Lecture #7

- Polymorphism
- Virtual Functions
- Virtual Destructors
- Pure Virtual Functions

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?

We know we can do this:

```
void main(void)
{
  Person p;

  LemonadeStand(p);
}
```

But can we do this?

```
void main(void)
{
  Student s;

  LemonadeStand(s);
}
```

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?";
```

I'd like to buy some lemonade.

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

Yes. I'm a person. I have a name and everything.

We only serve people. Are you a person?

Ok. How many cups of lemonade would you like?

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

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

Person

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?";
}
```

Well, you can see by my class declaration that all students are just a more specific sub-class of people.

Since I'm based on a Person, I have every-thing a Person has... Including a name! Look!

We only serve people. Are you a person?

```
class Student :
        public Person
{
pu
pr
}
```

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

private:
   string m_sName;
   int    m_nAge;
```

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(void)
   { 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();
}

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!

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 " <<

}

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(void);
   ...

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();
}


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

   SayHi(&s);
}
```

Polymorphism and Chopping

You MUST use a pointer or reference for polymorphism to work!

Otherwise something called "chopping" happens… and that's a bad thing!

Now the SayHi function isn't dealing with the original Student variable!

It has a chopped temporary variable that has no Student parts!

So right now, variable s would be "chopped".

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!

Polymorphism only works when you use a reference or a pointer to pass an object!

p

Person's Stuff

```
string getName()
  { return m_name; }
int  getAge()
  { return m_age; }
m_name "Carey"  m_age 38
```

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

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

   SayHi(s);
}
```

Polymorphism

Square has its own c'tor as well as an updated getArea function that overrides the one from Shape.

```
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;
};
```

Since all shapes have an *area*, we define a member function called getArea.

For simplicity, we'll omit other member functions/variables like getX(), setX(), getY(), getPerimeter(), etc.

Now let's consider two derived classes: Square and Circle.

```
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;
};
```

Similarly, Circle has its own c'tor and an updated getArea function.

Polymorphism

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

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


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;
}



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...

So how about if we do this...

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;
};
```

5 * 5 = 25

```
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;
};
```

3.14*10*10 = 314

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

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

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

s	m_side	5
c	m_rad	10

When you call a virtual func, C++ figures out which is the correct function to call…

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;
};
```

```
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;
}

main()
{

   Shape sh;

   PrintPrice(sh);
 }
```

sh

It works in this case too…

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 those function(s).

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

```
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;
}

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!
}
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!

Polymorphism

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

```
class Square: public Shape
{
public:
 Squar

   { r
priva
 int 
};
```

```
  { return (3.14*m_rad*m_rad); }
private:
 int m_rad;
};
```

Hmm. The user is calling the function getArea.

Since x is a Shape variable, I'll call Shape's getArea function.

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

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

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

s: m_side 5

c: m_rad 10

Lets see what happens if we forget to use the virtual keyword.

Polymorphism

When should you use the virtual keyword?

1. Use the virtual keyword in both your base and derived classes *any time* you redefine a function in a derived class.

2. Always use the virtual keyword for destructors in your base and derived classes.

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

Polymorphism and Pointers

```
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;
};
```

```
main()
{
   Square s(5);
   Square *p;

   p = &s;
   cout << p->getArea();
}
```

Polymorphism works with pointers too. Let's see!

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

Polymorphism and Pointers

```
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;
};
```

```
main()
{
  Square s(5);
  Shape *p;

  p = &s; // OK????
  ...
}
```

Question: Can we point a Shape pointer to a Square variable?

