

# Список литературы

1. [И. А. Волкова, А. В. Иванов, Л. Е. Карпов. Основы объектно-ориентированного программирования. Язык программирования С++. Учебное пособие для студентов 2 курса](#) (969.27 Кбайт). – М.: Издательский отдел факультета ВМК МГУ, 2011.
2. [И. А. Волкова, А. А. Вылиток, Т. В. Руденко. Формальные грамматики и языки. Элементы теории трансляции \(3-е издание\)](#) (1.59 Мбайт). – М.: Изд-во МГУ, 2009 (версия от 06.02.2010).
3. [И. А. Волкова, И. Г. Головин, Л. Е. Карпов. Системы программирования \(Учебное пособие\)](#) (1.2 Мбайт). – М.: Издательский отдел факультета ВМиК МГУ, 2009.
4. [И. А. Волкова, А. А. Вылиток, Л. Е. Карпов.](#) Сборник задач и упражнений по языку С++ (Учебное пособие для студентов 2 курса). – М.: Издательский отдел факультета ВМК МГУ, 2013.
5. Д. Грис. Конструирование компиляторов для цифровых вычислительных машин. – М.: Мир, 1975.
6. Ф. Льюис, Д. Розенкранц, Р. Стирнз. Теоретические основы проектирования компиляторов. – М.: Мир, 1979.
7. А. Ахо, Дж. Ульман. Теория синтаксического анализа, перевода и компиляции, т. 1,2 – М.: Мир, 1979.
8. Л. Бек. Введение в системное программирование. – М.: Мир, 1988.
9. А. Ахо, Р. Сети, Дж. Ульман. Компиляторы. – М.: Изд. дом «Вильямс», 2001. (Шифр в библиотеке МГУ: 5ВГ66 А-955)
10. А. В. Гордеев, А. Ю. Молчанов. Системное программное обеспечение. – СПб.: Питер, 2001

11. [Г. Буч. Объектно-ориентированный анализ и проектирование с примерами приложений на C++ \(zip\)](#), 2-е издание. – М. СПб.: «Издательство Бином» – «Невский диалект», 1998.
12. А. Элиенс. Принципы объектно-ориентированной разработки программ, 2-е издание. – М.: Издательский дом «Вильямс», 2002.
13. И. О. Одинцов. Профессиональное программирование. Системный подход. – СПб.: БХВ-Петербург, 2002.
14. Н. Н. Мансуров, О. Л. Майлингова. Методы формальной спецификации программ: языки MSC и SDL. – М.: Изд-во «Диалог-МГУ», 1998.
15. [А. М. Вендров. CASE-технологии. Современные методы и средства проектирования информационных систем](#). – Электронная публикация на CITFORUM.RU
16. М. Фаулер, К. Скотт. UML в кратком изложении. Применение стандартного языка объектного моделирования. – М.: Мир, 1999.
17. Г. Майерс. Искусство тестирования программ. – М.: «Финансы и статистика», 1982
18. С. Канер, Дж. Фолк, Е. К. Нгуен. Тестирование программного обеспечения. – М.: «DiaSoft», 2001
19. Дж. Макгрегор, Д. Сайкс. Тестирование объектно-ориентированного программного обеспечения. Практическое пособие. – М.: «DiaSoft», 2002.
20. Б. Страуструп. Язык программирования C++. Специальное издание. – М.: Издательство «БИНОМ», 2001.
21. Б. Страуструп. Программирование: принципы и практика использования C++.: Пер. с англ. – М. ООО «И.Д.Вильямс», 2011. – 1248 с.
22. Г. Шилдт. Самоучитель C++. 3-е изд. – СПб: БХВ-Петербург, 2002.

# Электронные ссылки

Материалы по курсу можно найти на сайте:

<http://cmcmsu.no-ip.info/2course/>

Некоторые электронные ссылки на полезные книги:

<http://povt.zaural.ru/edocs/uml/content.htm> –

Г Буч, Д Рамбо, А Джекобсон «Язык UML. Руководство пользователя»

<http://vmk.ugatu.ac.ru/book/buch/index.htm> –

Гради Буч «Объектно-ориентированный анализ и проектирование с примерами приложений на C++»

# Язык C++

C++ позволяет справиться с возрастающей сложностью программ (в отличие от C).

Автор – Бьярне Страуструп.

Стандарты (комитета по стандартизации ANSI) – 1998, 2011.

C++:

- лучше C,
- поддерживает абстракции данных,
- поддерживает объектно-ориентированное
- программирование (ООП).

