**Заголовки** - **Arial, 12, темно-красный, bold** (чтобы закинуть в менюшку слева, нужно для начала выделить нужный текст и выбрать стиль слева сверху, а потом уже менять его, как нужно)

**Текст** - Arial, 11, черный

#### **Если вам что-то не нравится в билете ~~то это ваши проблемы~~ напишите где-нибудь здесь исправление, а уже потом решим, чё оставить. Не забудьте создать новый фолоап дискашн**

Билеты к экзамену. С\С++ 2 семестр

**1. Встроенные​ типы данных. Представление чисел в памяти. Прямой, обратный и дополнительный код. Числа с плавающей точкой.**

**1 byte:** bool(появился в с++, в си не было) , char,small \_\_int8 (char in range -128:127),unsigned\_\_int8(char in range 0:255), signed char(-128:127), unsigned char (0:255)

**2 byte:** short, \_\_int16(-:+), unsigned \_\_int16(0:+), unsigned short(0:+)

**4 byte:** int, unsigned int, \_\_int32(-,+), unsigned \_\_int32(0:+), long(-,+), unsigned long(0:+), float,

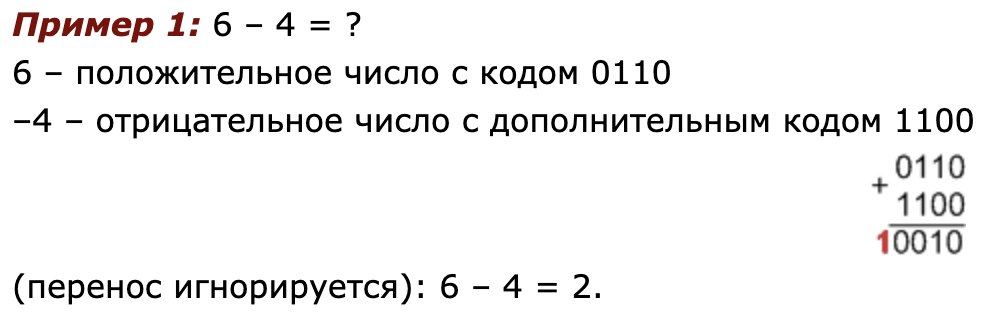
**8 byte:** \_\_int64, unsigned \_\_int64, long long (\_\_int64), unsigned long long, double

**void**: Синтаксически тип void является фундаментальным типом. Этот тип используется либо для указания того, что функция не возвращает значения, либо в качестве параметра функции обозначает отсутствие параметров. Не применяется для объявления переменных, значит не занимает место в памяти

***Прямой код*** представляет собой одинаковое представление значимой части числа для положительных и отрицательных чисел и отличается только знаковым битом. В прямом коде число 0 имеет два представления «+0» и «–0».

***Обратный код*** для положительных чисел имеет тот же вид, что и прямой код, а для отрицательных чисел образуется из прямого кода положительного числа путем инвертирования всех значащих разрядов прямого кода. В обратном коде число 0 также имеет два представления «+0» и «–0».

***Дополнительный код*** для положительных чисел имеет тот же вид, что и прямой код, а для отрицательных чисел образуется путем прибавления 1 к обратному коду. Добавление 1 к обратному коду числа 0 дает единое представление числа 0 в дополнительном коде. Однако это приводит к асимметрии диапазонов представления чисел относительно нуля.



В языке С значения с плавающей запятой это 32-битные контейнеры.

1 бит для хранения знака

8 бит для экспоненты (интервал между двумя соседними степенями двойки)

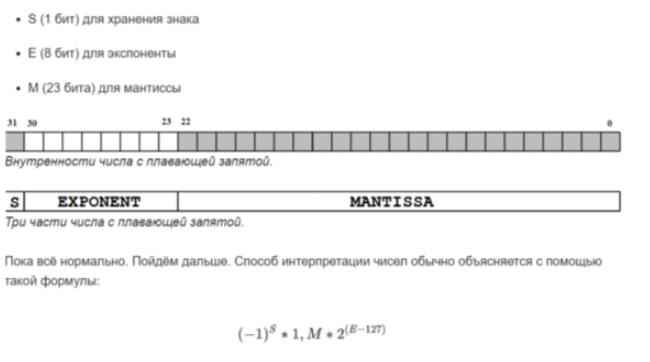
23 бита для мантиссы (дробная часть логарифма числа)(смещение в этом интервале)

Переменные типа с плавающей точкой отлично подходят для хранения очень больших или очень маленьких (в том числе и дробных) чисел до тех пор, пока они имеют ограниченное количество значащих цифр (не превышают точность определенного типа данных).

Всякое вещественное число X можно записать в виде произведения мантиссы m и основания системы счисления в некоторой целой степени n, которую называют порядком:

.

Например, число 25,324 можно записать в таком виде: 0,25324 • 10^2. Здесь m = 0,25324 — мантисса, n = 2 — порядок. Порядок указывает, на какое количество позиций и в каком направлении должна сместиться десятичная запятая в мантиссе.



#### **2. Хранение данных в памяти. Доступ к данным в памяти. Указатели. Операции с указателями.**

**Статическая память** — это область памяти, *выделяемая при запуске программы* до вызова функции *main* из свободной оперативной памяти для размещения глобальных и статических объектов, а также объектов, определённых в пространствах имён.

|  |
| --- |
| Объект называют **глобальным**, если он определен вне функции, класса или пространства имен. Объект, определенный с использованием ключевого слова *static*, называют **статическим**.  Глобальные объекты, а также объекты, определённые в пространствах имён или статически в классах, размещаются в памяти (конструируются) до вызова функции *main*, а разрушаются после завершения работы этой функции.  Пространства имён — это способ разделения программы на логические составляющие; механизм ограничения области видимости имен. |

**Автоматическая память** — это специальный регион памяти, *резервируемый при запуске программы* до вызова функции *main* из свободной оперативной памяти и *используемый в дальнейшем* для размещения локальных объектов: объектов, определяемых в теле функций и получаемых функциями через параметры в момент вызова. Автоматическую память часто называют **стеком**.

В автоматической памяти(стеке) инициализируются любые не динамические структуры и типы данных. Так в стек помещается информация о создаваемых переменных в различных функциях, которые впоследствии будут удалены, по завершении работы функции.

**Динамическая память** — это совокупность блоков памяти, выделяемых из доступной свободной оперативной памяти непосредственно *во время выполнения программы* под размещение конкретных объектов.

Динамическая память нужна для того, чтобы любые динамически выделяемые типы данных можно было бы куда-то поместить, при этом указатель на начало информации в динамической памяти хранится в стеке.

Обращаться к данным в памяти мы можем посредством указателей, ссылок, а также прямым обращением к переменной, которая была инициализирована в нашей программе.

**Указатель** - хранит адрес другого объекта, а адрес — это номер ячейки памяти,в 32-разрядных системах он занимает 4 байта памяти, а в 64 - 8(поясн. 32 бита == 4 байта, 64 бита == 8 байт)

**Операции с указателями:**

**Операция присваивания** - присваивает указателю адрес какого-либо объекта в памяти. Можно присваивать нулевое значение(NULL, nullptr etc.) , потому что тогда это говорит компилятору, что наш указатель ни на что не указывает.

**Можно определять ссылки на указатели, но не наоборот( информация в принципе бесполезная)**

**Разыменование указателя -** Прямое обращение в ячейку памяти, на которую указывает наш указатель, делается посредством указания на указатель.

**Операция сравнения -** можно сравнивать указатели только одного типа , сравниваются при этом адреса, а не сами значения по адресам.

**Инкремент, декремент, сложение , вычитание -** с указателем можно сделать операции инкремента и декремента: pi++, pi--, что приведёт к смещению на size byte вперёд или назад, в зависимости от типа данных , к которому принадлежит указатель, так же pi + const, pi - const, смещает на n\*size byte позиций наш указатель, т.е. pi + 2 сместит наш указатель на 2\*size byte вперёд, пример : int \*pi = &a, pi += 2; сместит нас на 8 байт вперед от текущего положения. Вычитание так же можно применять к указателям одного типа, это покажет, сколько позиций между ними или сколько byte/size byte ячеек памяти того типа, которым являются наши указатели между ними.

#### **3. Массивы. Объявление и инициализация. Устройство в памяти. Операции над массивами. Двумерные массивы. Арифметика указателей.**

Количество элементов массива задается при его объявлении и заключается в квадратные скобки.

Чтобы описать элементы массива сразу при его создании, можно использовать фигурные скобки. В фигурных скобках значения элементов массива перечисляются через запятую. В конце закрывающей фигурной скобки ставится точка с запятой.

Массив типа int из 10 элементов описывается с помощью адреса его первого элемента и количества байт, которое может вместить этот массив. Если для хранения одного целого числа выделяется 4 байта, то для массива из десяти целых чисел будет выделено 40 байт.

**Память под массив может выделяться автоматически или динамически.**

Автоматическое выделение памяти используют, когда размер массива **известен** на этапе компиляции (т. е. при написании кода).

Динамическое выделение памяти используют, когда размер массива **неизвестен** на этапе компиляции (допустим, запрашивается у пользователя).

Для массивов, использующих динамическое выделение памяти, память распределяется из «кучи».

Глобальные и локальные автоматические массивы различаются тем, что для локальных память выделяется в **стеке,** а он может **переполниться**, так что следи за тем, что делаешь))

С глобальными автоматическими все ок, тк память выделяется **статическая** *(до мейна из свободной оперативной)*

*ОПЕРАЦИИ НАД МАССИВАМИ:*

* *обращение к элементу*
* *вывод/ввод элементов массива*
* *вычисление суммы элементов массива*
* *поиск максимального/минимального элемента и его номер*
* *сортировка*
* *удаление элемента*

***int A[n][m]****. Данное объявление создает массив из n объектов, каждый из которых в свою очередь является массивом типа* ***int [m]****. Тогда* ***A[i]****, где i принимает значения от 0 до n−1 будет в свою очередь одним из n созданных обычных массивов, и обратиться к элементу с номером j в этом массиве можно через* ***A[i][j]****.*

*АРИФМЕТИКА УКАЗАТЕЛЕЙ:*

**Инкремент, декремент, сложение , вычитание -** с указателем можно сделать операции инкремента и декремента: pi++, pi--, что приведёт к смещению на size byte вперёд или назад, в зависимости от типа данных , к которому принадлежит указатель, так же pi + const, pi - const, смещает на n\*size byte позиций наш указатель, т.е. pi + 2 сместит наш указатель на 2\*size byte вперёд, пример : int \*pi = &a, pi += 2; сместит нас на 8 байт вперед от текущего положения. Вычитание так же можно применять к указателям одного типа, это покажет, сколько позиций между ними или сколько byte/size byte ячеек памяти того типа, которым являются наши указатели между ними.

