**Объектно-ориентированное программирование (ООП) в C++**

Объектно-ориентированное программирование (ООП) — подход к программированию, при котором основными концепциями являются понятия объектов и классов.

**Класс** — это определяемый разработчиком тип данных.

Тип данных характеризуется:

1. Способом хранения и представления этих данных.
2. Назначением этих данных (профилем их использование).
3. Набором действий, которые над этими данными могут производится.

Например, тип int предназначен для хранения целых чисел и подразумевает ряд операция над ними (+,-,\*,/,% и пр.).

Класс — это тип, описывающий устройство объектов, их поведение и способ представления.

**Объект** — сущность обладающая определённым поведением и способом представления, т. е. относящееся к классу (говорят, что объект — это экземпляр класса, если он к нему относится).

Класс можно сравнить с чертежом, согласно которому создаются объекты. Обычно классы разрабатывают таким образом, чтобы их объекты соответствовали объектам предметной области решаемой задачи.

Описание класса начинается со служебного слова class, вслед за которым указывается имя класса. Затем в блоке следует описание класса и после закрывающей скобки блока ставится точка с запятой.

Описание класса состоит из полей и методов.

**Поля** (или свойства, в рамках C++ это можно считать синонимом) описывают то, какие данные смогут хранить экземпляры класса (т.е. объекты). Конкретные значения сохраняются уже внутри объектов. Поля объявляются в теле класса.

К полям внутри класса можно обращаться непосредственно по именам полей.

**Методы класса** — это функции, которые смогут применяться к экземплярам класса. Грубо говоря, метод — это функция объявленная внутри класса и предназначенная для работы с его объектами.

Методы объявляются в теле класса. Описываться могут там же, но могут и за пределами класса (внутри класса в таком случае достаточно представить прототип метода, а за пределами класса определять метод поставив перед его именем — имя класса и оператор ::).

Методы и поля входящие в состав класса называются членами класса. При этом методы часто называют функциями-членами класса.

Пример:

class Complex {

double img;

double real;

};

В примере описан класс Complex с двумя полями img и real.

**Абстракция данных**

Абстракция данных — это выделение существенных характеристик объекта, которые отличают его от прочих объектов, четко определяя его границы.

Абстракция данных в ООП предусматривает выделение характеристик, существенных в рамках решаемой задачи и рассматриваемой предметной области.

Например, создавая программу для автоматизации работы склада, важно учитывать вес контейнера, размер контейнера, его положение на складе, но совсем не важен цвет контейнера.

Вес, размер и положение — это поля будущего объекта-контейнера.

Методы, которые к этому объекту смогут применяться возможны такие: создать в программе новый объект-контейнер, переместить объект на указанную позицию, удалить объект со склада, пометив занятое им место как свободное.

**Ключевые черты ООП**

1. **Инкапсуляция** — это принцип, согласно которому любой класс должен рассматриваться как чёрный ящик — пользователь класса должен видеть и использовать только интерфейсную часть класса (т. е. список декларируемых свойств и методов класса) и не вникать в его внутреннюю реализацию. Поэтому данные принято инкапсулировать в классе таким образом, чтобы доступ к ним по чтению или записи осуществлялся не напрямую, а с помощью методов. Принцип инкапсуляции (теоретически) позволяет минимизировать число связей между классами и, соответственно, упростить независимую реализацию и модификацию классов.
2. **Наследование** — это порождение нового класса-потомка от уже существующего класса-родителя. Класс-родитель называют также супер-классом, а класс-потомок — подклассом. Наследование происходит с передачей всех или некоторых полей и методов от класса-родителя к классу-потомку. В процессе наследования возможно, при необходимости, добавлять новые свойства и методы. Набор классов, связанных отношением наследования, называют иерархией.
3. **Полиморфизм** — это явление, при котором функции (методу) с одним и тем же именем соответствует разный программный код (полиморфный код) в зависимости от того, в каком контексте он вызывается (объектами какого класса или с какими параметрами).

**Уровни доступа к членам класса**

По уровню доступа все члены класса делятся на **открытые** (public), **закрытые** (private) и **защищённые** (protected).

