4/19
4/21
4/26
4/28
5/3
5/5

4/19

Project 3- due 4/25 (monday) instead of 4/22 (Friday)

Most subtle - proxies look to compiler like temporaries. With proxy, we have to copy other proxies but valarrays are stored as references

Scalars are treated like valarrays of infinite length but certainly not represented that way

Current ConcatString doesn't work anymore (specificallly operator[]) b/c a char* (string literal) doesn't have a .size() method. Need to create a wrapper for the literal to give it the size function (also put the operator[] in the wrapper). For int: size = max_int, operator[] = the int itself, ALSO give it a value_type AND a print method.

Can’t we just promote literal string to SmartString? Not a good idea (this has already been done in C++ for backward compatibility with C strings). We do not want to allocate an array on the heap, just to print stuff. (otherwise this idea would have worked.)

Remember life times are issues, especially for lazy evaluation. And we only want to copy inexpensive things (like our proxy or a scalar wrapper)

Base types in proj 3 are int, float, double, complex(float/double)

Currently we have two operator+ for string literals b/c need one for literal on the left & literal on the right and this is just with SmartString. Need to then also do it with the proxy too = a million methods (actually 2 more) BUT the total operator+ is 8 b/c of smartstring + proxy combos. Chase claims he can take it down to 1!! But 3 might be better b/c 1 is too complex whereas 3 can be pretty simple each.

Going to make his proxy/concatstring a smartstring but templates make magic happen. OH. Oh god… turns the smartstring class into basically a wrapper. Clients feel like they are using a

SmartString but Chase will secretly wrap it (using string = Wrap<std::string>). This way you have to be wrapped up to use our operator+ (-, *, /). Gotta remember to wrap the result back up before returning so it can also be used with our operators

We can redefine regular string types into a wrapper, by writing: using string = Wrap<std::string>; His operator +(Wrap<S1> …) only works if the objects are wrapped up. Inside the function, we don't care about the wrapper, we unwrap immediately. What we care about is just the string inside. (Basically wrap is just an interface key. To make sure we are not declaring a promiscuous overloaded function). Here S1 and S2 are types that implement the String concepts.

```
Template <typename S>
struct Wrap : public S { // inherit from S so that the wrap can easily be stripped
	using S::S; // inheriting constructors
	Wrap( S const& s) : S(s) {}
};

template<typename S1, typename S2>
Wrap<ConcatString<S1, S2>>
operator+(Wrap<S1> const& lhs, Wrap<S2> const& rhs) {
	S1 const& left {lhs};   // inherited from S so that the wrap can easily be stripped
	S2 const& right {rhs};

	ConcatString<S1, S2> result {left, right};
	Wrap<ConcatString<S1, S2>> wrapped_result {result};

	return wrapped_result;
}
```

The 3 operatorX per arithmetic operator (+, -, *, /)

- 1 w/ wrapper to replace proxy + valarray, valarray + proxy, proxy + proxy, and valarray + valarray
- 2 for scalar/literals - one with scalar on left, one with scalar on right. Yay!
- the main point here is that we still have all the types as we had before, it's just the number of overloaded operators that will be reduced in an efficient manner.

Now operator<< can just take Wrap<T> as parameter, now it's not so promiscuous. Because, it's only able to print a wrapper (not any shit, just a wrapper shit d=)

Expression templates = all about delayed evaluation until a more "global view" is attained. Want to communicate more through types than state = the compiler can optimize more.

Also, wrappers are the best. Used both for scalars & proxies. Hot dang, so good. Need to make sure just to wrap up valarrays, not vectors as a whole. Unwrap instantly and wrap back up absolutely last. Just like a rain jacket

4/21

You can mock up a new language in C++ using the aliasing…

Units matter (10.4 cm, or meter or etc.)
- NASA mars lander crashed to to using different metric systems (US vs Standard)
- Sol: Design a set of basic types (that other types can be derived from). Addition of two numbers respect the declared types (by conversion if possible, like adding distances in two different units, inch and cm can be converted to one of them). Or like dividing distance by time, which makes the type to be speed! Also, compiler does some checking, like adding speed to distance (units don't match, so it won't compile)

