CSC 371

Part A

Question 1

1. Templates of functions can be written to not explicitly specify the parameters/return types, thus reducing repetition of code. In comparison to function overloading where a function may be written many times just with different types.
2. Template metaprogramming can be used to do calculation of values determined at compile time, thus saving time during run-time. Other polymorphic techniques do not have this ability.

Question 2

- Multiple inheritance – when a derived class inherits from two or more base classes
- The derived class combines the functionalities of all base classes
- The Diamond problem - this is when a derived class A inherits from two other classes, B and C, which both inherit from the same class D
- This means class A will maintain two inheritance chains and will have two copies of all members of D
- C++ overcomes this by using the virtual keyword on derived classes you inherit from
- This tells the compiler not to cascade the derived classes when they are jointly inherited

Question 3

a) Encapsulation:
   - This is when data is hidden to protect it from outside interference, while exposing only the interface which controls communication with the outside
   - In C++ this is implemented as a class
   - The hidden data in a class is denoted by the private (accessed only by the class) or the protected (accessed by derived classes too) keywords
   - The interface uses the public keyword (accessed by everything)

b) Runtime polymorphism:
   - This is when the specific function implementation to be used is determined at run time based on the type of object used
   - In C++ this can be implemented by overloading functions of base classes

- A base class can have a virtual function and a derived class can overload it
-

Part B

Question 4

- The static keyword defines that there is only one instance of a member for all objects of the class
- If a function of class is static it can only access static data members

Question 5

a) Two issues with the code:
   1. Issue 1:
      - Total_marks is being changed when the object is constructed but its denoted as const
      - Variables with the const keyword can't be changed
      - To resolve the issue, remove the const keyword from total_marks
   2. Issue 2:
      - Reveal_answers() in the .cpp is missing is the const keyword
      - The compiler will the treat the reveal_answers() in .cpp and .h files as different functions
      - To resolve the issue add the const keyword to reveal_answers() in .cpp -> bool Exam::reveal_answers() const

b) Printing exam objects:
   1. Alice should use operator overloading
   2. std::ostream &operator<<(std::ostream &os, const Exam &rhs) {
      os << "Exam(" << rhs.total_marks << ")";
      return os;
      }

c) MockExam:
   1. Bob must make the reveal_answers() function in Exam virtual
   2. The Exam class should have: virtual bool reveal_answers() const;
   3. Bob must also make the Exam destructor virtual
   4. The Exam class should have: virtual ~Exam();

Part C

Question 6

- #Pragma pack sets the alignment of members to a specified by boundary
-
- This may not be ideal in practice because the CPU may need more than one cycle to read a variable

Question 7

a) A:

1. The copy constructor and copy assignment operator could be called if we want to make a copy of an employee
2. Copy constructor:

Employee employeeOne = Employee();

Employee employeeTwo(employeeOne);

3. Copy assignment:

Employee employeeThree = Employee();

employeeThree = employeeOne;

b) B:

1. It is **not** possible to implement a move assignment operator
2. This is because employeeNum is **const** and cant be changed
3. How to explicitly flag an operator….

Question 8

```
struct Cat *produce_litter(struct Cat *mother, unsigned int litterSize) {
  struct Cat *cat_array = (struct Cat *) (int *) malloc(litterSize * sizeof(struct Cat));


  for (int i = 0; i < litterSize; i++) {
    struct Cat cat;
```

```
        cat_array[i] = cat;

        cat_array[i].mother = (struct Cat *) (int *) malloc(sizeof(struct Cat));
        memcpy(cat_array[i].mother, mother, sizeof(struct Cat));

        cat_array[i].age = 0;
        mother->offspring[mother->num_offspring] = &cat;
        mother->num_offspring = mother->num_offspring + 1;
    }

    return cat_array;
}
```