БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ РАДИОФИЗИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ
КАФЕДРА ИНФОРМАТИКИ И КОМПЬЮТЕРНЫХ СИСТЕМ

Н. В. Серикова
Н. В. Левкович

ПРАКТИЧЕСКОЕ РУКОВОДСТВО
к лабораторному практикуму

КЛАССЫ

по дисциплине

«ПРОГРАММИРОВАНИЕ НА C++»

2024
МИНСК

Практическое руководство к лабораторному практикуму «КЛАССЫ» по дисциплине «ПРОГРАММИРОВАНИЕ НА С++» предназначено для студентов, изучающих базовый курс программирования на языке С++, специальностей «Компьютерная безопасность», «Прикладная информатика», «Радиофизика».

Руководство содержит некоторый справочный материал, примеры решения типовых задач с комментариями.

Автор будет признателен всем, кто поделится своими соображениями по совершенствованию данного пособия.

Возможные предложения и замечания можно присылать по адресу:
E-mail: Serikova@bsu.by

ОГЛАВЛЕНИЕ

ЭТАПЫ РАЗРАБОТКИ ООП СИСТЕМ

- **Объектно-ориентированный анализ:** исследование задачи с точки зрения объектов реального мира и определение требований к программной системе

- **Объектно-ориентированное проектирование:** разработка программных классов и логики их функционирования и взаимодействия

- **Объектно-ориентированное программирование:** реализация классов проектирования на выбранном языке программирования.

ДЕКОМПОЗИЦИЯ

При проектировании сложной программной системы необходимо разделять ее на подсистемы, каждую из которых можно совершенствовать независимо. Построив модели ограниченного числа подсистем, можно, комбинируя их различным образом строить множество гораздо более сложных систем.

Алгоритмическая декомпозиция

- Основана на разделении алгоритмов по модулям системы.
- Каждый модуль выполняет один из этапов общего процесса.
- Реализуется средствами структурного программирования

Объектно-ориентированная декомпозиция

- Мир представляется совокупностью автономно действующих объектов, моделирующих объекты реального мира.
- Каждый объект обладает своим собственным поведением.
- Послав объекту сообщение, можно попросить его выполнить присущее ему действие.
- Объекты взаимодействуют друг с другом, моделируя поведение системы, соответствующее более высокому уровню.
- Реализуется средствами объектно-ориентированного программирования.

ООП

Объектно-ориентированное программирование (ООП) – методология программирования, основанная на представлении программы в виде совокупности **объектов**, каждый из которых является **экземпляром определенного класса**, а классы образуют иерархию наследования.

Объект характеризуется:

- совокупностью своих элементов и их текущих значений
- совокупностью допустимых для объекта действий

Класс – определенный пользователем проблемно-ориентированный тип данных, описывающий внутреннюю структуру объектов, которые являются его экземплярами.

Объект (экземпляр класса) находится в таком же отношении к своему классу, в каком переменная находится по отношению к своему типу.

Данные и функции внутри класса называются **членами класса**.

Данные, входящие в класс, называются **данными-членами или полями**.

Функции, принадлежащие классу, называют **функциями-членами или методами**.

Объекты можно применять так же как и переменные встроенных типов, например, передавать в качестве параметра в функцию, получать в качестве результата, объединять в массивы, включать в структуры и другие классы.

Использование объекта одного класса в качестве поля другого класса называется **композицией**.

Каждый объект некоторого класса занимает определенное количество байтов памяти, где хранятся значения полей данных для этого объекта. Методы не занимают места в памяти объекта и хранятся в единственном экземпляре.

КОНЦЕПЦИИ ООП

1. Инкапсуляция – механизм, связывающий воедино программный код и данные, которыми он манипулирует, а также обеспечивающий их защиту от внешнего вмешательства и неправильного использования.

2. Полиморфизм – это свойство различных объектов выполнять одно и то же действие (с одним и тем же названием) по-своему.

3. Наследование – это такое отношение между объектами, когда дочерний объект повторяет элементы структуры и поведения родительского.

Классы верхних уровней обычно не имеют конкретных экземпляров объектов. Такие классы называют **абстрактными**. Конкретные экземпляры объектов относятся, как правило, к классам самых нижних уровней объектной иерархии.

СИНТАКСИС ОБЪЯВЛЕНИЯ КЛАССА

Синтаксис объявления класса, который не является наследником никакого другого класса:

```
class  Имя_Класса
 {
    спецификатор доступа:
       данные и функции
                  …
    спецификатор доступа:
       данные и функции
};
```

Спецификатор доступа определяет, где в программе будут доступны описанные за ним члены класса.

Имеются 3 спецификатора доступа:

- **Public** (общий) – члены класса будут доступны как в классе, так и в любой точке программы внутри области видимости класса, к которому они относятся.
- **Private** (личный - спецификатор по умолчанию) – члены класса будут доступны только внутри класса (членам класса) или друзьям класса
- **Protected** (защищенный) – члены класса будут доступны только внутри класса и внутри потомков класса

Действие спецификатора доступа распространяется до следующего спецификатора или до конца описания класса.

По умолчанию, все члены класса, объявленные после ключевого слова class до первого спецификатора доступа имеют спецификацию доступа private.

МЕТОДЫ КЛАССА

```
class  CBox                    //  объявление класса
{
public:                        //  спецификатор доступа
        double m_length;       //   поля        -    данные-члены класса
        double m_width;
        double m_height;
        double Volume( );      //  методы       -     функции-члены класса
};
```

Определение методов
Определение методов **либо внутри класса, либо вне класса**.

При внешнем определении в классе помещается только прототип метода**:**

тип имя_функции (**спецификация_и_инициализация_параметров**);

Определение метода вне класса:

тип **имя_класса::имя_функции** (спецификация_формальных_параметров)
{тело_функции_члена_класса}

Методы класса имеют неограниченный доступ ко всем элементам класса независимо ни от порядка объявления элементов класса, ни от спецификаторов доступа.

Методы можно перегружать. Перегрузка методов – одно из проявлений принципа полиморфизма.

ОБЪЯВЛЕНИЕ ОБЪЕКТОВ

Объявление класса является логической абстракцией, которая задает **новый тип объекта**. Объявление объекта создает физическую сущность объекта такого типа. То есть, объект занимает память, а задание типа нет.

Объявление экземпляров класса (объектов)

CBox mybox1, mybox2;

Каждый объект класса имеет собственную копию переменных, объявленных внутри класса.

Объявление указателей на класс

CBox *pbox;
pbox = &mybox1;

Динамическое выделение памяти для экземпляра класса (объекта)

CBox *pmybox = new CBox;

Доступ к членам – аналогично структурам

mybox1.m_length = 2.5;
mybox2.m_width = mybox1.m_length;
pmybox->m_height = mybox1.m_length;
pmybox->m_length = pmybox->m_height ;

КОНСТРУКТОР

Конструктор класса – специальная функция класса, которая вызывается при создании нового объекта класса. Она позволяет инициализировать объекты во время их создания и захватывать ресурсы, необходимые для их функционирования.

Конструкторы всегда называются по имени класса и не имеют типа возврата.

Нельзя получить указатель на конструктор.

Для глобальных объектов конструктор объекта вызывается тогда, когда начинается выполнение программы. Для локальных объектов конструктор вызывается всякий раз при выполнении инструкции объявления переменной.

Конструктору можно передавать аргументы. Тогда при объявлении объекта необходимые аргументы записываются в качестве параметров.

CBox mybox(1,2,3);

Конструктор копирования – конструктор, получающий в качестве единственного параметра указатель на объект этого же класса. Этот конструктор вызывается, когда новый объект создается путем копирования существующего:

- при описании нового объекта с инициализацией другим объектом;
- при передачи объекта в функцию по значению;
- при возврате объекта из функции.

Компилятор предоставляет два типа конструкторов:

- конструктор по умолчанию
- конструктор копирования:

Класс может иметь несколько конструкторов, которые можно перегружать.

Если Вы определили какой-либо **свой конструктор копирования**, Вы обязаны явно определить конструктор по умолчанию.

Стандартный конструктор копирования нельзя использовать при работе с указателями.

ДЕСТРУКТОР

Деструктор класса – специальная функция класса, которая уничтожает объект, когда необходимость в нем отпадает или время его жизни завершено.

Имя деструктора совпадает с именем класса, которому предшествует знак ~(тильда).

Деструктор не принимает параметров и не возвращает значения. Таким образом, деструктор в классе **всегда один**.

Компилятор предоставляет деструктор по умолчанию.

Однако, если Вы захватывали какие-либо ресурсы при создании объекта (например, динамически выделяли память), Вы обязаны переопределить деструктор для корректного освобождения ресурсов.

УКАЗАТЕЛЬ THIS

Указатель this – это скрытая внутренняя переменная каждого объекта, неявно используется внутри методов для ссылок на элементы объекта.

Указатель **this** инициализируется при определении объекта класса (каждый объект класса при создании получает значение указателя на начало, выделенной ему памяти).

***this** – операция разыменования для получения значения объекта.

Указатель **this** может использоваться только в нестатических компонентных функциях класса, причем использовать его необходимо без предварительного определения – он определяется транслятором автоматически. Указатель **this** хранит адрес объекта, для которого произведен вызов метода класса.

Выражение **this –> имя_члена** обеспечивает доступ к члену конкретного объекта (данному или функции) в приватной памяти объекта.

Дружественным функциям указатель **this** не передается.