#### **4. Ссылки. Разница между ссылкой и указателем. Зачем в С++ были добавлены ссылки.**

Ссылка - по своей сути является псевдонимом какого-либо объекта.

Разница между ссылкой и указателем - ссылка всегда ссылается на какой-то объект, что помогает исключить обращение к нулевому значению(NULL, nullptr etc.), хоть ссылки и реализованы с помощью указателей, её плюс в том, что не нужно её явно разыменовывать. Главное их отличие в том, что указатель допускает над собой операции сложения, вычитания и пр. А ссылка допускает только копирование и присваивание.

Ссылки были добавлены для того, чтобы упростить работу с перегрузками и пр., ведь ссылка всегда ссылается на какой-то объект и что-то пустое мы никак не передадим, с помощью ссылок мы можем спокойно работать с передаваемыми данными в функции и не заботиться о том, чтобы как-то разбирался с указателями.

#### **5. Сборка программ. Препроцессор, компилятор, компоновщик. Сборка проекта, состоящего из нескольких файлов. Макросы**

Сборка программы С++ - это компиляция исходного кода из одного или нескольких файлов и последующее связывание этих файлов в исполняемый файл.

Процесс базовой компиляции С++ состоит из трёх основных этапов:

1) **Препроцессинг**

Самая первая стадия компиляции программы.(работа с текстом)

Препроцессор — это *макропроцессор*, который преобразовывает вашу программу для дальнейшего компилирования. На данной стадии происходит работа с препроцессорными директивами. Например, препроцессор добавляет хэдеры в код (#include), убирает комментирования, заменяет макросы (#define) их значениями, выбирает нужные куски кода в соответствии с условиями #if, #ifdef и #ifndef.

* Замена комментариев пустыми строками
* Текстовое включение файлов — #include
* Макроподстановки — #define
* Обработка директив условной компиляции — #if, #ifdef, #elif, #else, #endif

2) **Компиляция**

На данном шаге компилятор выполняет свою главную задачу — компилирует, то есть преобразует полученный на прошлом шаге код без директив в *ассемблерный код*. Это промежуточный шаг между высокоуровневым языком и машинным (бинарным) кодом.

Ассемблерный код — это доступное для понимания человеком представление машинного кода.

Процесс компиляции состоит из следующих этапов:

1. **Лексический анализ.** Последовательность символов исходного файла преобразуется в последовательность лексем.
2. **Синтаксический анализ.** Последовательность лексем преобразуется в дерево разбора.
3. **Семантический анализ.** Дерево разбора обрабатывается с целью установления его семантики (смысла) — например, привязка идентификаторов к их декларациям, типам, проверка совместимости, определение типов выражений и т. д.
4. **Оптимизация.** Выполняется удаление излишних конструкций и упрощение кода с сохранением его смысла.
5. **Генерация кода.** Из промежуточного представления порождается объектный код.

Результатом компиляции является **объектный код.**

**Объектный код** — это программа на языке машинных кодов с частичным сохранением символьной информации, необходимой в процессе сборки.

3) **Компоновка**

Компоновщик (линкер) связывает все объектные файлы и статические библиотеки в единый исполняемый файл, который мы и сможем запустить в дальнейшем. Для того, чтобы понять как происходит связка, следует рассказать о *таблице символов*.

Таблица символов — это структура данных, создаваемая самим компилятором и хранящаяся в самих объектных файлах. Таблица символов хранит имена переменных, функций, классов, объектов и т.д., где каждому идентификатору (символу) соотносится его тип, область видимости. Также таблица символов хранит адреса ссылок на данные и процедуры в других объектных файлах.

Именно с помощью таблицы символов и хранящихся в них ссылок линкер будет способен в дальнейшем построить связи между данными среди множества других объектных файлов и создать единый исполняемый файл из них.

1. **Препроцессор** С++ преобразует все определения директив и макросов в каждом исходном файле.
2. **Компилятор** С++ компилирует каждую единицу трансляции в объектные файлы, применяя заданные параметры компилятора
3. **Компоновщик** объединяет объектные файлы в один исполняемый файл, применяя заданные параметры компоновщика

Компиляция нескольких файлов:

1. При сборке проекта компилируются только изменённые cpp-файлы.
2. Внесение изменений в h-файл приводит к перекомпиляции всех cpp-файлов, в которые он включён.
3. Если перенести определения функций и методов классов в cpp-файлы, то они будут пересобираться только после изменений.
4. Это ускоряет пересборку проекта.

**Макрос**(Макрокоманда) - программный алгоритм действий, записанный пользователем. Часто макросы применяют для выполнения рутинных действий. (типо #define BUFFER\_SIZE 1020)

#### **6. Запуск программ. Процессы и потоки. Виды памяти. Стек вызова функций.**

**Процесс** – это некоторая часть работы ОС, обладающая уникальным идентификационным номером – id, и адресным пространством. Адресное пространство – некоторый список адресов в памяти, с которыми происходит работа этого процесса. С другими адресами процессу приходится работать через системный вызов. Одна программа может включать как несколько процессов, так и один, причем последнее используется наиболее часто. Разбиение на процессы позволяет распараллелить задачи, благодаря чему ускорить работу, но в большинстве случаев для этого проще и выгоднее использовать потоки, которые намного быстрее взаимодействуют друг с другом и обладают рядом других положительных моментов, что и привело к меньшей используемости процессов. **Процесс** - создание некого окружения или оболочки для работы с потоками со стороны ОС под исполняемый файл, запущенный на выполнение, чтобы внутри этого процесса началось выполнение какого-то исполняемого файла.

**Поток** – это часть уже самого процесса, выполняющая определенный список действий. У каждого процесса есть как минимум один поток, и их увеличение обеспечивает распараллеливание процесса. Чем выгодно такое увеличение потоков внутри процесса увеличения количества самих процессов? **Каждый поток, как часть процесса, имеет доступ ко всему адресному пространству процесса**, ко всем его устройствам и переменным. Поэтому взаимодействие двух отдельных потоков реализуется очень просто и не требует обращения к системе. По этой причине использование потоков более распространено, чем процессов.

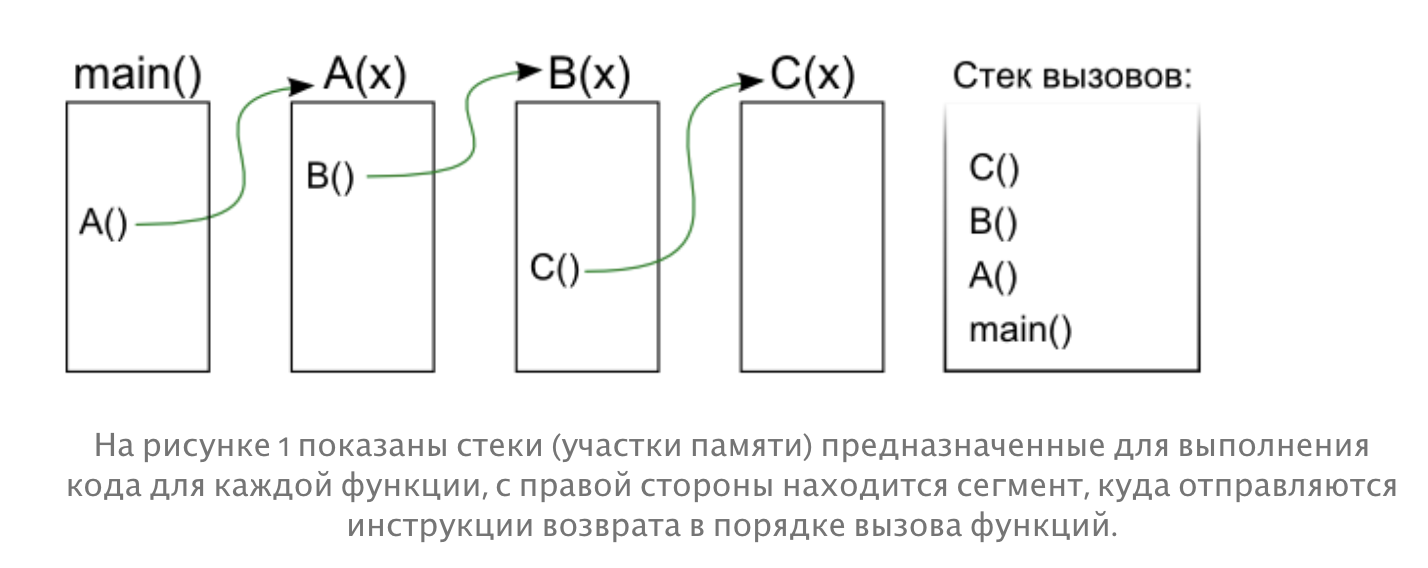
Сам поток представляет собой **стек команд со счетчиком**, обладающий несколькими важными свойствами, такими как состояние и приоритет. Состояний потока всего три:

**1.** состояние **активности**, то есть поток выполняется на данный момент

**2.** состояние **неактивности**, когда поток ожидает выделения процессора для перехода в состояние активности

**3.** состояние **блокировки**, когда потоку не выделяется время (соответственно он не занимает место в очереди, освобождая ресурсы) вне зависимости от его приоритета.

Сегмент стека (который мы сейчас по простоте называем стеком вызова) это отдельный участок памяти, в котором хранятся инструкции о возврате результатов вызванных функции. Используется метод: кто пришел последний — уйдет первый (LIFO).



