Контрольные вопросы

10 октября 2022 г.

1 На чём основано объектно-ориентированное программирование?

Объектно-ориентированное программирование (ООП) основано на создании объектов, обладающих определённым набором каких-либо свойств (атрибутов) и возможных состояний. С одной стороны, объект является частью памяти для хранения значений, но при подходе ООП он также обладает свойствами и состояниями. Задача ООП и состоит в определении объектов. Объекты принадлежат классам. Классы являются основной структурной единицей ООП. Подход ООП облегчает написание и понимание кода, а также увеличивает возможность его повторного использования. Основополагающими концепциями ООП являются:

- инкапсуляция – процесс слияния приватных данных и публичных методов в единое целое (то есть к приватным данным возможно обращаться только с помощью общедоступного интерфейса, осуществляемого публичными методами); инкапсуляция позволяет использовать класс без знаний о его реализации, а потому предоставляет следующие преимущества:
  - защита данных,
  - более удобное изменение класса,
  - более удобная отладка;
- полиморфизм – способность представлять разные реализации (например, на основании различных базовых данных) одинаковым интерфейсом;
- наследование – создание новых объектов приобретением атрибутов и методов других объектов с их последующим расширением и специализацией (передача чего-либо от родителей потомкам);
- абстракция – процесс, при котором оказываются доступными актуальные данные и скрываются ненужные (инкапсуляция делает возможной реализацию данной концепции).

2 Какие аспекты следует учитывать при проектировании классов?

При проектировании классов следует учитывать следующие аспекты:

- Обычно классам даётся имя с заглавной буквы.
- Данные-члены и функции-члены класса следует располагать в следующем порядке (обусловленным тем, что пользователю далеко не всегда интересно внутреннее строение класса):
  - конструктор;
  - деструктор;
  - иные специальные функции-члены;
  - геттеры;
  - сеттеры;
  - данные-члены;
- Существуют приватная, публичная и защищённая секции класса:
  - по умолчанию данные-члены (атрибуты) и функции-члены (методы) класса являются приватными (private);
  - обычно данные-члена класса объявляются приватными, а функции-члены публичными (public), т.е. фактически стоит создавать инкапсулированные классы, поскольку с ними удобно работать;
  - для получения значения или изменения приватных данных членов следует создавать общедоступные функции доступа (access functions):
    - геттеры (getters) возвращают значения приватных атрибутов и должны предоставлять доступ к ним только для чтения (поэтому возвращение значения разумно осуществлять по значению или по константной ссылке);
    - сеттеры (setters) устанавливают значения приватных атрибутов;
  - защищённые (protected) члены класса могут использоваться только определённым образом (см. наследование).
- Конструктор класса – специальная функция, вызываемая при создании объекта класса и предназначенная для его инициализации:
  - конструктор определяет может ли быть создан объект данного класса;
  - конструктор может быть использован для инициализации объекта класса;
  - конструктор не имеет возвращаемого значения и должен иметь то же имя, что и класс;
  - если никакой конструктор не объявлен в коде, то C++ создаёт публичный конструктор по умолчанию, в противном случае – его нужно объявлять в коде (что лучше делать явно, ипользуя ключевое слово default);
  - иногда бывает достаточно ограничиться конструктором по умолчанию (но благодаря перегрузке функций в классе может быть объявлено несколько конструкторов).