Работа методов с полями именно того объекта, для которого они были вызваны, обеспечивается неявной передачей в функцию параметра **this**, в котором хранится константный указатель на вызвавший функцию объект.

Модификация указателя this в функциях класса недопустима.

СТАТИЧЕСКИЕ ЧЛЕНЫ КЛАССА

Глобальные данные и глобальные методы класса называют **статическими**.

Статические члены класса являются глобальными, принадлежат всему классу и существуют в единственном экземпляре.

В объявлении класса перед оператором описания типа глобальной компоненты записывается спецификатор **static**.

Статические данные класса полезны, когда необходимо, чтобы все объекты включали в себя какое-либо одинаковое значение.

Статические данные-члены класса

- являются частью класса, но не является частью объекта этого класса.
- объявляются внутри класса.
- должны быть определены вне класса.

Статические функции-члены класса

- функции, общие для всех объектов этого класса
- имеют доступ только к статическим членам класса
- для доступа к нестатическим членам они должны получить адрес объекта как параметр
- не могут быть константными и виртуальными

ДРУЖЕСТВЕННЫЕ ФУНКЦИИ И КЛАССЫ

Переменные–члены классов являются закрытыми, доступ к ним через методы класса.

Дружественные функции объявляются в описании класса с помощью ключевого слова **friend,** получают доступ к переменным-членам класса, сами не будучи его членами.

Дружественная функция становится расширением интерфейса класса, и этот интерфейс реализует взаимодействие объекта с другими объектами программы.

Дружественная функция – это не член класса и она не может быть задана через имя объекта. Она должна вызываться точно также как и обычная функция.

В качестве параметра такой функции должен передаваться объект или ссылка на объект класса, т.к. указатель *this* ей не передается.

Объявление дружественности может помещаться в любое место описания класса (спецификаторы доступа не имеют отношения к объявлению дружественности).

Одна функция может быть дружественной нескольким классам.

Прототип дружественной функции записывается в объявлении всех классов, с которыми она дружит.
Дружественная функция не наследуется.

Если все методы класса должны иметь доступ к скрытым полям другого, то весь класс объявляется дружественным.

ВЛОЖЕННЫЕ КЛАССЫ

Класс, объявленный внутри другого класса, называется **вложенным** классом.

Класс, в котором объявлен вложенный класс, называется **объемлющим** классом.

Функции- члены объемлющего класса **не имеют прав доступа к закрытым членам** своего вложенного класса.

Также и функции-члены вложенного класса **не имеют прав доступа к закрытым** членам класса, внутри которого он объявлен.

Чтобы предоставить такие права вложенному или объемлющему классу, нужно объявить такой класс другом соответственно объемлющего или вложенного класса.

ШАБЛОН КЛАССОВ

Шаблон классов – класс, в котором определены данные и методы, но фактический тип (типы) данных задаются в качестве параметра (параметров) при создании объекта класса.

Шаблоны классов позволяют многократно использовать один и тот же код, позволяющий компилятору автоматизировать процесс реализации типа.

Основные свойства шаблона классов:

- Шаблон позволяет передать в класс один или несколько типов в виде параметров.

- Параметрами шаблона могут быть не только типы, но и константные выражения.
- Объявление шаблона должно быть только глобальным.
- Статические члены-данные специфичны для каждой реализации шаблона.
- Спецификация и реализация шаблона классов при раздельной компиляции обязательно должны находиться в одном файле.

Объявление шаблона классов

Методы должны быть объявлены как шаблоны функций, поэтому заголовок метода, определение которого находится за пределами спецификации класса, имеет следующий формат:

template <class *Параметр_Типа1***, … , class** *Параметр_ТипаN***>**
Тип_функции Имя_шаблона **<** *Параметр_Типа1***, … ,** *Параметр_ТипаN***>**
::*Имя_функции* **(***список параметров функции***)**

Параметр_типа – вводимый пользователем идентификатор, который затем используется в реализации как имя типа. Если параметр – константное выражение, в списке параметров ключевое слово класс перед ним не указывается.

Параметры шаблона могут иметь **значения по умолчанию.** В этом случае в объявлении шаблона классов в угловых скобках после имени параметра типа ставится знак = , а за ним указывается значение по умолчанию.

Например, **template <class** *U = int, int Size = 100***>**

Объявление объектов шаблона классов

При объявлении переменных шаблона классов (объектов шаблона) создается конкретная реализация шаблона с типом, указанным в качестве параметра типа.

Объявление объекта шаблона классов имеет следующий формат:

Имя_Шаблона **<***Тип1***, … ,** *ТипN***>** *Имя_Объекта***;**

ПЕРЕГРУЗКА ОПЕРАТОРОВ

Операторы, как и функции, можно перегружать.

Перегружать можно все существующие в С++ операторы, кроме:

- . (точка, оператор доступа к члену класса),
- .*(оператор доступа к члену класса через указатель),
- :: (оператор разрешения области видимости,
- :? (условный оператор),
- #,## (препроцессорные операторы),
- sizeof и typeof,
- операторы преобразования типов данных: static_cast, const_cast, reinterpret_cast, dynamic_cast.

Перегруженные операторы можно определять и как члены класса, для которого выполняется перегрузка и как функции не члены класса (часто – дружественные функции).

Только методами класса допускается перегружать 4 оператора:

- = присваивание
- () вызов функции
- [] индексирование
- -> доступ по указателю

Операция присваивания (=) единственная создается системой автоматически.

Если оператор перегружен как член класса, то он не должен иметь спецификатор static и его первым операндом по умолчанию является объект класса, вызывающий этот оператор.

Один оператор может быть перегружен несколько раз для различных типов аргументов.

Используйте перегрузку операторов только там, где сохраняется их первоначальный смысл, и где это действительно необходимо.

Операторы, за исключением оператора =, наследуются производным классом. Могут перегружаться в производном классе.

Перегрузка операторов – проявление полиморфизма в С++. Перегрузка операторов является одним из видов перегрузки функций. Однако, при этом вводятся дополнительные правила.

Перегрузка операторов должна удовлетворять следующим требованиям:

- не допускается перегрузка операторов со встроенными типами данных в качестве параметров, тип, по крайней мере, одного параметра перегруженного оператора должен быть классом;
- **нельзя вводить новые операторы;**
- **нельзя изменять количество параметров оператора;**
- ассоциативность перегруженных операторов не изменяется;
- **приоритеты перегруженных операторов не изменяются;**
- **оператор не может иметь аргументов по умолчанию**, за исключением оператора вызова функции ();
- оператор не может иметь неопределенное количество параметров, за исключением оператора вызова функции ().

Как методы класса лучше перегружать операторы, у которых левый аргумент является текущим объектом (например, операции с присваиванием). Если же левым аргументом может быть объект, тип которого отличается от реализуемого типа, такую операцию лучше реализовывать как внешнюю.

Операция	**Рекомендуемая форма перегрузки**
Унарные операции	Методы класса
- [] () -> ->*	Обязательно методы класса
+= -= *= /= %= &= |= ^= <<= >>=	Метод класса
Остальные бинарные операции	Внешняя функция

В языке C++ операторы рассматриваются как функции, имеющие следующий прототип:

Имя_Типа **operator** @ **(***список_параметров***);**

Здесь символом @ обозначено имя оператора

Унарные операторы могут быть перегружены как нестатические члены класса без параметров:

Имя_Типа **operator** @ **( );**

либо как не члены класса с одним параметром:

Имя_Типа **operator** @ **(***Имя_Класса &Имя_Параметра* **);**

здесь @ обозначает один из следующих унарных операторов:

& * + - ~ !

Обращение к операции в этом случае выполняется двумя способами:

- **инфиксная форма**

```
@ параметр
```

- **функциональная форма**

```
оператор @ (параметр)
```

Бинарные операторы могут быть перегружены как нестатические члены класса с одним параметром – значением второго (правого) операнда:

Имя_Типа **operator** @ **(***Тип_Параметра Имя_Параметра***);**

либо как не члены класса с двумя параметрами, при этом левый операнд передается первому параметру, а правый - второму:

Имя_Типа **operator** @ **(***Тип_Параметра Имя_1, Тип_Параметра Имя_2)***;**

здесь @ обозначает один из следующих бинарных операторов:

+ - * / % == <= >= != && || << >>

Обращение к операции в этом случае выполняется двумя способами:

- **инфиксная форма**

```
параметр1 @ параметр2
```

- **функциональная форма**

```
оператор @ (параметр1, параметр2)
```

Оператор присваивания может быть перегружен **только как нестатический член класса** и должен иметь следующий прототип:

Имя_Класса& **operator** = **(const** *Имя_Класса &Имя_Параметра***);**

- Если оператор присваивания не определен в классе, то компилятор генерирует оператор присваивания по умолчанию, который выполняет почленное копирование атрибутов класса.
- Оператор присваивания целесообразно перегружать только в том случае, если дополнительно к копированию атрибутов нужно выполнить еще какие-либо действия. Как правило, это приходится делать, если объект работает с указателями и динамически выделяет память, чтобы избежать простого копирования указателей (см. ранее пример с конструктором копирования).
- При реализации оператора присваивания следует проверять возможность присваивания объекта самому себе.

Оператор индексирования может быть перегружен **только как нестатический член класса с одним параметром** – значением индекса:

Имя_Типа& **operator** [] = **(const int& i);**

В C++ при перегрузке оператор [] рассматривается как бинарный оператор.

