CS 2B Final Exam

Due Mar 24 at 11:59pm **Points** 39.9 **Questions** 35
Available Mar 24 at 12am - Mar 24 at 11:59pm about 24 hours
Time Limit 120 Minutes

Instructions

Finish this test before midnight. Once you begin, you will have 120 minutes to complete. If you do not submit before that time your incomplete exam will be automatically submitted as is.

You can look at lectures or texts and use IDEs. But you may not consult any other individuals or Internet sites for help. Attempting this test implies that you agree to abide by the Foothill honor code that forbids cheating.

Each question is worth 1.14 points, for an exam total of about 40 points.

Multiple choice questions with square check-boxes may have more than one correct answer. Multiple choice questions with round radio-buttons have only one correct answer.

Any code fragments you are asked to analyze are assumed to be contained in a program that has all the necessary variables defined and/or assigned.

Best of luck,

&

Attempt History

	Attempt	Time	Score
LATEST	Attempt 1	118 minutes	25.94 out of 39.9

ⓘ Correct answers are hidden.

Score for this quiz: **25.94** out of 39.9
Submitted Mar 24 at 11:05am
This attempt took 118 minutes.

Incorrect

Question 1 0 / 1.14 pts

Assume `p` is a ***private pointer member*** of class `DeepClass` that gets assigned ***dynamically allocated data*** in one or more of `DeepClass`'s instance methods. `p` ***controls*** this dynamically allocated memory.

Check all that apply to the client statement `myDeepB = myDeepA;` between two `DeepClass` objects.

- [ ] The ***default assignment operator*** that C++ provides will result the dynamically allocated memory to be duplicated and copied over to `myDeepB`.
- [ ] ***The default assignment operator that C++ provides*** will result in a compiler error.
- [ ] ***The default assignment operator that C++ provides*** will cause ***both*** `myDeepA`'s and `myDeepB`'s "`p`-controlled" memory be to simultaneously modified if `myDeepA`'s is used to modify its copy of that memory (say, immediately after the assignment statement).
- [x] ***The default assignment operator that C++ provides*** will immediately crash the program when the assignment statement is encountered during run-time.
- [x] If `DeepClass` has a ***user-defined destructor*** that deallocates `p`'s memory, but ***no overloaded assignment operator*** is provided, then the assignment statement, followed immediately by both objects going out-of-scope (as in the program or method in which both objects are defined, ending), will cause a run-time err

The default C++ assignment operator will only copy the pointer, not the memory, so nothing is duplicated in that case. No compiler error and no program crash at the assignment statement itself -- it is legal. However, if a destructor deletes the memory, then the there will be a run-time problem when the second object goes out of scope, since the destructor will try to delete the same memory twice. Finally, since both object's p-memory are identical (not cloned) either object modifying it will cause both object's p-memory to be modified.

Question 2

1.14 / 1.14 pts

If `p` is a pointer member of class `DeepClass` that gets assigned dynamically allocated data in one or more of `DeepClass`'s instance methods then the memory to which `p` controls is considered to be technically "***part of the*** `DeepClass` ***objects***."

- ○ True
- ◉ False

We said that it **is not part of the object** in the course modules.

Incorrect

Question 3

0 / 1.14 pts

If a Boolean function does not have a simple algebraic or logical description then it is most easily implemented by a method that has which of the following?

- A ***for*** loop.
- An ***array***.
- A long ***if*** statement.
- Several user ***input*** statements with appropriate ***prompts*** to the user.
- A statement that uses ***multiplication*** and ***division***.

We saw in the modules that an array is the right choice.

Incorrect

Question 4

0 / 1.14 pts

C++ STL **list**s have which of the following properties?

(Check all that apply.)

- [x] They enable us to insert new. randomly generated, items into the list in some pre-determined order, without having to supply our own client-designed logic or code.
- [ ] They do not have STL iterators associated with them.
- [x] They are supplied to us as a template class, meaning we can declare, in one statement, a list of user-defined class data just as easily as if we were to define a list of some primitive type.
- [ ] They do not require that any ordering relation (like an operator<()) be defined for the underlying data type.

STL lists have no implicit ordering relation imposed on their type parameter. One consequence is that there is no built-in ability to create sorted lists. We, as clients, would have to write such insert() or remove() code ourselves.

Question 5

1.14 / 1.14 pts

Method ***overriding*** happens when ...

- (●) ... a derived class method of the same name as a method in the base class takes the same number and type of parameters as the base class method.
- ( ) ... a derived class method of the same name as a method in the base class takes a different number or type of parameters as the base class method.

