Polymorphism

Polymorphism is how you make inheritance truely useful.
It's used to implement video game NPCs, circuit simulation programs, and graphic design programs.

The idea behind polymorphism is that once I define a function that accepts a reference or pointer to a class (Person)...
Not only can I pass Person variables to that class...
But I can also pass any variable that is derived from Person
A Student is a Person. Everything a Person can do, it can do.

```
void SayHi(Person &p)
{
    cout << "Hello " << p.getName();
}
int main()
{
    float GPA = 1.6;
    Student s("David", 19, GPA);
    SayHi(s); //pass in a Student variable to the SayHi() method
}
```

The `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`!

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

```
void SayHi(Person *p)
{
    cout << "Hello " << p->getName();
}
int main()
{
    Student s("Carey", 38, 3.9);
}
```

Inheritance:
We publicly derive one or more classes from a common base class.
All of the derived classes, by definition, inherit a common set of functions from our base class.
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.
The same function function call automatically causes difference actions to occur, depending on what type of variable is currently being referred/pointed to.

When you omit the `virtual` keyword in a non-destructor function, 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.

```
class Shape

{

    public:

        virtual double getArea();

};

class Square : public Shape

{

    public:

        virtual double getArea();

};

class Circle : public Shape

{

    public:

        virtual double getArea();

};
```

```
class Shape

{

    public:

        double getArea();
```

```
};
class Square : public Shape
{
    public:
        double getArea();
};
class Circle : public Shape
{
    public:
        double getArea();
};
```

```
void PrintPrice(Shape &x)
{
    cout << "Cost is: ";
    cout << x.getArea() * 3.25; //since the virtual keyword is omitted, no matter what kind of
                                //shape is passed into PrintPrice, Shape's getArea() function
                                //is the getArea() that is called
}
int main()
{
    Square s(5);
    Circle c(10);
    PrintPrice(s);
    PrintPrice(c);
}
```

Since `x` is a `Shape` variable, and `getArea()` is **NOT** virtual in the base class, `PrintPrice()` will just call `Shape` 's `getArea()` function.
**This might be on the midterm

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.

> Always have a virtual destructor in your base class if you're doing polymorphism.

**Take a look at the animations on slide 31 of Lecture 7 to see the explanation of what happens when the destructor in your base class is not virtual.

```
class Person
{
    public:
        ...
        ~Person()
        {
            cout << "I'm old!";
        }
};
class Prof : public Person
{
    public:
        ...
        ~Prof()
        {
            cout << "Argh! No tenure!";
        }
}; //this doesn't actually cause an error
//Output:
//Argh! No tenure!
//I'm old!
//it's only when you do polymorphism with these classes that this becomes a problem
```

Missing a virtual destructor will only cause problems if you're using polymorphism because then you're deleting through a pointer.

C++ will only call Person's destructor since p is a Person pointer and Person's destructor isn't virtual.

```
int main()
{
    Person *p = new Prof;
    ...
    delete p; //problem!
}
```

It's optional to make destructors of the derived class virtual, but it's good practice!

```
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);
        }
};
```

When you define a variable of a class, C++ adds an invisible table to your object that points to the set of functions to use. The table ("vtable") has one entry for every virtual function. All three pointers in the vtable of a Shape variable point to all three virtual functions.
If we define a Square variable, its vtable still has three entries, but while getX() and getY() point to Shape's functions, getArea() will point to Square's function.

Summary of Polymorphism:

- First we figure out what we want to represent
- Then we define a base class that contains functions common to all of the derived classes
- Then you write your derived classes, creating specialized versions of each common function

- You ccan access derived variables with a base class pointer or reference.
- **Finally, you must always define a virtual destructor in your base class, whether it needs it or not.

Pure Virtual Functions
We must define functions that are common to all derived classes in our base class or we can't use polymorphism!
But these functions in our base class are never actually used - they just define common functions for the derived classes.

```
class Shape

{

    public:

        virtual float getArea() { return 0; }

        virtual float getCircum() { return 0; }

}
```

It would be better if we could totally remove this useless logic from our base class!
C++ has an "official" way to define such "abstract" functions.

```
virtual float getArea() = 0;
```

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!
Therefore, derived classes must re-define all pure virtual functions so they do something useful!

> Make a base class function pure virtual if you realize the base-class version of your function doesn't do anything useful.

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

If you define an abstract base class, its derived classes

1. Must either provide code for all pure virtual functions
2. Or the derived class becomes an abstract base class itself!

**You can't define a regular variable with an abstract base class

You should use pure virtual functions and create abstract base classes to force the user to implement certain functions to prevent common mistakes!

Even though you can't create a variable with an ABC, you can still use ABCs like regular base classes to implement polymorphism.

**You always need virtual destructor in your base class when using polymorphism.

```
class Animal

{

    public:

        virtual void GetNumLegs(void) = 0;

        virtual void GetNumEyes(void) = 0;

        virtual ~Animal() {...}

};
```

Cheat Sheet
*You can't access private members of the base class from the derived class. (children can't touch their parents' privates)
*Always make sure to add a virtual destructor to your base class.
*Don't forget to use `virtual` to define methods in your base class, if you expect to redefine them in your derived classes.
*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)!
*To call a base-class method that has been redefined in a derived class, use the `base::` prefix.

```
class SomeBaseClass

{

    public:

        virtual void aVirtualFunc()

        {

            cout << "I'm virtual";

        }

        void notVirtualFunc()

        {

            cout << "I'm not";

        }

        void tricky()

        {
```

```
            aVirtualFunc();
            notVirtualFunc();
        }
};

class SomeDerivedClass : public SomeBaseClass
{
    public:
        void aVirtualFunc()
        {
            cout << "Also virtual";
        }
        void notVirtualFunc()
        {
            cout << "Still not";
        }
};

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

    //calls the derived class's aVirtualFunc()
    cout << b->aVirtualFunc(); //"Also virtual"
    //calls the base class's notVirtualFunc()
    cout << b->notVirtualFunc();
    //calls the base class's tricky() function
    //which calls derived class's aVirtualFunc(),
    //and then the base class's notVirtualFunc()
    b->tricky();
```

```
}
```

When you use a **base** pointer to 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.

When you use a **base** pointer to access a **derived** object, and you call a non-virtual function, your code will call the **base** version of the function.

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, if if the function calling the virtual function is **not virtual** itself.

Diary Class

```
class Diary

{

    public:

        Diary(string title)

        {

            m_title = title;

        }

        string getTitle() const

        {

            return m_title;

        }

        virtual void writeEntry(const string &entry)

        {

            m_sEntries += entry;

        }

        virtual string read() const

        {

            return m_sEntries;

        }

        virtual ~Diary()

        {



        }
```

```
    private:
        string m_title;
        string m_sEntries;
}
```

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

        }
        virtual void writeEntry(const string &entry)
        {
            Diary::writeEntry(encode(s));
        }
        virtual string read() const
        {
            return decode(Diary::read());
        }
}
```