1. Процедурная парадигма программирования. Процедуры и функции.

Процедурная парадигма: Реализацию алг-ов вычислений необходимо создавать с помощью мелких, независимых друг от друга процедур, кот. вызывают друг друга в соот. с логикой проги.
Процедурное программирование — парадигма, основ. на испол. процедур. Процедура (иногда также называемая подпрограммой или методом) — это посл-сть команд, кот. следует выполнить. ПП широко исп. в крупномасштабных проектах, когда на первый план выходят след. преимущества:
·легкость повторного исп. фрагментов кода, оформленных как процедуры ;
·легкость прослеживания логики программы;
·возможность сопровождения прог. продукта через длительное время после написания кода или кем-то, кроме его автора.
·легкость в отладке: нужно убедиться в "работоспособности" каждой из процедур, что намного проще, чем отладка всего алгоритма целиком
ПП является частным случаем императивной парадигмы. Императивная парадигма программирования описывает процесс вычислений посредством описания управляющей логики программы, т.е. в виде пос-ти отдельных команд, которые должен выполнить компьютер. Каждая команда является инструкцией, изменяющей состояние программы. Программа, написанная в императивном стиле, похожа на набор приказов, выражаемых повелительным наклонением в естественных языках.
В современном программировании процедура может иметь несколько точек выхода (return в C-подобных языках), несколько точек входа (с помощью yield в Python или статических локальных переменных в C++), иметь аргументы, возвращать значение как результат своего выполнения (тогда это функция).

2. Глобальные и локальные переменные. Время жизни переменных.

Каждая переменная имеет свою область видимости, то есть такую область, в которой можно работать с переменной. За пределами этой области, о данной переменной ничего известно не будет, а значит и использовать её нельзя. Итак, переменная находится в области видимости, если к ней можно получить доступ.
Существуют локальные и глобальные переменные. Переменные, объявленные внутри метода, называются локальными. Локальные переменные имеют свои области видимости, этими областями являются методы, в которых объявлены переменные. Таким образом, в разных методах можно использовать переменные с одинаковыми именами, что в свою очередь очень удобно.

Глобальные переменные объявляются вне тела какого-либо метода, и поэтому область видимости таких переменных распространяется на всю программу. Переменная, объявленная вне всех методов, помещается в глобальную область видимости. Доступ к таким переменным может осуществляться из любого места программы. Такие переменные располагаются в глобальном пуле памяти, поэтому время их жизни совпадает со временем жизни программы.
Переменная, объявленная внутри блока, принадлежит локальной области видимости. Такая переменная не видна (поэтому и недоступна) за пределами блока, в котором она объявлена. Самый распространенный случай локального объявления – переменная, объявленная внутри функции.
Переменная с глобальным временем жизни характеризуется тем, что в течение всего времени выполнения программы с ней ассоциирована ячейка памяти и значение. Переменной с локальным временем жизни выделяется новая ячейка памяти при каждом входе в блок, в котором она определена или объявлена.

3. Формальные и фактические параметры. Передача параметров по значению , по ссылке и по указателю.

формальный пара. — аргумент, указ. при объявлении или опр-и функ.
фактический парам. — аргумент, перед. в функцию при её вызове
При вызове по значению, выражение-аргумент вычисляется, и полученное значение связывается с соответствующим формальным параметром функции (обычно посредством копирования этого значения в новую область памяти). При этом, если язык разрешает функциям присваивать значения своим параметрам, то изменения будут касаться лишь этих локальных копий
При вызове-по-ссылке или передаче-по-ссылке, функция неявно получает ссылку на переменную, использованную в качестве аргумента, вместо копии её значения. Обычно это означает, что функция может осуществлять модификацию (то есть изменять состояние) переменной, переданной в качестве параметра, и это будет иметь эффект в вызывающем контексте. Системы типов некоторых языков, использующих вызов по значению и непосредственно не поддерживающих вызов по ссылке, предоставляют возможность явно определять ссылки (объекты, ссылающиеся на другие объекты), в частности, указатели (объекты, представляющие собой адреса других объектов в памяти ЭВМ). Их использование позволяет симулировать вызов по ссылке внутри семантики вызова по значению. Такое решение применяется, например, в языке Си. Оно не является самостоятельной стратегией вычисления — язык по-прежнему вызывает по значению — но иногда его называют «вызовом-по-адресу» (call-by-address) или «передачей-по-адресу» (pass-by-address). В небезопасных языках, например в

Си или C++, оно может приводить к ошибкам доступа к памяти, таким как разыменование нулевого указателя.

4. Типы данных. Статическая и динамическая типизация.

Тип данных — множество значений и операций на этих значениях. Тип определяет возможные значения и их смысл, операции, а также способы хранения значений типа. Операция назначения типа информационным сущностям называется типизацией. Назначение и проверка согласования типов может осуществляться заранее (статическая типизация), непосредственно при использовании (динамическая типизация) или совмещать оба метода. Типы могут назначаться «раз и навсегда» (сильная типизация) или позволять себя изменять (слабая типизация).

- Статически типизированные языки проверяют типы и ищут ошибки типизации на стадии компиляции.
- Динамически типизированные языки проверяют типы и ищут ошибки типизации на стадии исполнения.

