1/21

POD example today
Set-up SVN:
https://lennon.ece.utexas.edu/ee380l.5/sp2016
https://lennon.ece.utexas.edu/ee380l.5_admin

<user, pass> -> <eid, eplsp2016>

Talking about struct elements in memory, yay! (in order)
POD (Plain Old Data) can include structs with primitive data types
Can't create a temporary 'aggregate' type (but my gosh, that seems ok to me)
Initialize ClassName var_name{attribute1, attribute2}; //Weird, no new

- like: Complex val1{1.0, 0.0} // **aggregate initialization**
- this type of initialization is discouraged, because if he construct designer changes the order of variables, we are screwed!

Class vs Struct: no meaningful difference - only default access modifiers (struct defaults to public)

```
Complex foo(void) {
      return {real_val, imag_val}; // strange syntax d=
}
```

Constructors:

- **Proper initialization of a class**
- **Hiding implementation details**
- Can make "it" (class or struct) not POD when written explicitly
- Can be set to default by ClassName(void) = default, which is equal to CLassName(void) {}

Destructors:

- Chase loves them! Hates Java for not having it
- Deallocate memory when obj goes out of scope

Last object constructed = first deconstructed, LIFO:

Complex val1{...}
Complex val2{...}
-> val2 destructor will be called prior to val1's

Copy Constructors:

- Compiler may invoke them implicitly (when compiler needs to make a copy form of the existing obj)
- One argument, duh, of the same type, also duh, but it's got to be by reference, so use *&* (**otherwise we'll be stuck in a conceptual loop when we need to call *void bar(Complex param)***)
- **Use *const* because don't want to change the original object being cloned (by convention)**. Also, use *const* liberally
- Can also be set to *default* which works ok for POD

```
// argument is passed by value
// compiler calls copy constructor without us explicitly saying…
// C++ recognizes not making a deep copy may cause problems for non-pod types
void bar(Complex param) {

}
```

Assignment Operator:

- method signature: void operator= (ClassName const & rhs)

- when you are reassigning a variable to another obj, you need to clean up the current one! (otherwise, it causes memory leak)
- Hack: call destructor on *this* & reassign *this* to the param by calling copy constructor
- Better: basically the same as copy constructor but CC is initialization (happens once), AO is just assignment (can happen multiple times after initialization)
- Default = OK for POD

```
void operator=(Complex const & rhs) {
     this->~Complex(); // Calling destructor explicitly is horrible, wouldn't work well if this IS rhs
     new (this) Complex { rhs }; // We need to rebuild this obj
}
```

in C, group assignment is allowed:

```
     val1 = val2 = val1; // being assigned right to left, as well as *
```

because of that in assignment operator method, we cannot simply return void- > val1 = (val2.operator=(val1));
So we do this instead:

```
Complex& operator=(Complex const & rhs) {
     this->~Complex(); // Calling destructor explicitly is horrible
     new (this) Complex { rhs }; // We need to rebuild this obj
     return *this;
}
```

Complex& -> returning reference for the sake of performance (not to creating a dummy intermediate obj)

The above assignment operator (with destructor and copy constructor) was bad, because if you assign obj x to itself, you are gonna first throw away x and then doing the copy! So, we'll add this if statement below:

```
Complex& operator=(Complex const & rhs) {
     if (this != &rhs) {
          this->real = rhs.real;
```

```
            this->imag = rhs.imag;
      }
      return *this;
}
```

#include<cstdint>: has guaranteed 32/64 bit integers

new keyword - nullptr (like null but for pussies), points to address 0

little endian -> intel machines: the most significant byte comes last

1/26

Strings! A char array that's null terminated b/c backwards compatibility with C
SimpleString example today

In C the .h file for interface-idea didn't work out so great b/c it became too implementation heavy
C++ - templates (yay, what's that?), don't try to divide things between .h and template; just dump it all into one

Put *const* everywhere, for example the following method header:
uint32_t size(void) const {return length;}
const above makes the method itself a constant meaning that the *this* (implicit parameter) is constant. For the above, the implicit parameter is *String const* this* in the String class.

