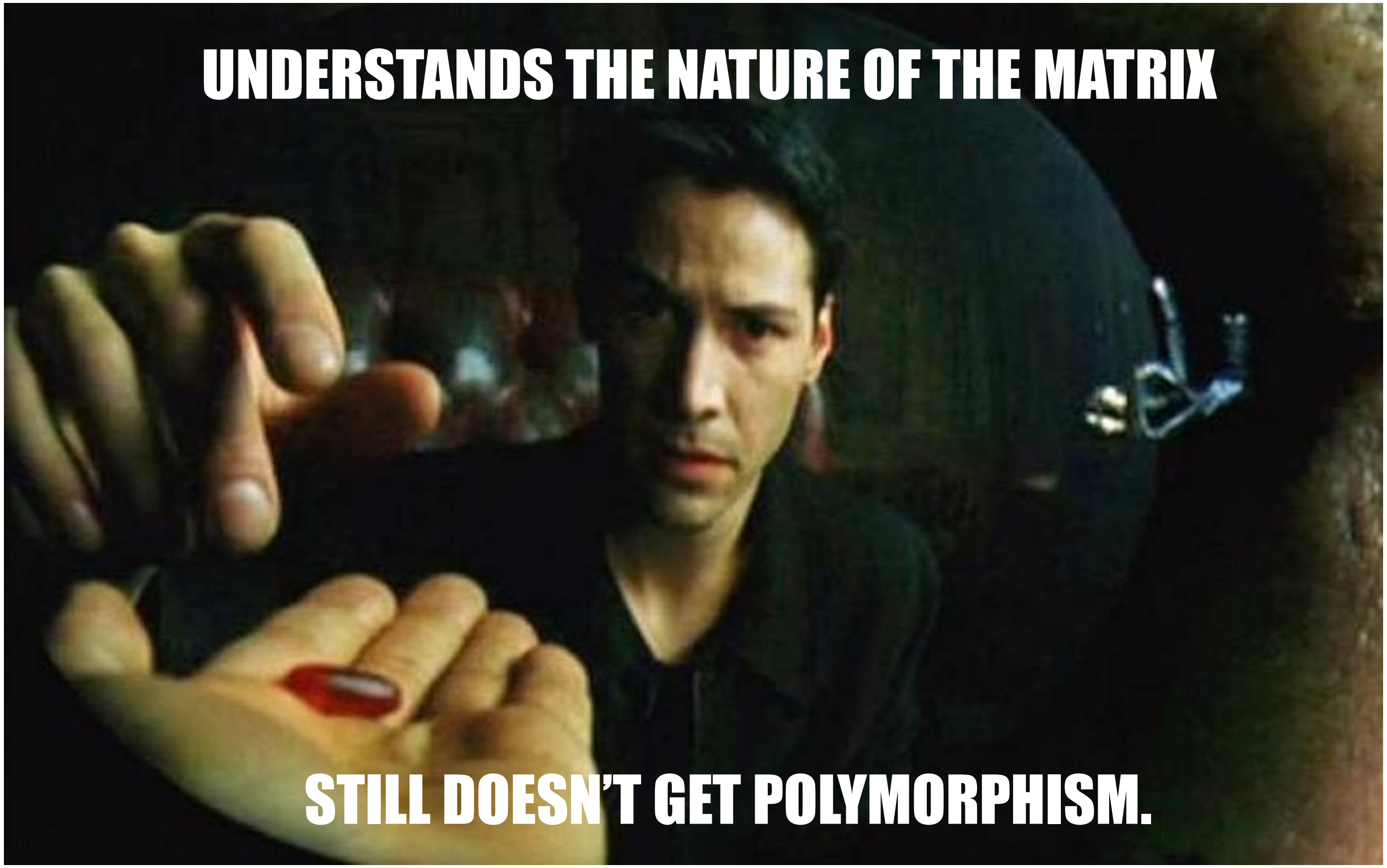
# Lecture #7

- Polymorphism
  - Introduction
  - Virtual Functions
  - Virtual Destructors
  - Pure Virtual Functions
  - Abstract Base Classes

# Polymorphism



# Polymorphism

## What's the big picture?

You may pass Students and Profs to a function f that accepts Persons. If f calls their methods, each will behave not as a Person, but in its own specialized way.

```
class Person {
public:
  string getName()
  string talk()
};
```

```
class Student: public Person {
   string talk() { return "Go bruins!"; }
};

class Prof: public Person {
   string talk() { return "Pop quiz!"; }
};
```

```
void AskPersonToTalk(Person &p) {
  cout << p.getName() << " says " << p.talk();
}

int main() {
  Student stud("Sam");
  AskPersonToTalk(stud); // Prints "Sam says Go Bruins!"
  Professor prof("Juan");
  AskPersonToTalk(prof); // Prints "Juan says Pop Quiz!"
}
```

Looks like just a person…

But behaves

**Uses:**

Video games, where each monster behaves in its own way when asked to attack(), circuit simulations, etc.

# Polymorphism

Consider a function that accepts a Person as an argument

Can we also pass a Student as a parameter to it?

```
void LemonadeStand(Person &p)
{
   cout << "Hello " << p.getName();
   cout << "How many cups of ";
   cout << "lemonade do you want?";
}
```

```
class Person
{
public:
   string getName();
   ...

private:
   string m_sName;
   int    m_nAge;
};
```

I'd like to buy some lemonade.