Динамическая типизация — приём, широко используемый в языках программирования и языках спецификации, при котором переменная связывается с типом в момент присваивания значения, а не в момент объявления переменной. Таким образом, в различных участках программы одна и та же переменная может принимать значения разных типов.
Недостатки:

- Статическая типизация позволяет уже при компиляции заметить простые ошибки «по недосмотру». Для динамической типизации требуется, как минимум выполнить данный участок кода.
- Развитая статическая система типов (такая как Хиндли-Милнер) играет значительную роль в самодокументировании программы; динамическая типизация по определению не проявляет этого свойства, что затрудняет разработку структурно сложных программ.
- Снижение производительности из-за трат процессорного времени на динамическую проверку типа, и излишние расходы памяти на переменные, которые могут хранить «что угодно». К тому же большинство языков с динамической типизацией интерпретируемые, а не компилируемые.

Статическая типизация — приём, широко используемый в языках программирования, при котором переменная, параметр подпрограммы, возвращаемое значение функции связывается с типом в момент объявления и тип не может быть изменён позже (переменная или параметр будут принимать, а функция — возвращать значения только этого типа).
Недостатки:

· Тяжело работать с данными из внешних источников (например, в реляционных СУБД / десериализация данных).

5. Динамическая память. Работа с динамической памятью.

Динамическое распределение памяти — способ выделения оперативной памяти компьютера для объектов в программе, при котором выделение памяти под объект осуществляется во время выполнения программы.
При динамическом распределении памяти объекты размещаются в т.н. «куче» (англ. heap): при конструировании объекта указывается размер запрашиваемой под объект памяти, и, в случае успеха, выделенная область памяти, условно говоря, «изымается» из «кучи», становясь недоступной при последующих операциях выделения памяти. Противоположная по смыслу операция — освобождение занятой ранее под какой-либо объект памяти: освобождаемая память, также условно говоря, возвращается в «кучу» и становится доступной при дальнейших операциях выделения памяти.
По мере создания в программе новых объектов, количество доступной памяти уменьшается. Отсюда вытекает необходимость постоянно освобождать ранее выделенную память. В идеальной ситуации программа должна полностью освободить всю память, которая потребовалась для работы. По аналогии с этим, каждая подпрограмма (процедура или функция) должна обеспечить освобождение всей памяти, выделенной в ходе её выполнения. Неправильное управление памятью приводит к т.н. «утечкам» памяти, когда выделенная память не освобождается. Многократные утечки памяти могут привести к исчерпанию всей оперативной памяти и нарушить работу операционной системы.
Другая проблема — это проблема фрагментации памяти. Выделение памяти происходит блоками — непрерывными фрагментами оперативной памяти (таким образом, каждый блок — это несколько идущих подряд байт). В какой-то момент, в куче попросту может не оказаться блока подходящего размера и, даже если свободная память достаточна для размещения объекта, операция выделения памяти окончится неудачей.
Для управления динамическим распределением памяти в некоторых языках используется «сборщик мусора» — программный объект, который следит за выделением памяти и обеспечивает её своевременное освобождение. Сборщик мусора также следит за тем, чтобы свободные блоки имели максимальный размер, и, при необходимости, осуществляет дефрагментацию памяти.

6. Обработка исключений

Обработка исключительных ситуаций— механизм языков программирования, предназначенный для описания реакции программы на ошибки времени выполнения и другие возможные проблемы (исключения), которые могут возникнуть при выполнении программы и приводят к невозможности (бессмысленности) дальнейшей отработки программой её базового алгоритма.
Во время выполнения программы могут возникать ситуации, когда состояние внешних данных, устройств ввода-вывода или компьютерной системы в целом делает дальнейшие вычисления в соответствии с базовым алгоритмом невозможными или бессмысленными (например, исчерпание доступной памяти).
Исключительные ситуации, возникающие при работе программы, можно разделить на два основных типа: синхронные и асинхронные, принципы реакции на которые существенно различаются.

- Синхронные исключения могут возникнуть только в определённых, заранее известных точках программы. Так, ошибка чтения файла или коммуникационного канала, нехватка памяти — типичные синхронные исключения, так как возникают они только в операции чтения или в операции выделения памяти соответственно.
- Асинхронные исключения могут возникать в любой момент времени и не зависят от того, какую конкретно инструкцию программы выполняет система. Типичные примеры таких исключений: аварийный отказ питания или поступление новых данных.

Обработка исключительных ситуаций самой программой заключается в том, что при возникновении исключительной ситуации управление передается некоторому заранее определенному обработчику — блоку кода, процедуре, функции, которые выполняют необходимые действия.
Существует два принципиально разных механизма функционирования обработчиков исключений.

- Обработка с возвратом подразумевает, что обработчик исключения ликвидирует возникшую проблему и приводит программу в состояние, когда она может работать дальше по основному алгоритму. В этом случае после того, как выполнится код обработчика, управление передаётся обратно в ту точку программы, где возникла исключительная ситуация.
- Обработка без возврата заключается в том, что после выполнения кода обработчика исключения управление передаётся в некоторое, заранее заданное место программы, и с него продолжается исполнение. То есть, фактически, при возникновении исключения команда, во время работы которой оно возникло, заменяется на безусловный переход к заданному оператору.

7. Модульность. Области видимости переменных.

Модульное программирование — это организация программы как совокупности небольших независимых блоков, называемых модулями, структура и поведение которых подчиняются определенным правилам. Использование модульного программирования позволяет упростить тестирование программы и обнаружение ошибок. Аппаратно-зависимые подзадачи могут быть строго отделены от других подзадач, что улучшает мобильность создаваемых программ.

Модуль — функционально законченный фрагмент программы. Во многих языках оформляется в виде отдельного файла с исходным кодом или поименованной непрерывной её части. Некоторые языки предусматривают объединение модулей в пакеты.