Операторы инкремента ++ и декремента -- могут быть перегружены как члены класса без аргументов или как не члены класса с одним аргументом. Для того, чтобы префиксные операторы отличать от постфиксных, в объявлении последних вводят дополнительный фиктивный параметр типа int .

Имя_Класса **operator @ (); // префиксный**
Имя_Класса **operator @ (int); // постфиксный**

Имя_Класса **operator @ (***Имя_Класса &Имя_Параметра* **);**
Имя_Класса **operator @ (***Имя_Класса &Имя_Параметра ,***int);**

Оператор вызова функции ( ) может быть перегружен только как нестатический члены класса:

Имя_Типа **operator** @ **(***список_параметров***);**

Здесь количество параметров может быть произвольным, и допускается определять значения параметров по умолчанию. Вызывается оператор вызова функции путем применения списка фактических параметров к объекту класса, в котором он определен. Так как в этом случае объект может использоваться как функция, он иногда называется функциональным объектом.

Операторы преобразования типа (конвертор) –функция-член класса, которая преобразует тип объекта класса в некоторый другой тип. Конвертор имеет следующий прототип:

operator *Имя_Типа* **();**

Здесь *Имя_Типа* задает тип данных (встроенный или пользовательский), к которому приводится объект. Конвертер может вызываться как явно, так и неявно – при преобразованиях типов в выражениях, вызове функций и т.п.

Перегруженные операторы >> и << для ввода и вывода. Для перегрузки определяются как дружественные операторы класса, которые имеют следующие прототипы:

friend istream& operator>> (istream&, *Имя_Класса& Имя_параметра***);**
friend ostream& operator<< (ostream&, const *Имя_Класса& Имя_параметра***);**

АТД

В современных языках программирования пользователю разрешается определять свои типы, которые трактуются в языке практически так же, как встроенные. Такие типы обычно называют **абстрактными типами данных (АТД)**, хотя лучше, пожалуй, их называть просто пользовательскими.

Абстрактный тип данных (**АТД** – *abstract data type*) – это множество значений и совокупность операций над этими значениями этого типа, доступ к которым осуществляется только через интерфейс.

Для представления АТД **используются структуры данных**, которые представляют собой набор переменных, возможно, различных типов данных, объединенных определенным образом.

Создание АТД предполагает **полное сокрытие реализации** АТД в виде структур данных от пользователя.

Реализация АТД предполагает **расширение понятия операции над значениями типа данных с помощью интерфейсных процедур и функций**.

Важный инструмент, позволяющий добиться определения операций над АТД – перегрузка операций.

ПРИМЕР 1. Простейший класс. Объединение данных и методов

Создать объект «параллелепипед». Вычислить объем.
Поля объекта: длина, ширина, высота фигуры.
Первый принцип инкапсуляции: объединение данных и методов.

```
#include <iostream>  // for cin cout
using namespace std;
class  CBox                        //   объявление класса
{
public:                            //   спецификатор  доступа
     double m_length;              //  поле -  длина
     double m_width;               //  поле -  ширина
     double m_height;              //  поле -   высота
     double Volume( );//объявление метода – вычисление объема
};
          //  определение метода класса вне объявления класса
double CBox::Volume()
          { return m_length * m_width * m_height;}

int main()
{   CBox mybox1;
         // создание статического объекта (экземпляра класса)
// поля – public  -  нарушение второго принципа инкапсуляции
     mybox1.m_length = 2;      //  инициализация  полей  !!!
     mybox1.m_width  = 3;
     mybox1.m_height = 4;
     cout << mybox1.Volume() << endl; //вычисление объема =24

                    //создание динамического объекта
 // поля – public - нарушение второго принципа инкапсуляции
     CBox *pmybox2 = new CBox;
     pmybox2->m_length = 5;      //  инициализация  полей  !!!
     pmybox2->m_width = 6;
     pmybox2->m_height = 7;
                               // вычисление объема =210
     cout << pmybox2->Volume() << endl;
     delete pmybox2;    // удаление динамического объекта
     return 0;
}
```

ПРИМЕР 2. Доступ к данным через интерфейс

Пример тот же.

Второй принцип инкапсуляции: защита от внешнего вмешательства. Доступ к данным через интерфейс.

```
#include <iostream>  // for cin cout
using namespace std;

class  CBox            //   объявление класса
{               //  по умолчанию спецификатор доступа private
     double m_length;            //  поле -  длина
     double m_width;             //  поле -  ширина
     double m_height;            //  поле -   высота

public:        //   спецификатор  доступа
               //  определение методов класса при объявлении
               // инициализация полей через методы класса
     void Setlength(double sl)
                 {m_length = sl; }
     void Setwidth(double sw)
                  {m_width = sw; }
     void Setheight(double sh)
                   {m_height = sh; }
              // доступ к полям через методы класса
     double Getlength()
                 {return m_length;}
     double Getwidth()
                  {return m_width;}
     double Getheight()
                   {return  m_height;}

     double Volume( );//объявление метода – вычисление объема
  // объявление перегруженного метода  для вычисления объема
     double Volume(CBox *);
};
        //  определение метода класса вне объявления  класса
double CBox::Volume()
      {return m_length * m_width * m_height;}

             // определение метода класса Volume()
             // для демонстрации указателя  this
double CBox::Volume(CBox * )
   {return this->m_length * this->m_width * this->m_height;}

int main()
```

```
{      // создание статического объекта (экземпляра класса)
     CBox mybox1;

//   mybox1.m_length = 2;      //  ошибка  !!!
//   mybox1.m_width  = 3;      //  ошибка  !!!
//   mybox1.m_height = 4;      //  ошибка  !!!

              // инициализация полей  через методы класса
     mybox1.Setlength(2);
     mybox1.Setwidth (3);
     mybox1.Setheight(4);
     cout<<mybox1.Getheight()<<"  "<< mybox1.Volume()<< endl;
                                                    // 4   24
     cout << mybox1.Volume(&mybox1) << endl;
                                                    //    24
                         // создание динамического объекта
     CBox *pmybox2 = new CBox;
                    // инициализация полей  через методы класса
     pmybox2->Setlength(5);
     pmybox2->Setwidth(6);
     pmybox2->Setheight(7);

    cout<<pmybox2->Getlength()<<" "<<pmybox2->Volume()<<endl;


     delete pmybox2;      // удаление динамического объекта
     return 0;
}
```

ПРИМЕР 3. Независимость интерфейса от реализации

Пример тот же. Меняем описание полей, определение методов. Главная программа не изменяется.

Независимость интерфейса от реализации, разрешаются только операции, определенные через интерфейс.

```
#include <iostream>  // for cin cout
using namespace std;

class  CBox                    //   объявление класса
{
              //  по умолчанию спецификатор доступа private
    double m_Size[3];              //  поле -  массив

public:    //  определение методов класса при объявлении
    void Setlength(double sl)
                {m_Size[0] = sl; }
    void Setwidth(double sw)
                 {m_Size[1] = sw; }
    void Setheight(double sh)
                  {m_Size[2] = sh; }

                    // доступ к полям через методы класса
    double Getlength()
                   { return m_Size[0]; }
    double  Getwidth()
                    { return m_Size[1]; }
    double Getheight()
                    { return  m_Size[2]; }
    double Volume( );        // метод – вычисление объема
};

        //  определение метода класса вне объявления класса

double CBox::Volume()
         { return m_Size[0] * m_Size[1] * m_Size[2]; }
```

```
                                    //  программа не изменяется
int main()
{
        // создание статического объекта (экземпляра класса)
   CBox mybox1;
                // инициализация полей  через методы класса
   mybox1.Setlength(2);
   mybox1.Setwidth(3);
   mybox1.Setheight(4);
   cout<<mybox1.Getheight()<<" "<<mybox1.Volume()<< endl;
                                          // 4   24
                        // создание динамического объекта
   CBox *pmybox2 = new CBox;

            // инициализация полей  через методы класса
   pmybox2->Setlength(5);
   pmybox2->Setwidth(6);
   pmybox2->Setheight(7);
   cout<<pmybox2->Getlength()<<" "<<pmybox2->Volume()<<endl;
   delete pmybox2;       // удаление динамического объекта
   return 0;
}
```

ПРИМЕР 4. Конструктор, деструктор

Пример тот же. Добавляем конструктор, десруктор. Определение конструкторов при объявлении класса.

```
#include <iostream>  // for cin cout
using namespace std;

class  CBox                    //   объявление класса
{         //  по умолчанию спецификатор доступа private
     double m_length;                //  поле -  длина
     double m_width;                 //  поле -  ширина
     double m_height;                //  поле -   высота
public:
    CBox()//конструктор для инициализации полей значениями =0
      {  m_length = 0;  m_width = 0;  m_height = 0; }

        //  перегруженный конструктор
        //  для инициализации двух полей значениями,
        //  третье поле инициализируется 0
    CBox(double l, double w)
      { m_length = l;  m_width = w;  m_height = 0;}

       //  перегруженный конструктор  для инициализации полей
    CBox(double l, double w, double h)
      { m_length = l;  m_width = w;  m_height = h; }

      ~CBox ()                          //  деструктор
          { cout << "destructor CBox done"<<endl;}

    void Print ()
      { cout<<" L = "<<m_length<<" W = "<<m_width<<
              " H="<<m_height<<endl;
       }

     double Volume( )
       { return m_length * m_width * m_height; }
};
```

```
void main()
{           // вызов конструктора для инициализаци полей
            // значениями 0 0 0
     CBox mybox1;
     mybox1.Print();
               // вызов конструктора для инициализаци полей
               // значениями 1 2 3
     CBox mybox2(1, 2, 3);
     mybox2.Print();
              // вызов конструктора для инициализаци полей
              // значениями 1 2 0
     CBox mybox3(1, 2);
     mybox3.Print();

               // вызов стандартного конструктора копирования
     CBox mybox4 = mybox2;
     mybox4.Print();
           // вызов оператора =
     mybox1 = mybox2;
     mybox1.Print();
}            // вызов четырех деструкторов
```

