1. Основные идеи ООП: использование объекта в качестве основной компоненты программы и децентрализация управления, реализуемая представлением программы совокупностью взаимодействующих объектов.

**Основными понятиями, используемыми в ООП, являются класс, объект, наследование, инкапсуляция и полиморфизм.**

Данные и методы обработки данных неразрывно связаны : вместе они являются классом.

Экземпляры классов - объекты

ОО программа - совокупность объектов, взаимодействующих посредством передачи сообщения для выполнения требуемых функция. >> ***Объектная декомпозиция***

Идеи:

1. Моделирование поведения реального мира

Концептуальная модель предметной области

*(понять, что именно необходимо сделать для решения поставленной задачи)*

2. Возможность создавать свой тип данных

АТД (Абстрактные типы данных) – классы.

Классы включают в себя данные и их обработку(функции)

*Данные-свойства*

*Обработка-поведение*

Класс => объекты

3. Инкапсуляция

**Скрытая реализация данных и открытая реализация обработки.**

*Свойства - закрытые*

*Поведение - открытое*

4. Наследования и шаблоны

Позволяет повысить степень повторного использования кода, то есть меньше кода для отдельных случаев, один код для всех типовых событий.

Для реализации параметризованных типов используется ключевое слово template. Шаблоны определения класса и шаблоны определения функции позволяют многократно использовать код простым способом, безопасным по отношению к типу данных, который разрешает компилятору автоматически реализовать этот тип данных.

Объявление шаблона имеет вид:

*template <шаблонные\_аргументы> объявление*

**Наследование** – это механизм получения нового класса на основе существующего класса. Существующий класс может быть дополнен или изменен для создания производного класса.

5. Полиморфная обработка родственных объектов путем интерпретации вызова функции на этапе вызова программы.

*Открытое наследование + замещение*

2. Сущность ООП: моделирование поведения объектов реального мира, наглядное представление объектов, создание и использование АТД, производные АТД.

Концепция объектно- Сущность ООП: моделирование поведения объектов реального мира, наглядное представление объектов, создание и использование АТД, производные АТД.

**Объектно-ориентированное программирование — это программирование, сфокусированное на данных, причем данные и поведение неразрывно связаны. Вместе данные и поведение представляют собой класс, а объекты являются экземплярами класса. ООП рассматривает вычисления как моделирование поведения. То, что моделируется, является объектами, представленными вычислительной абстракцией.**

**Абстрактный тип данных, АТД (abstract data type, ADT) – определяемые пользователем расширения исходных типов языка. АТД состоит из набора значений и операций, которые могут влиять на эти значения. Например, в С нет типа данных для комплексных чисел, а C++ позволяет добавить такой тип и интегрировать его с существующими.**

**ООП - это не просто набор некоторых свойств, добавленных в уже существующие языки. Это новый шаг в осмыслении задач и разработки ПО.**

3. Концепции ООП. Инкапсуляция и расширяемость типов.

**Инкапсуляция – сокрытие поведения объекта внутри него. Объекту «водитель» не нужно знать, что происходит в объекте «машина», чтобы она ехала. Это ключевой принцип ООП.**

**Инкапсуляция (encapsulation) - это механизм, который объединяет данные и код, манипулирующий этими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования. В объектно-ориентированном программировании код и данные могут быть объединены вместе; в этом случае говорят, что создаётся так называемый "чёрный ящик". Когда коды и данные объединяются таким способом, создаётся объект (object). Другими словами, объект - это то, что поддерживает инкапсуляцию.**

**Внутри объекта коды и данные могут быть закрытыми (private). Закрытые коды или данные доступны только для других частей этого объекта. Таким образом, закрытые коды и данные недоступны для тех частей программы, которые существуют вне объекта. Если коды и 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 определённого пользователем типа. Может показаться странным, что объект, который объединяет коды и данные, можно рассматривать как переменную. Однако применительно к объектно-ориентированному программированию это именно так. Каждый элемент данных такого типа является составной переменной.**

Скрытая реализация данных и открытая реализация обработки.

*Свойства - закрытые*

*Поведение - открытое*

Инкапсуляция – это способность скрывать внутренние детали при предоставлении открытого интерфейса к определяемому пользователем типу.

В С++ в структурах и классах возможно реализация инкапсуляции благодаря ключевым словам:

*private (закрытый)*

*protected(защищенный)*

*public (открытый)*

**Без этого ключевого слова члены класса по умолчанию закрыты**

4. Концепции ООП. Семантика объекта: данные как отражение состояния объекта и методы как средства обеспечения взаимодействия объектов. Представление данных. Классификация методов: конструкторы, деструкторы, селекторы, модификаторы.