# Парадигмы программирования

Все программы состоят из кода и данных и каким-либо образом концептуально организованы вокруг своего кода и\или данных.

**Основные парадигмы** (технологии) программирования определяют способ построения программ:

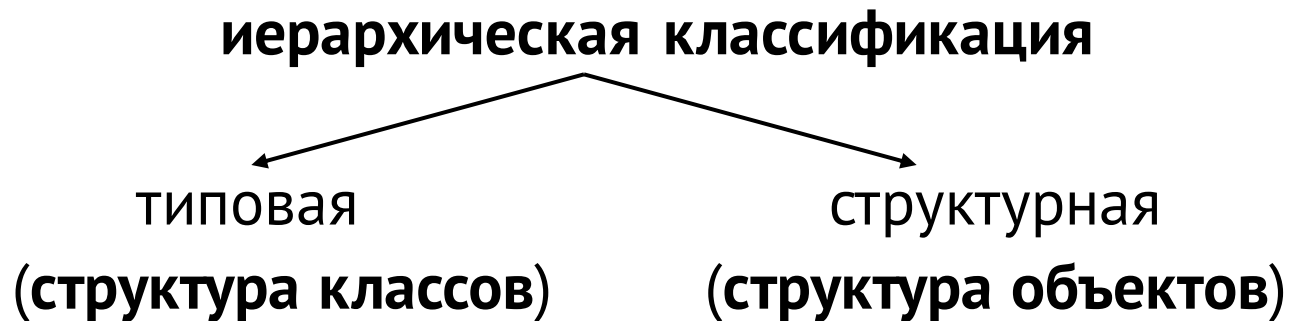
- **процедурно-ориентированная** (при кот. программа — это ряд последовательно выполняемых операций, причём код воздействует на данные, например в программах на С),
- **объектно-ориентированная** (при кот. программа состоит из объектов — программных сущностей, объединяющих в себе код и данные, взаимодействующих друг с другом через определенные интерфейсы, при этом доступ к коду и данным объекта осуществляется только через сам объект, т.е. данные определяют выполняемый код),
- функциональная,
- логическая.

# Постулаты ООП.

**Абстракция** – центральное понятие ООП.

**Абстракция** позволяет программисту справиться со сложностями решаемых им задач.

Мощный способ создания абстракций –



Основные механизмы (постулаты) ООП:

- **инкапсуляция,**
- **наследование,**
- **полиморфизм.**

# ИНКАПСУЛЯЦИЯ

**Инкапсуляция** — механизм,

- связывающий вместе код и данные, которыми он манипулирует;
- защищающий их от произвольного доступа со стороны другого кода, внешнего по отношению к рассматриваемому.

Доступ к коду и данным жестко контролируется интерфейсом. Основой инкапсуляции является **класс**.

**Класс** — это механизм (пользовательский тип данных) для создания объектов.

**Объект** класса — переменная типа класс или экземпляр класса. Любой объект характеризуется **состоянием** (значениями полей данных) и **поведением** (операциями над объектами, задаваемыми определенными в классе функциями, которые называют **методами** класса).

# НАСЛЕДОВАНИЕ

**Наследование** — механизм, с помощью которого один объект (**производного класса**) приобретает свойства другого объекта (**родительского, базового класса**).

Наследование позволяет объекту производного класса наследовать от своего родителя общие атрибуты, а для себя определять только те характеристики, которые делают его уникальным внутри класса.

Производный класс конкретизирует, в общем случае **расширяет** базовый класс.

Наследование поддерживает концепцию иерархической классификации.

Новый класс не обязательно описывать, начиная с нуля, что существенно упрощает работу программиста.



# ПОЛИМОРФИЗМ

**Полиморфизм** — механизм, позволяющий использовать один и тот же интерфейс для общего класса действий.

В общем случае концепция полиморфизма выражается с помощью фразы «один интерфейс — много методов»

Выбор конкретного действия (метода) применительно к конкретной ситуации возлагается на компилятор.

Программисту же достаточно запомнить и применять один интерфейс, вместо нескольких, что также упрощает его работу.

Различаются следующие виды полиморфизма:

- **статический** (на этапе компиляции, с помощью перегрузки функций),
- **динамический** (во время выполнения программы, реализуется с помощью виртуальных функций) и
- **параметрический** (на этапе компиляции, с использованием механизма шаблонов).

# Декомпозиция задачи

При программировании в объектно-ориентированном стиле на первое место выходит **проектирование** решения задачи, т.е. определение того, какие классы и объекты будут использоваться в программе, каковы их свойства и способы взаимодействия.

