Full Name: _____

Student ID#: _____

CSE 1325 OBJECT-ORIENTED PROGRAMMING

Exam #3 «---» R «---» Exam #3

Instructions

1. Students are allowed pencils, erasers, and beverage only.
2. All books, bags, backpacks, phones, **smart watches**, and other electronics, etc. must be placed along the walls. **Silence all notifications.**
3. PRINT your name and student ID at the top of this page **and every coding sheet**, and verify that you have all pages.
4. **Read every question completely before you start to answer it.** If you have a question, please raise your hand. You may or may not get an answer, but it won't hurt to ask.
5. If you leave the room, you may not return.
6. You are required to SIGN and ABIDE BY the following Honor Pledge for each exam this semester.

NOTE: The number of questions in each section, and the topic of Free Response questions, may vary on the actual Final Exam.

Honor Pledge

On my honor, I pledge that I will not attempt to communicate with another student, view another student's work, or view any unauthorized notes or electronic devices during this exam. I understand that the professor and the CSE 1325 Course Curriculum Committee have zero tolerance for cheating of any kind, and that any violation of this pledge or the University honor code will result in an automatic grade of zero for the semester and referral to the Office of Student Conduct for scholastic dishonesty.

Student Signature: _____

WARNING: Questions are on the BACK of this page!

Vocabulary

Write the word or phrase from the Words list below to the left of the definition that it best matches. Each word or phrase is used at most once, but some will not be used. {10 at 2 points each}

Vocabulary

Word	Definition
1	The provision of a single interface to multiple subclasses, enabling the same method call to invoke different subclass methods to generate different results
2	Reuse and extension of fields and method implementations from another class
3	Bundling data and code into a restricted container
4	Specifying a general interface while hiding implementation details
5	A template encapsulating data and code that manipulates it

Word List

Abstract Class	Abstract Method	Abstraction	Algorithm	Class
Constructor	Container	Declaration	Definition	Destructor
Encapsulation	Exception	Field	Friend	Inheritance
Invariant	Iterator	Method	Multiple Inheritance	Namespace
Object	Operator	Operator Overloading	Override	Polymorphism
Shadowing	Standard Template Library	Subclass	Superclass	

Vocabulary

Word	Definition
1 Polymorphism	The provision of a single interface to multiple subclasses, enabling the same method call to invoke different subclass methods to generate different results
2 Inheritance	Reuse and extension of fields and method implementations from another class
3 Encapsulation	Bundling data and code into a restricted container
4 Abstraction	Specifying a general interface while hiding implementation details
5 Class	A template encapsulating data and code that manipulates it

Vocabulary

Word	Definition
6	An instance of a class containing a set of encapsulated data and associated methods
7	A short string representing a mathematical, logical, or machine control action
8	A class member variable
9	A special class member that creates and initializes an object from the class
10	A special class member that cleans up when an object is deleted

Word List

Abstract Class	Abstract Method	Abstraction	Algorithm	Class
Constructor	Container	Declaration	Definition	Destructor
Encapsulation	Exception	Field	Friend	Inheritance
Invariant	Iterator	Method	Multiple Inheritance	Namespace
Object	Operator	Operator Overloading	Override	Polymorphism
Shadowing	Standard Template Library	Subclass	Superclass	

Vocabulary

Word	Definition
6 Object	An instance of a class containing a set of encapsulated data and associated methods
7 Operator	A short string representing a mathematical, logical, or machine control action
8 Field	A class member variable
9 Constructor	A special class member that creates and initializes an object from the class
10 Destructor	A special class member that cleans up when an object is deleted

Vocabulary

Word	Definition
11	A function that manipulates data in a class
12	A class or a function that is granted access to its friend class' private class members
13	Providing a user-defined meaning to a pre-defined operatorfor a user-defined type
14	A subclass inheriting class members from two or more superclasses
15	The class from which members are inherited

Word List

Abstract Class	Abstract Method	Abstraction	Algorithm	Class
Constructor	Container	Declaration	Definition	Destructor
Encapsulation	Exception	Field	Friend	Inheritance
Invariant	Iterator	Method	Multiple Inheritance	Namespace
Object	Operator	Operator Overloading	Override	Polymorphism
Shadowing	Standard Template Library	Subclass	Superclass	

Vocabulary

Word	Definition
11 Method	A function that manipulates data in a class
12 Friend	A class or a function that is granted access to its friend class' private class members
13 Operator Overloading	Providing a user-defined meaning to a pre-defined operatorfor a user-defined type
14 Multiple Inheritance	A subclass inheriting class members from two or more superclasses
15 Superclass	The class from which members are inherited

Vocabulary

Word	Definition
16	The class inheriting members
17	A class that cannot be instantiated
18	A method declared with no implementation
19	A subclass replacing its superclass' implementation of a virtual method
20	A named scope

Word List

Abstract Class	Abstract Method	Abstraction	Algorithm	Class
Constructor	Container	Declaration	Definition	Destructor
Encapsulation	Exception	Field	Friend	Inheritance
Invariant	Iterator	Method	Multiple Inheritance	Namespace
Object	Operator	Operator Overloading	Override	Polymorphism
Shadowing	Standard Template Library	Subclass	Superclass	

Vocabulary

Word	Definition
16 Subclass	The class inheriting members
17 Abstract Class	A class that cannot be instantiated
18 Abstract Method	A method declared with no implementation
19 Override	A subclass replacing its superclass' implementation of a virtual method
20 Namespace	A named scope

Vocabulary

Word	Definition
21	A statement that introduces a name with an associated type into a scope
22	A declaration that also fully specifies the entity declared
23	A variable declared in a narrower scope than that of a variable of the same name declared in a broader scope
24	A library of well-implemented algorithms focused on organizing code and data as C++ templates
25	An object that stores and manages other objects

Word List

Abstract Class	Abstract Method	Abstraction	Algorithm	Class
Constructor	Container	Declaration	Definition	Destructor
Encapsulation	Exception	Field	Friend	Inheritance
Invariant	Iterator	Method	Multiple Inheritance	Namespace
Object	Operator	Operator Overloading	Override	Polymorphism
Shadowing	Standard Template Library	Subclass	Superclass	

