Урок 1

Основные понятия ООП. Инкапсуляция

История и причины возникновения ООП. Классы и объекты. Методы классов. Инкапсуляция данных и методы доступа. Средства ограничения доступа. Список инициализации членов класса. Конструкторы и деструкторы.

Введение в ООП

Объектно-ориентированное программирование — это парадигма программирования, основные концепции которой — понятия объектов и классов. Ранее вы изучали процедурное программирование, подразумевающее простое последовательное выполнение набора команд. В ООП все команды также выполняются последовательно, но есть существенное отличие в подходе к архитектуре: программист рассматривает все составные части программы как объекты со свойствами и поведением. ООП позволяет создавать объекты, которые объединяют несколько разнотипных данных, их свойства и функции.

Чтобы понять концепцию ООП, обратимся к реальной жизни. Вокруг нас находятся объекты: магазины, автомобили, книги, люди. Все они имеют два основных компонента:

- список свойств (цвет, размер, вес, форма и другие);
- список поведений (делать что-то, открывать и т.д.).

В программировании мы не ориентируемся на написание функций, а сосредотачиваемся на определении объектов, которые имеют поля (свойства) и методы (виды поведения).

Основные причины популярности ООП:

- повышение производительности и надежности программ;
- создание больших блоков программного кода, пригодных для многократного использования;
- упрощение написания и понимания кода.

Принципы ООП не заменяют традиционные методы программирования, а дополняют их.

Основными концепциями ООП являются абстракция, инкапсуляция, наследование и полиморфизм — рассмотрим их на ближайших уроках.

Классы и объекты

Для решения относительно простых задач язык C++ предоставляет ряд примитивных типов данных (**int**, **double**, **char** и другие). Для более сложных их может не хватать, поэтому C++ позволяет создавать пользовательские типы данных — например, перечисления (**enum**) и структуры (**struct**).

В следующем коде создается структура для хранения даты:

```
struct DateStruct
{
    int day;
    int month;
    int year;
};
```

Создадим перечисления для типа данных, хранящего дни недели:

```
enum days_of_week { Sun, Mon, Tue, Wed, Thu, Fri, Sat };
```

Перечисления и структуры — это средства традиционного подхода к программированию, поскольку с их помощью можно только хранить данные. Чтобы с ними можно было работать, необходимо

создавать отдельные функции. Для вывода текущей даты напишем соответствующую функцию — вот полный код программы:

```
#include <iostream>
struct DateStruct
{
    int day;
    int month;
    int year;
} today = {12, 12, 2018};

void print(DateStruct &date)
{
    std::cout << date.day<< "/" << date.month << "/" << date.year;
}

int main()
{
    today.day = 18; // используем оператор выбора члена для выбора члена
структуры
    print(today);

    return 0;
}
```

Результат выполнения программы: **18/12/2018**.

В парадигме ООП типы данных могут содержать не только данные, но и функции для их обработки. Чтобы определить такой тип данных, используется ключевое слово **class**. Структуры и классы по своему строению очень похожи, однако у классов больше возможностей. Фактически, следующая структура и класс идентичны:

```
struct DateStruct
{
    int day;
    int month;
    int year;
};

class DateClass
{
public:
    int m_day;
    int m_month;
    int m_year;
};
```

Единственное отличие — наличие ключевого слова **public**, о котором мы поговорим чуть позже.

Объявление класса, как и структуры, не приводит к выделению памяти до тех пор, пока мы не создадим переменную этого типа. Такая переменная называется **экземпляром класса** или **объектом**. Следующий фрагмент создает объект класса:

```
DateClass today { 12, 11, 2018 }; // объявляем переменную класса DateClass
```

Для объекта **today** уже будет выделена память.

Структуры и классы выступают шаблонами, по которым может быть создано несколько объектов. В этом и заключается их главное достоинство — формирование единообразных данных, с которыми можно одинаковым образом работать.

