор с параметрами в C++11
```

Константный метод — это метод, который гарантирует, что не будет изменять объект или вызывать неконстантные методы класса (поскольку они могут изменить объект).

Для методов, определенных вне тела класса, ключевое слово **const** должно использоваться как в прототипе функции (в теле класса), так и в определении функции

Константные объекты класса могут явно вызывать только константные методы класса

Запрещается как изменение переменных-членов напрямую (если они являются **public**), так и вызов методов (**сеттеров**), с помощью которых можно установить значения переменным-членам.

class Anything

{
public:

int m_value;

Anything() { m_value= 0; }

int getValue() const

{ return m_value; } // ключевое слово const находится

после списка параметров, но перед телом функции

};

Чтобы сделать метод константным, нужно добавить ключевое слово **const** к прототипу функции после списка параметров, но перед телом функции

Классы и const

Другим способом создания константных объектов является передача объектов в функцию по константной ссылке.

Функцию можно перегрузить таким образом, чтобы иметь константную и неконстантную версии одной и той же функции

Константная версия функции будет вызываться для константных объектов, а неконстантная версия будет вызываться для неконстантных объектов:

Статические члены класса

```
static int s_id = 0;
```

Статические переменные сохраняют свои значения и не уничтожаются даже после выхода из блока, в котором

Переменные-члены класса можно сделать статическими, используя ключевое слово **static**. В отличие от обычных переменных-членов, статические переменные-члены являются общими для всех объектов класса.

Ключевое слово **static** имеет другое значение, когда речь идет о глобальных переменных — оно предоставляет им внутреннюю связь (что ограничивает их видимость/использование за пределами файла, в котором они определены)

```
#include <iostream>
class Anything{
public:
    static int s_value;
};
int Anything::s_value = 3;

int main(){
    Anything first;
    Anything second;
    first.s_value = 4;
    std::cout << first.s_value << '\n'; //4
    std::cout << second.s_value <<
    '\n';//4
    return 0;
}
```

- Статические члены существуют, даже если объекты класса не созданы! Подобно глобальным переменным, они создаются при запуске программы и уничтожаются, когда программа завершает свое выполнение.
- Статические члены принадлежат классу, а не объектам этого класса.
- Доступ к стат. перемен. осуществляется через имя класса, а не через объект этого класса

Необходимо явно определить статический член вне тела класса — в глобальной области видимости. Вы можете определить и инициализировать **s_value**, даже если он будет **private** (или **protected**)

Зачем нужны статические члены класса?

Для присваивания уникального идентификатора каждому объекту класса (как вариант)

Статические методы класса

C++ не поддерживает статические конструкторы!

Если статические переменные-члены являются закрытыми, мы можем сделать метод доступа к такой переменной статическим.

Подобно статическим переменным-членам, статические методы не привязаны к какому-либо одному объекту класса

```
class Anything
{
private: // статическая переменная
    static int s_value;
public: //статический метод
    static int getValue() { return s_value; }
};
int Anything::s_value = 3; // определение
статической переменной-члена класса
int main()
{
    std::cout << Anything::getValue() << '\n';
}
```

У статических методов есть две особенности:

- они не имеют скрытого указателя `*this`!
- они могут напрямую обращаться к другим статическим членам (переменным или функциям), но не могут напрямую обращаться к нестатическим членам.

Будьте осторожны при написании классов со всеми статическими членами:

- ❖ Поскольку все статические члены создаются только один раз, то несколько копий «чисто статического класса» быть не может (без клонирования класса и его дальнейшего переименования).
- ❖ Глобальные переменные опасны, поскольку все члены принадлежат классу (а не его объектам), а классы имеют глобальную область видимости, то в «чисто статическом классе» мы объявляем глобальные функции и переменные со всеми минусами, которые они имеют.

Проблема взаимного доступа.

Допустим у вас есть класс и функция, которая работает с этим классом, но которая не находится в его теле.

Есть два варианта решения:

- ✓ **Сделать открытыми методы класса** и через них функция будет взаимодействовать с классом. Однако здесь есть несколько нюансов. Во-первых, эти открытые методы нужно будет определить, на что потребуется время, и они будут загромождать интерфейс класса. Во-вторых, в классе нужно будет открыть методы, которые не всегда должны быть открытыми и предоставляющими доступ извне.
- ✓ **Использовать дружественные классы и дружественные функции**, с помощью которых можно будет предоставить функции доступ к закрытым данным класса. Это позволит функции напрямую обращаться ко всем закрытым переменным-членам и методам класса, сохраняя при этом закрытый доступ к данным класса для всех остальных функций вне тела класса!

Дружественные функции

Дружественная функция — это функция, которая имеет доступ к закрытым членам класса, как если бы она сама была членом этого класса.

Во всех других отношениях дружественная функция является обычной функцией. Ею может быть, как обычная функция, так и метод другого класса.

Для объявления дружественной функции используется ключевое слово **friend** перед прототипом функции, которую вы хотите сделать дружественной классу.