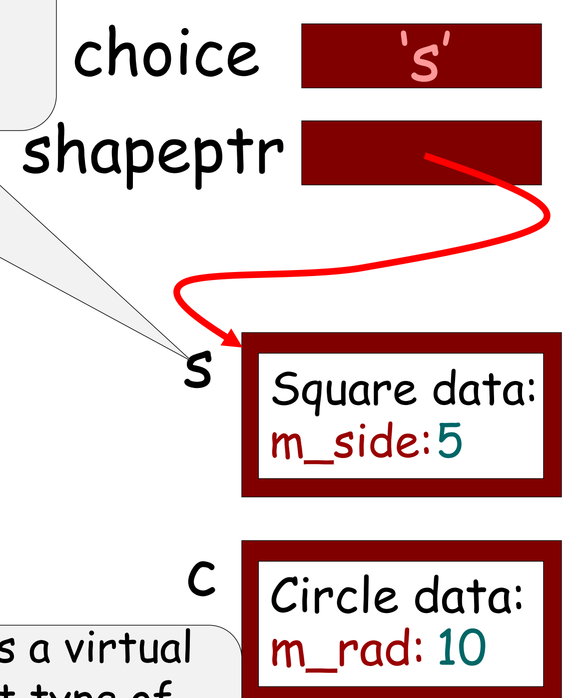
Polymorphism and P

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

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

```
main()
{
    Square s(5);
    Circle c(10);
    Shape * shapeptr;
    char choice;

    cout << "(s)quare or a (c)ircle:";
    cin >> choice;
    if (choice == 's')
      shapeptr = &s;     // upcast
    else shapeptr = &c; // upcast

    cout << "The area of your shape is: ";
    cout << shapeptr->getArea();
}
```

Aha! The shapeptr variable points to a Square. I'll call Square's getArea function.



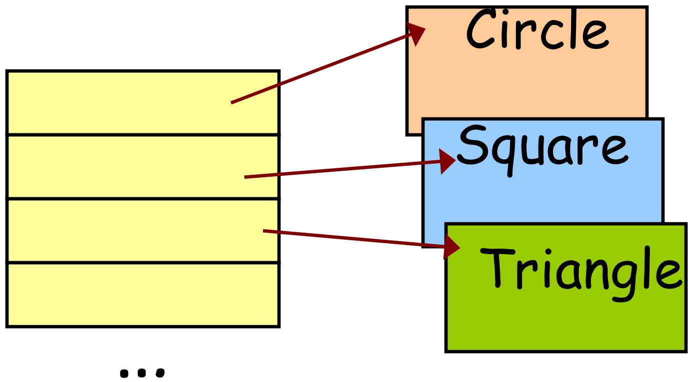
Hmm. getArea is a virtual function. What type of variable does shapeptr point to?

(s)quare or a (c)ircle: s

The area of your shape is:

Polymorphism

```
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();
    }
}
```



Now our program simply asks each object to draw itself and it does!

Polymorphism and Pointers

```
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;
};
```

```
main()
{
  Square *ps;
  Shape sh;

  ps = &sh; // OK????
  ...
}
```

Question: Can we point a Square pointer to a Shape variable?

LL!

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

C++: "Hmmm.. I'm really a HighPitchedGeek..."

C++: "And laugh() is a virtual method..."

C++: "So I'll call the proper, HighPitchGeek version of laugh()!"

```
ain()

  Geek *ptr = new
    HighPitchGeek;

  ptr->tickleMe(); // ?

  d lete 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 line is using polymorphism! We're using a base (Geek) pointer to access a Derived (HighPitchedGeek) object!

ptr

HighPitchedGeek variable

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(void)
 {
   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???)

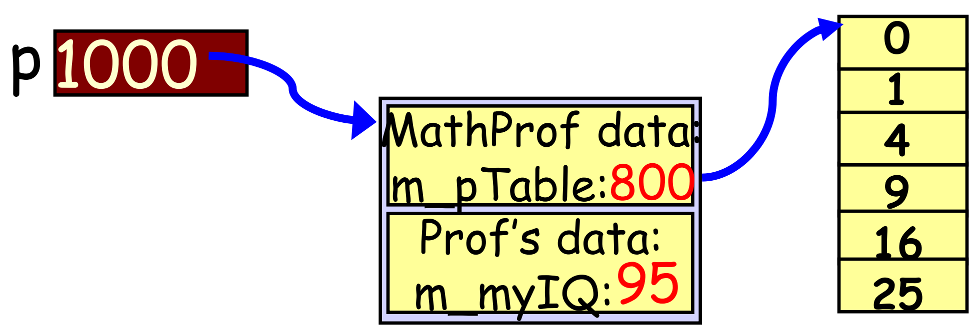
All math professors keep a set of flashcards with the first 6 square numbers in their head.

Virtual Destructors

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

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