Vocabulary

Word	Definition
21 Declaration	A statement that introduces a name with an associated type into a scope
22 Definition	A declaration that also fully specifies the entity declared
23 Shadowing	A variable declared in a narrower scope than that of a variable of the same name declared in a broader scope
24 Standard Template Library	A library of well-implemented algorithms focused on organizing code and data as C++ templates
25 Container	An object that stores and manages other objects

Vocabulary

Word	Definition
26	A procedure for solving a specific problem, expressed as an ordered set of actions
27	A pointer-like standard library abstraction for objects referring to elements of a container
28	An object created to represent an error or other unusual occurrence and then propagated via special mechanisms until caught by special handling code
29	Code for which specified assertions are guaranteed to be true

Word List

Abstract Class	Abstract Method	Abstraction	Algorithm	Class
Constructor	Container	Declaration	Definition	Destructor
Encapsulation	Exception	Field	Friend	Inheritance
Invariant	Iterator	Method	Multiple Inheritance	Namespace
Object	Operator	Operator Overloading	Override	Polymorphism
Shadowing	Standard Template Library	Subclass	Superclass	

Vocabulary

Word	Definition
26 Algorithm	A procedure for solving a specific problem, expressed as an ordered set of actions
27 Iterator	A pointer-like standard library abstraction for objects referring to elements of a container
28 Exception	An object created to represent an error or other unusual occurrence and then propagated via special mechanisms until caught by special handling code
29 Invariant	Code for which specified assertions are guaranteed to be true

Multiple Choice

Read the full question and every possible answer. Choose the one best answer for each question and write the corresponding letter in the blank next to the number. {15 at 2 points each}

1. ____ **To call a method polymorphically in C++,**

   A. The superclass must implement `virtual` inheritance and the call must be via a `const`

   B. The superclass method must be marked `static` and the subclass method must be marked `override`

   C. The class must inherit from at least two superclasses that are both marked `virtual`

   D. The superclass method must be marked `virtual` and the call must be via a pointer or reference

1. ____ **To call a method polymorphically in C++,**

    A. The superclass must implement `virtual` inheritance and the call must be via a `const`

    B. The superclass method must be marked `static` and the subclass method must be marked `override`

    C. The class must inherit from at least two superclasses that are both marked `virtual`

    D. The superclass method must be marked `virtual` and the call must be via a pointer or reference

CORRECT: D

2. ____ **Given superclass** `Truck` **and subclass** `F150`**, what is the surprising result of the C++ upcast** `Truck t = F150{};`**?**

   A. It will not compile without an explicit upcast operator

   B. It will segfault

   C. The `F150` object will be allocated on the heap, not the stack

   D. Variable `t` will contain an instance of `Truck`, not `F150`

2. ____ **Given superclass** `Truck` **and subclass** `F150`**, what is the surprising result of the C++ upcast** `Truck t = F150{};`**?**

    A. It will not compile without an explicit upcast operator

    B. It will segfault

    C. The `F150` object will be allocated on the heap, not the stack

    D. Variable `t` will contain an instance of `Truck`, not `F150`

CORRECT: D

3. ____ **In C++, a** `std::map` **by default is**

A. Sorted by value

B. Sorted by key

C. Sorted by hash code

D. Unsorted

3. ____ **In C++, a** `std::map` **by default is**

    A. Sorted by value

    B. Sorted by key

    C. Sorted by hash code

    D. Unsorted

CORRECT: B

4. ____ **A C++ stream will evaluate as FALSE unless the stream is in state**

A. bad

B. eof

C. good

D. All of the above

4. ____ **A C++ stream will evaluate as FALSE unless the stream is in state**

A. bad

B. eof

C. good

D. All of the above

CORRECT: C

5. ____ **When no more data can be read from a C++ input stream, its state becomes**

A. bad

B. eof

C. end

D. empty

5. ____ **When no more data can be read from a C++ input stream, its state becomes**

A. bad

B. eof

C. end

D. empty

CORRECT: B

6. ____ **A recoverable error in a C++ stream changes its state to**

A. fail

B. eof

C. good

D. bad

6. ____ **A recoverable error in a C++ stream changes its state to**

A. fail

B. eof

C. good

D. bad

CORRECT: A

7. ____ **A non-recoverable error in a C++ stream changes its state to**

A. eof

B. bad

C. end

D. fail

7. ____ **A non-recoverable error in a C++ stream changes its state to**

A. eof

B. bad

C. end

D. fail

CORRECT: B

8. ____ **The two types of iterators in C++ are the basic iterator and the**

A. List iterator

B. Const interator

C. Virtual iterator

D. Reverse iterator

8. ____ **The two types of iterators in C++ are the basic iterator and the**

A. List iterator

B. Const interator

C. Virtual iterator

D. Reverse iterator

CORRECT: B

9. ____ **The "Rule of 3" states**

A. Each method in a "version 1.0" program contains an average of 3 bugs

B. If 3 or more developers work on a program, they should use version control to coordinate their work

C. Each feature of your program should consist of at least 3 substantial commits to your git repository

D. If any of the destructor, copy constructor, or copy assignment operator are needed, all 3 are needed

9. ____ **The "Rule of 3" states**

A. Each method in a "version 1.0" program contains an average of 3 bugs

B. If 3 or more developers work on a program, they should use version control to coordinate their work

C. Each feature of your program should consist of at least 3 substantial commits to your git repository

D. If any of the destructor, copy constructor, or copy assignment operator are needed, all 3 are needed

CORRECT: D

10. ____ **In which instances would a copy constructor be called by C++?**

A. Dereferencing a pointer to an object

B. Pass-by-reference method parameters and returns

C. Pass-by-value method parameters and returns

D. Only when explicitly invoked by the programmer

10. ____ **In which instances would a copy constructor be called by C++?**

A. Dereferencing a pointer to an object

B. Pass-by-reference method parameters and returns

C. Pass-by-value method parameters and returns

D. Only when explicitly invoked by the programmer

CORRECT: C

11. ____ **Most STL containers use** `v[index] = value;` **to overwrite a value, but** `std::set` **has no index. How would you overwrite a value in a** `std::set`**?**

