Header Text

Probably should have scripts to check the code meets these constraints

Discussion:

Pure Virtual Functions

- If add a pure virtual function to an existing class (eg: virtual blah() = 0)
  then if the class has concrete methods already, it is called an Abstract Base Class
- If the class is just made up of pure virtual functions, it is an Interface
- An Abstract Base Class is kind of a mix of class and Interface
- Code is clearer if it is either an interface or a concrete class
- Might be nicer to move pure virtual functions in to just interfaces
- And then make abstract classes inherit interfaces to define their pure virtual methods
- Not sure this can work in practise because the concrete methods want to call
  the interface methods, but then i think can't compile

Rules:

Only classes inheriting from InterfaceBase or AbstractClassBase can contain pure virtual functions

If inherit from InterfaceBase, class name must end with 'Interface' and must only contain pure virtual functions

If inherit from AbstractClassBase, class name must start with 'Abstract' and must contain a mix of pure virtual and concrete functions

Discussion:

Non-pure Virtual Functions

- causes object instances of the class to require a vtable pointer so increases the size of the objects
- requires that there is a virtual destructor if polymorphically deleted
- possible exceptions may be if only ever capture in a shared_ptr and delete via it or never polymophically delete
- however can never really ensure these constraints across a large project as more class inherit etc
- safer to enforce that it must require a virtual destructor

- it is fine if the base class or base of base etc has a virtual destructor - do we need a naming convention?

Rules:

TODO: need a rule that can be verified to enforce this, a naming convention for polymophic classes?

Discussion:

Callbacks

- In C code, a common pattern is to pass a function pointer to another function as a callback
- often the callback will take parameters and commonly they might be void* for some user defined data
- this is not type safe - common cause of programming error is passing the wrong type/value but can't be catched by compiler if void* cast
- has advantage in that it is simple and compact - doesn't generate more versions of the code just for different types
- alternatives include using templates and using polymophism
- polymorphic way is to pass an object with virtual functions to the function instead of callback and invoking the virtual functions
- this can be through an interface or some concrete class with virtuals.
- the way with templates is to make the callback function a template parameter, but exposes the details in header and makes more versions of the function
- advantage of templates here is the possibility for inlining and better performance than even C code that uses callbacks
- perhaps using type erasure it is possible to reduce the permutations from exploding if there are templates upon templates
- better if performance is not critical to use the polymohpic approach, but in performance critical code paths, templates are best

Rules:

Don't cast to void*

Discussion:

Naming

Header Text

- consistent naming conventions help distinguish types from members etc
- consistent naming helps to align expectation of what a named thing is
- reduces cognitive load
- …

Rules:

Use camel case and not use under-scores
Names of Types / Classes etc should start with a capital letter
Member functions start with a lower case letter
Getter functions shouldn't start with get, but just be the name of the attribute
Setters should start with set, eg void setMyProp(const MyProp& prop)
See other rules about naming of interfaces and abstract classes
Functions that check a state need to be carefully named to not imply that change the state
Functions that change a state should be carefully names to not imply that just check a state
eg: Take a checkbox example, bool isTicked() const; this is clear it is testing the state, not changing it
void setTicked(bool ticked); this is clear it is changing the state. void setTick(); and void clearTick(); or void unsetTick(); are also okay.
Bad is something like: bool tick(); or void tick(); the name is ambiuous as to if it sets it ticked or it tests it is ticked.
Avoid ambuguinity but while remaining concise
If a function name is long and complicated, perhaps it needs to be broken up or it doesn't need to be a seperate function. If the name is
not concise, perhaps it is a helper function for another function, in this case do not make it public, hide it as private and give it a name that
explains it is a helper function to another function by naming it with this, eg: public: void foo(); private: void fooHelper();