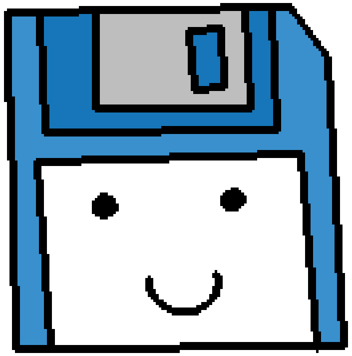
Polymorphism

Written by Rachel J. Morris, last updated 2016-03-13

Before we get into virtual functions, let's review a couple things about classes and inheritance...



Pointers to Classes

We might want to have a pointer of the base class' type, and point to any children, to write generic code that can handle any specialization...

```
Bear* ptrBearFamily = &panda1;
ptrBearFamily->EatFish();
```

```
Bear eats fish
```

So how do we make sure that the function called is the one that belongs to

Function Overriding

#1

When you have one class inherit from another class, you can **override** (or overwrite) a parent class' functions with new functionality:

```
class Bear
{
    public:
    void EatFish()
    {
        cout << "Bear eats fish" << endl;
    }
};
```

Original

```
class Panda : public Bear
{
    public:
    void EatFish()
    {
        cout << "Panda eats fish" << endl;
    }
};
```

Overwritten

Pointers to Classes

#2

We can also utilize pointers to point to a specific class

```
Bear bear1;
Panda panda1;

Bear* ptrBear = &bear1;
Panda* ptrPanda = &panda1;
```

When two classes are related, we can create a pointer for the parent data-type, and point it to a child's address.

```
Bear* ptrBearFamily = &panda1;
```

However, if we do this, when we call the overwritten function, it will call the original parent version.

```
Bear* ptrBearFamily = &panda1;
ptrBearFamily->EatFish();
```

```
Bear eats fish
```

Pointers to Classes

We might want to have a pointer to the base class, and point to any children, to write generic code that can handle any specialization...

```
Bear* ptrBearFamily = &panda1;
ptrBearFamily->EatFish();
```

```
Bear eats fish
```

So how do we make sure that the function called is the one that belongs to the correct data-type, and not just the generic parent?

```
class Bear
{
    public:
    void EatFish()
    {
        cout << "Bear eats fish" << endl;
    }
};
```

```
class Panda : public Bear
{
    public:
    void EatFish()
    {
        cout << "Panda eats fish" << endl;
    }
};
```

Virtual Functions

Virtual Functions

```cpp
class Bear
{
    public:
    virtual void Roar()
    {
        cout << "Bear roar" << endl;
    }

    void EatFish()
    {
        cout << "Bear eats fish" << endl;
    }
};
```

```cpp
class Panda : public Bear
{
    public:
    virtual void Roar()
    {
        cout << "Panda roar" << endl;
    }

    void EatFish()
    {
        cout << "Panda eats fish" << endl;
    }
};
```

Here we have a parent class and a child class.

In the parent class, there is one **virtual** function, and one non-virtual function. Both of these are overwritten in the child class.

```cpp
Bear* ptrBear = &bear1;
Bear* ptrPanda = &panda1;
```

We will have a `Bear*` pointer to a `Bear` object and a `Panda` object.

Virtual Functions

```cpp
class Bear
{
    public:
    virtual void Roar();
    void EatFish();
};
```

```cpp
Bear* ptrBear = &bear1;
Bear* ptrPanda = &panda1;
```

```cpp
class Panda : public Bear
{
    public:
    virtual void Roar();
    void EatFish();
};
```

What happens if we call each function through the pointers?

Simple Polymorphism

```
Bear roar
Panda roar
Bear eats fish
Bear eats fish
```

```cpp
ptrBear->Roar();
ptrPanda->Roar();

ptrBear->EatFish();
ptrPanda->EatFish();
```

Notice that with the non-virtual function, the `Bear` version gets called both times.

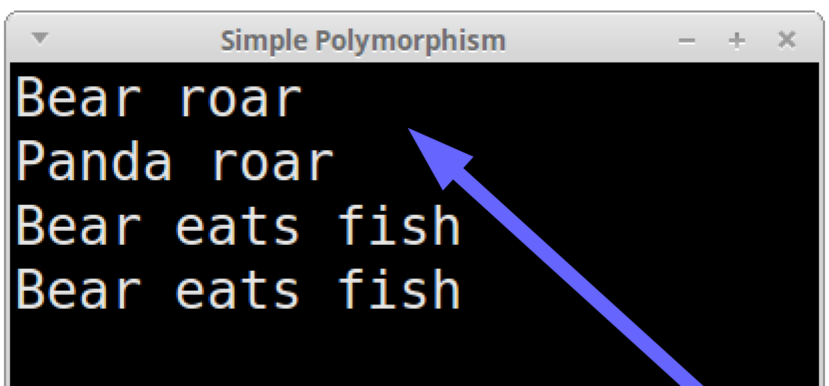
Virtual Functions

```
class Bear
{
    public:
    virtual void Roar();
    void EatFish();
};
```