ПРИМЕР 5. Определение методов вне класса

Пример тот же. Определение конструкторов вне объявления класса.

```
#include <iostream>  // for cin cout
using namespace std;

class  CBox              //   объявление класса
{           //  по умолчанию спецификатор доступа private
    double m_length;           //  поле -  длина
    double m_width;            //  поле -  ширина
    double m_height;           //  поле -   высота

public:
     //  конструктор для инициализации полей значениями =0
    CBox();

     // перегруженный конструктор для инициализации двух
     // полей значениями, третье поле инициализируется 0
    CBox(double l,double w);

      //  перегруженный конструктор  для инициализации полей
    CBox(double l, double w, double h);

    ~CBox ();                          //  деструктор

    void Print ()
    { cout <<" L = "<< m_length << "  W = "<< m_width <<
             " H = "<<m_height<<endl;
     }
    double Volume( )
    { return m_length * m_width * m_height; }
};

              //  определение конструкоров вне класса
CBox:: CBox() : m_length(0), m_width(0), m_height(0)  { }

CBox::CBox(double l,double w):
            m_length(l), m_width(w), m_height(0) { }

CBox:: CBox(double l, double w, double h):
            m_length(l), m_width(w), m_height(h){ }
                                    //  деструктор
CBox:: ~CBox ()
    {cout << "destructor CBox done"<<endl; }
```

```
void main()
{
//вызов конструктора для инициализаци полей значениями 0 0 0
     CBox mybox1;
     mybox1.Print();

//вызов конструктора для инициализаци полей значениями 1 2 3
     CBox mybox2(1, 2, 3);
     mybox2.Print();

//вызов конструктора для инициализаци полей значениями 1 2 0
     CBox mybox3(1, 2);
     mybox3.Print();

               // вызов стандартного конструктора копирования
     CBox mybox4 = mybox2;
     mybox4.Print();

               // вызов стандарного конструктора копирования
     mybox1 = mybox2;
     mybox1.Print();
}                    // вызов деструктора
```

ПРИМЕР 6. Явный и неявный вызов деструктора

```
#include "stdafx.h"
#include <iostream>
using namespace std;

class CMyStr
{
  char* m_pStr;            // указатель на строку
  int   m_nLength;         // длина строки
public:
  CMyStr() : m_nLength(0)  // конструктор по умолчанию
  {
    m_pStr = new char[1];
    m_pStr[0] = 0;
  }
  ~CMyStr();
};

CMyStr::~CMyStr()        // деструктор
{
  delete[] m_pStr;     // освобождение памяти
  m_pStr = nullptr;
  cout<<"destructor"<<endl;
}

int main(int argc, char* argv[])
{
  CMyStr* pStr = new CMyStr();
  for (int i = 0; i < 5; i++)
  {
    cout << i;
    CMyStr s;
  }      // автоматический вызов деструктора для объекта s

  delete pStr;  // явный вызов деструктора для объекта *pStr
                // и освобождение памяти
  return 0;
}
```

ПРИМЕР 7. Константные поля, методы и объекты

```
#include <iostream>  // for cin cout
using namespace std;

class  CBox                          //   объявление класса
{
     double m_length;              //  поле -  длина
     double m_width;               //  поле -  ширина
     double m_height;              //  поле -   высота

   // инициализируется конструктором нельзя изменить методом
     const int n;                  // число размерностей

public:
  CBox();//конструктор для инициализации полей значениями =0
  CBox(double l,double w);    //  перегруженные конструкторы
  CBox(double l, double w, double h);
  ~CBox ();                      //  деструктор

  void Set_l(double l)    { m_length = l;}
  void Set_w(double w)    { m_width = w;}
  void Set_h(double h)    { m_height = h;}

        // константные методы не могут изменять поля класса
  double Get_l() const { return m_length;}
  double Get_w() const { return m_width;}
  double Get_h() const { return m_height;}

  void Print () const
   {cout<< n << " L = " << m_length << " W = " << m_width <<
                 " H = "<< m_height << endl; }
   double Volume( )  const
    { return m_length * m_width * m_height; }
};
                 //  в определении конструкторов
                 // обязательна инициализация поля-константы
CBox:: CBox() :
     m_length(0), m_width(0), m_height(0),n(3)  { }

CBox:: CBox(double l,double w):
     m_length(l), m_width(w), m_height(0),n(3) { }

CBox:: CBox(double l, double w, double h) :
                 m_length(l), m_width(w), m_height(h),n(3){
}
```

```
CBox:: ~CBox ()           //  деструктор
    { cout << "destructor CBox done"<<endl; }


void main()
{
    CBox mybox1(1, 2, 3);
    mybox1.Print ();
    mybox1.Set_l (4);
    mybox1.Print();
    cout<< mybox1.Get_l() <<endl;

    const CBox mybox2(5, 6, 7);   // константный объект
    // mybox2.Print();            // запрещено !!!
    // mybox2.Set_l(4);           // запрещено !!!
    // mybox2.Print();            // запрещено !!!
    //  cout<<mybox2.Get_l()<<endl;   // запрещено !!!

    CBox mybox3 = mybox2;         //  но  !!!
    mybox3.Print();
}
```

ПРИМЕР 8. Стандартный конструктор копирования

Использование стандартного конструктора копирования.

```
#include <iostream>  // for cin cout
using namespace std;

class CPoint
{
    short  m_a;           // поле  1
    short  m_k;           // поле  2
public:
    CPoint()
        {m_a = 4;   m_k = 1;}      // конструктор
    ~CPoint()   {}                 // деструктор
    void Setk(short k) {m_k=k;}    // инициализация   поля 2
    void Seta(short a)  {m_a=a;}   // инициализация поля 1
    void Print ()           //  вывод значений полей на экран
    { cout << " a = "<< m_a << " k = " << m_k <<endl;}
};

int main()
{    // вызов конструктора – инициализация полей объекта x
    CPoint  x;
    x.Print();              // 4 1
          // вызов стандартного конструктора  копирования
    CPoint y = x;
    y.Print();              // 4 1
    x.Setk(5);              //  инициализация поля 2
    x.Seta(8);              //  инициализация поля 1
    x.Print();              // 8 5
    y.Print();              // 4 1
    return 0;
}
```

ПРИМЕР 9. Стандартный конструктор копирования не работает

Стандартный конструктор копирования нельзя использовать, если Вы работаете с членами – указателями.

```
#include <iostream>  // for cin cout
using namespace std;

class CPoint
{
     short   m_a;              // поле  1
     short   *m_k;             // поле  2
public:
     CPoint()                  // конструктор
       { m_a = 4;
         m_k = new short(1);
        }
     ~CPoint()    {}           // деструктор
     void Setk(short k)
        {*m_k = k;}            // инициализация значения поля 2
     void Seta(short a)
         {m_a = a; }           // инициализация поля 1
     void Print ()             //  вывод значений полей на экран
     {  cout<<" a = "<<m_a<<" k = "<< *m_k<<" "<<m_k<<endl; }
};

int main()
{       // вызов конструктора - инициализация полей объекта x
     CPoint  x;
     x.Print();       // 4 1
             // вызов стандартного конструктора  копирования
     CPoint y = x;
     y.Print();       // 4 1
     x.Setk(5);       //  инициализация значения поля 2
     x.Seta(8);       //  инициализация поля 1
     x.Print();       //  8 5
     y.Print();       //  4 5  !!!   но должно быть 4 1

     return 0;
}
```

ПРИМЕР 10. Собственный конструктор копирования

Определение собственного конструктора копирования.

```
#include <iostream>  // for cin cout
using namespace std;

class CPoint
{    short   m_a;           // поле  1
     short   *m_k;          // поле  2
public:
     CPoint()               // конструктор
      { m_a = 4;
        m_k = new short(1);
      }
              // собственный конструктор копирования
     CPoint (const CPoint  &p)
     { m_a = p.m_a;  m_k = new short(*p.m_k); }
     ~CPoint(){}              // деструктор
     void Setk(short k)
       {*m_k = k;}           // инициализация значения поля 2
     void Seta(short a)
        {m_a = a;}           // инициализация поля 1
     const void Print ()     //  вывод значений полей на экран
     {cout << " a="<<m_a<<"  k=" << *m_k<<"  " << m_k<<endl;}
};

int main()
{             // вызов конструктора - инициализация полей
     CPoint  x;
     x.Print();         // 4 1
            // вызов  собственного конструктора  копирования
     CPoint y (x);
     y.Print();    // 4 1
     x.Setk(5);    // инициализация значения поля 2
     x.Seta(8);    // инициализация поля 1
     x.Print();    // 8 5
     y.Print();    // 4  1
     return 0;
}
```

