1. Основная идея ООП — использование вместо чёткого алгоритма одной цельной программы или разбиение её на насколько повторяющихся составляющих, взаимодействие некоторых объектов. Объекты — это экземпляры классов, абстрактных типов данных. Позволяют моделировать поведение реальных объектов и тот, кто решает задачу уже может действовать в терминах этой задачи, а не в терминах программирования.
2. Вся сущность ООП как раз в моделировании поведения объектов реального мира. Позволяет решающему задачу использовать термины этой задачи, а не программирования, что сильно облегчает задачу.

АТД представляют собой абстракции данных, которые определяются не через конкретные реализации, а через операции, которые можно выполнять с этими данными. Например, стек и очередь - это абстрактные типы данных. В ООП, классы часто используются для создания экземпляров, которые представляют собой реализацию АТД.

Каждый класс имеет свои данные и методы для их обработки. Эти поля можно унаследовать, то есть можно создать класс, который будет точно таким же, как другой, но с некоторыми добавлениями. Это производный АТД или производный класс.

1. Концепции. Моделирование поведения реального мира, возможность создания своих типов данных, инкапсуляция (данные закрыты, методы открыты), повышение степени повторного использования кода благодаря наследованию и шаблонам, полиморфная обработка родственных объектов.   
   Инкапсуляция — это принцип ООП, по которому каждый класс имеет закрытую часть с данными (свойствами) и открытую часть с методами их обработки (поведения). Открытую может использовать любая программа, закрытую только сам объект.

Преимущества:

* Сокрытие деталей: Инкапсуляция позволяет скрыть внутренние детали реализации, обеспечивая абстракцию и упрощая использование объекта.
* Безопасность: Защищает данные объекта, предоставляя контролируемый доступ через методы.
* Модульность: Позволяет изменять внутреннюю реализацию без воздействия на внешний код.

Определение: Расширяемость типов позволяет использовать один и тот же интерфейс для работы с различными типами данных или объектами. Это может быть достигнуто через перегрузку операторов, наследование и интерфейсы.

Преимущества:

* Универсальность: Позволяет обрабатывать различные типы данных с использованием общего интерфейса.
* Гибкость: Позволяет легко добавлять новые типы данных, не изменяя существующий код.
* Читаемость кода: Улучшает читаемость и понимание кода, так как один и тот же интерфейс может использоваться для различных объектов.

1. Концепции. Моделирование поведения реального мира, возможность создания своих типов данных, инкапсуляция (данные закрыты, методы открыты), повышение степени повторного использования кода благодаря наследованию и шаблонам, полиморфная обработка родственных объектов.   
   Названия классов начинаются с большой буквы, названия методов с маленькой. В данных может храниться любая информация про этот объект. В том числе и его состояние, например положение в пространстве или температура, которая может часто меняться. Для этого используются методы, при помощи которых объект может менять свои свойства. Методы также могут вызывать другие методы других объектов, поэтому получается, что один объект действует на другой. Данные могут быть представлены в виде любых типов данных. Переменные, константы, структуры (в том числе динамические).

* Конструкторы — методы, которые создают объект и приводят его в состояние, в котором с ним можно работать.
* Деструкторы удаляют объект и очищают память, которую он использовал.
* Селекторы предоставляют доступ к значениям закрытых полей объекта. Они обеспечивают чтение данных объекта извне.
* Модификаторы предназначены для изменения значений закрытых полей объекта извне. Они обеспечивают запись данных в объект.

1. Концепции. Моделирование поведения реального мира, возможность создания своих типов данных, инкапсуляция (данные закрыты, методы открыты), повышение степени повторного использования кода благодаря наследованию и шаблонам, полиморфная обработка родственных объектов.

Класс – это абстрактный тип данных, определяемый пользователем, представляющий собой модель объекта в виде данных и функций для работы с ним.

Семантика:

Класс - это абстракция, описывающая общие характеристики и поведение группы объектов.

Объекты - это конкретные экземпляры класса, созданные на основе его описания. Когда создается объект, он занимает место в памяти и получает доступ к методам и данным, определенным в классе.

Создание объекта:

* Объекты создаются с использованием конструкторов класса.
* В результате создания объекта выделяется память для его хранения.

Инициализация объекта:

* Инициализация объекта часто включает в себя установку начальных значений его полей.
* Может выполняться в конструкторе или с использованием специальных методов инициализации.

Обработка объектов:

* Обработка объектов включает в себя вызов методов класса, изменение состояния объекта и выполнение операций, определенных в классе.

Разрушение объекта:

* Деструкторы (в языках, где они поддерживаются) или сборка мусора отвечают за освобождение памяти, занятой объектом, когда объект больше не нужен.

1. Конструкторы и деструкторы - это методы, предназначенные для определенных операций во время создания и разрушения объектов.