Область видимости в программировании — часть программы, в пределах которой идентификатор, объявленный как имя некоторой программной сущности (обычно — переменной), остаётся связанным с этой сущностью, то есть позволяет посредством себя обратиться к ней. За пределами области видимости тот же самый идентификатор может быть связан с другой переменной или функцией, либо быть свободным (не связанным ни с какой из них). Область видимости может, но не обязана совпадать с областью существования объекта, с которым связано имя.

В одномодульной программе без вложенных функций и без использования ООП может существовать только два типа области видимости: глобальная и локальная. Прочие типы существуют только при наличии в языке определённых синтаксических механизмов.

- Глобальная область видимости — идентификатор доступен во всем тексте программы.
- Локальная область видимости — идентификатор доступен только внутри определённой функции (процедуры).
- Видимость в пределах модуля может существовать в модульных программах, состоящих из нескольких отдельных фрагментов кода, обычно находящихся в разных файлах. Идентификатор, чьей областью видимости является модуль, доступен из любого кода в пределах данного модуля.
- Пакет или пространство имён. В глобальной области видимости искусственно выделяется поименованная подобласть. Имя «привязывается» к этой части программы и существует только внутри неё. Вне данной области имя либо вообще недоступно, либо доступно ограниченно.

В ООП-языках дополнительно к вышеперечисленным могут поддерживаться специальные ограничения области видимости, действующие только для членов классов, объявленных внутри класса или относящихся к нему:

· Приватная
область видимости означает, что имя доступно только внутри методов своего класса.
· Защищённая область видимости означает, что имя доступно только внутри своего класса и его классов-потомков.
· Общая область видимости означает, что имя доступно в пределах области видимости, к которой относится его класс.

8. Классы, интерфейсы , абстрактные классы.

Объектно-ориентированное программирование (ООП) — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.
Класс — это элемент ПО, описывающий абстрактный тип данных и его частичную или полную реализацию.
Интерфейс— программная/синтаксическая структура, определяющая отношение между объектами, которые разделяют определённое поведенческое множество и не связаны никак иначе. При проектировании классов, разработка интерфейса тождественна разработке спецификации (множества методов, которые каждый класс, использующий интерфейс, должен реализовывать).
Абстрактный класс в объектно-ориентированном программировании — базовый класс, который не предполагает создания экземпляров. Абстрактные классы реализуют на практике один из принципов ООП — полиморфизм. Абстрактный класс может содержать (и не содержать) абстрактные методы и свойства. Абстрактный метод не реализуется для класса, в котором описан, однако должен быть реализован для его не абстрактных потомков. Абстрактные классы представляют собой наиболее общие абстракции, то есть имеющие наибольший объём и наименьшее содержание.

9. Конструкторы и деструкторы.

Конструктор — функция, предназначенная для инициализации объектов класса. ООП дает возможность программисту описать функцию, явно предназначенную для инициализации объектов. Поскольку такая функция конструирует значения данного типа, она называется *конструктором*. Конструктор всегда имеет то же имя, что и сам класс. Когда класс имеет конструктор, все объекты этого класса будут инициализироваться.
Конструктор по умолчанию

Конструктор, не требующий параметров, называется *конструктором по умолчанию*. Это может быть конструктор с пустым списком параметров или конструктор, в котором все аргументы имеют значения по умолчанию. Конструкторы могут быть перегруженными, но конструктор по умолчанию может быть только один.

Конструктор копирования обязателен, если в программе используются функции-элементы и переопределенные операции, которые получают формальные параметры и возвращают в качестве результата такой объект не по ссылке, а по значению

При наличии в объекте указателей на динамические переменные и массивы или идентификаторов связанных ресурсов, такое копирование требует дублирования этих переменных или ресурсов в объекте-приемнике. С этой целью вводится конструктор копии, который автоматически вызывается во всех перечисленных случаях. Он имеет единственный параметр-ссылку на объект-источник

Деструктор обеспечивает соответствующую очистку объектов указанного типа. Имя деструктора представляет собой имя класса с предшествующим ему знаком «тильда» ~. Так, для класса X есть ~X(). Многие классы используют динамическую память, которая выделяется конструктором, а освобождается деструктором.

В Java нет деструкторов, но есть метод finalize (его никто не использует, т.к. эта хуйня не работает). Если необходим явный вызов деструктора, как в c++ можно написать соответствующие методы, например, freeResources() и вызывать их только по надобности.

10. Виртуальные методы.

В объектно-ориентированном программировании виртуальная функция или виртуальный метод - это функция или метод, поведение которых можно переопределить внутри наследующего класса посредством функция с той же сигнатурой. Это понятие является очень важной частью части полиморфизма ООП.

В ООП, когда производный класс наследуется от базового класса, объект производного класса может ссылаться через указатель или ссылку типа базового класса вместо типа производного класса.

Если рассматриваемая функция является «виртуальной» в базовом классе, реализация этой функции самого производного класса вызывается в соответствии с фактическим типом упомянутого объекта, независимо от объявленного типа указателя или ссылки.

Виртуальные функции позволяют программе вызывать методы, которые не обязательно существуют даже в момент компиляции кода.