(типо если в мейне вызывается А, а в ней В, а в ней С, то сначала выполнится С, вернет результат в В, выполнится В, вернёт рез в А и тд…)

Виды памяти: статическая, автоматическая, динамическая

#### **7. ООП. Базовые принципы. Абстракция, инкапсуляция, наследование полиморфизм. Принцип подстановки Барбары-Лисков**

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

Базовые принципы структурирования в ООП:

1. ***Инкапсуляция*** – это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и скрыть детали реализации от пользователя.

Представьте, что водитель едет в автомобиле по оживленному участку движения. Понятно, что в этот момент он не будет задумываться о химическом составе краски автомобиля, особенностях взаимодействия шестерён в коробке передач или влияния формы кузова на скорость (разве что, автомобиль стоит в глухой пробке и водителю абсолютно нечем заняться). Однако, руль, педали, указатель поворота он будет использовать регулярно.

1. ***Абстрагирование*** – это способ выделить набор значимых характеристик объекта, исключая из рассмотрения незначимые. Соответственно, абстракция – это набор всех таких характеристик.
2. ***Наследование*** – это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым или родительским. Новый класс – потомком, наследником или производным классом.

Необходимо отметить, что производный класс полностью удовлетворяет спецификации родительского, однако может иметь дополнительную функциональность. С точки зрения интерфейсов, каждый производный класс полностью реализует интерфейс родительского класса. Обратное не верно.

1. ***Полиморфизм*** – это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

**(Полиморфизм - возможность работать с разными типами одинаковым образом. Статический - перегрузка функций, шаблоны. Динамический - виртуальные функции.)**

Принцип подстановки Барбары Лисков: Пусть *q*(*x*) является свойством верным относительно объектов *x* некоторого типа *Т*. Тогда *q*(*y*) также должно быть верным для объектов y типа *S*, где *S* является подтипом типа *Т*.

**Что это за хуйня значит на практике:** Если у нас есть класс А (не виртуальный, а вполне реально используемый в коде) и унаследованный от него класс В, то если мы заменим все использования класса А на В, ничего не должно измениться в работе программы. Ведь В всего лишь расширяет функционал класса А. Если эта проверка работает, то поздравляю: ваша программа соответствует подстановки Лисков! Если нет, стоит ~~написать ПСЖ~~ задуматься: “а правильно ли спроектированы классы?”

#### **8. ООП. Классы. Устройство в памяти. Инкапсуляция. Модификаторы доступа. Наследование. Множественное наследование. Проблемы множественного наследование.**

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

**Классы** - Совокупность объектов и методов их обработки. в программировании состоят из свойств и методов. Методы класса - его функции. Свойства класса - его переменные. По сути классы являются типами данных, определёнными программистом. Это абстракция, описывающая методы и свойства, ещё не существующих объектов.

**Устройство классов в памяти** зависит от объектов и его расположения внутри, тк существует такая штука, как выравнивание, которая выравнивает память для объектов. Таким образом, если сначала будет стоять тип, занимающий меньше памяти, то он будет выровнен до следующего (большего) типа. Поэтому для оптимального использования памяти при создании классов желательно располагать объекты от бОльших типов к мЕньшим.

***Инкапсуляция*** – это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и скрыть детали реализации от пользователя.

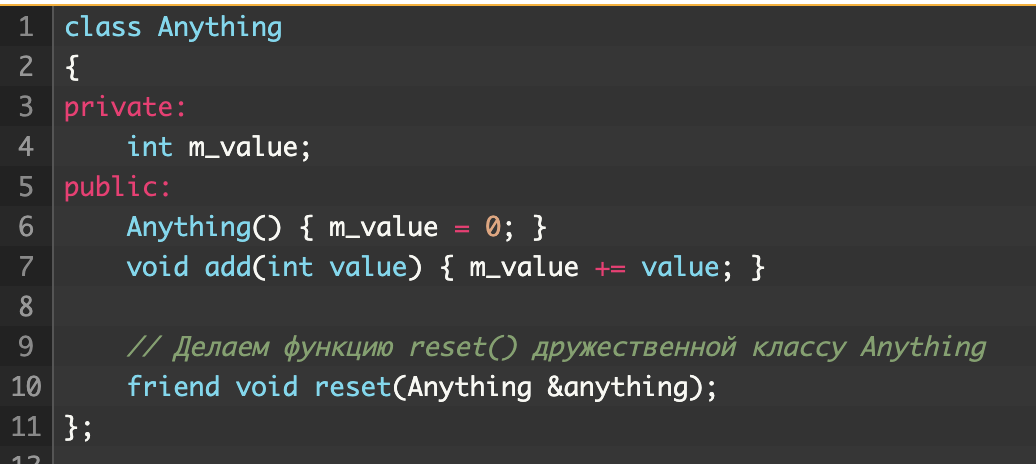
( это использование private полей в классах)

**Модификаторы(спецификаторы доступа):**

**public**: делает члены открытыми;

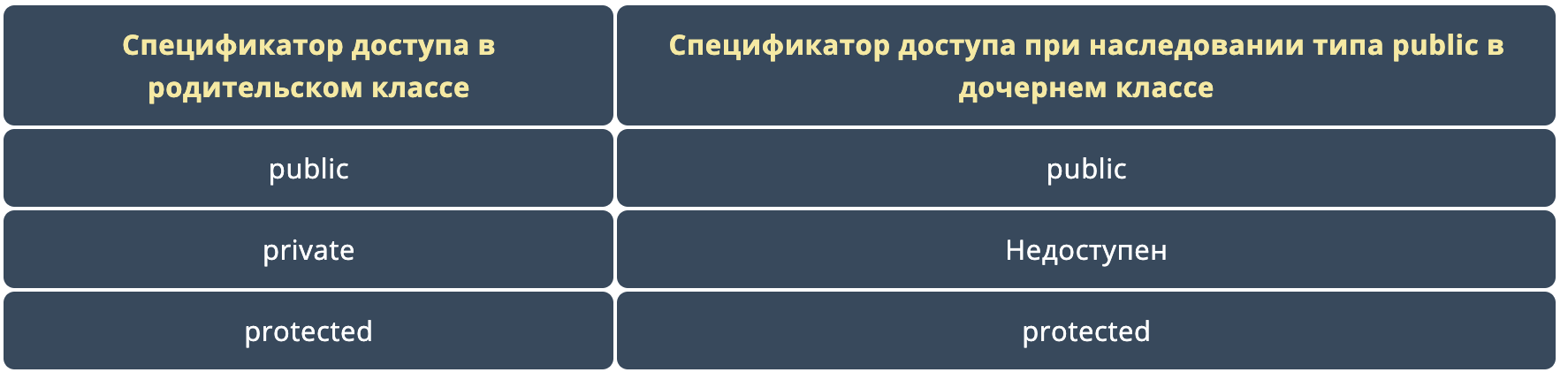
**private**: делает члены закрытыми;

**protected:** открывает доступ к членам только для дружественных и дочерних классов (Доступ к члену protected вне тела класса закрыт.)



дружественный класс, который имеет доступ к прайват полям класса Anything

**Существует три типа наследований классов:**

****

**Множественное наследование** позволяет одному дочернему классу иметь несколько родителей. Предположим, что мы хотим написать программу для отслеживания работы учителей. Учитель — это Human. Тем не менее, он также является Сотрудником (Employee).

**Проблема множественного наследования:**

1. могут возникать проблемы с одинаковыми именами в родительских классах
2. “Алмаз смерти” (ахпх) когда один класс наследуется от двух других, которые в свою очередь наследуются от одного и того же. Проблема в том, что могут быть ситуации, когда в наш самый нижний дочерний класс инфа будет приходить 2 раза.

#### **9. ООП. Классы. Инициализация и уничтожение. Виды конструкторов. Виртуальные методы. Таблица виртуальных функций. Виртуальные деструкторы.**

1. **ООП** - методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.
2. **Классы** Совокупность объектов и методов их обработки. В программировании состоят из свойств и методов. Методы класса - его функции. Свойства класса - его переменные. По сути классы являются типами данных, определёнными программистом. Это абстракция, описывающая методы и свойства, ещё не существующих объектов.
3. **Инициализация и уничтожение:**

Рассмотрим следующее определение класса

*class Data{*

*public:*

*int ival;*

*char \*ptr;};*

Если с его помощью описываются служащие компании, то ptr, вероятно, указывает на фамилию служащего, а ival - его уникальный номер. Тогда отрицательные или нулевые значения ошибочны. Инициализация членов класса зависит от назначения данного класса. Чтобы компилятор понимал наши намерения, мы должны предоставить одну или несколько перегруженных функций инициализации - конструкторов. Подходящий конструктор выбирается в зависимости от множества начальных значений, указанных при определении объекта.

Мнемонические имена класса и обоих его членов сделали бы, конечно, его назначение более понятным для читателя программы, но не дали бы никакой дополнительной информации компилятору. Чтобы компилятор понимал наши намерения, мы должны предоставить одну или несколько перегруженных функций инициализации - **конструкторов.**

**Конструктор** - функция инициализации

**Деструктор** - специальная определяемая пользователем функция-член, которая автоматически вызывается, когда объект выходит из области видимости или когда к указателю на объект применяется операция delete.

1. **Виды конструкторов:**
2. конструктор с параметрами
3. конструктор по умолчанию, не принимающий аргументов
4. конструктор копирования - конструктор, принимающий в качестве аргумента объект того же класса (или ссылку на него)
5. Конструктор преобразования - конструктор, принимающий один аргумент. Задает преобразование типа своего аргумента в тип конструктора
6. Конструктор перемещения - конструктор, принимающий на входе значение не константной ссылки на объект класса, и используется для передачи владения ресурсами этого объекта.

5. **Виртуальный метод** - в ООП метод (функция) класса, который может быть переопределен в классах-наследниках так, что конкретная реализация метода для вызова будет определяться во время исполнения.

