**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **25 june 2020** | **Name:** | **Sanketh S Acharya** |
| **Course:** | **C++ Programming** | **USN:** | **4AL17EC084** |
| **Topic:** | **Inheritance & polymorphism** | **Semester & Section:** | **6TH SEM & ‘B’ SEC** |
| **Github Repository:** |  |  |  |

|  |
| --- |
| **FORENOON SESSION DETAILS** |
| **Image of session**  **C:\Users\cw\Desktop\25 j1.png** |
| **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 features.**https://api.sololearn.com/DownloadFile?id=2465  **Access Specifiers**  **Up to this point, we have worked exclusively with public and private access specifiers. Public members may be accessed from anywhere outside of the class, while access to private members is limited to their class and friend functions.**  **Protected**  **There is one more access specifier - protected. A protected member variable or function is very similar to a private member, with one difference - it can be accessed in the derived classes.**  **class Mother { public: void sayHi() { cout << var; }  private: int var=0;  protected: int someVar; };**  **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. P**  **Inheritance**  **When inheriting classes, the base class' constructor and destructor are not inherited. However, they are being called when an object of the derived class is created or deleted.  To further explain this behavior, let's create a sample class that includes a constructor and a destructor:class Mother { public: Mother()  { cout <<"Mother ctor"<<endl; } ~Mother() { cout <<"Mother dtor"<<endl; } }; Creating an object in main results in the following output:**  **int main() { Mother m; } /\* Outputs Mother ctor Mother dtor \*/**  **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**can be demonstrated more clearly using an example: Suppose you want to make a simple game, which includes different enemies: monsters, ninjas, etc. All enemies have one function in common: an **attack**function. However, they each attack in a different way. In this situation, polymorphism allows for calling the same **attack**function on different objects, but resulting in different behaviors.  The first step is to create the **Enemy**class.class Enemy { protected:  int attackPower; public: void setAttackPower(int a){ attackPower = a; } };  **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; } };**    **Pure Virtual Function**  **In some situations you'd want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class. The virtual member functions without definition are known as pure virtual functions. They basically specify that the derived classes define that function on their own. The syntax is to replace their definition by =0 (an equal sign and a zero):class Enemy { public: virtual void attack() = 0; };** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:25 june 2020** |  | **Name:Sanketh S Acharya** |  |
| **Course:C++ Programming** |  | **USN:4AL17EC084** |  |
| **Topic:module 8: Templates ,Exceptions and, Files** |  | **Semester & Section:6th sem & ‘B’ sec** |  |
| **AFTERNOON SESSION DETAILS** | | | |
| **Image of session**  **C:\Users\cw\Desktop\25 j2.png** | | | |
| **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; }**  **Now we can use our generic data type T in the function:**  **template <class T> T sum(T a, T b) { return a+b; }  int main () { int x=7, y=15; cout << sum(x, y) << endl; }  // Outputs 22**  **Function templates also make it possible to work with multiple generic data types. Define the data types using a comma-separated list. Let's create a function that compares arguments of varying data types (an int and a double), and prints the smaller one.template <class T, class U>**  **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**  **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 }**  **Template Specialization**  **In case of regular class templates, the way the class handles different data types is the same; the same code runs for all data types. Template specialization allows for the definition of a different implementation of a template when a specific type is passed as a template argument.  For example, we might need to handle the character data type in a different manner than we do numeric data types. To demonstrate how this works, we can first create a regular template.template <class T> class MyClass { public: MyClass (T x) { cout <<x<<" - not a char"<<endl; } };**  **Exceptions**  **Problems that occur during program execution are called exceptions. In C++ exceptions are responses to anomalies that arise while the program is running, such as an attempt to divide by zero.**  **Working with Files**  **Another useful C++ feature is the ability to read and write to files. That requires the standard C++ library called fstream. Three new data types are defined in fstream: ofstream: Output file stream that creates and writes information to files. ifstream: Input file stream that reads information from files. fstream: General file stream, with both ofstream and ifstream capabilities that allow it to create, read, and write information to files.  To perform file processing in C++, header files <iostream> and <fstream> must be included in the C++ source file.#include <iostream> #include <fstream>**  **Working with Files**  **You can also provide the path to your file using the ofstream objects constructor, instead of calling the open function.#include <fstream> using namespace std;  int main() { ofstream MyFile("test.txt");  MyFile << "This is awesome! \n"; MyFile.close(); }** | | | |