Перед объявлением членов внутри класса ставятся соответствующие ключевые слова. Если такое слово не поставлено, то считается, что член объявлен с уровнем private. В примере выше класса Complex, соответственно, оба поля имеют уровень доступа private.

Члены объявленные как private доступны только внутри класса.

Члены объявленные как protected доступны внутри класса и внутри всех его потомков.

Члены объявленные как public доступны как внутри, так вне класса (в том числе в потомках).

|  |  |  |  |
| --- | --- | --- | --- |
| **Доступность членов класса в зависимости от уровня доступа** | **private** | **protected** | **public** |
| Внутри класса | + | + | + |
| Внутри потомка класса | - | + | + |
| В несвязанной с классом части программы | - | - | + |

Методы объявленные в открытой части класса называются его интерфейсом

Пример объявления различных уровней доступа:

class Complex {

private: // Закрытая часть класса, её элементы доступны только внутри класса

double img;

public: // Открытая часть класса, её элементы доступны везде

double real;

double getImg() { // Через этот метод мы сможем получить значение закрытого поля

return img; // Поле закрытое, но метод открытый

};

};

**Интерфейс класса**

Все методы класса, находящиеся в его открытой части (с уровнем доступа public) называют **интерфейсом класса**

Интерфейс предназначен для взаимодействия класса с остальной программой. Зная методы интерфейса и их назначение можно использовать класс, не вникая в его внутреннее устройство. Соответственно, интерфейс реализует принцип инкапсуляции.

**Конструктор и деструктор**

При создании объектов одной из наиболее широко используемых операций которую вы будете выполнять в ваших программах, является инициализация элементов данных объекта. Чтобы упростить процесс инициализации элементов данных класса, C++ использует специальную функцию, называемую конструктором, которая запускается для каждого создаваемого вами объекта. Также C++ обеспечивает функцию, называемую деструктором, которая запускается при уничтожении объекта.  
Конструктор представляет собой метод класса, который облегчает вашим программам инициализацию полей при создании объекта класса.

Конструктор имеет такое же имя, как и сам класс.

Конструктор не имеет возвращаемого значения (по сути, результатом его работы является ссылка на созданный объект).

Каждый раз, когда ваша программа создает объект, C++ вызывает конструктор класса, если подходящий (с соответствующими параметрами) конструктор существует.

Конструкторы относят к интерфейсу класса, чтобы с их помощью можно было создавать объекты данного класса из внешней части программы.

**Конструктор по умолчанию**

Конструктор по умолчанию — это конструктор без параметров.

Если он не задан явно и при этом не создано других конструкторов с параметрами, то конструктор по умолчанию создастся автоматически. При этом все свойства нового объекта не будут никак проинициализрованы (получат «мусорные» значения из памяти).

Пример:

class Complex {

private: // Закрытая часть класса

double real; // Поле, действительная часть

double img; // Поле, мнимая часть

public: // Открытая часть класса

void printComplex() { // Этот метод мы сможем вызывать за приделами класса

cout &lt< real &lt< " + " &lt< img &lt< 'i' &lt< endl; // Выводим поля

}

};

int main() {

Complex a; // Работает конструктор по умолчанию,

Complex b; // поля получают мусорные значения

a.printComplex(); // Выводим первый и второй объекты

b.printComplex(); // с помощью созданного метода printComplex()

return 0;

}

В представленном примере отсутствует явно заданный конструктор, поэтому будет создан конструктор по умолчанию. Мы можем явно задать конструктор по умолчанию и обязаны это сделать, когда у нас имеется какой-либо конструктор с параметрами.

Пример класса с двумя конструкторами (первый из них — без параметров):

class Complex {

private:

double real; // Действительная часть

double img; // Мнимая часть

public:

Complex() {

real = 0;

img = 0;

}

Complex(double a, double b) {

real = a;

img = b;

}

void printComplex() {

cout &lt< real &lt< " + " &lt< img &lt< 'i' &lt< endl;

}

};

int main() {

Complex a;

Complex b(3.14, 2.71);

a.printComplex();

b.printComplex();

return 0;

}

При создании объекта мы должны либо в круглых скобках указать параметры, чтобы заработал нужный конструктор (как для объекта b), либо не указывать ничего, чтобы использовался конструктор по умолчанию (как для объекта a).