6. **Таблица виртуальных методов** - механизм для поддержки динамического соответствия. Допустим, у нас есть класс Cat и два подкласса DomesticCat и Lion. Класс Cat определяет виртуальную функцию speak, так что его подклассы могут обеспечивать соответствующую реализацию (соответственно “мяу” или “ррр”)

7. **Виртуальный деструктор:** нужен, когда вы используете виртуальные функции.

Если у вас есть класс А и подкласс В, но деструктор будет только для базового класса, то произойдёт утечка памяти, потому как деструктор производного класса не вызывается.

Если бы мы выводили порядок вызова конструкторов и деструкторов базового и производного классов, то это выглядело бы так:

A()

B()

~A()

Происходит это потому что удаление производится через указатель на базовый класс и для вызова деструктора компилятор использует раннее связывание. Деструктор базового класса не может вызвать деструктор производного, потому что он о нём ничего не знает. Чтобы этого избежать, деструктор в базовом классе должен быть объявлен как виртуальный.

#### **10. ООП. Операторы. Перегрузка операторов.**

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

**Операторы:**

* Арифметические
* Сравнения
* Составное присваивание (+=, -=...)
* Работы с указателями и членами класса (a.b, a -> b, \*b...)
* Другие (функтор, тернарная a ? b : c…)

**Перегрузка операторов** - один из способов реализации полиморфизма, заключающийся в возможности одновременного существования в одной области видимости нескольких различных вариантов применения оператора имеющих одно и то же имя, но различающихся типами параметров, к которым они применяются.

#### **11. Исключения. Обработка ошибок в Си. Assert. Исключения. Преимущества и недостатки исключений.**

При работе программы могут возникать так называемые исключительные ситуации, когда дальнейшее нормальное выполнение программы становится невозможным. Причиной исключительных ситуаций могут быть как ошибки в программе, так и неправильные действия пользователя, неверные данные и т.д.

Здесь на помощь приходят такие средства языка, как: защитные блоки (try-блоки) и перехваты исключений (catch-блоки), а также инициализация исключений (инструкция throw)

**try-catch**:

try {операторы защищенного блока}

catch(...) {обработчик ошибочной ситуации}

**throw**:

Обработка исключений, возбужденных оператором throw, идёт по следующей схеме:

1. Создается статическая переменная со значением, заданным в операторе throw.Она будет существовать до тех пор, пока исключение не будет обработано. Если переменная-исключение является объектом класса, при ее создании работает конструктор копирования.
2. Завершается выполнение защищенного try-блока: раскручивается стек подпрограмм, вызываются деструкторы для тех объектов, время жизни которых истекает и т.д.
3. Выполняется поиск первого из catch-блоков, который пригоден для обработки созданного исключения.

Обработка ошибок в Си:

1. **errno** - переменная, хранящая целочисленный код последней ошибки. В каждом потоке существует своя локальная версия errno, чем и обусловливается её безопасность в многопоточной среде. Обычно errno реализуется в виде макроса, разворачивающегося в вызов функции, возвращающей указатель на целочисленный буфер. При запуске программы значение errno равно нулю.
2. *void perror(const char \*s);*

печатает в stderr содержимое строки s, за которой следует двоеточие, пробел и сообщение об ошибке. После чего печатает символ новой строки ‘\n’.

1. *char \*strerror(int errnum);*

Возвращает строку, содержащую описание ошибки errnum. Язык сообщения зависит от локали, но обычно поддерживается лишь английский.

1. Макрос **assert()** - макрос, проверяющий условие expression (его результат должен быть числом) во время выполнения. Если условие не выполняется (expression = 0), он печатает в stderr значения \_\_FILE\_\_, \_\_LINE\_\_, \_\_func\_\_ и expression в виде строки, после чего вызывает функцию abort()
2. Функции **atexit()**, **exit()** и **abort()**

1. *int atexit(void (\*func)(void));*

Регистрирует функции, вызываемые при нормальном завершении работы программы в порядке, обратном их регистрации. Можно зарегистрировать до 32 функций.

2. *\_Noreturn void exit (int exit\_code);*

Вызывает нормальное завершение программы, возвращает в среду число exit\_code.

3. *\_Noreturn void abort(void);*

Вызывает аварийное завершение программы, если сигнал не был перехвачен обработчиком сигналов.

1. Функции **setjmp()** и **longjmp()** - работают по принципу goto, но в отличие от него позволяют перепрыгивать из одного места в другое в пределах все программы, а не одной функции.

*int setjmp(jmp\_buf env);*

Сохраняет информацию о контексте выполнения программы (регистры микропроцессора и прочее) в evn. Возвращает 0, если была вызвана напрямую или value, если из longjmp().

*void longjmp(jmp\_buf env, int value);*

Восстанавливает контекст выполнения программы из env, возвращает управление setjmp() и передаёт ей value.

**Assert** - специальная конструкция, позволяющая проверять предположения о значениях произвольных данных в произвольном месте программы. Эта конструкция может автоматически сигнализировать при обнаружении некорректных данных, что обычно приводит к аварийному завершению программы с указанием места обнаружения некорректных данных.

Преимущества исключений:

1. Исключения отделяют код обработки ошибок от нормального алгоритма программы, тем самым повышая разборчивость, надежность и расширяемость кода.
2. Генерация исключения – единственный чистый способ сообщить об ошибке из конструктора.
3. Исключения трудно игнорировать, в отличие от кодов ошибок.
4. Исключения легко передаются из глубоко вложенных функций.
5. Исключения могут быть, и часто являются, определяемыми пользователем типами, несущими гораздо больше информации, чем код ошибки.
6. Объекты исключений сопоставляются с обработчиками с помощью системы типов.

Недостатки исключений:

1. Исключения нарушают структуру кода, создавая множество скрытых точек выхода, что затрудняет чтение и изучение кода.
2. Исключения легко вызывают утечки ресурсов, особенно в языке, не имеющем встроенного сборщика мусора и блоков finally(в конце).
3. Тяжело научиться писать безопасный код исключений.
4. Исключения дорогостоящие и нарушают обещание платить лишь за используемое.
5. Исключения тяжело ввести в устаревший код.
6. Исключения неверно используются для выполнения задач, относящихся к нормальному алгоритму программы.

#### **12. Шаблоны. Шаблоны функций, классов.**

1. **Шаблоны** - средство языка С++, предназначенное для кодирования обобщенных алгоритмов, без привязки к некоторым параметрам (типам данных, размерам буферов, значениям по умолчанию)
2. **Шаблон функции** начинается с ключевого слова template, за которым в угловых скобках следует список параметров. Затем следует объявление функции. Для вызова шаблонной функции необходимо указать значения для всех параметров шаблона.
3. **Шаблоны классов** - аналогично шаблонам функции.
4. **Функции в шаблонах** - функции могут быть определены как внутри шаблона класса, так и за его пределами.

**Шаблоны вложенных классов** - шаблоны можно определить в классах или шаблонах классов. Шаблоны вложенных классов объявляются как шаблоны классов внутри области внешнего класса. Их можно определить во включающем классе или вне его.

Дружественными объектами класса-шаблона могут быть классы или шаблоны классов, функции или шаблоны функций

#### **13. Шаблоны. Частичная специализация и полная специализация**

**Шаблоны** - средство языка С++, предназначенное для кодирования обобщённых алгоритмов, без привязки к некоторым параметрам (типам данных, размерам буферов, значениям по умолчанию)

Частичная специализация шаблона класса определяется как некая конфигурация параметров первичного класса, которая служит аргументом специализированной реализации.

Частичная специализация позволяет частично настроить код шаблона для определенных типов, например:

* Шаблон имеет несколько типов, и только некоторые из них требуют специализации. Результат для остальных типов параметризован шаблоном.
* Шаблон имеет только один тип, но специализация необходима для типов указателя, ссылки, указателя на член или указателя на функцию. Специализация сама по себе по-прежнему является шаблоном с типом, на который задан указатель или ссылка.

Такая специализация является полной

Хотим, чтобы наш класс для всех типов работал одинаково, а для int по-другому

template<> class A<int> {}; // здесь int - это аргумент шаблона

Явная полная специализация: В С++ явная специализация шаблона осуществляется через создание дополнительного класса-шаблона, в котором входные параметры первичного класса полностью или частично переопределяются конкретными типами данных

#### **14. Variadic template**

Шаблоном с переменным числом параметров называется шаблон функции или класса, принимающий так называемый parametr pack. При объявлении шаблона это выглядит следующим образом: template<typename… Args> struct some\_type;

Такая запись значит то, что шаблон может принять 0 или более типов в качестве своих аргументов. В теле же шаблона синтаксис использования немного другой: template<typename… Args> *// Объявление*

void foo(Args… args); *// Использование*

Вариадик параметр можно использовать в качестве аргумента вызова функции, применять к нему операции каста и т.п.

Количество параметров в паке можно получить используя оператор sizeof.

#### 

#### **15. Преобразования типов. Явные и неявные. С-cast, static\_cast, dynamic\_cast, const\_cast, reinterpret\_cast. CRTP**

**Преобразование типов** - преобразование значения одного типа в значение другого типа.

Неявное приведение типов происходит в следующих случаях:

1. После вычисления операндов бинарных арифметических, логических, битовых операций, операций сравнения, а также 2-го или 3-го операнда операции “?:”; значения операндов приводятся к одинаковому типу.
2. Перед выполнением присваивания.
3. Перед передачей аргумента функции.
4. Перед возвратом функцией возвращаемого значения.
5. После вычисления конструкции switch значение приводится к целочисленному типу.
6. После вычисления выражений конструкций if, for, while значение приводится к типу bool

Для явного приведения типов имя типа указывается в круглых скобках перед переменной или выражением:

int X;

int Y = 200;

char C = 30;

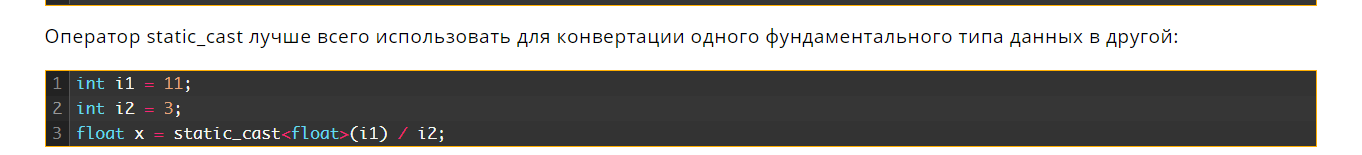
X = (int)C \* 10 + Y; *// переменная С приводится к типу int*

1. **static\_cast.**

Синтаксис: TYPE static\_cast<TYPE> (object);

static\_cast преобразует выражения одного статического типа в объекты и значения другого статического типа. Поддерживается преобразование численных типов, указателей и ссылок по иерархии наследования как вверх, так и вниз. Проверка производится на уровне компиляции, так что в случае ошибки сообщение будет получено в момент сборки приложения или библиотеки.

