**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **25/June/2020** | **Name:** | **Prashantha naik** |
| **Course:** | **C++ (Sololearn)** | **USN:** | **4al17ec074** |
| **Topic:** | **Module 7: Inheritance & Polymorphism** | **Semester & Section:** | **6th b** |
| **GitHub Repository:** | **prashanth\_course** |  |  |

|  |
| --- |
| **FORENOON SESSION DETAILS** |
| **Image of session** |
| **Report – Report can be typed or hand written for up to two pages.**  **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 **.**  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() {}; };  This syntax derives the Daughter class from the Mother class.class Daughter : public Mother { public:  Daughter() {}; };  **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;  };  **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.  **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; } }; |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Date:** | **25/June/2020** | **Name:** | **Prashantha naik** | |
| **Course:** | **C++ (Sololearn)** | **USN:** | **4al17ec074** | |
| **Topic:** | **Module 8: Templates, Exceptions, and Files** | **Semester&Section:** | **6th b** | |
| **Git hub repository** | **prashanth\_couse** |  |  | |
| **AFTERNOON SESSION DETAILS** | | | |
| **Image of session** | | | |
| **Report – Report can be typed or hand written for up to two pages.**  **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;  }  **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 {  };  **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.  **Throwing Exceptions**  C++ exception handling is built upon three keywords: try, catch, and throw.  throw is used to throw an exception when a problem shows up.  For example:  int motherAge = 29;  int sonAge = 36;  if (sonAge > motherAge) {  throw "Wrong age values";  }  The code looks at sonAge and motherAge, and throws an exception if sonAge is found to be the greater of the two.  **Catching Exceptions**  A try block identifies a block of code that will activate specific exceptions. It's followed by one or more catch blocks. The catch keyword represents a block of code that executes when a particular exception is thrown.  Code that could generate an exception is surrounded with the try/catch block.  You can specify what type of exception you want to catch by the exception declaration that appears in parentheses following the keyword catch.  For example:  try {  int motherAge = 29;  int sonAge = 36;  if (sonAge > motherAge) {  throw 99;  }  }  catch (int x) {  cout<<"Wrong age values - Error "<<x;  }  //Outputs "Wrong age values - Error 99"  Try It Yourself  The try block throws the exception, and the catch block then handles it.  The error code 99, which is an integer, appears in the throw statement, so it results in an exception of type int.  **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>  **Opening a File**  A file must be opened before you can read from it or write to it. Either the ofstream or fstream object may be used to open a file for writing. Let's open a file called "test.txt" and write some content to it:#include <iostream> #include <fstream> using namespace std;  int main() { ofstream MyFile; MyFile.open("test.txt");  MyFile << "Some text. \n"; }  **Closing a File**  When you've finished working with a file, close it using the member function close().#include <iostream> #include <fstream> using namespace std;  int main() { ofstream MyFile; MyFile.open("test.txt");  MyFile << "Some text! \n"; MyFile.close(); } | | | |