Как правило, при этом необходимо произвести декомпозицию задачи.

**Декомпозиция** — научный метод, использующий структуру задачи и позволяющий разбить решение одной большой задачи на решения серии меньших задач, возможно взаимосвязанных, но более простых.

# Синтаксис класса

```
class имя_класса {  
  [private:]  
    закрытые члены класса (функции, типы и поля-данные)  
  public:  
    открытые члены класса (функции, типы и поля-данные)  
  protected:  
    защищенные члены класса  
} список_объектов;
```

Описание объектов – экземпляров класса:

```
имя_класса список_объектов; // служ. слово class не требуется
```

Классы С++ отличаются от структур С++ **только** правилами определения **по умолчанию**:

- **прав доступа** к первой области доступа членов класса и
- **типа наследования**:

для **структур** – **public**,  
для **классов** – **private**.

# Члены класса

- Члены-данные;
- Члены-функции (методы);
- Члены-типы – вложенные пользовательские типы,

Правила доступа к членам класса и поиска их имен единообразны для всех членов класса и не зависят от их вида.

Ex.:

```
class X {  
    double t;           // данное  
public:  
    void f ( );        // метод  
    int a;             // данное  
    enum { e1, e2, e3 } g;  
private:  
    struct inner {     // вложенный класс  
        int i, j;  
        void g ( );  
    };  
    inner c;  
};
```

...

```
X x;  x.a = 0; x.g = X::e1;
```

## Действия над объектами классов

Над объектами класса можно производить следующие действия:

- присваивать объекты одного и того же класса (при этом производится почленное копирование членов данных)
- получать адрес объекта с помощью операции **&**
- передавать объект в качестве формального параметра в функцию
- возвращать объект в качестве результата работы функции.
- осуществлять доступ к элементам объекта с помощью операции «.», а если используется указатель на объект, то с помощью операции «->»
- вызывать методы класса, определяющие поведение объекта

# Пример класса

```
...
class A {
    int a;
public:
    void set_a (int n);
    int get_a ( ) const { return a; }
        // Константные методы класса
        // не изменяют состояние своего объекта
};

void A::set_a (int n) {
    a = n;
}

int main () {
    A obj1, obj2;
    obj1.set_a(5);
    obj2.set_a(10);
    cout << obj1.get_a ( ) << '\n';
    cout << obj2.get_a ( ) << endl;
    return 0;
}
```

## АТД (абстрактный тип данных)

**АТД называют тип данных с полностью скрытой (инкапсулированной) структурой**, а работа с переменными такого типа происходит только через специальные, предназначенные для этого функции.

В С++ АТД реализуется с помощью классов (структур), в которых нет открытых членов-данных.

Класс А из предыдущего примера является абстрактным типом данных.

## О терминологии

**Оператор** (statement) – действие, задаваемое некоторой конструкцией языка.

**Операция** (operator, для обозначения операций языка: « + », « \* », « = », и др.) – используются в выражениях.

**Определение (описание) переменной** (definition) – при этом отводится память, производится инициализация, определение возможно только 1 раз.

**Объявление переменной** (declaration) – дает информацию компилятору о том, что эта переменная где-то в программе описана.

Для преобразования типов используются два термина – **преобразование** (conversion) и **приведение** (cast).



# Некоторые отличия C++ от C

- Введен логический тип ***bool*** и константы логического типа ***true*** и ***false***.
- В C++ отсутствуют типы по умолчанию (например, обязательно ***int main () {...}***).
- Локальные переменные можно описывать в любом месте программы, в частности внутри цикла `for`. Главное, чтобы они были описаны до их первого использования.  
По стандарту C++ переменная, описанная внутри цикла `for`, локализуется в теле этого цикла.
- В C++ переработана стандартная библиотека.  
В частности, в стандартной библиотеке C++ файл заголовков ввода/вывода называется **<iostream>**, введены классы, соответствующие стандартным (консольным) потокам ввода — класс ***istream*** — и вывода — класс ***ostream***, а также объекты ***cin*** (класса *istream*) и ***cout*** и ***cerr*** (класса *ostream*).  
Через эти объекты доступны операции ввода **>>** из стандартного потока ввода (например, `cin >> x ;`), и вывода **<<** в стандартный поток вывода (например, `cout << "string" << S << '\n'`);, при использовании которых не надо указывать никакие форматирующие элементы.