Overriding implies the same parameters. When parameters differ, the concept is called *overloading*.

Partial

Question 6

0.76 / 1.14 pts

Searching an array using a ***binary search*** (check all that apply):

- [x] ... requires a pre-sorted array in order to work.
- [ ] ... is usually slower for each search than a simple linear search.

- [x] ... is usually faster for each search than a simple linear search.
- [ ] ... requires more code and logic than a simple linear search.

A binary search takes less time for each search than a simple linear seach, but requires that the array be pre-sorted. The algorithm is much more complex than a linear search, requiring a recursive method call.

Question 7

1.14 / 1.14 pts

Consider the following statement:

```
ifstream galSource("galaxyData.txt");
```

Check the true statements (there may be more than one):

- [ ] If the file is successfully opened, and it is a plain text file, then we would use the statement `getline(cin, someStringVar);` to read a line from it.
- [x] If the file it wants to access has too restrictive permissions, as set by the file owner or administrator, then the variable **galSource** will be assigned the value = **NULL** (i.e., 0).
- [ ] If the file it wants to access is not in the executable path (usually the same director as the program), then the variable **galSource** will be assigned the value = **-1**.
- [ ] It attempts to open a file named "**galSource**" for reading/input.
- [x] It attempts to open a file named "**galaxyData.txt**" for reading/input.

- [x] If the file is successfully opened, and it is a plain text file, then we would use the statement `getline(galSource, someStringVar);` to read a line from it.

- [ ] It attempts to open a file named "**galaxyData.txt**" for writing/output.

Question 8

1.14 / 1.14 pts

Check the true statement out of the two statements below (there is only one correct choice):

- ( ) Without type coercion being employed, a base class pointer can only point to a base class object, not a derived class object.

- (•) Without type coercion being employed, a derived class pointer can only point to a derived class object, not a base class object.

The answers are true by definition.

Question 9

1.14 / 1.14 pts

Consider the boolean expression (where **XOR** is the exclusive-or operator defined in the modules or throughout the internet):

(P **OR** Q) **XOR** (P **AND** Q)

Match the inputs P and Q with the output that this expression produces (assume 0 = false and 1 = true).

P = 1, Q = 0 1

P = 1, Q = 1 0

Here are all the combinations:

P = 0, Q = 0:
(0 OR 0) XOR (0 AND 0) = 0 XOR 0 = 0

P = 1, Q = 1:
(1 OR 1) XOR (1 AND 1) = 1 XOR 1 = 0

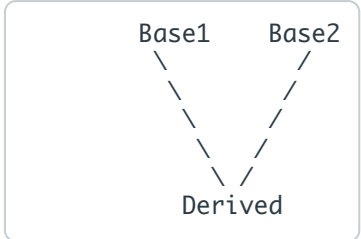
P = 0, Q = 1:
(0 OR 1) XOR (0 AND 1) = 1 XOR 0 = 1

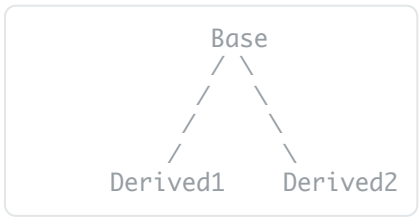
P = 1, Q = 0:
(1 OR 0) XOR (1 AND 0) = 1 XOR 0 = 1

Question 10

1.14 / 1.14 pts

Which diagram illustrates multiple inheritance?





Question 11

1.14 / 1.14 pts

Which describe a design in which **B** is a derived class of base class **A**? (Check all that apply.)

- [ ] **Class A** is **Building** that shelters people and equipment.

  Class B is a **CityVisualizer** which will render one or more **Building**s in 3D on a computer or smart phone screen.

- [x] **Class A** is an **AutoPart** which goes into a car or truck. It has a part number, weight and price.

  Class B is a **FuelInjector**, a special kind of part that goes into cars and trucks.

- [ ] **Class A** is an **AutoPart** which goes into a car or truck. It has a part number, weight and price.

  Class B is a **Automobile** -- either a **Car** or a **Truck**.

- [x] **Class A** is **Building** that shelters people and equipment.

  Class B is a **HighRise**, which has a minimum of five floors (stories) with stairs and elevators and utilized by government agencies, condo complexes and companies.

A derived class ***is a*** base class object. This informs the answers.

Question 12

1.14 / 1.14 pts

Match the concept with the phrase that best describes it.

