3/22

Not allowed to - can't be psychic. Can't use static, global variables, etc that allows the LifeForms to share with each other. Shouldn't know your position. No low level direct manipulation, like changing virtual table pointer (dang)

stl - standard template library

For a templated method, when calling it you can specify the type but don't have to
template <typename T>
void swap(T whatever) {}
Call it with swap<int>(whatever); or just swap(whatever);

Compiler can't mix type conversion with template argument deduction (invoking the max function with (double, int) gives compiler error! Because we only have "template<typename T>")

Goals for STL
1. Generic with respect to element type
2. Generic with respect to data structure type (container type/native arrays vs ADT collection lists) -> harder to achieve in C++ (comparing to Java)
3. Predictable time complexity (who cares, repOK is a predicate!) -> running a binary search on a LinkedList will not compile in C++ (in Java it does, just it's inefficient)
4. Comparable/competitive performance with hand-written custom code (Java's generic library does not satisfy this one) -> here in C++ we expect inlining of swap (in the class example)

For generic stuff dealing with containers, don't pass the container. You pass pointers to the beginning and end of the data structure instead. Then to get the size, simply int size = end - begin

Iterators/cursors - modeled after pointers

Fundamental operators on pointers/used when they are treated like iterators

- Dereference (operator*)
- Check for equality (operator==) - to see if I'm done/it's empty for iterators
- Advance (operator++)

Above = forward

- Reverse (operator--)

Above = bidirectional

- Advance by integer amount (operator+)
- Calculate distance (operator-)

Above = random access

For other operators like, != or <, use a combination of the above
P != q is really !(p == q), p < q is p - q < 0, p[k] is *(p+k)

Generic algorithms are ugly as fuck (like Nima's babies d=)

End iterator points one after the last element. So when passing iterators, you do [start, end) to actually go through it. AKA end is one after the last element. Use this to check if it's empty because if you point to the last element, how do you point to nothing? And convention

Data structures should have the functions begin() & end() to make the iterator easy. Iterators are going to be a nested type, fully fledged and ready to leave the nest. The conventional name of the iterator type for any data structure is ClassName::iterator. Ex: list<int>::iterator. But just use auto if you want to capture the iterator. auto e = my_list.end(); (C++ is statically typed, so compiler knows what the type of my_list.end() is. Still for the purpose of this course we should know that iterator is a nested type in a data structure) You need the three first pointer/iterator operators but hopefully all 6.

Templates not syntax checked until the code is actually going to be used. Compiler needs to see the actual invocation for templates, and then the provided context helps the compiler to check if the code is semantically correct or not. Like if the type used have the proper operators defined like ++ or *.

Linked list provided by C++ doesn't have the last 3 operators (--, +, and -) for iterators so it won't compile if you try. (like int n = e - b; in the printContainer meta function, when invoked on a LinkedList (which cannot tell you the result in constant time (more parentheses d=)))

If your algorithm expects bidirectional iterators, it won’t compile on a singly linkedlist

C++ for each loop:

```
//for (auto p = begin(y); p != end(y); ++p)
// different begin() functions (based on if the container used is an array or a higher level
data structure, like LinkedList)
for (int val : y) {
        cout << val;
}

//even better
for (const auto& val : y) {
        cout << val;
}
```

Need to acknowledge that < operator might not be defined for all types. So now Chase is writing a comparison, templated class that overrides operator(). Then we end up with:
sort<std::list<int>::iterator, GenComp<int>>(y.begin(), y.end()); // SO UGLY.
So you just make the third argument in sort be the comparator
sort(y.begin(), y.end(), GenComp<int>{}); // Need {}, you're not passing a type
Also, you can keep the two argument sort & just call the three argument with a generic comparison class that uses < operator

3/24

Quicksort - making errors on purpose

comp() -> anti-reflexive, i.e., x cannot be smaller than itself (convention)

lo < hi cannot be determined in O(1) in a LinkedList (even in the most general version :P). Less than operator can only be used with random access iterators.