В C ++ виртуальные методы объявляются путем добавления ключевого слова *virtual* к объявлению функции в базовом классе. Этот модификатор наследуется всеми реализациями этого метода в производных классах, что означает, что они могут продолжать переходить друг в друга и быть опозданием. И даже если методы, принадлежащие базовому классу, вызовут виртуальный метод, вместо этого они будут вызывать производный метод. Перегрузка происходит, когда два или более метода в одном классе имеют одно и то же имя метода, но разные параметры. Переопределяющее средство имеет два метода с тем же именем и параметрами метода. Перегрузка также называется отображением динамических функций и переопределением в качестве соответствия функций.

Все методы в Java по умолчанию являются виртуальными. Это означает, что любой метод может быть переопределен при использовании в наследовании, если этот метод не объявлен как final или static.

11. Статические и нестатические члены классов.

в C++: Члены класса могут использоваться с ключевым словом static. Когда член класса объявляется как статический, то тем самым компилятору дается указание, что должна существовать только одна копия этого члена, сколько бы объектов этого класса ни создавалось. Статический член используется совместно всеми объектами данного класса. Все статические данные инициализируются нулями при создании первого объекта, и другая инициализация не предусмотрена.
При объявлении статического члена данных класса этот член не определяется. Вместо этого необходимо обеспечить для них глобальное определение вне класса. Это делается путем нового объявления статической переменной, причем используется оператор области видимости для того, чтобы идентифицировать тот класс, к которому принадлежит переменная. Это необходимо для того, чтобы под статическую переменную была выделена память.

в Java: Статические переменные - переменные, созданные с ключевым словом 'static'. К статическим переменным класса обращаются только статические методы. Есть только **одна** копия статической области на класс - независимо от того, сколько экземпляров класса создано.
Есть два пути ссылки на статическую переменную:

- Через ссылку на любой экземпляр класса

- Через имя класса

Java обеспечивает статические методы, которые определяются в описании класса и к ним можно обращаться, не создавая экземпляр класса.
Статическую переменную или метод также называют переменной класса или методом, так как она принадлежит непосредственно классу, а не экземпляру этого класса

- В теле статического метода можно использовать только статические методы и переменные
- Статический метод не имеет 'this'

Блок инициализации - блок кода между фигурными скобками, который выполнен прежде, чем будет создан объект класса. Статический блок инициализации - определенный блок, использующий ключевое слово, static, он выполняется один раз, когда загружается класс и может только инициализировать статических членов данного класса.

12. Инкапсуляция.

Инкапсуляция - это свойство объекта/класса регулировать доступ к определенным своим компонентам извне самого объекта/класса.
Внутри объекта данные могут быть закрытыми (private). Закрытые данные доступны только для других частей этого объекта. Таким образом, закрытые данные недоступны для тех частей программы, которые существуют вне объекта. Если данные являются открытыми, то, несмотря на то, что они заданы внутри объекта, они доступны и для других частей программы. Характерной является ситуация, когда открытая часть объекта используется для того, чтобы обеспечить контролируемый интерфейс закрытых элементов объекта.
На самом деле объект является переменной определенного пользователем типа. Может показаться странным, что объект, который объединяет данные, можно рассматривать как переменную. Однако применительно к ООП это именно так. Каждый элемент данных такого типа является составной переменной.

13. Наследование. Виды наследования.

Наследование - это процесс, посредством которого один объект может приобретать свойства другого. Точнее, объект может наследовать основные

свойства другого объекта и добавлять к ним черты, характерные только для него. Наследование является важным, поскольку оно позволяет поддерживать концепцию иерархии классов. Применение иерархии классов делает управляемыми большие потоки информации. Например, подумайте об описании жилого дома. Дом - это часть общего класса, называемого строением. С другой стороны, строение - это часть более общего класса - конструкции, который является частью ещё более общего класса объектов, который можно назвать созданием рук человека. В каждом случае порожденный класс наследует все, связанные с родителем, качества и добавляет к ним свои собственные определяющие характеристики. Без использования иерархии классов, для каждого объекта пришлось бы задать все характеристики, которые бы исчерпывающе его определяли. Однако при использовании наследования можно описать объект путём определения того общего класса (или классов), к которому он относится, с теми специальными чертами, которые делают объект уникальным. Наследование играет очень важную роль в ООР.

и тут тоже примерчик(код)

14. Управление динамической памятью . Очистка мусора.

C++

Если размер объекта или массива заранее неизвестен, или размер объекта слишком большой, чтобы создавать его внутри функции, значит необходимо воспользоваться механизмом динамической памяти C++, использующую отдельную область памяти называемой кучей. Для этого вам необходимо знать 2 оператора:**new** - выделение памяти, если выделение памяти не произошло возвращается нулевой указатель; **delete** - освобождение памяти, не во всех компиляторах после освобождения памяти указателю присваивается 0.
В C++ нет автоматической сборки мусора. Другими словами, если указатель на выделенную память потерян, то она становится недоступной. Это называется утечкой памяти. Другой крайностью является попытка освободить одну и ту же память более одного раза, что приводит к системной ошибке. Частично или полностью эти проблемы решаются созданием классов, реализующих "умные указатели".

Класс auto_ptr из библиотеки STL имеет следующие ограничения:

- объектом может владеть только один указатель,
- объектом не может быть массив,
- нельзя использовать адресную арифметику.

Единственное предназначение этого класса автоматизировать уничтожение выделенной ранее памяти. Класс shared_ptr из Boost обладает расширенными возможностями:

- объект может иметь несколько владельцев;
- можно указать дополнительный класс, отвечающий за уничтожение объекта.