    A. `v.replace(old_value, value);`

    B. `v.overwrite(old_value, value);`

    C. `v.insert(value);`

    D. `v[v.find(old_value)] = value;`

11. ____ **Most STL containers use** `v[index] = value;` **to overwrite a value, but** `std::set` **has no index. How would you overwrite a value in a** `std::set`**?**

    A. `v.replace(old_value, value);`

    B. `v.overwrite(old_value, value);`

    C. `v.insert(value);`

    D. `v[v.find(old_value)] = value;`

CORRECT: C

12. ____ **In C++, the** `std::sort` **function accepts two iterators rather than a container (like** `std::vector`**) because**

    A. The iterators allow directly sorting any subset of the container

    B. Collections cannot be passed as parameters to a function

    C. Most containers are only accessible indirectly through iterators

    D. Collections already have a sort *method*, so the function would be redundant

12. ____ **In C++, the** `std::sort` **function accepts two iterators rather than a container (like** `std::vector`**) because**

    A. The iterators allow directly sorting any subset of the container

    B. Collections cannot be passed as parameters to a function

    C. Most containers are only accessible indirectly through iterators

    D. Collections already have a sort *method*, so the function would be redundant

CORRECT: A

13. ____ **In C++, a package-private method**

A. may be called by any object in the system

B. may be called only by methods in the same class or its friends

C. may be called only by methods in the same class

D. does not exist

13. ____ **In C++, a package-private method**

A. may be called by any object in the system

B. may be called only by methods in the same class or its friends

C. may be called only by methods in the same class

D. does not exist

CORRECT: D

14. ____ **C++ inheritance is different from Java inheritance in that**

A. C++ destructors may have parameters, but Java destructors never have parameters

B. C++ supports multiple inheritance of classes, but Java supports only single inheritance of classes

C. Java interfaces support multiple inheritance, but C++ interfaces only support single inheritance

D. C++ constructors inherit, but Java constructors do not inherit

14. ____ **C++ inheritance is different from Java inheritance in that**

A. C++ destructors may have parameters, but Java destructors never have parameters

B. C++ supports multiple inheritance of classes, but Java supports only single inheritance of classes

C. Java interfaces support multiple inheritance, but C++ interfaces only support single inheritance

D. C++ constructors inherit, but Java constructors do not inherit

CORRECT: B

15. ____ **Which is TRUE about enum classes in C++?**

A. An enum class does not support inheritance

B. An enum class may include fields

C. An enum class may be used as a `std::map` index

D. An enum class may include constructors

E. An enum class is a type

F. An enum class is identical to a C enum

G. An enum class cannot be compared to an integer

H. An enum class may include methods

15. ____ **Which is TRUE about enum classes in C++?**

A. An enum class does not support inheritance

B. An enum class may include fields

C. An enum class may be used as a `std::map` index

D. An enum class may include constructors

E. An enum class is a type

F. An enum class is identical to a C enum

G. An enum class cannot be compared to an integer

H. An enum class may include methods

CORRECT: ACEG

16. ____ **A difference between C++** `std::string` **and Java** `String` **is**

A. std::string is just an alias for char*, while String is a true class

B. std::string is mutable (can change value), but String is immutable (cannot change value once instanced)

C. std::string must be zero-terminated (end with a null char), while String does not

D. std::string supports Unicode characters, but String only supports 8-bit ASCII characters

16. ____ **A difference between C++** `std::string` **and Java** `String` **is**

A. std::string is just an alias for char*, while String is a true class

B. std::string is mutable (can change value), but String is immutable (cannot change value once instanced)

C. std::string must be zero-terminated (end with a null char), while String does not

D. std::string supports Unicode characters, but String only supports 8-bit ASCII characters

CORRECT: B

17. ____ **To override the** + **operator in C++ class Complex,**

A. Override method `operator+(const Complex& c)`

B. Write function `operator_add(const Complex& c)`

C. Override method `+(const Complex& c1, const Complex& c2)`

D. C++, like Java, does not support operator overloading

17. ____ **To override the** + **operator in C++ class Complex,**

A. Override method `operator+(const Complex& c)`

B. Write function `operator_add(const Complex& c)`

C. Override method `+(const Complex& c1, const Complex& c2)`

D. C++, like Java, does not support operator overloading

CORRECT: A

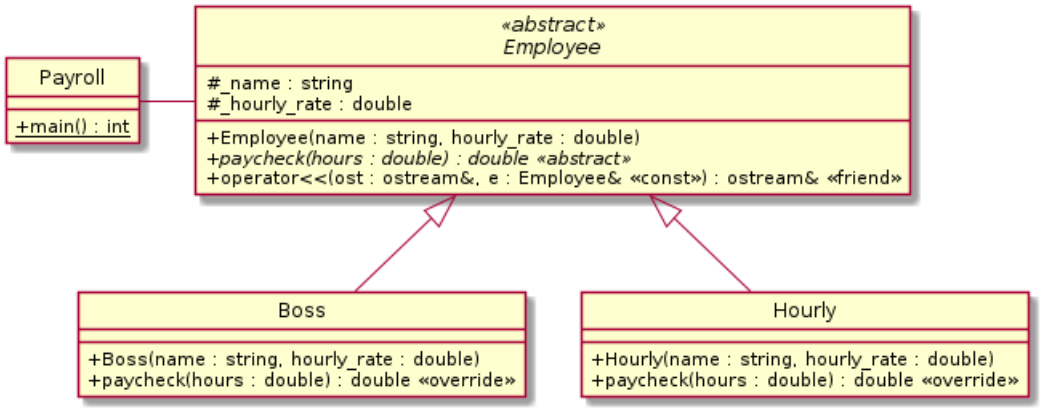
Free Response

Provide clear, concise answers to each question. Write only the code that is requested. You will NOT write an entire application! You need NOT copy any code provided to you - just write the additional code specified. You need NOT write `#include` statements - assume you have what you need. You will write a .h guard only once (question 1.a) - skip them on all other .h files to save time.

While multiple questions may relate to a given application or class diagram, **each question is fully independent and may be solved as a stand-alone problem.** Thus, if you aren't able to solve a question, skip it until the end and move on to the next.