He wrote a partition for quicksort to use only forward iterators. Just gotta look at the code. Same time complexity as random access partition but less efficient (the bidirectional version is twice is fast, because it has half swaps). This means he has to write both versions of Partition so he can choose what's best for the situation. (He can have one quick sort, but he needs 2 partitions, forward and bidirectional)

Can we efficiently dispatch to the right partition function? (for singly vs. doubly linked list?) Use the iterators given by the container to tell if it's "fast" (bidirectional/random access) or "slow"(forward).
Attempts that suck:

- Iterator has function is_fast() to tell if it's fast
- use iterator_category w/ a switch-case or if (how can we get rid of it completely, so that we can compete with the custom hand-written code, req 4 of STL?)

Goal: want function overload resolution so that the compiler makes the selection (static instead of dynamic)

- Both functions have the same name but different parameter lists (duh, definition of overloading)
- Add unused argument that's an std::whatever_iterator_tag solely for the purpose of overloading/signalling. The "tags" are really just structs that inherit appropriately (bidirectional inherits forward, random access inherits bidirectional). Three possible tags - forward, bidirectional, and random access
- You get the type of iterator tag with ClassName::iterator_category tag_var{}; Omg, you have to put one byte on the stack. This gets rid of a conditional though, so overall win
- But native arrays have trouble with this because they don't have the ClassName needed to get the iterator_category, their iterators are just pointers. Using a level of indirection! Of course. Making structs so that one handles real class iterators and the other handles primitive pointers as iterators. Primitive pointers as iterators = random_access

"Thinking and typing at the same time is a bad idea" - Chase (Not said today but before)

typename - relatively new keyword. Some people used class instead as an overloaded keyword to make templates.

Default assumes :: is used to access a value. If you put typename in front of it, it makes it look like a typedef AKA you're getting a type with :: (and typedef analogous of C++ is called *using*) -> using interator_category = typename Iterator::iterator_category;

It’s better to be explicit when we are using a dependent nested type (using the *typename* keyword) (explicit is good)

Template specialization & meta-programming to come. Yay

T does not work. T is the type of the iterator not what it contains!
We solve it today with not dicktype, instead (d=):
using Elem = decltype(*b);
quickSort(b, e, less<Elem>{}); // Where hopefully Elem is int or string or whatever the container holds

Convention to the rescue again. Similarly to iterator_category with iterator_traits, use value_category to get the type that a container holds

3/29

"My favorite example, 'Jello World'" - Chase

Use auto to get iterator types. auto it = p.begin();

Iterators - what if you don't want to allow changing the underlying structure from the iterator. Adding *const* to an iterator causes the iterator to be unmodifiable so no assignment operator (yay), but you can't even increment the iterator. Adding const to pointers (char const *) gives the behaviors we desire for iterators.

Defining an iterator as const does not give us the behavior of defining a primitive type as const (like char const *, or const char *). Because it defines the iterator itself to be a const! So ++p gives compile error. Similar to char * const const_p2 = &s[0] (which the pointer itself is const not the values it points to).

To get around the *const* troubles, define a const_iterator, which points to immutable data! You might even make the "non-const" iterator actually be const if you always want the data to be immutable. So you overload begin() and end() to return the const_iterator when the function is called in a const context (the implicit receiver obj is const, and that’s how we are overloading the method, o.w. we know overloading is not possible on method return values)

iterator begin(void) {} vs const_iterator begin(void) const {}

Making an iterator:

- Write operator* first, yay.
- Then operator++ & --. He has operator++(int), why?
- begin() and end() (end = doing ++ one too many times). Non-const
- operator== (make it a const method, checking equality shouldn't change values). And operator != which just uses ==.
- can write a general templated != instead of operator overloading in the iterator class! (if there are both, which one will compiler choose? Of course the one that's non-template, chooses the most specific. ????? WTF! Maybe? std::rel_ops doesn't really work that great I guess. std::rel_ops sucks!

Common to make the iterator class a *friend* class of the main class. For example, putting *friend String;* in the iterator for the String class. But any use of the *friend* keyword is a hack. It's a good hack in this case but Chase wouldn't do this for real. Make String and iterator_type friend so that String can access the members of the iterator class. The iterator already has access to the private components of String because it's inside the String class.

