# Основы объектно-ориентированного программирования на C++

Объектно-ориентированное программирование, ООП – это одна из парадигм разработки, подразумевающая организацию программного кода, ориентируясь на данные и объекты, а не на функции и логические структуры. Объекты в подобном коде представляют собой полноценные блоки с данными, которые имеют определенный набор характеристик и возможностей. Объект может олицетворять что угодно – от человека с именем, фамилией, номером телефона, паролем и другой информацией до мелкой утилиты с минимумом характеристик из реального мира, но увеличенным набором функций. Объекты могут взаимодействовать друг с другом, пользователем и любыми другими компонентами программы. Объектно-ориентированный подход позволяет создавать программы модульным способом, что упрощает не только написание и понимание кода, но и обеспечивает более высокий уровень возможности повторного использования кода.

Объектно-ориентированное программирование основано на нескольких концепциях, таких как наследование, инкапсуляция, полиморфизм.

Инкапсуляция — это процесс скрытого хранения деталей реализации объекта . Инкапсуляция позволяет разделить реализацию объекта на две части: интерфейс и реализацию. Обращение к объекту происходит через открытый интерфейс. В C++, как правило, все переменные-члены класса являются закрытыми, скрывая детали реализации, а большинство методов являются открытыми, формируя открытый интерфейс для пользователя. Требование к пользователям использовать публичный интерфейс может показаться более обременительным, нежели просто открыть доступ к переменным-членам, но на самом деле это предоставляет большое количество полезных преимуществ, которые улучшают возможность повторного использования кода и его поддержку.

В рамках языка Си можно было написать подобную структуру:

```
struct str_complex{
 double re, im;
}
```

Также отдельно от это структуры можно было бы реализовать функцию вычисления модуля комплексного числа:

```
double mod(struct str_complex *c){
 return sqrt(c->re * c->re + c->im * c->im);
}
```

В языке C++ появляется возможность сделать тоже самое, но уже в объектно-ориентированном стиле:

```
struct str_complex{
 double re, im;
 double mod(){return sqrt(re*re+im*im);}
}
```

Функцию можно внести внутрь структуры, в таком случае использование этого кода может иметь следующий вид:

```
str_complex z;
z.re = 3.8;
z.im = 4.9;
double mod = z.mod();
```

Функция mod() называется методом структуры str_complex. Пример выше показывает, что метод вызывается не сам по себе, а для конкретного объекта. В примере метод mod() вызывается для объекта z, так что упоминаемые в его реализации поля re и im будут соответствовать полям z.re и z.im. Вызов метода на самом деле представляет собой абсолютно тоже самое, что и вызов функции, первым параметром которой является адрес объекта, для которого функция вызывается. К этому объекту возможно обратиться из функции с помощью специального ключевого слова this. По сути this можно воспринимать как локальную константу, имеющую тип A*, где A это имя описываемой структуры. Пример выше можно переписать с использованием  ключевого слова this.

```
struct str_complex{
 double re, im;
 double mod(){
  return sqrt(this->re*this->re + this->im*this->im);
 }
}
```

Чтобы стать полноценным объектом, структуре `str_complex` необходимо свойство закрытости, т.к. поля `re` и `im` сейчас доступны из любого места программы. Для того, чтобы скрыть детали реализации в языке C++ существует механизм защиты, который позволяет запретить доступ к некоторым полям и методам структуры из других мест. Для поддержания этого механизма были введены ключевые слова public и private, которыми могут быть помечены поля и методы. Например:

```
struct str_complex{
private:
 double re, im;
public:
 double mod(){return sqrt(re*re+im*im);}
}
```

Теперь поля `re` и `im` доступны только из тела метода mod() и пользоваться объектом структуры `str_complex` теперь невозможно, так как невозможно задать значения для этих полей. Решить проблему можно с помощью добавления нового метода, который будет ответственным за инициализацию полей.
Например:

```
struct str_complex{
private:
 double re, im;
public:
 void set( double a_re, double a_im ){
   re = a_re;
   im = a_im;
  }
 double mod(){return sqrt(re*re+im*im);}
}

…

str_complex z;
z.set( 3.8, 4.9 );
double mod = z.mod();
```

Такое решение имеет достаточно серьёзный недостаток. Дело в том, что объект z с момента объявления до момента вызова метода set оказывается в неопределённом состоянии, то есть попытки его использования будут заранее ошибочными. Кроме всего прочего принципы ООП предполагают, что вне объекта не следует делать предположения о его внутреннем состоянии, и для «неопределенного» состояния вряд ли стоит делать исключение. Очевидно правильным способом будет инициализировать объект в момент его создания и также следует запретить создание объекта в обход инициализации. Для этих целей вводится понятие конструктора объекта.