1. (polymorphism, abstract, operator overloading, file I/O) Consider the following class diagram for a C++ application. (You will NOT write the entire application!)

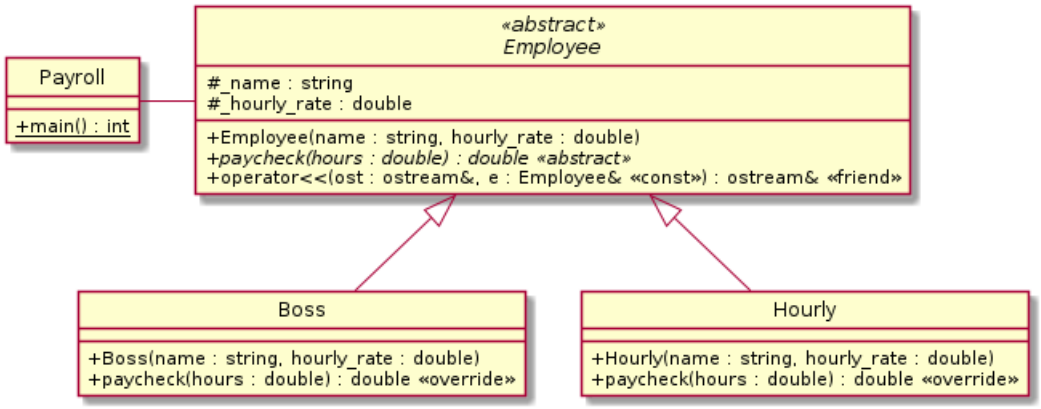
```
NOTE: The subclass names varied from exam to exam.
      The Boss subclass ($manager below) could be Manager, Supervisor, or Boss.
      The Hourly subclass ($hourly below) could be Hourly, Associate, or Contributor.
```



Class Employee represents someone who works for your company. $hourly employees are paid their hourly rate for each of the first 40 hours worked each week and 1½ x their hourly rate for each additional hour. $manager employees are paid 40 x their hourly rate regardless of the number of hours worked. `operator<<` is a friend function that overloads the << operator for these classes, printing their name and hourly rate.

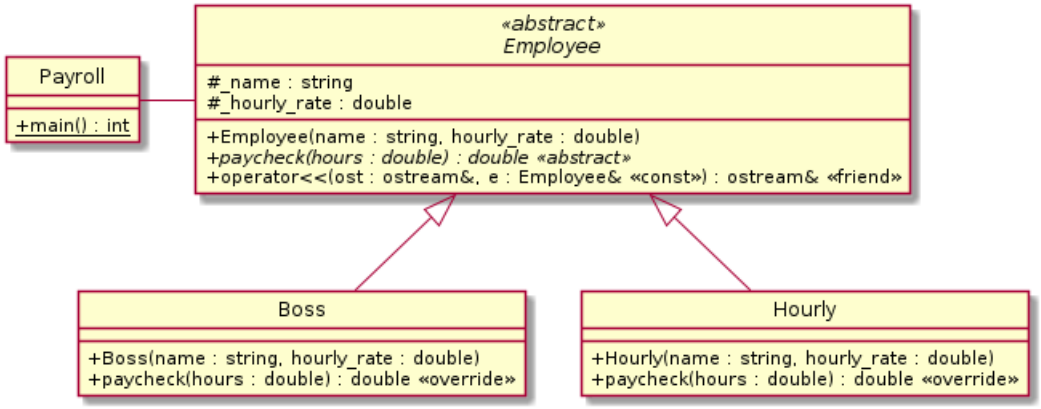
a. {3 points} In file Employee.h, write the guard, class declaration, and the protected section with its two fields. Do NOT write the rest of the class, except where requested below.

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

class Employee {
  protected:
    std::string _name;
    double _hourly_rate;

#endif
```



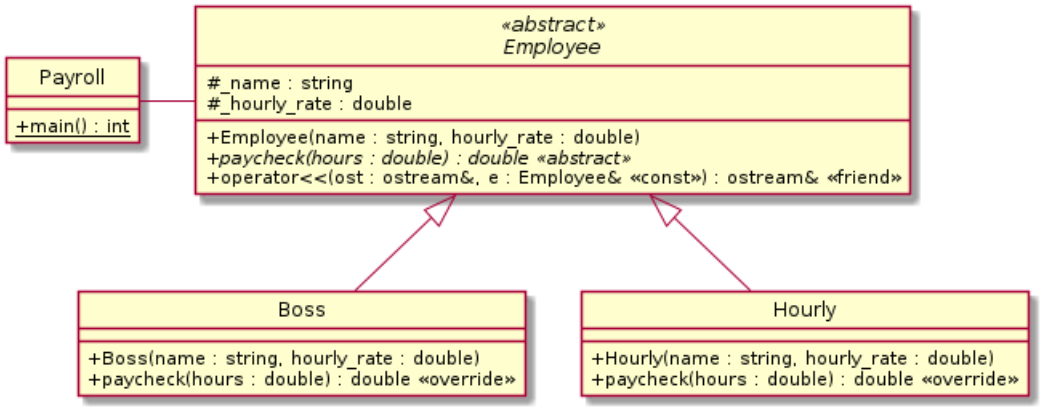
b. {2 points} In file Employee.h, write ONLY the declaration for abstract method `paycheck`.

```
virtual double paycheck(double hours) = 0;
```



c. {2 points} In file employee.cpp, write the implementation of the constructor for class Employee. Use an init list to specify construction of each field to copy the corresponding parameter. If the `hourly_rate` parameter is less than 0, throw a runtime error with your choice of message.