ПРИМЕР 11. Стандартный конструктор копирования не работает

```
#include <iostream>
using namespace std;

class CMyStr
{
  char* m_pStr;       // указатель на строку
  int   m_nLength;    // длина строки
public:
  CMyStr() : m_nLength(0)    // конструктор по умолчанию
  { m_pStr = new char[1];
    m_pStr[0] = 0;
  }
     //  конструктор с параметром
   CMyStr(const char* pStr)
  {  m_nLength = pStr ? strlen(pStr) : 0;
     //Реализация strlen вылетает, если ей передать nullptr
     // добавляем проверку
     m_pStr = new char[m_nLength + 1];
       //Резервируем дополнительное место на концевой ноль
    if (pStr)
      strcpy_s(m_pStr, m_nLength + 1, pStr);
    // второй параметр - размер памяти для строки,
    // если попытаться скопировать в m_pStr больше -
    // программа вылетит с ошибкой
     else
       m_pStr[0] = 0;
  }
    // отображает адрес расположения_строки в памяти и её содержимое
 void Print() const
  {
  cout << reinterpret_cast<void*>(m_pStr) << ": " << m_pStr << endl;
  }
  ~CMyStr();
};

CMyStr::~CMyStr()     // деструктор
{
  delete[] m_pStr;
  m_pStr = nullptr;
  cout<<"destructor"<<endl;
}
```

```
int main(int argc, char* argv[])
{
    CMyStr s("any string");
    s.Print();

    CMyStr s2 = s;
          // После конструктора копирования_по умолчанию
         //  два объекта ссылающихся на одну и ту же область памяти
    s2.Print();
    return 0;
// В этом месте будут вызваны деструкторы обоих объектов (s и s2).
// Как следствие, программа завершится с ошибкой: попытка освободить
// одну и ту же память дважды.
}
```

ПРИМЕР 12. Конструктор и оператор копирования

```
#include <iostream>
using namespace std;
class CMyStr
{
  char* m_pStr;       // указатель на строку
  int   m_nLength;    // длина строки
public:
  CMyStr() : m_nLength(0)    // конструктор
  {
    m_pStr = new char[1];
    m_pStr[0] = 0;
  }
                  //  Конструктор с параметрами
   CMyStr(const char* pStr)
{ m_nLength = pStr ? strlen(pStr) : 0;
  m_pStr = new char[m_nLength + 1];
  if (pStr)
    strcpy_s(m_pStr, m_nLength + 1, pStr);
  else   m_pStr[0] = 0;
}
                    //  Конструктор копирования
CMyStr(const CMyStr& str)
{ m_nLength = str.m_nLength;
  m_pStr = _strdup(str.m_pStr);  // копия строки
}

// Оператор копирования перегружает операцию присвоения
//объекта, когда объект-назначение уже создан ранее.
// Реализация аналогична конструктору копирования:
// присваиваемое значение передаётся по константной ссылке
CMyStr& operator = (const CMyStr& str)
{   delete[] m_pStr;
 // Не забываем, что объект уже выделил память для
  // какой-то другой строки и её нужно освободить
    m_nLength = str.m_nLength;
    m_pStr = _strdup(str.m_pStr);
    return *this;
// Для возможности каскадного присвоения возвращаем ссылку на текущий
// объект,хотя можно возвращать результат любого другого типа или void
  }

   // Дополнительно можно перегрузить оператор копирования
   CMyStr& operator = (const char* pStr)
  {
    delete[] m_pStr;
    m_nLength = pStr ? strlen(pStr) : 0;
```

```
    m_pStr = new char[m_nLength + 1];
    if (pStr)
      strcpy_s(m_pStr, m_nLength + 1, pStr);
    else
      m_pStr[0] = 0;
    return *this;
  }

// отображает адрес расположения_строки в памяти и её содержимое
 void Print() const
{
cout<<reinterpret_cast<void*>(m_pStr)<< ": "<<m_pStr<<endl;
}
  ~CMyStr();
};

CMyStr::~CMyStr()     // деструктор
{ delete[] m_pStr;
  m_pStr = nullptr;
  cout<<"destructor"<<endl;
}

CMyStr GetMagic()    // внешняя функция
{    return CMyStr("magic");
}



// Пример использования оператора копирования
int main()
{
  CMyStr s("any string");      // конструктор из строки
  CMyStr s2("another string"); // конструктор из строки
  CMyStr sCopy1(s);             // конструктор копирования
  CMyStr sCopy2 = s;       // тоже конструктор копирования
  sCopy1 = s2;             // оператор присвоения
  sCopy2 = GetMagic();     // конструктор объекта в функции
                       // потом оператор копирования в main
  return 0;
}
```

ПРИМЕР 13. Конструктор и оператор перемещения

```
#include <iostream>
using namespace std;
class CMyStr
{
    char* m_pStr;       // указатель на строку
    int   m_nLength;    // длина строки
public:
    CMyStr() : m_nLength(0)    // конструктор
    {   m_pStr = new char[1];
        m_pStr[0] = 0;
    }
                        //  Конструктор с параметрами
    CMyStr(const char* pStr)
    {   m_nLength = pStr ? strlen(pStr) : 0;
        m_pStr = new char[m_nLength + 1];
        if (pStr)
            strcpy_s(m_pStr, m_nLength + 1, pStr);
        else  m_pStr[0] = 0;
    }
                         //  Конструктор копирования
    CMyStr(const CMyStr& str)
    {   m_nLength = str.m_nLength;
        m_pStr = _strdup(str.m_pStr);  // копия строки
    }
                        // Оператор копирования
    CMyStr& operator=(const CMyStr& str)
    {
        delete[] m_pStr;
        m_nLength = str.m_nLength;
        m_pStr = _strdup(str.m_pStr);
        return *this;
    }
                // перегруженный оператор копирования
    CMyStr& operator=(const char* pStr)
    {
        delete[] m_pStr;
        m_nLength = pStr ? strlen(pStr) : 0;
        m_pStr = new char[m_nLength + 1];
        if (pStr)
            strcpy_s(m_pStr, m_nLength + 1, pStr);
        else     m_pStr[0] = 0;
        return *this;
    }
```

```
                // конструктор перемещения
                // Rvalue-ссылка объявляется с &&
    CMyStr(CMyStr&& str)
    {
        m_nLength = str.m_nLength;
        m_pStr = str.m_pStr;
        str.m_pStr = nullptr;
        str.m_nLength = 0;
    }
                // оператор перемещения
                // Rvalue-ссылка объявляется с &&
    CMyStr& operator=(CMyStr&& str)
    {   swap(m_nLength, str.m_nLength);
        swap(m_pStr, str.m_pStr);
                    // обмениваем буферы двух объектов
        return *this;
    }
    void Print() const
    {  cout <<  m_pStr << endl;
    }
    ~CMyStr();
};
CMyStr::~CMyStr()     // деструктор
{   delete[] m_pStr;
    m_pStr = nullptr;
    cout << "destructor" << endl;
}

CMyStr SomeFunc(const char* pStr)
{
    // если функция возвращает объект по значению,
    // то при его передаче будет использован
    // конструктор перемещения или оператор перемещения
    // (естественно если они объявлены и реализованы)
    CMyStr ss(pStr);
    ss.Print();
    return ss;
}


int main()
{
    CMyStr s("any string");
    CMyStr s2("another string");
    s.Print();
```

```
    s2.Print();
    cout << endl;

    s = move(s2); // явный вызов оператора перемещения
    s.Print();
    s2.Print();
    cout << endl;

    CMyStr s3 = move(s); // явный вызов конструктора перемещения

    s = SomeFunc("test"); // неявный вызов оператора перемещения,
                          // поскольку объект возвращается
                          // из функции по значению
    return 0;
}
```

ПРИМЕР 14. Статические члены класса

В класс добавим статическую переменную **m_noboxes** и статический метод **Getnoboxes()**, позволяющий вернуть значение статической переменной.

Определение статической переменной обязательно вне класса.

Статические функции-члены класса имеют доступ только к статическим членам класса. Для доступа к нестатическим членам они должны получить адрес объекта как параметр

```
#include <iostream>  // for cin cout
using namespace std;

class  CBox                    //   объявление класса
{
     double m_length;          //  поле - длина
     double m_width;           //  поле - ширина
     double m_height;          //  поле - высота
     static int m_noboxes;     //  поле - статическая переменная

public:
     CBox()                                    //  конструктор
     {m_length = 0; m_width = 0; m_height = 0; m_noboxes++;}
                               //  перегруженный конструктор
     CBox(double l,double w)
     {m_length = l; m_width = w; m_height = 0; m_noboxes++; }
                              //  перегруженный конструктор
     CBox(double l, double w, double h)
     {m_length = l; m_width = w; m_height = h; m_noboxes++;}

     ~CBox ()                                  //  деструктор
     {
          cout << "destructor CBox done" << endl;
          m_noboxes--;
     }

     void Set_l(double l)     { m_length = l;}
     void Set_w(double w)     { m_width = w; }
     void Set_h(double h)     { m_height = h;}

     const double Get_l()     { return m_length;}
     const double Get_w()     { return m_width; }
     const double Get_h()     { return m_height;}

     const void Print()
     { cout << " L = " << m_length << " W = " << m_width <<
               " H = " << m_height << endl;}
```

```
    const double Volume( )
         { return m_length * m_width * m_height; }

    //   статические функции-члены класса имеют доступ
    //   только к статическим членам класса.
    static int Getnoboxes()
    {  return m_noboxes;  }

    //  для доступа к нестатическим членам они должны
    //   получить адрес объекта как параметр
    static int Getnoboxes(CBox & C)
    {   C.Set_l(0);  return m_noboxes;  }
};

// определение статической переменной обязательно вне класса
int CBox::m_noboxes=0;

      // значение статической переменной в примере равно
      // количеству объектов описанного класса
void main()
{
    CBox mybox1;                                  //  1 объект
    mybox1.Print();
    cout << mybox1.Getnoboxes() << endl;

    CBox mybox2(1, 2);                           //  2 объект
    mybox2.Print();
    cout << mybox1.Getnoboxes() << endl;

    CBox mybox3(3, 4, 5);                        //  3 объект
    mybox3.Print();
    cout << CBox::Getnoboxes(mybox3) << endl;
    mybox3.Print();
}
```