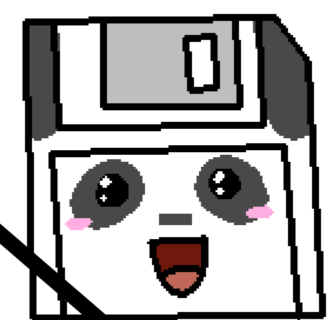
```
Bear* ptrBear = &bear1;
Bear* ptrPanda = &panda1;
```

```
class Panda : public Bear
{
    public:
    virtual void Roar();
    void EatFish();
};
```

What happens if we call each function through the pointers?




```
ptrBear->Roar();
ptrPanda->Roar();

ptrBear->EatFish();
ptrPanda->EatFish();
```



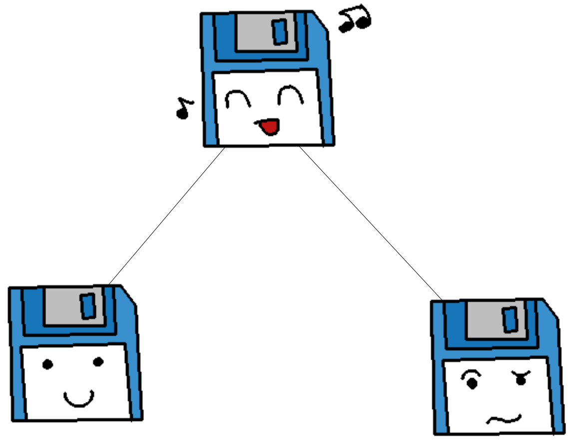
But with the **virtual** function, even though the pointer is a `Bear*` type, the `Panda` version is called.

Virtual Functions

```
class Panda : public Bear
{
    public:
    virtual void Roar();
    void EatFish();
};
```

By utilizing **virtual functions**, we can now write programs that utilize pointers to handle many different classes that have a common parent.

The program doesn't care about what exactly the class is, but can operate on the **specialized implementations** (child classes) via the
common interface, which was defined by the parent class.



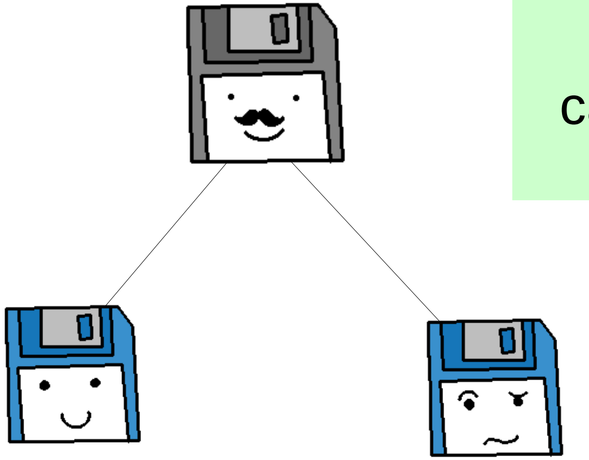
Abstract Classes & Pure Virtual Functions

Abstract Classes

By utilizing **virtual functions**, we can now write programs that utilize pointers to handle many different classes that have a common parent.

Sometimes, we don't want the user to actually *create* an instantiation of the parent object – perhaps it is just defining the **interface** that all the children will use.

We can make a class into an **abstract class**, so that it cannot be instantiated as an object – but its children can.



Abstract Classes

```
// Document Interface
class IDocument
{
    public:
    virtual void Load( const string& filename ) = 0;
    virtual ofstream Display( ofstream& out ) = 0;
};
```

If we were writing a program to handle multiple types of Documents on a computer, we might know that they all share a common functionality.

We can declare a **`Load`** and **`Display`** function in the class as
pure virtual functions
by ending the declared line with a `= 0;`

By doing this, we can no longer create an instantiation of `IDocument`, but when we create a child of `IDocument`, it is <u>required</u> that we define these functions in each child.

Abstract Classes

```
// Document Interface
class IDocument
{
    public:
    virtual void Load( const string& filename ) = 0;
    virtual ofstream& Display( ofstream& out ) = 0;
};
```

If we were writing a program to handle multiple types of Documents on a computer, we might know that they all share a common functionality.