**Конструкторы**

Основное назначение:

Инициализация объекта:

* Конструкторы предназначены для инициализации объекта, то есть установки начальных значений его полей и выполнения других необходимых операций при создании объекта.

Характерные черты:

* Имя совпадает с именем класса: Обычно конструктор имеет то же имя, что и класс, и вызывается автоматически при создании нового объекта.
* Может принимать параметры: Конструктор может принимать параметры для передачи начальных значений.

**Деструкторы**

Основное назначение:

* Освобождение ресурсов: Деструкторы выполняют операции, необходимые перед тем, как объект будет уничтожен, такие как освобождение занимаемой им памяти или других ресурсов.

Характерные черты:

* Имя с префиксом "~": В некоторых языках (например, в C++), деструктор имеет префикс "~" перед именем класса.
* Вызывается перед удалением объекта: Деструктор вызывается автоматически перед тем, как объект будет удален.

1. Перегрузка операторов и функций в C++ позволяет определить свою собственную логику для операций с объектами пользовательских классов. Ключевые слова friend и operator используются в контексте перегрузки операторов.

* Ключевое слово friend используется, чтобы получить доступ к приватным членам класса.
* Ключевое слово operator используется в комплекте с каким-нибудь оператором вместо имени функции.

// Перегрузка оператора сложения

Complex operator+(const Complex & other) const {

return Complex(real + other.real, imag + other.imag);

}

// Перегрузка оператора вывода

friend std::ostream& operator<<(std::ostream& os, const Complex& obj) {

os << obj.real << " + " << obj.imag << "i";

return os;

}

1. **Шаблоны и обобщенное программирование** в C++ предоставляют механизм для написания универсального кода, который может работать с различными типами данных, а также с контейнерами.

Шаблоны:

Определение:

* Шаблоны позволяют написать универсальные функции и классы, которые могут работать с разными типами данных, без необходимости создания разных версий кода.

Ключевое слово template:

* Ключевое слово template используется для объявления шаблона. Далее следует параметризованный код с типами данных, которые будут заменены при использовании шаблона.

template <typename T>

T add(T a, T b) {

return a + b;

}

int main() {

int sum\_int = add(3, 5);

double sum\_double = add(2.5, 4.3);

return 0;

}

**Обобщенные контейнеры:**

1. **Определение:**
   * Обобщенные контейнеры предоставляют универсальные структуры данных, которые могут хранить объекты любого типа.
2. **Пример контейнерного класса - шаблонный вектор:**

template <typename T>

class Vector {

private:

T\* array;

size\_t size;

public:

Vector(size\_t s) : size(s), array(new T[s]) {}

T& operator[](size\_t index) {

return array[index];

}

~Vector() {

delete[] array;

}

};

int main() {

Vector<int> intVector(5);

intVector[0] = 1;

intVector[1] = 2;

Vector<double> doubleVector(3);

doubleVector[0] = 2.5;

doubleVector[1] = 4.2;

return 0;

}

1. Стандартная библиотека шаблонов (STL) в C++ предоставляет набор готовых компонентов, которые упрощают создание эффективных и универсальных программ. Основные составляющие STL включают в себя контейнеры, итераторы и алгоритмы.

**Контейнеры:**

* **Определение:** Контейнеры представляют собой структуры данных для хранения и управления коллекциями объектов.
* **Примеры контейнеров:**
  + **std::vector:** Динамический массив.
  + **std::list:** Связанный список.
  + **std::set:** Множество уникальных элементов.
  + **std::map:** Отображение ключ-значение.
  + **std::queue:** Очередь.
  + **std::stack:** Стек.

**Итераторы:**

* **Определение:** Итераторы предоставляют универсальный интерфейс для доступа к элементам коллекции без знания ее внутренней структуры.
* **Пример итератора**: Vector.begin(), Vector.end()  
    
  **Алгоритмы:**
* **Определение:** Алгоритмы представляют собой функции, работающие с итераторами и предназначенные для выполнения различных операций на коллекциях.
* **Примеры алгоритмов:**

**std::sort:** Сортировка коллекции.

**std::find:** Поиск элемента в коллекции.

**std::transform:** Применение операции к каждому элементу коллекции.

1. Наследование в объектно-ориентированном программировании (ООП) позволяет создавать новый класс, используя уже существующий класс. В этом контексте выделяют базовый (родительский) и производный (дочерний) классы, а также формируется иерархия классов.

**Базовый (родительский) класс:**

**Определение:**

* + Базовый класс - это класс, который предоставляет общие атрибуты и методы, которые будут унаследованы дочерними классами.
  + Он служит основой для создания новых классов.

**Производный (дочерний) класс:**

**Определение:**

* Производный класс - это класс, который наследует атрибуты и методы от базового класса и может добавлять свои собственные.
* Производные классы обогащают или изменяют функциональность базового класса.

