**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **24/June/2020** | **Name:** | **Prashantha naik** |
| **Course:** | **C++ (Solo learn)** | **USN:** | **4al17ec074** |
| **Topic:** | **Module 5: Classes and Objects** | **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.**  **What is an Object**  Object Oriented Programming is a programming style that is intended to make thinking about programming closer to thinking about the real world. In programming, objects are independent units, and each has its own identity, just as objects in the real world do.  **Objects**  An object might contain other objects but they're still different objects.  Objects also have characteristics that are used to describe them. For example, a car can be red or blue, a mug can be full or empty, and so on. These characteristics are also called attributes. An attribute describes the current state of an object.  Objects can have multiple attributes (the mug can be empty, red and large).  **What is a Class**  Objects are created using classes, which are actually the focal point of OOP. The class describes what the object will be, but is separate from the object itself. In other words, a class can be described as an object's blueprint, description, or definition. You can use the same class as a blueprint for creating multiple different objects. For example, in preparation to creating a new building, the architect creates a blueprint, which is used as a basis for actually building the structure. That same blueprint can be used to create multiple buildings. Programming works in the same fashion. We first define a class, which becomes the blueprint for creating objects.  **Declaring a Class**  Begin your class definition with the keyword class. Follow the keyword with the class name and the class body, enclosed in a set of curly braces. The following code declares a class called BankAccount:class BankAccount {  };  Define all attributes and behavior (or members) in the body of the class, within curly braces. You can also define an access specifier for members of the class. A member that has been defined using the public keyword can be accessed from outside the class, as long as it's anywhere within the scope of the class object.  **Creating a Class**  Let's create a class with one public method, and have it print out "Hi".class BankAccount { public: void sayHi() { cout << "Hi" << endl; } }; The next step is to instantiate an object of our BankAccount class, in the same way we define variables of a type, the difference being that our object's type will be BankAccount.  int main()  { BankAccount test; test.sayHi(); }  **Abstraction** Data abstraction is the concept of providing only essential information to the outside world. It's a process of representing essential features without including implementation details.  A good real-world example is a *book*: When you hear the term book, you don't know the exact specifics, i.e.: the page count, the color, the size, but you understand the idea of a book - the abstraction of the book.  **Encapsulation**  Part of the meaning of the word encapsulation is the idea of "surrounding" an entity, not just to keep what's inside together, but also to protect it. In object orientation, encapsulation means more than simply combining attributes and behavior together within a class; it also means restricting access to the inner workings of that class.  The key principle here is that an object only reveals what the other application components require to effectively run the application. All else is kept out of view.  **Constructors**  Class constructors are special member functions of a class. They are executed whenever new objects are created within that class.  The constructor's name is identical to that of the class. It has no return type, not even void.  For example:  class myClass { public: myClass() { cout <<"Hey"; } void setName(string x) { name = x; } string getName() { return name; } private: string name; };  int main() { myClass myObj;  return 0; }  //Outputs "Hey" |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Date:** | **24/June/2020** | **Name:** | **Prashantha naik** | |
| **Course:** | **C++ (Solo learn)** | **USN:** | **4al17ec074** | |
| **Topic:** | **Module 6: More on Classes** | **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.**  **Destructors**  Remember constructors? They're special member functions that are automatically called when an object is created. Destructors are special functions, as well. They're called when an object is destroyed or deleted.  The name of a destructor will be exactly the same as the class, only prefixed with a tilde (~). A destructor can't return a value or take any parameters.class MyClass { public:  ~MyClass() { // some code } };  After declaring the destructor in the header file, we can write the implementation in the source file MyClass.cpp:#include "MyClass.h" #include <iostream> using namespace std;  MyClass::MyClass() { cout<<"Constructor"<<endl; }  MyClass::~MyClass() { cout<<"Destructor"<<endl; }  **#ifndef & #define**  We created separate header and source files for our class, which resulted in this header file.#ifndef MYCLASS\_H #define MYCLASS\_H  class MyClass { public: MyClass(); protected: private: };  #endif // MYCLASS\_H  ifndef stands for "if not defined". The first pair of statements tells the program to define the MyClass header file if it has not been defined already. endif ends the condition**.**  **Member Functions**  Let's create a sample function called myPrint() in our class. MyClass.hclass MyClass { public: MyClass(); void myPrint(); }; MyClass.cpp#include "MyClass.h" #include <iostream> using namespace std;  MyClass::MyClass() { }  void MyClass::myPrint() { cout <<"Hello"<<endl; }  **Dot Operator**  Next, we'll create an object of the type MyClass, and call its myPrint() function using the dot (.) operator:  #include "MyClass.h"  int main() { MyClass obj; obj.myPrint(); }  // Outputs "Hello"  **Constant Objects**  As with the built-in data types, we can make class objects constant by using the const keyword.const MyClass obj; All const variables must be initialized when they're created. In the case of classes, this initialization is done via constructors. If a class is not initialized using a parameterized constructor, a public default constructor must be provided - if no public default constructor is provided, a compiler error will occur.  **Friend Functions**  Normally, private members of a class cannot be accessed from outside of that class. However, declaring a non-member function as a friend of a class allows it to access the class' private members. This is accomplished by including a declaration of this external function within the class, and preceding it with the keyword friend. In the example below, someFunc(), which is not a member function of the class, is a friend of MyClass and can access its private members.class MyClass { public: MyClass() { regVar = 0; } private: int regVar;  friend void someFunc(MyClass &obj); };  **This**  Every object in C++ has access to its own address through an important pointer called the this pointer. Inside a member function this may be used to refer to the invoking object. Let's create a sample class:class MyClass { public: MyClass(int a) : var(a) { } private: int var; };  **Operator Overloading**  Most of the C++ built-in operators can be redefined or overloaded. Thus, operators can be used with user-defined types as well (for example, allowing you to add two objects together).  This chart shows the operators that can be overloaded**.** | | | |