Конструктор - это метод специального вида, который может иметь параметры или не иметь их. Тело этого метода представляет собой порядок действий, которые необходимо выполнить при создании объекта данного типа. Компилятор отличает конструкторы от методов по имени, которое совпадает с именем описываемого типа. Ввиду особой роли конструктора накладывается ограничение на его прямой вызов — это возможно в C++, но весьма нежелательно. Пример использования конструктора:

```
struct str_complex{
private:
 double re, im;
public:
 str_complex( double a_re, double a_im ){
  re = a_re;
  im = a_im;
 }
 double mod(){return sqrt(re*re+im*im);}
}
```

Для получившейся выше структуры возможен следующий код:

```
str_complex z( 3.8, 4.9 );
double mod = z.mod();
```

После введения конструктора имеющего параметры, компилятор откажется создавать объекты типа `str_complex` без указания требуемых значений:

```
str_complex z;  // не скомпилируется
```

Чтобы вернуть возможность пользоваться такими конструкциями можно написать ещё один конструктор, только уже без параметров. В таком случае вышеописанный синтаксис более не будет помехой для компилятора, конструктор имеющий пустой список параметров называется конструктором по умолчанию.

```
struct str_complex{
private:
 double re, im;
public:
 str_complex( double a_re, double a_im ){
  re = a_re;
  im = a_im;
 }
 str_complex(){
  re = 0;
  im = 0;
 }
 double mod(){return sqrt(re*re+im*im);}
}

…

str_complex z;  // теперь скомпилируется
```

Для комплексного числа иногда необходимо определить отдельно его реальную и мнимую части. Для этого можно ввести отдельные методы:

```
struct str_complex{
private:
 double re, im;
public:
 str_complex( double a_re, double a_im ){
  re = a_re;
  im = a_im;
 }

 double get_re(){ return re; }
 double get_im(){ return im; }
 double mod(){return sqrt(re*re+im*im);}
 double argument(){ return atan2( im, re ); };
}
```

Кроме прочего иногда требуется хранить комплексное число в полярных координатах. Пример такой реализации:

```
struct str_complex{
private:
 double mod, arg;
public:
 str_complex( double a_re, double a_im ){
  mod = sqrt( re*re + im*im );
  arg = atan2( im, re );
 }
 double get_re(){ return mod * cos( arg ); }
 double get_im(){ return mod * sin( arg ); }
 double mod(){ return mod; }
 double argument(){ return arg; };
}
```

Возможно, иногда потребуется первая версия структуры, она будет работать быстрее за счет того, что в основном требуется получать мнимую и реальную части, а не модуль и аргумент, в таком случае будет иметь место меньше вычислений. Версии можно переключать например с помощью условий компиляции:

```
#ifdef FIRST
struct str_complex{
//  первая реализация
}
#else
struct str_complex{
//  вторая реализация
}
#endif
```

В итоге может быть, в итоге выгодней, что и те и те вычисления производить достаточно накладно, а памяти достаточно много. В таком случае может быть решено хранить как декартово, так и полярные представления числа.

```
struct str_complex{
private:
 double re, im, mod, arg;
public:
 str_complex( double a_re, double a_im ){
  re = a_re;
  im = a_im;
  mod = sqrt( re*re + im*im );
  arg = atan2( im, re );
 }
 double get_re(){ return re; }
 double get_im(){ return im; }
 double mod(){ return mod; }
 double argument(){ return arg; };
}
```

В таком случае в объекте будут 4 поля. Вызванный sizeof от объекта такой структуры вернет 4 * sizeof(double). В такой реализации 2 числа задают комплексное число, два остальных можно получить из двух первых. Можно сказать, что хранимая информация избыточна. Однако это может иметь место, но только в тех случаях, когда можно быть уверенным в том, что обеспечена целостность информации, т.е. гарантированно, что значения всех полей всегда будут находиться в зафиксированном для них соотношении. Для открытой структуры трудно было бы гарантировать целостность, поскольку в любом месте программы эти поля могли бы быть изменены.