**Копирующий конструктор**

Часто при создании нового объекта бывает удобным инициализировать его поля теми же значениями, что уже имеются у какого-то существующего объекта того же типа. Проще говоря, создавая новый объект порой удобно создать его как копию уже существующего.

Конструктор, который в качестве аргумента принимает другой объект того же типа — называется **копирующим конструктором**.

Обычно аргумент копирующего конструктора объявляется как ссылка на константу, чтобы копия никак не влияла на оригинал.

Пример:

class Complex {

private:

double real; // Действительная часть

double img; // Мнимая часть

public:

Complex() {

real = 0;

img = 0;

}

Complex(const Complex& c) { // Копирующий конструктор

real = c.real;

img = c.img;

}

Complex(double a, double b) {

real = a;

img = b;

}

void printComplex() {

cout &lt< real &lt< " + " &lt< img &lt< 'i' &lt< endl;

}

};

int main() {

Complex b(3.14, 2.71);

Complex d(b);

d.printComplex();

return 0;

}

Копирующий конструктор создаётся по умолчанию, если не задан явно. В поля нового объекта он копирует значения соответствующих полей объекта-аргумента.

В некоторых случаях это приводит к серьезным проблемам. Например, если объект-аргумент имеет поля, хранимые в динамической памяти, то в создаваемый объект будут скопированы адреса динамической памяти, и изменение полей нового объекта, повлечёт аналогичные изменения в исходном объекте (по сути два объекта будут хранить в своих полях ссылки на одну и ту же область динамической памяти).

**Деструктор**

После того, как вы по ходу программы перестаёте пользоваться некоторым объектом, для него автоматически вызывается специальный метод-**деструктор**, если он описан в классе.

Представлять конструктор можно конструктор как функцию, которая помогает вам строить (конструировать) объект. Подобно этому, деструктор представляет собой функцию, которая помогает корректно уничтожать объект. Деструктор обычно используется, если при уничтожении объекта нужно освободить динамическую память, занимаемую объектом.

Деструктор имеет такое же имя, как и класс, но с символом тильды (~) в качестве префикса. Например, в классе Complex деструктор имеет имя ~Complex().

Деструктор не имеет возвращаемого значения и не имеет аргументов. Деструктор в классе всегда один (его нельзя перегружать). По умолчанию деструктор не создаётся, если он требуется, то его надо описывать явно.

**Наследование**

Пример наследования:

// Класс товаров в некотором магазине

class ShopItem {

public:

char model[100]; // модель

char brand[100]; // производитель

double price; // цена

};

// Мобильные телефоны — наследник класса товаров

class MobilePhone : public ShopItem {

public:

char phoneColor[100]; // цвет

int batteryLife; // время работы от батареи в часах

};

// Карты памяти — наследник класса товаров

class MemoryCard : public ShopItem {

public:

char formFactor[100]; // формат карточки памяти

};

// Смартфоны — наследник класса мобильных телефонов

class SmartPhone : public MobilePhone {

public:

char os[100]; // операционная система

};

Когда объявляется класс-наследник, то сразу после его имени ставится двоеточие и указывается уровень (или способ) наследования (public, private, protected), а затем имя класса родителя. Уже потом открывается блок с описанием класс-наследника.

**Private-наследование**

Те члены, что в родителе были protected и public, становятся private в потомке.

**Protected-наследование**

Те члены, что в родители были protected и public становятся protected в потомке. Такой вариант используется редко.

**Public-наследование**

Те члены, что в родители были protected и public, остаются, соответственно, protected и public в наследнике, сохраняя свой уровень доступа.

Если уровень явно не указан, то происходит private-наследование.

Можно говорить о том, что public-наследование — это наследование интерфейса, а private-наследование — это наследование реализации (т.е. все элементы наследуются, но становятся закрытыми от доступа из вне).

**Статические члены класса**

Члены класса, объявленные с модификатором доступа static, называются статическими членами класса. Статические поля и методы доступны без создания объекта данного класса.

