



Объектно-ориентированное программирование на C++

Причины возникновения объектно-ориентированного программирования

**С ростом объема кода программы
становится невозможным
удерживать в памяти все детали**

**Необходимо структурировать
информацию, выделять главное и
отбрасывать несущественное**

Этот процесс называется
повышением степени абстракции
программы

**Первым шагом к повышению
абстракции является
использование функций**

Это позволяет отвлечься от
деталей ее реализации, поскольку
для вызова функции требуется
знать только ее интерфейс

**Следующий шаг — описание
собственных типов данных,
позволяющих структурировать и
группировать информацию**

**Процедурное программирование -
подход, при котором функции и
переменные, относящиеся к какому-то
конкретному объекту свободно
располагаются в коде и никак между
собой не связаны**

**Объектно-ориентированное
программирование -
подход, при котором функции и
переменные, относящиеся к
конкретному объекту объединены в
коде определенным образом и тесно
связаны между собой**

**В мире ООП всё состоит из
объектов**

**Программа представляет собой
набор объектов, имеющих
состояние и поведение**

**Концепция «черного ящика»
является одной из базовых
концепций ООП**

Снаружи объект принято
рассматривать как «черный ящик», т.
е. некий прибор с кнопками

Благодаря тому, что программа представляется в терминах поведения объектов, при программировании используются понятия, более близкие к предметной области

**Следовательно, программа легче
читается и понимается**

**ООП - это стиль программирования,
который фиксирует поведение
реального мира так, что детали
разработки скрыты**

Это позволяет программисту
мыслить в терминах предметной
области, а не в терминах
программирования

Основные понятия ООП

- Инкапсуляция
- Наследование
- Полиморфизм

**Инкапсуляция -
это объединение полей и методов
объекта в единое целое - класс**

Важнейшее требование
инкапсуляции - скрывание состояния
объекта от внешнего мира

Инкапсуляция повышает степень абстракции программы: данные класса и реализация методов класса находятся ниже уровня абстракции, и для написания программы информация о них не требуется

Инкапсуляция позволяет изменить реализацию класса без модификации основной части программы, если интерфейс остался прежним

Наследование -
это механизм, который позволяет
расширять существующие классы,
сохраняя их функциональность и
добавляя им новые свойства и методы

Полиморфизм -

это возможность использовать в различных классах иерархии одно название для обозначения сходных по смыслу действий и гибко выбирать требуемое действие во время выполнения программы

**Главный принцип полиморфизма –
один интерфейс и множество
реализаций**

**Класс - общее абстрактное
описание некоторой сущности**

Синтаксис объявления класса

```
class имя_класса
{
  [private | protected | public]:
  тип_поля1 имя_поля1;
  тип_поля2 имя_поля2;
  тип_поля3 имя_поля3;
  ...
  тип1 имя_метода1(список_параметров)
  {
    ...
  }

  тип2 имя_метода2(список_параметров)
  {
    ...
  }
  ...
} [список_переменных];
```

Способы доступа к компонентам класса

- Открытый (public)
- Защищенный (protected)
- Закрытый (private)

Пример объявления класса

```
class Date
{
private: // доступ к этим полям будут иметь только методы класса
    int year;
    int month;
    int day;
public: // открытые методы класса(интерфейсная часть класса)
    void put_date()
    {
        cout << "Year : ";
        cin >> year;
        cout << "Month : ";
        cin >> month;
        cout << "Day : ";
        cin >> day;
    }
    void print_date()
    {
        cout << "Date: " << year << "/"
            << month << "/" << day << endl;
    }
};
```

Объект как экземпляр класса – это некоторая уникальная единица, имеющая свои переменные (поля) и функции (методы), эти переменные обрабатывающие

Поля объекта - это переменные, описывающие его состояние, а методы - это способ перевести объект из одного состояния в другое

Пример создания объекта класса

```
void main()
{
    Date d; // создание объекта класса Date
    d.put_date(); // вызов метода класса Date put_date()
    d.print_date(); //вызов метода класса Date print_date()
}
```

Методы-аксессоры

- Инспекторы позволяют получить значения полей
- Модификаторы позволяют установить значения полей

Методы-аксессоры

```
class Date{
private: // доступ к этим полям будут иметь только методы класса
    int year;
    int month;
    int day;
public: // открытые методы класса(интерфейсная часть класса)

    void setYear(int y){ // модификатор для поля year
        year = y;
    }
    int getYear(){ // инспектор для поля year
        return year;
    }
    void setMonth(int m) { // модификатор для поля month
        month = m;
    }
    int getMonth() { // инспектор для поля month
        return month;
    }
    void setMonth(int d) { // модификатор для поля day
        month = d;
    }
    int getDay() { // инспектор для поля day
        return day;
    }
};
```

**Конструктор -
это специальный метод класса,
который вызывается для
конструирования объекта в момент
его создания**

Конструктор не возвращает
значение, даже типа `void`

**Класс может иметь несколько
конструкторов с разными
параметрами для разных видов
инициализации**

**Конструктор, вызываемый без
параметров, называется
конструктором по умолчанию**

Параметры конструктора могут
иметь любой тип, кроме этого же
класса

**Если программист не указал ни
одного конструктора, компилятор
создаст его автоматически**

**Деструктор -
это специальный метод класса,
который вызывается при
уничтожении объекта**

Деструктор не принимает
никаких параметров и не
возвращает значений

**Класс может иметь только один
деструктор**

Если деструктор явным образом
не определен, компилятор
автоматически создает пустой
деструктор