Pointers fudged by compiler. Compiler puts a size value just before where the pointer points to

MISTAKES:

- delete arr -> only calls destructor on the 1st elem of the array and then deallocates the mem (leak)
- delete[] single_elem -> calls destructor garbadge number of times! (the variable that compiler keeps to keep track of the number of elems in the array)

Delete

- calls destructor & frees memory on heap.
- for arrays with things that have destructors, need to use *delete[] arr*, non-array uses *delete something*. If you delete an array of POD, [] aren't needed.
- [] tells the compiler to look past the fudged size value or else it crashes. Using [] unnecessarily will cause the compiler to look for the size value and it will find a random value
- Treating singletons like array of size 1 = solution to programmer having to know whether to put [] or not (just don't). But not memory efficient, really only 4 or 8 bytes though

Destructor

- **silly to set a pointer to *nullptr* after it's deleted b/c it's going away anyways! As long as all memory is freed, field values are of no consequence so don't worry about setting them**
- **Don't *delete this*; it's stupid. You can't be sure that *this* points to the heap. Also recursively calls the destructor**

Compiler is not involved with the operation of the heap as evidenced by my ability to write my own new

There is a difference between a multidimensional array & arrays or arrays. Multidimensional arrays are rectangular, arrays of arrays are ragged

l-value reference - lhs reference, like *char&*.

Operators!

- Shallow syntactic manipulation/syntactic sugar. I think it's cool
- can invoke method with *obj.operator[](param)* or *obj[param]*.
- operator overload - gotta be member functions
- Every operator except . & ? can be overridden
- **Writes += then +**, uses += in +. + Params are left and right. Just use left+=right. BOOM. For a member function, the left is actually *this* and doesn't need to be listed
- Put + in .h file, need to put *inline* before method header

Function overloading, yay

- Same name, different params. Return type is not a distinguisher
- ***const* on the function makes the implicit parameter different so the explicit parameters can be the same**

Const

- Compiler wants to use the non-const version of a method, the more general version
- Passing by constant reference (*ClassName const& param*) is fastest. Used a bunch, also yay

Shallow copies

- Two pointers pointing to the same location = a shallow copy was made

- method header: *void doit(ClassName param)* = a shallow copy is made. When doit is done, the destructor for param is invoked and now the original pointer points to GARBAGE

1/28

Project 1 is out, yay
He writes copy & destroy methods for all non-POD classes. These sound just like copy constructor and destructor. Makes them private. **Also, copy() should never be called on an obj that has been initialized.** the assignment operator calls both (a) destructor, and (b) copy const using the copy & destroy methods

String literal is the type char const*

String s2 = s1 + “!”

compiler promotes the string literal “!” to s1 obj (by calling the required constructor, and creating a temp obj to do the + operator, and then it will destruct the temp obj

Move semantics, rvalue reference

- exists for temporary variables but their values are needed but they go out of scope and are deallocated after they are copied into another variable. Thinking returning a value from a function
- **Uses a shallow copy instead of deep copy**, save memory, oh joy. But this means you have to be concerned with the deallocation/destruction of the temporary variable
- Need a constructor to detect the temporaries, parameter with the stupid &&. Method header: *ClassName(ClassName&& temp)*. Basically do a copy of the temp variable then set any pointers of temp to nullptr (so it won't deallocate/destruct)
- Need a shallow *operator=(ClassName&& rhs)* as well, looks the same as constructor with additionally destructing *this* at the beginning. OR do a swap so that *this*'s data is automatically destructed b/c it's reassigned to the variable that's going to automatically destructed (rhs).
- Can't use a &&tmp from one method to call another explicitly. Because then it's not temporary anymore… Have to cast it with (ClassName&&). OR use std::move(tmp) to state tmp is still a temporary

    - String(String&& tmp) { foo((String&&)tmp); } // Need the casting, without = compiler error with Visual Studios
    - String(String&& tmp) { foo(std::move(tmp)); }

Pointer is null? just do if(pointer), boom! And deleting a null pointer is harmless

copy = lvalue reference, deep copy, &
move = rvalue reference, shallow copy, &&

Templates - C++ generics, replace C macros

- Text substitution mechanic, not totally just generics. Can use this to simulate generics though

2/2

Foo example today
Phase A - due Friday, maybe? Probably
balance checking on operator[] (duh, out of range)