Доступ к статическому элементу осуществляется с использованием оператора принадлежности :: , а не через имя экземпляра класса с точкой.

Статические методы могут вызывать и использовать только другие статические методы и статические переменные. Из статического метода нельзя выполнять вызов не статического метода класса.

Статическую переменную можно рассматривать как аналог глобальной переменной, которая связана с конкретным классом.

Часто статические переменные называют переменными класса, а «не статические» переменные — переменными экземпляра. Для статических переменных могут использоваться указатели.

**Указатель this**

Указатель **this** — это константный указатель на текущий объект (т. е. на объект, к которому применяется вызываемый метод). Этот указатель существует всюду внутри класса, его не надо как-то явно объявлять, к нему можно обращаться внутри любого метода класса.

Предположим, что в классе у нас есть поле с именем f и есть метод, который к этому свойству должен обращаться, но в котором имеется локальная переменная с тем же именем f. Указатель this поможет одновременно использовать поле и локальную переменную: через указатель поле будет доступно как this->f.

Пример:

class Example {

private:

float f; // Какое-то поле класса

public:

void func(float f) {

this->f = f;

}

};

Данные у каждого объекта свои, а методы класса общие для всех объектов. Указатель thisпомогает, определить с данными какого объекта будет работать метод.

Разыменовав указатель this можно получить ссылку на объект.

**Перегрузка операторов**

В С++ можно создавать методы со специальными именами вида: operator \*, где \* — это один из существующих операторов языка (например, +, -, /, \*, = и т.д.).

При создании такого метода, говорят, что оператор \* был перегружен, т.е. теперь его можно использовать для взаимодействия с объектами создаваемого класса. Перегружать можно [многие, но не любые операторы](http://itedu.ru/faq/kakie-operatory-mozhno-peregruzhat-v-cpp).

Вызывать этот метод можно обращаясь к нему по указанному символу (в данном случае, \*). При этом аргументом метода будет считаться объект стоящий справа от символа, а вызываться метод будет для объекта, стоящего слева от символа.

Так, например, удобно для объектов своего класса использовать символ + в качестве метода, складывающего объекты, вместо именованного метода (например, sum).

Перегрузка операторов, как и перегрузка функций в целом, реализует принцип полиморфизма.

Пример:

Complex opertor + (Complex a) {

Complex tmp;

tmp.real = real + a.real;;

tmp.img = img + a.img;

}

Пример использования:

Complex a(2,3);

Complex b(3,2);

Complex c;

с = a + b; // Использование перегруженного оператора + для класса Complex

cout &lt< c; // 5 + 5i

К сожалению, свои собственные операторы в C++ создавать нельзя, а можно только перегружать существующие (не все, но многие). Такое ограничение вызвано прежде всего тем, что синтаксический анализатор языка с созданием каждого нового оператора пришлось бы тоже учить как-то его распознавать (проверять корректность использования и т.д.).

Перечень операторов доступных для перегрузки:

+ - \* / % ^ & |

~ ! , >= <= > < =

++ -- << >> == != && ||

+= -= /= %= ^= &= |= \*=

<<= >>= [ ] ( ) -> ->\* new delete

Соответственно, даже среди существующих операторов есть такие, которые нельзя перегрузить. Например, оператор разрешения области видимости :: или оператор ., используемый для обращения к полям и методам объекта.

Отдельно хочется отметить, что перегрузка операторов не всегда является процедурой необходимой. Во многих современных языках такой возможности нет (например, в Java).

Во-первых, перегружая оператор можно наделить его не очевидным смыслом для всех прочих разработчиков, кроме автора программы. Например, логичным является перегрузка оператора + для класса строк (он может использоваться для склейки / объединения / конкатенации двух и более строк), но какой результат должен давать, например, оператор \* примененный к двум строкам?

Во-вторых, перегрузка оператора в контексте конкретной задачи (конкретного класса) может получиться неоднозначной. Например, если для класса векторов мы перегружаем оператор \*, то результатом должно быть векторное или скалярное произведение?

**Примеры некоторых классов**

**Класс комплексных чисел**

Далее будет рассмотрен пример реализации класса комплексных чисел с некоторыми операциями над ними.