**По функциональному назначению методы класса делятся на следующие категории:**

**- конструкторы – предназначены для инициализации состояния экземпляров класса при их создании;**

**- деструкторы – предназначены для выполнения каких-то дополнительных действий в момент уничтожения экземпляров класса;**

**- селекторы – предназначены для обработки состояния класса без его изменения;**

*void print() const;*

**- модификаторы – предназначены для изменения состояния класса;**

*void input(int a,int b,char \*c);*

**- итераторы – предназначены для организации последовательного доступа к элементам данных, определяющих состояние некоторого (одного) экземпляра класса.**

**По отношению к классу методы делятся на следующие две категории:**

**- функция-член класса – функция, принадлежащая самому классу и не существующая вне класса; прототипы функций-членов класса включены в определение класса;**

**- функция-друг класса – внешняя по отношению к классу функция, которая может существовать вне класса, но имеет доступ к закрытой (и защищенной) части класса. Прототип функции-друга класса также включается в определение класса, но начинается специальным ключевым словом** friend**.**

5. Концепции ООП. Классы объектов: назначение и семантика, классы и абстрактные типы данных, наглядное представление классов. Объекты как экземпляры классов. Основные действия с объектами: создание, инициализация, обработка, разрушение.

Инициализация открытых объектов происходит напрямую, закрытых объектов с помощью конструктора. **Конструктор** — это особый тип метода класса, который автоматически вызывается при создании объекта этого же класса.

**Деструктор** — это функция, которая вызывается при разрушении объекта.

Зачастую классы представляют собой проекцию некоторых реальных объектов с четко выраженными свойствами: товар, у которого есть личный номер, тип, размеры и т.д.

Объекты - это переменные классов. То есть переменные обладающие типом класс.

6. Концепции ООП. Конструкторы и деструкторы, основное назначение.

Однако, в языке С++ есть механизм, позволяющий решить эту проблему иным путём. Это - конструкторы.

**Конструктор (construct - создавать) - это специальная функция-член класса,объявленная с таким же именем, как и класс.**

**Еще кое-что о конструкторах…**

* **Конструктор без параметров называют конструктором по умолчанию. Такой конструктор обычно присваивает переменным-членам класса наиболее часто используемые значения.**

**Деструктор - это специальная функция класса, которая автоматически вызывается при уничтожении объекта - например, когда объект выходит из области видимости. Деструктор может выполнять любые задачи, в момент удаления объекта. Например, если в конструкторе была динамически выделена память, то деструктор должен освободить эту память перед удалением объкта класса.**

**Основные особенности при работе с деструктором.**

1. **Деструктор не принимает никаких параметров и не возвращает никаких значений.**
2. **Класс может иметь только один деструктор.**

**Пример последовательности, в которой вызываются конструкторы и деструкторы.**

**Конструкторы - создают объекты**

Спец. компонентные функции, предназначенные для создания объектов

*Под созданием объектов понимается выделение памяти и заполнение начальными значениями.*

Конструкторы по-умолчанию

*pixel()*

Копирующие конструкторы

*pixel(const pixel&m)*

*const запрещает функции менять значение параметра*

Конструкторы с параметрами

*pixel(int)*

*pixel(int a,int b, char \*c)*

**Деструкторы - уничтожают объекты, т.е. освобождают память**

Освобождение памяти

Ситуации вызова деструкторов:

1. Когда объекты становятся ненужными. Деструктор вызывает явно.
2. Когда выходим за пределы локальной области видимости, где были созданы эти объекты. Т.е. за {}
3. *Сложные объекты.* При уничтожении сложного объекта, вначале удаляются объекты меньше, из которых состоит данный сложный объект.
4. Динамические объекты

*~pixel()*

7. Концепции ООП. Перегрузка операторов и функций. Ключевые слова: friend, operator.

**Перегруженные операторы должны быть нестатической функцией-членом класса или глобальной функцией. Глобальная функция, которой требуется доступ к частным или защищенным членам класса, должна быть объявлена в качестве дружественной функции этого класса. Глобальная функция должна принимать хотя бы один аргумент, имеющий тип класса или перечисляемый тип либо являющийся ссылкой на тип класса или перечисляемый тип.**

· **Перегруженные операторы не могут иметь аргументов по умолчанию.**

· **Все перегруженные операторы, кроме присваивания (**operator=**), наследуются производными классами.**

· **Первым аргументов операторов, перегруженных в виде функций-членов, всегда является тип класса объекта, для которого вызывается этот оператор (класса, в котором объявлен оператор, или класса, производного от этого класса). Для первого аргумента никакие преобразования не предоставляются.**

Дружественные функции нарушают принцип инкапсуляции. Они могут быть как компонентными, так и внешними. Они получают данные только из своих объявлений и только через параметры.