ПРИМЕР 15. Статические члены класса

```
#include <iostream>
using namespace std;
class CMyStr
{
    char* m_pStr;       // указатель на строку
    int   m_nLength;    // длина строки
    static int g_StrCnt;
    // Статические поля класса объявляются внутри класса
    //  Проинициализированы они должны быть вне объявления класса

public:
    CMyStr() : m_nLength(0)    // конструктор
    {
        m_pStr = new char[1];
        m_pStr[0] = 0;
        g_StrCnt++;
    }
                        //  Конструктор с параметрами
    CMyStr(const char* pStr)
    {
        m_nLength = pStr ? strlen(pStr) : 0;
        m_pStr = new char[m_nLength + 1];
        if (pStr)
            strcpy_s(m_pStr, m_nLength + 1, pStr);
        else
            m_pStr[0] = 0;
        g_StrCnt++;
    }
                          //  Конструктор копирования
    CMyStr(const CMyStr& str)
    {
        m_nLength = str.m_nLength;
        m_pStr = _strdup(str.m_pStr);  // копия строки
        g_StrCnt++;

    }
  // Статические методы класса - вызываются без указания конкретного объекта
  // не получают указатель this - не имеют доступа к не статическим полям и методам.
  // Для доступа к нестатическим членам они должны получить адрес объекта как параметр.
    static int GetGlobalStrCnt()
    {
        return g_StrCnt;
    }
```

```
    void Print() const
    {
        cout <<  m_pStr << "  "<< g_StrCnt<<endl;
    }

    ~CMyStr();
};

CMyStr::~CMyStr()     // деструктор
{
    delete[] m_pStr;
    m_pStr = nullptr;
    g_StrCnt--;
    cout << "destructor" << endl;
}
int CMyStr::g_StrCnt;
        // или   int CMyStr::g_StrCnt = 0;
        // Если иное не указано, то статическая переменная
        //  инициализируется нулём


int main()
{
    CMyStr s("any string");
    s.Print();
    cout << CMyStr::GetGlobalStrCnt() << endl;
    CMyStr s2("another string");
    s2.Print();
    cout << CMyStr::GetGlobalStrCnt() << endl;
    CMyStr s3(s2);
    s3.Print();
    cout << CMyStr::GetGlobalStrCnt() << endl;
    return 0;
}
```

ПРИМЕР 16. Управление ресурсом

Пример демонстрирует использование операторов копирования и перемещения для корректного управления указателем на массив.

```
     // класс CMyArray обеспечивает управление ресурсом(указателем на массив)
class CMyArray
{
private:
    int* m_Ptr;
                    // оператор копирования -запрещён,
                    // чтобы избежать утечки памяти
                    // при присвоении поверх другого указателя
    void operator=(const CMyArray&);
    CMyArray(const CMyArray&);
public:
    CMyArray(int* ptr) :   m_Ptr(ptr)    {}
    CMyArray(CMyArray&& other) : m_Ptr(other.m_Ptr)
    {
        other.m_Ptr = nullptr;
    }
                  // деструктор освобождает память
    ~CMyArray()
    {         delete[] m_Ptr;     }
                 // оператор перемещения позволяет
                 // возвращать объекты CMyArray из функций
    void operator=(CMyArray&& other)
    {
        swap(m_Ptr, other.m_Ptr);
    }
            // метод get обеспечивает доступ к указателю
    int* get()
    {     return m_Ptr;     }
};

// Функция GetMagic возвращает массив с "правом владения",
// то есть явным указанием, что его надо после использования удалить.
CMyArray  GetMagic()
{
    int* pArr = new int[16];
    for (int i = 0; i < 16; i++)
        pArr[i] = rand();
    return pArr;
    // оператор return неявно вызывает конструктор CMyArray
}
```

```
int DoAction()
{
    CMyArray pArr = GetMagic();
    CMyArray pOtherArr = GetMagic();

  // Вся память, управляемая с помощью CMyArray,
  // освобождается автоматически при выходе из функции
  // (или при выходе из области видимости соответствующих переменных)
    if (pArr.get()[0] == 3)
        return 1;
    if (pArr.get()[1] == pOtherArr.get()[1])
        return 2;
    if (pArr.get()[2] == pOtherArr.get()[2])
        return 3;
    return 0;
}

int main()
{
    srand(time(NULL));
    cout << DoAction() << endl;
    return 0;
}
```

ПРИМЕР 17. Массив объектов

Пример тот же. Создаем массив объектов данного класса.

```
#include <iostream>  // for cin cout
using namespace std;
class  CBox                    //   объявление класса
{    double m_length;          // поле -  длина
     double m_width;           // поле -  ширина
     double m_height;          // поле -  высота
     static  int m_noboxes;    // поле - статическая переменная
public:
     CBox()                                  //  конструктор
     {m_length = 0; m_width = 0; m_height = 0; m_noboxes++; }

     CBox(double l,double w)   //  перегруженный конструктор
     {m_length = l; m_width = w; m_height = 0; m_noboxes++; }
                               //  перегруженный конструктор
     CBox(double l, double w, double h)
     {m_length = l; m_width = w; m_height = h; m_noboxes++; }
     ~CBox ()                  //  деструктор
     {    cout << "destructor CBox done"<<endl; }

     void Set_l(double l)     { m_length=l;}
     void Set_w(double w)     { m_width=w;}
     void Set_h(double h)     { m_height=h;}
     const double Get_l()     { return m_length;}
     const double Get_w()     { return m_width;}
     const double Get_h()     { return m_height;}
     const void Print ()
     { cout<<" L = "<<m_length<<" W = "<< m_width<<" H = "<<
               m_height<<" m_noboxes = "<<m_noboxes<<endl;}
     const double Volume( )
       { return m_length * m_width * m_height; }

     //   статические функции-члены класса имеют доступ
     //   только к статическим членам класса.
     static int Getnoboxes() {return m_noboxes;}
     //  для доступа к нестатическим членам они должны
     //   получить адрес объекта как параметр

     static int Getnoboxes(CBox &C)
          {C.Set_l(0);  return m_noboxes;}
};
// определение статической переменной обязательно вне класса
int CBox::m_noboxes=0;
```

```
// значение статической переменной в примере равно
// количеству объектов  описанного класса

void main()
{            //  массив объектов. конструктор по умолчанию!
     CBox mybox[4];
     cout<<" massiv1  "<<endl;
     for(int i = 0;i < 4; i++)
     {
          cout<<i<<endl;
          mybox[i].Print();
     }
                  //  другой массив объектов. инициализация
     CBox mybox1[3];
     mybox1[0]=CBox();
     mybox1[1]=CBox(1,2);
     mybox1[2]=CBox(3,4,5);

     cout<<" massiv2  "<<endl;
     for(int i = 0; i < 3; i++)
     {
          cout<<i<<endl;
          mybox1[i].Print();
     }
             //  или так  массив объектов. инициализация
     CBox mybox2[3]={CBox(),CBox(1,2),CBox(3,4,5)};

     cout<<" massiv3  "<<endl;
     for(int i = 0; i < 3; i++)
     {
          cout<<i<<endl;
          mybox2[i].Print();
     }
}
```

ПРИМЕР 18. Вложенные классы

Пример объявление вложенного класса.

```
#include <iostream>  // for cin cout
using namespace std;

class Outer                 //  объявление объемлющего класса
{
    class Inner             //  объявление вложенного класса
    {
        int n;
    public:
        Inner(){ n = 0;}    // конструктор вложенного класса
        int count()
           { return ++n; } //  метод вложенного класса
    };

    Inner n;
public:
                            //  метод объемлющего класса
    int count()
        { return n.count(); }
};

int main()
{

    Outer n;
    cout<<n.count()<<endl;              //  1!!!

    // Inner m;                         //  нельзя!!!
    // cout<<m.count()<<endl;

    //  Outer:: Inner m;                // нельзя!!!
    //  cout<<m.count()<<endl;

    return 0;
}
```

ПРИМЕР 19. Вложенные классы

Пример объявление вложенного класса доступного вне объемлющего класса.

```
#include <iostream>  // for cin cout
using namespace std;

class Outer              //  объявление объемлющего класса
{
public:
     class Inner         //  объявление вложенного класса
     {
          int n;
     public:
          Inner()
            { n = 0;}          // конструктор вложенного класса
          int count()
             { return ++n; }   //  метод вложенного класса
     };
private:
     Inner n;
public:
     int count()
        {return n.count();}      //  метод объемлющего класса
};

int main()
{

     Outer n;
     cout<<n.count()<<endl;                    // 1!!!

     //Inner m;                              //  нельзя!!!
     //cout<<m.count()<<endl;

     Outer:: Inner m;
     cout<<m.count()<<endl;                    //  1!!!

     return 0;
}
```