**Иерархия классов** - это структура, в которой классы организованы в виде дерева, где каждый класс, за исключением вершины, имеет один родительский класс и ноль или более дочерних классов.

1. **Полиморфизм** в объектно-ориентированном программировании (ООП) позволяет использовать общие интерфейсы для работы с объектами различных классов. Одной из форм полиморфизма является **полиморфизм через** использование абстрактных базовых классов.

**Абстрактный базисный класс** — это класс, объекты которого не создаются, это просто базис, который объединяет в себе характеристики, которые есть в целой куче других классов, но не имеют смысла сами по себе.

1. Обработка исключительных ситуаций (или исключений) и проверка утверждений - это важные аспекты программирования, направленные на обеспечение корректной работы программы и предотвращение непредвиденных сбоев. Вот несколько общих принципов и методов для этих задач:

**Обработка исключений:**

Использование блоков try-catch (попробовать-поймать):

Обертывайте код, который может вызвать исключение, в блок try.

Используйте блок catch, чтобы обработать исключение и предпринять соответствующие действия.  
**Проверка утверждений:**

**Использование оператора assert:**

* + Оператор **assert** позволяет вставлять проверки в коде для выявления неверных предположений.

**Применение условных проверок:**

* Используйте конструкции if и else для проверки условий и предпринятия соответствующих действий.

1. Преимущества и недостатки ООП:

**Преимущества:**

Модульность:

Программы, построенные с использованием ООП, могут быть легко структурированы в виде независимых модулей (классов).

Повторное использование кода:

Классы и объекты позволяют создавать повторно используемые компоненты, что уменьшает дублирование кода.

Разработка более понятного кода:

ООП способствует созданию более читаемого и понятного кода за счет абстракций и ясного описания отношений между объектами.

Упрощение сопровождения и тестирования:

Изоляция функциональности в объектах упрощает тестирование и обеспечивает легкость сопровождения.

Инкапсуляция:

Сокрытие внутренних деталей реализации в классах обеспечивает защиту данных и реализации от внешних изменений.

**Недостатки:**

Сложность:

Проектирование в стиле ООП может быть сложным, особенно для начинающих разработчиков.

Перфоманс:

Некоторые реализации ООП могут быть менее эффективными с точки зрения производительности из-за дополнительных слоев абстракции.

Интерфейс и абстракции:

Некорректное проектирование абстракций и интерфейсов может привести к усложнению и запутанности кода.

**Основные свойства языка ООП:**

Инкапсуляция:

Способность объединять данные и методы, которые с ними работают, в единый объект (класс), обеспечивая тем самым сокрытие деталей реализации.

Наследование:

Возможность создания новых классов на основе существующих, что способствует повторному использованию кода и уменьшению дублирования.

Полиморфизм:

Способность объектов одного типа использовать методы, определенные для объектов другого типа, что обеспечивает гибкость и расширяемость кода.

Место языка C++ в ООП:

C++ является многопарадигменным языком программирования, который поддерживает как процедурное программирование, так и объектно-ориентированное программирование. Он расширяет язык C, добавляя поддержку классов и объектов, что делает его мощным инструментом для разработки в стиле ООП.

Значение C++:

Производительность:

C++ предоставляет возможность низкоуровневого управления памятью и близкое к металлу программирование, что позволяет достичь высокой производительности.

Расширяемость:

Возможность использовать как процедурное, так и объектно-ориентированное программирование делает C++ универсальным для различных типов проектов.

Богатые библиотеки:

C++ имеет обширные стандартные библиотеки, что облегчает разработку и ускоряет процесс.

Наследование от C:

Совместимость с языком C позволяет использовать существующий код, написанный на C, в проектах на C++.

Инструменты разработки:

C++ имеет широкую поддержку различных инструментов разработки, включая отладчики и профилировщики.

**Значение C++ для профессиональной подготовки программистов:**

Обучение основам ООП:

Изучение C++ предоставляет хорошую возможность освоить основы объектно-ориентированного программирования.

Развитие навыков проектирования:

Программисты, обученные на C++, могут развивать навыки проектирования с использованием принципов ООП.

Переход между парадигмами:

C++ позволяет программистам работать как в процедурном, так и в объектно-ориентированном стиле, что полезно в различных контекстах.

Углубленное изучение языка:

Профессиональное использование C++ может включать более глубокое изучение тем, таких как шаблоны (templates) и продвинутые концепции ООП.

1. **Классы** — это пользовательские типы данных, которые кроме определённой информации (символ, число или указатель) имеют ещё и встроенные в них методы обработки этих данных.   
   **Компонентные данные** — это данные, которые являются частью того или иного класса. По принципу инкапсуляции должны быть доступны только компонентным функциям.