```
Employee::Employee(std::string name, double hourly_rate)
   : _name{name}, _hourly_rate{hourly_rate} {
   if(hourly_rate < 0) throw std::runtime_error{"Negative hourly rate"};
}
```



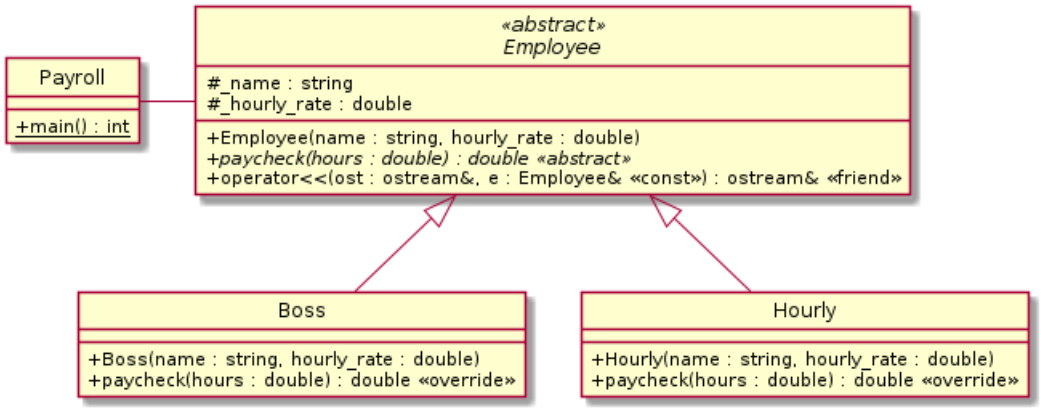
d. {2 points} In file employee.cpp, write the implementation of `operator<<`. For Employee "Prof Rice" with an hourly rate of "8.25", for example, stream out "Prof Rice ($8.25)". Use I/O manipulators (but NOT `printf`) to ensure 2 digits follow the decimal point on the salary.

```
std::ostream& operator<<(std::ostream& ost, const Employee& e) {
```

```
std::ostream& operator<<(std::ostream& ost, const Employee& e) {
   return ost << e._name << " ($" << std::fillchar('0') << std::setprecision(2)
              << std::fixed << e._hourly_rate << ")"; // fillchar is hidden bonus
}
```

```
}
```

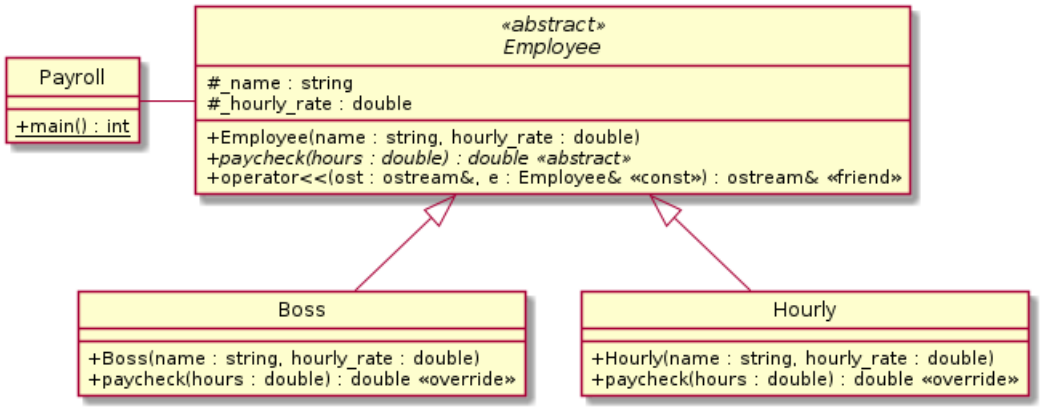
Notes: `fillchar` was NOT required and no one added it - if you did, email me (it was the secret bonus)! Relatively few students used any I/O manipulators at all, which was disappointing.



e. {2 point} In file $hourly.cpp, implement ONLY the constructor. Use an init list to construct the superclass parameters using chaining.

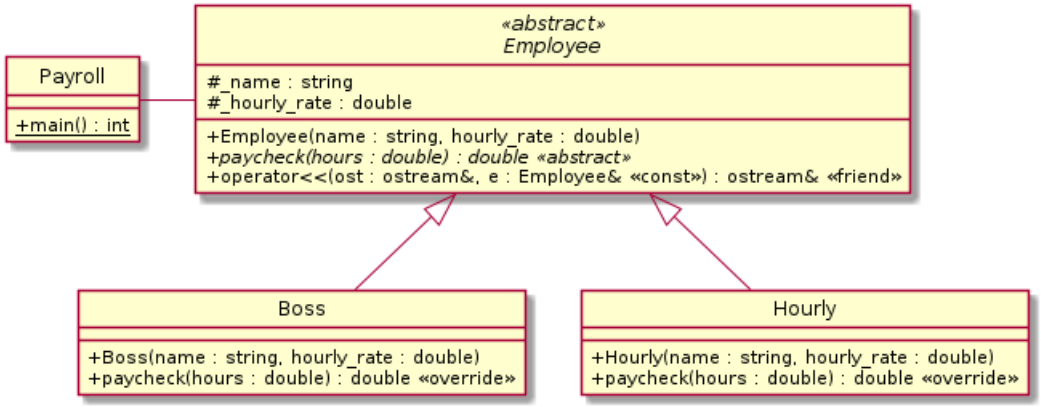
```
Hourly::Hourly(std::string name, double hourly_rate)
    : Employee(name, hourly_rate) { }
```

Note: Curly braces after `Employee` were also acceptable.



f. {1 point} In file $hourly.cpp, implement ONLY the `paycheck` method. The first 40 hours are paid at the superclass field's `_hourly_rate`, and the rest are paid at 1.5 times that amount. If `hours` are negative, pay $0.00.