Making stuff

- Foo x; // constructor & destructor called
- Foo* x; // no constructor or destructor called
- Foo* x = (Foo*) malloc(sizeof(Foo)); // no c or d called but space allocated
- Foo* x = new Foo; // constructor but no destructor
- **Foo* x = (Foo*) ::operator new(n * sizeof(Foo));** // no c or d but allocates space for n number of Foos, NEED FOR PROJECT (don't bother to use the new[] version, it makes no sense!). Later, free(x); it changes the annotation for the busy block of heap and update the bookkeeping information
- **new (x) Foo{constructor_params};** // Initialize but don't allocate space, wtf. Horrible hack. Runs constructor. NEED FOR PROJECT (2nd step), yay! Also use instead of assignment operator within push_back & push_front as well as resizing
- **Foo{}; // Initialization without variable name, use to create a temporary/rvalue** (use in expression, argument, return value)

Destroying stuff

- free(x); // used after malloc so free that space
- **::operator delete(x);** //Just deallocates without destructing, mirror to ::operator new
- x->~Foo(); // Hack to destruct a single object, hacked NEED FOR PROJECT
- **x[i].~Foo();** // Hack to destruct object with [] syntax, i is the index, use in loop

malloc - **selects** memory within pre-allocated (by the OS) block.
basically annotates heap to indicate that block of memory is taken.

operator new is different from just calling *new operator function* (same with *delete*). Create loophole so people don't use malloc to create space but not initialize (functionally it is equal to call malloc)
Standard Template Library (STL): not recommended to use for mem alloc, dealloc, but you still can use if you want.
http://en.cppreference.com/w/cpp/memory/allocator

Make a standard allocator just as syntactic sugar so that it's less ugly to allocate and deallocate. std allocator. All methods are const because the allocator object itself doesn't have a state. Can use multiple allocators to allocate, construct, destroy, and deallocate a single object

```
template<typename T>
struct allocator {
	T* allocate(uint64_t num) const {
		return (T*) ::operator new(num * sizeof(T));
	}

	void deallocate(T* p) const {
		::operator delete(p);
	}

	void construct(T* p) const {
		new (p) T{};
	}

	void construct(T* p, T const& lref) const {
		new (p) T{ lref };
	}

	void destroy(T* p) const { p->~T(); }
}

int main(void) {
	allocator<Foo> my_alloc{};
	Foo* p = my_alloc.allocate(10);
	allocator<Foo>{}.construct(p);
```

```
      my_alloc.destroy(p);
      allocator<Foo>{}.deallocate(p);
}
```

Gonna use move initialization in phase B…

Bounds checking/exception stuff

- throw std::out_of_range{"Dis is my message"}; //out_of_range is a class, the message is the constructor argument
- Need the good ol' try-catch blocks
  try { } catch(...) {} // ... is a catch all exceptions
  try { } catch(std::out_of_range x) { cout << x.what() << std::endl; }
- Can easily make your own exceptions
- catch (expection& x) with the & makes copy constructor not happen

```
// defining our own exception
struct my_except {
      // constructor
      // what what in the butt
      // copy constructor (for fun)
}
```

Smart pointers
Chase - "I love garbage collectors"

you will lose type safety if the programmer is allowed to manage memory; that's one of the reasons Java language does not allow that.
delete operator allows you to violate the type safety

Dangling pointers, blah blah blah

operator* is a const method for a pointer because it doesn't change the value of the pointer itself but could change the value it points to.

C++ - all inner classes are static

Smart pointer

- Has a control block that keeps track of how many references there are to a specific object. When the reference count = 0, then deallocate
- Control block has to be a separate chunk of memory that all smart pointers also point to so that they all keep the same reference count
- Has the actual content/pointer as a field
- Supposed to make garbage collection nicer and makes an application programmer not have to explicitly call new or delete or deal with pointers directly (my dream!)
- Do move because don't want to needlessly increment & decrement for rvalue smart pointers
- Can combine the controlblock & object of interest by using std::pair, glue them together. But makes it depend on a default constructor being there for the T (object of interest) OR can use arguments passed to the SmartPointer but how do you know how many parameters do they have? Oh god. So use ... , good ol' ...
- There is a standard library smart pointer - std::shared_ptr (multiple pointers can point to it) and std::unique_ptr (only one

pointer should point to it). Just no cycles. Use std::make_shared or std::make_unique

