**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **25-06-2020** | **Name:** | **Dhanya Shetty** |
| **Course:** | **C PROGRAMMING** | **USN:** | **4AL17EC026** |
| **Topic:** | **1.INHERITANCE & POLYMORPHISM**  **2.TEMPLATES, EXCEPTIONS & FILES** | **Semester & Section:** | **6th A** |
| **Github Repository:** | **Dhanya Shetty\_026** |  |  |

|  |
| --- |
| **FORENOON SESSION DETAILS** |
| C:\Users\Hp\Desktop\report\25june111.PNG  C:\Users\Hp\Desktop\report\25june222.PNG  **C:\Users\Hp\Desktop\report\25june333.PNG**  **C:\Users\Hp\Desktop\report\25june444.PNG**  **C:\Users\Hp\Desktop\report\25june555.PNG**  **C:\Users\Hp\Desktop\report\25june11111.PNG**  **C:\Users\Hp\Desktop\report\25june22222.PNG**  **C Programming :**  **C** is highly portable and is **used for** scripting system applications which form a major part of Windows, UNIX, and Linux operating system. **C** is a general-purpose **programming language** and can efficiently work on enterprise applications, games, graphics, and applications requiring calculations, etc.  **Inheritance :**  **Inheritance**is one of the most important concepts of object-oriented programming. Inheritance allows us to define a class based on another class. This facilitates greater ease in creating and maintaining an application.  The class whose properties are inherited by another class is called the **Base** class. The class which inherits the properties is called the **Derived** class. For example, the **Daughter**class (derived) can be inherited from the **Mother**class (base). The derived class inherits all feature from the base class, and can have its own additional features.  To demonstrate inheritance, let's create a **Mother**class and a **Daughter**class:class Mother { public: Mother() {}; void sayHi() { cout << "Hi"; }  };  class Daughter  { public:  Daughter() {}; }; The Mother class has a public method called **sayHi()**.  **Polymorphism :**  The word **polymorphism**means "having many forms". Typically, polymorphism occurs when there is a hierarchy of classes and they are related by **inheritance**.  C++ polymorphism means that a call to a member function will cause a **different**implementation to be executed depending on the **type**of object that invokes the function.  Second step is to create classes for two different types of enemies, **Ninjas**and **Monsters**. Both of these new classes inherit from the **Enemy**class, so each has an attack power. At the same time, each has a specific **attack**function.class **Ninja**: public Enemy { public: void attack() { cout << "Ninja! - "<<attackPower<<endl; }};  class **Monster**: public Enemy { public: void attack() { cout << "Monster! - "<<attackPower<<endl; } }; As you can see, their individual **attack**functions differ. Now we can create our **Ninja**and **Monster**objects in main.int main() {  **Ninja** n; **Monster** m;  } **Ninja**and **Monster**inherit from **Enemy**, so all **Ninja**and **Monster**objects are **Enemy**objects. This allows us to do the following:**Enemy** \*e1 = &n; **Enemy** \*e2 = &m;  **Virtual Functions :**  The previous example demonstrates the use of base class pointers to the derived classes. Why is that useful? Continuing on with our game example, we want every Enemy to have an **attack()** function. To be able to call the corresponding attack() function for each of the derived classes using Enemy pointers, we need to declare the base class function as **virtual**. Defining a virtual function in the base class, with a corresponding version in a derived class, allows polymorphism to use Enemy pointers to call the derived classes' functions. Every derived class will override the attack() function and have a separate implementation:class Enemy { public: **virtual** void attack() { } };  class Ninja: public Enemy { public: void attack() { cout << "Ninja!"<<endl; } };  class Monster: public Enemy { public: void attack() { cout << "Monster!"<<endl; } };  **Function Templates :**  Functions and classes help to make programs easier to write, safer, and more maintainable. However, while functions and classes do have all of those advantages, in certain cases they can also be somewhat limited by C++'s requirement that you specify types for all of your parameters.  For example, you might want to write a function that calculates the sum of two numbers, similar to this: int sum(int a, int b) { return a+b; }  We can now call the function for two integers in our main.  int sum(int a, int b) { return a+b; }  int main () { int x=7, y=15; cout << **sum**(x, y) << endl; } // Outputs 22  **Function templates**give us the ability to do that! With function templates, the basic idea is to avoid the necessity of specifying an exact type for each variable. Instead, C++ provides us with the capability of defining functions using placeholder types, called **template type parameters**.  To define a function template, use the keyword **template**, followed by the template type definition:**template** <class T>  Template functions can save a lot of time, because they are written only once, and work with different types. Template functions reduce code maintenance, because duplicate code is reduced significantly. |

|  |
| --- |
|  |
|  |
|  |