Оператор new можно перегрузить для новых классов, а оператор delete для классов автоматически вызывает деструктор. Как альтернативой можно воспользоваться С функциями (объявлены в stdlib.h) как malloc и free. Эти функции нельзя смешивать с операторами динамической памяти, то есть нельзя выделить память оператором new и освободить ее функцией free или наоборот.

Java

Управление памятью - это процесс размещения новых объектов и удаление неиспользуемых объектов, чтобы освободить место для этих новых ассигнований объектов. Традиционным для языков программирование способом управления памятью является ручной. В Java все объекты находятся в области памяти под названием куча. Куча создается, когда JVM запускается и может увеличиваться или уменьшаться в размерах во время выполнения приложения. Когда куча становится полной, происходит механизм сборки мусора. Все объекты, которые никогда больше не будут использоваться, очищаются. тем самым освобождая место для новых объектов. Также нужно обратить внимание, что JVM использует больше памяти, чем занимает куча..Размер кучи зависит от используемой платформы, но, как правило, это где-то между 2 и 128 Кб. Механизм сборки мусора - это процесс освобождения места в куче, для возможности добавления новых объектов. Объекты создаются посредством оператора new, тем самым присваивая объекту ссылку. Закончив работу с объектом, вы просто перестаете на него ссылаться — достаточно присвоить переменной ссылку на другой объект или значение null либо прекратить выполнение метода, чтобы его локальные переменные завершили свое существование естественным образом.

Виртуальная машина Java, применяя механизм сборки мусора, гарантирует, что любой объект, обладающий ссылками, остается в памяти — все объекты, которые недостижимы из выполняемого кода, удаляются с высвобождением отведенной для них памяти.

Память освобождается сборщиком мусора по его собственному "усмотрению" и обычно только в тех случаях, когда для дальнейшей работы программы необходим фрагмент свободной памяти большего размера, нежели тот, который имеется в распоряжении виртуальной машины в данный момент, либо если сборщик мусора "предвидит" потенциальную нехватку памяти в ближайшем будущем. Программа может завершить работу, не исчерпав ресурсов свободной памяти или даже не приблизившись к этой черте, и поэтому ей так и не потребуются "услуги" сборщика мусора. Объект считается "более недостижимым", если ни одна из переменных в коде, выполняемом в данный момент, не содержит ссылок на него либо цепочка ссылок, которая могла бы связать объект с некоторой переменной программы, обрывается. Мусор собирается системой без вашего вмешательства, но это не значит, что процесс не требует внимания вовсе.

15. Объектно-ориентированное программирование. Языки C ++ и Java.

Глобальные переменные В Java единственным глобальным пространством имен является классовая иерархия. В этом языке просто невозможно создать глобальную переменную, не принадлежащую ни одному из классов. В C++

можно создавать глобальные переменные (не обязательно в классе).

GoTo: в Java нет, в C++ есть

Указатели: Указатели или адреса в памяти - наиболее мощная и наиболее опасная черта C++. Хотя в Java дескрипторы объектов и реализованы в виде указателей, в ней отсутствуют возможности работать непосредственно с указателями. Нельзя преобразовать целое число в указатель, а также обратиться к произвольному адресу памяти.

Распределение памяти: Также в C++ опасной чертой является - распределение памяти. Распределение памяти в C, а значит и в C++, опирается на вызовы библиотечных функций malloc() и free().Если вызвать free() с указателем на блок памяти, который вы уже освободили ранее, или с указателем, память для которого никогда не выделялась - будет плохо. Обратная проблема, когда вы просто забываете вызвать free(), чтобы освободить ненужный больше блок памяти, ещё хуже. "Утечка памяти" приводит к постепенному замедлению работы программы по мере того, как системе виртуальной памяти приходится сбрасывать на диск неиспользуемые страницы с мусором. И, наконец, когда все системные ресурсы исчерпаны, программа неожиданно аварийно завершается. В C++ добавлены два оператора - new и delete, которые используются во многом аналогично функциям malloc() и free(). Необходимо отвечать за то, чтобы каждый неиспользуемый объект, созданный с помощью оператора new, был уничтожен оператором delete.

В Java нет функций malloc() , free(). Поскольку в ней каждая сложная структура данных - это объект, память под такие структуры резервируется в куче (heap) с помощью оператора new. Реальные адреса памяти, выделенные этому объекту, могут изменяться во время работы программы, но вам не нужно об этом беспокоиться. Java - система с так называемым сборщиком мусора. Сборщик мусора запускается каждый раз, когда система простаивает, либо когда Java не может удовлетворить запрос на выделение памяти.

Приведение типов в C и C++ - мощный механизм, который позволяет произвольным образом изменять тип указателей. Поскольку объекты в C++ - это просто указатели на адреса памяти, в этом языке во время исполнения программы нет способа обнаруживать случаи приведения к несовместимым типам. Дескрипторы объектов в Java включают в себя полную информацию о классе, представителем которого является объект, так что Java может выполнять проверку совместимости типов на фазе исполнения кода, возбуждая исключение в случае ошибки.

Препроцессорная обработка: Работа препроцессора C++ которого заключается в поиске специальных команд, начинающихся с символа #. Эти команды позволяют выполнять простую условную трансляцию и расширение макроопределений. Java управляется со своими задачами без помощи

препроцессора, вместо принятого в С стиля определения констант с помощью директивы #define в ней используется ключевое слово final.

16. Массивы и списки.