2/9

project 2

- OOP, abstract classes, etc.
- smart pointers
- event driven simulation
- functional programming

Classes - powerpoint that he says he updated, nice

- OOP stuff I know. (you are knowledgeable :D)
- Inheritance
  - public inheritance is what you expect/want from inheritance
  - private inheritance is stupid. If you use private inheritance, all the public methods of the base class become private to you.
  - Can inherit constructors but it's ugly (two ways shown below)
  - cascading the assignment operator, while you have inherited the base class, changes the type...
  - base_cnst, derived_cnst, derived_des, base_des (the order of const/dest invocation)
  - derived class is appended to base class
  - Multiple inheritance is a thing

```
vector<int> x(10); // Not curly braces {} b/c then it's a list of size 1
x.push_back(42);
x.pop_back();
cout << x[10] << endl; // does not do bound checking (still prints 42)

now we extend it (inherit it) and do the bound checking ourselves!

class myvec : public vector<int> {
public:
      using vector<int>::vector; // Other constructor option
      myvec(int sz) : vector<int>(sz) {} // : vector<int>(sz) is super()
```

```
	int& operator[](int k) {
		if (k < 0 || k >= size()) {
			cout << “WTF yo?\n”;
		}

		//fully qualified method name = call super
		return this->vector<int>::operator[](k);
	}


}
```

another alternative for calling the super method using type casting:

```
	vector<int>& base = *this;
	return base[k];
```

Chase has officially given up, and I quote "I give up. I don't know"

```
class Base1 {
public:
	int x;
	int y;
}

class Derived : public Base1 {
public:
	int z;
}

// With just Base1 b, Derived can be used but will be truncated
// for the sake of efficiency, no truncation
void doit(const Base1& b) {}

int main(void) {
	int x_off = offsetof(Base1, x); // it’s a directive, not func
	int y_off = offsetof(Base1, y);
```

```
    cout << “in Base1 x has offset: ” << x_off; // 0
    cout << “ and y has offset: ” << y_off; // 4
    cout << endl;

    int z_off = offsetof(Derived, z); // 8 (z will be directly appended)


    Base1 b;
    Derived d;
    d = b; // compile error
    b = d; // works, the derived obj will be truncated

}
```

now multiple inheritance:

- Appending order is determined by the order in which they are listed. Ex: Derived : public Base1, public Base2 - Base1 is first then Base2
- But what if both base classes have the same function? just do derived.Base1::doit() or derived.Base2::doit(). Can't just do derived.doit(). Need to be fully qualified
- In Base1::doit() *this* points to the beginning but Base2::doit() *this* points to the beginning of the Base2 in the derived
- If the function also has a doit() function, don't need to be fully qualified, just derived.doit(). The *this* in that doit() function points to beginning of Base1

```
class Base2 {
public:
    int z;
}

// offset of z in Base2: 0
// offset of z in Derived: 8 (the current Derived class does not have attribute z, Chase removed it for this example)
```

Note: the Base2 obj knows the address of z to be 0, because of that, the *this* address printed in *doit()* of Base2 function points to middle of the obj, instead of the beginning. (so that z is still at idx 0)

Note2: if we have a *doit()* inside the Derived class as well, the *this* address would be the beginning, since Derived has all the variables extended inside itself and the compiler know the whole derived obj.

virtual provides dynamic (late) binding (the binding would not happen during compile time). During run time, it picks which one to use (the virtual method in the base or derived class). Need to learn more about virtual

the diamond fucking pattern!

C++ has a virtual inheritance mechanism to deal with such a case (only inheriting state once, the Counter example on the slides)

```
class B1 : public virtual Counter {};
class B2 : public virtual Counter {};

class D : public B1, B2 {}; // only once the inheritance of Counter happens (?) (I think yes because of the virtual)
```

2/11

Jo: Chase joked around with me, trying to block my way. Then I coughed a lot

Just because you say it's an rvalue (Foo&& var) doesn't mean it'll be used as such in functions. Gotta cast it when calling the function (doit((Foo&&) var) or doit(std::move(var)). But a const&& would rather use copy constructor (Foo const & var) than the move constructor (Foo && var). std::move doesn't strip the const, so compiler picks copy constructor because the compiler doesn't want to strip the const either

typeid(var), std::is_const<Foo>::value, or std::is_reference<decltype(var)>::value gives metadata, like Reflection in Java but not as powerful