[Подробно про статик каст. Для чего нужен, что может и что не может](https://ru.stackoverflow.com/questions/387985/%D0%94%D0%BB%D1%8F-%D1%87%D0%B5%D0%B3%D0%BE-%D0%BD%D1%83%D0%B6%D0%B5%D0%BD-static-cast-%D0%BA%D0%B0%D0%BA-%D0%BE%D0%BD-%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%B0%D0%B5%D1%82-%D0%B8-%D0%B3%D0%B4%D0%B5-%D0%B5%D0%B3%D0%BE-%D0%BF%D1%80%D0%B8%D0%BC%D0%B5%D0%BD%D1%8F%D1%8E%D1%82)



2. **dynamic\_cast**

Синтаксис:

TYPE& dynamic\_cast<TYPE&> (object);

TYPE\* dynamic\_cast<TYPE\*> (object);

Используется для динамического приведения типов во время выполнения. В случае неправильного приведения типов для ссылок вызывается исключительная ситуация std::bad\_cast, а для указателей будет возвращен 0. Использует систему RTTI (Runtime Type Information). Безопасное приведение типов по иерархии наследования, в том числе для виртуального наследования.

3. **const\_cast.**

Синтаксис: TYPE const\_cast<TYPE> (object);

Пожалуй самое простое приведение типов. Снимает cv qualifiers — const и volatile, то есть константность и отказ от оптимизации компилятором переменной. Это преобразование проверяется на уровне компиляции и в случае ошибки приведения типов будет выдано сообщение.

4. **reinterpret\_cast**

Синтаксис: TYPE reinterpret\_cast<TYPE> (object);

Приведение типов без проверки. reinterpret\_cast — непосредственное указание компилятору. Применяется только в случае полной уверенности программиста в собственных действиях. Не снимает константность и volatile. применяется для приведения указателя к указателю, указателя к целому и наоборот.

5. **C-style cast**

Синтаксис: TYPE (TYPE\*) object;

Си-шный метод приведения типов. Пожалуй самый нежелательный способ приведения типов. Страуструп пишет:

«Например, что значит это выражение — *x = (T)y;*. Мы не знаем. Это зависит от типа T, типов x и y. T может быть названием типа, typedef или может быть параметр template-а. Может быть, х и у являются скалярными переменными и Т представляет собой значение преобразования. Может быть, х объекта класса, производного от класса Y и Т — нисходящее преобразование. По этой причине программист может не знать, что он делает на самом деле.»

Вторая причина нежелательного использования приведения типов в C-style — трудоемкость процесса поиска мест приведения типов.

[про устройсво С-style](http://artlang.net/post/c++-otlichiye-reinterpret_cast-ot-privedeniya-tipov-v-stile-c/)

**CRTP** (Curiously Recurring Template Pattern)(Странный рекурсивный шаблон) - это идиома проектирования, заключающаяся в том, что класс наследует от базового шаблонного класса с самим собой в качестве параметра шаблона базового класса. Такая конструкция делает возможным обращение к производному классу из базового.

#### **16. Стандартная библиотека. Итераторы. Алгоритмы. Основные классы алгоритмов.**

1. **Стандартная библиотека** - коллекция классов и функций, написанных на базовом языке. Стандартная библиотека поддерживает несколько основных контейнеров, функций для работы с этими контейнерами, объектов-функции, основных типов строк и потоков, поддержку некоторых языковых особенностей, и часто используемые функции для выполнения таких задач, как, например, нахождение квадратного корня числа. Функциональные особенности стандартной библиотеки объявляются внутри пространства имен std.
2. **Итератор** - интерфейс, предоставляющий доступ к элементам коллекции (массива или контейнера) и навигацию по ним.

Категории итераторов:

1. **Входные** - обеспечивают доступ для чтения в одном направлении. Позволяют выполнить присваивание или копирование с помощью оператора присваивания и конструктора копирования (operator++, operator\*, operator==, operator=, operator!=, конструктор копирования)

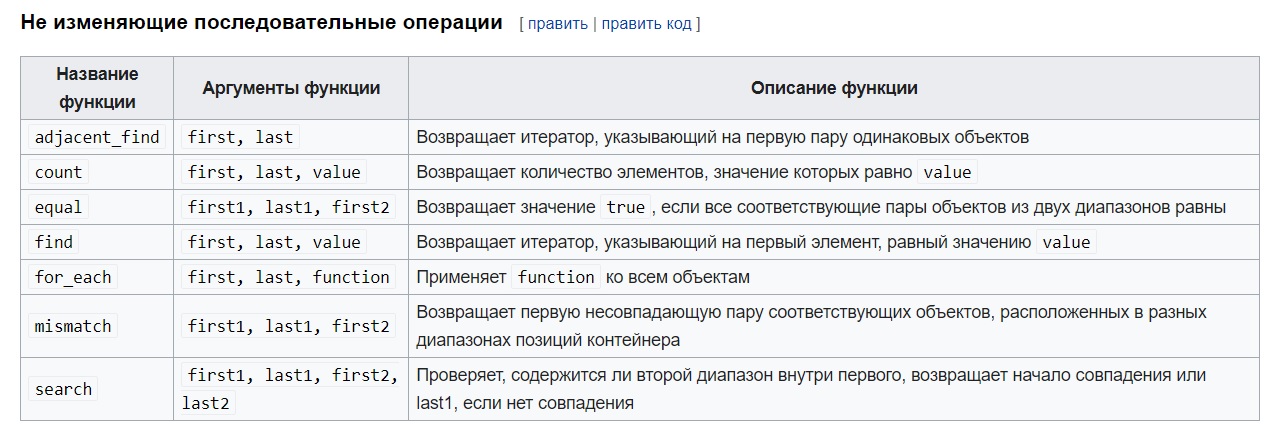
2. **Выходные** - обеспечивают доступ для записи в одном направлении. Их нельзя сравнивать на равенство (operator++, operator\*, конструктор копии)

3.**Однонаправленные** - обеспечивают доступ для чтения и записи в одном направлении. Позволяют выполнить присваивание или копирование с помощью оператора присваивания и конструктора копирования. Их можно сравнивать на равенство. (operator++, operator\*, конструктор копирования, конструктор по умолчанию, operator=, operator==, operator!=)

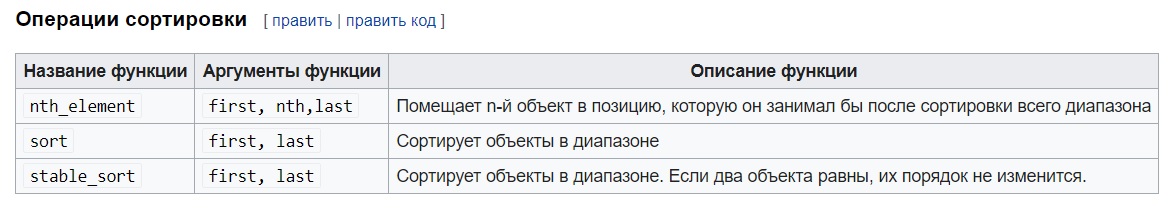
4. **Двунаправленные** - поддерживают все функции, описанные для однонаправленных итераторов. Кроме того, они позволяют переходить к предыдущему элементу. (operator++, operator--, operator\*, конструктор копирования, конструктор по умолчанию, operator=, operator==, operator!=)

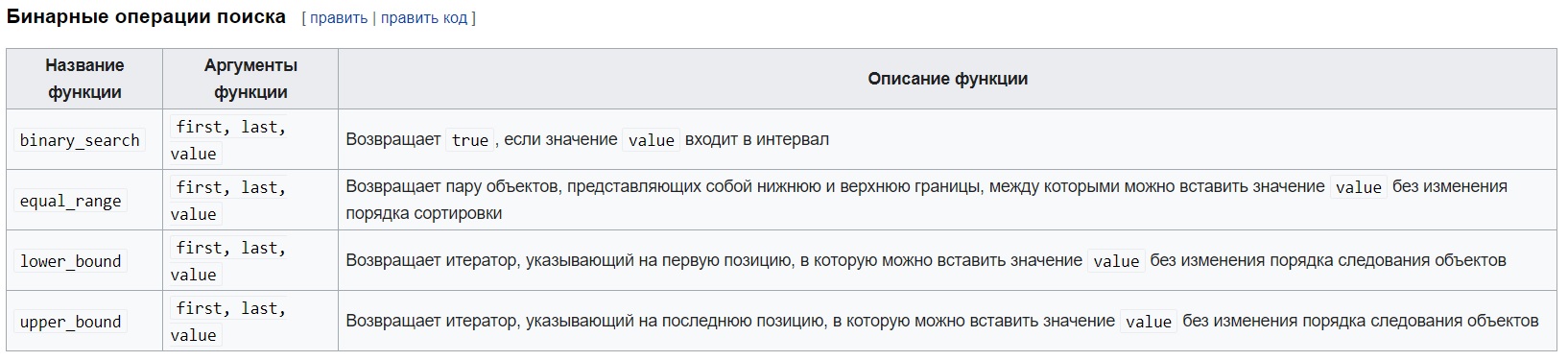
5. **Произвольного доступа** - эквивалентны обычным указателям: поддерживают арифметику указателей, синтаксис индексации массивов и все формы сравнения. (operator++, operator--, operator\*, конструктор копирования, конструктор по умолчанию, operator=, operator==, operator!=, operator+=, operator-=, operator<, operator>, operator<=, operator>=, operator[])