Массивы

Массив - это совокупность однотипных данных, расположенных непрерывно в памяти. Доступ к элементу осуществляется по индексу за O(1) - мы обращаемся непосредственно к нужному участку памяти. Количество индексов, необходимое для получения доступа к одному элементу называется размерностью массива. В зависимости от размерности массивы могут одномерными [] и двумерными [][] и т.д. В одномерных массивах, иначе их еще называют векторами, для доступа к элементам нужен только один индекс, потому что такой массив представляет собой один ряд следующих друг за другом элементов. В двумерных - 2 индекса и т.д. Привести пример(код)

списки

Связный список является простейшим типом данных динамической структуры, состоящей из элементов. Каждый узел включает в себя в классическом варианте два поля: *данные* и *указатель на следующий узел в списке*. Элементы связанного списка можно помещать и исключать произвольным образом.

Связный список, содержащий только один указатель на следующий элемент, называется ***односвязным***.

Связный список, содержащий два поля указателя – на следующий элемент и на предыдущий, называется ***двусвязным***. **По способу связи элементов** различают линейные и циклические списки. Связный список, в котором, последний элемент указывает на NULL, называется ***линейным***. Связный список, в котором последний элемент связан с первым, называется ***циклическим***.

массив	**список**
Выделение памяти осущ. единовременно под весь массив до начала его исп.	Выделение памяти осущ. по мере ввода новых элементов
При уд./доб. эл-та требуется копирование	Уд./доб. эл-та осущ. переустановкой

всех последующих эле-ов для осущ. их сдвига	указателей, при этом сами данные не копир.
Для хранения элемента требуется V памяти, необходимый только для хранения данных этого элемента	Для хранения эл-та требуется V памяти, достаточный для хранения данных эл-та и указ-й (1/2) на другие эл-ты списка
Доступ к элементам может осущ. в произвольном порядке	Возможен только посл. доступ к элементам

17. Очереди и стеки.

Очередь — структура данных, добавление и удаление элементов в которой происходит путём операций push и pop соответственно. Притом первым из очереди удаляется элемент, который был помещен туда первым, то есть в очереди реализуется принцип «первым вошел — первым вышел» (*FIFO*). У очереди имеется **голова** и **хвост**. Когда элемент ставится в очередь, он занимает место в её хвосте. Из очереди всегда выводится элемент, который находится в ее голове. Очередь поддерживает следующие операции:

- empty — проверка очереди на наличие в ней элементов,
- push — операция вставки нового элемента,
- pop — операция удаления нового элемента,
- size — операция получения количества элементов в очереди.

Из-за того что нам не нужно снова выделять память, каждая операция выполняется за O(1)O(1) времени.

Плюсы: 1) проста в разработке 2) по сравнению с реализацией на списке есть незначительная экономия памяти.

Минусы: 1) количество элементов в очереди ограничено размером массива (исправляется написанием функции расширения массива) 2) при переполнении очереди требуется перевыделение памяти и копирование всех элементов в новый массив.

Реализации: на списке, на 2 стеках, на 6 стеках

Стек

Стек — структура данных, представляющая из себя упорядоченный набор элементов, в которой добавление новых элементов и удаление существующих производится с одного конца, называемого вершиной стека. Притом первым из стека удаляется элемент, который был помещен туда последним, то есть в стеке реализуется стратегия «последним вошел — первым вышел» (LIFO). Примером стека в реальной жизни может являться стопка тарелок : когда мы

хотим вытащить тарелку, мы должны снять все тарелки выше. Вернемся к описанию операций стека:

- empty — проверка стека на наличие в нем элементов,
- push — операция вставки нового элемента,
- pop — операция удаления нового элемента

Для стека с n элементами требуется O(n) памяти, так как она нужна лишь для хранения самих элементов.

Реализации: на массиве, на саморасширяющемся массиве, на списке

18. Ассоциативный массив на основе хэш-таблицы и на основе бинарного дерева.

Ассоциативный массив — абстрактный тип данных (интерфейс к хранилищу данных), позволяющий хранить пары вида «(ключ, значение)» и поддерживающий операции добавления пары, а также поиска и удаления пары по ключу

Для доступа к элементам индексного массива используются обычные целые числа, называемые индексами. У ассоциативного массива (словаря) эту функцию выполняют ключи. Они, в отличие от индексов, могут быть заданы не только числовым типом данных, но и, например строковым или булевым. Каждому элементу ассоциативного массива соответствует пара «ключ-значение» (key, value), и на нем определены четыре базовые операции:
INSERT – операция добавления пары в массив;
REASSIGN – операция изменения существующей пары;
DELETE – операция удаления пары из массива;
SEARCH – операция поиска пары в массиве.

Хеш-табли́ца(XT) — структура данных, реализующая интерфейс асс. мас. XT хранит пары (key, value) и может 3 операции: add pair, search и remove by pair key. 2 вида XT: с цепочками и открытой адресацией. XT содержит некоторый массив H, элементы которого есть пары (XT с открытой адресацией) или списки пар (XT с цепочками).Количество коллизий зависит от хеш-функции; чем лучше используемая хеш-функция, тем меньше вероятность их возникновения.
В реализациях, основанных на хэш-таблицах, среднее время оценивается как O(1). Но при этом не гарантируется высокая скорость выполнения отдельной операции: время операции INSERT в худшем случае оценивается как O(n). Операция INSERT выполняется долго, когда коэффициент заполнения становится высоким и необходимо перестроить индекс хэш-таблицы.

Хэш-таблицы плохи также тем, что на их основе нельзя реализовать быстро работающие дополнительные операции MIN, MAX и алгоритм обхода всех хранимых пар в порядке возрастания или убывания ключей.

