**Logo

Description automatically generated San Francisco Bay University**

**CS360L - Programming in C and C++ Lab**

**2022 Summer Final Exam**

**Student Name: Student ID:**

1. Fill in the blanks in each of the following statements:
   1. \_\_\_\_\_\_\_\_\_\_is a form of software reuse in which new classes absorb the data and behaviors of existing classes and embellish these classes with new capabilities.
   2. A base class’s \_\_\_\_\_\_\_\_\_members can be accessed in the base-class definition, in derived class definitions and in *friends* of the base class its derived classes.
   3. In a(n)\_\_\_\_\_\_\_\_\_ relationship, an object of a derived class also can be treated as an object of its base class.
   4. In a(n) \_\_\_\_\_\_\_\_\_relationship, a class object has one or more objects of other classes as members.
   5. In single inheritance, a class exists in a(n)\_\_\_\_\_\_\_\_\_relationship with its derived classes.
   6. A base class’s\_\_\_\_\_\_\_\_\_ members are accessible within that base class and anywhere that the program has a handle to an object of that class or one of its derived classes.
   7. A base class’s *protected* access members have a level of protection between those of *public* and \_\_\_\_\_\_\_\_\_ access.
   8. C++ provides for\_\_\_\_\_\_\_\_\_, which allows a derived class to inherit from many base classes, even if the base classes are unrelated.
   9. When an object of a derived class is instantiated, the base class’s\_\_\_\_\_\_\_\_\_ is called implicitly or explicitly to do any necessary initialization of the base-class data members in the derived-class object.
   10. When deriving a class with *public* inheritance, *public* members of the base class become \_\_\_\_\_\_\_\_\_ members of the derived class, and *protected* members of the base class become \_\_\_\_\_\_\_\_\_ members of the derived class.
   11. When deriving a class from with *protected* inheritance, *public* members of the base class become \_\_\_\_\_\_\_\_\_ members of the derived class, and *protected* members of the base class become \_\_\_\_\_\_\_\_\_ members of the derived class.
2. State whether each of the following is *true* or *false*. If *false*, explain why.
   1. Base-class constructors are not inherited by derived classes.
   2. A *has-a* relationship is implemented via inheritance.
   3. A *Car* class has an *is-a* relationship with the *SteeringWheel* and *Brakes classes*.
   4. Inheritance encourages the reuse of proven high-quality software.
   5. When a derived-class object is destroyed, the destructors are called in the reverse order of the constructors.
3. Fill in the blanks in each of the following statements:
   1. Treating a base-class object as a(n) \_\_\_\_\_\_\_\_\_can cause errors.
   2. Polymorphism helps eliminate \_\_\_\_\_\_\_\_\_ logic.
   3. If a class contains at least one pure *virtual* function, it’s a(n) \_\_\_\_\_\_\_\_\_class.
   4. Classes from which objects can be instantiated are called \_\_\_\_\_\_\_\_\_classes.
   5. Operator\_\_\_\_\_\_\_\_\_ can be used to downcast base-class pointers safely.
   6. Operator *typeid* returns a reference to a(n) \_\_\_\_\_\_\_\_\_ object.
   7. \_\_\_\_\_\_\_\_\_involves using a base-class pointer or reference to invoke *virtual* functions on base-class and derived-class objects.
   8. Overridable functions are declared using keyword\_\_\_\_\_\_\_\_\_.
   9. Casting a base-class pointer to a derived-class pointer is called\_\_\_\_\_\_\_\_\_.
4. State whether each of the following is *true* or *false*. If *false*, explain why.
   1. All virtual functions in an abstract base class must be declared as pure virtual functions.
   2. Referring to a derived-class object with a base-class handle is dangerous.
   3. A class is made abstract by declaring that class virtual.
   4. If a base class declares a pure virtual function, a derived class must implement that function to become a concrete class.
   5. Polymorphic programming can eliminate the need for switch logic.
5. State which of the following are *true* and which are *false*. If *false*, explain why.
   1. Keywords *typename* and *class* as used with a template type parameter specifically mean “any user-defined class type.”
   2. A function template can be overloaded by another function template with the same function name.
   3. Template parameter names among template definitions must be unique.
   4. Each member-function definition outside its corresponding class template definition must begin with *template* and the same template parameters as its class template.
6. Fill in the blanks in each of the following:
   1. Templates enable us to specify, with a single code segment, an entire range of related functions called \_\_\_\_\_\_\_\_\_, or an entire range of related classes called\_\_\_\_\_\_\_\_\_.
   2. All template definitions begin with the keyword\_\_\_\_\_\_\_\_\_, followed by a list of template parameters enclosed in\_\_\_\_\_\_\_\_\_.
   3. The related functions generated from a function template all have the same name, so the compiler uses\_\_\_\_\_\_\_\_\_ resolution to invoke the proper function.
   4. Class templates also are called\_\_\_\_\_\_\_\_\_ types.
   5. The \_\_\_\_\_\_\_\_\_ operator is used with a class-template name to tie each member-function definition to the class template’s scope.
7. (Find and Correct Code Errors) Identify the error in each of the following statements and explain how to correct it.
   1. *cout << "Value of x <= y is: " << x <= y;*
   2. The following statement should print the integer value of 'c'.

*cout << 'c';*

* 1. *cout << ""A string in quotes"";*

1. (Show Outputs) For each of the following, show the output and EXPLAIN why.
   1. *cout << "12345" << endl;*

*cout.width( 5 );*

*cout.fill( '\*' );*

*cout << 123 << endl << 123;*

* 1. *out << setw( 10 ) << setfill( '$' ) << 10000;*
  2. *cout << setw( 8 ) << setprecision( 3 ) << 1024.987654;*
  3. *cout << showbase << oct << 99 << endl << hex << 99;*
  4. *cout << 100000 << endl << showpos << 100000;*
  5. *cout << setw( 10 ) << setprecision( 2 ) << scientific << 444.93738;*

1. Write a simple function template for predicate function *isEqualTo* that compares its two arguments of the same type with the equality operator (*==*) and returns *true* if they are equal and *false* otherwise. Use this function template in a program that calls *isEqualTo* only with a variety of fundamental types. Now write a separate version of the program that calls *isEqualTo* with a user-defined class type, but does not overload the equality operator.

What happens when you attempt to run this program? Now overload the equality operator (with the operator function) operator *==.* Now what happens when you attempt to run this program?