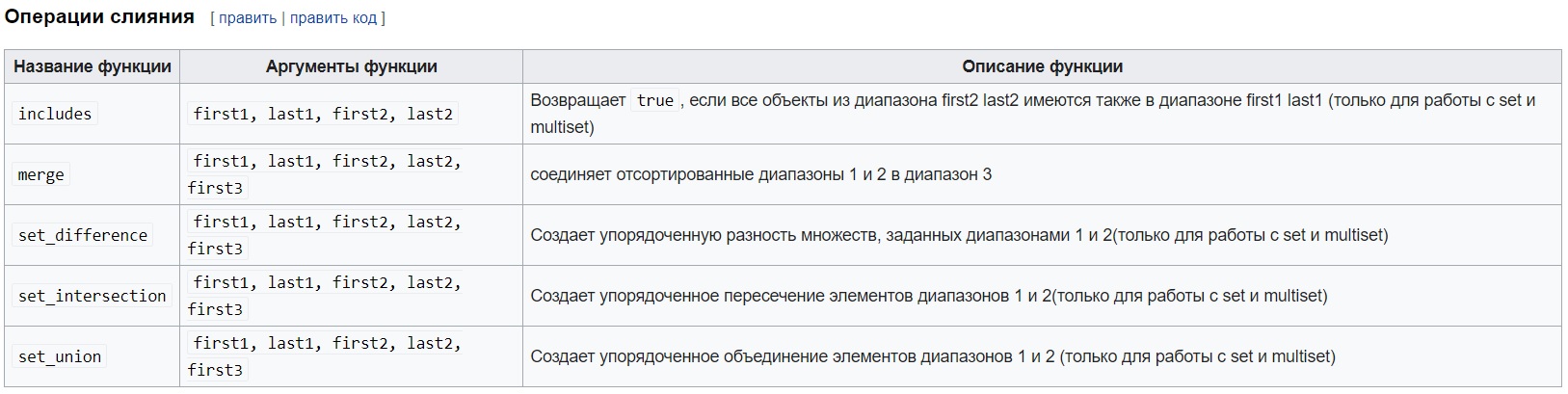
1. Алгоритмы и основные классы алгоритмов

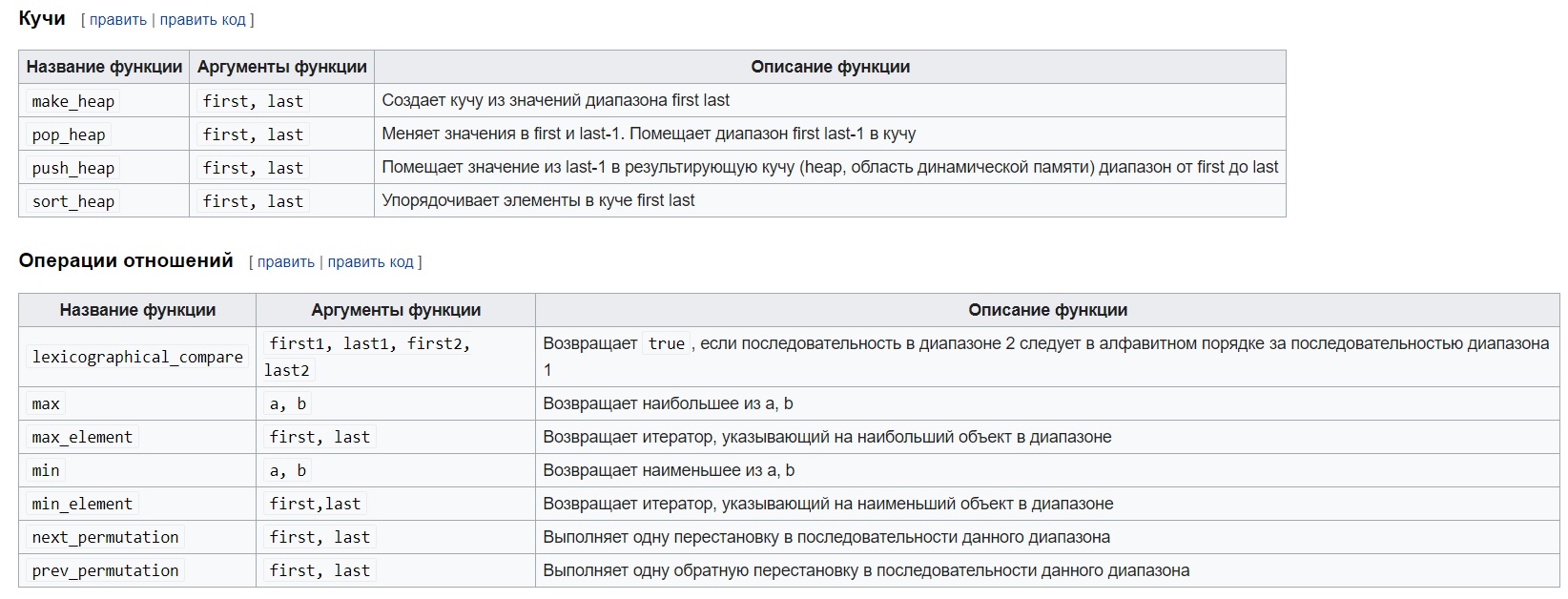












#### **17. Стандартная библиотека. Контейнеры. Последовательные контейнеры.**

1. **Стандартная библиотека** - коллекция классов и функций, написанных на базовом языке. Стандартная библиотека поддерживает несколько основных контейнеров, функций для работы с этими контейнерами, объектов-функции, основных типов строк и потоков, поддержку некоторых языковых особенностей, и часто используемые функции для выполнения таких задач, как, например, нахождение квадратного корня числа. Функциональные особенности стандартной библиотеки

объявляются внутри пространства имен std.

1. Контейнеры:

**<bitest>** - реализует специализированный класс контейнеров std::bitest - битовый массив.

**<deque>** -реализует шаблон класса контейнера std::deque — [двусвязная очередь](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D1%83%D1%81%D0%B2%D1%8F%D0%B7%D0%BD%D0%B0%D1%8F_%D0%BE%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D1%8C).

**<list>** - реализует шаблон класса контейнера std::list — [двусвязный список](https://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D0%BD%D0%B5%D0%B9%D0%BD%D1%8B%D0%B9_%D1%81%D0%BF%D0%B8%D1%81%D0%BE%D0%BA).

**<map>** - реализует шаблоны классов контейнеров std::map и std::multimap — а[ссоциативный массив](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D1%81%D0%BE%D1%86%D0%B8%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2) и [мультиотображение](https://ru.wikipedia.org/w/index.php?title=%D0%9C%D1%83%D0%BB%D1%8C%D1%82%D0%B8%D0%BE%D1%82%D0%BE%D0%B1%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5_(%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)&action=edit&redlink=1).

**<queue>** - реализует класс адаптер-контейнера std::queue — односторонняя [очередь](https://ru.wikipedia.org/wiki/%D0%9E%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D1%8C_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).

**<set>** - реализует шаблоны классов контейнеров std::set и std::multiset — сортированные ассоциативные контейнеры или [множества](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%BE_(%D1%82%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)).

**<stack>** - реализует класс адаптер-контейнера std::stack — [стек](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B5%D0%BA).

**<vector>** - реализует шаблон класса контейнеров [std::vector](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%BA%D1%82%D0%BE%D1%80_(STL)) — [динамический массив](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2).

**<algorithm>** - реализует определения многих [алгоритмов](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC) для работы с контейнерами.

**<functional>** -реализует несколько [объект-функций](https://ru.wikipedia.org/w/index.php?title=%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F&action=edit&redlink=1), разработанных для работы со стандартными алгоритмами.

**<iterator> -** реализует классы и шаблоны для работы с [итераторами](https://ru.wikipedia.org/wiki/%D0%98%D1%82%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80).

**<locale> -** реализует классы и шаблоны для работы с [локалями](https://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%BA%D0%B0%D0%BB%D1%8C).

**<memory> -** реализует инструменты [управления памятью](https://ru.wikipedia.org/wiki/%D0%A3%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C%D1%8E) в C++, включая шаблон класса [std::auto\_ptr](https://ru.wikipedia.org/w/index.php?title=Auto_ptr&action=edit&redlink=1).

**<stdexcept> -** содержит стандартную обработку ошибок классов, например, std::logic\_error и std::runtime\_error, причем оба происходят из std::exception.

**<utility> -** реализует шаблон класса std::pair для работы с парами (двучленными [кортежами](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D1%80%D1%82%D0%B5%D0%B6_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0))) объектов.

**<string>** -реализует стандартные [строковые](https://ru.wikipedia.org/w/index.php?title=%D0%A1%D1%82%D1%80%D0%BE%D0%BA%D0%B0_(C%2B%2B)&action=edit&redlink=1) классы и шаблоны.

**<regex>** -новое в [C++11](https://ru.wikipedia.org/wiki/C%2B%2B11). Предоставляет утилиты для сопоставления строк с шаблоном с помощью [регулярных выражений](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B3%D1%83%D0%BB%D1%8F%D1%80%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D1%8F).

**<fstream>** - реализует инструменты для файлового ввода и вывода. Смотрите [fstream](https://ru.wikipedia.org/wiki/Fstream).

**<ios>** - реализует несколько типов и функций, составляющих основу операций с iostreams.

**<iostream>** - реализует основы ввода и вывода языка C++. Смотрите [iostream](https://ru.wikipedia.org/wiki/Iostream).

**<iosfwd>** - реализует [предварительные объявления](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D0%B4%D0%B2%D0%B0%D1%80%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D1%8A%D1%8F%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5) нескольких шаблонов классов, связанных с вводом-выводом.

**<iomanip>** - реализует инструменты для работы с форматированием вывода, например [базу](https://ru.wikipedia.org/w/index.php?title=Radix&action=edit&redlink=1), используемую при форматировании целых и [точных](https://ru.wikipedia.org/w/index.php?title=%D0%90%D1%80%D0%B8%D1%84%D0%BC%D0%B5%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%82%D0%BE%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C&action=edit&redlink=1) значений [чисел с плавающей запятой](https://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D0%BB%D0%B0_%D1%81_%D0%BF%D0%BB%D0%B0%D0%B2%D0%B0%D1%8E%D1%89%D0%B5%D0%B9_%D0%B7%D0%B0%D0%BF%D1%8F%D1%82%D0%BE%D0%B9).

