**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **25/06/2020** | **Name:** | **Gaganashree P** |
| **Course:** | **C++** | **USN:** | **4AL16EC042** |
| **Topic:** | **Inheritance**  **Files,exceptions** | **Semester & Section:** | **8 A** |
| **Github Repository:** | **Gaganashree-P** |  |  |

|  |
| --- |
| **FORENOON SESSION DETAILS** |
| **Image of session** |
| **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.   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 \*/ Virtual Functions Virtual functions can also have their implementation in the base class:class Enemy { public: virtual void attack() { cout << "Enemy!"<<endl; } };  class Ninja: public Enemy { public: void attack() { cout << "Ninja!"<<endl; } };  class Monster: public Enemy { public: void attack() { cout << "Monster!"<<endl; } }; Now, when you create an Enemy pointer, and call the attack() function, the compiler will call the function, which corresponds to the object's type, to which the pointer points:  int main() { Ninja n; Monster m; Enemy e;  Enemy \*e1 = &n; Enemy \*e2 = &m; Enemy \*e3 = &e;  e1->attack(); // Outputs "Ninja!"  e2->attack(); // Outputs "Monster!"  e3->attack(); // Outputs "Enemy!" } 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 {  }; 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: 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(); } 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> |