Stack data Structure	... uses push() and po
A Queue data structure	... is first-in, first-out (I
A Linked List data structure	... typically uses dyna
A simple Array	... allows fast random

Question 13

1.14 / 1.14 pts

This is a ***sum-the-data-in-the-tree question***. It asks whether a method, **sumAll()** is a ***well-written recursive method***. You will see three different versions of this question throughout the exam, but the opening assumptions, *are identical for all such versions*. The only difference between the various questions is the code that implements the method **sumAll()**.

Assumptions:

- The general tree in this problem is assumed to be ***physical***, i.e., there is ***no lazy or soft deletion*** designed into this tree.

- We are considering a **recursive work-horse method to sum all the (assumed) integer data** of the sub-tree.
- The ***sub-tree is specified by the root pointer*** passed in. As usual, some client would pass a **root** to this method, then this recursive method would generate other (child or sibling) **root**s to pass to itself when recursing.
- The members **sib** and **firstChild** have the same meanings as in our modules.

True of False:

> ***The method, as defined below, is a good recursive method for summing the data of the sub-tree, based at the root node passed in.***

To be true, it must satisfy ***all*** the following criteria. If it misses one, it is false:

1. It ***gives the right sum*** for the sub-tree, that is, it does not miss counting any data.
2. it ***does no unnecessary work***, micro-management or superfluous testing.
3. It ***covers all situations*** (empty trees, NULL roots, handles all the children, etc.).

```
int TreeClass::sumAll(Node *root)
{
   int sibSum, thisSum, childrenSum;

   if (root == NULL)
      return 0;

   sibSum = sumAll(root->sib);
   childrenSum = sumAll(root->firstChild);
   thisSum = root->data;

   return childrenSum + sibSum + thisSum;
}
```

HINT: There are three true-false questions that start out the same in this exam, but each has a different method definition. You may want to come back and review your answer to this question after seeing the other method definitions. Only one of the three is true, and the other two are false.

- ◉ True

False

This is a perfect recursive method for summing the data in the tree. It does no extraneous testing and uses recursion efficiently.

Question 14

1.14 / 1.14 pts

It the client requests the removal of a single node in a general tree (a tree with no limit on the number of children per node) as we have studied it, this might result in many more nodes than the one requested to be removed (from the client's perspective).

While the answer will the same for normal and lazy deletion, you can assume this tree does normal, *not lazy*, deletion, to avoid unnecessary thinking time.

True

False

If a node is removed, there is no natural way to preserve its children as active parts of the tree, so these nodes, if they exist, will be removed from the tree in addition to the requested node.

Partial

Question 15

0.57 / 1.14 pts

Assume `p` is a ***private pointer member*** of class `DeepClass` that gets assigned ***dynamically allocated data*** in one or more of `DeepClass`'s

instance methods. `p` ***controls*** this dynamically allocated memory. Check all that apply.

- [ ] Memory that `p` controls must be allocated at object construction.
- [ ] The memory that `p` controls will be deallocated when a `DeepClass` object goes out of scope automatically by ***C++'s default destructo***r.
- [x] A ***user-defined destructor*** of `DeepClass` can deallocate the memory that `p` controls.
- [x] A ***non-destructor*** instance method of `DeepClass` ***may not*** deallocate the memory that `p` controls.
- [x] The memory that `p` controls will be deallocated when a `DeepClass` object goes out of scope if an appropriately written ***user-defined destructor*** is provided for `DeepClass`.

C++ default destructors will not deallocate deep memory. Any instance method, including a destructor, might correctly deallocate this memory.

`p`'s memory might not be allocated until an object's mid-life if the class designer decides that this is the best time to do it, as long as all methods work in concert to make sure there are no memory leaks. For example the constructor might only assign `NULL` to `p`, and only later will it get `new`-ed memory.

Question 16

1.14 / 1.14 pts

Match each data structure with its best description.

Data is a LIFO: There is only one data item that is naturally fetched (returned) to the client at any given moment, and it is the item most recently stored.	Stack
Data is both LIFO and FIFO: The are potentially two data items that could be naturally fetched (returned) to the client at any given moment, either the the item most recently stored or the oldest (earliest) item stored.	Deque
Data is neither LIFO nor FIFO: Client can naturally fetch (return) any of the many data items stored.	Array
Data is a FIFO: There is only one data item that is naturally fetched (returned) to the client at any given moment, and it is the oldest (earliest) item stored.	Queue

Question 17

1.14 / 1.14 pts

Lazy Deletion, *as we have implemented it* in a general tree, is good for ...

(Check all that apply.)

- [ ] ... conserving memory.
- [x] ... fast removal.
- [x] ... avoiding garbage collection for a small number of removal calls.
- [ ] ... fast insertion.