- Допустимыми являются как инициализация атрибутов объекта в самом классе, так и с помощью конструктора класса при создании объекта:
  - инициализация атрибутов класса может происходить по значению (value-initialization) и по умолчанию (default-initialization), и в большинстве случаев они приведут к вызову конструктора (если он существует); однако инициализация по значению может перед вызовом конструктора присвоить значения атрибутам, которые затем будут изменены с помощью конструктора; этого недостатка лишена инициализация атрибутов по умолчанию, но при такой инициализации необходимо знать, что в классе инициализируются все атрибуты;
  - с помощью конструктора возможна прямая инициализация (direct-initialization) и инициализация списком (list-initialization) атрибутов объекта класса;
  - для инициализации атрибутов класса (а не присваивания им значения) используются списки инициализации конструкторов классов (они позволяют инициализировать типы данных, которые могут быть инициализированы только при объявлении – например, константы);
  - списки инициализаторов членов в конструкторах классов работают как с фундаментальными типами, так и с членами, которые сами являются классами;
  - списки инициализаторов членов позволяют инициализировать массивы (ненулевыми значениями начиная со стандарта C++11) и классы;
  - переменные из списка инициициализаторов инициализируются в порядке объявления в классе;
  - если в классе объявлено несколько конструкторов, то вместо указания в них значений по умолчанию (default value) для некоторых атрибутов (если значения одинаковы) может быть использована нестатическая инициализация членов (non-static member initialization) с помощью инициализации данных-членов копированием или фигурными скобками.

- При наличии конструкторов с одинаковыми инструкциями для избегания дублирования кода может быть использовано делегирование конструкторов (delegating constructors) (обычный вызов конструктора приводит к UB):
  - конструкторы не могут делегировать и инициализировать данные-члены одновременно;
  - конструкторы могут делегировать друг другу одновременно, что может привести к переполнению стека вызова;
  - в теле конструктора разрешён вызов функции, не являющейся конструктором, которая может быть использована вместо делегирования для выполнения повторяющихся в конструкторах инструкций (в этом случае конструктор сможет инициализировать данные-члены).

- Деструктор класса – специальная функция, вызываемая при уничтожении объекта класса и предназначенная для очистки памяти:
  - класс может иметь только один деструктор;
  - деструктор имеет то же имя, что и класс, которому предшествует символ $\sim$;
  - деструктор не может принимать аргументы;
  - деструктор не имеет возвращаемого типа;

  - для классов, в которых данные-члены только инициализируются, C++ предоставляет деструктор по умолчанию;
  - как правило, деструкторы не вызываются явно;
  - деструкторы могут безопасно вызывать другие функции-члены.

- Создание константных объектов класса:
  - функции-члены, не изменяющие данные-члены, должны быть объявлены как константные;
  - константные функции-члены могут быть вызваны неконстантными объектами;
  - две функции, отличающиеся константностью считаются разными (поэтому возможна перегрузка);
  - конструктор не может быть константным.

- Реализация функций-членов классов:
  - маленькие и специальные (конструктор, деструктор и т.д.) лучше реализовывать внутри класса;
  - большие лучше выносить за пределы класса или в отдельные cpp-файлы;
  - вывод типов auto следует использовать только для маленьких функций-членов.

- Инвариант класса (invariant) – связь, при которой на протяжении всей жизни объекта сохраняется некоторый набор условий или утверждений:
  - соблюдение условий инвариатности класса необходимо проверять самостоятельно, когда в классе происходят какие-либо изменения (обычно после изменений);
  - при нарушении инвариантности необходимо обнаруживать ошибку.

- При проектировании классов могут быть использованые статические данные-члены и статические функции-члены:
  - статические данные-члены и статические функции-члены объявляются с помощью ключевого слова static;
  - статические данные-члены не связаны с конкретным объектом класса: они характеризуют весь класс;
  - при инициализации статических данных-членов к ним всегда может быть получен доступ через объекты класса или через имя самого класса;
  - статический член должен быть определён, и это возможно сделать следующими способами:
    * инициализацией вне класса;
    * член, являющийся константным целочисленным или константным перечислением, может быть инициализирован внутри класса;
    * статические члены constexpr могут быть инициализированы внутри определения класса;
    * начиная с C++17, неконстантные члены могут быть инициализированы в определении класса при объявлении их встроенными (inline);
  - для доступа к статическим данным-членам следует создавать статические функции-члены;

  - статические функции-члены также не привязаны к какому-либо конкретному объекту класса, и доступ к ним может быть получен как через имя класса, так и через конкретный объект (последнее не рекомендуется);
  - C++ не поддерживает статические конструкторы (для инициализации статических данных-членов).