В примере показано, как можно объявлять методы в классе (предъявляя прототипы методов), а описывать — за пределами класса (используя оператор ::).

Пример:

#include <iostream>

#include <cmath> // Этот заголовочный файл содержит функцию sqrt для вычисления корня

using namespace std;

class Complex {

private:

double real;

double img;

public:

Complex() { // Конструктор по умолчанию

real = 0;

img = 0;

}

Complex(const Complex& c) { // Копирующий конструктор

this->real = c.real;

this->img = c.img;

}

Complex(double, double); // Конструктор объявлен в классе, а определен будет вне его

void operator = (Complex c) { // Перегрузка оператора присваивания

this->real = c.real;

this->img = c.img;

}

bool operator == (Complex c) { // Перегрузка оператора сравнения

if (this->real == c.real && this->img == c.img) {

return true;

} else {

return false;

}

}

Complex operator + (Complex c) { // Перегрузка оператора сложения

Complex tmp;

tmp.real = this->real + c.real;

tmp.img = this->img + c.img;

return tmp;

}

Complex operator \* (Complex c); // Перегрузка оператора умножения

// В классе только прототип, определение будет за пределами класса

double module(); // Модуль комплексного числа, в классе только прототип

void invert() { // Обращение комплексного числа в сопряженное

img = -img;

}

void printComplex() { // Вывод комплексного числа

cout &lt< real &lt< '+' &lt< img &lt< 'i' &lt< endl;

}

};

Complex::Complex(double a, double b) { // Конструктор определен будет вне класса

real = a;

img = b;

}

double Complex::module() { // Метод определен вне класса

return sqrt(real\*real + img\*img);

}

Complex Complex::operator \* (Complex c) { // Оператор определен вне класса

Complex tmp;

tmp.real=(real\*c.real)-(img\*c.img);

tmp.img=(real\*c.img)+(img\*c.real);

return tmp;

}

int main() {

Complex a(2,3);

Complex b(3,2);

a.printComplex();

b.printComplex();

Complex c;

c = a+b; // Перегруженный оператор сложения в действии

c.printComplex();

Complex z; // Создастся нулевое число конструктором по умолчанию

z.printComplex();

return 0;

}

##### **1. Зачем нужен виртуальный деструктор?**

**Ответ:** Чтобы избежать возможной утечки ресурсов или другого неконтролируемого поведения объекта, в логику работы которого включен вызов деструктора.  
**Пример:**

**class** Base

{

**public**:

**virtual** ~Base()

{

std::cout << "Hello from ~Base()" << std::endl;

}

};

**class** Derived : **public** Base

{

**public**:

**virtual** ~Derived()

{

*// Здесь могла бы быть очистка ресурсов*

std::cout << "Hello from ~Derived()" << std::endl;

}

};

Base \*obj = **new** Derived();

**delete** obj;

**Output:**  
Hello from ~Derived()  
Hello from ~Base()  
  
Без ключевого слова **virtual** у родительского класса **Base** деструктор порожденного класса не был бы вызван. Т.е. вызвался бы только **~Base()**:  
  
**Output:**  
Hello from ~Base()  
  
Более подробно об этом написал [GooRoo](https://habrahabr.ru/users/gooroo/) [здесь](http://habrahabr.ru/blogs/cpp/64280/). Наглядный пример возможной утечки в следующем вопросе.

##### **2. Что стоит помнить при использовании исключений в конструкторе объекта?**

**Ответ:** Если исключение не обработано, то c логической точки зрения разрушается объект, который еще не создан, а с технической, так как он еще не создан, то и деструктор этого объекта **не будет** вызван.  
**Пример:**

**class** Base

{

**private**:

HANDLE m\_hFile;

**public**:

Base()

{

std::cout << "Hello from Base()" << std::endl;

m\_hFile = ::CreateFileA(...);

*// Вызываем код, который в ходе своего выполнения бросает исключение*

SomeLib.SomeFunc(...);

}

**virtual** ~Base()

{

std::cout << "Hello from ~Base()" << std::endl;

*// Здесь мы планировали закрыть хэндл*

::CloseHandle(m\_hFile);

}

};

**try**

{

Base b;

}

**catch**(**const** std::exception &e)

{

std::cout << "Exception message: " << e.what() << std::endl;

}

**Output:**  
Hello from Base()  
Exception message: Something failed

Я немного модифицировал предыдущий пример, чтобы проблема была наглядней. Здесь объект **m\_hFile** (если был открыт) утечет т.к. до **CloseHandle()** выполнение не дойдет. Т.е. имеем такие же проблемы как в первом примере: возможная утечка ресурсов или другие проблемы из-за нарушения логики работы класса.  
  