HashMap — реал-ия интерфейса асс. мас. с исп. ХТ; unordered_map— реализация интерфейса асс. мас. исп. ХТ

Внутри HashMap - это массив. Элементы массива в документации называются бакетами (buckets). Будем следовать этому наименованию и мы. В бакете хранится первый элемент связанного списка. Вообще связанные списки можно обсудить в другом посте, главное, что нужно знать, связанный список - это цепочка объектов, каждый из которых имеет ссылку на следующий объект из цепочки. Имея первый элемент, можно по цепочке добраться до всех элементов списка. Таким образом, HashMap внутри - массив связанных списков. Элемент связанного списка - объект класса Entry, содержит ключ, значение и ссылку на следующий Entry.
==========
Класс **TreeMap** расширяет класс **AbstractMap** и реализует интерфейс **NavigatebleMap**. Он создает коллекцию, которая для хранения элементов применяет **дерево**. Объекты сохраняются в **отсортированном** порядке по возрастанию. Время доступа и извлечения элементов достаточно мало, что делает класс TreeMap блестящим выбором для хранения больших объемов отсортированной информации, которая должна быть быстро найдена.TreeMap **не синхронизирован**. По этому, если имеется множество потоков, работающих с данной коллекцией и хотя бы один поток может её изменять , то коллекцию нужно синхронизировать внешне.
TreeMap для хранения элементов применяет красно-черное дерево.
Красно-черное дерево это частный случай двоичного дерева поиска.
Двоичным деревом поиска (ДДП) называют дерево, все вершины которого упорядочены, каждая вершина имеет не более двух потомков (назовём их левым и правым), и все вершины, кроме корня, имеют родителя. Вершины, не имеющие потомков, называются листами. Подразумевается, что каждой вершине соответствует элемент или несколько элементов, имеющие некие ключевые значения, в дальнейшем именуемые просто ключами.
ДДП позволяет выполнять следующие основные операции:
1) Поиск вершины по ключу. 2) Определение вершин с мин. и макс. знач. ключа. 3) Переход к пред. или послед. вершине, в порядке, определяемом ключами.4)Вставка вершины. 5)Удаление вершины.
сложность операций в деревьях, близких к оптимуму будет O(log(n))

19. Задача сортировка массива. Алгоритмы сортировки.

Пусть требуется упорядочить N элементов. Каждый элемент представляет из себя запись, содержащую некоторую информацию и ключ, управляющий процессом сортировки. На множестве ключей определено отношение порядка

«<» так, чтобы для любых трёх значений ключей a,b,c выполнялись следующие условия:
закон трихотомии: либо a<b, либо a>b, либо a=b;
закон транзитивности: если a<b и b<c, то a<c.
Данные условия определяют математическое понятие линейного или совершенного упорядочения, а удовлетворяющие им множества поддаются сортировке большинством методов.
Задачей сортировки является нахождение такой перестановки p(1), p(2)…p(n) с индексами {1,2,3…N}после которой ключи расположились бы в порядке неубывания
Алгоритм сортировки — это алгоритм для упорядочения элементов в списке. Сортировка массива очень типичная задача на работу с массивами.
Время — основной параметр, характеризующий быстродействие алгоритма. Называется также вычислительной сложностью. Для упорядочения важны худшее, среднее и лучшее поведение алгоритма в терминах мощности входного множества A. Если на вход алгоритму подается множество A, то обозначим n = |A|. Для типичного алгоритма хорошее поведение — это O(n log n) и плохое поведение — это O(n^2). Идеальное поведение для упорядочения — O(n).
Память — ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных. Как правило, эти алгоритмы требуют O(log n) памяти. При оценке не учитывается место, которое занимает исходный массив и независящие от входной последовательности затраты, например, на хранение кода программы (так как всё это потребляет O(1)). Алгоритмы сортировки, не потребляющие дополнительной памяти, относят к сортировкам на месте.
Устойчивость — устойчивая сортировка не меняет взаимного расположения элементов с одинаковыми ключами

Пузырёк:

```
void bubble(int* a, int n) {
  for (int i=n-1; i>=0; i--) {
    for (int j=0; j<i; j++) {
      if (a[j] > a[j+1]) {
        int tmp = a[j];
        a[j] = a[j+1];
        a[j+1] = tmp;
      } } } }
```

вставками

```
void insert_sort(int *a, int n){
    int value;
    for(int i = 1; i < n; i++){
        value = a[i];
        for (int j = i - 1; j >= 0 && a[j] > value; j--){
            a[j + 1] = a[j];
```

```
            }
        a[j + 1] = value;   }   }
```

быстрая

```
void quicksort(int* a, int first, int last) {
   int i = first, j = last, x = a[(first + last) / 2];
   do {  while (a[i] < x) i++;
      while (a[j] > x) j--;
      if(i <= j) {
         if (i < j) swap(a[i], a[j]);
         i++; j--;   }
   } while (i <= j);
   if (i < last)       quicksort(a, i, last);
   if (first < j)      quicksort(a, first,j);
}
```

пузырьком — для каждой пары индексов производится обмен, если элементы расположены не по порядку. O(n^2).

вставками — определяем, где текущий элемент должен находиться в упорядоченном списке, и вставляем его туда. O(n^2).

слиянием — выстраиваем первую и вторую половину списка отдельно, а затем объединяем упорядоченные списки. O(n log n)

Сортировка с помощью двоичного дерева. O(n log n).

Блочная сортировка — требуется O(k) дополнительной памяти и знание о природе сортируемых данных, выходящее за рамки функций «переставить» и «сравнить». O(n).