ПРИМЕР 20. Шаблон классов

Шаблон классов для одномерного массива размерности Size.

```
#include <iostream>       // cin cout
using namespace std;

template <class U = int, int Size = 100>   // шаблон классов
class MyArray                              // объявление класса
{
    U m_array [Size];
public:
    MyArray(void)                          //     конструктор
    { for (int i = 0; i <= Size - 1; i++)
        m_array[i]=0;
     }
    ~MyArray(void) {}                      // деструктор
                // метод получения значения i элемента
    U Get (int i);
              //  метод инициализации i элемента значением x
    void Put (int i, U x);
              //  метод вывода элементов массива на экран
    void Print()
    {
       for (int i = 0; i < Size; i++)
           cout << m_array[i] << "   " ; cout<<endl;
     }
};
                                       //  Определения методов
template <class U, int Size>           //  шаблон метода
U  MyArray < U, Size>::Get (int i)
{
    if (i < 0)
       i = 0;
    if (i > Size - 1)
       i = Size - 1;
    return m_array[i];
}


template <class U, int Size>          //  шаблон метода
void  MyArray < U, Size>::Put (int i, U x)
{
    if (i < 0)
       i = 0;
    if (i > Size - 1)
```

```
      i = Size - 1;
    m_array[i]=x;
}


int main()
{
          //объект - массив из 5 вещественных элементов
    MyArray <double, 5> darray;

          // объект - массив из 100 целых элементов
    MyArray <> iarray;

    cout<<" darray = "<<endl;
    darray.Print();
    cout<<" iarray = "<<endl;
    iarray.Print();

    darray.Put(1,3.14);
    iarray.Put(0,2);
    darray.Put(0,iarray.Get(0));
    cout<<" darray = "<<endl;
    darray.Print();

    return 0;
}
```

ПРИМЕР 21. Перегрузка операторов

Пример шаблона классов CVecn, позволяющего работать с n-мерными векторами, заданными своими координатами в прямоугольной системе координат. Внутренне представление этого вектора – одномерный массив элементов типа double размером n элементов.

```
#include <iostream>  // for cin cout
using namespace std;

template <int Size = 3>
class CVecn                          // объявление класса
{
    double m_array[Size];           // массив – поле класса
public:
                                     // конструктор
    CVecn (void)
     {for (int i = 0; i < Size; i++)
         m_array[i]=0;
      }
    ~CVecn (void) {}                 // деструктор
    CVecn operator - ();             //  унарный –
                            //  сложение векторов с замещением
    CVecn operator +=(const CVecn &r);
                            //  сложение векторов
    CVecn operator +(const CVecn &r);
                           //  вычитание векторов с замещением
    CVecn operator -=(const CVecn &r);
                           //  вычитание векторов
    CVecn operator -(const CVecn &r);
                          //  скалярное произведение
    double operator *(const CVecn &r);
                          //  умножение вектора на скаляр
    CVecn operator *(double l);
                          //  умножение скаляра на вектор
    friend CVecn operator *(double l,CVecn &r)
    {  return r*l; }
                 // логическая операция равенства векторов
    bool operator ==(const CVecn &r);
                 // логическая операция неравенства векторов
    bool operator !=(const CVecn &r);
//  оператор индексирования
//  позволяет обращаться к элементам массива
//  не нужны функции для доступа к полям (типа Set или Get).
    double& operator [] (const int i);
};
```

```
                                    //  унарный -
template <int Size>
CVecn<Size> CVecn<Size>::operator -()
{
     CVecn <Size> Result;
     for (int i = 0; i < Size; i++)
          Result.m_array[i] =- m_array[i];
     return Result;
}
                                    //  оператор индексирования
template <int Size>
double& CVecn <Size>::operator [](const int i)
{
     if (i < 0)          return m_array[0];
     if (i >= Size)      return m_array[Size - 1];
     return m_array[i];
}
                          //  сложение векторов с замещением
template <int Size>
CVecn <Size> CVecn <Size>::operator +=(const CVecn<Size> &r)
{
     for (int i = 0; i < Size; i++)
          m_array[i] += r.m_array[i];
     return *this;
}
                                     //  сложение векторов
template <int Size>
CVecn <Size> CVecn <Size>::operator +(const CVecn<Size> &r)
{
     CVecn <Size> Rezult;
     for (int i = 0; i < Size; i++)
          Rezult.m_array[i] = r.m_array[i] + m_array[i];
     return Rezult;
}
                               //вычитание векторов с замещением
template <int Size>
CVecn <Size> CVecn <Size>::operator -=(const CVecn<Size> &r)
{
     for (int i = 0; i < Size; i++)
          m_array[i] -= r.m_array[i];
     return *this;
}
                                  // вычитание векторов
template <int Size>
CVecn <Size> CVecn <Size>::operator -(const CVecn<Size> &r)
{
```

```
    CVecn <Size> Rezult;
    for (int i = 0; i < Size; i++)
        Rezult.m_array[i] = r.m_array[i] - m_array[i];
    return Rezult;
}
                                //  скалярное произведение
template <int Size>
double CVecn <Size>::operator *(const CVecn<Size> &r)
{
    double sum = 0;
    for (int i = 0; i < Size; i++)
        sum += m_array[i] * r.m_array[i];
    return sum;
}
                            //  умножение вектора на скаляр
template <int Size>
CVecn <Size> CVecn <Size>::operator *(double l)
{
    CVecn <Size> Result;
    for (int i = 0; i < Size; i++)
        Result.m_array[i] = m_array[i] * l;
    return Result;
}
            // логическая операция равенства векторов
template <int Size>
bool CVecn <Size>::operator ==(const CVecn<Size> &r)
{
    for (int i = 0; i < Size; i++)
        if (m_array[i] != r.m_array[i])
           return false;
    return true;
}
           // логическая операция неравенства векторов
template <int Size>
bool CVecn <Size>::operator !=(const CVecn<Size> &r)
{
    for (int i = 0; i < Size; i++)
        if (m_array[i] != r.m_array[i])
           return true;
    return false;
}

/*              // можно перегрузить оператор =

template <int Size>
CVecn<Size>& CVecn <Size>::operator =(const CVecn <Size> &r)
```

```
{
   if (&r !=this)
     for (int i = 0; i < Size; i++)
        m_array[i] = r.m_array[i];
    return *this;
}
*/
                          //  внешняя функция
                          //  вывод элементов массива на экран
template <int Size>
void Print(CVecn <Size> vect)
{
    for (int i = 0; i < Size; i++)
        cout << vect[i] << "   " ;
    cout<<endl;
}




int main()
{   const SIZE = 5;
                          // объявляем 3 объекта-вектора
    CVecn  <SIZE> vector1, vector2, vector3;
                          // инициализация 1 вектора
    for (int i = 0; i < SIZE; i++)
       vector1[i] = i;
    cout<<" vector1 =  "<<endl;
    Print(vector1);
                                  // унарный минус
    cout<<" -vector1=  "<<endl;
    Print(-vector1);
                                // инициализация 2 вектора
    for (i = 0; i < SIZE; i++)
        vector2[i] = i + 5;
    cout<<endl<<" vector2 = "<<endl;
    Print(vector2);
                        // сложение двух векторов с замещением
    vector1 += vector2;
    cout<<endl<<" vector1 += vector2  "<<endl;
    Print(vector1);
                        // сложение двух векторов
    vector3 = vector1 + vector2;
```

```
    cout<<endl<<" vector1 + vector2 =  "<<endl;
    Print(vector3);
                              // умножение вектора на скаляр
    vector1 = vector1 * 2;
    cout<<endl<<" vector1 *= 2  "<<endl;
    Print(vector1);
                               // умножение скаляра на вектор
    vector1 = 2 * vector2;
    cout<<endl<<" vector2 * 2 =  "<<endl;
    Print(vector1);
                            // сравнение на равенство векторов
    cout<<endl<<"vector1 == vector2 = "<<
              (vector1 == vector2)<<endl;
                    // скалярное произведение двух векторов
    cout<<endl<<" scalarnoe=  "<<(vector1 * vector2)<<endl;
    return 0;
  }
```

ПРИМЕР 22. АТД - «комплексное число»