Person

# Polymorphism

Consider a function that accepts a Person as an argument

```
void LemonadeStand(Person &p)
{
   cout << “Hello “ << p.getName();
   cout << “How many cups of ”;
   cout << “lemonade do you want?”;
}
```

Can we also pass a Student as a parameter to it?

```
class Student :
      public Person
{
public:
   // new stuff:
   int getStudentID();
private:
   // new stuff:
   int m_nStudentID;
};
```


Student

# Polymorphism

The idea behind polymorphism is that once I define a function that accepts a (*reference* or *pointer* to a) Person…

Not only can I pass Person variables to that class…

But I can also pass any variable that was derived from a Person!

```
class Person
{
public:
   string getName()
   { return m_name; }

pr

};
```

```
class Student : public Person
{
public:
   // new stuff:
   int getGPA();
private:
   // new stuff:
   float m_gpa;
};
```

```
void SayHi(Person &p)
{
   cout << "Hello " <<
     p.getName();
}

int main()
{
   float GPA = 1.6;
   Student s("David",19, GPA);

   SayHi(s);
}
```

# Polymorphism

- Why is this? Well a Student *IS* a Person. Everything a Person can do, it can do.
- So if I can ask for a Person's name with getName, I can ask for a Student's name with getName too!
- Our SayHi function now treats variable p as if it referred to a Person variable…
- In fact, SayHi has no idea that p refers to a Student!
- Notice how the Student parts of variable s are greyed out. They're still there, but the SayHi function has no idea that those parts are there… and can't use them!

s

**Person's Stuff**

```
string getName()
  { return m_name; }
int  getAge()
  { return m_age; }
```

m_name "David" m_age 52

**Student's Stuff**

```
float getGPA()
  { return m_gpa; }
```

m_gpa 1.6

```
void SayHi(Person &p)
{
    cout << "Hello " <<

}

int main()
{
    float GPA = 1.6;
    Student s("David",52, GPA);

    SayHi(s);
}
```

# Polymorphism

Any time we use a base pointer or a base reference to access a derived object, this is called polymorphism.

```
class Person
{
public:
   string getName();
   ...

private:
   string
   int
};
```

```
class Student :
        public Person
{
public:
   // new stuff:
   int getStudentID();
private:
   // new stuff:
   int m_nStudentID;
};
```

```
void SayHi(Person *p)
{
   cout << "Hello " <<
     p->getName();
}

int main()
{
   Student s("Carey",38,3.9);

   SayHi(&s);
}
```

# Polymorphism and Chopping!

- Polymorphism only works when you use a reference or a pointer to pass an object!
- You MUST use a pointer or reference for polymorphism to work! Otherwise something called "chopping" happens…
- C++ will basically chop off all the data/methods of the derived (Student) class and only send the base (Person) parts of variable s to the function!
- In this example, the SayHi function isn't dealing with the original Student variable!
- It has a chopped temporary variable that has no Student parts!

p

**Person's Stuff**

```
string getName()
  { return m_name; }
int  getAge()
  { return m_age; }
```

m_name "Carey" m_age 38

bad!

```
void SayHi(Person  p)
{
   cout << "Hello " <<
      p.getName();
}


int main()
{
   Student s("Carey",38,3.9);

   SayHi(s);
}
```

s

**Person's Stuff**

```
string getName()
  { return m_name; }
int  getAge()
  { return m_age; }
```

m_name "Carey" m_age 38

**Student's Stuff**

```
float getGPA()
 { return m_gpa; }
```

m_gpa 3.9

# Polymorphism

```
class Shape
{
public:
 virtual double getArea()
  { return (0); }
   ...
private:
   ...
};
```

```
class Square: public Shape
{
public:
 Square(int side){ m_side=side; }
 virtual double getArea()
  { return (m_side*m_side); }
private:
 int m_side;
};
```

- Let's define a new class called Shape, which represents an abstract shape.
- Since all shapes have an *area*, we define a member function called getArea.
- Now let's consider two classes derived from Shape: Square and Circle.
- Square has its own c'tor as well as an updated getArea function that overrides the one from Shape.
- Similarly, Circle has its own c'tor and an updated getArea function.
- Notice that in the Shape base class, getArea() returns zero. Why? Well, what is the area of an "abstract" shape? Who knows!?! We'll just assume it's zero!

```
class Circle: public Shape
{
public:
 Circle(int rad){ m_rad=rad; }
 virtual double getArea()
  { return (3.14*m_rad*m_rad); }
private:
 int m_rad;
};
```

# Polymorphism

- Let's say we're a company that sells glass windows.
- We want to write a program to compute the cost of each window.
- For example, assume that each window is $3.25 per square foot.
- Let's look at a program that computes the cost for both square and circular windows.

```
void PrintPriceSq(Square &x)
{
  cout << "Cost is: $";
  cout << x.getArea() * 3.25;
}

void PrintPriceCir(Circle &x)
{
  cout << "Cost is: $";
  cout << x.getArea() * 3.25;
}

int main()
{
  Square s(5);
  Circle c(10);

  PrintPriceSq(s);
  PrintPriceCir(c);
}
```

s: m_side 5

c: m_rad 10

```
class Shape
{
public:
  virtual double getArea()
    { return (0); }
    ...
private:
    ...
```

```
class Square: public Shape
{
public:
  Square(int side){ m_side=side; }
  virtual double getArea()
    { return (m_side*m_side); }
private:
  int m_side;
```

```
class Circle: public Shape
{
public:
  Circle(int rad){ m_rad = rad; }
  virtual double getArea()
    { return (3.14*m_rad*m_rad); }
private:
  int m_rad;
};
```

# Polymorphism

```
class Shape
{
public:
 virtual double getArea()
```

```
class Square: public Shape
```

```
class Circle: public Shape
{
public:
 Circle(int rad){ m_rad = rad; }
 virtual double getArea()
  { return (3.14*m_rad*m_rad); }
private:
 int m_rad;
};
```

```
void PrintPrice(Shape &x)
{
   cout << "Cost is: $";
   cout << x.getArea() * 3.25;
}

int main()
{
   Square s(5);
   Circle c(10);

   PrintPrice  (s);
   PrintPrice   (c);
}
```

- It works, but it's inefficient. Why should we write two functions to do the same thing?
- Both Squares and Circles are Shapes, and we know that you can get the area of a Shape since all Shape variables have a getArea() method (see the Shape class above).
- So how about if we create a single PrintPrice() function that takes a reference to a Shape?
- Now the PrintPrice() function can accept any type of object as long as it's derived from the Shape class! You can pass in Circles, Squares, Triangles - any Shape with a getArea() function.

