# DAILY ASSESSMENT

|  |  |  |  |
| --- | --- | --- | --- |
| Date: | 25/06/2020 | Name: | Chesmi B R |
| Course: | **C++** | USN: | 4AL16EC100 |
| Topic: | **Module 7: Inheritance and polymorphism** | Semester & Section: | 8TH SEM & A Section |
| Github Repository: | chesmibr |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **FORENOON SESSION DETAILS**     Report:C++ Inheritance One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.  When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class.  The idea of inheritance implements the is a relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on. Base and Derived Classes A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form −  class derived-class: access-specifier base-class  Where access-specifier is one of public, protected, or private, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.  Consider a base class Shape and its derived class Rectangle as follows −  #include <iostream>    using namespace std;  // Base class  class Shape {  public:  void setWidth(int w) {  width = w;  }  void setHeight(int h) {  height = h;  }    protected:  int width;  int height;  };  // Derived class  class Rectangle: public Shape {  public:  int getArea() {  return (width \* height);  }  };  int main(void) {  Rectangle Rect;    Rect.setWidth(5);  Rect.setHeight(7);  // Print the area of the object.  cout << "Total area: " << Rect.getArea() << endl;  return 0;  }  When the above code is compiled and executed, it produces the following result −  Total area: 35 Access Control and Inheritance A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.  We can summarize the different access types according to - who can access them in the following way −   |  |  |  |  | | --- | --- | --- | --- | | Access | public | protected | private | | Same class | yes | yes | yes | | Derived classes | yes | yes | no | | Outside classes | yes | no | no |   A derived class inherits all base class methods with the following exceptions −   * Constructors, destructors and copy constructors of the base class. * Overloaded operators of the base class. * The friend functions of the base class.  Type of Inheritance When deriving a class from a base class, the base class may be inherited through public, protected or private inheritance. The type of inheritance is specified by the access-specifier as explained above.  We hardly use protected or private inheritance, but public inheritance is commonly used. While using different type of inheritance, following rules are applied −   * Public Inheritance − When deriving a class from a public base class, 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 − When deriving from a protected base class, public and protected members of the base class become protected members of the derived class. * Private Inheritance − When deriving from a private base class, public and protected members of the base class become private members of the derived class.  Multiple Inheritance A C++ class can inherit members from more than one class and here is the extended syntax −  class derived-class: access baseA, access baseB....  Where access is one of public, protected, or private and would be given for every base class and they will be separated by comma as shown above. Let us try the following example −  #include <iostream>    using namespace std;  // Base class Shape  class Shape {  public:  void setWidth(int w) {  width = w;  }  void setHeight(int h) {  height = h;  }    protected:  int width;  int height;  };  // Base class PaintCost  class PaintCost {  public:  int getCost(int area) {  return area \* 70;  }  };  // Derived class  class Rectangle: public Shape, public PaintCost {  public:  int getArea() {  return (width \* height);  }  };  int main(void) {  Rectangle Rect;  int area;    Rect.setWidth(5);  Rect.setHeight(7);  area = Rect.getArea();    // Print the area of the object.  cout << "Total area: " << Rect.getArea() << endl;  // Print the total cost of painting  cout << "Total paint cost: $" << Rect.getCost(area) << endl;  return 0;  }  When the above code is compiled and executed, it produces the following result −  Total area: 35  Total paint cost: $2450 Polymorphism in C++ 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 function to be executed depending on the type of object that invokes the function.  Consider the following example where a base class has been derived by other two classes −  #include <iostream>  using namespace std;    class Shape {  protected:  int width, height;    public:  Shape( int a = 0, int b = 0){  width = a;  height = b;  }  int area() {  cout << "Parent class area :" <<endl;  return 0;  }  };  class Rectangle: public Shape {  public:  Rectangle( int a = 0, int b = 0):Shape(a, b) { }    int area () {  cout << "Rectangle class area :" <<endl;  return (width \* height);  }  };  class Triangle: public Shape {  public:  Triangle( int a = 0, int b = 0):Shape(a, b) { }    int area () {  cout << "Triangle class area :" <<endl;  return (width \* height / 2);  }  };  // Main function for the program  int main() {  Shape \*shape;  Rectangle rec(10,7);  Triangle tri(10,5);  // store the address of Rectangle  shape = &rec;    // call rectangle area.  shape->area();  // store the address of Triangle  shape = &tri;    // call triangle area.  shape->area();    return 0;  }  When the above code is compiled and executed, it produces the following result −  Parent class area :  Parent class area :  The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the program is executed. This is also sometimes called **early binding** because the area() function is set during the compilation of the program.  