Здесь могут спросить: «Как бы вы поступили при подобной ситуации».

1)

Have exceptions fully managed in the constructor:

class Foo

{

public:

Foo()

try

{

p = new p;

throw /\* something \*/;

}

catch (...)

{

delete p;

throw; //rethrow. no memory leak

}

private:

int \*p;

};

Правильный ответ: «Воспользовался бы умными указателями». **Простой** пример умного указателя:

class Resource

{

public:

Resource()

{

std::cout << "constructing resource" << std::endl;

}

~Resource()

{

std::cout << "destroying resource" << std::endl;

}

};

class Owner

{

::std::unique\_ptr <Resource> theResource;

public:

Owner()

: theResource(new Resource())

{

std::cout << "Map some huge resources\n";

throw "hi";

}

~Owner()

{

std::cout << "Free some huge resources\n";

}

};

Теперь и без вызова деструктора **Base** хэндл будет закрыт, т.к. при уничтожении класса **Base** будет уничтожен объект **m\_hFile** класса **CHandle**, в деструкторе которого и будет закрыт хэндл.   
  
Изобретать велосипед, конечно, не надо, все уже написано до нас, это пример который можно написать на бумажке при соответствующем вопросе. А так есть **boost**, **Loki**, **ATL** и т.п., где это уже реализовано.

In modern C++ programming, the Standard Library includes smart pointers, which are used to help ensure that programs are free of memory and resource leaks and are exception-safe.  The main goal of this idiom is to ensure that resource acquisition occurs at the same time that the object is initialized, so that all resources for the object are created and made ready in one line of code. In practical terms, the main principle of RAII is to give ownership of any heap-allocated resource—for example, dynamically-allocated memory or system object handles—to a stack-allocated object whose destructor contains the code to delete or free the resource and also any associated cleanup code.

In most cases, when you initialize a raw pointer or resource handle to point to an actual resource, pass the pointer to a smart pointer immediately. In modern C++, raw pointers are only used in small code blocks of limited scope, loops, or helper functions where performance is critical and there is no chance of confusion about ownership.

The following example compares a raw pointer declaration to a smart pointer declaration.

void UseRawPointer()

{

// Using a raw pointer -- not recommended.

Song\* pSong = new Song(L"Nothing on You", L"Bruno Mars");

// Use pSong...

// Don't forget to delete!

delete pSong;

}

void UseSmartPointer()

{

// Declare a smart pointer on stack and pass it the raw pointer.

unique\_ptr<Song> song2(new Song(L"Nothing on You", L"Bruno Mars"));

// Use song2...

wstring s = song2->duration\_;

//...

} // song2 is deleted automatically here.

##### **3. Для каких целей применяется ключевое слово**const**?**

**Ответ:** 

1. Позволяет задать константность объекта
2. Позволяет задать константность указателя
3. Позволяет указать, что данный метод не модифицирует члены класса, т.е. сохраняет состояние объекта

**Пример 1. Не можем изменить значение объекта:**

**const** **int** i = 1;

i = 2; *// error C3892: 'i' : you cannot assign to a variable that is const*

**Пример 2. Не можем изменить указатель на объект:**

**int** i = 1;

**int**\* **const** **j**(&i);

**int** k = 2;

\*j = k; *// Ok*

j = &k; *// error C3892: 'j' : you cannot assign to a variable that is const*

**Пример 3. Не можем изменить члены класса:**

**class** Foo

{

**private**:

**int** i;

**public**:

**void** **func**() **const**

{

i = 1; *// error C3490: 'i' cannot be modified because it is being accessed through a const object*

}

};

Дополнение: константный метод может изменять члены класса, если они объявлены как **mutable**. Подробнее про mutable можно прочитать у Алены CPP [здесь](http://alenacpp.blogspot.com/2005/10/mutable-constcast.html).