We can declare a **`Load`** and **`Display`** function in the class as **pure virtual functions** by ending the declared line with `a = 0;`

By doing this, we can no longer create an instantiation of `IDocument`, but when we create a child of `IDocument`, it is <u>required</u> that we define these functions in each child.

Abstract Classes

```
// Document Interface
class IDocument
{
    public:
    virtual void Load( const string& filename ) = 0;
    virtual ofstream& Display( ofstream& out ) = 0;
};
```

```
class TextDocument : public IDocument
{
    public:
    virtual ofstream& Display( ofstream& out );

    private:
    string m_content;
};
```

(missing `Load`)

```
8        TextDocument text;
```

```
8      error: cannot declare variable 'text' to be of abstract type 'TextDocument'
```

If we do not overwrite these pure virtual functions within a child class, then the child class is also abstract and cannot be instantiated.

Abstract Classes

```
class TextDocument : public IDocument
{
    public:
    virtual void Load( const string& filename );
    virtual ofstream& Display( ofstream& out );

    private:
    string m_content;
};
```

```
class CsvDocument : public IDocument
{
    public:
    virtual void Load( const string& filename );
    virtual ofstream& Display( ofstream& out );

    private:
    vector<string> m_rows;
};
```

```
class WebDocument : public IDocument
{
    public:
    virtual void Load( const string& filename );
    virtual ofstream& Display( ofstream& out );

    private:
    ofstream OutputHeader( ofstream& out );
    ofstream OutputFooter( ofstream& out );

    vector<Element> m_elements;
};
```

Each type of document might be implemented wildly differently, but we know that each one needs a **`Load`** function and a **`Display`** function, so we enforce it via the common parent class, the `IDocument` interface.

But what good is having an interface if we have to write duplicate code for each child?

```cpp
int main()
{
    TextDocument text;
    CsvDocument csv;
    WebDocument html;

    text.Load( "file.txt" );
    csv.Load( "spreadsheet.csv" );
    html.Load( "page.html" );

    // Blah blah blah

    return 0;
}
```

Well, we don't have to...

```
IDocument* document;

if ( doctype == 1 )
{
    document = new TextDocument;
}
else if ( doctype == 2 )
{
    document = new WebDocument;
}
else if ( doctype == 3 )
{
    document = new CsvDocument;
}

document->Load( filename );
```

Using pointers, we can treat <u>all documents</u> like an `IDocument`, and our code doesn't need to care or differentiate between each child type.

We point to a child, but our pointer is of the type parent`*`...

Polymorphism

Polymorphism

Now we get into **Polymorphism**

```
IDocument* doc = NULL;

if ( ext == "txt" )           { doc = new TextDocument; }
else if ( ext == "csv" )      { doc = new CsvDocument; }
else if ( ext == "html" )     { doc = new WebDocument; }
else
{
    cout << "Unknown file type" << endl;
    continue;
}

doc->GetInput();
doc->Save( filename );
delete doc;
```

Each `IDocument` child has `GetInput()` and `Save(...)`

Utilizing an abstract base class is not required to use polymorphism; we can use a pointer of any base-class-pointer type

The main idea is that we can write generic code.
No need to duplicate the same code for each type of child;
we simply treat them all the same through their common parent interface.

Polymorphism

There is a common interface, but each child class (**specialization**) can have additional functions to get its job done.

```
class TextDocument : public IDocument
{
    public:
    virtual void Load( const string& filename );
    virtual void Save( const string& filename );
    virtual ofstream& Display( ofstream& out );
    virtual void GetInput();

    void SetContent( const string& content );

    private:
    string m_content;
};
```

```
class WebDocument : public IDocument
{
    public:
    virtual void Load( const string& filename );
    virtual void Save( const string& filename );
    virtual ofstream& Display( ofstream& out );
    virtual void GetInput();

    void AddElement( const string& element );

    private:
    void OutputHeader( ofstream& out );
    void OutputFooter( ofstream& out );

    vector<Element> m_elements;
};
```

Polymorphism

There is a common interface, but each child class (**specialization**) can have additional functions to get its job done.

```
class TextDocument : public IDocument
{
    public:
    virtual void Load( const string& filename );
    virtual void Save( const string& filename );
    virtual ofstream& Display( ofstream& out );
    virtual void GetInput();

    void SetContent( const string& content );
```

```
void TextDocument::Save( const string& filename )
{
    cout << "Text document save at "
        << filename << endl;

    ofstream output( filename.c_str() );
    output << m_content;
    output.close();
}
```

Each specialization has its own inner-workings, but to the program in general, all it cares about is that a "Save" function can be called.