Классы — это тип переменной, похожий на структуру, однако отличающийся тем, что к полям класса доступ по умолчанию есть только из методов самого этого класса.
Пример реализации комплексного числа в виде класса:

```
class Complex{
 double re, im, mod, arg;
public:
 Complex( double a_re, double a_im ){
  re = a_re;
  im = a_im;
  mod = sqrt( re*re + im*im );
  arg = atan2( im, re );
 }
 double get_re(){ return re; }
 double get_im(){ return im; }
 double mod(){ return mod; }
 double argument(){ return arg; };
}
```

В C++ разрешено перегружать конструктор, в связи с этим существует способ вызова одного конструктора из другого. Это явление называется делегирующим конструктором.

```
class Rectangle
{
    size_t m_Width;
    size_t m_Height;
public:
    Rectangle(size_t width, size_t height)
    {
        m_Width = width;
       m_Height = height;
    }
    Rectangle(size_t width): Rectangle(width, width)
    {
       // вызов делегирующего конструктора
    }
};
```

Стоит отметить, что часто приватные поля классов именуют начиная с "`m_`", по первой буква слова `member` (`data member prefix`).

В C++ есть возможность в конструкторе использовать так называемые цепочки инициализации (`member initialization list`). Цепочка инициализации позволяет инициализировать члены класса и базовые классы до выполнения тела конструктора. Это делает инициализацию более эффективной и обеспечивает гарантированное начальное состояние объекта.

```
class Rectangle
{
    size_t m_Width;
    size_t m_Height;
public:
    Rectangle(size_t width, size_t height):
m_Width(width), m_Height(height){}

};
```

Использование цепочки инициализации рекомендуется в C++, так как это обеспечивает более предсказуемое и эффективное поведение объектов и способствует читаемости и поддерживаемости кода.

В C++ существует понятие конструктора копирования — это специальный конструктор класса, который используется для создания нового объекта, который является копией существующего объекта того же класса. Конструктор копирования принимает в качестве аргумента другой объект того же типа и создает новый объект, идентичный переданному. В C++, конструктор копирования имеет следующую сигнатуру:

```
Rectangle( const Rectangle &other);
```

Конструктор копирования выполняет копирование всех членов класса из объекта `other` в новый объект. Обычно это копирование значений данных, выделение новой памяти (если объект содержит указатели), и выполнение других действий, необходимых для создания полной копии объекта. Конструктор копирования может иметь доступ к приватным членам `other`, для наиболее удобного копирования.

Оператор присваивания (`operator=`) в C++ — это специальная функция-член класса, которая позволяет присваивать один объект другому объекту того же класса. Оператор присваивания используется для копирования значений членов класса из одного объекта в другой. Обычно он вызывается в случае выполнения присваивания одного объекта другому, например: `rect1 = rect2;` или при копировании объекта в другой объект, `Rectangle rect1 = rect2;`. Сигнатура оператора присваивания обычно выглядит следующим образом:

```
Rectangle& operator=(const Rectangle &other);
```

Оператор присваивания должен возвращать ссылку на объект (`Rectangle&`), чтобы обеспечить возможность цепочки присваиваний (например, `a = b = c;`) и использования оператора присваивания в выражениях. Важно исключить ситуацию когда объект присваивается самому себе, чтобы избежать нежелательных побочных эффектов.