Методы и свойства класса

Переменные, объявленные внутри класса, называются **свойствами** или **переменными-членами**. Помимо этого класс может содержать функции: **методы** или **функции-члены**. Методы могут быть определены как внутри класса, так и вне него. Программа объявления метода вывода даты, объявленного внутри класса:

```
class DateClass
{
public:
    int m_day;
    int m_month;
    int m_year;

    void printDate() // определяем функцию-член
    {
        std::cout << m_day << "/" << m_month << "/" << m_year;
    }
    void printYear();
};

// определяем функцию-член вне класса
void DateClass::printYear()
{
    std::cout << m_year << " year";
}
```

К методам и свойствам класса обращаются с помощью оператора «точка» (.):

```
#include <iostream>

class DateClass
{
public:
    int m_day;
    int m_month;
    int m_year;

    void print()
    {
        std::cout << m_day << "/" << m_month << "/" << m_year;
    }
};

int main()
{
    DateClass today { 12, 12, 2018 };

    today.m_day = 18;
    today.print(); // используем оператор (.) для вызова метода объекта today класса DateClass

    return 0;
}
```

Результат выполнения программы: 18/12/2018.

Вызов функций-членов должен быть связан с объектом класса. Так компилятор может понять, для какого объекта вызывается функция. В определении самой функции мы не указываем, какому объекту принадлежат переменные. В данном примере мы просто обратились к членам-переменным с помощью строчки:

```
std::cout << m_day << "/" << m_month << "/" << m_year;
```

Такая запись возможна потому, что компилятор автоматически интерпретирует **m_day** как **today.m_day** и т.д. Связанный объект неявно передается функции-члену, поэтому такой объект называется **неявным объектом**.

Спецификаторы доступа public и private

Рассмотрим следующую программу, в которой нет ошибки:

```
struct DateStruct // члены структуры являются открытыми по умолчанию
{
    int day; // открыто по умолчанию, доступ имеет любой объект
    int month; // открыто по умолчанию, доступ имеет любой объект
    int year; // открыто по умолчанию, доступ имеет любой объект
};

int main()
{
    DateStruct date;
    date.day = 12;
    date.month = 11;
    date.year = 2018;

    return 0;
}
```

Данные члены структуры — открытые, или публичные (**public-члены**). К ним можно получить доступ извне этой структуры.

Теперь рассмотрим следующий код:

```
class DateClass // члены класса являются закрытыми по умолчанию
{
    int m_day; // закрыто по умолчанию, доступ имеют только другие члены класса
    int m_month; // закрыто по умолчанию, доступ имеют только другие члены
класса
    int m_year; // закрыто по умолчанию, доступ имеют только другие члены класса
};

int main()
{
    DateClass date;
    date.m_day = 12; // ошибка
    date.m_month = 11; // ошибка
    date.m_year = 2018; // ошибка

    return 0;
}
```

Компилятор сообщает об ошибках, так как переменные класса являются закрытыми по умолчанию. **Члены private** — это члены класса, которые не могут быть изменены извне, и доступ к которым возможен только для других членов класса.

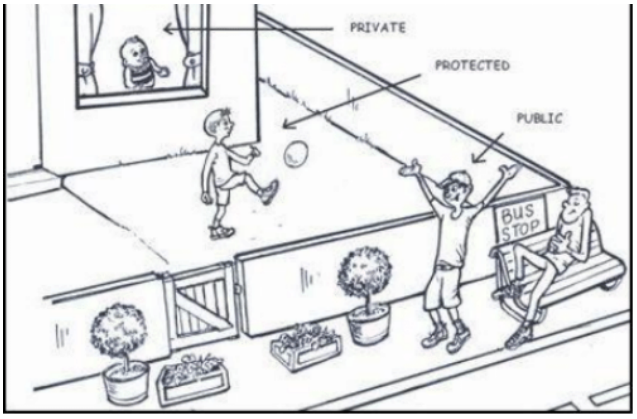
Хотя по умолчанию члены класса являются **private**, мы можем сделать их открытыми:

```
class DateClass
{
public: // обратите внимание, ключевое слово public и двоеточие
    int m_day; // открыто, доступ имеет любой объект
    int m_month; // открыто, доступ имеет любой объект
    int m_year; // открыто, доступ имеет любой объект
};

int main()
{
    DateClass date;
    date.m_day = 12; // ок, так как m_day имеет спецификатор доступа public
    date.m_month = 11;  // ок, так как m_month имеет спецификатор доступа public
    date.m_year = 2018;  // ок, так как m_year имеет спецификатор доступа public

    return 0;
}
```

Ключевое слово **public** является спецификатором доступа. В C++ есть три уровня доступа: **public**, **private**, **protected**. Спецификаторы доступа указывают, кто имеет доступ к членам класса. Спецификатор доступа **protected** (защищенный) рассмотрим далее.



