2/23

PriorityQueue of events will all be member function events in Project 2. Gotta use Chase's smart pointer

we now want to design a more generic FunObj

the FunObj for traditional C style we might have needed to make it a template; however, here we don't b/c the operator () is of type void and does not take any argument(s). Yet for the FunObj that is for PMemFun we are gonna use template cause of the receiver obj.

BasicFunObj, MemFunFunObj -> both inherit from one pure virtual class called FunObjBase that has the operator() method

```
std::vector<FunObjBase*> queue; // Chase hasn't put * yet

LifeForm bug;

MemFunFunObj<LifeForm> fun { bug, &LifeForm::doit };
queue.push_back(&fun);
// OR
queue.push_back(new MemFunFunObj<LifeForm> {bug, &LifeLorm::doit });
while (!queue.empty()) {
        auto& fun = queue.front(); // compiler declares the type on the left hand side! without & it will give you compile error (b/c FunObjBase is pure virtual)
        (*fun)(); // gross, gonna change it to accept fun();
}
```

auto keyword - tells the compiler to set the type. auto x = something. Boom, compiler knows what type of variable x needs to be.

Side Note: in a derived constructor, base class should be initialized after colon and before the body of the derived constructor starts:

```
        MemFunFunObj(T& o, PMemFun f) : obj(o) {
                pmfun = f; // this is only in derived
```

```
	}
```
References need to be done before the body of a constructor with the : syntax. Something about initializing and assigning

Now he's making Fun to add another layer of indirection. Basically just takes the pointer and make the syntax for operator() look nicer by already doing the (*ptr)(); Will probably be inlined.

In general a wrapper, should have all the behavior of the concrete obj. Let's say if we were wrapping the Shape obj, we needed the interface of draw() method etc. to be duplicated in the wrapper.

Keep in mind that the code in the wrapper (Fun in this example) will be inlined, so we don't expect an overhead for our level of indirection.

We decoupled the PMemFun from Liveform by bringing it to the global scope and declare it as a template.

The Fun class we have is conceptually a Factory! It's the constructors that are overloaded with different parameters for different types of FunObjs

Want objects on the heap because they can be different sizes b/c polymorphic/inheritance stuff. std::shared_ptr is great, always runs the right destructor BUT use Chase's smart pointer

If you have operator overloading in a class, you usually do not like the clients to dereference objs before calling those operators (so wrapper is a good choice here.)

Use std::function instead of Chase's created FunObj for project 2. A bit of a different syntax

Lambda:

```
template <typename Fun>
void doSomething(Fun f) {
	f(); // Fun is a class with the method header - void operator()(void)
}

class Silly {
public:
	void operator()(void) {
		cout << “HW!\n”;
	}
}

int main(void) {
```

```
        Silly blah;
        doSomething(blah);
}
```

anonymous classes in Java somehow address this problem. But still ugly

Lambda -> makes the compiler write the darn silly class for you!

auto blah = [] (void) { cout << “HW!\n”; }; // [] - signals a lambda, can put objects in there for state
doSomething(blah);

we need the lambda syntax [] to support setting the state of the Silly unwritten class.

Foo2 f;
auto blah2 = [& f] (void) {cout << “...”;};
doSomething(blah2);

2/25

2 hrs for 2 questions, Tuesday night… Tuesday sometime. Maybe 3 windows of 2 hours

Discrete time step simulations - good for science stuff, at each time step update everything, uniform time step lengths
discrete event simulations - moves when an event happens, time is not discrete or uniform (like: from t=10 to t=13, etc.)

We also are going to not discretize space/2D grid

sort events by timestamp (in a priority Q)

Event::now() // for debugging, timestamp of last event processed, 0 initially

Events are expected to be on the heap and they delete themselves when they execute, so don't make local Events/use new (right? yes)

Event* **p** = new Event {1.0, &doSomething }; // keeping pointers is OK if you need to (you don’t really need to in this example) //Use this pointer below to cancel an event from happening

```
new Event {0.5,
        [p] (void) {
                cout << "Something" << endl;
                p->cancel(); // Cancels the event the pointer p (above) is pointing to
```

```
        }
};
```
//Use lambda for events, will happen. And get Nima to check your syntax (Lol Jo you're funny! d=)
[] = capture expression, this one is empty

Craig::hunt(void) shows a good example of how to create an event at the end of the method. Need to use a SmartPointer instead of just using a dumb pointer like *this**

LifeForm Objs are not supposed to take advantage of the global time (basically in this project they don't know it) -> so instantiating an event with 10.0, means 10 from now (relative)

WTF is a capture vs just passing it as a parameter? Capture = will be treated as data members in the "Silly" class that the compiler creates, used for state, held onto. Parameter = it needs to be provided when the method is executed, only used within the method

QuadTrees, yay! Chase is drawing. Craig, whatever. So is Kevin (hopefully)
Invariants

- For each box, there's at most one object. If there's more than one, gotta divide up the box into four "equal" quadrants. Keep dividing until this invariant is restored. Should not put objects right on top of each other, segfault!