**Компонентные функции** — это такие функции, которые привязаны к тому или иному классу. Как правило используются для обработки данных в объекте, но ничто не мешает создать класс без данных и затолкать туда ворох функций, которые будут просто вызываться из класса, как будто это структурное программирование.

Доступ бывает общий (public) или частный (private). Всё, что в общем доступе может быть использовано любой внешней функцией, в том числе main. То, что в частном доступе может быть использовано только компонентными функциями, в том числе общего доступа.

При определении сначала идёт ключевое слово класс, потом имя класса и в фигурных скобках поля private со свойствами и public для методов их обработки. Объявление класса происходит в файле-заголовке, если он не шаблонный, то определение в файле исходного кода, то есть позже. Инициализация уже напрямую с объектом проводится.

Объекты, объявленные как константы могут использовать только функции-селекторы.

1. Область видимости класса начинается и заканчивается там, где подключены в виде заголовочных файлов.

Оператор двойного двоеточия позволяет пользоваться полиморфизмом, когда у нескольких классов могут быть одинаковые по лексике методы и отличить какой будет использоваться, или при определении компонентной функции. Например, унарная форма позволяет пользоваться глобальной переменной i даже если используется другая локальная переменная с тем же именем. Бинарная позволяет компилятору определить, к какому классу принадлежит та или иная функция, если их несколько в области видимости с одинаковым названием. Постоянно используется для определения функций вне заголовочного файла. Вложенные классы — это классы, которые объявлены внутри другого класса и могут быть использованы только им. То есть, используется в поле свойств. Часто используется в очень крупных классах, когда некоторые его части могут быть востребованы очень редко и эффективнее просто разбить класс на несколько составляющих.

1. Компонентными данным может быть фактические любые данные, любого типа, встроенного или пользовательского.

Используются для описания класса, его свойств и отличительных черт, которые нужны для решения задачи.

Объявляются внутри блока private, то есть не могут быть изменены ничем, кроме собственных компонентных функций.

Статические данные (static) — это такие данные, которые могут быть использованы любым объектом класса и для всех объектов они одинаковы, так как привязаны только к самому классу. Кстати, к ним имеется доступ даже до создания хотя бы одного объекта, прямо из класса.   
Константные статические данные ровно то же самое, только их нельзя менять. Удобно для экономии памяти, если нужна какая-то константа для вычислений, лучше вынести именно туда, если объектов создать много, то получится небольшая экономия памяти.   
Изменяемые компонентные данные (mutable) можно изменять, при этом даже если объект объявлен как константа.

1. Компонентные функции используются для симуляции реального поведения объектов. В теории. На самом деле используются для изменения данных тем или иным способом, обработки и вывода этих данных, а также упрощения использования класса программистом. Функции бывают селекторами и модификаторами:

* Селекторы предоставляют доступ к значениям закрытых полей объекта. Они обеспечивают чтение данных объекта извне.
* Модификаторы предназначены для изменения значений закрытых полей объекта извне. Они обеспечивают запись данных в объект.

1. Как правило, практически все функции определяются вне тела класса, чтобы сэкономить дополнительные строки. Иногда функции могут быть определены прямо там, если функция короткая, это чуть быстрее писать. Однако, удобство в последующем использовании на стороне определяемых вне класса функций. Потому что при определении функции можно использовать простые названия для входных данных, которые удобны для именно этой маленькой задачи. А во время определения использовать длинные, но понятные, по которым легко понять, что именно запрашивается на вход такой функции.

Синематика у встроенных простая настолько, насколько возможно. Просто определение функции в поле public как будто она глобальная.

У внешнего определения нужно отдельно прописать область видимости класса, чтобы эта функция была видна только классу и/или его объектам.

1. **Указатель this:**

В C++ ключевое слово **this** используется внутри методов класса для обозначения указателя на текущий экземпляр объекта. Он указывает на объект, для которого вызывается метод. Это позволяет обращаться к членам объекта внутри его методов.

**Компонентные функции типа static:**

Методы класса могут быть статическими, что означает, что они принадлежат классу, а не конкретному объекту. Они вызываются от имени класса, а не от экземпляра объекта.

* Статические методы удобны, когда вы хотите иметь функцию, принадлежащую классу, а не конкретному объекту.
* Они не могут обращаться к нестатическим членам класса, потому что у них нет доступа к конкретному экземпляру объекта.

**Компонентные функции типа const:**

Методы класса могут быть объявлены как константные, что означает, что они не изменяют состояние объекта (кроме изменения **mutable** членов).

* Константные методы полезны, когда вы хотите предоставить возможность использовать объекты в контексте, где не должны быть изменены (можно использовать для селекторов чтобы они точно не могли изменить поля).

