Polymorphism

So many morphs...

Keith Lancaster, Ph.D.

Department of Computer Science and Engineering

University of Nevada, Reno

Types of Polymorphism

- **Subtype**: implemented in C++ through inheritance. Each subclass can implement its own version of a function defined in a parent class
- **"Ad Hoc", or function overloading (*not overriding!*)**: creating functions that have the same name but different signatures, so that they can act on different parameter types
- **Parametric:** a way of writing a function or data type in a generic way that can be *parameterized* at *compile time* to work a specific type. In C++, this is implemented via templates
- **Coercion**: an object is explicitly or implicitly *cast* to another type

See `https://catonmat.net/cpp-polymorphism`

Subtype Polymorphism
(aka Runtime Polymorphism)

- Through *virtual* functions, we can have each subclass have its own behaviour for a given function.
- We have seen this already with the `Animal` -> `Cat` examples
- We must be able to substitute any subtype for a parent and not break anything. This is known as the *Liskov Substitution Principle*

Let's look at some code...

Ad Hoc / Overloading

What if I told you that I needed a method called `doubleMe` that could handle `int`s or `float`s or even `string`s?

- for ints, I just want 2 times the number
- same for floats
- for strings, I just want to duplicate what was given (”john” becomes ”johnjohn”)

This is where *overloading* can come into play.

Overloading

```
1 int doubleMe(int n) { return n * 2};
2 std::string doubleMe(const std::string& input){
3     return input + input;
4 }
```

Coercion

This occurs when one type, say an int, is cast into a float:

```
float x = 4;
```

This is *implicit*, meaning that we did not say someplace ”Hey, convert this here int to a float”.
In C, we might *explicitly* cast a variable by doing something like this:

```
float x = (int) 4;
```

In C++, we might *explicitly* cast a variable by doing something like this:

```
float x = static_cast<float> 4;
```

Coercion and Constructors

What is going on here?

```
1  class MyClass{
2  public:
3      MyClass(int n)...
4      void print();
5  }
6  void doSomething(MyClass m) {
7      m.print();
8  }
9  // in main...
10 doSomething(20);
```

Code Demo

Let’s do some experimentation…

What about parametric?

You said there were four. I count only three. What gives?

Templates are their own special animal (no pun intended). Stay tuned…