For his string iterator, internally his begin is actually one before the first character & the end is the last character. Oh Chase! But it should still behave like a regular iterator, hence the p[-1] for the operator*

Challenges:

- allow iterators to be demoted to const_iterators.
- Const_iterators won't come from mutable Strings without typecasting the string to be const

He expects us to honor the fact that the iterator type should export (as public members) the value_type and iterator_category. Alternatively, we can make our iterator to extend -> std::iterator<std::random_access_iterator_tag, Character> (example), which is a safer choice to do!

class iterator : public std::iterator<std::iterator_tag, value_type>

Vector iterators can be tricky. If the iterator is a pointer, doing a resize makes the iterator/pointer point to the wrong address. Makes the iterator invalid. Generally speaking, If you have an iterator to a place in the data structure and the DS is modified (not the values but change the positioning), the iterator may be no longer valid. (for example, pop_back can cause it) -> we should invalidate them? Just changing the value of an element should NOT create an invalid iterator. (in specific examples, we can have more relaxed rules…)

"Java doesn't like undefined behavior, good for Java" - Chase being awesome

In 1c, we're going to determine if our iterator might have become invalid & to generate a specific exception, like Java (ConcurrentModificationException). Different levels of exception based on severity.

How do you detect that the iterator is invalid though? The iterator needs to have the data structure that they pointed to saved (Chase calls it home). Keep a version number in the data structure, give that version number to the iterator. Whenever the iterator is used, check the data structure's current version number. If it's changed, iterator could be invalid. Gotta make sure to increment/change the version number for every method that changes the data structure in a way that can make an iterator invalid.

The type of an array is stripped when passed as a parameter, it's just a pointer.

All the arguments to a template must be a constant, can't use a variable.

Decltype is awesome

Created a class that has a member function that is templated.(the class itself is not a template) That way you don't have to specify the type of the template when making the object! It’s so cool! You're so cool. Only if no type conversion is implied

3/31

The convention to pass a data structure to a function is to pass a pair of iterators, the begin & end iterators (ON EXAM) like the following code

```
template <typename It>
Vector(It begin, It end) : Vector()  { //Delegating constructor after the :
        While (begin != end) {
                push_back(*begin);
                ++begin
        }
}
```

When writing code in a template class, it is sometime necessary to explicitly provide the *this* parameter. (for the compiler to figure out what the heck is push_back for example)

We can check if the type is valid, using an static_assert (we’re going to see it later). Project 1c doesn't require intelligent error messages though

Characters can be promoted to integers without errors/warnings in C++.

C++ is colon happy! :::::::::::) So C++ is a spider

Overloading the constructor (for initialization) and maybe assignment operator (for re-assignment) to promote a primitive array to a vector? Below is how it's actually done:
Ex: Vector<int> x = {1,2,3,4};
In Vector, make constuctor w/ header Vector(std::initializer_list<T> list) {}
The initializer_list (i_l) acts as a holder of two iterators, you can get list.begin() & list.end(). Then just delegate it to the iterator constructor we made above, yay

Chase is offended by double constructing
- Some don't have move constructors, POD (consider a complex object, it's not pointer based so unlike strings, complex objects have heavy moves (like 16 bytes))
- Silly to construct a temporary then move & destroy original, why can't we just use all of the orignal

So, emplace_back
- Two versions - one with & one without arguments. Without arguments = default constructed T
- With argument, using overloaded constructor (like creating an string from a char), that's why we use a new template typename Arg (not T). In our example, T is a big obj!
- With multiple arguments, same as overloaded constructors, just calls constructor with multiple arguments.
- OH NO, so many arguments, how do we do it?? Variadic templates (Can have 0+ arguments)

Variadic templates, that beautiful syntactic sugar. All the … :
```
template<typename… Args>
void method(Args... x) // 0 or more like *
{
	method_overloaded_with_various_args( x… );
}
```

Just a reminder, put const everywhere

Now what if you make a method that takes in a templated type that's actually an rvalue reference. Using std::move on them is awful because it will also cast lvalue references to rvalue references, oh no. This is the argument forwarding problem. (so far, on both cases of casting to r-value ref or not casting it we have a problem. How can we fix it? With a doit proxy!).
Suggestions:
- Use **two** functions with the same name (one with & params and one with && params) to find out what kind of values are being used. A "doit proxy", a forward
- Std::forward, like doit but I'm sure better

template<typename… Args>

```
void method(Args&&... x) // 0 or more like *
{
	method_overloaded_with_various_args( std::forward<Args>(x)... );
}
```