# Работа с динамической памятью

```
int *p,*m;  
p = new int;
```

или

```
p = new int (1);
```

или

```
m = new int [10];
```

— для массива из 10 элементов;

Массивы, создаваемые в динамической памяти инициализировать нельзя.

```
...  
delete p;
```

или

```
delete [ ] m;
```

— для удаления всего массива.

# Значения параметров функции по умолчанию

Пример:

```
void f (int a, int b = 0, int c = 1);
```

Обращения к функции:

```
f(3)           // a = 3, b = 0, c = 1  
f(3, 4)       // a = 3, b = 4, c = 1  
f(3, 4, 5)    // a = 3, b = 4, c = 5
```

# Пространства имен

Пространства имен вводятся только на уровне файла, но не внутри блока.

```
namespace std {  
    // объявления, определения  
}
```

Ex:       std::cout << std::endl;

```
namespace NS {  
    char name [ 10 ] ;  
    namespace SP {  
        int var = 3;  
    }  
}
```

Ex:       ... NS::name ...;                   NS::SP::var += 2;

```
#include <iostream>  
using namespace std;  
  
using NS::name;
```

## Указатель **this**

Иногда для реализации того или иного метода возникает необходимость иметь указатель на «свой» объект, от имени которого производится вызов данного метода.

В C++ введено ключевое слово **this**, обозначающее «указатель на себя», которое можно трактовать как неявный параметр любого метода класса:

```
<имя класса> * const this;
```

**\*this** — сам объект.

Таким образом, любой метод класса имеет на один (первый) параметр больше, чем указано явно.

**This**, участвующий в описании функции, перегружающей **операцию**, всегда указывает на **самый левый** (в выражении с этой операцией) операнд операции.

В реальности поле **this** не существует (не расходует память), и при сборке программы вместо **this** подставляется соответствующий адрес объекта.

## Специальные методы класса

**Конструктор** — метод класса, который

- имеет имя, в точности совпадающее с именем самого класса;
- не имеет типа возвращаемого значения;
- **всегда** вызывается при создании объекта (сразу после отведения памяти под объект в соответствии с его описанием).

**Деструктор** — метод класса, который

- имеет имя, совпадающее с именем класса, перед первым символом которого приписывается символ « ~ »;
- не имеет типа возвращаемого значения и параметров;
- **всегда** вызывается при уничтожении объекта (перед освобождением памяти, отведенной под объект).

# Специальные методы класса

```
class A {
    ...
    public:
        A ( );           // конструктор умолчания
        A (A & y);       // A (const A & y);
                        // конструктор копирования (КК)
[explicit] A (int x);
                        // конструктор преобразования;
                        // explicit запрещает компилятору
                        // неявное преобразование int в A
        A (int x, int y); // A (int x = 0, int y = 0);
                        // заменяет 1-ый, 3-ий и 4-ый
                        // конструкторы
        ~A ( );         // деструктор
    ...
};

int main ( ) {
    A a1, a2 (10), a3 = a2;
    A a4 = 5, a5 = A(7); // Err!, т.к. временный объект
                        // не может быть параметром для
                        // неконстантной ссылки в КК
    A *a6 = new A (1);  // O.K., если будет A (const A & y)
}

```

# Правила автоматической генерации специальных методов класса

- Если в классе **явно не описан никакой конструктор**, то конструктор умолчания генерируется автоматически с пустым телом в **public** области.
- Если в классе явно не описан конструктор копирования, то он **всегда генерируется автоматически** в **public** области с телом, реализующим почленное копирование значений полей-данных параметра конструктора в значения соответствующих полей-данных создаваемого объекта
- Если в классе явно не описан деструктор, то он **всегда генерируется автоматически** с пустым телом в **public** области.



# Класс Box

```
class Box {  
    int l;           // length - длина  
    int w;           // width  - ширина  
    int h;           // height - высота  
public:  
    int volume () const { return l * w * h ; }  
    Box (int a, int b, int c ) { l = a; w = b; h = c; }  
    Box (int s) { l = w = h = s; }  
    Box ( ) { w = h = 1; l = 2; }  
    int get_l ( ) const { return l; }  
    int get_w ( ) const { return w; }  
    int get_h ( ) const { return h; }  
};
```

Автоматически сгенерированные конструктор копирования и операция присваивания:

```
Box (const Box & a) { l = a.l; w = a.w; h = a.h; }  
Box & operator = ( const Box & a)  
{ l = a.l; w = a.w; h = a.h; return * this; }
```