# Polymorphism

```
class Shape
{
public:
  virtual double getArea()
   { return (0); }
   ...
private:
   ...
};
```

```
class Square: public Shape
{
public:
  Square(int side){ m_side=side; }
  virtual double getArea()
   { return (m_side*m_side); }
pri
  int
};
```

```
class Circle: public Shape
{
public:
  Circle(int rad){ m_rad = rad; }
  virtual double getArea()
   { return (3.14*m_rad*m_rad); }
private:
  int m_rad;
};
```

```
void PrintPrice(Shape &x)
{
   cout << "Cost is: $";
   cout << x.getArea()*3.25;
}

int main()
{
   Square s(5);
   Circle c(10);

   PrintPrice(s);
   PrintPrice(c);
```

- When you call a virtual function of an object, C++ figures out the correct version to use and calls it automagically!
- So if you pass Circle c to PrintPrice(), then the call to x.getArea() will call Circle's version of getArea(), which has access to c's member variables.
- And if you pass Square s to PrintPrice() then the same call to x.getArea() will call Square's version which has access to s's member variables!
- And if you pass a basic Shape object to PrintPrice() then x.getArea() will call Shape's version of the function! It all just works!

# Polymorphism

```
class Shape
{
public:
  virtual double getArea()
   { return (0); }
   ...
 private:
   ...
};
```

```
class Square: public Shape
{
public:
 Square(int side){ m_side=side; }
 virtual double getArea()
  { return (m_side*m_side); }
pr
 i
};
```

```
class Circle: public Shape
{
public:
 Circle(int rad){ m_rad = rad; }
 virtual double getArea()
  { return (3.14*m_rad*m_rad); }
private:
 int m_rad;
};
```

```
void PrintPrice(Shape &x)
{
   cout << "Cost is: $";
   cout << x.getArea()*3.25;
}

int main()
{
   Square s(5);
   Circle c(10);

   PrintPrice(s);
   PrintPrice(c);
```

When you use the virtual keyword, C++ figures out what class is being referenced and calls the right function.

So the call to getArea()…

Might go here… Or here…

Or even here…

# Polymorphism

```
class Shape
{
public:
  virtual double getArea()
   { return (0); }
   ...
private:
   ...
};
```

```
class Circle: public Shape
{
public:
  ...
  virtual double getArea()
   { return (3.14*m_rad*m_rad); }

  void setRadius(int newRad)
   { m_rad = newRad; }

private:
  int m_rad;   10
};
```

```
void PrintPrice(Shape &x)
{
   cout << "Cost is: $";
   cout << x.getArea()*3.25;
   x.setSide(10);  // ERROR!
}
int main()
{
   Square s(5);
   PrintPrice(s);

   Circle c(10);
   PrintPrice(c);
```

As we can see, our PrintPrice method THINKS that every variable you pass in to it is JUST a Shape.

It thinks it's operating on a Shape - it has no idea that it's really operating on a Circle or a Square!

This means that it only knows about functions found in the Shape class!

Functions specific to Circles or Squares are TOTALLY invisible to it!

# So What is Inheritance? What is Polymorphism?

## Inheritance:

We publicly derive one or more classes $D_1 \ldots D_n$ (e.g., Square, Circle, Triangle) from a common base class (e.g., Shape).

All of the derived classes, by definition, inherit a common set of functions from our base class: e.g., getArea(), getCircumference()

Each derived class may re-define any function originally defined in the base class; the derived class will then have its own specialized version of that function.

## Polymorphism:

Now I may use a Base pointer/reference to access any variable that is of a type that is derived from our Base class:

```
void printPrice(Shape *ptr)
{
  cout << "At $10/square foot, your price is: ";
  cout << "$" << 10.00 * ptr->getArea();
}
```

```
Circle c(10); // rad=10
Square s(20); // width=20
printPrice(&c);
printPrice(&s);
```

The same function call automatically causes different actions to occur, depending on what type of variable is currently being referred/pointed to.

# Why use Polymorphism?

With *polymorphism*, it's possible to design and implement systems that are more easily *extensible*.

Today: We define Shape, Square, Circle and PrintPrice(Shape &s).

Tomorrow: We define Parallelogram and our PrintPrice function automatically works with it too!

Every time your program accesses an object through a base class reference or pointer, the referred-to object automatically behaves in an appropriate manner - all without writing special code for every different type!

# Polymorphism

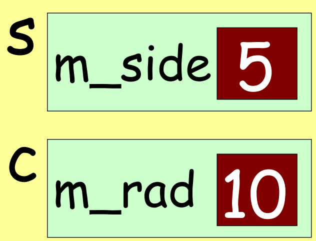
NOT VIRTUAL!

```
class Shape
{
public:
 double getArea()
  { return (0); }
  ...
private:
  ...
};
```

```
class Square: public Shape
{
public:
 Square(int side){ m_side=side; }
 double getArea()
  { return (m_side*m_side); }
priv
 int
};
```

```
class Circle: public Shape
{
public:
 Circle(int rad){ m_rad = rad; }
 double getArea()
  { return (3.14*m_rad*m_rad); }
private:
 int m_rad;
};
```

```
void PrintPrice(Shape &x)
{
  cout << "Cost is: $";
  cout << x.getArea()*3.25;
}

int main()
{
  Square s(5);
  Circle c(10);

  PrintPrice(s);
  PrintPrice(c);
```




- WARNING: When you omit the virtual keyword, C++ can't figure out the right version of the function to call…
- So it just calls the version of the function defined in the base class!
- In this example, whether we pass in s or c to PrintPrice(), the call to x.getArea() will go to Shape's version of getArea(), which always returns zero!
- This can result in nasty bugs, so don't forget "virtual!"

# Polymorphism

When should you use the virtual keyword?

1. Use the virtual keyword in your base class *any time* you expect to redefine a function in a derived class.

2. Use the virtual keyword in your derived classes *any time* you redefine a function (for clarity; not req'd).

3. Always use the virtual keyword for the destructor in your base class (& in your derived classes for clarity).

4. You can't have a virtual constructor, so don't try!

(The constructor is always called at class creation, and there you always know what type the class is, so virtual doesn't make any sense for a constructor. Constructors are class local, so you can't override the constructor of the parent class.)

# Polymorphism and Pointers

```
class Person
{
public:
  string getName()
   { return m_name; }
   ...
private:
   ...
};
```

```
class Politician: public Person
{
public:
  void tellALie()
   { cout << m_myLie; }
  void wasteMoney(int dollars)
   { m_specialInterest += dollars; }
private:
  ...
};
```

```
int main()
{
  Politician jack;
  Politician *p;

  p = &jack;
  cout << p->tellALie();
}
```

Polymorphism works with pointers too! Let's see!