Purpose - is there anyone close to me? the closest? (Chase is sounding like that Sicilian guy in the Princess Bride, Vizzini) Answer in roughly log time. Yay, get a lot of pruning, just like Sarfraz likes
Movement

- don't get to know where you are located (x & y)
- set course - course is in radians, 0 = to the right. Protected, need to write it myself
- ask the quadtree how far to my boundary? returns a distance, divide by speed & boom, you get a time. Time tells you when to schedule the event for crossing the border
- You CAN fall off the edge of the world

calculate my new position
update the obj
cancel the previous event
create the new one

3/1

Each LifeForm should have one event in the event queue for when it crosses the next border (except for maybe the age event)

We need to ensure the obj is alive! Always…

when one obj passes a border, quadtree may divide the space more to satisfy one elem per cell. this action may cause other objs' infos to be out of sync with the new divided space. so, quadtree invokes the region resize callback on all the affected objs (takes care of it for us).
-> so basically, border_cross of X may result in QTree invoking region_resize of Y

- there can be at most 3 objs affected when an obj moves (a region that now has two instead of one & the old region that now has one instead of two) ->X Y and Z are affected

Possible problem - a region resize that causes a LifeForm to cross a border it didn't expect to. X enters a region and a dividing happens. Y is updated and it has just crossed the newly formed boundary. Solution - careful that update_time & position stored are managed correctly. Check the delta_time (he told us this in the manual)

Collision detection
Modeling problems

- Totally OK to not detect a collision if no boundary is crossed even if they are both right by the boundary. Too hard to model that
- Run into a LifeForm Y right after X crosses a border & are very close. So the tree divides A LOT. When X tries to run, it crosses a lot of borders still within range of Y. Just has to run away A LOT.
- Three LifeForms within range of encounter - ONLY fight the closest LifeForm in encounter distance. Do not encounter again until you cross another border

For detecting collision, we should ensure that the pair of objs within the range of 1 distance unit, have updated positions. If you update the position and found they are no longer in the 1 distance unit, you are done. No need of finding the 2nd next obj….

Encounters

- When encountering, you get an ObjInfo - sterilized description of who you bumped into
- Get back two actions - one from each combatant. The actions will either be eat or ignore
- Both ignore or only one successfully eats = straight forward
- Hard - both attempt to eat each other & both succeed. Tie breaker! Do Params tie breakers & take care of all 5 cases

Death - Check is_alive flag constantly! Even in encounter resolution. (b/c X may die during execution of its encounter method and when it returns, Y cannot eat it anymore.)

Perceive

- Can be invoked whenever the LifeForm wants
- have to update_position on yourself, like always.

- Other objects' positions are stale in the QuadTree & information returned will be stale. Could update those returned objects positions (loop through & do update_position) but you don't have to
- As you get closer to an object, the data is less stale b/c the regions are smaller AKA borders are being crossed more often
- Suicide - just perceive until death so they avoid being eaten (Chase don't like it)

Reproduce

- Sexual (Why?) or asexual
- LifeForms can give birth to other types of LifeForms (Craig give birth to an Algae)
- Decide whether or not to add the given LifeForm should be added to the simulation. is_alive is by default false, reproduce can decide to turn it to true. (basically we cannot prevent the derived (client) classes from calling new, but is_alive is our private field and we only simulate the objects we want)
- Put within encounter distance of the parent but hopefully not within distance of another LifeForm. If not possible, just try 5 times then too bad.
- There is a minimum cooldown between reproduction to prevent zerg rushes

Species name - whatever you want, used for communication
Player name - MY name/EID, has to be real, can't lie

Can submit/turn in more than one LifeForm

Creating instances of classes that we don't know it's name.

3/3

Project 2a - de-emphasize collisions. encounter() will be tested simply for grad student. No corner cases

perceive - can be stale data, more fun when it's not

Punting/factory/making no progress

Have each derived class add themselves to the universal map with an initializer class. Java - you would use a static initializer class. Reflection is a good problem solver too

static - on a global variable = local variable to said file?? wtf...

Global variable object construction order shouldn't matter. They are constructed in the order they appear in the file. If the universal map isn't created before the initializers, race condition! So use a function, getMap(), to make sure the map is set up and return a static map (static has way too many meanings, Chase wants shared). The static map will be constructed only once & the getMap() function will always refer to the same map.

smart pointer vs wrapper - wrapper you feel like you're using an object directly, smart pointer acts like a pointer.