**<istream>** - реализует шаблон класса std::istream и других необходимых классов для ввода.

**<ostream>** - реализует шаблон класса std::ostream и других необходимых классов для вывода.

**<sstream>** - реализует шаблон класса std::sstream и других необходимых классов для работы со строками.

**<complex>** - реализует шаблон класса std::complex и связанные функции для работы с [комплексными числами](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BB%D0%B5%D0%BA%D1%81%D0%BD%D0%BE%D0%B5_%D1%87%D0%B8%D1%81%D0%BB%D0%BE).

**<numeric>** - реализует алгоритмы для [числовой обработки](https://ru.wikipedia.org/w/index.php?title=%D0%A7%D0%B8%D1%81%D0%BB%D0%BE%D0%B2%D0%B0%D1%8F_%D0%BE%D0%B1%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0&action=edit&redlink=1)

**<valarray>** - реализует шаблон класса std::valarray — класс [массивов](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D1%81%D0%B8%D0%B2_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), оптимизированный для числовой обработки.

**<exception>** - реализует несколько типов и функций, связанных с [обработкой исключений](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D0%B8%D1%81%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%BD%D0%B8%D0%B9), включая std::exception — базовый класс всех перехватов исключений в Стандартной Библиотеке.

**<limits>** - реализует шаблон класса std::numeric\_limits, используемый для описания свойств базовых числовых типов.

**<new>** - реализует операторы new и delete, а также другие функции и типы, составляющие основу [управления памятью](https://ru.wikipedia.org/wiki/%D0%A3%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C%D1%8E) в C++.

**<typeinfo>** - реализует инструменты для работы с [динамической идентификацией типа данных](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%B8%D0%B4%D0%B5%D0%BD%D1%82%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F_%D1%82%D0%B8%D0%BF%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) в C++.

1. Последовательные контейнеры поддерживают указанный пользователем порядок вставляемых элементов.

Контейнер **vector** ведёт себя как массив, но может автоматически увеличиваться по мере необходимости. Он поддерживает прямой доступ и связанное хранение и имеет очень гибкую длину.

Контейнер **array** имеет некоторые сильные vector стороны, но длина не столь гибкая

Контейнер **deque** (двусторонняя очередь) обеспечивает быструю вставку и удаление в начале и в конце контейнера. Он разделяет случайный доступ и гибкие преимущества длины vector, но не является смежным.

Контейнер **list** - это вдвойне связанный список, который обеспечивает двунаправленный доступ, быстрые вставки и быстрые удаления в любом месте контейнера, но вы не можете случайно получить доступ к элементу в контейнере.

Контейнер **forward\_list** - однонаправленный список. Это версия контейнера list только с доступом в прямом направлении.

#### 

#### **18. Стандартная библиотека. Контейнеры. Ассоциативные контейнеры.**

1. **Стандартная библиотека** - коллекция классов и функций, написанных на базовом языке. Стандартная библиотека поддерживает несколько основных контейнеров, функций для работы с этими контейнерами, объектов-функции, основных типов строк и потоков, поддержку некоторых языковых особенностей, и часто используемые функции для выполнения таких задач, как, например, нахождение квадратного корня числа. Функциональные особенности стандартной библиотеки объявляются внутри пространства имен std.
2. Контейнеры:

**<bitest>** - реализует специализированный класс контейнеров std::bitest - битовый массив.

**<deque>** -реализует шаблон класса контейнера std::deque — [двусвязная очередь](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D1%83%D1%81%D0%B2%D1%8F%D0%B7%D0%BD%D0%B0%D1%8F_%D0%BE%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D1%8C).

**<list>** - реализует шаблон класса контейнера std::list — [двусвязный список](https://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D0%BD%D0%B5%D0%B9%D0%BD%D1%8B%D0%B9_%D1%81%D0%BF%D0%B8%D1%81%D0%BE%D0%BA).

**<map>** - реализует шаблоны классов контейнеров std::map и std::multimap — а[ссоциативный массив](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D1%81%D0%BE%D1%86%D0%B8%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2) и [мультиотображение](https://ru.wikipedia.org/w/index.php?title=%D0%9C%D1%83%D0%BB%D1%8C%D1%82%D0%B8%D0%BE%D1%82%D0%BE%D0%B1%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5_(%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)&action=edit&redlink=1).

**<queue>** - реализует класс адаптер-контейнера std::queue — односторонняя [очередь](https://ru.wikipedia.org/wiki/%D0%9E%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D1%8C_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).

**<set>** - реализует шаблоны классов контейнеров std::set и std::multiset — сортированные ассоциативные контейнеры или [множества](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%BE_(%D1%82%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)).

**<stack>** - реализует класс адаптер-контейнера std::stack — [стек](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B5%D0%BA).

**<vector>** - реализует шаблон класса контейнеров [std::vector](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%BA%D1%82%D0%BE%D1%80_(STL)) — [динамический массив](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2).

**<algorithm>** - реализует определения многих [алгоритмов](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC) для работы с контейнерами.

**<functional>** -реализует несколько [объект-функций](https://ru.wikipedia.org/w/index.php?title=%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F&action=edit&redlink=1), разработанных для работы со стандартными алгоритмами.

**<iterator> -** реализует классы и шаблоны для работы с [итераторами](https://ru.wikipedia.org/wiki/%D0%98%D1%82%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80).

**<locale> -** реализует классы и шаблоны для работы с [локалями](https://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%BA%D0%B0%D0%BB%D1%8C).

**<memory> -** реализует инструменты [управления памятью](https://ru.wikipedia.org/wiki/%D0%A3%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C%D1%8E) в C++, включая шаблон класса [std::auto\_ptr](https://ru.wikipedia.org/w/index.php?title=Auto_ptr&action=edit&redlink=1).

**<stdexcept> -** содержит стандартную обработку ошибок классов, например, std::logic\_error и std::runtime\_error, причем оба происходят из std::exception.

**<utility> -** реализует шаблон класса std::pair для работы с парами (двучленными [кортежами](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D1%80%D1%82%D0%B5%D0%B6_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0))) объектов.

**<string>** -реализует стандартные [строковые](https://ru.wikipedia.org/w/index.php?title=%D0%A1%D1%82%D1%80%D0%BE%D0%BA%D0%B0_(C%2B%2B)&action=edit&redlink=1) классы и шаблоны.

**<regex>** -новое в [C++11](https://ru.wikipedia.org/wiki/C%2B%2B11). Предоставляет утилиты для сопоставления строк с шаблоном с помощью [регулярных выражений](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B3%D1%83%D0%BB%D1%8F%D1%80%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D1%8F).

**<fstream>** - реализует инструменты для файлового ввода и вывода. Смотрите [fstream](https://ru.wikipedia.org/wiki/Fstream).

**<ios>** - реализует несколько типов и функций, составляющих основу операций с iostreams.

**<iostream>** - реализует основы ввода и вывода языка C++. Смотрите [iostream](https://ru.wikipedia.org/wiki/Iostream).

**<iosfwd>** - реализует [предварительные объявления](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D0%B4%D0%B2%D0%B0%D1%80%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D1%8A%D1%8F%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5) нескольких шаблонов классов, связанных с вводом-выводом.

**<iomanip>** - реализует инструменты для работы с форматированием вывода, например [базу](https://ru.wikipedia.org/w/index.php?title=Radix&action=edit&redlink=1), используемую при форматировании целых и [точных](https://ru.wikipedia.org/w/index.php?title=%D0%90%D1%80%D0%B8%D1%84%D0%BC%D0%B5%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%82%D0%BE%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C&action=edit&redlink=1) значений [чисел с плавающей запятой](https://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D0%BB%D0%B0_%D1%81_%D0%BF%D0%BB%D0%B0%D0%B2%D0%B0%D1%8E%D1%89%D0%B5%D0%B9_%D0%B7%D0%B0%D0%BF%D1%8F%D1%82%D0%BE%D0%B9).

**<istream>** - реализует шаблон класса std::istream и других необходимых классов для ввода.

**<ostream>** - реализует шаблон класса std::ostream и других необходимых классов для вывода.

**<sstream>** - реализует шаблон класса std::sstream и других необходимых классов для работы со строками.

**<complex>** - реализует шаблон класса std::complex и связанные функции для работы с [комплексными числами](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BB%D0%B5%D0%BA%D1%81%D0%BD%D0%BE%D0%B5_%D1%87%D0%B8%D1%81%D0%BB%D0%BE).

**<numeric>** - реализует алгоритмы для [числовой обработки](https://ru.wikipedia.org/w/index.php?title=%D0%A7%D0%B8%D1%81%D0%BB%D0%BE%D0%B2%D0%B0%D1%8F_%D0%BE%D0%B1%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0&action=edit&redlink=1)

**<valarray>** - реализует шаблон класса std::valarray — класс [массивов](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D1%81%D0%B8%D0%B2_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), оптимизированный для числовой обработки.

**<exception>** - реализует несколько типов и функций, связанных с [обработкой исключений](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D0%B8%D1%81%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%BD%D0%B8%D0%B9), включая std::exception — базовый класс всех перехватов исключений в Стандартной Библиотеке.

**<limits>** - реализует шаблон класса std::numeric\_limits, используемый для описания свойств базовых числовых типов.

**<new>** - реализует операторы new и delete, а также другие функции и типы, составляющие основу [управления памятью](https://ru.wikipedia.org/wiki/%D0%A3%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C%D1%8E) в C++.

**<typeinfo>** - реализует инструменты для работы с [динамической идентификацией типа данных](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%B8%D0%B4%D0%B5%D0%BD%D1%82%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F_%D1%82%D0%B8%D0%BF%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) в C++.