```
class Anything
{
private:
    int m_value;
public:
    Anything() { m_value = 0; }
    void add(int value) { m_value += value; }
    // Делаем функцию reset() дружественной классу Anything
    friend void reset(Anything &anything);
};
// Функция reset() теперь является другом класса Anything
void reset(Anything &anything)
{
    // И мы имеем доступ к закрытым членам объектов класса
    Anything
    anything.m_value = 0;
}
int main()
{
    Anything one;
    one.add(4); // добавляем 4 к m_value
    reset(one); // сбрасываем m_value в 0
    return 0;
}
```

Дружественные методы

Вместо того, чтобы делать дружественным целый класс, мы можем сделать дружественными только определенные методы класса. Их объявление аналогично объявлениям обычных дружественных функций, за исключением имени метода с префиксом **имяКласса::** в начале.

```
#include <iostream>

class Values; // предварительное объявление
class Display
{
private:
    bool m_displayIntFirst;
public:
    Display(bool displayIntFirst)
    { m_displayIntFirst = displayIntFirst; }
    void displayItem(Values &value);
};
```

```
class Values // полное определение класса Values
{
private:
    int    m_intValue;
    double m_dValue;
public:
    Values(int intValue, double dValue){
        m_intValue = intValue;
        m_dValue = dValue;
    }
    // Делаем метод Display::displayItem() другом класса Values
    friend void Display::displayItem(Values& value);
};
```

Дружественные методы. Продолжение

// Теперь мы можем определить метод `Display::displayItem()`, которому требуется увидеть полное определение класса `Values`

```
void Display::displayItem(Values &value)
{
    if (m_displayIntFirst)
        std::cout << value.m_intValue << " " << value.m_dValue << '\n';
    else // или выводим сначала double
        std::cout << value.m_dValue << " " << value.m_intValue << '\n';
}
```

```
int main()
{
    Values value(7, 8.4);
    Display display(false);
    display.displayItem(value);

    return 0;
}
```

Дружественные классы

Один класс может быть дружественным другому классу. Это откроет всем членам первого класса доступ к закрытым членам второго класса

```
#include <iostream>
class Values{
private:
    int m_intValue;
public:
    Values(int intValue){
        m_intValue = intValue;
    }
    // Делаем класс Display другом
    // класса Values
    friend class Display;
};
```

```
class Display{
private:
    bool m_displayIntFirst;
public:
    Display(bool displayIntFirst)
        { m_displayIntFirst = displayIntFirst; }

    void displayItem(Values &value){
        if (m_displayIntFirst)
            std::cout << value.m_intValue << '\n';
        }
    };
};
```

Если вы хотите сделать оба класса дружественными, то каждый из них должен указать в качестве друга противоположный класс. Наконец, если класс А является другом В, а В является другом С, то это не означает, что А является другом С.

```
int main(){

    Values value(7);
    Display display(false);
    display.displayItem(value);
    return 0;
}
```

Примечания о дружественных классах:

- несмотря на то, что Display является другом Values, Display не имеет прямой доступ к указателю *this объектов Values.
- даже если Display является другом Values, это не означает, что Values также является другом Display.

Будьте внимательны при использовании дружественных функций и классов, поскольку это может нарушать принципы инкапсуляции

Анонимные объекты

Анонимные объекты в языке C++ используются для передачи или возврата значений без необходимости создавать большое количество временных переменных. Динамическое выделение памяти также выполняется через анонимные объекты (поэтому адрес выделенной памяти должен быть присвоен указателю, иначе мы не имели бы способа сослаться/использовать её).

```
class Dollars{
private:
    int m_dollars;
public:
    Dollars(int dollars): m_dollars(dollars){ }
    int getDollars() const { return m_dollars; }
};

Dollars add(const Dollars &d1, const Dollars &d2)
{
    return Dollars(d1.getDollars() + d2.getDollars());
// возвращаем анонимный объект класса Dollars
}
```

Анонимные объекты можно использовать только один раз, так как они имеют область видимости выражения.

```
int main()
{

// выводим анонимный объект класса Dollars
    std::cout << "I have " << add(Dollars(7),
                                   Dollars(9)).getDollars() << " dollars." <<
    std::endl;

    return 0;
}
```

П. Цепочки методов класса

```
class Mathem
{
private:
    int m_value;

public:
    Mathem() { m_value = 0; }

    void add(int value) { m_value += value; }
    void sub(int value) { m_value -= value; }
    void multiply(int value) { m_value *= value; }
}

int getValue() { return m_value; }
};
```

```
#include <iostream>

int main()
{
    Mathem operation;
    operation.add(7); // возвращает void
    operation.sub(5); // возвращает void
    operation.multiply(3); // возвращает void

    std::cout << operation.getValue() << '\n';
    return 0;
}
```

Это еще не цепочка

П. Цепочки методов класса

```
class Mathem {  
private:  
    int m_value;  
public:  
    Mathem() { m_value = 0; }  
  