Lazy deletion allows for a fast and easy removal, but does not improve insertion as we implemented it, and may even make insertion slightly more complicated than it otherwise would be (even if we changed insertion to attempt to better utilize the deleted member). It does not make the memory of the tree any more efficient. In fact it requires more memory since we are not releasing dead nodes unless and until we do a garbage collection. It does reduce the garbage collection, by its very definition, limiting garbage collection to points in time when the the client requests it.

Partial

Question 18

0.76 / 1.14 pts

A C++ **stl vector** (check all that apply) ...

- [ ] ... is a type of ***constructor***.
- [ ] ...can be accessed using notation practically identical to (or identical to) a simple array.
- [x] ... is a type of ***container***.
- [x] ... is a type of pre-defined (in stl) template class.

Incorrect

Question 19

0 / 1.14 pts

MyClass has an internal 2-D array of dynamically allocated doubles, pointed to by the member named **data**:

```
class MyClass
{

private:
   double  **data;
   int width, height;

   // other stuff
}
```

Assume **width** and **height** are set by constructors and mutators, and the class deals with them and all other data management correctly. Here is the method we are interested in analyzing:

```
void MyClass::allocateDynArray(int newHeight, int newWidth)
{
   int row;

   if ( !valid( newHeight, newWidth ) )
      return;
   height = newHeight;
   width = newWidth;

   // delete and NULLs the data
   deallocateDynArray();

   data = new double*[height];
   for ( row = 0; row < height; row++ )
      data[row] = new double[width];

   setArrayToAllZeros();
}
```

Check the true statements (there will be one and possible more):

- [ ] It's fine as is.
- [x] The invocation of **deallocateDynArray()** needs to be repositioned to a different place in the code in order to avoid a potential crash or memory leak.

- [x] **setArrayToAllZeros()** needs parameters to know what bounds to use for its (likely) internal loops.
- [ ] A ***destructor*** is essential for this class.
- [ ] We have to set **data = NULL** before we do our error return near the top

As is, the method loses its old **width** and **height** information before the call to **deallocate()** so that method would end up using the new values of **width** and **height**, which will usually be different from the correct, older, values that were used when the old data was allocated. We have to deallocate using the same sizes that were used during allocation. The other items should be self evident.

Partial

Question 20

0.86 / 1.14 pts

Assume **MyClass** defines two nested exception classes, **BadNumber** and **BadString**. Also, assume that **MyClass**'s method **doSomething()** throws these exceptions under certain error conditions, and returns without throwing anything when it decides no error has occurred. We call **doSomething()** four times in a try/catch block:

```
MyClass  a;

try
{
   a.doSomething();
   a.doSomething();
   a.doSomething();
   a.doSomething();
}
catch (MyClass::BadNumber)
{
   cout << "\n bad number \n"  << endl;
}
catch (MyClass::BadString)
{
   cout << "\n bad string \n"  << endl;
```

```
}
catch (...)
{
   cout << "\n just generally bad\n" << endl;
}
```

Check the true statements (there will be more than one correct answer).

[Ignore whether spaces or '\n' are displayed - it is the ***words*** of the **cout** statements that we are interested in.]

- [ ] It is possible ***neither*** "bad number" ***nor*** "bad string" is displayed on the screen after this block.
- [ ] It is possible ***both*** "bad number" and "bad string" is displayed on the screen after this block.
- [ ] It is possible that **doSomething()** will return ***without error*** in ***one or more*** of these invocations, yet we ***still have one or more*** of "bad number", "bad string" or "just generally bad" displayed on the screen after this block.
- [ ] It is possible ***none of*** "bad number", "bad string" or "just generally bad" is displayed on the screen after this block.
- [ ] It is possible that this block will be called without anything being displayed on the screen.

Until **doSomething()** throws an exception, it will continue to be called up to four times. So even if it fails at some point, it may return without error and later result in a throw, which prints our one of our error messages.

We can never get more than one error message, since the first throw will stop the method calls and go to a catch block.

If there is no error in any of the calls, no error message will be displayed.

Question 21

1.14 / 1.14 pts

If a base class pointer, **p**, points to a derived class instance, and both base and derived classes have a method **void doIt()** which takes no parameters, then **p->doIt()** will invoke the derived class version ...:

- ...if the derived class method is declared **virtual**, but the base is not declared virtual.
- ... *only* if *both* the base class method *and* the derived class method are declared **virtual**.
- ... if the base class method is declared **virtual**, even if the derived class method is not declared virtual.