```
class MathProf: public Prof
{
public:
 MathProf(void)
 {
   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;
};
```

```
main()
{
   Prof *p;

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



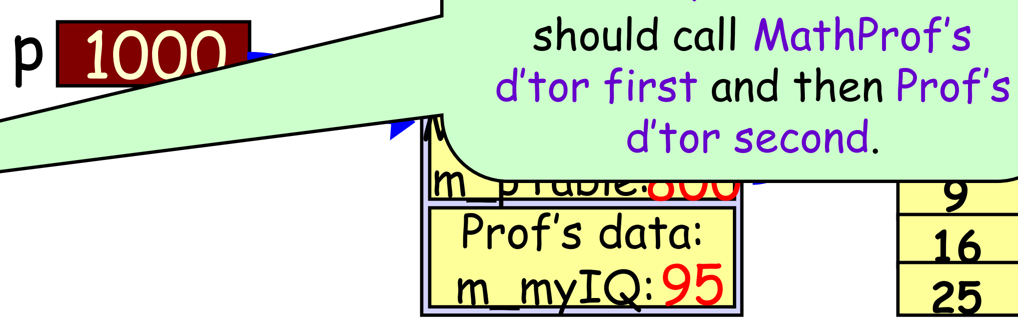
Polymorphism and Virtual Destructors

```
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(void)
 {
   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;
};
```

```
main()
{
   Prof *p;

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

Hmm. Let's see... Even though p is a Prof pointer, it actually points to a MathProf variable. So I should call MathProf's d'tor first and then Prof's d'tor second.



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(void)
 {
   m_pTable = new int[6];

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

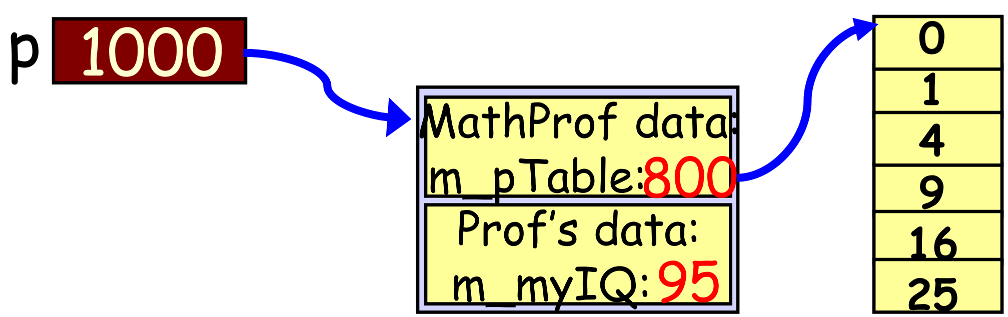
* Technically, if you don't make your destructor virtual your program will have undefined behavior (e.g., it could do anything, including crash), but what I'll show you is the typical behavior.

Polymorphism and Virtual Destructors

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

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

```
main()
{
    Prof *p;

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

```
class MathProf: public Prof
{
public:
 MathProf(void)
 {
   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;
};
```



Polymorphism and Virtual Destructors

```
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(void)
 {
   m_pTable = new int[6];

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

```
main()
{
   Prof *p;

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

Hmm. Let’s see...
The variable p is a Prof pointer.
So all I need to call is Prof’s destructor.

Utoh! MathProf’s destructor was never called and the table was never freed!

This means we have a memory leak!