*Указатель на себя не работает*

*Параметры не могут быть указателями*

Ситуации:

1. Одновременная работа с несколькими классами
2. На время отладки
3. Неявное преобразование типов данных, перегрузка бинарных операторов

Перегрузка операторов:

Практически все унарные и бинарные операторы могут быть перегружены, как внешними, так и компонентными функциями.

**Бинарные: + - \* / > < && и т.д.**

**Унарные: + - ! ~**

Операторы **= [] () ->** перегружаются только нестатическими компонентными функциями

Внешними функциями это реализовать невозможно

Операторы **new и delete** тоже можно перегрузить.

Они перегружаются только статическими компонентными функциями.

Операторы :: **sizeof ? : . ,** перегрузить нельзя

*pixel operator = (const pixel&m)*

*friend pixel operator - (pixel a, pixel b)*

8. Концепции ООП. Шаблоны и обобщенное программирование. Ключевое слово: template. Контейнерные классы.

Шаблоны — средство языка C++, предназначенное для кодирования обобщенных алгоритмов, без привязки к некоторым параметрам.

Обобщённое программирование –описании данных и алгоритмов, которое можно применять к различным типам данных.

Ответственность - смысл создания конкретного типа данных, назначение класса.

В примере с точками будет параметром типа данных задания координат (int, float, double)

чтобы конкретный пользователь сам определял необхдимый для него тип.

**Это дает возможность повышении повторного использования кода.**

Контейнерные классы - это универсальные шаблонные классы, предназначенные для хранения элементов заданного типа в смежных областях памяти.

9. Концепции ООП. Основные составляющие STL: контейнеры, итераторы, алгоритмы.

**В языке C ++ был разработан мощный набор инструментов для создания разнообразных программ — *библиотека стандартных шаблонов* (*Standard Template Library*, STL). Эта библиотека, фактически, является неотъемлемой составляющей языка C++, что является большим достижением ее разработчиков. Библиотека STL включена в стандарт языка C++ и содержит универсальные шаблонные классы и функции, которые реализуют широкий спектр алгоритмов и структур данных. Эти средства программирования можно применять практически к любым типам данных.**

**Различают следующие основные составляющие (компоненты) библиотеки STL:**

· **контейнеры (container)**

· **алгоритмы (algorithm)**

· **итераторы (iterator)**

· функциональные объекты или функторы (functor).

В программировании *контейнер* — это объект, который сохраняет другие объекты внутри себя. Объекты, хранящиеся в контейнерах, могут быть как базовых (примитивных) типов так и экземплярами классов с соответствующим необходимым функционалом.

В библиотеке STL реализованы следующие контейнеры:

· bitset — набор битов;

· deque — двухсторонняя очередь;

· list — линейный список;

· map — ассоциативный контейнер, построенный по принципу key:value, в котором каждому ключу key соответствует значение value;

· multimap — ассоциативный контейнер, в котором одному значению (key) соответствует несколько значений (value1, value2, …, valueN);

· multiset — множество, в котором один и тот же элемент может встречаться несколько раз;

· priority\_queue — очередь с приоритетами;

· queue — очередь;

· set — множество, в котором каждый элемент встречается только один раз;

· stack — стек;

· vector — динамический массив.

Содержимое контейнеров обрабатывается с помощью алгоритмов. Алгоритмы позволяют обрабатывать контейнеры на любой вкус: инициализировать, сортировать содержимое контейнеров, преобразовывать, реализовывать различные виды поиска и тому подобное.

10. Концепции ООП. Наследование: базовый и производный класс, иерархия классов.

**Наследование - это механизм получения нового класса на основе уже существующего. Существующий класс может быть дополнен или изменен для создания нового класса.**

**Существующие классы называются базовыми, а новые – производными. Производный класс наследует описание базового класса; затем он может быть изменен добавлением новых членов, изменением существующих функций- членов и изменением прав доступа. Таким образом, наследование позволяет повторно использовать уже разработанный код, что повышает производительность программиста и уменьшает вероятность ошибок. С помощью наследования может быть создана иерархия классофв, которые совместно используют код и интерфейсы.**

**Наследуемые компоненты не перемещаются в производный класс, а остаются в базовых классах. Сообщение, обработку которого не могут выполнить методы производного класса, автоматически передается в базовый класс. Если для обработки сообщения нужны данные, отсутствующие в производном классе, то их пытаются отыскать автоматически и незаметно для программиста в базовом классе.**