В классе можно использовать несколько спецификаторов доступа. Как правило, для переменных-членов устанавливается доступ **private**, а для функций-членов — **public**. Порядок написания не имеет значения.

```
#include <iostream>

class DateClass // члены класса являются закрытыми по умолчанию
{
private:
    int m_day; // закрыто по умолчанию, доступ имеют только другие члены класса
    int m_month; // закрыто по умолчанию, доступ имеют только другие члены
класса
    int m_year; // закрыто по умолчанию, доступ имеют только другие члены класса

public:
    void setDate(int day, int month, int year) // открыто, доступ имеет любой
объект
    {
        // метод setDate() имеет доступ к закрытым членам класса, так как сам
является членом класса
        m_day = day;
        m_month = month;
        m_year = year;
    }

    void print() // открыто, доступ имеет любой объект
    {
        std::cout << m_day << "/" << m_month << "/" << m_year;
    }
};

int main()
{
    DateClass date;
    date.setDate(12, 12, 2018); // ок, так как setDate() имеет спецификатор
доступа public
    date.print(); // ок, так как print() имеет спецификатор доступа public

    return 0;
}
```

Мы не получим напрямую доступ к переменным класса, но можем через public-методы. Это основной механизм, который позволяет скрыть одно и открыть другое.

Следует отметить, что спецификаторы доступа работают на основе класса, то есть public-методы могут работать с private-переменными как данного объекта, так и любого другого:

```
#include <iostream>

class DateClass // члены класса являются закрытыми по умолчанию
{
private:  // данный спецификатор можно не указывать
    int m_day; // закрыто по умолчанию, доступ имеют только другие члены класса
    int m_month; // закрыто по умолчанию, доступ имеют только другие члены
класса
    int m_year; // закрыто по умолчанию, доступ имеют только другие члены класса

public:
    void setDate(int day, int month, int year)
    {
        m_day = day;
        m_month = month;
        m_year = year;
    }
    void print()
    {
        std::cout << m_day << "/" << m_month << "/" << m_year;
    }
    // Обратите внимание на этот дополнительный метод
    void copyFrom(const DateClass &b)
    {
        // Мы имеем прямой доступ к закрытым членам объекта b
        m_day = b.m_day;
        m_month = b.m_month;
        m_year = b.m_year;
    }
};

int main()
{
    DateClass date;
    date.setDate(12, 12, 2018); // ок, так как setDate() имеет спецификатор
доступа public

    DateClass copy;
    copy.copyFrom(date); // ок, так как copyFrom() имеет спецификатор доступа
public
    copy.print();

    return 0;
}
```

Функция **copyFrom()** имеет доступ к private-членам класса **m_day**, **m_month**, **m_year** — так как private-членам класса имеют доступ только функции, принадлежащие этому классу. Это означает, что она может напрямую обращаться как к закрытым членам неявного объекта, так и к членам объекта **b** (другого).

Инкапсуляция

Инкапсуляция (сокрытие информации) — это процесс скрытого хранения деталей реализации объекта. В C++ инкапсуляция реализована с помощью спецификаторов доступа. Инкапсулированные классы проще в применении и уменьшают сложность программ. Один и тот же класс можно использовать в разных программах, при этом не заботиться о том, как устроен класс изнутри: достаточно просто знать его методы. Также инкапсуляция защищает ваши данные и предотвращает неправильное использование. Рассмотрим класс с public-переменной-массивом:

```
class IntArray
{
public:
    int m_array[10];
};
```

Если бы пользователи напрямую обращались к массиву, могли бы использовать недопустимый индекс:

```
int main()
{
    IntArray array;
    array.m_array[16] = 2; // некорректный индекс
}
```

Но если мы сделаем массив закрытым, сможем заставить пользователя вызывать функцию, которая первым делом проверяет корректность индекса:

```
class IntArray
{
private:
    int m_array[10]; // пользователь не имеет прямой доступ к этому члену

public:
    void setValue(int index, int value)
    {
        // Если индекс недействителен, то не делаем ничего
        if (index < 0 || index >= 10)
            return;

        m_array[index] = value;
    }
};
```

Таким образом защитим программу от ошибок.