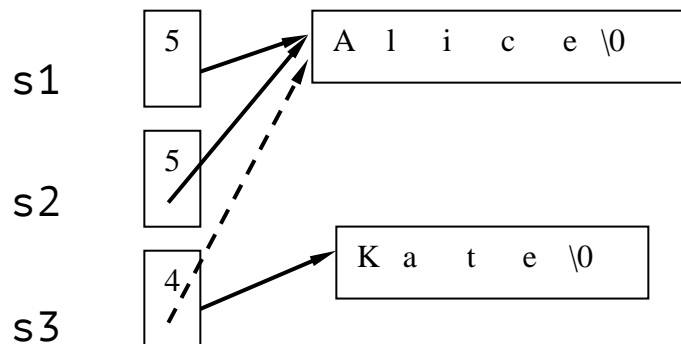
Конструктор копирования и операцию присваивания можно переопределить.

## Неплоский класс string

```
class string {  
    char * p; // здесь потребуется динамическая память,  
    int size;  
public:  
    string (const char * str);  
    string (const string & a);  
    ~string ( ) { delete [ ] p; }  
    string & operator= (const string & a);  
    ...  
};  
  
string :: string (const char * str) {  
    p = new char [ ( size = strlen (str) ) + 1];  
    strcpy (p, str);  
}  
  
string :: string (const string & a) {  
    p = new char [ (size = a.size) + 1];  
    strcpy (p, a.p);  
}
```

## Пример использования класса string

```
void f {  
    string s1 ("Alice");  
    string s2 = s1;  
    string s3 ("Kate");  
    ...  
    s3 = s1;  
}  
{... s1...s2 {...s3...}...s1...s2}
```



## Переопределение операции присваивания

```
string & string :: operator = (const string & a) {  
  
    if (this == & a)  
        return * this;    // если a = a  
    delete [ ] p;  
    p = new char [ (size = a.size) + 1];  
    strcpy (p, a.p);  
    return * this;  
}
```

При этом:  $s1 = s2 \sim s1.operator=(s2);$

# Композиция (строгая агрегация) объектов

```
class Point {  
    int x;  
    int y;  
public:  
    Point ( );  
    Point ( int, int );  
    ...  
};
```

```
class Z {  
    Point p;  
    int z;  
public:  
    Z ( int c ) { z = c; };  
    ...  
};
```

```
Z * z = new Z (1);    // Point (); Z(1);  
delete z;            // ~Z(); ~Point();
```

Использование **списка инициализации** при **описании** конструктора:

```
Z::Z ( int c ) : p (1, 2) { z = c; }
```

или

```
Z::Z ( int c ) : p (1, 2), z (c) { }
```

# Ссылки 1

**Ссылочный тип данных** задается так:

<тип> &

**Ссылка** (reference) – переменная ссылочного типа.

Единственная **операция над ссылками – инициализация** (установление связи с инициализатором) при создании, при этом ссылка обозначает (именует) тот же **адрес** памяти, что и ее инициализатор (L-value выражение).

После описания и обязательной инициализации ссылку можно использовать точно так же, как и соответствующий ей инициализатор.

Фактически ссылка является синонимом своего инициализатора.

Ссылочный тип данных в C++ используется в следующих случаях:

а) **Описание переменных-ссылок** (локальных или глобальных).

Например,

```
int i = 5;
int & yeti = i; // ссылка обязательно должна быть инициализирована
                // yeti - синоним имени i ; &i ≡ &yeti;
i = yeti + 1;
yeti = i + 1;
cout << i << yeti; // напечатается 7 7
```

## Ссылки 2

### b) Передача параметров в функции по ссылке.

Инициализация формального параметра ссылки происходит в момент передачи фактического параметра (L-value выражения), и далее все действия, выполняемые с параметром-ссылкой, выполняются с соответствующим фактическим параметром.

Пример:

```
void swap (int & x, int & y) {  
    int t = x;  
    x = y;  
    y = t;  
}
```

Пример обращения к функции swap:

```
int a = 5, b = 6;  
swap (a, b);
```

c) **Возвращение результата работы функции в виде ссылки** — для более эффективной реализации функции - т.к. не надо создавать временную копию возвращаемого объекта — и в том случае, когда возвращаемое значение должно быть L-value-выражением.

Инициализация возвращаемой ссылки происходит при работе оператора **return**, операндом которого должно быть L-value выражение. **Не следует возвращать ссылку на локальный объект функции**, который перестает существовать при выходе из функции.