Virtual Destructors – What Happens?

```
class Person
{
public:
   ...

   ~Person()
   {
     cout << "I'm old!"
   }
};
```

```
class Prof: public Person
{
public:
   ...

   ~Prof()
   {
     cout << "Argh! No tenure!"
   }
};
```

So what happens if we've forgotten to make a class's destructor virtual?

And then define a derived variable in our program?

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(void)
{
   Prof carey;

      ...

} // carey's destructed
```

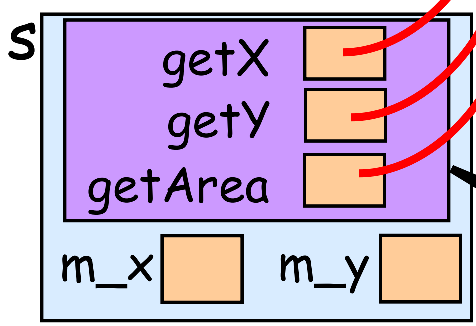
Argh! No tenure!

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.




```
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
{
publ
  vi
   {
   ...
};
```

This table is called a "vtable."

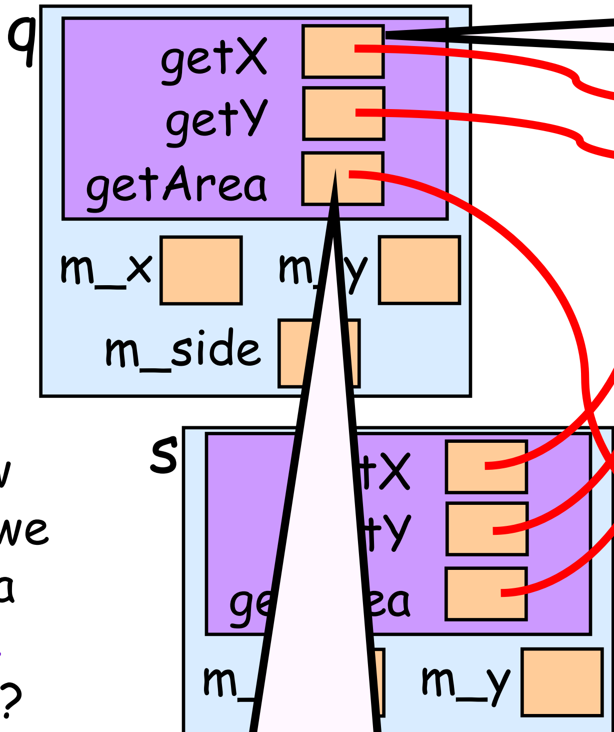
It contains an entry for every virtual function in our class.

In the case of a Shape variable, all three pointers in our vtable point to our Shape class's functions.

How does it all work?

However, our Square basically uses our Shape's getX and getY functions, so our other entries will point there.

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



Ok, how about if we define a Square variable?

```
int main()
{
    Shape s;

    Square q;

}
```

Well, our Square has its own getArea() function…

So its vtable entry points to that version…

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

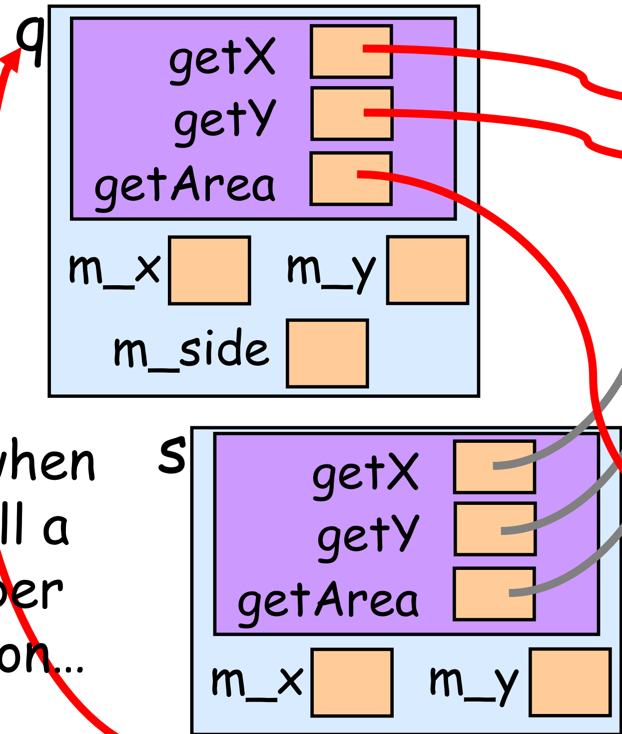
```
class Circle: public Shape
{
public:
  virtual int getArea()
   { return (3.14*m_rad*m_rad); }
   ...
};
```

How does it all work?

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.