But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword **virtual** so that it looks like this −  class Shape {  protected:  int width, height;    public:  Shape( int a = 0, int b = 0) {  width = a;  height = b;  }  virtual int area() {  cout << "Parent class area :" <<endl;  return 0;  }  };  After this slight modification, when the previous example code is compiled and executed, it produces the following result −  Rectangle class area  Triangle class area  This time, the compiler looks at the contents of the pointer instead of it's type. Hence, since addresses of objects of tri and rec classes are stored in \*shape the respective area() function is called.  As you can see, each of the child classes has a separate implementation for the function area(). This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations. Virtual Function A **virtual** function is a function in a base class that is declared using the keyword **virtual**. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.  What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage**, or **late binding**. Pure Virtual Functions It is possible that you 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.  We can change the virtual function area() in the base class to the following −  class Shape {  protected:  int width, height;  public:  Shape(int a = 0, int b = 0) {  width = a;  height = b;  }    // pure virtual function  virtual int area() = 0;  };  The = 0 tells the compiler that the function has no body and above virtual function will be called **pure virtual function**. |

**DAILY ASSESSMENT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **25/06/2020** | **Name:** | **Chesmi B R** |
| **Course:** | **C++** | **USN:** | **4AL16EC100** |
| **Topic:** | **Module 8: Templates, exceptions and files** | **Semester & Section:** | **8TH SEM & A Section** |
| **Github Repository:** | **Chesmibr** |  |  |

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

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Report**- C++ Templates Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.  A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.  There is a single definition of each container, such as vector, but we can define many different kinds of vectors for example, vector <int> or vector <string>.  You can use templates to define functions as well as classes, let us see how they work − Function Template The general form of a template function definition is shown here −  template <class type> ret-type func-name(parameter list) {  // body of function  }  Here, type is a placeholder name for a data type used by the function. This name can be used within the function definition.  The following is the example of a function template that returns the maximum of two values −  #include <iostream>  #include <string>  using namespace std;  template <typename T>  inline T const& Max (T const& a, T const& b) {  return a < b ? b:a;  }  int main () {  int i = 39;  int j = 20;  cout << "Max(i, j): " << Max(i, j) << endl;  double f1 = 13.5;  double f2 = 20.7;  cout << "Max(f1, f2): " << Max(f1, f2) << endl;  string s1 = "Hello";  string s2 = "World";  cout << "Max(s1, s2): " << Max(s1, s2) << endl;  return 0;  }  If we compile and run above code, this would produce the following result −  Max(i, j): 39  Max(f1, f2): 20.7  Max(s1, s2): World Class Template Just as we can define function templates, we can also define class templates. The general form of a generic class declaration is shown here −  template <class type> class class-name {  .  .  .  }  Here, type is the placeholder type name, which will be specified when a class is instantiated. You can define more than one generic data type by using a comma-separated list.  Following is the example to define class Stack<> and implement generic methods to push and pop the elements from the stack −  #include <iostream>  #include <vector>  #include <cstdlib>  #include <string>  #include <stdexcept>  using namespace std;  template <class T>  class Stack {  private:  vector<T> elems; // elements  public:  void push(T const&); // push element  void pop(); // pop element  T top() const; // return top element    bool empty() const { // return true if empty.  