1. В ассоциативных контейнерах элементы вставляются в предварительно определенном порядке - например, с сортировкой по возрастанию. Также доступны неупорядоченные ассоциативные контейнеры. Ассоциативные контейнеры можно объединить в два подмножества: сопоставления (set) и наборы (map)

Контейнер **map**, который иногда называют словарем, состоит из пар “ключ-значение”. Ключ используется для упорядочивания последовательности, а значение связано с ключом.

**set** - контейнер уникальных элементов, упорядоченных по возрастанию. Каждое его значение также является и ключом.

Контейнеры map и set разрешают вставку только одного экземпляра ключа или элемента. Если необходимо включить несколько экземпляров элемента, следует использовать контейнер **multimap** и **multiset**.

Упорядоченные контейнеры map и set поддерживают двунаправленные итераторы, а их неупорядоченные аналоги (**unordered\_map**, **unordered\_set**, **unordered\_multimap** и **unordered\_multiset**) - итераторы с перебором в прямом направлении.

#### **19. Стандартная библиотека. Адаптеры.**

1. **Стандартная библиотека** - коллекция классов и функций, написанных на базовом языке. Стандартная библиотека поддерживает несколько основных контейнеров, функций для работы с этими контейнерами, объектов-функции, основных типов строк и потоков, поддержку некоторых языковых особенностей, и часто используемые функции для выполнения таких задач, как, например, нахождение квадратного корня числа. Функциональные особенности стандартной библиотеки объявляются внутри пространства имен std.
2. **Контейнер-адаптер** - это разновидность последовательного или ассоциативного контейнера, который ограничивает интерфейс для простоты и ясности. Контейнерные адаптеры не поддерживают итераторы

Контейнер **queue** соответствует стратегии “первым вошёл - первым вышел”.

Контейнер **priority\_queue** упорядочен таким образом, что первым в очереди всегда оказывается элемент с наибольшим значением.

Контейнер **stack** соответствует стратегии “последним вошёл - первым вышел”.

#### **20. Умные указатели. auto\_ptr, unique\_ptr, shared\_ptr, weak\_ptr. Устройство, преимущества и недостатки**

**Умный указатель** — это объект, работать с которым можно как с обычным указателем, но при этом, в отличии от последнего, он предоставляет некоторый дополнительный функционал (например, автоматическое освобождение закрепленной за указателем области памяти).

Умные указатели призваны для борьбы с утечками памяти, которые сложно избежать в больших проектах. Они особенно удобны в местах, где возникают исключения, так как при последних происходит процесс раскрутки стека и уничтожаются локальные объекты. В случае обычного указателя — уничтожается переменная-указатель, при этом ресурс останется не освобожденным. В случае умного указателя — вызовется деструктор, который и освободит выделенный ресурс.

**auto\_ptr**

Этот шаблон класса устарел с C ++ 11.

auto\_ptr — это интеллектуальный указатель, который управляет объектом, полученным с помощью нового выражения, и удаляет этот объект, когда сам auto\_ptr уничтожается.

Объект, описанный с использованием класса auto\_ptr, сохраняет указатель на один выделенный объект, который гарантирует, что когда он выходит из области видимости, объект, на который он указывает, должен автоматически уничтожаться. Он основан на модели исключительного владения, то есть два указателя одного типа не могут одновременно указывать на один и тот же ресурс.

Конструктор копирования и оператор присваивания auto\_ptr фактически не копируют сохраненный указатель, а передают его, оставляя первый объект auto\_ptr пустым. Это был один из способов реализации строгого владения, так что только один объект auto\_ptr может владеть указателем в любой момент времени, т.е. auto\_ptr не должен использоваться там, где необходима семантика копирования.

*Почему auto\_ptr устарел?*

Он принимает владение указателем таким образом, что никакие два указателя не должны содержать один и тот же объект. Назначение передает владение и сбрасывает автоматический указатель rvalue в нулевой указатель. Таким образом, они не могут быть использованы в контейнерах STL из-за вышеупомянутой невозможности копирования.

**unique\_ptr**

*std::unique\_ptr* является доработанной версией устаревшего *std::auto\_ptr*, точнее идеи, которая стояла за оным. Из самого имени класса ясно, что он предназначен для эксклюзивного хранения указателя, т.е. в этом он повторяет поведение своего предтечи. Объект класса *unique\_ptr* является *перемещаемым*, но не *копируемым*, что в терминах С++ означает, что у него определен оператор перемещения(operator=(T&&)) и конструктор перемещения, а оператор копирования и конструктор копирования отсутствуют. После перемещения(равно как и при создании посредством конструктора по умолчанию) *unique\_ptr* содержит в себе *nullptr*.

Еще одним отличием является возможность хранения указателя на массив объектов. Для этого при создании объекта *unique\_ptr* необходимо использовать особый синтаксис: *std::unique\_ptr<T[]>.* При создании *unique\_ptr*, для хранения массива, теряется смысл в операторах “–>” и “.”, поэтому они исключены в соответствующей специализации шаблона. Зато добавлен оператор “[]”. Благодаря этой специализации мы получаем полноценный умный контейнер для хранения указателя на массив, с полным сохранением семантики массива.

*unique\_ptr* является отличной заменой *auto\_ptr* и должен быть использован везде, где указатель подразумевает эксклюзивное хранение. К примеру, если указатель является членом класса, то он является отличным кандидатом для хранения его в *unique\_ptr*.

(Также стоит знать, что у auto\_ptr и unique\_ptr есть метод get() который возвращает сырой указатель, метод reset(), который затирает данные, на которые указывает умный указатель, а также метод release(), который не трогает данные, на которые указывает умный указатель, однако делает его пустым)

Недостатки обоих указателей являются в том, что они не могут одновременно указывать на одни и те же данные.(Если это сделать, то в случае auto\_ptr это приведет к ошибке, а в случае unique\_ptr код даже не будет компилироваться)

**shared\_ptr**

**(**Указатель , который может разделять между собой владение с другим схожим с ним shared\_ptr, не очень классная штука, лучше unique, потому что он единоправный владелец, а несколько владельцев одного и того же кластера данных могут и программиста путать и вызывать траблы, если хотят меняться одновременно.)

В отличии от рассмотренных выше указателей, shared\_ptr реализует подсчет ссылок на ресурс. Ресурс освободится тогда, когда счетчик ссылок на него будет равен 0.

**weak\_ptr**

Шаблон класса описывает объект, указывающий на ресурс, который управляется одним или несколькими объектами shared\_ptr. Объект weak\_ptr указывает на ресурс, если он был создан из объекта shared\_ptr, который владеет этим ресурсом; если он был создан из объекта weak\_ptr, который указывает на этот ресурс; или если этот ресурс был назначен ему с помощью operator=. Объект weak\_ptr не предоставляет прямой доступ к ресурсу, на который он указывает.

Устройство: в конструкторе класса SmartPointer мы получаем указатель на какой-то объект, память под который мы выделили, и адрес в памяти, на который указывает тот указатель, мы будем записывать в наш указатель в private-секции. В деструкторе очищаем память, на которую указывает указатель.

Приложу на всякий случай свой класс SmartPointer:

template <typename T>

class SmartPointer{

public:

SmartPointer(T \*ptr) {

this->ptr = ptr;

cout << "\nConstructor" << endl; }

~SmartPointer() {

delete ptr;

cout << "Destructor" << endl; }

T &operator\*() {

return \*ptr; }

private:

T \*ptr;};

Недостатки:

1. Умные указатели не помогают избежать циклов в графических структурах.
2. Большинство библиотек предлагают разные реализации. К сожалению, в большинстве случаев эти разные варианты несовместимы, что становится проблемой при микшировании библиотек.
3. Нет определённого объекта, который уничтожает объект. Часто желательно знать, когда и как разрушается объект.
4. Возникает проблема с подсчётом ссылок в некоторых типах структур данных, которые имеют циклы.
5. Могут возникать проблемы с доступом к умным указателям из нескольких потоков, одновременный доступ к подсчётам ссылок может вызвать проблемы.

Преимущества:

1. Помогает избегать утечки памяти

#### **21. Rvalue ссылки, Move семантика. Lambda​-выражения**

**Rvalue ссылка** - это составной тип, очень похожий на традиционную ссылку в С++. Чтобы различать эти два типа, мы будем называть традиционную С++ ссылку Ivalue ссылка. Когда будет встречаться термин ссылка, то это относится к обоим видам ссылок.

Ivalue ссылка формируется путём помещая & после некоторого типа.

T a;

T &a\_ref1 = a; *// это lvalue ссылка*

Если после некоторого типа поместить &&, то получится rvalue ссылка

T a;

T&& a\_ref2 = a; *// это rvalue ссылка*

Rvalue ссылка ведёт себя точно так же, как и Ivalue ссылка, за исключением того, что она может быть связана с временными объектом, тогда как Ivalue связать с временным объектом нельзя.

~~Нахуя~~ Зачем нам это может понадобиться: Комбинация ссылок это то, что необходимо для легкой реализации move semantics. Rvalue ссылка может также использоваться для достижения perfect forwarding, что ранее было нерешённой проблемой в С++. Для большинства программистов rvalue ссылки позволяют создать более производительные библиотеки.

**Семантика перемещения(move semantics)** - собирательное название специализированных средств языка программирования С++, которые предназначены для осуществления перемещения данных во время инициализации и конструирования новых объектов, что позволяет сократить издержки на копирование. Главная задача rvalue ссылок здесь состоит в том, чтобы позволить нам реализовать перемещение без переписывания кода и издержек времени выполнения.

(вероятно убрали из экзамена)

(Влада так сказала)

(но это не точно, как можно верить человеку с именем Влада)

(убрали, Лёня так сказал, он конечно тот ещё душнила, но я ему верю) (а Мне не веРиШь???)

(хочу пиво)

Мне можна верить...ну если не мне, то Хвосту:

**Да верю верю**



**Лямбда-выражение** - специальный синтаксис для определения функциональных объектов (конструкция, позволяющая использовать объект как функцию). Применяется как правило для объявления анонимных функций (функции, которые объявляются в месте использования и не получают уникального идентификатора для доступа к ним) по месту их использования. Используя лямбда-выражения, можно объявлять функции в любом месте когда