1. **Конструкторы в C++:**

**Назначение:** Конструкторы в C++ - это специальные методы класса, которые вызываются при создании объекта. Их основное назначение - инициализировать объект валидными значениями или выполнять другие необходимые действия в начале его жизни.

Пример конструктора (конструктор объявляется как метод и имеет имя как класс):

class MyClass {

public:

// Конструктор

MyClass() {

// Инициализация объекта

}

};  
**Наличие нескольких конструкторов у одного класса:**

**Причины:**

1. **Различные способы инициализации:**
   * Разные конструкторы могут предоставлять различные варианты инициализации объекта, чтобы обеспечить удобство для пользователей класса.
2. **Перегрузка конструкторов:**
   * Возможность перегружать конструкторы позволяет классу иметь различные варианты создания объекта в зависимости от переданных параметров.

**Способы задания конструктора: по умолчанию или явное:**

**Конструктор по умолчанию**:

* Конструктор, который не принимает аргументов. Если вы не предоставляете свой конструктор, компилятор может создать конструктор по умолчанию автоматически.

**Явные конструкторы:**

* Конструкторы, которые принимают параметры. Эти конструкторы используются для инициализации объекта значениями, переданными при создании.

**Назначение копирующий конструктор:**

* Копирующий конструктор в C++ предназначен для создания копии объекта класса.

Пример:

// По умолчанию

Point() {

x = 0;

y = 0;

}

// Явный

Point(int newX, int newY) {

x = newX;

y = newY;

}

// Копирующий конструктор

MyClass(const MyClass &other) {

// Тело копирующего конструктора

}

1. **Явное определение конструктора:**

**Определение в теле класса:**

class MyClass {

public:

// Явное определение конструктора в теле класса

MyClass(int value) {

// Тело конструктора

}

};

Такой конструктор компилируется в каждую единицу трансляции, в которой используется.

**Определение вне своего класса:**

class MyClass {

public:

// Явное объявление конструктора в теле класса

MyClass(int value);

};

// Явное определение конструктора вне своего класса

MyClass::MyClass(int value) {

// Тело конструктора

}

Когда конструктор объявлен в теле класса, а его определение размещено вне класса, это позволяет вынести тело конструктора из тела класса и разместить его в отдельном файле исходного кода (.cpp).

Конструкторы можно использовать только как конструкторы и нельзя вызывать нигде, кроме как в конструкторе. И даже в другом конструкторе вызывается конструктор не как обычная компонентная функция, его нужно вызвать до фигурных скобок определения конструктора. Иначе по завершению этого конструктора выйдет из области видимости и созданный объект, что приведёт к вызову деструктора. Однако, конструировать можно не только по значению или копию объекта этого же класса, но и на основе объекта другого класса, таким образом выполняя преобразование типа данных.

1. Конструктор - это специальный метод класса, который вызывается при создании нового объекта этого класса. Конструкторы могут выполнять различные задачи, но их основная цель - устанавливать начальные значения для переменных объекта.

Инициализаторы конструктора предоставляют удобный способ инициализировать переменные класса при их объявлении. Они добавляются к синтаксису объявления переменных.

MyClass(int initial\_value) : my\_variable(initial\_value) {

// Дополнительные действия, если нужно

}

Преобразование типов в C++ - это процесс изменения типа данных переменной из одного типа в другой. Конструкторы могут быть использованы для создания пользовательских преобразований типов. Ключевое слово **explicit** применяется к конструкторам, чтобы указать, что преобразование должно происходить явно, и не должно автоматически вызываться компилятором.

1. Конструкторы по умолчанию — это конструкторы, которые ничего не принимают на вход. Компилятор сам выдаст такой, если его не сделает программист и тогда такой конструктор просто выполнит весь код (кроме объявления функций) из тела самого класса. Объявит нужные переменные, проинициализирует те, которые были проинициализированы и будет готовый объект. Если в классе нет динамического распределения памяти, то такой способ имеет место быть и даже удобен. Если же его описывать, то нужно сначала объявить конструктор в теле цикла, потом (рекомендуется) этот самый конструктор определить в исполняемом файле. У конструкторов нет возвращаемого типа данных, даже void, он просто работает и всё. Также, всегда необходимо при определении показать, часть какого класса этот конструктор, даже несмотря на одинаковое имя функции и класса, компилятору этого мало.
2. Конструктор копирования в C++ предназначен для создания нового объекта, который является копией существующего объекта. Это происходит при инициализации нового объекта с использованием уже существующего объекта того же типа. Конструктор копирования выполняется автоматически в различных ситуациях, например, при передаче объекта в функцию по значению, при возвращении объекта из функции и в других случаях.

class MyClass {

public:

// Пользовательский конструктор копирования

MyClass(const MyClass& other) {

// Логика копирования, если необходимо

}

};