Clearly, we can use a Politician pointer to access a Politician variable...

# Polymorphism and Pointers

Superclass

```
class Person
{
public:
  string getName()
   { return m_name; }
   ...
private:
   ...
};
```

Subclass

```
class Politician: public Person
{
public:
  void tellALie()
   { cout << m_myLie; }
  void wasteMoney(int dollars)
   { m_specialInterest += dollars; }
private:
  ...
};
```

Subclass variable

Superclass pointer

```
int main()
{
  Politician carey;
  Person *p;


  p = &carey; // OK????
  cout << p->getName();
}
```

Question: Can we point a Person pointer at a Politician variable?

Yes: In general, you may point a superclass pointer at a subclassed variable.

# Polymorphism and Pointers

Superclass

```
class Person
{
public:
  string getName()
   { return m_name; }
   ...
private:
   ...
};
```

Subclass

```
class Politician: public Person
{
public:
 void tellALie()
   { cout << m_myLie; }
 void wasteMoney(int dollars)
   { m_specialInterest += dollars; }
private:
  ...
};
```

Subclass pointer

Superclass variable

```
int main()
{
  Politician *p;
  Person david;



  p = &david; // NO!!

  ...
}
```

Question: Can we point a Politician pointer at a Person variable?

Answer: NO! David is not a Politician so we can't treat him like one! He's just a Person! He has no Politician parts.

It's not allowed.

In general, you can never point a subclass pointer at a superclass variable!

# Polymorphism and Pointers!

In this example, we'll use a Shape pointer to point to either a Circle or a Square, then get its area!

choice 's'

ptr

```
int main()
{
    Square sq(5);
    Circle cr(10);
    char choice;
    Shape *ptr;

    cout << "Pick (s)quare,(c)ircle:";
    cin >> choice;
    if (choice == 's')
      ptr = &sq;
    else ptr = &cr;

    cout << "Your shape's area is: ";
    cout << ptr->getArea();
}
```

Pick (s)quare, (c)ircle: s

Your shape's area is:

sq

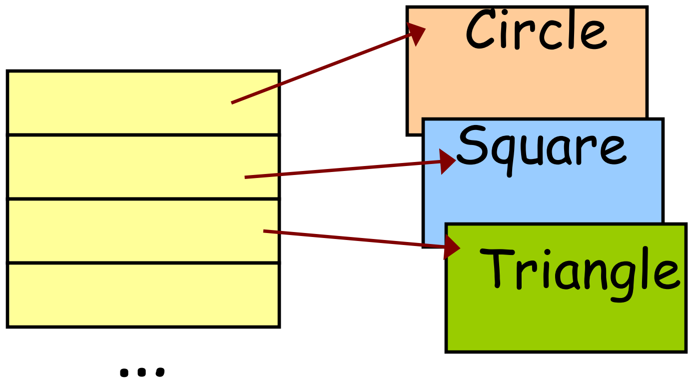
```
class Square: public Shape
{
public:
  …
  virtual double getArea()
   { return (m_side*m_side);
private:
  int m_side;  5
};
```

cr

```
class Circle: public Shape
{
public:
  …
  virtual double getArea()
   { return (3.4159*m_rad*m_
private:
  int m_rad;  10
};
```

# Polymorphism and Pointers

```
int main()
{
    Circle        c(1);
    Square        s(2);
    Triangle      t(4,5,6);
    Shape         *arr[100];

    arr[0] = &c;
    arr[1] = &s;
    arr[2] = &t;

    // redraw all shapes
    for (int i=0;i<3;i++)
    {
       arr[i]->plotShape();
    }
}
```

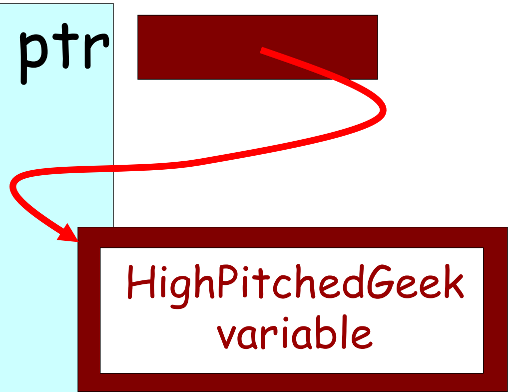



- Here's another example where polymorphism is useful.
- What if we were building a graphics design program and wanted to easily draw each shape on the screen?
- We could add a virtual plotShape() method to our Shape, Circle, Square and Triangle classes.
- Now our program simply asks each object to draw itself and it does!

# Virtual HELL!

## What does it print?

```
class Geek
{
public:
   void tickleMe()
   {
      laugh();
   }
   virtual void laugh()
   { cout << "ha ha!"; }
};
```




```
int main()
{
   Geek *ptr = new
     HighPitchGeek;

    ptr->tickleMe(); // ?

    delete ptr;
}
```

```
class HighPitchGeek: public Geek
{
public:
  virtual void laugh()
  { cout << "tee hee hee"; }
};
```

```
class BaritoneGeek: public Geek
{
public:
  virtual void laugh()
  { cout << "ho ho ho"; }
};
```

- This one's tricky. ptr points to a HighPitchGeek
- But we then call the tickleMe() function, which is only defined in the base Geek class (it's not redefined in HighPitchedGeek)
- So when tickleMe() calls laugh(), you might think it will use the base version of the function in Geek.
- But that's not right. C++ sees that laugh() is virtual and that it has been redefined.
- It also sees that ptr really points to a high-pitched geek.
- So it calls laugh() from HighPitchGeek!
- C++ always calls the most-derived version of a function associated with a variable, as long as it's marked virtual!

# Polymorphism and Virtual Destructors

You should always make sure that you use virtual destructors when you use inheritance/polymorphism.

Next, we'll look at an example that shows a program with and without virtual destructors.

# Polymorphism and Virtual Destructors

```
class Prof
{
public:
  Prof()
  {
    m_myIQ = 95;
  }

  virtual ~Prof()
  {
    cout << “I died smart: ”
    cout << m_myIQ;
  }
private:
  int m_myIQ;
};
```

```
class MathProf: public Prof
{
public:
 MathProf()
 {
   m_pTable = new int[6];

   for (int i=0;i<6;i++)
    m_pTable[i] = i*i;
 }
 virtual ~MathProf()
 {
   delete [] m_pTable;
 }
private:
 int *m_pTable;
};
```

Summary:

All professors think they're smart. (Hmm… is 95 smart???)