**При наследовании некоторые имена методов и данных базового класса могут быть по-новому определены в производном классе. В этом случае соответствующие компоненты базового класса становятся недоступными из производного класса. Для доступа к ним используется операция указания области видимости '::'.**

**В иерархии производный объект наследует разрешенные для наследования компоненты всех базовых объектов (public, protected).**

**Допускается множественное наследование – возможность для некоторого класса наследовать компоненты нескольких никак не связанных между собой базовых классов. В иерархии классов соглашение относительно доступности компонентов класса следующие:**

**private – Член класса может использоваться только функциями-членами данного класса и функциями-"друзьями" своего класса. В производном классе он недоступен.**

**protected – То же, что и private, но дополнительно член класса с данным атрибутом доступа может использоваться функциями-членами и функциями-"друзьями" классов, производных от данного.**

**public – Член класса может использоваться любой функцией, которая является членом данного или производного класса, а также к public -членам возможен доступ извне через имя объекта.**

**Следует иметь в виду, что объявление friend не является атрибутом доступа и не наследуется.**

**Синтаксис определение производного класса:**

**class имя\_класса : список\_базовых\_классов**

**{список\_компонентов\_класса};**

11. Концепции ООП. Полиморфизм. Абстрактный базовый класс.

Полиморфизм - это способ придать одним и тем же командам и операторам различные назначения, в зависимости от заданных параметров.

Абстрактный класс — это класс, содержащий хотя бы один абстрактный метод, он описан в программе, имеет поля, методы и не может использоваться для непосредственного создания объекта. В полиморфизм подтипов предполагает использование единого имени (идентификатора) при обращении к объектам нескольких разных классов, при условии, что все они являются подклассами одного общего надкласса (суперкласса).

Полиморфизм подтипов состоит в том, что несколько типов формируют подмножество другого типа (их базового класса) и потому могут использоваться через общий интерфейс.

12. Концепции ООП. Обработка исключительных ситуаций, проверка утверждений.

Обработка исключительных ситуаций — механизм языков программирования, предназначенный для описания реакции программы на ошибки выполнения и другие возможные проблемы (исключения), которые могут возникнуть при выполнении программы и приводят к невозможности дальнейшей отработки программой её алгоритма.

Область действия обработчиков начинается специальным ключевым словом try и заканчивается перед описанием обработчиков (catch). Обработчиков может быть несколько, один за одним, и каждый может указывать тип исключения, который он обрабатывает.

13. Концепции ООП. Преимущества и недостатки ООП. Основные свойства языка ООП. Место языка Си++ во внедрении и развитии средств ООП. Значение языка Си++ для профессиональной подготовки программистов.

Достоинства ООП: - «более естественная» декомпозиция программного обеспечения, которая существенно облегчает его разработку.

- объектный подход предлагает новые способы организации программ, основанные на механизмах наследования, полиморфизма, композиции, наполнения.

- механизмы позволяют конструировать сложные объекты из сравнительно простых. В результате существенно увеличивается показатель повторного использования кодов и появляется возможность создания библиотек классов для различных применений.

Недостатки ООП обуславливаются следующим: - разработка библиотек классов и их использование требуют существенных трудозатрат. - не всегда легко определить, какие поля и методы фактически относятся к данному классу.

- Код для обработки сообщения иногда «размазан» по многим методам (иначе говоря, обработка сообщения требует не одного, а многих методов, которые могут быть описаны в разных классах). Основной недостаток ООП

- некоторое снижение быстродействия за счет более сложной организации программной системы.

К основным свойствам объектно-ориентированных языков можно отнести то, что основными понятиями в ООП являются класс и объект. Класс является описанием на языке программирования какой-либо сущности

Объект - это какая-либо сущность в адресном пространстве, которая строится на основании класса. С++ можно назвать языком программирования, который в наибольшей мере соответствует парадигме ООП. В него входят все основные концепции. Для профессиональной подготовки программистов язык C++ является важным по причине того, что он включает в себя все концепции + обладает ручным управлением памятью, что позволяет сформировать у программиста методику работы с ресурсами компьютера.

14. Классы: компонентные данные, компонентные функции, способы задания доступа, синтаксическая структура определения класса, объявление, определение и инициализация объектов, объекты объявленные как константы.

**Объект** – структура данных, содержащая описание свойств внешнего объекта программирования. **Метод** – функция, работающая с объектом. **Класс** – описание структуры объекта и методов работы с ним.