1. Конструктор вызывается всегда, когда нужно создать новый объект. В зависимости от переданных параметров выбирается вид конструктора. Если нет параметров, используется конструктор по умолчанию. Конструктор можно вызвать внутри другого конструктора, для этого нужно сразу перед фигурными скобками в определении нового конструктора через двоеточие вызвать необходимый конструктор. Например, часто используется в копирующем. Явный вызов конструктора — это вызов со скобочками, неявный это просто определение новой переменной (вернее объекта) типа класса.
2. Деструктор — это компонентная функция, обратная конструктору. Он используется для очистки выделенной динамически памяти. Также, после срабатывания деструктора объект выходит из области видимости программы и больше не может быть использован. Деструктор имеет точно такое же имя, как и класс, объект которого он удаляет, но с символом ~ в начале. На вход не принимает никаких значений. В конце любой функции, где создавались объекты, вызываются деструкторы для всех этих объектов, но можно и явно его задать, чтобы очистить память заранее.
3. Описание деструктора может быть как явным, так и по умолчанию, если явного не задано. Как правило, описывается деструктор в теле класса, а определяется уже в исполняемом файле. Разница ровно такая же, как и для любой другой компонентной функции. Иногда ускоряет работу, но создаёт дополнительные проблемы при компиляции многих проектов с использованием этого класса. У деструктора, как и у любой другой компонентной функции имеется всегда как минимум один параметр, указатель на используемый объект this. Но, в нём можно делать и другие параметры, если это необходимо. К примеру, если есть два и более способа удаления объекта.
4. Вызывать деструктор можно явно или он будет вызвать автоматически после выхода из области видимости функции, где был создан. В целом, можно спокойно создавать программы, явно не вызывая деструкторов. Однако, если объекты создаются в крупной функции и их там много это может привести к очень большому расходу памяти. Да, она освободится по завершению работы функции, но освобождать её можно и заранее.
5. Специальный полиморфизм — это полиморфизм, то есть называние, одним и тем же словом, разных вещей, который используется в специальных случаях. Его можно (и нужно) использовать для преобразования АТД, то есть пользовательского класса к встроенному типу. Конечно, это не всегда имеет смысл, но может пригодиться. Например, если АТД это класс столов, а для задачи нужно посчитать кроме всего прочего их общий вес, то конструкцию int + table \* 5 вполне нормально перегрузить, чтобы из класса table бралось определённое поле и использовалось как целое число.
6. Специальный полиморфизм — это полиморфизм (придача одной команде или оператору разных значений) который используется в специальных ситуациях. Перегружаются очень часто именно функции. Выбирается один их вариантов перегрузки функции по очень простому алгоритму. Поиск совпадений по списку входных параметров.
7. Дружественные функции:

**Основные свойства:**

Объявление внутри класса: Дружественные функции объявляются внутри класса с использованием ключевого слова friend, но они не являются членами класса.

Доступ к закрытым членам: Дружественные функции имеют доступ к закрытым (private) членам класса, что делает их особенно полезными в ситуациях, когда нужно разрешить внешним функциям работать с закрытыми данными класса.

**Номенклатура:**

Дружественные функции объявляются с использованием ключевого слова friend внутри класса.

Они могут быть обычными функциями или методами других классов.

**Необходимость:**

Дружественные функции необходимы, когда вам нужно предоставить внешней функции (которая не является членом класса) доступ к закрытым членам класса.

**Полезность:**

Позволяют ограничивать доступ к закрытым членам, сохраняя контроль над инкапсуляцией.

Полезны при перегрузке операторов, когда необходим доступ к закрытым членам двух классов.

**Опасность применения:**

Влияют на инкапсуляцию, поэтому должны использоваться осторожно.

Могут нарушать принципы объектно-ориентированного программирования, такие как инкапсуляция и скрытность данных.

**Наборы дружественных функций:**

Класс может иметь несколько дружественных функций.

Дружественные функции могут быть свободными функциями или методами других классов.

**Дружественные классы:**

Класс может быть объявлен дружественным другому классу, что дает доступ к закрытым членам обоих классов.

**Дружественные функции:**

**Объявление (размещение, синтаксис, семантика, требования):**

* **Размещение:** Дружественные функции объявляются внутри класса с использованием ключевого слова **friend**.
* **Синтаксис:** **friend <тип\_возвращаемого\_значения> <имя\_функции>(<параметры>);**
* **Семантика:** Дружественные функции являются независимыми от экземпляров класса, но имеют доступ к закрытым членам класса.
* **Требования к параметрам и возвращаемому значению:** Дружественные функции могут иметь параметры и возвращаемое значение любого типа, в том числе и связанные с классом.

**Определение (размещение, область видимости):**

* **Размещение:** Дружественные функции обычно определяются вне класса.
* **Область видимости:** Определения дружественных функций находятся в глобальной области видимости и не являются членами класса.