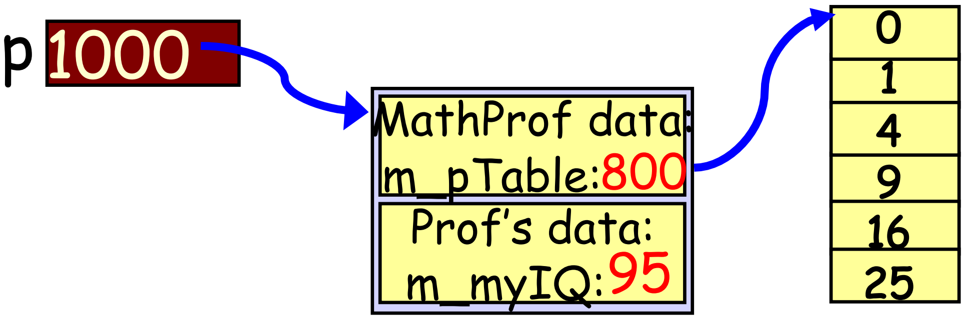
All math professors keep a set of flashcards with the first 6 square numbers in their head.

# Virtual Destructors

```
class Prof
{
public:
  Prof()
  {
    m_myIQ = 95;
  }

  virtual ~Prof()
  {
    cout << "I died smart:"
    cout << m_myIQ;
  }
private:
  int m_myIQ;
```

```
int main()
{
   Prof *p;

   p = new MathProf;
   ...
   delete p;
}
```

```
class MathProf: public Prof
{
public:
 MathProf(){…}
 virtual ~MathProf()
 {
   delete [] m_pTable;
 }
private:
 int *m_pTable;
};
```




- This code works great.
- When we "delete p;" because the Prof destructor is virtual, C++ first calls MathProf's destructor and THEN calls Prof's destructor, all automatically.
- By the way, you don't need to make your MathProf destructor virtual so long as the destructor in your base class is virtual.
- The derived class's virtual destructor will automatically become virtual if the base destructor is

# Virtual Destructors

Now let's see what happens if our destructors aren't virtual functions*.

```
class Prof
{
public:
  Prof()
  {
    m_myIQ = 95;
  }

  ~Prof()
  {
    cout << "I died smart:"
    cout << m_myIQ;
  }
private:
  int m_myIQ;
};
```

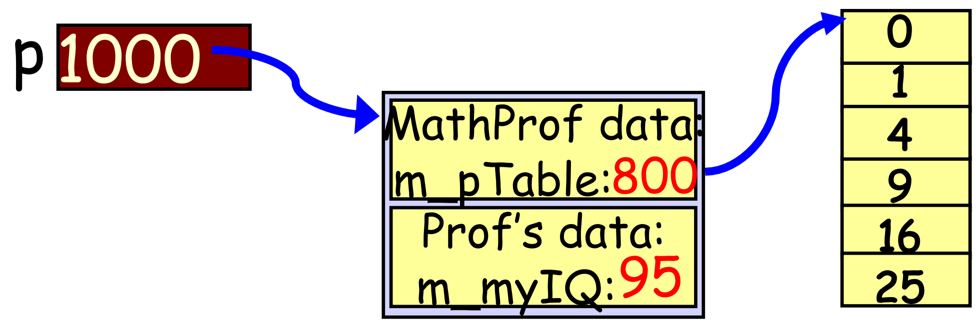
```
class MathProf: public Prof
{
public:
 MathProf()
 {
   m_pTable = new int[6];

   for (int i=0;i<6;i++)
    m_pTable[i] = i*i;
 }
 ~MathProf()
 {
   delete [] m_pTable;
 }
private:
 int *m_pTable;
};
```

* Technically, if you don't make your destructor virtual your program will have undefined behavior (e.g., it could do anything, including crash), but what I'll show you is the typical behavior.

# Virtual Destructors

```
class Prof
{
public:
  Prof()
  {
    m_myIQ = 95;
  }

  ~Prof() // No virtual here!
  {
    cout << "I died smart:"
    cout << m_myIQ;
  }
private:
  int m_myIQ;
```

```
class MathProf: public Prof
{
public:
 MathProf(){…}
 ~MathProf()
 {
   delete [] m_pTable;
 }
private:
 int *m_pTable;
};
```




```
int main()
{
   Prof *p;

   p = new MathProf;
   ...
   delete p;
}
```

- Now we have a problem.
- When we go to delete p on the last line in main(), C++ can't tell that there's a MathProf destructor to call because we didn't make our Prof destructor virtual.
- So this code will only call Prof's destructor, since all C++ has a Prof pointer to go by
- C++ will not call MathProf's destructor, which means that our MathProf will never delete its array of square numbers.
- This will result in a memory leak!

# Virtual Destructors – What Happens?

```
class Person
{
public:
   ...
   virtual ~Person( )
   {
     cout << "I'm old!"
   }
};
```

```
class Prof: public Person
{
public:
   ...
   ~Prof()
   {
     cout << "Argh! No tenure!"
   }
};
```

- So what happens if we forget to make a base class's destructor virtual and then define a derived variable in our program with no polymorphism?
- Will both destructors be called?
- In fact, our code works just fine in this case.
- If you forget a virtual destructor, it only causes problems when you use polymorphism.
- But to be safe, if you use inheritance ALWAYS use virtual destructors – just in case.

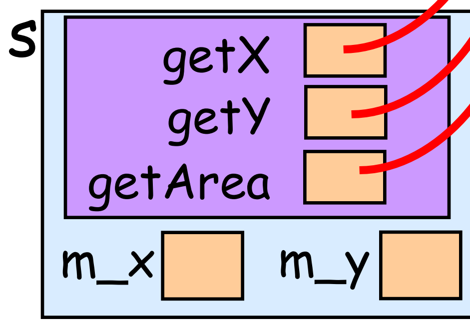
```
int main()
{
   Prof carey;

     ...

} // carey is destructed fine
```

Argh! No tenure!

I'm old!