- Дружественная для данного класса функция (friend fucntion) – это функция, которая может обращаться к приватным (и защищённым) членам этого класса, как если бы она была членом данного класса:
  - дружественная функция может быть либо обычной функцией, либо функцией-членом другого класса;
  - функция может быть дружественной к нескольким классам одновременно;
  - чтобы сделать функцию-член дружественной, компилятор должен увидеть полное определение класса дружественной функции-члена (а не только предварительное объявление);
  - для дружественных функций также может быть использовано предварительное объявление.

- Весь класс может быть сделан другом другого класса, что дает всем членам дружественного класса доступ к приватным членам другого класса.

- Дружественные сущности объявляются с помощью ключевого слова friend.

- "Отношение дружественности"не является симметричным и транзитивным.

3 Почему удобно разделять классы на интерфейс и реализацию?

Разделение классов на интерфейс и реализацию позволяет пользователю/клиенту/программисту, не вникая в строение (т.е. реализацию) класса, использовать предоставляемые интерфейсом возможности (обычно использование функций-членов класса). Интерфейс – всё, что им необходимо для понимания прицепов работы с классом. Собственно говоря, разделение класса на интерфейс и реализацию и воплощает концепцию инкапсуляции, позволяющей:

- упростить использование классов и снизить сложность программы;
- защитить данные (поскольку доступ к осущетсвляется лишь с помощью функций-членов);
- вносить изменения в код более быстро;
- проводить отладку более быстро (поскольку если данные-члены имеют неправильные значения, то это можно будет отследить по реализации класса при вызове методов, изменяющих рассматриваемые атрибуты)

4 Чем внутреннее связывание отличается от внешнего связывания?

Связывание (linkage) – характеристика индетификатора, определяющая ссылается ли то или иное объявление имени сущности на ту же сущность или нет.

При внутреннем связывании (internal linkage) идентификатор может быть увиден и задействован только в данном файле: он недоступен из других файлов (то есть он не доступен компоновщику). Стоит отметить, что каждая внутренняя сущность имеет только одно определение (правило одного определения всегда выполняется), поскольку все внутренние сущности считаются независимыми. Не так по ошибке программиста может обстоять дело в случае внешнего связывания.

Идентификатор с внешней связью (external linkage) может быть увиден и использован, как в классе, в котором он определён, так и из других файлов кода с использованием предварительных объявлений (forward declaration). При этом предварительное объявление сообщает компилятору о существовании функции, а компоновщик связывает вызовы функций с фактическим определением функции.

5 Какими особенностями обладают именованные пространства имён?

Простанства имён (namescape) – это области, позволяющие объявлять внутри себя имена с целью устранения неоднозначности. Пространтство имён может быть глобальным, безымянным или именованным. Именованные пространства обладают следующими свойствами:

- пространства имён создают область видимости пространства имён (namescape scope);
- аддитивность, т.е. пространства имён могут быть разделены: несколько частей кода могут быть объявлены в одном и том же пространстве имён;
- вложенность, т.е. одно пространство имён может быть объявлено внутри другого пространства имён;
- классы и функции-члены классов могут быть определены как внутри пространства имён, так и вне его;
- ключевое слово using может быть использовано как директива для представления всего пространства имён;
- доступ к пространству имён может быть получен из других файлов;
- для пространств имён могут быть созданы и использованы псевдонимы пространств имён.

Литература

[1] https://www.learncpp.com/

[2] https://stackoverflow.com/questions/1031273/what-is-polymorphism-what-is-it-for-and-how-is-it-used

[3] https://stackoverflow.com/questions/16014290/simple-way-to-understand-encapsulation-and-abstraction

[4] https://cplusplus.com/doc/tutorial/classes2/

[5] https://www.geeksforgeeks.org/what-is-class-invariant/

[6] https://cplusplus.com/doc/tutorial/namespaces/