Something with void doit(int& &&) //wtf, rvalue ref to an lvalue
In writing general argument list, we use rvalue ref (&&) and let the compiler deduce the type

Now writing is_ref (how we can implement it just like it is in std::is_reference). He created two templated structs, where one is default & the other is used when the templated type is a reference

4/4

// How to determine the type of the returned value?

```
template<typename T1, typename T2>
T? max(T1 const& x, T2 const& y) {
	If (x < y) { return y; }
	else { return x; }
}
```

maybe promoting to the stronger type is a solution (like choosing double between <int, double>)

const vs. constexpr:

const -> a variable that is assigned prior to execution (not necessarily at compile time, but once the program starts up, a value will be assigned to that variable. It can be at compile time, load time, etc.)

constexpr -> chosen by compiler (should be decided at compile time), AKA, compiled time const. Usually with numeric values

4/6

Again looking at max from above. Trying to figure out the type of T1, is T1 a double?

- If (typeid(T1).name() == "double") <- SO GROSS
- Two methods called isDouble - one that accepts double as a parameter and one that is templated and will catch everything else. First method returns true, second returns false.

Then isDouble(x). if using constexpr, you can get the compiler to run a method at compile time with constexpr values/parameters (or something).

- Template specialization - makes structs with templates. Each struct has a static constexpr bool result which is true or false. The right one is picked based on template specialization.
    - You can only run it on the types that are defined (so far for me, just int and double).
    - if (IsDouble<T1>::result). value is convention, but Chase likes result.
    - Chase likes this form because it's completely compile time decision wise. The if statements are run time but the templates are compile time.
    - You can't specialize the template until you've declared the template
    - Example
        - template <typename T> struct IsDouble;
        - template <> struct isDouble<double> {static constexpr bool result = true; };
        - template <> struct isDouble<int> {static constexpr bool result = false; };

Can't forget that all of this is also in the pursuit of finding the proper return type. Should be a deductive analysis performed at compile time based on T1 and T2. Don't forget to sprinkle in typename wherever you're using a type where you'd traditionally think it'd be a value

- *typename* WhatType<T1>::result - first we're just going to look at T1. typename is silly because duh, a return type (what we're trying to pick) is a type
    - Get the type from a value:
        - template<bool> struct PickDouble
        - template <> struct PickDouble<true>
          { typedef double result; // OR using result = double; };
        - template <> struct PickDouble<false>
          { using result = int; };
    - Now put it all together, basically a sequence of two calls. One to get a value based on type then use that value to get a type again.
        - template <typename T> struct WhatType {
          static constexpr value = IsDouble<T>:: result;
          using result = *typename* PickDouble<value>:: result;
          };
    - Can use the following to get rid of typename in front of the return type. This is a template alias used to abbreviate the code/hopefully make it easier to use. Oh syntax sugar.
        - Template <typename T> using whatType = typename WhatType<T>:: result;
        - whatType<T1> max (T1 const& x, T2 const& y) { /*blah*/ }

So he just erased all that. We're going to look beyond just double or not

- Like IsDouble but instead of true false, the value has meaning in itself. Like 1 = an int, 2 = a float, etc. Limited to the number of structs we make and the types we give them
    - template <> struct SRank<int> { static constexpr int value = 1; }

- Then a reverse set of templates
    - Template <> struct SType<1> { using type = int; }
- Now to actually choose the types. FYI he hates ? syntax
    - template <typename T1, typename T2>
      struct choose_type {
      static constexpr int t1_rank = SRank<T1>::value;
      static constexpr int t2_rank = SRank<T2>::value;
      static constexpr int max_rank = max(t1_rank, t2_rank); // He used ? to get compile time stuff, (t1_rank > t2_rank) ? t1_rank : t2_rank
      //But we gotta make sure the max function isn't recursive. Chase can't get it to work though without the ? in the max function with only T instead of T1 & T2. Used constexpr to describe the overloaded/single type max
      using type = typename SType<max_rank>::type;
      }
    - And now some sugar/type alias
    - template<typename T1, typename T2> using ChooseType = typename choose_type<T1,T2>::type;
    - Now the method header for max is: ChooseType<T1,T2> max (blah) {/*blah*/}

Using auto and decltype is great. I think he's saying typeid is better than decltype. When comparing max(5.5, 10), the return type was still double. Had to pick the type before it could actually compare them to find max.