# How does it all work?

- When you define a variable of a class…
- C++ adds an (invisible) table to your object that points to the proper set of functions to use.
- This table is called a "vtable" – shown below at the top of variable s
- It contains an entry for *every* virtual function in our class.
- In the case of a Shape variable, all three pointers in our vtable point to our Shape class's functions.



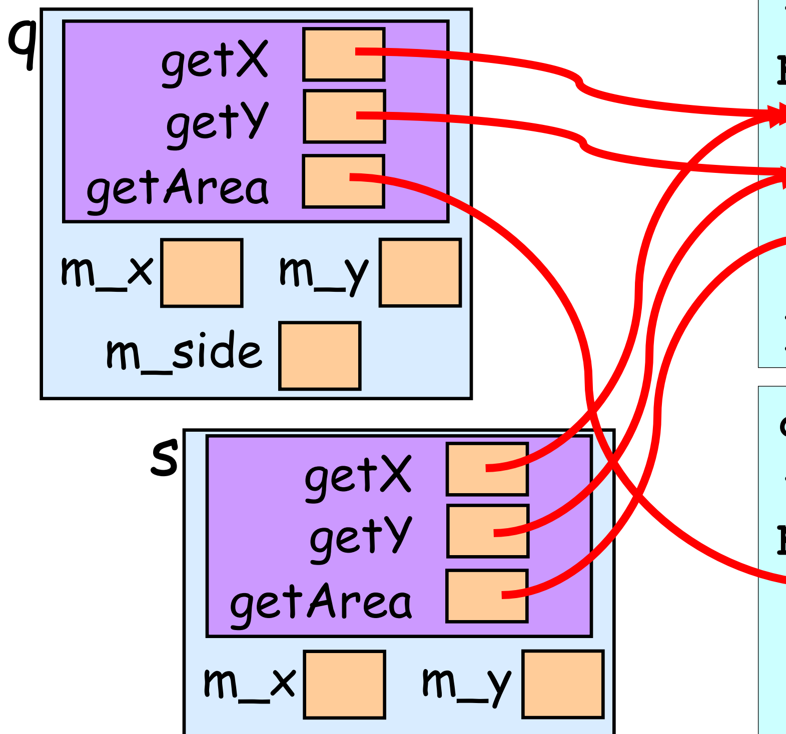

```
int main()
{
   Shape s;

}
```

```
class Shape
{
public:
 virtual int getX()  {return m_x;}
 virtual int getY()  {return m_y;}
 virtual int getArea()  {return 0;}
 ...
};
```

```
class Square: public Shape
{
public:
   virtual int getArea()
    { return (m_side*m_side); }
   …
};
```

```
class Circle: public Shape
{
public:
  virtual int getArea()
   { return (3.14*m_rad*m_rad); }
   …
};
```

# How does it all work?




```
class Shape
{
public:
 virtual int getX()  {return m_x;}
 virtual int getY()  {return m_y;}
 virtual int getArea()  {return 0;}
 ...
};
```

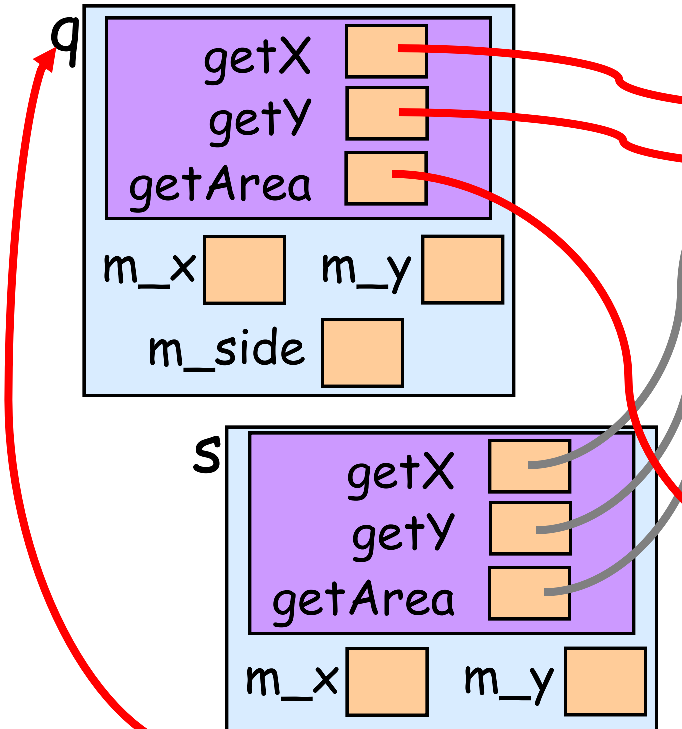
```
class Square: public Shape
{
public:
  virtual int getArea()
    { return (m_side*m_side); }
   ...
};
```

```
class Circle: public Shape
{
public:
  virtual int getArea()
   { return (3.14*m_rad*m_rad); }
   ...
};
```

- Ok, how about if we define a Square variable?
- Well, our Square has its own getArea() function so its vtable entry points to that version...
- However, our Square basically uses our Shape's getX & getY functions, so our other entries will point there.

```
int main()
{
  Shape s;

  Square q;
}
```

# How does it all work?




```
class Shape
{
public:
  virtual int getX()  {return m_x;}
  virtual int getY()  {return m_y;}
  virtual int getArea()  {return 0;}
  ...
};
```

```
class Square: public Shape
{
public:
   virtual int getArea()
   { return (m_side*m_side); }
   ...
};
```

```
int main()
{
   Shape s;
   Square q;
   cout << s.getArea();
   Shape *p = &q;
   cout << p->getArea(); // uses vtable!
}
```

p

C++ uses the vtable at run-time (not compile-time) to figure out which virtual function to call.

The details are a bit more complex, but this is the general idea.

# Summary of Polymorphism

- First we figure out what we want to represent (like a bunch of shapes)

- Then we define a base class that contains functions common to all of the derived classes (e.g. getArea, plotShape).

- Then we write our derived classes, creating specialized versions of each common function:

Square version of getArea

```
virtual int getArea()
{
  return(m_side * m_side);
}
```

Circle version of getArea

```
virtual int getArea()
{
  return(3.14*m_rad*m_rad);
}
```

- We can access derived variables with a base class pointer or reference.