15. Область видимости класса: оператор разрешения области видимости ( : : ) - унарная форма, бинарная форма; вложенные классы - классы, вложенные в классы, классы, вложенные в блоки, вложенные определения функций.

***Локальная область видимости*** – это область видимости внутри блока. Каждая функция – это отдельная область видимости. Внутри функции может быть несколько блоков, заключенных в фигурные скобки {…}, также образующих отдельные области видимости. Для переменных, объявленных в блоке, область видимости – от точки объявления до конца блока.

Каждый класс представляет отдельную область видимости. Имена членов класса локальны в этом классе. Область видимости членов класса (данных и функций) не зависит от точки их объявления, то есть член класса виден во всем классе. Каждая член-функция определяет свою собственную локальную область видимости, аналогично обычной функции. Можно явно указать область видимости в С++, используя оператор разрешения контекста (области видимости) ::.

Унарная форма записи: :: <идентификатор> Унарная форма используется для обращения к имени, относящемуся ко внешнему контексту и скрытому локальным контекстом или контекстом класса. Бинарная форма записи: <имя\_класса> :: <идентификатор> Бинарная форма используется для ссылки на контекст класса с целью устранения неоднозначности имен, которые могут повторно использоваться внутри класса. class cl1 {void f() {...} ...}; cl1 :: f(); // обращение к f() из cl1

Использование вложенных классов - это ограниченная форма вложенности функций. Функции должны определяться внутри локального класса и на них нельзя ссылаться внутри этого контекста. Вложенный класс не находится в области действия включающего его класса.

16. Компонентные данные: назначение, основные свойства, синтаксис и семантика определений компонентных данных, статические компонентные данные, статические константы, изменяемые компонентные данные (mutable).

Как уже отмечалось, каждый вновь создаваемый объект класса имеет свою собственную копию компонентных данных. Чтобы эти компоненты класса были в единственном экземпляре и не тиражировались при определении каждого нового объекта класса, они должны быть объявлены в классе как статические (***static***):

***static имя\_типа имя\_статического\_компонента\_данных;***

Как видим, статический компонент данных является частью класса, но не является частью объекта этого класса. Все объекты класса, в котором были объявлены статические компоненты данных, так и объекты его производных классов, теперь могут совместно использовать эти общие для них компоненты.

Статические компоненты данных класса размещаются в памяти отдельно от его нестатических компонентов, причем память статическим компонентам выделяется только после их определения вне определения класса.

Для внешнего определения статических компонентных данных можно воспользоваться следующими конструкциями:

***имя\_типа имя\_класса::имя\_статического\_компонента\_данных;***

или

***имя\_типа имя\_класса::имя\_статического\_компонента\_данных = инициализатор;***

или

***имя\_типа имя\_класса::имя\_статического\_компонента\_данных (инициализатор);***

Эти конструкции должны быть размещены в глобальной области (***global scope***) после определения класса.

17. Компонентные функции: назначение, основные свойства, доступ к компонентным функциям, доступ компонентных функций к компонентным данным - функции селекторы, функции модификаторы.

Параметры функций - интерфейс функций

Компонентные функции - это функции обработки, обязательно находящиеся в классе. Имеет доступ ко всем компонентам класса, в котором находится.

Предназначена для работы с объектами класса.

18. Встроенные компонентные функции: назначение, синтаксис и семантика определения, рекомендации по использованию; компонентные функции, определяемые вне класса: назначение, синтаксис и семантика определения и объявления, рекомендации по использованию.

Вне тела класса компонентная функция определяется таким образом: тип имя класса:: имя\_функции (спецификация\_формальных\_параметров) {тело\_принадлежащей\_классу\_функции} Описание класса с внешним определением его компонентных функций дает возможность, не меняя интерфейс объектов класса с другими частями программы, по-разному определять его компонентные функции.

19. Указатель this, компонентные функции типа static и const (статические и постоянные компонентные функции): назначение, синтаксис и семантика определения и объявления, рекомендации по использованию.

\*this - указатель на себя. Компонентные функции первым параметром видит переменную - объект через который вызывается. Она содержит указатель на этот объект.

*s=d.f(t,k)*

Чтобы компонент класса был в единственном экземпляре и не тиражировался при создании каждого нового объекта класса, он должен быть определен в классе как статический, т.е. должен иметь атрибут static.

static int k; //

объявление static-члена в объявлении класса

*- статические компоненты-функции не имеют указателя;*

*- не могут быть объявлены две одинаковые функции с одинаковыми именами и типами аргументов;*

20. Конструкторы: назначение, наличие нескольких конструкторов у одного класса - причины, разделение функций, способы задания конструктора: по умолчанию или явное.

