COS 2021 – Fundamental Data Structures – Spring 2024

Homework 4 (String Pattern Matching)

Documentation

**Task.cpp**

* #include <iostream> - calling exceptions and console output
* #include <string> - using all properties of the type string
* #include <vector> - using vectors
* #include “KMP.cpp” – allows us to use the KMP algorithm class in the class file “KMP.cpp”
* #include <algorithm> - using the sort function

using namespace std; - used for calling exceptions, checking whether there is a given GUID in the GUIDs vector, console output and to save time to not write everywhere ” std::” where it is required

enum DiscountType { amount, percentage }; - creating a enum for discount with two options – amount or percentage that can be called anywhere in the program.

vector<string> GUIDs; - used to check whether there is already a division with a specific GUID before creating a new one with the same GUID

class Division

* **private** members – the data slots
  + GUID (string)
  + Name (string)
  + PhoneNumber (string)
  + Description (long string)
  + Parent (Division)
  + void checkGUID(string guid) – checks whether there is already a division with a specific GUID in the vector GUIDs before creating this new one with the same GUID
    - If there is, it throws an invalid argument exception “There is already a division with this GUID.”.
    - If there is not, it proceeds to create the new division.

It is called in every constructor of the class Division.

* **public** members (using const )
  + Constructors (using the const keyword to prevent the new data from modification)
    - Standard (empty) constructor with default data for all data slots
      * GUID (string) – e.g. ABCD-EFGH-1234 through void checkGUID(“ABCD-EFGH-1234”)
      * Name (string) – e.g. Electronics
      * PhoneNumber (string) – e.g. 0888 123456
      * Description (long string) - empty
      * Parent (Division) – NULL pointer
    - Argument constructor – used to create a division with previously set data.
      * checkGUID(“”) is called here as well.
      * One of the arguments “Division\* parent” is by default = nullptr.
        + In case this constructor is used and the user

doesn’t define a parent, the division is said to not have any and gives the Parent value nullptr.

define a parent the Parent is given a value.

* + - Copy constructor – used to create a copy of another division in the currently generated new division
      * All of the data is being replicated, including the GUID.
  + Assignment operator= - it does the same operation just like the copy constructor

class Item – an abstract class

* **public** members
  + **virtual** double GetTotalPrice() const = 0;
  + **virtual** string GetName() const = 0;
  + **virtual** string GetGUID() const = 0;
    - using the **const** keyword to prevent data modification
    - virtual – allows us to get an access to Artifact’s GetName(), GetGUID() and GetTotalPrice() in the driver program, as well as Service’s ones (since Service inherits Artifact)
    - The **= 0** at the end of the function declaration signifies that the functions are pure virtual, meaning they have no implementation in the Item class itself. Pure virtual functions must be overridden by any concrete derived class before that class can be instantiated.

class Artifact (inherits Item)

* **private** members – the data slots
  + GUID (string) – we don’t need to check the GUID for Artifact since nowhere is required
  + Name (string)
  + Description (long string)
  + Category (string) – the artifact category
  + Division (Division) – division to which the artifact belongs; cannot be empty
  + Price (double) **- in Bulgarian leva.**
  + Discount (double) – used to reduce the price for the buyer
  + DiscountType (enum) – amount or percentage – depending of the type/reason the buyer gets
  + Quantity (number (int)) – the quantity that the person buys
* **public** members
  + Constructors (using the const keyword to prevent the new data from modification)
    - Standard (empty) constructor with default data for all data slots
      * GUID (string) - empty
      * Name (string) – e.g. Sony Vaio
      * Description (long string) - empty
      * Category (string) – the artifact category – e.g. Laptop
      * Division (Division) – new empty Division
      * Price (double) – e.g. 185.99 BGN
      * Discount (double) – e.g. 30
      * DiscountType (enum) – **amount (chosen** at random**)** or percentage
      * Quantity (number (int)) – 0
    - Argument constructor – used to create an artifact with previously set data.
      * Here we check whether the given price, discount and/or the quantity are less than 0. They cannot be negative numbers!
        + If any of them is, then the program throws an invalid argument exception.
        + If all of them are positive, the program continues.
    - Copy constructor – used to create a copy of another artifact in the currently generated new artifact
      * All of the data is being replicated, including the GUID.
  + Assignment operator= - it does the same operation just like the copy constructor
  + Methods (using the const keyword to prevent data modification)
    - double GetEffectivePrice()
      * Calculating the effective price in terms of what type of discount we have.
        + If it is an amount, we just reduce the price by it.
        + If it is a percentage, we calculate it by it.
      * Here we check whether the effective price is less than 0. It cannot be a negative number!
        + If it is, then the program throws an invalid argument exception.
        + If it is a positive, the program returns the effective price.
    - double GetTotalPrice()
      * Calculating the total price of the artifact by multiplying the effective price with the quantity the buyer requested.
      * Here we check whether the total price is less than 0. It cannot be a negative number!
        + If it is, then the program throws an invalid argument exception.
        + If it is a positive, the program returns the total price.
      * Abstraction
        + Artifact inherits Item. Therefore, when we call it from a pointer of Item with a data value of Artifact, this function will be used.
        + Also, there is no longer need to have **virtual** keyword in this instance of GetTotalPrice(). When Service calls the function, it automatically takes it to its instance.
        + When Service class inherits an Artifact, the requirements for getting the total price changes.

Other variables are needed which are not presented in Artifact when inherited by Service

Not to break the calculations of service and use only a part of the data.