- Finally, we should (MUST) always define a virtual destructor in our base class, whether it needs it or not. *(no vd in the base class, no points!)*

# Useless Functions

```
class Shape
{
public:
  virtual double getArea()  { return(0);}
  virtual double getCircum(){return(0);}
  virtual ~Shape() { … }
};
```

```
class Square: public Shape
{
public:
  virtual double getArea()
    { return (m_side*m_side); }
  virtual double getCircum()
    { return (4*m_side); }
   ...
```

```
class Circle: public Shape
{
public:
  virtual double getArea()
    { return (3.14*m_rad*m_rad); }
  virtual double getCircum()
    { return (2*3.14*m_rad); }
  …
```

- When I call the PrintInfo function and pass in a Square, C++ calls Square's getArea function
- And when I call the PrintInfo function and pass in a Circle, C++ calls Circle's getArea function
- And if I defined a Triangle, we'd use its getArea and getCircum methods.
- In fact, there's no real reason (at least in this example) why we'd ever want to call the Shape class's version of getArea() or getCircum()

```
void PrintInfo(Shape &x)
{
  cout << "The area is " <<
     x.getArea();
  cout << "The circumference is "
     x.getCircum();
}

int main()
{
   Square s(5);
   Circle c(10);

   PrintInfo(s);
   PrintInfo(c);
```

# Useless Functions

```
class Shape
{
public:
  virtual double getArea()  { return(0);}
  virtual double getCircum(){return(0);}
   ...
};
```

- So Shape's version of these function are just dummy functions… They return zero!
- They were never meant to be used…
- But if we don't define these functions in the base class, we can't use polymorphism as shown below with the PrintInfo(Shape &x) function.
- Because unless the Shape class has these functions defined, the code below won't work!

```
class Square: public Shape
{
public:
  virtual double getArea()
    { return (m_side*m_side); }
  virtual double getCircum()
    { return (4*m_side); }
   ...
```

```
class Circle: public Shape
{
public:
 virtual double getArea()
   { return (3.14*m_rad*m_rad); }
 virtual double getCircum()
   { return (2*3.14*m_rad); }
  …
```

```
void PrintInfo(Shape &x)
{
 cout << "The area is " <<
     x.getArea();
 cout << "The circumference is "
     x.getCircum();
}

int main()
{
   Square s(5);
   Circle c(10);

   PrintInfo(s);
   PrintInfo(c);
```

# Pure Virtual Functions

- If we left the getArea() function out of our base Shape class, as shown to the left, our code to the right wouldn't compile! Why? Because we're trying to call x.getArea(), but the Shape class has no getArea()!
- So we MUST define these functions in our base class or we can't do polymorphism…
- But these functions in our base class are never actually used – they just define the "interface" to common functions that are shared by all of our derived classes.

```
void PrintPrice(Shape &x)
{
  cout << "Cost is: $";
  cout << x.getArea()*3.25;
}

int main()
{
  Square s(5);
  PrintPrice(s);
}
```

```
class Shape
{
public:
           ?

 virtual float getCircum()
   { return (0); }
  ...
};
```

```
class Square: public Shape
{
public:
  virtual float getArea()
    { return (m_side*m_side); }
  virtual float getCircum()
    { return (4*m_side); }
   ...
```

```
class Circle: public Shape
{
public:
 virtual float getArea()
   { return (3.14*m_rad*m_rad); }
 virtual float getCircum()
   { return (2*3.14*m_rad); }
  ...
```

# Pure Virtual Functions

So what we've done so far is to define a dummy version of these functions in our base class:

```
class Shape
{
public:
  virtual float getArea()= 0;

  virtual float getCircum()= 0;

   ...
private:
};
```

But it would be better if we could totally remove this useless logic from our base class!

C++ actually has an official way to define such "abstract" functions that have no official { logic }.

We make them "pure virtual" functions. We just add =0; after the function header and get rid of its { body }.

So now, the getArea() and getCircum() functions are pure virtual within the Shape base class. They have no { code } in the base class. They simply define an "interface" of how these functions should be called (what's passed in, what they return) for use in derived classes.

# Pure Virtual Functions

- A pure virtual function is one that has no actual { code }.
- If your base class defines a pure virtual function…
- You're basically saying that the base version of the function will never be called!
- It means that the base-class version of your function doesn't (or can't logically) do anything useful.
- Therefore, your derived classes must re-define all pure virtual functions so they do something useful!
- If a class has a pure virtual function, you can't even define a regular variable with this class!
- So the code that tries to define the variable s below won't even compile
- So classes like Square and Circle MUST define useful versions of getArea() and getCircum() or you can't define regular variables with them either.
- In this example, Circle does define getArea and getCircum, so we can define variables with it!
- As you can see, the definition of c is just fine, and we can even call c's getCircum() function!

```
class Shape
{
public:
  virtual float getArea()   = 0;

  virtual float getCircum()= 0;

   ...
private:
};
```

```
class Square: public Shape
{
p
p
}
```

```
class Circle: public Shape
{
public:
  Circle(int rad){ m_rad = rad; }
  virtual float getArea()
   { return (3.14*m_rad*m_rad); }
  virtual float getCircum()
   { return (2*3.14*m_rad); }
private:
   ...
};
```

```
int main()
{
 Shape s;          // Error!

 cout << s.getArea();    // ??

 Circle c;               // OK!
 cout << c.getCircum(); // OK!
}
```

# Pure Virtual Functions

If you define *at least one* pure virtual function in a base class, then the class is called an "Abstract Base Class."

```
class Shape
{
public:
 virtual double getArea() = 0;
 virtual void someOtherFunc()
 {
   cout << "blah blah blah\n";
   ...
 }
 ...
private:
};
```

So, in the above example...
getArea is a pure virtual function,
and Shape is an *Abstract Base Class.*

# Abstract Base Classes (ABCs)

```
class Robot
{
public:
   virtual void talkToMe() = 0;
   virtual int getWeight( ) = 0;
...
};
```

```
class FriendlyRobot: public Robot
{
public:
   virtual void talkToMe()
     { cout << "I like geeks."; }
...
};
```

```
class KillerRobot: public Robot
{
public:
   virtual void talkToMe()
     { cout << "I must destroy geeks."; }
   virtual int getWeight() { return 100; }
...
};
```

If you define an Abstract Base Class, its derived class(es):

1. Must either provide { code } for *ALL* pure virtual functions,
2. Or the derived class becomes an Abstract Base Class itself!

