**DAILY ASSESSMENT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **25/06/2020** | **Name:** | **Dhavala** |
| **Course:** | **C++ Programming** | **USN:** | **4AL17EC027** |
| **Topic:** | * **Module 5: Inheritance and Polymorphism** | **Semester & Section:** | **6TH SEM & A Section** |
| **Github Repository:** | **Dhavala27** |  |  |

|  |
| --- |
| **SESSION DETAILS** |
|  |

|  |
| --- |
| **Report** 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 featuresInheritance 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().Type of Inheritance Access specifiers are also used to specify the type of inheritance. Remember, we used public to inherit the Daughter class:class Daughter: public Mother private and protected access specifiers can also be used here. Public Inheritance: public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class. Protected Inheritance: public and protected members of the base class become protected members of the derived class. Private Inheritance: public and protected members of the base class become private members of the derived class.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.Polymorphism Our 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; |

**DAILY ASSESSMENT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **25/06/2020** | **Name:** | **Dhavala** |
| **Course:** | **C++ Programming** | **USN:** | **4AL17EC027** |
| **Topic:** | * **Module 4: Classes and objects** | **Semester & Section:** | **6TH SEM & A Section** |
| **Github Repository:** | **Dhavala27** |  |  |

|  |
| --- |
| **SESSION DETAILS** |
|  |

|  |
| --- |
| **Report** 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; }Function Templates 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 It becomes necessary to write a new function for each new type, such as doubles.double sum(double a, double b) { return a+b; } Wouldn't it be much more efficient to be able to write one version of sum() to work with parameters of any type? 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>Class Templates Just as we can define function templates, we can also define class templates, allowing classes to have members that use template parameters as types. The same syntax is used to define the class template:template <class T> class MyClass { };Class Templates As an example, let's create a class Pair, that will be holding a pair of values of a generic type.template <class T> class Pair { private: T first, second; public: Pair (T a, T b): first(a), second(b) { } };Class Templates A specific syntax is required in case you define your member functions outside of your class - for example in a separate source file. You need to specify the generic type in angle brackets after the class name. For example, to have a member function bigger() defined outside of the class, the following syntax is used:template <class T> class Pair { private: T first, second; public: Pair (T a, T b): first(a), second(b){ } T bigger(); }; template <class T> T Pair<T>::bigger() { // some code } |