* + - string GetName() - returning the name of the Artifact
    - string GetGUID() - returning the GUID of the Artifact

class Service (Inherits Artifact)

* **private** members – the data slots
  + Since the class inherits Artifact, we don’t need to implement again the previous data members (GUID, Name, etc,).
  + Duration (double) (service execution duration in hours)
  + Rate (double)
  + RateDiscount (double)
  + RateDiscountType – amount or percentage
* **public** members
  + Constructors (using the const keyword to prevent the new data from modification)
    - Standard (empty) constructor with default data for all data slots
      * It also inherits the Artifact empty constructor as we don’t need to present default values of the other data slots.
      * Duration (double) – e.g. 1.5 hours
      * Rate (double) – e.g. 8.50
      * RateDiscount (double) – e.g. 30
      * RateDiscountType – amount or **percentage (chosen** at random**)**
    - Argument constructor – used to create a service with previously set data.
      * It also inherits the Artifact argument constructor as we don’t need to present values of the other data slots.
      * Here we check whether the given duration, rate and/or the rateDiscount are less than 0. They cannot be negative numbers!
        + If any of them is, then the program throws an invalid argument exception.
        + If all of them are positive, the program continues.
    - Copy constructor – used to create a copy of another service in the currently generated new artifact
      * It also inherits the Artifact copy constructor as we don’t need to present values of the other data slots.
      * All of the data is being replicated, including the GUID.
  + Assignment operator= - it does the same operation just like the copy constructor
  + Methods (using the const keyword to prevent data modification)
    - double GetEffectivePrice() and string GetName() are inherited and there is no need to modify them.
      * That’s the reason why they are not virtual in Artifact class.
    - double GetEffectiveRate()
      * Calculating the effective rate in terms of what type of rate discount we have.
        + If it is an amount, we just reduce the rate by it.
        + If it is a percentage, we calculate it by it.
      * Here we check whether the effective rate is less than 0. It cannot be a negative number!
        + If it is, then the program throws an invalid argument exception.
        + If it is a positive, the program returns the effective rate.
    - double GetTotalPrice() – inheriting the one in Artifact class
      * Calculating the total price of the service by getting the total price through the instance of the method in the Artifact parent and adding the effective rate multiplied by the duration.
        + from virtual – when Service class inherits an Artifact, the requirements for getting the total price changes.

Other variables are needed which are not presented in Artifact when inherited by Service

Not to break the calculations of service and use only a part of the data.

* + - * Here we check whether the total price is less than 0. It cannot be a negative number!
        + If it is, then the program throws an invalid argument exception.
        + If it is a positive, the program returns the total price.

int main() – testing the program as it is required by the task.

1. Creating a vector pointer of the respected class – Item. Artifact inherits Item and Service inherits Artifact.
2. Creating three division pointers with sample data.
   * The GUIDs are generated from a website for examples.
   * The second division has the first one as its parent.
     + To show that the argument constructor for divisions work with and without a value for a parent.
3. Creating three artifact pointers that have a correct Division from the available and instantiated divisions in the previous step.
4. Creating three service pointers, that have a valid Division.
5. Adding the three artifacts to the Items list.
6. Adding the three services to the Items list.
7. Creating a variable for the search string which you enter when you are asked for.
8. Creating a vector of pairs from integers and Item pointers called *“itemsWithMatches”*.
   * The first element will be the number of occurrences of the match of the respective Item.
   * The second will be the Item pointer.
9. Perform pattern matching
   * Creating a variable of the *KMP* class called *“matcher”*.
   * Iterating with for-loop through the Items
     + Creating an instance of the current item’s name.
     + Calculating the number of matches of the search string in the item’s name.
       - Using the matcher’s SearchString function.
     + If there are matches in the item’s name, then the item is added in the vector for the items with at least one match of the requested string.
10. Sorting the *itemsWithMatches* by the number of occurrences of the string, from high to low.
11. Use a loop to show the Name and GUID of each artifact and service (the *itemsWithMatches*) in the store.
    * Making the loop so that you can see
      + the requested data
      + the search string
      + the number of the item added to the collection.
    * If there are no items with even one occurrence, we display “None”.
12. Items.clear(); - Clearing the vector
13. itemsWithMatches.clear(); // Clearing the second vector
14. Calling the *delete* operator so that all unneeded data (division1, division2, division3, artifact1, artifact2, artifact3, service1, service2, service3, itemsWithMatches and Items) gets collected as a garbage.
    * The program may be terminating after that but it is still a good practice.

**KMP.cpp**

* #include <iostream> - calling exceptions and console output
* #include <string> - using all properties of the type string
* #include <vector> - using vectors

using namespace std; - used for calling exceptions, checking whether there is a given GUID in the GUIDs vector, console output and to save time to not write everywhere ”std::” where it is required

class KMP

* **private** members
  + vector<int> computeLPS(const string& pattern)
    - This function computes the Longest Prefix Suffix (LPS) array for a given pattern string. The LPS array is used in the Knuth-Morris-Pratt (KMP) algorithm to efficiently search for occurrences of the pattern within a text. The LPS array stores the length of the longest proper prefix which is also a suffix for each position in the pattern.
* **public** members
  + int SearchString(const string& text, const string& pattern)
    - The function that is called in the main method.
    - This function implements the Knuth-Morris-Pratt (KMP) algorithm for string pattern matching. Given a text string and a pattern string, it searches for all occurrences of the pattern within the text. It returns a vector of pairs, where each pair contains the starting index of an occurrence of the pattern in the text and the pattern itself. This algorithm utilizes the LPS array computed for the pattern string to efficiently skip comparisons while searching for matches in the text.

*The program is tested with different values of different types of value. With the presented one, it should not give any exceptions. If it is changed so that exceptions are required, it will do so.*