Wrapper - can change from one type to another in inheritance hierarchy. HAVE to use a wrapper

3/8

Test - two take home questions - one for each project. Rules - have two hours to upload answers. Must complete before 11AM thursday. Two separate files, .cpp files (one for each question). Make sure to put header. 30% of midterm score. One submission! Explicit - do it solo (like Han), don't use the internet, cppreference site is ok, no stackoverflow, notes from class OK, svn repo OK

In-class exam: "make it reasonably doable" HA. No questions during exam. Closed book. Inexcusable syntax - accidently leave off {} or () for constructors or putting it when unwanted.

Material -

- C++ basic mechanics (struct layout, implicit this, member functions)
    - operator overloading
    - function overloading
- inheritance
    - how does it really work? Layout. This.
    - code reuse? source code vs binary code
    - multiple inheritance
- memory management
    - doing evil memory hacks - operator new or explicitly calling destructor
    - placement new -> new (ptr) T{}
- project 1
    - copy & move semantics
        - l&r value references
    - copy & move assignments & constructors
- project 2
    - oop
        - factory method & scalability
        - objects changing type (morph) -> the polymorphic clone method
        - copy construction (clone)
        - pure virtual functions, hierarchies
        - abstract base class may require a constructor (if it has some state)
        - virtual function tables (implementation of OO in C++), dynamic binding
        - run time type identification is based on the existence of a virtual function table

        - recall virtual destructors from C++ (the memory leak issue)
        - multiple inheritance and the awkward things it causes (the fudging of this pointer and all those discussion)
    - SmartPointer and reference counting/control blocks
        - vs wrappers - looks like objects, don't need -> or *
    - standard functions
    - event driven simulation (not in midterm)
    - lambdas (basics)

static polymorphism (final exam)
    trade offs between static & dynamic polymorphism
    Can't do event driven sim w/ static polymorphism

Question:
    void doit(Foo f, int x) {...}
    doit(Foo{}, x); // would it invoke the Foo move constructor? (cause Foo is a rvalue ref)

Abstract classes should still follow rule of ⅗
Puzzle - Craig make offspring - uses no argument constructor of Craig. What if it ran its own copy constructor? Free other LifeForm? How would you prevent this? (Make the copy constructor private in LifeForm?)

public private protected - in bounds but boring. He'll try to make things public

Pitfalls of array of polymorphic types (because the derived classes may have different sizes)
- array of consecutive objects in memory - circle then triangle then rectangle
- delete[] won't work (because it will invoke the base destructor, not the right destructor)
- truncations
- should just use an array of pointers to object… jesus, so true

dynamic, c style, static cast

dynamic cast checks dynamically if the cast is valid. If not it will return nullptr (for pointer assignments). if it's a non pointer, it throws exception.

this is parameter 0 and *this* would be pushed on the stack first.
Upon end of method, *this* is destroyed (because it's a parameter) but what about the object *this* points to? Killed when returning & stack is popped.

//Pretend we're in a method
Foo f;
f.~Foo(); //added by compiler at end, releases the resources *this* owns. Looks if *this* == &f

this is of type: *Foo const **

~Foo(void){
	this->p = 0; //worthless, seasoning the food then throwing it in the garbage
	this = 0; //Is this even legal? why bother? *this* is just a parameter
	delete this; // So dumb, horrible. Unterminated recursion b/c runs destructor
}

Templated virtual member functions - god, no please. Out of bounds. Also trying to avoid member templates in general

	member templates can't be virtual b/c of virtual function tables - has a fixed number of entries but templates can be instantiated in an infinite number of ways and can't be realistically be made in a table.

function objects

- use lambdas & std::function b/c they're awesome (overall) but not interesting
- **Look at case study from class**
- the convention - function objects are classes & must have operator() method (operator function call)
- the way of building the inheritance hierarchy.
- Make func obj by the composition of two functions: like: f(g())
    - build class w/ data members f & g
    - and a member function in a class that executes both of them (f & g) in sequence
- Remember pointer to functions vs member functions

"Capture expressions are a pain in the ass" - Chase

Project 2 - why build our own smart pointer? Invariant: LifeForm objects must exist as long as there is the object in the quadtree and/or there's a pending event. The event itself must use a smart pointer. Promote from this to self (a smart pointer), need a smart pointer that can look at a dumb pointer that already has a reference count (LifeForm inherits from ControlBlock) & properly promote it to a smart pointer. Thanks Jo! :)

virtual & overwriting return types - don't do it for like int or double. What about returning T* - base returns T1* & derived returns T2*. Want that chocolate cake. Subclass needs to give same or subtype for the return type of overwritten function with pointers. With returning by value, should be the same type! Gotta be the right size & (is-a rule) the right type. Maybe similar rules for parameter passing

Know those first principles… what?