Быстрая сортировка, в варианте с минимальными затратами памяти — O(nlog n) — среднее время O(n^2)— худший случай; широко известен как быстрейший из известных для упорядочения больших случайных списков; с разбиением исходного набора данных на две половины так, что любой элемент первой половины упорядочен относительно любого элемента второй половины; затем алгоритм применяется рекурсивно к каждой половине. При использовании O(n) дополнительной памяти, можно сделать сортировку устойчивой.

Сортировка выбором (англ. Selection sort) — поиск наименьшего или наибольшего элемента и помещение его в начало или конец упорядоченного списка. Сложность алгоритма O(n^2).

20. Бинарный поиск.

БП— алгоритм поиска объекта по заданному признаку в множестве объектов, упорядоченных по тому же самому признаку, работающий за логарифмическое время.

Задача: Пусть нам дан упорядоченный массив, состоящий только из целочисленных элементов. Требуется найти позицию, на которой находится заданный элемент.

Алгоритм: идея поиска заключается в том, чтобы брать элемент посередине, между границами, и сравнивать его с искомым. Если искомое больше(в случае

правостороннего — не меньше), чем элемент сравнения, то сужаем область поиска так, чтобы новая левая граница была равна индексу середины предыдущей области. В противном случае присваиваем это значение правой границе. Проделываем эту процедуру до тех пор, пока правая граница больше левой более чем на 1.

Код:

```
int binSearch(int[] a, int key):   // Запускаем бинарный поиск
   int l = -1                      // l, r — левая и правая границы
   int r = len(a)
   while l < r - 1                 // Запускаем цикл
      m = (l + r) / 2              // m — середина области поиска
      if a[m] < key
         l = m
      else
         r = m                    // Сужение границ
   return r
```

21. Алгоритмы на графах.

Граф состоит из вершин и ребер. Вершины графа - объекты любой природы; поскольку их должно быть конечное число, то мы будем обозначать их натуральными числами. Ребра графа соединяют некоторые из его вершин. Если ребра имеют направление, то граф называется ориентированным; в противном случае он неориентированный. Если в графе есть ребро C из вершины A в вершину B, то говорят, что ребро C инцидентно вершинам A и B, а также что вершина A смежна с вершиной B.
Матрица смежности Таблица, где как столбцы, так и строки соответствуют вершинам графа. В каждой ячейке этой матрицы записывается число, определяющее наличие связи от вершины-строки к вершине-столбцу (либо наоборот). Это наиболее удобный способ представления плотных графов.
Матрица инцидентности Таблица, где строки соответствуют вершинам графа, а столбцы соответствуют связям (рёбрам) графа.
Список смежности Список, где каждой вершине графа соответствует строка, в которой хранится список смежных вершин. Такая структура данных не является таблицей в обычном понимании, а представляет собой «список списков».
Список рёбер Список, где каждому ребру графа соответствует строка, в которой хранятся две вершины, инцидентные ребру.

Поиск в ширину — это один из основных алгоритмов на графах. В результате поиска в ширину находится путь кратчайшей длины в невзвешенном графе, т.е. путь, содержащий наименьшее число рёбер. Алгоритм работает за O(n+m), где n — число вершин, m — число рёбер.

Работа: Поиск в ширину работает путём последовательного просмотра отдельных *уровней* графа, начиная с узла-источника u. Рассмотрим все ребра (u,v), выходящие из узла u. Если очередной узел v является целевым узлом, то поиск завершается; в противном случае узел v добавляется в очередь. После

того, как будут проверены все рёбра, выходящие из узла u, из очереди извлекается следующий узел u, и процесс повторяется.

Описание: 1) Поместить узел, с которого начинается поиск, в изначально пустую очередь. 2) Извлечь из начала очереди узел и пометить его как развёрнутый. а) Если узел u является целевым узлом, то завершить поиск с результатом «успех». б) В противном случае, в конец очереди добавляются все преемники узла u, которые еще не развернуты и не находятся в очереди. 3) Если очередь пуста, то все узлы связного графа были просмотрены, =>, целевой узел недостижим из начального; завершить поиск с результатом «неудача». 4) Вернуться к п. 2.

```
queue <int> turn;
int used[1000];
int matrix[1000][1000];
void bfs ( ) {
     while ( !turn.empty() ) {
          int ind=turn.front();
          turn.pop();
          for ( int i=0; i<1000; i++ )   {
               if ( matrix[ind][i]==1 ) {
                    turn.push(i);
               }         }      }  }
```

Поиск в глубину — один из методов обхода графа. Стратегия поиска в глубину состоит в том, чтобы идти «вглубь» графа, насколько это возможно. Алгоритм поиска описывается рекурсивно: перебираем все исходящие из рассматриваемой вершины рёбра. Если ребро ведет в вершину, которая не была рассмотрена ранее, то запускаем алгоритм от этой не рассмотренной вершины, а после возвращаемся и продолжаем перебирать рёбра. Возврат происходит в том случае, если в рассматриваемой вершине не осталось рёбер, которые ведут в не рассмотренную вершину. Если после завершения алгоритма не все вершины были рассмотрены, то необходимо запустить алгоритм от одной из нерассмотренных вершин.

```
List<Integer>[] graph = readGraph();
boolean[] used = new boolean[graph.length];
public static void dfs(int pos) {
   used[pos] = true;
   System.out.println(pos);
   for (int next : graph[pos]){
      if (!used[next]){          dfs(next);
      }   } }
```