Question 22

1.14 / 1.14 pts

Consider the boolean expression (where **XOR** is the exclusive-or operator defined in the modules or throughout the internet):

(P **OR** Q) **XOR** (P **AND** Q)

Check the (one) true statement.

- This expression is equivalent to the simpler expression P **OR** Q
- This expression is equivalent to the simpler expression P **AND** Q
- This expression cannot be simplified to any of the options, above.
- This expression is equivalent to the simpler expression P **XOR** Q

Here are all the combinations:

P = 0, Q = 0:
(0 OR 0) XOR (0 AND 0) = 0 XOR 0 = 0

P = 1, Q = 1:
(1 OR 1) XOR (1 AND 1) = 1 XOR 1 = 0

P = 0, Q = 1:
(0 OR 1) XOR (0 AND 1) = 1 XOR 0 = 1

P = 1, Q = 0:
(1 OR 0) XOR (1 AND 0) = 1 XOR 0 = 1

So, the output is 1 if and only if exactly one of P and Q is 1 (and the other is 0). That's the same as XOR.

Partial

Question 23

0.57 / 1.14 pts

Assume that a class, **Deep**, has a pointer member

```
int *somePointer;
```

which will be assigned some dynamically allocated (deep/heap) memory during object construction (in the constructor).

Assume, further, that we have ***not overloaded*** the assignment operator for class **Deep**.

Two **Deep** objects, **deepA** and **deepB** are involved in an assignment statement:

```
deepB = deepA;
```

Check the true statements:

(There may be more than one.)

- [x] This is a ***shallow copy***; deep memory controlled by **deepA** is not copied and pointed to by **deepB**.
- [ ] **deepB**'s **somePointer** will have its own copy of int memory associated with its **somePointer** member after the assignment statement (distinct from **deepA**'s)
- [x] **deepB**'s **somePointer** will be undefined (unaffected) by this assignment statement.
- [x] After this assignment, both **deepA** and **deepB** will have their respective **somePointers** pointing to the same exact memory.

deepB's **somePointer** will be assigned the value of **deepA**'s **somePointer**, since C++ object assignment does a member-wise shallow copy, by default. So **deepB**'s **somePointer** will be defined/oeverwritten, but it won't have its own dedicated memory associated with that pointer. It will share **deepA**'s memory. The address of the dynamically allocated memory is stored in **deepA**, so that address is what gets copied. Incidentally, it may create a memory leak, since, by hypothesis, the constructor has allocated memory for **deepB**'s **somePointer**, and unless this has been deleted prior to this assignment, that memory may be inaccessible after the pointer is overwritten.

Incorrect

Question 24

0 / 1.14 pts

Bob defines a base class, `Base`, with instance method `func()`, but ***does not*** declare it to be `virtual`.

Alicia (who has no access to Bob's source code) defines three derived classes, `Der1`, `Der2` and `Der3`, of the base class and overrides `func()`.

Ying has no access to Bob or Alicia's source code and does not know anything about the derived classes, including their names or how many derived classes exist. Ying ***does know*** about the `Base` class and knows about its public instance function `func()`. She is also aware that there may be derived classes that override `func()`.

Ying uses a `Base` ***pointer,*** `p`, to loop through a list of `Base` object pointers. Ying is aware that the list pointers may point to `Base` objects or some subclass objects of `Base`.

Check all that apply.

- [ ] Ying can call `p->func()` on each object in the loop and get distinct behavior as defined by the object pointed to.
- [x] Ying can call `p->func()` on each object in the loop and get distinct behavior as defined by the object pointed to if she uses type coercion to assist her.
- [ ] Ying can call `p->func()` on each object in the loop and get no compiler error.
- [ ] Since Ying's loop pointer, `p`, is type `Base`, no compiler error will occur.
- [x] Ying can call `p->func()` on each object in the loop (no coercion) and will always get the Base behavior of this function.

Ying can use the base class pointer to loop through the list of pointers. However, since she knows nothing about the derived classes, she can't use type coercion to get at the specific class methods. Furthermore, since the base class does not declare the function virtual, only the base class behavior will be seen by the client and user.

Partial

Question 25

0.76 / 1.14 pts

Select the derived class constructor definitions that represent actual constructor chaining (never mind the reason or effect, as long as it demonstrates a derived class constructor chaining to a base class constructor). There is more than one correct answer.

