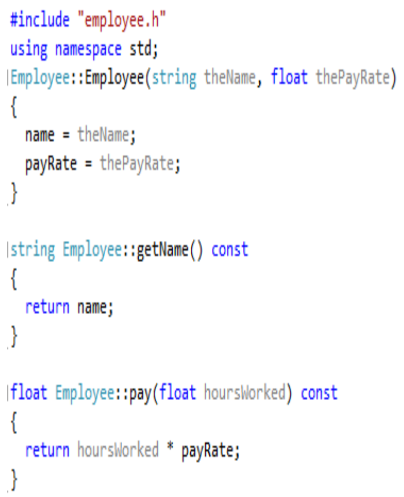
Composition verses Inheritance

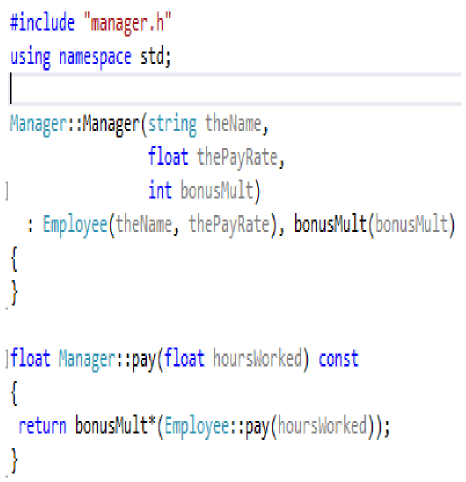
Think of composition as a has a relationship. A car "has an" engine, a person "has a" name, etc.
Think of inheritance as an is a relationship. A car "is a" vehicle, a person "is a" mammal, etc.

Prefer Composition over Inheritance (ALWAYS). Large (and deep) inheritance hierarchies are difficult to debug (you tend to jump around in the class structure) and are delicate (change a base class member overridden in derived class and you can break the whole thing.
Composition aggregates other objects and calls on them (delegation) when their services needed. Ex. I (a person) have a watch. If someone asks me the time, I defer to the watch. (see Composition_Intro project)

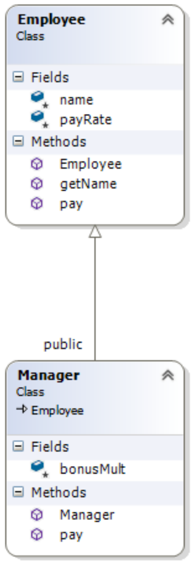
Inheritance implements IS_A



```
#include "employee.h"
using namespace std;
Employee::Employee(string theName, float thePayRate)
{
  name = theName;
  payRate = thePayRate;
}

string Employee::getName() const
{
  return name;
}

float Employee::pay(float hoursWorked) const
{
  return hoursWorked * payRate;
}
```



```
#include "manager.h"
using namespace std;

Manager::Manager(string theName,
                 float thePayRate,
                 int bonusMult)
  : Employee(theName, thePayRate), bonusMult(bonusMult)
{
}

float Manager::pay(float hoursWorked) const
{
 return bonusMult*(Employee::pay(hoursWorked));
}
```



```
class Employee {
public:
  Employee(std::string theName, float thePayRate);

  std::string getName() const;
  float pay(float hoursWorked) const;

protected:
  std::string name;
  float payRate;
};
```

```
#include "employee.h"
class Manager : public Employee {
public:
  Manager(std::string theName,
          float thePayRate,
          int bonusMult);

float pay(float hoursWorked) const;

protected:
  int bonusMult;
};
```

Manager inherits all base class members and data

Calling base class
Manager must call base class if needed
    See constructor IL
      Employee::pay(hoursworked)
     (Scope it)

Pay is in 2 places do not even need pay in manager, but may want to change how pay works override pay to get diff behavior
Or just use base implementation (code reuse).
You must scope or get infinite recursion (remove Employee:: in manager)
If do not call base class employee in constructor. Compiler will attempt to create default constructor to call, if base does not have one will not compile.
    Demo remove employee IL from manager
Do not need to call base class in destructor, compiler handles it.

Base class constructors are automatically called for you if they have no argument. If you want to call a superclass constructor with an argument, you must use the subclass's constructor initialization list. Unlike Java, C++ supports multiple inheritance (for better or worse), so the base class must be referred to by name, rather than "super()".
Protected:
    Way for derived classes to get at innards of base class (member vars and functions) without exposing implementation details to world.
Why? Without it no derived class can get at base class members

In general, it's a good idea to prefer less inheritance. Use composition wherever possible, and inheritance only in the specific situations in which it's needed. Large inheritance hierarchies in general, and deep ones in particular, are confusing to understand and therefore difficult to maintain. Inheritance is a design-time decision and trades off a lot of runtime flexibility.

Go to Composition_intro project

Composition:

When an object posseses something

    I have a watch
    I have a Pixel XL phone

Model it, create a human that takes a watch pointer (see above project)
add a method to give the human a watch and ask the human for the time, if he has a watch he gives the time, if not he says I dont have a watch

Which to Use:

Think of composition as a **has a** relationship. A car "has an" engine, a person "has a" name, etc.
Think of inheritance as an **is a** relationship. A car "is a" vehicle, a person "is a" mammal, etc.