```
virtual int getY() {return m_y;}
virtual int getArea() {return 0;}
...
};
```



Now, when we call a member function...

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

```
class Circle: public Shape
...
)
*m_rad); }
```

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

p

C++ knows exactly where to go!

It just looks at the vtable for "s" and uses the right function!

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 you write your derived classes, creating specialized versions of each common function:

Square version of getArea

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

Circle version of getArea

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

- You can access derived variables with a base class pointer or reference.
- Finally, you should (MUST) always define a virtual destructor in your 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
  virtual double getCircum(){return
  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); }
  ...
```

Question: When I call the PrintInfo function and pass in a Square, what getArea and getCircum functions does it call?

…and when I call the PrintInfo function and pass in a Circle, what getArea and getCircum functions does it call?

So here's my question:
When would Shape's getArea() and getCircum() functions ever be called?

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

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

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

Useless

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

Well, I guess they'd be called if you created a Shape variable in main…

But why would we ever want to get the area and circumference of an "abstract" shape?

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

Those are just dummy functions… They return zero!

They were never meant to be used…

```
}

main()
{
   Shape p;
   PrintInfo(p);
```

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

Pure Virtual Functions

We *must* define functions that are common to all derived classes in our base class or we can't use polymorphism!

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

But these functions in our base class are never actually used – they just define common functions for the derived classes.

```
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:
};
```

Since these funcs in our base class are never used, we could just as easily change their logic to ...

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

C++ actually has an "officially correct" way to define such "abstract" functions. Let's see how!

These are called "pure virtual" functions.

Rule: Make a base class function pure virtual if you realize:

the base-class version of your function doesn't (or can't logically) do anything useful.

Pure Virtual Functions

A pure virtual function is one that has no actual { code }.

If your base class has a pure virtual function…

Your derived classes *must* define their own version of it.

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

 virtual float getCircum()= 0;

   ...
private:
};
```

```
class Circle: public Shape
{
p


p

}
```

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

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 is Robot a regular class or an ABC?

Right! It's an ABC

How about FriendlyRobot? Regular class or an ABC?

Finally, how about BigHappyRobot? Is it a regular class or an ABC?

How about KillerRobot? Regular class or an ABC?

```
class BigHappyRobot: public FriendlyRobot
{
public:

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

Abstract Base Classes (ABCs)

Why should you use Pure Virtual Functions and create Abstract Base Classes anyway?

You *force* the user to implement certain functions to prevent common mistakes!

For example, what if we create a Rectangle class that forgets to define its own getCircum( )?

Had we made getArea( ) and getCircum( ) pure virtual, this couldn't have happened!

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

pr
};
```

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

 virtual float getCircum()
  { return (2*m_w+2*m_h); }
...
};
```

Ack- our rectangle should have a circumference of 60, not 0!!! This is a bug!

```
main()
{
 Rectangle      0,20);

 cout << r.getArea(); // OK
 cout << r.getCircum(); //?

}
```

What you can do with ABCs

Even though you can't create a variable with an ABC…

```
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…

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

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

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

Pure Virtual Functions/ABCs

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

!!Remember!! You ***always*** need a virtual destructor in your base class when using polymorphism!

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

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

```
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 notVirtuaFuncl() { cout << "Still not"; }         // #5
};

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

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

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.

```
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?