return elems.empty();  }  };  template <class T>  void Stack<T>::push (T const& elem) {  // append copy of passed element  elems.push\_back(elem);  }  template <class T>  void Stack<T>::pop () {  if (elems.empty()) {  throw out\_of\_range("Stack<>::pop(): empty stack");  }    // remove last element  elems.pop\_back();  }  template <class T>  T Stack<T>::top () const {  if (elems.empty()) {  throw out\_of\_range("Stack<>::top(): empty stack");  }    // return copy of last element  return elems.back();  }  int main() {  try {  Stack<int> intStack; // stack of ints  Stack<string> stringStack; // stack of strings  // manipulate int stack  intStack.push(7);  cout << intStack.top() <<endl;  // manipulate string stack  stringStack.push("hello");  cout << stringStack.top() << std::endl;  stringStack.pop();  stringStack.pop();  } catch (exception const& ex) {  cerr << "Exception: " << ex.what() <<endl;  return -1;  }  }  If we compile and run above code, this would produce the following result −  7  hello  Exception: Stack<>::pop(): empty stack C++ Exception Handling An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.  Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: try, catch, and throw.   * throw − A program throws an exception when a problem shows up. This is done using a throw keyword. * catch − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception. * try − A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.   Assuming a block will raise an exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch as follows −  try {  // protected code  } catch( ExceptionName e1 ) {  // catch block  } catch( ExceptionName e2 ) {  // catch block  } catch( ExceptionName eN ) {  // catch block  }  You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations. Throwing Exceptions Exceptions can be thrown anywhere within a code block using throw statement. The operand of the throw statement determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.  Following is an example of throwing an exception when dividing by zero condition occurs −  double division(int a, int b) {  if( b == 0 ) {  throw "Division by zero condition!";  }  return (a/b);  } Catching Exceptions The catch block following the try block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.  try {  // protected code  } catch( ExceptionName e ) {  // code to handle ExceptionName exception  }  Above code will catch an exception of ExceptionName type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows −  try {  // protected code  } catch(...) {  // code to handle any exception  }  The following is an example, which throws a division by zero exception and we catch it in catch block.  #include <iostream>  using namespace std;  double division(int a, int b) {  if( b == 0 ) {  throw "Division by zero condition!";  }  return (a/b);  }  int main () {  int x = 50;  int y = 0;  double z = 0;    try {  z = division(x, y);  cout << z << endl;  } catch (const char\* msg) {  cerr << msg << endl;  }  return 0;  }  Because we are raising an exception of type const char\*, so while catching this exception, we have to use const char\* in catch block. If we compile and run above code, this would produce the following result −  Division by zero condition! C++ Standard Exceptions C++ provides a list of standard exceptions defined in <exception> which we can use in our programs.  Here is the small description of each exception mentioned in the above hierarchy −   |  |  | | --- | --- | | Sr.No | Exception & Description | | 1 | std::exception  An exception and parent class of all the standard C++ exceptions. | | 2 | std::bad\_alloc  This can be thrown by new. | | 3 | std::bad\_cast  This can be thrown by dynamic\_cast. | | 4 | std::bad\_exception  This is useful device to handle unexpected exceptions in a C++ program. | | 5 | std::bad\_typeid  This can be thrown by typeid. | | 6 | std::logic\_error  An exception that theoretically can be detected by reading the code. | | 7 | std::domain\_error  This is an exception thrown when a mathematically invalid domain is used. | | 8 | std::invalid\_argument  This is thrown due to invalid arguments. | | 9 | std::length\_error  This is thrown when a too big std::string is created. | | 10 | std::out\_of\_range  This can be thrown by the 'at' method, for example a std::vector and std::bitset<>::operator[](). | | 11 | std::runtime\_error  An exception that theoretically cannot be detected by reading the code. | | 12 | std::overflow\_error  This is thrown if a mathematical overflow occurs. | | 13 | std::range\_error  This is occurred when you try to store a value which is out of range. | | 14 | std::underflow\_error  This is thrown if a mathematical underflow occurs. |  Define New Exceptions You can define your own exceptions by inheriting and overriding exception class functionality. Following is the example, which shows how you can use std::exception class to implement your own exception in standard way −  #include <iostream>  #include <exception>  using namespace std;  struct MyException : public exception {  const char \* what () const throw () {  return "C++ Exception";  }  };    int main() {  try {  throw MyException();  } catch(MyException& e) {  std::cout << "MyException caught" << std::endl;  std::cout << e.what() << std::endl;  } catch(std::exception& e) {  //Other errors  }  }  This would produce the following result −  MyException caught  C++ Exception  Here, what() is a public method provided by exception class and it has been overridden by all the child exception classes. This returns the cause of an exception. |