##### **6. Как защитить объект от копирования?**

**Ответ:** Сделать **private** конструктор копирования и оператор **=**.  
**Пример:**

**class** NonCopyable

{

**public**:

NonCopyable(){}

**private**:

NonCopyable(NonCopyable&){}

**private**:

**void** **operator**=(**const** NonCopyable&){}

};

NonCopyable a;

NonCopyable b = a; *// error C2248: 'NonCopyable::NonCopyable' : cannot access private member*

a = b; *// error C2248: 'NonCopyable::operator =' : cannot access private member*

##### **7. В чем разница между**struct**и**class**?**

**Ответ:** Практически ни в чем. В **struct** модификаторы доступа по умолчанию **public**, в **class** **private**. Также отличается и наследование по умолчанию, у **struct** — **public**, у **class** — **private**.  
**Пример:**

**struct** Foo

{

**int** i;

};

**class** Bar

{

**int** i;

};

Foo a;

a.i = 1; *// Ok*

Bar b;

b.i = 1; *// error C2248: 'Bar::i' : cannot access private member declared in class 'Bar'*

##### **16. Что такое чисто виртуальный метод и абстрактный класс?**

**Ответ:** Чисто виртуальный метод — это метод, у которого отсутствует реализация. Абстрактный класс — это класс имеющий хотя бы один чисто виртуальный метод. Как следствие, экземпляр подобного класса не может быть создан т.к. отсутствует реализация виртуального метода.  
**Пример:**

*// Абстрактный класс*

**class** Foo

{

**public**:

*// Чисто виртуальный метод*

**virtual** **void** **func**() = 0;

};

**class** Bar : **public** Foo

{

**public**:

**virtual** **void** **func**()

{

}

};

Foo f; *// error C2259: 'Foo' : cannot instantiate abstract class*

Bar b; *// Ok*

##### **17. Для чего используется вызов throw без аргументов?**

**Ответ:** Для повторного возбуждения предыдущего исключения и направления его следующему обработчику.  
**Пример:**

**try**

{

*//....*

**try**

{

*// Call something*

}

**catch**(**const** std::exception& )

{

*// Make/Check something..*

**throw**; *// Пересылаем исключение следующему обработчику*

}

*//...*

}

**catch**(**const** std::exception& e)

{

std::cout << e.what() << std::endl;

}

##### **18. В чем различия между delete и delete[]?**

**Ответ:** **delete** предназначен для уничтожения объектов, память под которые выделена при помощи **new()**. **delete[]** для объектов выделенных при помощи оператора **new[]()**.   
**Пример:**

**class** Foo

{

};

Foo \*pFoo = **new** Foo();

**delete** pFoo;

Foo \*pFooArray = **new** Foo[10]();

**delete**[] pFoo;

При неправильном использовании оператора **delete** (например, **delete** вместо **delete[]**) результат будет: **undefined behavior**.

##### **20. Для чего используется ключевое слово volatile?**

**Ответ:** Для указания компилятору, что доступ к переменной может осуществляться из мест, неподконтрольных ему. А как следствие, что работу с данной переменной не нужно подвергать разного рода оптимизациям.  
**Пример:**

**volatile** **int** i = 1; *// Независимо от прочего кода, данная переменная не будет оптимизирована.*

Т.е. если **volatile** присутствует в каком-то условии, которое не меняется со временем, то компилятор может оптимизировать его, чтобы избежать ненужных проверок, при использовании **volatile** компилятор скорее всего не будет этого делать.  
**Пример:**

**while** (1)

{

**if**(i == 1)

{

*// Какой-то код не изменяющий i*

}

}

*// Если бы volatile отсутствовало, то компилятор мог бы переделать код на что-то аля:*

**if**(i == 1) *// Нет необходимости проверять i все время, если и так известно, что оно не изменяется*

{

**while** (1)

{

*// Какой-то код не изменяющий i*

}

}