Polymorphism Cheat Sheet

You can't access private members of the base class from the derived class:

```
// BAD!
class Base
{
public:
…




private:
   int v;
};

class Derived: public Base
{
public:

   Derived(int q)
   {
      v = q;  // ERROR!
   }


   void foo()
   {
      v = 10; // ERROR!
   }
};
```

```
// GOOD!
class Base
{
public:
   Base(int x)
     { v = x; }
   void setV(int x)
     { v = x; }
   …
private:
   int v;
};

class Derived: public Base
{
public:

   Derived(int q)
     : Base(q)   // GOOD!
   {
     …
   }

   void foo()
   {
      setV(10);  // GOOD!
   }
};
```

Always make sure to add a virtual destructor to your base class:

```
// BAD!
class Base
{
public:
  ~Base() { … } // BAD!
  …
};

class Derived: public Base
{
  …
};
```

```
// GOOD!
class Base
{
public:
  virtual ~Base() { … } // GOOD!
  …
};

class Derived: public Base
{
  …
};
```

```
class Person
{
public:
  virtual void talk(string &s) { … }
  …
};

class Professor: public Person
{
public:
     void talk(std::string &s)
     {
        cout << "I profess the following: ";
        Person::talk(s); // uses Person's talk
     }
};
```

Don't forget to use virtual to define methods in your base class, if you expect to re-define them in your derived class(es)

To call a base-class method that has been re-defined in a derived class, use the base:: prefix!

So long as you define your BASE version of a function with virtual, all derived versions of the function will automatically be virtual too (even without the virtual keyword)!

```
class SomeBaseClass
{
public:
   virtual void aVirtualFunc() { cout << "I'm virtual"; } // #1
   void notVirtualFunc() { cout << "I'm not"; }           // #2
   void tricky()                                          // #3
   {
       aVirtualFunc();                                    // ***
       notVirtualFunc();
    }
};

class SomeDerivedClass: public SomeBaseClass
{
public:
   void aVirtualFunc()   { cout << "Also virtual!"; }     // #4
   void notVirtuaFuncl() { cout << "Still not"; }         // #5
};

int main()
{
   SomeDerivedClass d;
   SomeBaseClass  *b = &d;  // base ptr points to derived obj

   // Example #1
   cout << b->aVirtualFunc();      // calls function #4

   // Example #2
   cout << b->notVirtualFunc(); // calls function #2

   // Example #3
   b->tricky();       // calls func #3 which calls #4 then #2
}
```

Polymorphism Cheat Sheet, Page #2

Example #1: When you use a BASE pointer to access a DERIVED object, AND you call a VIRTUAL function defined in both the BASE and the DERIVED classes, your code will call the DERIVED version of the function.

Example #2: When you use a BASE pointer to access a DERIVED object, AND you call a NON-VIRTUAL function defined in both the BASE and the DERIVED classes, your code will call the BASE version of the function.

Example #3: When you use a BASE pointer to access a DERIVED object, all function calls to VIRTUAL functions (***) will be directed to the derived object's version, even if the function (tricky) calling the virtual function is NOT VIRTUAL itself.