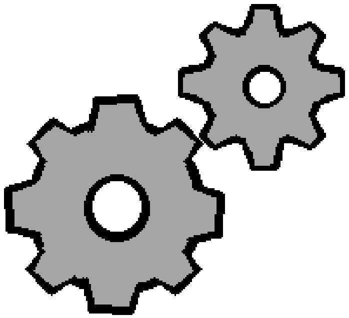
```
class WebDocument : public IDocument
{
    public:
    virtual void Load( const string& filename );
    virtual void Save( const string& filename );
    virtual ofstream& Display( ofstream& out );
    virtual void GetInput();
```

```
void WebDocument::Save( const string& filename )
{
    cout << "Web document save at " << filename << endl;

    ofstream output( filename.c_str() );

    OutputHeader( output );

    for ( int i = 0; i < m_elements.size(); i++ )
    {
        output << m_elements[i].html << endl;
    }

    OutputFooter( output );

    output.close();
}
```

Late Binding

Late Binding, or **Dynamic Binding**,
occurs when we utilize the **virtual** keyword.

This causes the program to figure out
which function to call at **run-time**.



The Virtual Table

Late Binding utilizes something called the **Virtual Table**, which is often also called the **vtable**.

This is essentially a table of functions declared as **virtual**.

The **vtable** contains **pointers to functions**, which we have not learned how to implement ourselves yet.

Every class that has virtual functions (either declared in its parent or in itself) has a **vtable**.

The Virtual Table

This is why the correct function gets called even if the pointer is of a parent* type.

```
class Bear
{
    public:
    virtual void Roar()
    {
        cout << "Bear roar" << endl;
    }

    void EatFish()
    {
        cout << "Bear eats fish" << endl;
    }
};
```

If the function is not **virtual**, the function called is the one that is the same data-type as the pointer.

```
ptrBear->Roar();
ptrPanda->Roar();

ptrBear->EatFish();
ptrPanda->EatFish();
```

Simple Polymorphism

```
Bear roar
Panda roar
Bear eats fish
Bear eats fish
```

Virtual

Not virtual

Virtual Destructors

When using **virtual functions** and **inheritance**, you will want to make sure that any **Destructor** that you create is **virtual.**

We want to make sure that the correct **destructor** is called when an item is destroyed.

```
class Canine
{

class Corgi : public Canine
{

int main()
{
    Canine* corgi = new Corgi( "Ein" );
    delete corgi;

    return 0;
}
```

Virtual Destructor

```
Canine Constructor
Corgi Constructor
Canine Destructor
```

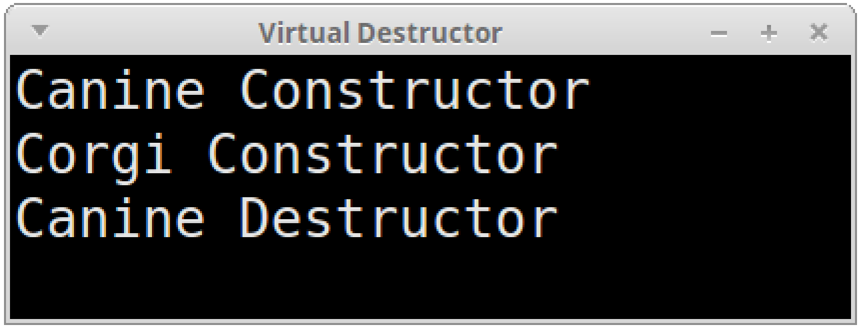
Notice that only the **Canine** destructor is called!

Virtual Destructors

If we created a dynamic variable with the **new** keyword, using a Parent*, without the **virtual** keyword added to the destructor it would try to call the parent's destructor!

This could cause memory problems if the child class allocates any memory as part of its implementation,

If the parent class destructor is called, then any memory managed by the child will not be handled appropriately!



Virtual Destructors

```cpp
class Canine
{
    public:
    Canine( const string& name )
    {

    virtual ~Canine()
    {
        cout << "Canine Destructor" << endl;
    }

    protected:
    string m_name;
};

class Corgi : public Canine
{
    public:
    Corgi( const string& name ) : Canine( name )
    {

    virtual ~Corgi()
    {
        cout << "Corgi Destructor" << endl;
    }
};
```

Virtual Destructor

```
Canine Constructor
Corgi Constructor
Corgi Destructor
Canine Destructor
```

With all destructors being **virtual**, each destructor in the family will be called.

Without this, there could be memory handled in one class somewhere in the family, but never freed.

So now that we've covered all this information about virtual functions, abstract classes, and polymorphism,

let's put it into practice with a sample program!