Template meta-programming was discovered by accident. Initially, people would use an enum within the struct and pretty much always used the ? syntax to find max b/c enums are ints and compile time constant. Don't do that though

In the last 5 minutes he is going to try to add Complex, ha. Std::complex doesn't care if each part of the complex is single or double accuracy floating point. Double vs complex float? Pick complex double! You ask two questions, is it int float or double (what is your rank)? Then second question, totally independent, are you complex? FYI, he doesn't like complex int.

template <typename T> struct SRank<complex<T>> {
    static constexpr int value = SRank<T>::value;}
Let's us write this stuff recursively

4/12

I love this class - Chase is the best
Today's code: ExprTemplates

5 minute rant about how poorly vector & valarray are named. Should be array & valvector

Assignment/operators are different because:

1. The semantics are different. The operation on 2 valarrays will work with the minimum of the size (only the 1st 10 elements will be considered between 2 valarrays of 10 and 100 sizes) vs. in Vector, it was not this case.

Good thing to not have to write (*this) all the time, use Same& lhs = *this; Boom

Valarray is actually is wrapper, i.e., an outer interface for vector, by adding new features, changing some features.

We need to inherit all the constructors of vector in valarray basically without changing, with this: using std::vector<T>::vector; There is no state in valarray, so the base class initialization is perfectly fine for valarray. This will give us ALL vector constructors including from iterators, initialization list, size, etc. (Nice!)

The move assignment from vector does not make sense for valarray. Copy and move in valarray are the same

- Do we need to change the semantics of move semantics from vector? An exercise for students: Calling copy constructor in move assignment (currently)

Trivia: the += operator in C++ has to be a member function!

"I'm going to use dorky code to support the dorky requirement" - Chase on operator+=

He usually does += then makes + call +=. But he's done it backwards this time, + first then +=. += is two reads but one write in just the + alone, at least. Then you got to do assignment, which is looping through everything again and there's unnecessary allocation. Chase hates all this stuff. So keep the + and += separate, each one is a custom implementation.

But there's copy paste still! Use a function to take an operator along with the valarrays, apply_op. Call it with apply_op<std::plus<T>>(result, lhs, rhs); There's plus, minus, multiplies, divide (http://en.cppreference.com/w/cpp/utility/functional). Could put std::plus<T>{} as the last argument instead but it's ugly to Chase. Apply_op returns nothing right now but for better code, Chase would add some kind of return type.

The one you expect the compiler to deduce should be at the end of the list of template arguments. (<typename Op, typename T>, T will be deduced). Also default argument in the apply_op function come last. (maybe we don't wanna provide it if it's the default for more convenient programming style.)

Now, if you want a longer expression like w = x + y + z, the most efficient way would be a loop that accomplishes w[k] = x[k] + y[k] + z[k] all at the same time instead of temp[k] = y[k] + z[k] then w[k] = x[k] + temp[k]. Yaaay! Expression templates! (the hangouts bunny goes here)

The goal of this project is to be the most efficient! No compromises! Ever! We're going to lie, return something that acts like a vector but isn't actually populated until you need it/don't do the operation until you get to an evaluation (assignment operator, iterator, or output operator). Proxy!!! Or lazy evaluation. E.g. x+y+z without assigning to any variable should be done in constant time.

The return type will provide some metainformation (encoded in types not in values) so that we can propagate our ability to inline. (That’s what expression templates are all about).

=====================

Implementation of std::plus

std::plus<int> add_op; If used with a double, gives a warning then truncates. add_op(3.5, 4); will really use 3 and 4.