```
double Hourly::paycheck(double hours) { // Equivalent algorithms are OK
   if(hours <  0) hours = 0;
   if(hours > 40) hours += (0.5 * (hours - 40));
   return hours * _hourly_rate;
}
```



In file payroll.cpp, begin writing the main function which was begun for you starting on the next page.

This program reads file "payroll.dat" and prints the payroll to standard out for both `$manager` and `$hourly` employees. The fields on each line are employee type [H, A, C] for [Hourly, Associate, Contributor] or [M, S, B] for [Manager, Supervisor, Boss], hours worked, wage, and name, each whitespace separated. For example,

```
H 55.0 15.0 Lee Chen
A 55.0 15.0 Lee Chen
C 55.0 15.0 Lee Chen

M 70.3 31.5 Ursula Garcia
S 70.3 31.5 Ursula Garcia
B 70.3 31.5 Ursula Garcia
```

When complete, the program should output something like this from the above data:

```
Lee Chen ($15.00) is paid $937.50

Ursula Garcia ($31.50) is paid $1260.00
```

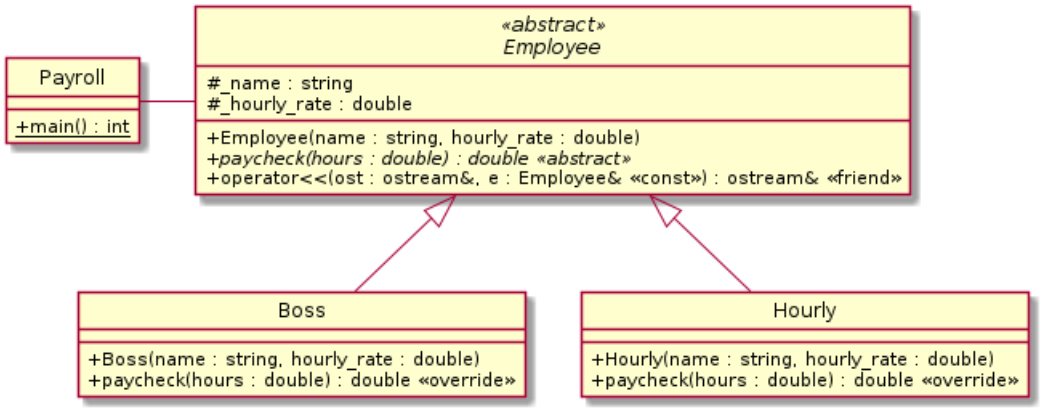
You will write the main function a few lines at a time beginning on the next page. If you can't solve one of the steps, simply skip it and continue with the next step. If you prefer to write the remainder of Free Response #1 as a single program rather than in code snippets, please simply ask for an additional sheet of paper and write it on that in its entirety.

```
#include "$hourly.h"  // $hourly may be Hourly, Associate, or Contributor
#include "$manager.h" // $manager may be Manager, Supervisor, or Boss

#include <vector>
#include <fstream>

int main() {
    std::string type, name;
    double wage, hours;
    Employee* e;
```

g. {2 points} Continuing to write the payroll.cpp main program, open file "payroll.dat" for input.

```
std::ifstream ifs{"payroll.dat"}; // (g)
```

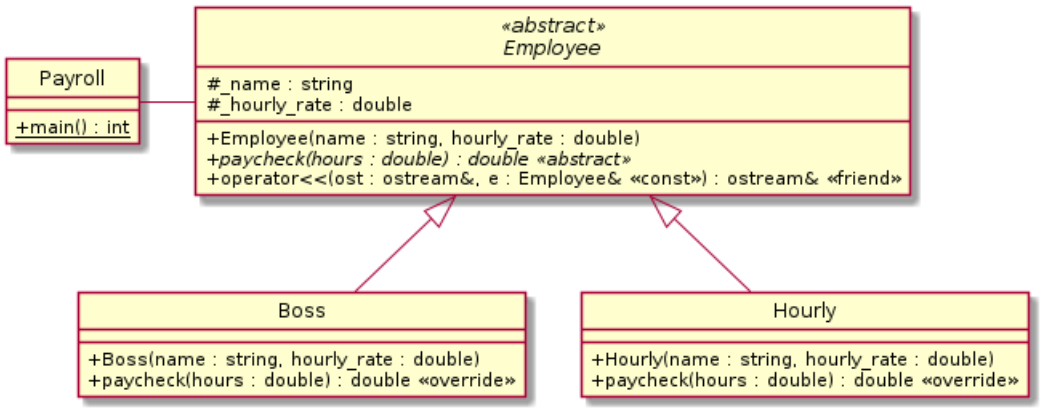
Note: Any stream name was acceptable as long as you were consistent through question 1. We accepted parentheses even though curly braces are strongly preferred here.



h. {2 points} Continuing to write the payroll.cpp main program, if the file failed to open in part (g), write "Open failed" to standard error and return an error code of -1 to the operating system.

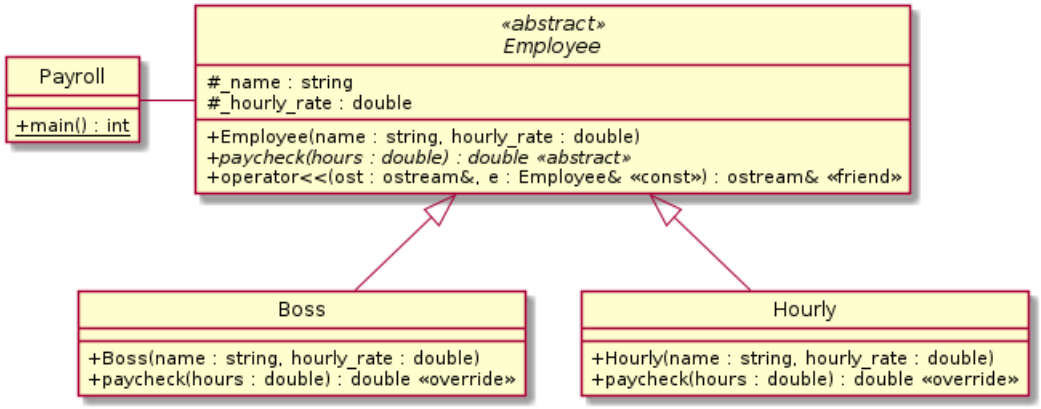
```
if(!ifs) {                          // (h)
    std::cerr << "Open failed" << std::endl;
    return -1; // or exit(-1);
}
```

Note: `if(!ifs.good())` was also acceptable.



i. {1 point} Continuing to write the payroll.cpp main program, loop while data is available from the file.