- [ ]
```
SubClass::SubClass(int a)
{
   BaseClass(a);
   // other stuff
}
```

- [ ]
```
SubClass::SubClass(int a)
{
   BaseClass::BaseClass(a);
   // other stuff
}
```

- [x]
```
SubClass::SubClass(int a) : BaseClass()
{
   // other stuff
}
```

- [x]
```
SubClass::SubClass() : BaseClass(4)
{
   // other stuff
}
```

```
SubClass::SubClass(int a) : BaseClass(a)
{
   // other stuff
}
```

While **BaseClass::BaseClass()** might compile, this has nothing to do with the derived class object being constructed and is not chaining. **BaseClass::BaseClass()** creates a temporary anonymous method call that immediately disappears once the statement is complete. Same with **BaseClass()** inside the method body, which will not even compile in some compilers. Only the use of the : operator on the function header results in constructor chaining. You can chain to any constructor from the derived class - even if the method signature of the derived class constructor doesn't happen to match the base class constructor to which you chain. One chains to whichever constructor does the right thing for your desired outcome. (This is in contrast to non-constructor chaining in which it is only, truly, chaining if you call the base class method with the same signature.)

Incorrect

Question 26

0 / 1.14 pts

Multiple Inheritance

Consider a **derived class B** that is ***multiply inherited*** from **base classes A1 and A2**. Which would be a reasonable example of classes **A1**, **A2** and **B**. (There may be more than one correct answer.)

A1 is a **Condominium** (One unit in a condo development in which a family might own and reside)
A2 is a **SingleFamilyDwelling** (One house+land in which a family might own and reside)
B is a **ResidentialProperty** (House+land, condo unit, townhome, or any other place in which a family might own and reside)

A1 is a person who has some degree in math: **DegreeHolderInMath**
A2 is a person who has some degree in computer science: **DegreeHolderInCS**
B is **DegreeHolderInBothMathAndComputerScience**

A1 is a **RealEstateProperty**
A2 is an **ItemForSale**
B is a **RealEstatePropertyForSale**

A1 is an **Arm**
A2 is a **Leg**
B is a **HumanBody**

A1 is an **Insect** (animal without a backbone, body in three sections, skeleton on the outside, usually give birth as eggs)
A2 is a **Mammal** (animal with backbone, has hair/fur, usually give birth alive)
B is an **Animal** (any living being on Earth which is neither a plant (vegetable) nor fungus)

A **RealEstatePropertySale** is ***both*** a **RealEstateProperty** and **ItemForSale**, so yes, M.I. applies.
A **DegreeHolderInMathAndComputerScience** is both a **DegreeHolderInMath** and a **MSInMaDegreeHolderInComputerScience**, so yes, M.I. applies.

A **ResidentialProperty** is not (necessarily) a **Condiminium**, so no it cannot be derived from it, simply or using M.I.
There is no "is a" relationship between **Leg**, **Arm** or HumanBody, so inheritance can never apply.
An **Animal** is not (necessarily) an **Insect**, nor is it necessarily a **Mammal**, so no it cannot be derived from either one, simply or using M.I.

Incorrect

Question 27

0 / 1.14 pts

Bob defines a base class, `Base`, with instance method `func()`, ***and declares it to be*** `virtual`.

Alicia (who has no access to Bob's source code) defines three derived classes, `Der1`, `Der2` and `Der3`, of the base class and overrides `func()`.

Ying has no access to Bob or Alicia's source code and does not know anything about the derived classes, including their names or how many derived classes exist. Ying ***does know*** about the `Base` class and knows about its public instance function `func()`. She is also aware that there may be derived classes that override `func()`.

Ying uses a `Base` ***pointer,*** `p`, to loop through a list of `Base` object pointers. Ying is aware that the list pointers may point to `Base` objects or some subclass objects of `Base`.

Check all that apply.

- [x] Ying can call `p->func()` on each object in the loop (no coercion) and will always get the Base behavior of this function.
- [ ] Ying can call `p->func()` on each object in the loop and force only base class behavior of that method by using type coercion.
- [ ] Ying can call `p->func()` on each object in the loop and get distinct behavior as defined by the object pointed to.
- [x] Since Ying's loop pointer, `p`, is type `Base`, no compiler error will occur.

Ying can use the base class pointer to loop through the list of pointers. Since the base class ***does*** declare the function **virtual**, she will get distinct behavior for each method call without any type coercion. However, if Ying type coerces the object pointed to in some way, e.g., something along the lines of**((Base)(*p)).func()**, then she will be forcing the object to be considered only a **Base** object and get only the **Base** behavior of **func()** regardless of what it points to.

Question 28

1.14 / 1.14 pts

The following are advantages of a **linked-list** over an an **array**:

(Check all that apply.)

- [x] Inserting an item into an already-ordered linked-list, so as to preserve the order, requires fewer loops and/or a shorter loop than inserting an item into an ordered array so as to preserve the order.