Only when you provide std::op<void> -> you are getting the new version in C++14 that works with different type (by deduction…)

4/14

Today's code: ExprTemplates2
Expression templates! Motivated by:

```
cout << "Hello " + first + " " + last << endl;
```

Three pluses = at least 3 allocation which copies things over and over again (Hello is copied at least 3 times). We want to avoid temporary allocations

Making a smarter string, will use lazy concatenation. Don't even need to put anything in the class since he's just doing operator+/concatenation. Don't concat yet, only do it when necessary (evaluation points - In the actual project eval points are: printing, assignment, maybe iterators or not)

Concept checking - like the generic sort function we wrote (the elements should be comparable!). The strict weak ordering concept...

Here for printing strings/operator<<, we need 2 things: size() and operator[] then we actually don’t care if our argument is a concrete SmartString (it should just have the necessary interface/concept)

He's making a concatString class to hold onto the two strings that are to be concatenated and added the needed methods (size & operator[]). This acts as a **proxy** for the result of concatenating two string

```
Class ConcatString {
        Std::string const& s1;
        Std::string const& s2;
Public:
        ConcatString(std::string const& left std::string const& right) : s1(left), s2(right) {}

        unit64_size(void) const {...}

        Char operator[](unit64_t k) const {...}
}
```

1. The proxy should not keep the actual data. So we need references instead of values!
2. The operator << still takes SmartString, one solution (not good) is inheritance which is defining ConcatString as a child of SmartString (don't forget to make the operator[](k) to be virtual! Other wise nothing will be printed in the << function cause, no dynamic binding will happen)
    a. Why using OO is not good in this context?
        i. Chase hates it
        ii. Efficiency (Inlining does not happen!!!)

No virtual in project 3! None! At all! Use const versions of operator[] and size (the concept). Also consider including value_type (a constant as well)

Now he changed his operator<< to call a print method instead. Now there's no size or operator[] and it now depends on the underlying structures (string) operator<<

But what if want to concat three? Right now it can only handle two.
Good debuggin fun:

```
Cout << one + two + three < endl;
Auto temp1 = one + two; // use typeid(temp1).name() to see what's going on
Auto temp2 = temp1 + three;
```

We need another overloaded version of operator+ with different arguments, which is operator+(ConcatString, SmartString). But we don't have a way to create a concatstring with a normal string. Enter templates! Give ConcatString a LHSType and RHSType, yippee. Those types are expected to be either ConcatString or SmartString. It has become super gross

Even more overloaded functions may be required (like all the combinations?) like the RHS version

Why?

- Suppressing copies/postponing evaluation

- Expression templates - operators return a proxy for calculated value. Stores all necessary info to make calculated value but doesn't actually do it
- Allow arbitrary number in sequence of cascaded operations
    - String + string = proxy
    - Proxy + string = super proxy!
    - String + proxy = mega proxy
    - Proxy + proxy = mega super proxy! :D

Proxy + proxy makes it SUPER ugly. He would write a template alias to give a shorter name. Using auto still makes you have to tell it the type so gross. Maybe could write a function to return the type and use decltype.

He's overwrite of operator<< now using typename T so he overwrote it for everything! He made it promiscuous. Must design it so it only interacts with the minimum of what's needed.

WORD OF THE DAY -> Promiscuous

Keeping references in ConcatStrings can be troublesome with the destructor issue! Cause in exprs like: greeting + first + last, compiler is allowed to get rid of the greeting + first ConCatString temporary/t1. In our example it was a coincidence that our code works even though t1's destructor is conceptually invoked. Btw by t1 we mean: greeting + first

Build proxies - want copies of other proxies but reference to original type (string today, valarray in project 3). This is ok because the proxies will be small b/c they're references. Std::conditional? (you're right Jo)

We need to worry about lhs, rhs cannot go out of scope -> this is WRONG! (clients can throw ()s everywhere) we need to do the decision making for both lhs and rhs

He wrote choose_ref to find out if it's SmartString or not. Just the two because the other option is Proxy, no one cares how deep the proxy layers go for picking the type. Leaves (SmartString/valarray) are big, the expression trees themselves are small (ConcatString/proxy)

Disabling destructors is not a feasible solution.

He didn't do string literals :(which is analogous to our dealing with scalars