Спец. компонентные функции, предназначенные для создания объектов

*Под созданием объектов понимается выделение памяти и заполнение начальными значениями.*

Конструкторы по-умолчанию

*pixel()*

Копирующие конструкторы

*pixel(const pixel&m)*

*const запрещает функции менять значение параметра*

Конструкторы с параметрами

*pixel(int)*

*pixel(int a,int b, char \*c)*

21. Явное описание (определение или объявление) конструктора, размещение определения в теле класса и влияние размещения на функции конструктора, размещение определения вне своего класса, параметры конструктора и возможности их использования.

22. Конструкторы для инициализации объектов, инициализаторы конструктора; конструкторы для преобразования типов, ключевое слово explicit.

Конструкторы в C++ (﻿﻿ctors), имеющие только один параметр, автоматически выполняют неявное преобразование типов.

Например, если передать int, когда конструктор ожидает параметр указателя на строку, компилятор добавит код, необходимый для преобразования int в указатель на строку.

Ключевое слово **explicit** применяется для указания явного преобразования типов.

*using System; class MyClass{ public MyClass(int i) { x=i; }*

*public static explicit operator MyClass(int i)*

*{ return new MyClass(i); }*

*private int x;*

оператор преобразования типа MyClass, который преобразует int

23. Конструкторы умолчания: назначение, способы задания, правила описания.

*pixel()*

Конструкторы по умолчанию позволяют создавать объекты без задания готовых значений.

*pixel t1 - конструктор по-умолчанию без параметров*

Конструктор по умолчанию — конструктор, который может быть вызван без аргументов.

Если программист не определил конструктор в классе - то компилятор создаст конструктор (без параметров) для создания объектов этого класса. Если производный класс не вызывает явным образом конструктор базового класса, то конструктор по умолчанию вызывается неявно.

*class myclass{}*

*;int main()*

*{myclass mc(); }*

24. Конструкторы копирования: назначение, способы задания, правила описания.

*pixel(const pixel&m)*

*const запрещает функции менять значение параметра*

Когда в классе есть хотя бы 1 указатель, то в этом классе обязательно должен быть ***копирующий конструктор***.

Это необходимо для того того, чтобы избежать ситуации двойного указания. Оно опасно тем, что изменив параметр с помощью одного указателя меняется и значение другого, т.к. в этом случае они указывают на одно и тоже место в памяти.

25. Вызов конструктора: назначение, способы реализации, задание аргументов; явный и автоматический вызовы: ситуации использования, рекомендации по применению.

26. Деструкторы: назначение, основные свойства, способы задания: по умолчанию или явное.

Освобождение памяти

Ситуации вызова деструкторов:

1. Когда объекты становятся ненужными. Деструктор вызывает явно.
2. Когда выходим за пределы локальной области видимости, где были созданы эти объекты. Т.е. за {}
3. *Сложные объекты.* При уничтожении сложного объекта, вначале удаляются объекты меньше, из которых состоит данный сложный объект.
4. Динамические объекты

*~pixel()*

27. Явное описание (определение или объявление) деструктора, размещение определения в теле класса и влияние размещения на функции деструктора, размещение определения вне своего класса, параметры деструктора и возможности их использования.

1. Когда объекты становятся ненужными. Деструктор вызывает явно.
2. Когда выходим за пределы локальной области видимости, где были созданы эти объекты. Т.е. за {}
3. *Сложные объекты.* При уничтожении сложного объекта, вначале удаляются объекты меньше, из которых состоит данный сложный объект.
4. Динамические объекты

28. Вызов деструктора:

назначение, способы реализации, явный и автоматический вызовы: ситуации использования, рекомендации по применению.

1. Когда объекты становятся ненужными. Деструктор вызывает явно.
2. Когда выходим за пределы локальной области видимости, где были созданы эти объекты. Т.е. за {}
3. *Сложные объекты.* При уничтожении сложного объекта, вначале удаляются объекты меньше, из которых состоит данный сложный объект.
4. Динамические объекты

29. Специальный полиморфизм – преобразование АТД к встроенному типу.

*Сец. полиморфизм* - это полиморфизм специального назначения.

*Полиморфизм* - это способ придавать одним и тем же командам и операторам различные назначения, в зависимости от заданных параметров.