See “Inheritence_Intro” project

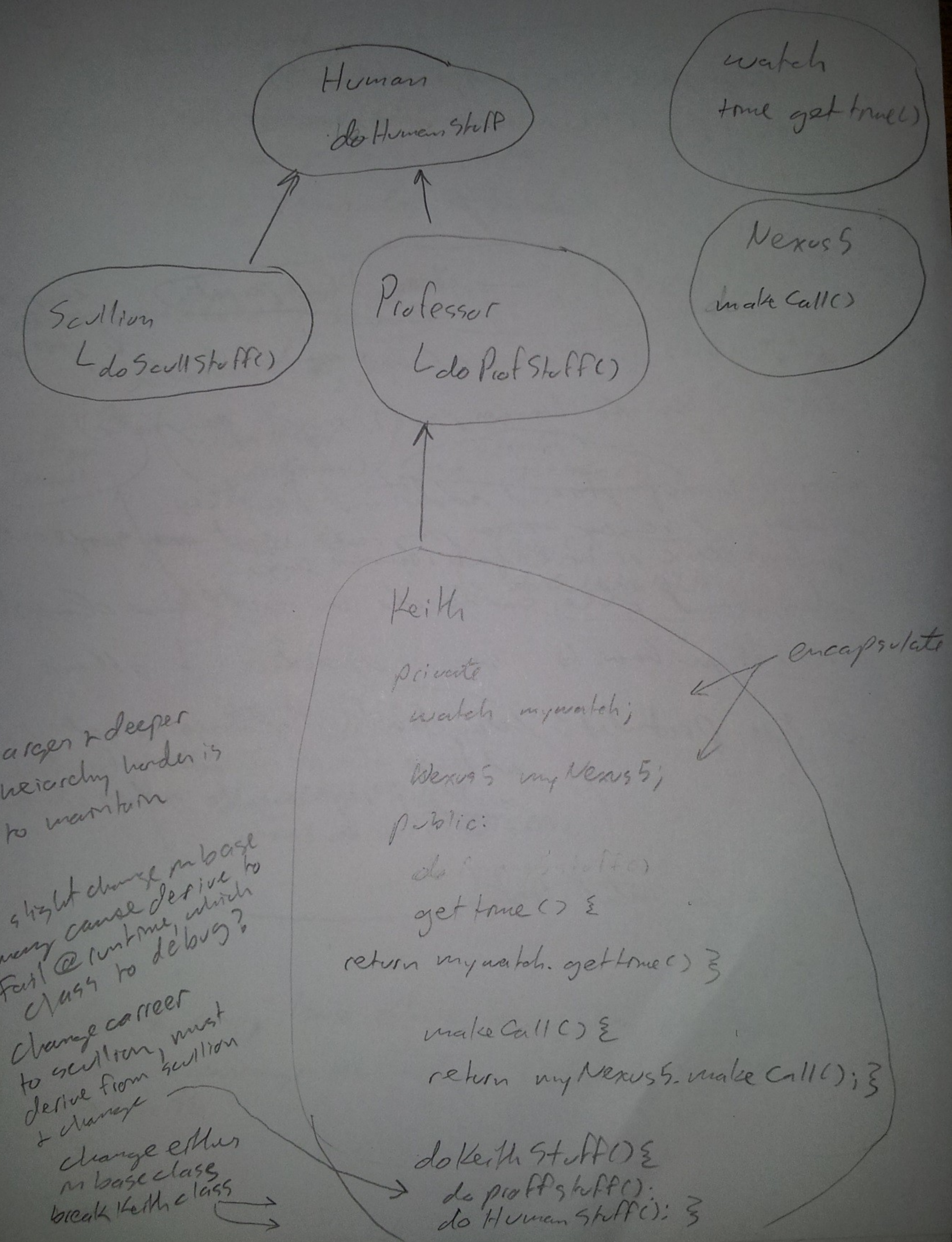
Virtual:

```
void outputPay(Employee *pEmp){
      // Assume all employees worked 40 hours this period.
      cout << "For Employee:" << endl;
      cout << "Name: " << pEmp->getName() << endl;
      cout << "Pay: " << pEmp->pay(40.0) << endl;
}
```

As long as not virtual;

Will always call employee::pay() will never go to most derived class
Unless you make it virtual in header of where virtual starts, every class derived therafter is virtual by default;
AS A COURTESY TO FUTURE DEVELOPERS PUT VIRTUAL IN ALL DERIVED CLASSES

Does it with a V-Table



```
Keith
private
 watch mywatch;

 Nexus5 myNexus5;
public:
 do[illegible]Stuff()
 getTime() {
return mywatch.getTime() }

  makeCall() {
  return myNexus5.makeCall(); }

 doKeithStuff() {
  do proffstuff();
  do HumanStuff(); }
```

larger & deeper heiarchy harder is to maintain

slight change in base may cause derive to fail @ runtime, which class to debug?

change carreer to scullion, must derive from scullion & change

change either in base class break Keith class