Пример:

```
int & f() {  
    int * p = new int(5);  
    return *p;  
}
```

Пример обращения к функции f:

```
int & x = f();
```

## Ссылки 3

d) Использование ссылок — членов-данных класса.

Инициализация поля-ссылки класса обязательно происходит через список инициализации конструктора, вызываемого при создании объекта.

Пример:

```
class A {
    int x;
public:
    int & r;
    A( ) : r (x) {
        x = 3;
    }
    A (const A &);           // !!!
    A & operator= (const A&); // !!!
    ...
};
int main () {
    A a;
    ...
}
```



# Ссылки 4

## Константные ссылки

е) Использование **ссылок на константу** – формальных параметров функций (для эффективности реализации в случае объектов классов).

Инициализация параметра – ссылки на константу происходит во время передачи фактического параметра, который, в частности, может быть **временным объектом**, сформированным компилятором для фактического параметра-**константы**.

Пример:

```
struct A {  
    int a;  
    A ( int t = 0) { a = t; }  
};  
int f (const int & n, const A & ob) {  
    return n+ob.a;  
}  
int main () {  
    cout << f (3, 5) << endl;  
    ...  
}
```

# Временные объекты

Временные объекты создаются в рамках выражений (в частности, инициализирующих), где их можно модифицировать (применять неконстантные методы, менять значения членов-данных).

В общем случае «живут» временные объекты до окончания вычисления соответствующих выражений.

НО! Если инициализировать ссылку на константу временным объектом (в частности, передавать временный объект в качестве параметра для формального параметра – ссылки на константу), время его жизни продлевается до конца жизни соответствующей ссылочной переменной.

НЕЛЬЗЯ инициализировать неконстантную ссылку временным объектом (в частности, неконстантные ссылки – формальные параметры).

```
Пример:      struct A {
                A (int);
                A (const A &);
            };
...          const A & r = A (1);      // если здесь и в КК убрать const,
                A a1 = A (2);          // все эти конструкции будут
                A a2 = 3; ...           // ошибочными
```

**Важно!** Компилятор ВСЕГДА сначала проверяет синтаксическую и семантическую (контекстные условия) правильность, а затем оптимизирует!!!

# Порядок вызова конструкторов и деструкторов

При вызове **конструктора** класса выполняются:

1. конструкторы базовых классов (если есть наследование),
2. конструкторы умолчания всех вложенных объектов в порядке их описания в классе,
3. собственный конструктор (при его вызове все поля класса уже проинициализированы, следовательно, их можно использовать).

**Деструкторы выполняются в обратном порядке:**

1. собственный деструктор (при этом поля класса ещё не очищены, следовательно, доступны для использования),
2. автоматически вызываются деструкторы для всех вложенных объектов в порядке, обратном порядку их описания в классе,
3. деструкторы базовых классов (если есть наследование).

## Вызов конструктора копирования

1. явно,
2. в случае:  
Вох a (1, 2, 3);  
Вох b = a; // a - параметр конструктора копирования,
3. в случае:  
Вох c = Вох (3, 4, 5);  
*// сначала создается временный объект и вызывается  
// обычный конструктор, а затем работает конструктор  
// копирования при создании объекта c; если компилятор  
// оптимизирующий, вызывается только обычный  
// конструктор с указанными параметрами;*
4. при передаче параметров функции по значению (при создании локального объекта);
5. при возвращении результата работы функции в виде объекта,
6. при генерации исключения-объекта.

# Вызов других конструкторов

- явно,
- при создании объекта (при обработке описания объекта),
- при создании объекта в динамической памяти (по new), при этом сначала в «куче» отводится необходимая память, а затем работает соответствующий конструктор,
- при композиции объектов наряду с собственным конструктором вызывается конструктор объекта – члена класса,
- при создании объекта производного класса также вызывается конструктор и базового класса,
- при автоматическом приведении типа с помощью конструктора преобразования.

# Вызов деструктора

1. явно,
2. при свертке стека - при выходе из блока описания объекта, в частности при обработке исключений, завершении работы функции;
3. при уничтожении временных объектов - сразу, как только завершается конструкция, в которой они использовались;
4. при выполнении операции delete для указателя на объект (инициализация указателя - с помощью операции new), при этом сначала работает деструктор, а затем освобождается память.
5. при завершении работы программы при удалении глобальных/статических объектов.

Конструкторы вызываются в порядке определения объектов в блоке. При выходе из блока для всех автоматических объектов вызываются деструкторы, в порядке, противоположном порядку выполнения конструкторов.