В C++, можно перегрузить (переопределить) множество операторов для пользовательских типов данных, чтобы предоставить более удобное и интуитивное поведение для объектов класса. Ниже представлен список некоторых операторов, которые можно перегрузить:

Оператор присваивания (=): Позволяет определить поведение при присваивании одного объекта другому. Это позволяет контролировать, какие данные будут скопированы и каким образом.
Операторы сравнения (==, !=, <, >, <=, >=): Позволяют определить поведение при сравнении объектов вашего класса. Это полезно для создания пользовательских типов данных, которые могут сравниваться между собой.
Операторы арифметических операций (+, -, *, /, %): Позволяют определить, как выполнять арифметические операции для объектов вашего класса. Это может быть полезно для создания математических типов данных.
Операторы инкремента и декремента (++, --): Позволяют определить, как изменяются объекты вашего класса при инкрементации и декрементации.
Операторы индексации ([]): Позволяют определить, как объекты вашего класса могут быть индексированы, например, как элементы массива.
Операторы доступа к членам (->, .): Позволяют определить, как доступ к членам объекта вашего класса будет работать.
Операторы ввода и вывода (<<, >>): Позволяют перегрузить операторы для работы с объектами вашего класса при вводе и выводе данных, что полезно для пользовательского форматированного вывода.
Операторы преобразования типов (typecast operators): Позволяют определить, как ваш объект может быть преобразован в другие типы данных.

Это лишь несколько примеров операторов, которые можно перегрузить. Какие именно операторы перегружать зависит от класса и того, какой функциональности необходимо достичь. При перегрузке операторов важно следовать правилам и ожидаемому поведению для этих операторов, чтобы код оставался интуитивным и безопасным.

Для перегрузки оператора вывода (<<) в C++ используется функция-член класса, которая принимает в качестве параметра объект `std::ostream` (обычно `std::ostream&`) и возвращает этот объект. Вот пример перегрузки оператора вывода для пользовательского класса:

```
friend std::ostream& operator<<(std::ostream& os,
const Rectangle& obj) {
    os << obj.m_Width << obj.m_Height;
    return os;
}
```

В этом примере `Rectangle` имеет перегруженный оператор вывода `operator<<`, который выводит значения полей класса `Rectangle` в поток `std::ostream`. Обратите внимание на использование ключевого слова `friend` для доступа к приватным членам класса `Rectangle`.

Деструктор в C++ — это специальная функция-член класса, которая вызывается автоматически при уничтожении объекта этого класса. Деструктор предназначен для освобождения ресурсов, выделенных объектом, и выполнения других завершающих операций перед уничтожением объекта. Синтаксис деструктора выглядит следующим образом:

```
~Rectangle() {
    // код деструктора
}
```

Деструктор не принимает аргументов и не имеет возвращаемого значения. Он вызывается автоматически, когда объект выходит из области видимости, или при вызове оператора `delete`, если объект был выделен динамически. Использование деструкторов важно для правильной работы с ресурсами, такими как динамически выделенная память, файловые дескрипторы и другие ресурсы, которые должны быть корректно освобождены при уничтожении объекта.

# Пространства имен

Конфликт имен возникает, когда два одинаковых идентификатора находятся в одной области видимости, и компилятор не может понять, какой из этих двух следует использовать в конкретной ситуации. Компилятор или линкер выдаст ошибку, так как у них недостаточно информации, чтобы решить эту неоднозначность. Как только программы увеличиваются в объемах, количество идентификаторов также увеличивается, следовательно, увеличивается и вероятность возникновения конфликтов имен.

Например два одинаковых идентификатора int doOperation(int a, int b), которые могут выполнять разные задачи но при этом иметь одинаковый интерфейс - набор аргументов и одинаковый тип возвращаемых значений.

```
int doOperation(int a, int b)
{
    return a + b;
}
```

```
int doOperation(int a, int b)
{
    return a - b;
}
```

…

```
std::cout << doOperation(5, 4) << std::endl; // какая версия doOperation будет выполнена ?
```

Пространство имен определяет область кода, в которой гарантируется уникальность всех идентификаторов. По умолчанию, глобальные переменные и обычные функции определены в глобальном пространстве имен. Обе версии `doOperation()` были включены в глобальное пространство имен, из-за чего и произойдёт конфликт имен.
Чтобы избежать подобных ситуаций, когда два независимых объекта имеют идентификаторы, которые могут конфликтовать друг с другом при совместном использовании, язык C++ позволяет объявлять собственные пространства имен через ключевое слово `namespace`. Всё, что объявлено внутри пользовательского пространства имен, — принадлежит только этому пространству имен, а не глобальному.