```
while(ifs) { // (i)
```



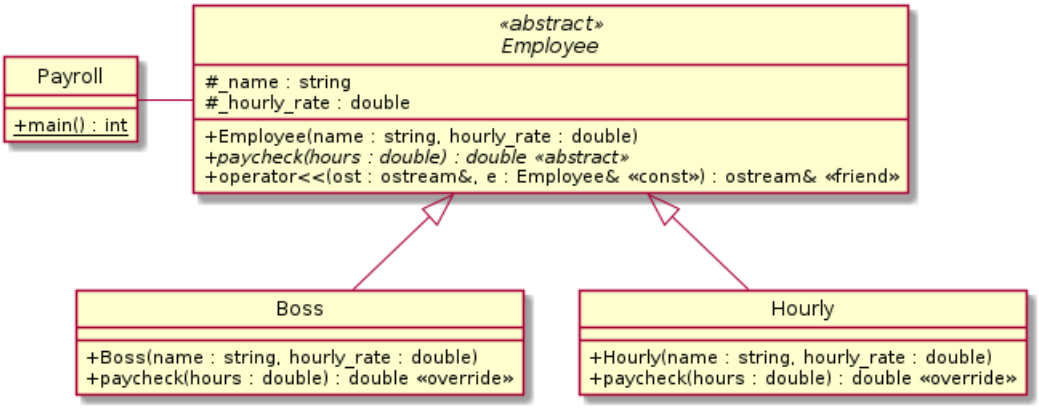
j. {3 points} Continuing to write the payroll.cpp main program, while in the loop begun in part (i), read `type`, `hours`, and `wage` (whitespace separated) and then `name` (the rest of the line to the newline) into variables defined above.

```
ifs >> type >> hours >> wage; // (j)
std::getline(ifs, name);
```

Note: We also accepted a `std::getline` into a string stream, and then parsing the string stream as above.

Note: The `std::getline` is required to read "the rest of the line to the newline" into variable `name`. Thus, `>> name` is NOT equivalent.

This was a most-missed question.

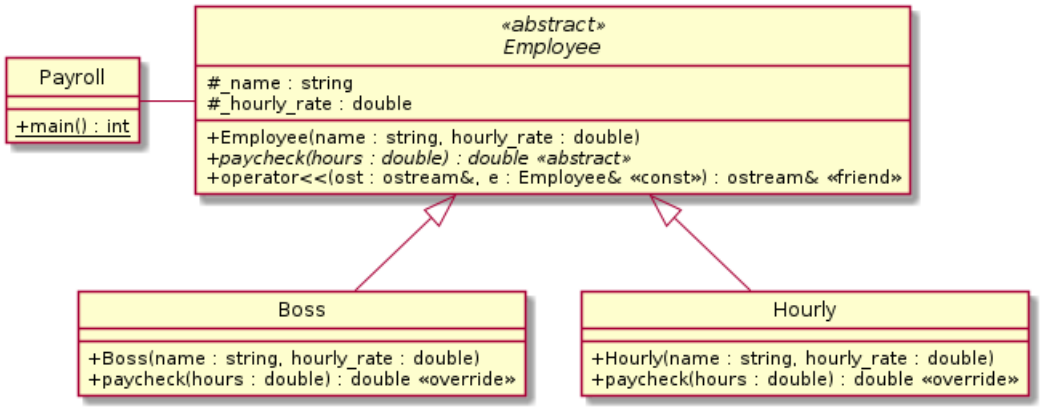


k. {2 points} Continuing to write the payroll.cpp main program, while in the loop begun in part (i), if `type` is [H, A, C] instance `$hourly` **on the heap** pointed to by a variable `e`, otherwise, instance `$manager` **on the heap** pointed to by variable `e`.

```
if(type == "H") e = new Hourly(name, wage); // (k)
else e = new Manager(name, wage);
```

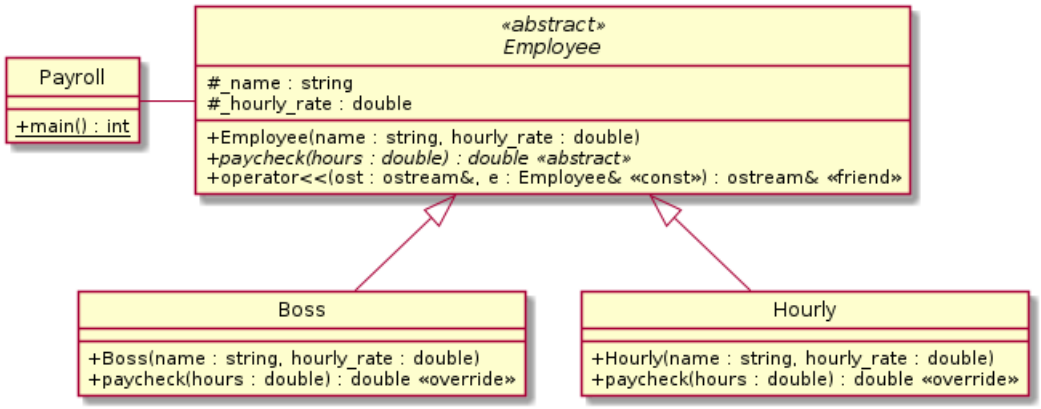
Note: We allowed you to redefine e here without penalty.

Note: A couple of students asked why we asked you to allocate memory on the heap. The answer is "To see if you know how to allocate memory on the heap". :D No engineering consideration would drive you to do this in a real program (in *this* context).



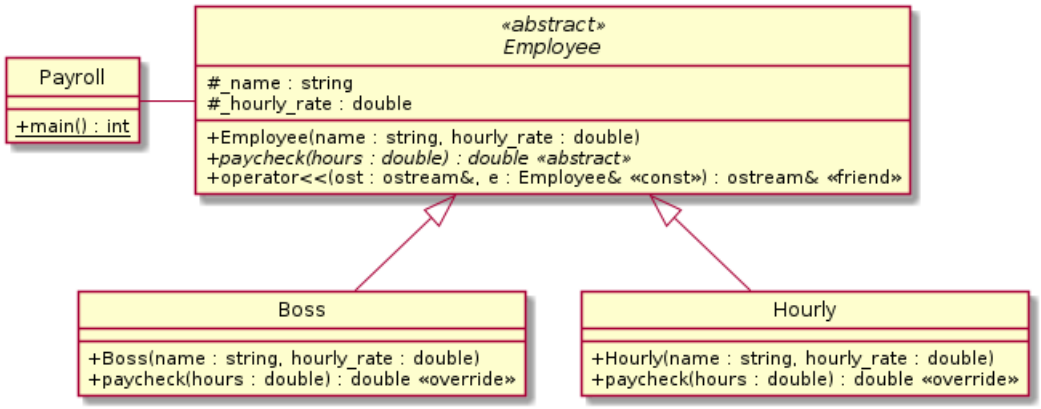
l. {3 points} Continuing to write the payroll.cpp main program, while in the loop begun in part (i), using the variable e created in part (k), print the employee pointed to by e and then *polymorphically* their paycheck amount using its method `paycheck`.