- [ ] A linked-list takes less memory to store a single element (a **Node** in the list) than an array takes to store its version of a single element (the ***k***th item in the array).

- [ ] Even if we don't have a pointer to the proper node, a linked-list can easily access any element in the middle of the list with a single statement, but an array requires a loop to gain access to that node.

- [x] Assuming we have a pointer to the proper node, a linked-list will enable fast insertion into the middle of the list, while insertion into the middle of an array, even if we know the int index of the desired insertion point, will not be as fast or simple.

If we have a pointer to the right place in a list, we can link in a new **Node** with just a couple assignment statements, but doing this in an array requires a loop to move the old elements out-of-the-way. This also explains why adding an element into a sorted array will require a longer loop (or a second loop): after we find the position in an array, we have to continue looping to get the remaining elements out-of-the-way. Meanwhile a linked-list requires only one loop (locating the position), and once found, we can put it in, and we don't have to worry about looping over the remaining elements in the array.

A linked-list takes more memory per element since it needs next pointers on top of the data, while arrays only require space for the data for each element.

You have to iterate through a linked list to get to a random location, but an array allows instant access through the array index.

Question 29

1.14 / 1.14 pts

Assume base class has an overridden method called **methodX()**. Let **baseObject** and **derivedObject** be base class and derived class ***objects***, respectively.

Check the true statements (there will be one or more):

- [x] **derivedObject.methodX()** will directly invoke the ***derived*** class version of **methodX()**.
- [ ] **derivedObject.methodX()** will directly invoke the ***base*** class version of **methodX()**.
- [x] **baseObject.methodX()** will directly invoke the ***base*** class version of **methodX()**.
- [ ] **baseObject.methodX()** will directly invoke the ***derived*** class version of **methodX()**.

We are talking about ***objects***, not *pointers* (which we will learn can invoke derived methods from a base pointer when **virtual** is declared). Also, there is no object type coercion involved, so the objects of each type can only invoke methods of their own "level" using the **object.method()** notation. Whether or not the derived methods chain to the base class partners, the syntax **object.method()** is *directly* accessing only the method of its own class at its own "level" (i.e., not above (base) or below (derived).

Question 30

1.14 / 1.14 pts

Overloading the assignment operator can be done so that `obj = x;` can be valid, even if `x` is in a ***different class*** than `obj`.

True

False

The right hand side of an overloaded assignment operator does not have to be a member of the same class as the operator.

Incorrect

Question 31

0 / 1.14 pts

Which describe a design in which **B** is a derived class of base class **A**? (Check all that apply -- there may be one or more checked box.)

Class A is an **InventoryItem**, something a business stocks and sells to a customer. It has an item number and price.

Class B is an **MensApparelItem**, which is stocked and sold by a department store like Macy's.

Class A is a **Jacket** something that is worn to keep a person warm.

Class B is an **JacketSleeve** (something that goes into a **Jacket**).

Class A is an **Sleeve**, something that goes into a shirt, jacket or sweater.

Class B is a **Jacket**.

Class A is an **MensApparelItem**, which is stocked and sold by a department store like Macy's.

Class B is an **InventoryItem**, something a business stocks and sells to a customer. It has an item number and price.

A derived class ***is a*** base class object. This informs the answers.

Question 32

1.14 / 1.14 pts

This is a ***sum-the-data-in-the-tree question***. It asks whether a method, **sumAll()** is a ***well-written recursive method***. You will see three different versions of this question throughout the exam, but the opening assumptions, *are identical for all such versions*. The only difference between the various questions is the code that implements the method **sumAll()**.

Assumptions:

- The general tree in this problem is assumed to be ***physical***, i.e., there is ***no lazy or soft deletion*** designed into this tree.
- We are considering a **recursive work-horse method to sum all the (assumed) integer data** of the sub-tree.
- The ***sub-tree is specified by the root pointer*** passed in. As usual, some client would pass a **root** to this method, then this recursive method would generate other (child or sibling) **root**s to pass to itself when recursing.
- The members **sib** and **firstChild** have the same meanings as in our modules.

True of False:

> ***The method, as defined below, is a good recursive method for summing the data of the sub-tree, based at the root node passed in.***

To be true, it must satisfy ***all*** the following criteria. If it misses one, it is false:

1. It ***gives the right sum*** for the sub-tree, that is, it does not miss counting any data.
2. it ***does no unnecessary work***, micro-management or superfluous testing.