Теперь один `doOperation()` находится в пространстве имен `namespace Operations1`, а второй `doOperation()` в пространстве имен `namespace Operations2`.

```
namespace Operations1
{
    int doOperation(int a, int b)
    {
        return a + b;
    }
}
```

```
namespace Operations2
{
    int doOperation(int a, int b)
    {
        return a - b;
    }
}
```

Существует два разных способа сообщить компилятору, какую версию `doOperation()` следует использовать: через оператор разрешения области видимости или с помощью using-стейтментов:

```
// первый способ
std::cout << Operations1::doOperation(5, 4) <<
std::endl; // >> 9
std::cout << Operations2::doOperation(5, 4) <<
std::endl; // >> 1

// второй способ
using namespace Operations1;
std::cout << doOperation(5, 4) << std::endl; //
>> 9
```

Допускается объявление пространств имен в нескольких местах (либо в нескольких файлах, либо в нескольких местах внутри одного файла). Всё, что находится внутри одного блока имен, считается частью только этого блока.

```
namespace Operations1
{
    int add(int a, int b)
    {
        return a + b;
    }
}
```

```
…

namespace Operations1
{
    int subtract(int a, int b)
    {
        return a - b;
    }
}
```

Одни пространства имен могут быть вложены в другие пространства имен. Например:

```
namespace Operations
{
    namespace MathOperations
    {
        const double PI = 3.1415926535;

        int add(int a, int b)
        {
            return a + b;
        }

        int subtract(int a, int b)
        {
            return a - b;
        }

    }
}

…

std::cout << Operations::MathOperations::PI << std::endl;
std::cout << Operations::MathOperations::add(5, 4) << std::endl;
std::cout << Operations::MathOperations::subtract(5, 4) << std::endl;
```

Так как такой вложенный доступ не всегда удобен и эффективен, то C++ позволяет создавать псевдонимы для пространств имен:

```
namespace Math = Operations::MathOperations; // Math теперь считается как Operations::MathOperations

std::cout << Math::PI << std::endl;
```

Пространства имен в C++ не были разработаны, как способ реализации информационной иерархии, они нужны в качестве механизма предотвращения возникновения конфликтов имен. Как доказательство этому, вся Стандартная библиотека шаблонов находится в единственном пространстве имен std::.
Вложенность пространств имен не рекомендуется использовать, так как при неумелом использовании увеличивается вероятность возникновения ошибок и дополнительно усложняется логика программы.

# Ключевое слово const

Ключевое слово const — одно из самых многозначных в C++. Правильно использование const позволяет организовать множество проверок ещё на этапе компиляции и избежать многих ошибок из числа тех, которые бывает трудно найти при помощи отладчиков или анализа кода.

Самый простой случай — константные данные. Все они делают одно и тоже — создают переменную, значение которой изменить нельзя. Возможно несколько вариантов записи:

```
const int i(1);
int const j(1);
int const k = 1;
```

```
const int k = 1;
k = 7; // ошибка на этапе компиляции
```

При использовании const с указателями, действие модификатора может распространяться либо на значение указателя, либо на данные на которые указывает указатель.

Выполнится без ошибок компиляции:

```
const char * a = "a";
a = "b";
```

Тоже самое и тоже работает:

```
char const * a = "a";
a = "b";
```

В данных примерах изменяйся указатель, а не данные, по этому выражение возможно. Если бы операция присвоения изменяла бы не указатель, а данные, то была бы ошибка:

```
*a = ‘Y’; //ошибка
```

В следующем примере получаем ошибку компиляции, так как константа защищает указатель:

```
char * const a = "a";
a = "b"; // ошибка
```

Существует мнемоническое правило, позволяющее легко запомнить, к чему относится const. Надо провести черту через "*", если const слева, то оно относится к значению данных; если справа — к значению указателя.

С функциями ( имеются ввиду аргументы и результаты ) слово const используется по тем же правилам, что при описании обычных данных. Кроме всего перечисленного, const можно написать дважды:

```
const char * const s = "data";
```