Инкапсулированные классы легче изменять и отлаживать, они проще в использовании, позволяют избегать ошибок и уменьшают сложность программы.

Set- и get-функции

В некоторых программах есть необходимость получать и изменять private-значения. Для этого разработчики пишут специальные public-функции для доступа к закрытым значениям. Для получения значения private-переменной (члена класса) используют **get-функцию**. Чтобы изменить ее значение — **set-функцию**.

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

public:
    int getDay() { return m_day; } // get-функция для day
    void setDay(int day) { m_day = day; } // set-функция для day

    int getMonth() { return m_month; } // get-функция для month
    void setMonth(int month) { m_month = month; } // set-функция для month

    const int& getYear() { return m_year; } // get-функция для year,возвращает
значение по константной ссылке
    void setYear(int year) { m_year = year; } // set-функция для year
};
```

- Пишите **set**- и **get**-функции только для тех классов, для которых это необходимо;
- Get-функции должны возвращать значения по значению или по константной ссылке, которая не дает права изменять значение переменной в функции **get()**. Не используйте неконстантные ссылки. Таким образом мы страхуемся от изменения значений.

Конструкторы

Конструктор — это особый тип метода класса, который автоматически вызывается при создании объекта этого класса. Конструкторы обычно используются для инициализации переменных-членов класса соответствующими значениями, которые предоставлены по умолчанию или пользователем. Еще они применяются для выполнения любых шагов настройки, необходимых для используемого класса.

В отличие от обычных методов, для конструкторов есть свои правила:

- они всегда должны иметь то же имя, что и класс (учитывается верхний и нижний регистры);
- у конструкторов нет типа возврата (даже void'a);
- конструкторы вызываются автоматически при создании объекта;
- нельзя вызвать конструктор для уже существующего экземпляра.

Конструктор, не имеющий параметров, называется **конструктором по умолчанию**. Чтобы создать объект с определенными значениями, используется **конструктор с параметрами**. В одном классе их может быть несколько — главное, чтобы их параметры были уникальными (учитывается количество и тип). Это возможно благодаря **перегрузке функций**.

```
#include <iostream>

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

public:
    void setDay(int day) { m_day = day; }
    int getDay() { return m_day; }
};

int main()
{
    Date date;
    date.setDay(7);
    std::cout << date.getDay() << '\n';
}
```

Старайтесь создавать хотя бы один конструктор, даже если он будет пустым.

Если один класс содержит другой в качестве переменной-члена, то конструктор внутреннего класса будет вызван раньше конструктора внешнего класса:

```
#include <iostream>

class Time
{
public:
    Time() { std::cout << "Time\n"; }
};

class Date
{
private:
    Time m_time; // Date содержит Time, как переменную-член

public:
    Date() { std::cout << "Date\n"; }
};

int main()
{
    Date date;
    return 0;
}
```

Результат выполнения данной программы:
Time
Date

При создании объекта **date** вызывается конструктор класса **Time**, а затем конструктор класса **Date**. В этом есть смысл, так как конструктор **Date()** может захотеть использовать переменную **m_time**, поэтому сначала нужно ее инициализировать.

Список инициализации членов класса

Константные типы данных должны быть инициализированы во время объявления. Если это константные переменные-члены, то инициализировать их с помощью конструктора нельзя. Для решения этой проблемы используются **списки инициализации членов**. Такой список располагается сразу после параметров конструктора и начинается с двоеточия (:), а затем в круглых скобках указывается значение для каждой переменной. Список инициализаторов не заканчивается точкой с запятой.

Пример использования списка инициализаторов членов:

```
class Date
{
private:
    const int m_day;
    const int m_month;
    const char* m_dayOfWeek; // День недели


public:
    Date(int day, int month, char* dayOfWeek="mon")
    : m_day(day), m_month(month), m_dayOfWeek(dayOfWeek) // напрямую
инициализируем переменные-члены класса
    {
        // Нет необходимости использовать присваивание
    }

    void print()
    {
        std::cout << "Date: " << m_day << ", " << m_month << ", " <<
m_dayOfWeek;
    }

};

int main()
{
    Date date(3, 5); // day = 3, month = 5, dayOfWeek = “mon” (значение по
умолчанию)
    date.print();
    return 0;
}
```

Результат: **Date: 3, 5, mon**

Используйте списки инициализации вместо операций присваивания, чтобы инициализировать переменные-члены вашего класса.

Переменные в списке инициализаторов инициализируются не в том порядке, в котором указаны в списке инициализации, а в котором объявлены в классе.

Инициализация членов класса

Если при написании класса с несколькими конструкторами указывать значения по умолчанию всем их членам, появится лишний код (DRY — Don’t Repeat Yourself). В таких случаях переменным-членам класса можно задать значение напрямую. При инициализации переменных через список инициализации членов приоритет будет отдан последнему.

Пример инициализации членов класса:

```
#include <iostream>

class Rectangle
{
private:
    double m_length = 3.5;
    double m_width = 3.5;

public:
    Rectangle() // конструктор по умолчанию (присваиваются значения по умолчанию выше)
    {
    }

    Rectangle(double length, double width)
    : m_length(length), m_width(width)
    {
        // m_length и m_width инициализируются конструктором (значения по умолчанию выше не используются)
    }

    void print()
    {
        std::cout << "length: " << m_length << " and width: " << m_width << '\n';
    }
};

int main()
{
    Rectangle one;
    one.print();
    Rectangle two(4.5, 5.5);
    two.print();
    return 0;
}
```

Программа выведет:

length: 3.5 and width: 3.5

length: 4.5 and width: 5.5

Деструкторы

Деструктор — это функция-член, которая выполняется при удалении объекта класса.

Если объект содержит динамически выделенную память, или файл, или базу данных, деструктор может освободить память перед уничтожением объекта.

Как и у конструкторов, у деструкторов есть правила, которые касаются их названий:

- деструктор должен иметь то же имя, что и класс, но со знаком «тильда» (~) перед ним;
- деструктор не может принимать аргументы (следовательно, не может быть перегружен);
- у деструктора нет типа возврата.

Рассмотрим пример программы, содержащей деструктор:

```
#include <iostream>
#include <cassert>

class Array
{
private:
    int *m_array;
    int m_length;

public:
    Array(int length) // конструктор
    {
        assert(length > 0);

        m_array = new int[length];
        m_length = length;
    }

    ~Array() // деструктор
    {
        // Динамически удаляем массив, который выделили ранее
        delete[] m_array ;
    }

    void setValue(int index, int value) { m_array[index] = value; }
    int getValue(int index) { return m_array[index]; }
    int getLength() { return m_length; }
};

int main()
{
    Array arr(15); // выделяем 15 целочисленных значений
    for (int count=0; count < 15; ++count)
        arr.setValue(count, count+1);
    std::cout << "The value of element 7 is " << arr.getValue(7);
    return 0;
} // объект arr
```

Результат выполнения: **The value of element 7 is 8**

В конце функции **main()** объект **arr** выходит из области видимости, что приводит к вызову деструктора и удалению массива.

Обратите внимание, что в данной программе используется оператор **assert** — это макрос препроцессора, который обрабатывает условное выражение во время выполнения. Если оно истинно, оператор **assert** ничего не делает. Если же оно ложное, то выводится сообщение об ошибке и программа завершается. Это сообщение содержит ложное условное выражение, а также имя файла с кодом и номером строки с **assert**. Таким образом можно легко понять, какая была проблема и где она возникла, что очень помогает при отладке программ.

Сам **assert** реализован в заголовочном файле **<cassert>** и часто используется для проверки корректности переданных параметров функции и ее возвращаемого значения.

Статические переменные-члены класса

В C++ ключевое слово **static** используется при создании статических переменных-членов и статических методов. Переменные-члены класса можно сделать статическими, используя ключевое

слово **static**. В отличие от обычных переменных-членов, статические являются общими для всех объектов класса. Рассмотрим следующую программу:

```
class Rectangle
{
public:
    static int m_square;
};

int Rectangle::m_square = 3;

int main()
{
    Rectangle first;
    Rectangle second;

    std::cout << first.m_square << '\n';
    first.m_square = 4;
    std::cout << second.m_square << '\n';
    return 0;
}
```

Результат:
3
4

Поскольку **m_square** — это статическая переменная-член, она является общей для всех объектов класса **Rectangle**. Следовательно, **first.m_square** — это та же переменная, что и **second.m_square**. Программа выше показывает, что к значению, которое мы установили через первый объект, можно получить доступ и через второй.

Хотя вы можете получить доступ к статическим членам через разные объекты класса (как в примере выше), оказывается, статические члены существуют, даже если объекты класса не созданы! Подобно глобальным переменным, они создаются при запуске программы и уничтожаются, когда программа завершает выполнение.

Следовательно, статические члены принадлежат классу, а не его объектам. Поскольку **m_square** существует независимо от любых объектов класса, то доступ к нему осуществляется напрямую через имя класса и оператор разрешения области видимости (в данном случае — **Rectangle::m_square**).

Обратите внимание: это определение статического члена не подпадает под действия спецификаторов доступа. Вы можете определить и инициализировать **m_square**, даже если оно будет **private**.

Практическое задание

1. Создать класс **Power**, который содержит два вещественных числа. Этот класс должен иметь две переменные-члена для хранения этих вещественных чисел. Еще создать два метода: один с именем **set**, который позволит присваивать значения переменным, второй — **calculate**, который будет выводить результат возведения первого числа в степень второго числа. Задать значения этих двух чисел по умолчанию.

2. Написать класс с именем RGBA, который содержит 4 переменные-члена типа **std::uint8_t: m_red**, **m_green**, **m_blue** и **m_alpha** (**#include cstdint** для доступа к этому типу). Задать 0 в качестве значения по умолчанию для **m_red**, **m_green**, **m_blue** и 255 для **m_alpha**. Создать конструктор со списком инициализации членов, который позволит пользователю передавать значения для **m_red**, **m_blue**, **m_green** и **m_alpha**. Написать функцию **print()**, которая будет выводить значения переменных-членов.

3. Написать класс, который реализует функциональность стека. Класс **Stack** должен иметь:

    - private-массив целых чисел длиной 10;
    - private целочисленное значение для отслеживания длины стека;
    - public-метод с именем **reset()**, который будет сбрасывать длину и все значения элементов на 0;
    - public-метод с именем **push()**, который будет добавлять значение в стек. **push()** должен возвращать значение **false**, если массив уже заполнен, и **true** в противном случае;
    - public-метод с именем **pop()** для вытягивания и возврата значения из стека. Если в стеке нет значений, то должно выводиться предупреждение;
    - public-метод с именем **print()**, который будет выводить все значения стека.

Код **main()**:

```
int main()
{
    Stack stack;
    stack.reset();
    stack.print();

    stack.push(3);
    stack.push(7);
    stack.push(5);
    stack.print();

    stack.pop();
    stack.print();

    stack.pop();
    stack.pop();
    stack.print();

    return 0;
}
```

Этот код должен выводить:

( )
(3 7 5)
(3 7)
( )

Дополнительные материалы

1. Бьерн Страуструп. Программирование. Принципы и практика использования C++.
2. Стивен Прата. Язык программирования C++. Лекции и упражнения.
3. Роберт Лафоре. Объектно-ориентированное программирование в C++.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Онлайн справочник программиста на C и C++. Инкапсуляция.
2. Бьерн Страуструп. Программирование. Принципы и практика использования C++.
3. Ральф Джонсон, Ричард Хелм, Эрих Гамма. Приемы объектно-ориентированного программирования. Паттерны проектирования.