- So Robot is an ABC – it lacks both talkToMe and getWeight implementations so there's no way you can define a Robot variable and call those functions!
- FriendlyRobot is also an ABC, because while it defines a valid talkToMe method, it still doesn't have a complete getWeight method. It's an ABC too. So you can't define variables with it either!
- KillerRobot is a regular class though. Why? Because it defines both talkToMe and getWeight, so it has complete versions of EVERY pure virtual function defined in its base class(es).
- So we can define a KillerRobot variable just fine.
- BigHappyRobot is also a complete class and you can define variables with it, since it inherits a complete talkToMe function from FriendlyRobot, and defines its own getWeight function. So it has complete versions of EVERY pure virtual function.

```
class BigHappyRobot: public FriendlyRobot
{
public:

   virtual int getWeight() { return 500; }
...
};
```

# Abstract Base Classes (ABCs)

```
class Shape
{
public:
 virtual float getArea()
  { return (0); }
 virtual float getCircum()
  { return (0); }
  ...
};
```

```
class Shape
{
public:
 virtual float getArea() = 0;
 virtual float getCircum() = 0;
...
};
```

```
class Rectangle: public Shape
{
public:
 virtual float getArea()
  { return (m_w * m_h); }
};
```

- Why should you use pure virtual functions and create ABCs?
- Because you prevent common mistakes!
- For example, what if we create a Rectangle class that forgets to define its own getCircum( ) (see just left)
- If we didn't use pure virtual methods in our base class (upper-left), our main() will compile but not work the way we want (lower left) – we'll get zero for the circumference and have a subtle bug!
- Had we made getArea( ) and getCircum( ) pure virtual (upper right), our main functions (lower right) won't even compile - we know we have a bug instantly!

```
int main()
{
 Rectangle r(10,20);

 // Looks like it works, but
 // has a subtle BUG!!
 cout << r.getCircum();
}
```

```
int main()
{
 // This results in a compiler
 // error; something is WRONG!
 Rectangle r(10,20); // ERROR!

 cout << r.getCircum();
}
```

# What you can do with ABCs

Even though you CAN'T create a variable with an ABC type…

```
int main()
{
 Shape s;  // ERROR!

 cout << s.getArea();
}
```

So to summarize, use pure virtual functions to:

(a) avoid writing "dummy" logic in a base class when it makes no sense to do so!

(b) force the programmer to implement functions in a derived class to prevent bugs

You can still use ABCs like regular base classes to implement polymorphism…

This is OK!

```
void PrintPrice(Shape &x)
{
   cout << "Cost is: $";
   cout << x.getArea()*3.25;
}

int main()
{
   Square s(5);
   PrintPrice(s);

   Rectangle r(20,30);
   PrintPrice(r);
}
```

# Pure Virtual Functions/ABCs

```
class Animal
{
public:
   virtual void GetNumLegs() = 0;
   virtual void GetNumEyes() = 0;

   virtual ~Animal() { … }
};
```

```
class Insect: public Animal
{
public:
   void GetNumLegs() { return(6); }
  // Insect does not define GetNumEyes
   …
};
```

```
class Fly: public Insect
{
public:
   void GetNumEyes() { return(2); }
   …
};
```

Animal is an ABC, since it has two pure virtual functions.

Insect is also an ABC, since it has at least one pure virtual function.

Fly is a regular class, since it has no pure virtual functions.

```
int main()
{
   Animal x;             // OK??
   Insect y;             // OK??
   Fly z;                // OK??
   Animal *ptr = &z;  // OK??
}
```

# 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)!

# Polymorphism Cheat Sheet, Page #2

```
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 notVirtualFunc() { 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
}
```

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.

# Challenge Problem: Diary Class

Write a Diary class to hold your memories...:

1. When a Diary object is constructed, the user must specify a title for the diary in the form of a C++ string.
2. All diaries allow the user to find out their title with a getTitle() method.
3. All diaries have a writeEntry() method. This method allows the user to add a new entry to the diary. All new entries should be directly appended onto the end of existing entries in the diary.
4. All diaries can be read with a read() method. This method takes no arguments and returns a string containing all the entries written in the diary so far.

(You should expect your Diary class will be derived from!)

# Diary Class Solution

```
class Diary
{
public:
    Diary(const string &s) { m_sTitle = s; }
    virtual ~Diary() { /* do nothing*/ }    // required!!!

    string getTitle() const { return(m_sTitle); }

    virtual void writeEntry(const string &sEntry)
    {
        m_sEntries += sEntry;
    }

    virtual string read() const { return(m_sEntries); }

private:
    string m_sEntries, m_sTitle;
};
```

# Challenge Problem Part 2

Now you are to write a derived class called "SecretDiary". This diary has all of its entries *encoded*.

1. Secret diaries always have a title of "TOP-SECRET".
2. Secret diaries should support the getTitle() method, just like regular diaries.
3. The SecretDiary has a writeEntry method that allows the user to write new *encoded* entries into the diary.
   - You can use a function called encode() to encode text
4. The SecretDiary has a read() method. This method should return a properly decoded string containing all of the entries in the diary.
   - You can use a function called decode() to decode text

```
Class SecretDiary: public Diary
{
public:
   SecretDiary() :Diary("TOP-SECRET")
   {
   }

   virtual void writeEntry(const string &s)
   {
      Diary::writeEntry(encode(s));
   }

   virtual string read() const
   {
      return decode(Diary::read());
   }

private:
};
```

# Challenge Problem Part 3

One of the brilliant CS students in CS32 is having a problem with your classes (let's assume you have a bug!). He says the following code properly prints the title of the diary, but for some reason when it prints out the diary's entries, all it prints is gobbledygook.

```
int main()
{
  SecretDiary     a;
  a.writeEntry("Dear diary,");
  a.writeEntry("Those CS32 professors are sure great.");
  a.writeEntry("Signed, Ahski Issar");
  Diary     *b = &a;
  cout << b->getTitle();
  cout << b->read();
}
```

What problem might your code have that would cause this?