**Вызов функции:**

* Дружественные функции могут быть вызваны так же, как и другие функции, без создания экземпляра объекта класса.
* Вызов осуществляется по имени функции.

1. **Специальный полиморфизм – перегрузка операторов:**

**1. Синтаксис:**

* Перегрузка операторов выполняется с использованием ключевого слова **operator**, за которым идет символ или символьная комбинация, представляющая перегружаемый оператор.
* Перегружаемая функция оператора может быть объявлена как член класса или как свободная функция.

**2. Семантика:**

* Семантика перегруженных операторов зависит от контекста и логики, определенной в перегруженной функции оператора.
* Перегруженные операторы могут использоваться с объектами класса так же, как встроенные операторы.

**3. Приоритет и порядок выполнения:**

* Приоритет и порядок выполнения перегруженных операторов сохраняются такими же, как у встроенных операторов.

**4. Доступные и недоступные для перегрузки операторы:**

* Некоторые операторы нельзя перегружать (например, **::**, **.\***, **sizeof**).
* Некоторые операторы можно перегружать только как члены класса (например, **=**, **()**, **[]**).
* Некоторые операторы можно перегружать как члены класса и как свободные функции (например, **+**, **-**, **\***).

**5. Перегружаемые как обычные функции:**

* Некоторые операторы можно перегружать как обычные функции, то есть вне класса.

1. Перегружать можно как унарные, так и бинарные операторы. Если это компонентные функции, то унарные операторы будут без явно заданных входных данных, а дружественные или просто внешние функции будут иметь один. Также бинарные операторы в компонентном виде будут иметь всего один, второй операнд. Ну а дружественным или внешним нужно передать оба.

**Синтаксис перегрузки оператора присваивания (=):**

* Оператор присваивания обычно перегружается как член класса и должен возвращать ссылку на объект класса.  
  Пример:

class MyClass {

public:

MyClass& operator=(const MyClass& other);

};

Перегруженный оператор присваивания принимает в качестве аргумента объект того же типа (**const MyClass& other**).

**Семантика перегрузки оператора присваивания:**

* Логика перегруженного оператора присваивания определяет, каким образом значения и состояния объекта, для которого вызывается оператор, копируются из другого объекта.

Пример:

MyString& operator=(const MyString& other) {

if (this != &other) {

// Логика копирования данных

}

return \*this;

}

**Синтаксис перегрузки оператора индексирования ([]):**

* Оператор индексирования обычно перегружается как член класса и должен возвращать ссылку на элемент, который будет использоваться для чтения или записи.  
  Пример:

int& operator[](int index);

**Семантика перегрузки оператора индексирования:**

* Логика перегруженного оператора индексирования определяет, как элементы доступны и изменяются через использование квадратных скобок.

Пример:

class MyArray {

private:

int\* elements;

int size;

public:

// Конструктор

MyArray(int s) : size(s) {

elements = new int[size];

// Логика инициализации

}

// Деструктор

~MyArray() {

delete[] elements;

}

// Перегрузка оператора индексирования

int& operator[](int index) {

// Логика доступа к элементу

return elements[index];

}

};

1. **Перегрузка операторов ввода-вывода (<<, >>):**

**Синтаксис перегрузки оператора вывода (<<):**

* Оператор вывода обычно перегружается как свободная функция, принимающая поток вывода (**std::ostream&**) и объект для вывода.

class MyClass {

friend std::ostream& operator<<(std::ostream& os, const MyClass& obj);

};

**Семантика перегрузки оператора вывода:**

* Логика перегруженного оператора вывода определяет, как объект класса представляется в потоке вывода.

class Point {

private:

int x, y;

public:

Point(int xVal, int yVal) : x(xVal), y(yVal) {}

// Перегрузка оператора вывода

friend std::ostream& operator<<(std::ostream& os, const Point& obj) {

os << "(" << obj.x << ", " << obj.y << ")";

return os;

}

};

**Синтаксис перегрузки оператора ввода (>>):**

* Оператор ввода обычно перегружается как свободная функция, принимающая поток ввода (**std::istream&**) и объект для ввода.

class MyClass {

friend std::istream& operator>>(std::istream& is, MyClass& obj);

};

**Семантика перегрузки оператора ввода:**

* Логика перегруженного оператора ввода определяет, как объект класса инициализируется значениями из потока ввода.

class Point {

private:

int x, y;

public:

// Перегрузка оператора ввода

friend std::istream& operator>>(std::istream& is, Point& obj) {

is >> obj.x >> obj.y;

return is;

}

};

**Использование перегруженных операторов:**

* Перегруженные операторы могут использоваться так же, как и стандартные операторы ввода-вывода.