3. It ***covers all situations*** (empty trees, NULL roots, handles all the children, etc.).

```
int TreeClass::sumAll(Node *root)
{
   int sibSum, thisSum, childrenSum;

   if (root == NULL)
      return 0;

   // set to 0 in case we don't recurse, below.
   sibSum = childrenSum = 0;

   if (root->sib == NULL)
      sibSum = sumAll(root->sib);
   if (root->firstChild == NULL)
      childrenSum = sumAll(root->firstChild);

   thisSum = root->data;

   return childrenSum + sibSum + thisSum;
}
```

There are three true-false questions that start out the same in this exam, but each has a different method definition. You may want to come back and review your answer to this question after seeing the other method definitions. Only one of the three is true, and the other two are false.

- ○ True
- ◉ False

There is unnecessary NULL testing in the method. The sib and firstChild do not need to be tested since the recursive call will catch those cases.

Incorrect

Question 33

0 / 1.14 pts

A Boolean function that has ***four*** inputs and one output. The programmer decides to implement it using an array of type **bool**. How many elements

(what size array) gives the programmer the maximum flexibility in defining the function?

- [x] 4
- [ ] 15
- [ ] 8
- [ ] 16
- [ ] 3

Four Boolean inputs means 2^4 = 16 possible combinations, so we'd need a table of size 16.

Incorrect

Question 34

0 / 1.14 pts

A class `MyClass` is instantiated by some client `main()` dynamically, using some `MyClass` pointers local to `main()`. Check all that apply.

- [ ] It is possible that `MyClass` ***does not need any*** of the following: a user defined destructor, copy constructor or assignment operator.

- [ ] It is `MyClass`'s instance methods that are responsible for deleting the two objects instantiated by `main()`.

- [x] If no deep memory is used in `MyClass` (i.e., it contains only simple data, no pointer members), then `main()` does not need to manually delete the objects that it dynamically allocates.

- [ ] `main()` is responsible for deleting the two objects it instantiated, either directly or by calling some method that deletes them.

- [x] The situation described means that `MyClass` ***will always need*** a user defined destructor, copy constructor and assignment operator.

The fact that a client's objects are dynamically allocated does not necessarily mean that there is any deep memory in the class. So it is possible that there is no need for deep memory methods like copy constructors. Furthermore, the client, not the class, owns the pointers that have been used to ***new*** the objects, so only the client or its agents can delete them. Instance methods of the class itself have no ability to do this.

Question 35

1.14 / 1.14 pts

This is a ***sum-the-data-in-the-tree question***. It asks whether a method, **sumAll()** is a ***well-written recursive method***. You will see three different versions of this question throughout the exam, but the opening assumptions, *are identical for all such versions*. The only difference between the various questions is the code that implements the method **sumAll()**.

Assumptions:

- The general tree in this problem is assumed to be ***physical***, i.e., there is ***no lazy or soft deletion*** designed into this tree.
- We are considering a **recursive work-horse method to sum all the (assumed) integer data** of the sub-tree.
- The ***sub-tree is specified by the root pointer*** passed in. As usual, some client would pass a **root** to this method, then this recursive method would generate other (child or sibling) **root**s to pass to itself when recursing.
- The members **sib** and **firstChild** have the same meanings as in our modules.

True of False:

> ***The method, as defined below, is a good recursive method for summing the data of the sub-tree, based at the root node passed in.***

To be true, it must satisfy ***all*** the following criteria. If it misses one, it is false:

1. It ***gives the right sum*** for the sub-tree, that is, it does not miss counting any data.
2. it ***does no unnecessary work***, micro-management or superfluous testing.
3. It ***covers all situations*** (empty trees, NULL roots, handles all the children, etc.).

```
int TreeClass::sumAll(Node *root)
{
   int sibSum, thisSum, childrenSum;
   FHsdTreeNode<Object> *child;

   if (root == NULL)
      return 0;

   // set to 0 for next computations
   sibSum = childrenSum = 0;

   if (root->sib == NULL)
      sibSum = sumAll(root->sib);

   for ( child = root->firstChild; child != NULL; child = ch
ild->sib )
      childrenSum += sumAll(child);

   thisSum = root->data;
```

```
    return childrenSum + sibSum + thisSum;
}
```

There are three true-false questions that start out the same in this exam, but each has a different method definition. You may want to come back and review your answer to this question after seeing the other method definitions. Only one of the three is true, and the other two are false.

- True
- False

This method won't even give right answer, but even if did, is doing way too much micro-management. Looping over the children is unnecessary a tree recursion that makes maximum use of the recursive assumption (recursive calls will return the right thing).

Quiz Score: **25.94** out of 39.9