Специальный полиморфизм:

1. Перегрузка функций
2. Перегрузка преобразования АТД
3. Перегрузка операторов

*c=x+ k*

*float=int + char > int=int+int*

*char>int int<float*

*pixel(int x) -* конструктор, который автоматически преобразует тип int в pixel

Преобразование типа класса в иной:

\_\_\_\_ (нет возвращаемого типа) operator <тип> ()

{ …

return (<тип>)

}

*y=(тип)x*

*y=(float)x >> y = static\_cast<тип>(x);*

*float=float*

30. Специальный полиморфизм – перегрузка функций, алгоритм выбора перегруженной функции.

Сигнатура параметров - количество параметров, их последовательность и т.д.

Выбор программы компилятором:

1. Проверка на строгое соответствие типа > end
2. Попытка применить стандартное повышение типа >end
3. Стандартное преобразование типа

*f(void\*) f(&x)*

*int\* float\* type\* int x\**

*void f (БК\*) базовый класс*

*f(ПК\*) преобразованный класс*

*Эллипсис* - функция с неопределенным количеством данных.

*void(...)*

31. Дружественные функции: основные свойства, номенклатура, необходимость, полезность и опасность применения, наборы дружественных функций, дружественные классы.

Могут быть и внешними, и компонентными.

*friend* - ключевое слово

Дружественные функции нарушают принцип инкапсуляции, т.е. дают доступ к данным тем лицам, кому этот доступ не предназначен изначально.

Дружественные функции получают данные только из своих объявлений, только через параметры.

1. указатель на себя не работает
2. параметры не могут быть указателями
3. Дружественные функции нельзя вызывать через объекты классов, друзьями которых они являются, а также через указатели на эти объекты.
4. На дружественную функцию не распространяется действие спецификаторов доступа (public, protected, private)
5. Дружественная функция не может быть компонентной функцией того класса, по отношению к которому определяется как дружественная, зато она может быть просто глобальной функцией, а также компонентной функцией другого ранее определенного класса.
6. Дружественная функция может быть дружественной по отношению к нескольким классам.

**Ситуации, в которых допустимо использование дружественных функции:**

1. Одновременная работа с несколькими классами

*void f(type1 a, type2 b, type3 c);*

***class type 1{***

***public:***

***friend void f(type1 a, type2 b, type3 c);***

***}***

1. На время отладки
2. Неявное преобразование типов данных при перегрузке бинарных операторов

Класс тоже может быть дружественным другому классу.

Особенности:

1. Дружественный класс должен быть определен вне тела класса, "предоставляющего дружбу".
2. Все компонентные функции класса-друга будут являться дружественными для другого класса без указания спецификатора friend.

Все компоненты класса доступны в дружественном классе, но не наоборот.

32. Дружественные функции: объявление – размещение, синтаксис, семантика, требования к параметрам и типу возвращаемого значения; определение – размещение, область видимости; вызов функции.

*void f(type1 a, type2 b, type3 c);*

***class type 1{***

***public:***

***friend void f(type1 a, type2 b, type3 c);***

***}***

33. Специальный полиморфизм – перегрузка операторов: синтаксис, семантика, приоритет и порядок выполнения, доступные и недоступные для перегрузки операторы, перегружаемые как обычные функции, только как компонентные функции.

Перегрузка операторов:

Практически все унарные и бинарные операторы могут быть перегружены, как внешними, так и компонентными функциями.

**Бинарные: + - \* / > < && и т.д.**

**Унарные: + - ! ~**

Операторы **= [] () ->** перегружаются только нестатическими компонентными функциями

Внешними функциями это реализовать невозможно

Операторы **new и delete** тоже можно перегрузить.

Они перегружаются только статическими компонентными функциями.

Операторы :: **sizeof ? : . ,** перегрузить нельзя

34. Перегрузка унарных и бинарных операторов: синтаксис и семантика.

Перегрузка операторов:

Практически все унарные и бинарные операторы могут быть перегружены, как внешними, так и компонентными функциями.

**Бинарные: + - \* / > < && и т.д.**

**Унарные: + - ! ~**

Операторы **= [] () ->** перегружаются только нестатическими компонентными функциями

Внешними функциями это реализовать невозможно

Операторы **new и delete** тоже можно перегрузить.

Они перегружаются только статическими компонентными функциями.

Операторы :: **sizeof ? : . ,** перегрузить нельзя

*friend pixel operator - (pixel a, pixel b) - объявление внешней функции*

*pixel operator - (pixel b) -* компонентная функция. Второй(первый) параметр она получает от указателя \*this от объекта, через который вызвана.

Оба параметра должны иметь один тип, иначе компилятор будет пытаться преобразовать все к одному типу, повышая его: int>float>double и т.д.

Всегда должна быть возможность приводить параметры внешней функции к одному типу.

35. Перегрузка операторов присваивания и индексирования: синтаксис и семантика.

