Jaryd Meek
PE8
IPW
Prof. Nath

Section 1:

1. Reference: Similar to a pointer, must be bound at initialization, can't be changed.
   Pointer: A variable that contains an address of another object.

2. Foo test(10);
   test.CalculateMysteries();

3. Foo::TotalBars();

4. It cannot access bar_ because the variable is not static, but the function is

5. Yes, but you'd have to create an object of Foo first.

6. Foo test(10);
   test.CalculateMysteries();

7. Foo *test = new Foo(10);
   test -> CalculateMysteries();

8. If you want to a store a reference to an object, rather than the object itself

Section 2:

1.

```
bool hasACat = some value from somewhere;

If (!hasACat) {
   returnItForADog();
}
```

2.

```
int x = some value from somewhere;
int y = some value from somewhere;


cout << "We're messing with numbers!" << endl;
```

```
if(y > x) {
  x += 1;
} else if(x > y) {
  y += 1;
} else {
  x = x + y;
}
```

Section 3:

1. For a field, so you don't change the value after initialization, for a method it doesn't allow the method to change the object that it is called on.

2. It either has to be marked static, or you set it in a member initializer.

3. The ability to share fields/methods between objects and share code structure.

4. When you want a child class to overwrite a method in the parent class. This is called abstraction, and it allows you to have different implementations for the same method name, which is crucial for inheritance to work well.

5. To compare any non-primitive data type or when you want to change the behavior of the operator.

Section 4:

1. A copy of the main code to work on implementing a new feature without messing up other people or the main working copy.

2.
   ```
   git checkout master
   git pull
   git checkout my-fabulous-feature
   git merge master
   ```

3. git pull grabs any changes to the remote repository, so they are reflected locally. A pull request is the request to merge your code from one branch into another.

4. Does the code function, comments and style, and does it pass tests?

Section 5:

1. Look at the output from the compiler/interpreter, Look for any lines with errors in the editor window

Section 6:

1. A test case is a set of code to test functionality of methods, a section is a part of code where everything above except other sections is reinstantiated each test. 1 method

should be tested in each test case or section.

Section 7:

1. Singleton: Doesn’t allow multiple instantiations of an object. Implemented with a private constructor, and static creation method.
   Flyweight: Allows copying of object for purpose of saving memory.
   Prototype: Allows you to declare an object before initializing it. You put the function declaration above the use.
   Factory: allows easy creation of objects. A class where each member returns a pointer to a new object.
   Iterator: Allows easy traversal of a data structure. You implement it yourself based on the data structure.

2. Because we rarely implement data structures ourselves from scratch, but often do traverse data structures.

Section 8:

1.

```
template <typename T>

void Swap (T &a, T &b) {

//do swap

}
```

2.

```
int main() {

int a = 0;

int b = 10;
//Work

Swap<int>(a,b);

Swap(a,b);

double c = 10;

//Don’t work
Swap<char>(a,b);
Swap(a,c);

}
```

3.

```
Swap(a,b)
Swap((double)a, c);
```

They did not work initially because I was telling the function the wrong data type to expect, and then because I mixed data types. I fixed it by making sure the data types matched, and make sure to tell the function the right data type.

4. Sometimes you only want your function to work with specified data types.

Section 9:

1. A function gets called, which handles the input as desired.

2. A single object can emit multiple different signals (a button can have clicked vs pressed and so on), and infinite slots can respond to a signal if connected.

3.

```
//In class declaration:

signals:

  void testSignal(int x);

//connect

connect(p, &ClassName::testSignal, this, &OtherClass::testSignalHandler
);


//When you want the object to be called.
```