Создадим АТД для комплексных чисел.

```
                // файл complex.h    объявление класса
class CCmplx
{
    double m_re;  // действительная часть комплексного числа
    double m_im;  // мнимая часть комплексного числа

public:
                // конструктор
    CCmplx (double re = 0, double im = 0):
            m_re(re), m_im(im){}
    ~CCmplx(void){}                  //  деструктор
    double & Re()
          {return m_re;}             //  действ. часть
    double & Im()
          {return m_im;}            //  мнимая часть
    double& operator [] (int i);    //  оператор []
                                    //  оператор ()
    CCmplx& operator ()(double x = 0, double y = 0)
       {  m_re = x;    m_im = y;
          return *this;
        }
    CCmplx& operator = (const CCmplx &r);   //  оператор =
    CCmplx& operator = (double r);     //  оператор =
    CCmplx& operator -  ();           //  оператор - унарный
    CCmplx& operator +  ();           //  оператор + унарный
    CCmplx& operator ++ ();              //  prefix
```

```
CCmplx operator ++ (int);          //  postfix
CCmplx operator + (const CCmplx &r) const;
                        // + два комплексных числа
CCmplx operator - (const CCmplx &r) const;
                        // - два комплексных числа
CCmplx operator *( const CCmplx &r) const;
                         // * два комплексных числа
CCmplx operator /( const CCmplx &r) const;
                       // /два комплексных числа
CCmplx operator + (double r); //комплексное + вещественное
CCmplx operator - (double r); // комплексное - вещественное
CCmplx operator * (double r); // комплексное * вещественное
CCmplx operator / (double r); // комплексное / вещественное
       // +  два комплексных  числа с замещением
CCmplx& operator += (const CCmplx &r);
         // -  два комплексных  числа с замещением
CCmplx& operator -= (const CCmplx &r);
         // *  два комплексных  числа с замещением
CCmplx& operator *=( const CCmplx &r);
        // /  два комплексных  числа с замещением
CCmplx& operator /=( const CCmplx &r);
        // компл  число + вещ.  с  замещением
CCmplx& operator += (double r);
           // компл  число - вещ.  с  замещением
CCmplx& operator -= (double r);
          // компл  число * вещ.  с  замещением
CCmplx& operator *= (double r);
              // компл  число / вещ.  с  замещением
CCmplx& operator /= (double r);
             // сравнение компл. чисел на равенство
bool operator ==( const CCmplx &r);
             // сравнение компл. чисел на неравенство
bool operator !=( const CCmplx &r);
           // сравнение компл. числа с вещ. на равенство
bool operator ==(double r);
    // сравнение компл. числа с вещ. на неравенство
bool operator !=(double r);
                    //   вещ.  +   компл.  число
friend CCmplx operator +(double l, CCmplx &r )
{  r.m_re = l + r.m_re;
   return r;
 }
                    //   вещ.  -   компл.  число
friend CCmplx operator -(double l, CCmplx &r )
{ r.m_re = l - r.m_re;
  r.m_im = -r.m_im;
```

```
  return r;
 }
                  //   вещ.  *   компл.  число
friend CCmplx operator *(double l, CCmplx &r )
{ r.m_re = l * r.m_re;
  r.m_im = l * r.m_im;
  return r;
  };
                  //   вещ.  /   компл.  число
friend CCmplx operator /(double l, CCmplx &r )
{ CCmplx z(l,0);
   r = z / r;
   return r;
 }
                        //   вещ.  +=   компл.  число
friend double operator +=(double l, CCmplx &r )
{ l = l + r.m_re;
  return l;
 }
                      //   вещ.  -=   компл.  число
friend  double operator -=(double l, CCmplx &r )
{ l = l - r.m_re;
  return l;
 }
                  //   вещ.  *=   компл.  число
friend double operator *=(double l, CCmplx &r )
{ l = l * r.m_re;
  return l;
 };
                   //   вещ.  /=   компл.  число
friend double operator /=(double l, CCmplx &r )
{ CCmplx z(l,0);
   z = z / r;
   l = z.Re();
   return l;
 }
    // сравнение вещ.  числа с компл. на равенство
friend bool operator ==(double l, const CCmplx &r)
{ return (r.m_re == l) && (r.m_im == 0)? true : false;}
        // сравнение вещ.  числа с компл. на неравенство
friend bool operator !=(double l, const CCmplx &r)
{return (r.m_re != l) || (r.m_im != 0) ? true : false;}



        // оператор >> для ввода  компл. числа
```

```
    friend istream& operator >> (istream &str, CCmplx &r)
    {   double re = 0, im = 0;
         str >> re >> im;
        r = CCmplx(re, im);
        return str;
    }
            // оператор  << для вывода  компл. числа
    friend ostream& operator << (ostream& str, CCmplx &r)
    {  if (r.m_im < 0) return str<<r.m_re<<r.m_im<<"i";
       else   return str << r.m_re << "+" << r.m_im << "i";
      }
};
      //  файл  complex.cpp       определение методов
#include <iostream>  // for cin cout
using namespace std;
#include "complex.h"
// если < …>  файл следует искать в «стандартных include»
// если  "…"  файл следует искать в текущем каталоге
double& CCmplx:: operator [] (int i)   //   оператор []
{ if (i <= 1)   return m_re;
  else return m_im;
}
                                       //  оператор =
CCmplx& CCmplx:: operator = (const CCmplx &r)
{ m_re = r.m_re;   m_im = r.m_im;
  return *this;
 }
                                       //  оператор =
CCmplx& CCmplx:: operator = (double r)
{ m_re = r;   m_im = 0;
  return *this;
 }
CCmplx& CCmplx::  operator - ()     //  оператор - унарный
{ m_re = -m_re;   m_im = -m_im;
  return *this;
 }
CCmplx& CCmplx:: operator + ()   //  оператор + унарный
{ return *this;}

CCmplx& CCmplx:: operator ++ ()      // prefix ++a
{ m_re++;
  return *this;
 }
CCmplx CCmplx:: operator ++ (int)     //postfix  a++
{   CCmplx old(*this);
    m_re++;
```

```
    return *this;
  }
                //    +   два комплексных  числа
CCmplx CCmplx:: operator +( const CCmplx &r) const
{ CCmplx z(m_re + r.m_re, m_im + r.m_im);
  return z;
}
               //   -   два комплексных  числа
CCmplx CCmplx:: operator - (const CCmplx &r)  const
{ CCmplx z(m_re - r.m_re, m_im - r.m_im);
  return z;
}
                // *  два комплексных  числа
CCmplx CCmplx::operator *( const CCmplx &r)  const
{ CCmplx z(m_re * r.m_re - m_im * r.m_im,
           m_re * r.m_im + m_im * r.m_im);
   return z;
}
                //  /  два комплексных  числа
CCmplx CCmplx::operator /( const CCmplx &r) const
{ double d = r.m_re * r.m_re + r.m_im * r.m_im;
  CCmplx z((m_re * r.m_re + m_im * r.m_im) / d,
           (m_im * r.m_re + m_re * r.m_im) / d);
  return z;    }
               // комплексное  число + вещ.
CCmplx CCmplx:: operator + (double r)
{ CCmplx z(m_re + r, m_im);
   return z;
}
                //    комплексное  число - вещ.
CCmplx CCmplx:: operator - (double r)
{ CCmplx z(m_re - r, m_im);
  return z;
}
             //    комплексное  число * вещ.
CCmplx CCmplx:: operator * (double r)
{ CCmplx z(m_re * r, m_im * r);
  return z;
}
             //    комплексное  число / вещ.
CCmplx CCmplx:: operator / (double r)
{ CCmplx z(m_re / r, m_im / r);
  return z;
}
              // +  два компл. числа с замещением
CCmplx& CCmplx:: operator += (const CCmplx &r)
```

```
{ m_re += r.m_re;
  m_im += r.m_im;
  return *this;
}
              // -  два компл.  числа с замещением
CCmplx& CCmplx:: operator -= (const CCmplx &r)
{m_re -= r.m_re;
 m_im -= r.m_im;
 return *this;
}
             // *  два компл.  числа с замещением
CCmplx& CCmplx:: operator *=( const CCmplx &r)
{ m_re = m_re * r.m_re - m_im * r.m_im;
  m_im = m_re * r.m_im + m_im * r.m_im;
  return *this;
}
               // /  два компл.  числа с замещением
CCmplx& CCmplx:: operator /=( const CCmplx &r)
{double d = r.m_re * r.m_re + r.m_im * r.m_im;
  m_re = (m_re * r.m_re + m_im * r.m_im) / d;
  m_im = (m_im * r.m_re + m_re * r.m_im) / d;
  return *this;
}
             // компл  число + вещ.  с  замещением
CCmplx& CCmplx:: operator += (double r)
{ m_re += r;
  return *this;
}
              // компл  число - вещ.  с  замещением
CCmplx& CCmplx:: operator -= (double r)
{ m_re -= r;
  return *this;
}
                  // компл  число * вещ.  с  замещением
CCmplx& CCmplx:: operator *= (double r)
{ m_re *= r;
  m_im *= r;
  return *this;
}
                 // компл  число / вещ.  с  замещением
CCmplx& CCmplx:: operator /= (double r)
{ m_re /= r;
  m_im /= r;
  return *this;
}
             // сравнение компл. чисел на равенство
```

```
bool CCmplx:: operator ==(CCmplx &r)
{return (m_re == r.m_re)&& (m_im == r.m_im) ? true : false;}
          // сравнение компл. чисел на неравенство
bool CCmplx:: operator !=(CCmplx &r)
{return(m_re != r.m_re) || (m_im != r.m_im) ? true : false;}
            // сравнение компл. числа с вещ. на равенство
bool CCmplx:: operator ==(double r)
{return (m_re == r) && (m_im == 0) ? true : false; }
            // компл. числа с вещ. на неравенство
bool CCmplx:: operator !=(double r)
{ return (m_re != r) || (m_im != 0) ? true : false;}
```

```
     // файл   Demo_complex.cpp
    // программа демонстрирует работу с комплексными числами
#include <iostream>  // for cin cout
using namespace std;
#include "complex.h"
int main()
{
    CCmplx   complex1, complex2, complex3;
    complex1(1,2);              // инициализация компл. числа
    cout<<complex1<<endl;       //  вывод компл. числа

    complex2 = complex1;        //  оператор =
    cout<<complex2<<endl;       //  вывод компл. числа

    complex1++;                 //  оператор ++
    cout<<complex1<<endl;

    complex3 = complex1 + complex2; //сумма двух компл.чисел
    cout<<complex3<<endl;

    complex3 = complex1 + 2.0;  //компл. число + вещ. число
    cout<<complex3<<endl;

    return 0;
}
```