Операторы **= [] () ->** перегружаются только нестатическими компонентными функциями

Внешними функциями это реализовать невозможно

*public:*

*float operator [] (int k){return p[k]}*

*privat:*

*float \*p*

*public:*

*float & operator() {(int a, int b}; -* создает матрицу

-> оператор указ. структуры

класс А

класс B содержит в закрытой части A\*p - закрытый указатель на класс А.

-> - унарный оператор, содержащий адреса всех объектов класса А.

Операнд обычно явл. указателем, но при перегрузке -> явл. либо объектом класса, либо ссылкой.

Идея в том, чтобы объекты класса В или ссылки на эти объекты могли иметь управляемый доступ к объектам класса А.

Чтобы объекты класса В имели доступ к компонентным функциям класса А, как к своим.

Если не перегружен: b1.f\_B()->f\_A;

Перегружен: b1->f\_A()

36. Перегрузка операторов ввода-вывода («, »): синтаксис и семантика.

37. Перегрузка оператора () для индексирования: синтаксис и семантика.

оператор индексирования обычно перегружается, когда тип класса представляет агрегат .

Теоретически, операция индексирования должна возвращать ссылку на элемент, который содержится в агрегате.

*double& operator [] (int);* // Перегруженный оператор индексации double& *Vector::operator [] (int n) {* // Перегруженный оператор индексации // для проверки выхода за границы массива

*if(n < 0)* //

В случае если индекс меньше нуля, // то возвращаем нулевой элемент массива *{ cout << ""; return vect[0]; }*

38. Перегрузка операторов доступа к элементу класса через указатель на объект : синтаксис и семантика.

-> оператор указ. структуры

класс А

класс B содержит в закрытой части A\*p - закрытый указатель на класс А.

-> - унарный оператор, содержащий адреса всех объектов класса А.

Операнд обычно явл. указателем, но при перегрузке -> явл. либо объектом класса, либо ссылкой.

Идея в том, чтобы объекты класса В или ссылки на эти объекты могли иметь управляемый доступ к объектам класса А.

Чтобы объекты класса В имели доступ к компонентным функциям класса А, как к своим.

Если не перегружен: b1.f\_B()->f\_A;

39. Перегрузка операторов new и delete: синтаксис и семантика.

Перегрузка операторов new и delete функция должна быть, либо не статичной компонентой функцией, либо иметь хотя бы один аргумент типа класс или ссылку на класс.

Для перегрузки операторов new и delete может использоваться следующий формат:

*void \*operator new(size\_t размер)* { // выполнение выделения return указатель\_на\_память; }

*void operator delete(void \*p)* { // освобождение памяти, на которую указывает р }

Функция delete получает указатель на область памяти, которую необходимо освободить.

Для перегрузки операторов new и delete применительно к масси­вам:

*void \*operator new[](size\_t размер)* { // выполнение выделения return указатель\_на\_память; }

*void operator delete[](void \*р)* { // освобождение памяти, на которую указывает р }

40. Параметрический полиморфизм – шаблонные классы и шаблонные функции: назначение, параметризованные типы данных, синтаксис и семантика.

Шаблоны классов­ – это предписание, по которому компилятор создаёт класс для заданных входных параметров

Компилятор автоматически создаст корректный код, основываясь на типе данных, указанном перед компиляцией.

Общая форма объявления класса-шаблона:

*template < class птип> class имя\_класса { ... }*

Шаблоны функций­ – это предписание, по которому компилятор создает функцию для заданных типов данных.

41. Шаблоны классов: назначение, синтаксис и семантика, объявление и определение шаблона, дружественные шаблонному классу функции, статические элементы шаблонного класса, аргументы шаблона класса.

Параметрами шаблонов могут быть:

* параметры-типы,
* параметры обычных типов,
* параметры-шаблоны

*template< class T > class List*

*{ /\* ... \*/*

*public: void Add( const T& Element );*

*bool Find( const T& Element ); /\* ... \*/ };*

Статические переменные не исчезают, когда содержащая их функция закончит свою работу, компилятор хранит их значения от одного вызова функции до другого.

Инициализации статического поля выполняется вне класса программист должен использовать следующим образом:

*template <class T1> T1 имя\_класса <T1>::название\_поля=число;*

*В шаблонных функциях не допускается неявного преобразования типов*

Если объекты лежат в одном шаблонном классе, но вызваны разными типами, то они условно *лежат в разных плоскостях.*

42. Шаблоны функций: назначение, синтаксис и семантика, объявление, определение, вызов и перегрузка шаблонных функций, аргументы шаблона функции.