Answers to Mann CIS200 Midterm. Expand to use as much space as necessary:

1

g++ “filename1.cpp” “filename2.cpp” -o “ObjectFileNameToCompileAllProgramCodeTo”

2a.

Classes of users: Owner, Groups, and other users

2b.

Permission levels: Read only, Read and Write, Read Write and Execute (rwx)

3.

Use the “passwd” command

4.

Template <class MyClass>

Int MyMethod(MyClass ClassObject) const { return item; }

5.

It is a 3-leg stool: Time, Cost, and Quality 🡪 and the content those 3 “legs” produce

6.

\*Recursion is the use of repetition through self-calling functions (so basically it is a recursive algorithm).

\*Inheritance is the principle of allowing functions to derive the members of the functions from which they are derived. It allows functions to share information and allows for special overloading. It consists of a base class, and then all classes inherited from the base class are known as inherited or derived classes.

\*Function Overloading is the principle of a function having the ability to accept multiple types of input, and the compiler will know which version of the function to call, or which default values should be used for certain variables not specified by the user when the function is called. It also has to do with inheritance, as you can overload derived classes with base class functions and vice versa, this is where copy constructures (aside from when an object contains pointers) can become really important as when you initialize a derived class object to be a base class or upper derived class object, then the compiler has to know which constructor to call (I learned how to do this when we were doing I think lab 4 or 5..whichever one had inheritance). Function overloading is finally very important for operator overloading, as operators are technically functions, and often times the default use of the operands will not suffice or be allowed especially when pointers are involved for an object being operated on; thus, you may need to overload the operator to tell the compiler what to do with a certain object when a given operator is called for a given operand (object).

\*classes: the principle of classes has to do with ADT’s (abstract data types). Classes allow for abstraction because they can be defined and used however a user needs the class to function for the program or a subprogram (functions and whatnot).

\*this leads me to DATA ABSTRACTION which I mentioned; data abstraction is the ability in programming to create any data type for many purposes all within some general (abstract) principle or purpose. For example, I can create an abstract data type through a class, a struct, or iterator (I think iterator is included), through the abstract data types you get objects of each abstract data type which you can use to solve some kind of problem. The key is that no programmer is locked into using the exact same data types to write a program, rather, through a set of non abstract data types, programmers can create many abstract data types, and abstract in the sense that the data of the objects produced from some created data type can be ordered and stored and manipulated as a programmer sees fit (but still within a general principle guideline). The guidelines are so vast, that the possibilities of data types are essentially endless – the combinations of the guidelines are so endless and can encapsulate each other in endless ways, that it is basically “abstract”. It’s similar to God’s creation – everything must follow general scientific principles – a guideline – but the guideline is so vast that the possibilities of creation is infinite.

\*Encapsulation is one of the most important principles for programming. It is what allows for data abstraction to exist. Here is how: we create a foundation with a set of data types with certain guidelines that they must follow; from there, we create more principles, and on top of those principles, we create even more principles..etc….etc…. so you can see that it is like a pyramid, but an abstract pyramid that can be of any size since the bottom layers allow for essentially infinite possibilities. Think of it like an infinite pyramid, where you can infinitely create more and more layers atop one another to accomplish whatever advanced solution you are looking for. It is what allows for algorithms (and especially very indepth recursive algorithms) to be so powerful so as to practically sequentially solve any problem (or at least give some measure of a solution – the number Pi cannot be solved but you can give some measure of it by constantly solving for the next digit up to some finite point of an inifinte set of digits.)

\*Variables, I already explained, but basically all variables are the encapsulation of some other variable. They are all essentially objects of other objects. The lowest level object goes down (I think) basically to bits – ons and offs. So when you declare an integer (int) you are declaring an object of the c++ standard library class. So you create an object – which we can call variables.

\*Polymorphism is the concept that allows you to use members or operators in different ways. It is what gives of function and operator and class overloading.

\*Programming is the use of sequential order (through encapsulation, recursion, etc) to solve a problem.

7.

ADT = Abstract Data Type

8.

ADT only differs from a class in the sense that from classes, we are allowed to declare a given object of that class type. The object is what we refer to as the abstract data type. Although, even a class itself is an abstract data type.

9.

CRC means cyclical redundancy check (I learned this from computer networking!) and it is used for error checking.

10.

3 types of inheritance: Public Private and Protected; they control how the members of the current class and subsequent classes will be accessed/accessible.

11.

12.

Class ParentClass{

Public:

Protected:

};

Class ChildClass : public Parent Class {};

13.

Class bike{

Private:

String color;

Int numWheels;

Public:

Bike() : color(“black”), numWheels(2){

cout << “\n\tConstructor called with values for color and numWheels as: “ << color << “ “ << numWheels;

}

};

14.

Always write your MAIN first and function declarations before writing the actual function code.

15.

Memory leak occurs when a pointer is assigned to point to a new address location, but its original memory location that was dynamically allocated was never destroyed (deleted) and reallocated to the heap.

16.

A dangling pointer occurs when two pointers point at the same memory location, but one of the pointers gets deleted; thus, deallocating the memory. When the pointer which had not been deleted is dereferenced it causes a runtime error (because it points to null) or it may be erroneous set of memory dereferenced.

17.

A shallow copy occurs by default when you pass a parameter into a non referencing function. It does what we call a “memberwise copy” (learned that from zybooks!!), copying each member verbatum – even pointer memory address location. This is what can cause dangling pointers if no copy constructor is properly called.

And this is where DEEP COPY comes to play. A deep copy specifies how each member will be copied into a function, and for pointers we can create a new pointer and allocate it its own memory, only assign the data VALUE stored (no the address) of the pointer passed into the function or referenced to the function.

18.

19.

A, d, e

20a.

20b.

20c.

21.