Pointer to member function in case of multiple inheritance (?) - what are the difficulties and how are they overcome?

public vs protected vs private - some little technical stuff that makes it work just not quite as expected

OOP

- a style, can be done in any language, even assembly
- work with families of related types
- delegate type specific decisions to the objects themselves
- liberating

Virtual

- Used to tell C++ that OOP behavior is going to happen. Signals for the use of dynamic binding/there's more than one potential choice so it picks at runtime
- Put at the base class level on methods to signal subclasses could override the method
- Virtual has no impact when added or removed on a method in a derived class. Put the override keyword in the signature of the

derived method (for virtual) to signal it's overriding the base method. Override is optional though, like @override in Java

- Methods that invoke these OOP/virtual functions need the parameter to be a reference. If it's just a value, it truncates the object when copying to just the base class.
- Once you go virtual, you don't go back. Can't take away the virtual-ness of function
- virtual function return types must match up between base & derived classes - the same type or a subtype of that type (only subtype when by reference). Can't do subtype by value because type strength/checking

Chase "Java is the best way to do object-oriented programming" BAM!

In a derived class, you cannot override the return type:

```
Base:
     virtual int doit(void) {return 42;}

Derived:
     virtual double doit(void) {return 4.2;} // compile error
```

Virtual function table pointer, when you use virtual keyword (even one) in a Base class. Also, it would be copied to all the subclasses

He promised he's going to draw pictures next time.

virtual table is shared among all instances of a class (kinda static)

virtual ptr variable will be initialized in the constructor, so there is no empty constructor. void** (type of __vfptr)

size of the obj does not change depending on how many virtual members we have. It’s always one virtual ptr b/c it just points to a table (the table will be extended though)

__vfptr	all elems assigned in constructor	void**
0x00000000	offset to 1st virtual function	void*
0x00000001	offset to 2nd virtual function	void*

Non-virtual functions have static binding so no need for it to point to a table.

If Derived and Base class objs have vptr, the order of function pointers respect the base order.

Derived and Base class obj have different ptr addresses if they point to different functions. If virtual functions aren't overridden in Derived, it'll point to the same place Base does for that function. Compiler would make two separate but identical tables if Derived overrides NOTHING from Base. (Because of runtime type identification, RTTI)

Base b;
Derived d;
Base bobj = d; // is-a? it’s a one way. from higher levels to lower. this line truncates the derived obj. cause it may not fit the the

space of Base. on this line, the constructor of base class will be invoked and set the vtptr to the Base class (it does not care what is the rhs)

Base& bobj = d; // Is fine, upcast StackOverflow
Derived& dobj = bobj; // Not fine, b/c is-a. Can't downcast
Derived& dobj = (Derived&) bobj; // OK but DON'T do this C-typecast
Derived& dobj = static_cast<Derived&> bobj; // C++ cast (similar to C) it does not do any runtime checking. (it's not like Java)
Derived& dobj = dynamic_cast<Derived&>(bobj); // does runtime checking, throws exception if it cannot be converted. If the type is ptr, it will just returns nullptr. (that's the answer to my question. they runtime check has to look at the vptr, that's why we duplicate tables even if they are identical…)

doing dynamic_cast without vptr is impossible. compiler cannot generate static codes that check this during runtime. Need a virtual table to look up for any dynamic casting/identifying

If a derived class inherits two base classes (and there are virtual members present), it will have virtual pointer**s** (one for each base)! Concatenate the two tables together, just have to remember offsets in the virtual pointer itself, not in the offsets to the functions

If derived has it's own virtual functions not found in its base classes, they are inserted into the virtual table anywhere (consecutively I assume)

Base& b1 = d; // there is no constructor call or shit, we are just defining a new reference to obj d. Will run the Derived.doit(), not Base.doit()

THUNK - trampoline function, exists to change something to invoke the real function. Ex: Adjust *this* to call the real function when there's multiple inheritance & we're looking at not the first inherited class

2/18

// we use pointers when working with polymorphic stuff (we don't want derived objs to be truncated)

```
int main(void) {
	Base b;
	Derived d;
	Base* p = &d; // so far OK

	// trouble begins (like Batman Begins, uh lol)
	Base* p = new Derived;

	delete p; // destructor + deallocation, runs Base class destructor not Derived's
}
```

that's why we should define destructors always as virtual! So that we have runtime dynamic destructor invocation. This way, casting won't cause problems. And helps with no resource leaks. Will destruct with derived destructor then base destructor. Will construct base then derived.