1. Перегрузка оператора () для индексирования в языке программирования C++ позволяет создавать объекты, которые могут быть использованы с синтаксисом, аналогичным использованию массивов или контейнеров. Это позволяет объектам класса быть "вызываемыми", как если бы они были функциями. Давайте рассмотрим синтаксис и семантику перегрузки оператора () для индексирования.

class ArrayWrapper {

private:

int array[5] = {1, 2, 3, 4, 5};

public:

// Перегрузка оператора () для индексирования

int operator()(int index) {

if (index >= 0 && index < 5) {

return array[index];

} else {

std::cout << "Индекс вне диапазона!" << std::endl;

return -1; // или какое-то значение по умолчанию

}

}

};

int main() {

ArrayWrapper myArray;

// Использование объекта с перегруженным оператором ()

std::cout << "Элемент с индексом 2: " << myArray(2) << std::endl;

std::cout << "Элемент с индексом 7: " << myArray(7) << std::endl;

return 0;

}

**Семантика:**

* Возвращаемый тип: Может быть любым типом, включая пользовательские типы данных.
* Параметры индексирования: Определяются пользователем и зависят от того, какой функциональности вы хотите достичь с вашим классом.

Реализация оператора: Оператор () реализуется внутри класса и выполняет логику индексирования. В приведенном выше примере это возвращает элемент массива по заданному индексу.

1. В C++, перегрузка операторов доступа к элементу класса через указатель на объект (→→) позволяет определить пользовательское поведение для операции доступа к членам класса через указатель. Оператор →→ используется для доступа к членам объекта, на который указывает указатель.

class Complex {

private:

double real;

double imag;

public:

Complex(double r, double i) : real(r), imag(i) {}

// Перегрузка оператора -> для доступа к членам через указатель

Complex\* operator->() {

return this;

}

void display() {

std::cout << "Real: " << real << ", Imaginary: " << imag << std::endl;

}

};

int main() {

Complex obj(3.0, 4.0);

Complex\* pObj = &obj;

// Использование оператора -> для доступа к членам через указатель

pObj->display();

**Семантика:**

1. **Возвращаемый тип:** Обычно это указатель на объект класса или что-то, что может быть использовано как указатель.
2. **Реализация оператора ->:** Оператор →→ реализуется внутри класса и возвращает указатель на сам объект (или другой объект, у которого определены соответствующие операции →→).
3. **Использование в коде:** Когда объект класса используется через указатель, оператор →→ вызывается автоматически для доступа к членам класса через указатель.

39. Позволяет делать приколы с конструкторами и деструкторами. В целом, как правило, перегрузка оператора new или delete позволяет автоматически добавлять пару (или много) строк кода, которые будут вызываться перед тем, как вызовется нормальный, обычный delete. Перегружать можно, само собой все их виды, хоть простые new и delete хоть с []. В остальном, оператор ведёт себя точно также как остальные операторы. В параметры передавать размер удаляемого объекта.

40. Шаблоны классов и функций задаются при помощи ключевых слов template <T>, где T теперь будет показывать тот тип данных, который может использоваться. По умолчанию в этот Т можно затолкать абсолютно всё, что угодно, но если объект будет пытаться вызвать функцию какого-то класса, а на вход подадут тип данных, в котором такой функции нет… ошибка компиляции. В этом причина полиморфной обработки, очень хорошо помогает, когда геттеры, сеттеры и прочее одинаково пишется и можно смело толкать всё, что угодно в какой-нибудь шаблонный класс-контейнер.

41. Шаблоны классов

Шаблоны классов – еще один способ повысить степень повторного использования кода. Например, у нас есть класс – контейнер, который хранит целые числа, содержит методы, связанные с получением и записью этих чисел. Однако может произойти так, что может понадобиться этот же самый контейнер, но для другого типа данных (прим. Float, double, даже string или иного пользовательского). Если бы шаблонов не было, то была бы необходимость создавать классы под каждый из этих типов вручную. Шаблоны позволяют программисту написать один универсальный шаблонный класс под разные типы данных.

Например:

template <class T>

class container {

public:

T\* a = new T[10];

void set(T data, int i) {

a[i] = data;

};

T get(int i) {

return \*a[i];

};

void remove(int i) {

a[i] = nullptr;

}

void out(int i) {

cout « a[i];

}

};

При создании экземпляра этого класса его можно будет проинстанцировать под любой желаемый тип, после чего его можно будет использовать с соответствующими значениями:

container<int> c;

c.set(27,2);

c.out(2);

container<double>c2;

c2.set(1, 1);

c2.remove(1);

c2.out(1);

container<char> c3;

c3.set('a', 4);

c3.out(4);

42. Шаблоны функций

Идея шаблонных функций схожа с идеей шаблонных классов. Можно создать универсальную функцию, которая будет проводить обработку любых предусмотренных значений без необходимости её перегрузки для каждого отдельного типа.