```
std::cout << *e << " is paid $" << e->paycheck(hours) << std::endl; // (l)

delete e;  // We neglected to ask for this, so no deduction if missing.
```



m. {1 points} Continuing to write the payroll.cpp main program, after the loop, if the program did NOT reach the end of the payroll file, print an error message "Incomplete payroll" to standard error.

```
if(!ifs.eof()) std::cerr << "Incomplete payroll" << std::endl; // (m)
```



2. (vector, map, enum, shuffle, iterators) If you prefer to write all of Free Response #2 as a single program rather than in code snippets, please ask for an additional sheet of paper.

a. {1 points} In file Component.h, write enum class `Component` with the values `RED`, `GREEN`, and `BLUE`. You may omit the guard to save time.

```
enum class Component {RED, GREEN, BLUE};
```

b. {2 points} In file Color.cpp, typedef `Color` as a `std::map` with the key a `Component` and the value an `int`. Assume you have all of the `#include` statements that you need; don't write them.

```
typedef std::map<Component, int> Color;
```

c. {1 point} Begin writing the main function. Accept command line parameters. Assume you have all of the `#include` statements that you need; don't write them.

```
int main(int argc, char* argv[]) {

int main(int argc, char** argv) {
```

Note: EITHER of the above is acceptable. We accepted any parameter names, although `argc` and `argv` are VERY well accepted by convention!

d. {1 point} As part of the main function, declare a vector named `colors` that will contain objects of type `Color`.

```
std::vector<Color> colors;
```

e. {1 point} As part of the main function, declare a variable of type `Color`.

```
Color color;
```

f. {3 points} As part of the main function, loop over the command line arguments *by threes*, adding each to the `Color` map in the order `RED`, `GREEN`, and then `BLUE`. Once the `Color` map is updated, add it to the `colors` vector. So if the command line was "main 32 64 96 128 192 255", the `colors` vector would contain 2 maps, a map with `RED` = 32, `GREEN` = 64, and `BLUE` == 96, and a second map with `RED` = 128, `GREEN` = 192, and `BLUE` == 255.

```
for(int i=1; i<argc; i += 3) {
    color[Component::RED]   = std::stoi(argv[i]);   // C's atoi is acceptable
    color[Component::GREEN] = std::stoi(argv[i+1]);
    color[Component::BLUE]  = std::stoi(argv[i+2]);

    colors.push_back(color);
}
```

Or alternately, using the magic of curly brace notation,

```
for(int i=1; i<argc; i += 3) {
    color = {
       {Component::RED,   std::stoi(argv[i])},
       {Component::GREEN, std::stoi(argv[i+1])},
       {Component::BLUE,  std::stoi(argv[i+2])}
    };

    colors.push_back(color);
}
```

Note: In the for loop, many students tried to loop while `i<argv.length` or `i<argv.size()`. Unlike Java, C++ has *no idea* how many elements are in an array, which is why we have `argc`.

Note: Many students forgot to specify the enum name and used bare `RED` et al instead.

Note: No points were deducted for not using `std::stoi` or `atoi`.

Note: A few students used `color.insert(std::pair{Component::RED, argv[i]});` or (more simply) `color.insert({Component::RED, argv[i]});` which we accepted, although this wasn't covered during lecture. Nobody used `color.at(Component::RED) = argv[i];`.

This was a most-missed question.

g. {2 points} As part of the main function, *after* the loop, shuffle the `colors` vector.

```
std::random_shuffle(colors.begin(), colors.end());
```

Note: Most students forgot the `random_` prefix.

Note: Unsurprisingly, many students went with `colors.shuffle();`, which is certainly reasonable!

h. {3 points} As part of the main function, **use iterators only** to print each of the `colors` vector's elements in order. You may assume types `Component` and `Color` have defined the << operator already; you need not code them yourself.

```
auto it = colors.begin();
while(it != colors.end())
    std::cout << *(it++) << std::endl;
```

Or, alternately,

```
for(auto it = colors.begin(); it != colors.end(); ++it)
    std::cout << *it << std::endl;
```

For students who attempted to write the exact type of the iterator, we didn't deduct (or mark) if it was wrong. We still recommend `auto` where the type is evident to the compiler!

This was a most-missed question.

Bonus 1: {+3 points} List the 3 types of C++ casting that we discussed, when they should be used, and give an example of each.

- C cast, do NOT use, `x = (RGB) y;`
- Static cast, when cast is not polymorphic, `x = static_cast<RGB>(y);`
- Dynamic cast, when cast IS polymorphic (at least one virtual method in the superclass), `x = dynamic_cast<RGB>(y);`

Of course, the question was far too vague, so we also accepted

- Upcast
- Downcast
- `const_cast` (which we mentioned only in passing)
- Polymorphic cast (this is the *same* as `dynamic_cast`, so you could only get credit for one or the other)

Note: We were expecting more than just bare names, so only partial bonus was awarded for a bare list.

Bonus 2: {+3 points} `std::find` returns an iterator pointing to the first matching element in its container (collection). How would you find ALL of the matching elements in the container? You may write C++ code (only correct enough to demonstrate the algorithm) or give a *brief* and *clear* text description.

- Start an iterator variable `it` at `begin()`.
- Begin loop.
    - Find from `it` through `end()`.
    - If result == `end()`, exit loop.
    - Otherwise, set `it` to result + 1 and loop again.

Note: We gave *partial* credit even if you didn't clearly state the (critical) result + 1 step.

Note: Describing an element-by-element search without using `std::find` doesn't meet the clear intent of this question, and did not receive any credit.