Sol:

```
	class Base {
		public:
			virtual ~Base(void) { std::cout << "in base destructor\n"; }
	}

	class Derived : public Base {
		public:
			virtual ~Derived(void) { std::cout << "in derived destructor\n"; }
	}
```

now both destructors will be executed for p in the above example (Base* p = new Derived; delete p;)

in constructors, the runtime type of objs are evolving (base->derived) and in the hierarchy of destructor calls, the type of obj will level by level degraded. (derived ->base)

very interesting!! even if there is no resource in your base class, declare the destructor as virtual to get the polymorphic behavior.

Rule of thumb: If a class has any virtual functions at all, it should have a virtual destructor!

virtual void draw(void) = 0; // ***pure*** virtual (makes the class abstract), definition of this method is null AKA = 0

you cannot instantiate abstract classes. There's no keyword for a class that's abstract to identify it as abstract. Can only call base constructor from a derived class if base is abstract.

Abstract, why?

- Get an interface
- Enforce all the subclasses to override (define a behavior) for the pure virtual method by not providing a default (like the sort algo class, with the virtual bool lessThan() = 0, method discussed in class). Good for no reasonable default

Shifting the *this* (fudging) in Java is not a problem, b/c we don't have multiple inheritance. Interfaces have no state so don't move the pointer

```
// C style sort (C trivia: int fits in void*, some people use it, not a good practice though!)
// pfun is a pointer to a function with parameters (void*, void*) & returns an int, our comparison function
void qsort(void* data[], int num_elems, int size_elem, int (*pfun)(void*, void*)) {
        int result = (*pfun) (nullptr, nullptr); // just the syntax, those are the parameters
}

int compare(void* x, void* y) {

}

void myApp(void) {
    qsort(&data, 500, 8, &compare);
}
```

C++ prefers lessThan & equals instead of Java's compareTo

```
class FunObject {
    public:
        //operator function call, yay. Makes it look like a normal function when used
        int operator()(double x, double y) {
            if (x < y) { return -1; }
            if (y < x)  { return 1; }
            return 0;
        }
```

```
}
```

you can make qsort to be a template to provide some flexibility, but it is not the optimal solution! we need inheritance to do it right!

You can make the parameter JUST the type (no f) and then call the function with SomeFun{}(param1, param2) AKA in the qsort argument, just as SomeFun (no var name)

```
template <typename SomeFun>
void qsort(double x[], int n, SomeFun f) {
        if (f(x[0], x[1]) < 0) { //f will be inline, yay (why in C it's not inlined? The built in sort function isn't inline b/c it calls a function. I think that's what he was saying wasn't)
                cout << "whoopie!\n";
        }
}

void myApp(void) {
        FunObject fun;
        double x[1000];
        int n = 1000;

        qsort(x, n, fun);
}
```

Event driven simulation: use a priority queue to add/remove events in the proper order. These events could be static or not, member functions or not, etc

while (queue not empty) { pop event e; e(); } //e has operator() method

Call a non-member function

```
void (*pfun)(void) // UGLY pointer to function syntax(void return & void param), replace with:
using PFun = void(*) (void); // Now just use it with:
PFun pfun;

void operator()(void) { (*pfun)(); }
```

And for a member function:

```
using PMemFun = void (ClassName::*) (void); // Points to ClassName member function
PMemFun pmfun;

void operator()(void) { (*pmfun)(); } // NOPE! No implicit this/receiver object
```

Need an attribute that's the ClassName type

```
ClassName& obj;
```

void operator()(void) { (obj.*pmfun)(); } // looks like a regex!! this is for member functions (we need *this*)

****

- *POD* stands for *Plain Old Data* - that is, a class (whether defined with the keyword `struct` or the keyword `class`) without constructors, destructors and virtual members functions.