    Mathem& add(int value) { m_value += value; return *this; }  
    Mathem& sub(int value) { m_value -= value; return *this; }  
    Mathem& multiply(int value) { m_value *= value; return *this; }  
    int getValue() { return m_value; }  
};
```

```
#include <iostream>
```

```
int main()  
{
```

```
    Mathem operation;
```

```
    operation.add(7).sub(5).multiply(3);
```

```
    std::cout << operation.getValue() << '\n';
```

```
    return 0;
```

```
}
```

А это уже
цепочка

П. Статические переменные-члены.

Если статический член является `const` интегральным типом (к которому относятся и **`char`**, и **`bool`**) или **`const enum`**, то статический член может быть инициализирован внутри тела класса:

С C++11 статические члены **`constexpr`** любого типа данных, поддерживающие инициализацию `constexpr`, могут быть инициализированы внутри тела класса:

```
class Anything
{
public:
    static const int s_value = 5; // статическую
    константную переменную типа int можно
    объявить и инициализировать напрямую
};
```

```
#include <array>
class Anything
{
public:
    static constexpr double s_value = 3.4; // хорошо
    static constexpr std::array<int, 3> s_array = { 3, 4, 5 }; // это
    работает даже с классами, которые поддерживают
    инициализацию constexpr
};
```

П. Создание уникальных идентификаторов.

```
class Anything {
private:
    static int s_idGenerator;
    int      m_id;
public:
    Anything() { m_id = s_idGenerator++; } // увеличиваем значение
идентификатора для следующего объекта
    int getID() const { return m_id; }
};

// Мы определяем и инициализируем s_idGenerator несмотря на то, что он
// объявлен как private.
// Это нормально, поскольку определение не подпадает под действия
спецификаторов доступа
int Anything::s_idGenerator = 1; // начинаем наш ID-генератор со значения 1
```

```
int main()
{
    Anything first;
    Anything second;
    Anything third;

    std::cout << first.getID() << '\n';
    std::cout << second.getID() << '\n';
    std::cout << third.getID() << '\n';
    return 0;
}
```

Статические переменные-члены также могут быть полезны, когда классу необходимо использовать внутреннюю таблицу поиска (например, массив, используемый для хранения набора предварительно вычисленных значений). Делая таблицу поиска статической, для всех объектов класса создается только одна копия (нежели отдельная для каждого объекта класса). Это поможет сэкономить значительное количество памяти.

П. Дружественные функции и классы

```
class Humidity;
class Temperature{
    int m_temp;
public:
    Temperature(int temp=0) { m_temp = temp; }
    friend void outWeather(const Temperature
                           &temperature, const Humidity &humidity);
};

class Humidity{
private:
    int m_humidity;
public:
    Humidity(int humidity=0) { m_humidity = humidity; }
    friend void outWeather(const Temperature
                           &temperature, const Humidity &humidity);
};
```

```
void outWeather(const Temperature &temperature,
                const Humidity &humidity)
{
    std::cout << "The temperature is " <<
        temperature.m_temp <<
        " and the humidity is " << humidity.m_humidity
        << '\n';
}

int main()
{
    Temperature temp(15);
    Humidity hum(11);

    outWeather(temp, hum);

    return 0;
}
```

Домашняя работа # 12

Напишите класс под названием **Retailitem** (Розничная торговая единица), который содержит данные о товаре в розничном магазине. Этот класс должен хранить данные в атрибутах (полях): описание товара, кол-во единиц на складе и цена. После написания класса напишите приложение, которая создает 3 объекта этого класса и сохраняет в них приведенные в таблице 1 .

Таблица 1.

	Описание	Кол-во на складе	Цена
Товар№ 1	Пиджак	12	59.95
Товар№ 2	Джинсы	40	34.95
Товар№ 3	Рубашка	20	24.95

Создайте класс **CashRegister** (Кассовый аппарат), который может использоваться вместе с классом **Retailitem**.

Класс **CashRegister** должен иметь внутренний список объектов **Retailitem**, а также приведенные ниже методы.

- Метод **buy _ item ()** (приобрести товар) в качестве аргумента принимает объект **Retailrtem**. При каждом вызове метода **buy _ item ()** объект **Retailrtem**, переданный в качестве аргумента, должен быть добавлен в список.
- Метод **get _ total ()** (получить сумму покупки) возвращает общую стоимость всех объектов **Retailrtem**, хранящихся во внутреннем списке объекта **CashRegister**.
- Метод **show items ()** (показать товары) выводит данные об объектах класса **Retailrtem**, хранящихся во внутреннем списке объекта класса **CashRegister**.
- Метод **clear ()** (очистить) должен очистить внутренний список объекта **CashRegister**.

Продемонстрируйте класс **CashRegister** в программе, которая позволяет пользователю выбрать несколько товаров для покупки. Когда пользователь готов рассчитаться за покупку, программа должна вывести список всех товаров, которые он выбрал для покупки, а также их общую стоимость.