**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **27-June-2020** | **Name:** | **Raziya Banu** |
| **Course:** | **C++ Programming** | **USN:** | **4AL16EC058** |
| **Topic:** | **Composition** | **Semester & Section:** | **8th sem & ‘B’ section** |
| **Github Repository:** |  |  |  |

|  |
| --- |
| **FORENOON SESSION DETAILS** |
| **Image of session** |
| **Report –**  In my first session today I have studied about –**Composite Design Pattern in C++**  **Prerequisite :**  Composite pattern is one of the most widely used patterns in the industry and addresses a very significant and subtle problem. It is used whenever the user wants to treat the individual object in the same way as the collection of those individual objects for e.g you might want to consider a page from the copy as same as the whole copy which is basically a collection of the pages or if you want to create a hierarchy of something where you might want to consider the whole thing as the object . Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.  In the case of photoshop where we draw many individual objects and then those objects compose a whole unique object and you might want to apply some operation on the whole object instead of the each of the individual objects. https://media.geeksforgeeks.org/wp-content/cdn-uploads/composite_design_pattern_diagram.png  Here in this diagram, as you can see both composite and Leaf implements Component diagram, thus allowing the same operation on both objects but the important part is Composite Class which also contain the Component Objects which is symbolized by the black diamond indicating composition relationship between Composite and Component class.  Then how to design our classes to accommodate such scenarios. We will try to understand it by implementing our copy example. Say you have to create a page which has operations like add, delete, remove and also a copy which will have the same operations as the individual pages.  Such situation is the best dealt with the composite pattern.  filter\_none  edit  play\_arrow  brightness\_4   |  | | --- | | // CPP program to illustrate  // Composite design pattern  #include <iostream>  #include <vector>  using namespace std;  class PageObject {  public:      virtual void Add(PageObject a)      {      }      virtual void Remove()      {      }      virtual void Delete(PageObject a)      {      }  };  class Page : public PageObject {  public:      void Add(PageObject a)      {          cout << "something is added to the page" << endl;      }      void Remove()      {          cout << "soemthing is removed from the page" << endl;      }      void Delete(PageObject a)      {          cout << "soemthing is deleted from page " << endl;      }  };    class Copy : public PageObject {      vector<PageObject> copyPages;    public:      void AddElement(PageObject a)      {          copyPages.push\_back(a);      }        void Add(PageObject a)      {          cout << "something is added to the copy" << endl;      }      void Remove()      {          cout << "something is removed from the copy" << endl;      }      void Delete(PageObject a)      {          cout << "something is deleted from the copy";      }  };    int main()  {      Page a;      Page b;      Copy allcopy;      allcopy.AddElement(a);      allcopy.AddElement(b);        allcopy.Add(a);      a.Add(b);        allcopy.Remove();      b.Remove();        return 0;  } |   something is added to the copy  something is added to the page  something is removed from the copy  soemthing is removed from the page |

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **27-June-2020** | **Name:** | **Raziya Banu** |
| **Course:** | **C++ Programming** | **USN:** | **4AL16EC058** |
| **Topic:** | **Friend Keyword** | **Semester & Section:** | **8th sem & ‘B’ section** |
| **AFTERNOON SESSION DETAILS** | | | |
| **Image of session** | | | |
| **Friend class and function in C++** **Friend Class** A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class. For example a LinkedList class may be allowed to access private members of Node.  filter\_none  edit  play\_arrow  brightness\_4   |  | | --- | | class Node {  private:      int key;      Node\* next;      /\* Other members of Node Class \*/        // Now class  LinkedList can      // access private members of Node      friend class LinkedList;  }; |   **Friend Function** Like friend class, a friend function can be given special grant to access private and protected members. A friend function can be: a) A method of another class b) A global function  filter\_none  edit  play\_arrow  brightness\_4   |  | | --- | | class Node {  private:      int key;      Node\* next;        /\* Other members of Node Class \*/      friend int LinkedList::search();      // Only search() of linkedList      // can access internal members  }; |   Following are some important points about friend functions and classes: **1)** Friends should be used only for limited purpose. too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.  **2)** Friendship is not mutual. If class A is a friend of B, then B doesn’t become a friend of A automatically.  **3)** Friendship is not inherited (See [this](https://www.geeksforgeeks.org/g-fact-34/)for more details)  **4)** The concept of friends is not there in Java.  **A simple and complete C++ program to demonstrate friend Class**  filter\_none  edit  play\_arrow  brightness\_4   |  | | --- | | #include <iostream>  class A {  private:      int a;    public:      A() { a = 0; }      friend class B; // Friend Class  };    class B {  private:      int b;    public:      void showA(A& x)      {          // Since B is friend of A, it can access          // private members of A          std::cout << "A::a=" << x.a;      }  };    int main()  {      A a;      B b;      b.showA(a);      return 0;  } |   Output:  A::a=0  **A simple and complete C++ program to demonstrate friend function of another class**  filter\_none  edit  play\_arrow  brightness\_4   |  | | --- | | #include <iostream>    class B;    class A {  public:      void showB(B&);  };    class B {  private:      int b;    public:      B() { b = 0; }      friend void A::showB(B& x); // Friend function  };    void A::showB(B& x)  {      // Since showB() is friend of B, it can      // access private members of B      std::cout << "B::b = " << x.b;  }    int main()  {      A a;      B x;      a.showB(x);      return 0;  } |   Output:  B::b = 0 | | | |