Case studies:
- Substitution failure!
    - 3 overloaded versions of foo (we expect the compiler to always choose the overload that is most specific with the use case)
        - foo (42); // calls the int const&
        - Blah b; foo(b); // general
        - foo(wrap(b)); // wrapped obj
    - So far so good. Now the wrap function:
      ```
      template <typename T>
      Wrap<T> wrap(T const& x) {
          return Wrap<T>{x};
      }
      ```
    - How can we force compiler to choose the less specific overloaded version of foo()? Naive sol: write specific versions of foo for specific types (but this is not good, we need a more general solution)
    - We can write the function in such a way that Blah encounters an issue with the Wrapped version of foo() due to the difference in (1) the arguments or (2) the return type
    - C++ obscure rule: In the overload resolution, compiler has to ignore the overloaded versions that do not compile and among the remaining ones pick the most specific one (Substitution failure is not an error, a.k.a. SFINAE)
    - Inside the Wrap class/struct, using magic = int;)
      ```
      template<> struct Wrap<Blah>  : Blah {
          using Blah::Blah;

      };
      ```

```
Template <typename T>
        void foo(Wrap<T> const&, typename<T>::magic v = 0) {...}
```

- Hack: Added an argument, magic, that's not really used but is a signal. The type of the second argument is in any Wrapper, but doesn't exist when using the Wrap<Blah> (the specific type we want to exclude). So that the specific version won’t compile, and compiler is forced to move forward with the general version (SFINAE)
- Now let’s do one example with operator[] (the method only being invoked, when operator[] is defined for the type), First step, declaring overloaded versions of bracket. We wanna make sure if type T does not have operator[], the 2nd one disappears). So insert a substitution failure - function overload resolution failure (Substitution failure happens for selecting, not after the compiler has selected which method to run and the compile error happens AKA needs to be a failure with the signature not the body of the method, and FYI signature is arguments and return type d=)

-
  ```
  Template <typename T>
  void bracket (T const& x) {
          decltype(x[0]) tmp;
          cout << “specific to types with op[]\n”;
  }
  int main(void) {
          bracket(42);
          Int x[10];
          bracket(x);
  }
  ```
- decltype lets us extract a type and use it, yay. Use statically. But setting default value to 0 is troublesome because what if it's a char *. And we're also assuming x[0] is valid
  ```
  void bracket (T const& x, decltype(x[0]) arg = 0) { // causes a signature compile
                                                      // error if passing int
          decltype(x[0]) tmp;
          cout << “specific to types with op[]\n”;
  }
  ```
- Wave that victory flag! Yaay \m/
- Adding an extra function is not the best practice (anti-pattern)
- void bracket (T const& x, decltype(x[0]) arg = (decltype(x[0])){}) {
    - Now we're depending on a default constructor, GROSS
    - Decltype does not evaluate the expr! It’s a compiler directive to statically typed expression. So providing x[0], while idx of 0 is out of bounds doesn’t cause a problem.
- Another technique to achieve the same goal is to add dummy shit to the template arguments.

    - template<typename T, typename X = declytype(x[0])> // But there's no x so make it T{}[0] // the expr will not be evaluated (so no redundant construction). THe problem is we don’t know if compilers exist.
    - typename X = decltype(*((T*)0)[0])
        - Use decltype(std::declval<T>()[0]) instead. Can only use declval when you're not going to execute that code. Only in type analysis
        - Makes me want to vomit. Use void pointer and other grossness to just get a type. This code if executed causes NPE, but the good point here is that it won’t, but still it had a valid type which we need
    - If you actually execute declval it’s gonna crash! Cause it dereference the nullptr. Yet, it generates an expression with the correct static type!
- All of these have been checking for operator[] that takes an int. What if it took other arguments? (Then we’re boned? d=)
- Problems with using return type to accomplish substitution failure - constructor has no return type and operator methods have specific return types
- Compiler is not able to distinguish which one of the overloaded function is more specific for int[]. We need to find a way to make the 1st version to become less good, so that compiler chooses the other one.
    - One technique is to add (T const&, …) the … to argument list. Make it variadic which a compiler doesn't prefer b/c the number of arguments don't match as well. But it's not working?
- Now we wanna put the substitution failures in the return types of the functions and here’s fred()!
  /* use only if T::foo exists as a non-static method with 0 args */
  template <typename T>
  T fred(T const& x) { cout << “general\n”; return x; }
- Now he's basically making enable_if. If it's false, the type doesn't exist/there isn't a type therefore compiler cannot use that for the type. Use a meta function to figure out if the predicate should be true or false

```
        template<bool p, typename T>
        struct enable_if {
                using type = T;
        };
        template <typename T> struct enable_if<false, T> {};

template <bool p, typename T>
using EnableIf = typename enable_if<p, T>::type;

template <typename T>
EnableIf<???, T> fred(T const& x) {...}
```

- Now we have to find a meta function that gives complementary behavior, i.e., true if T::foo exists and false o.w. Chase claims he doesn't know how. WTF?? :D

- Like_it thing -> constexpr false for general but true for the types we like (int, float, double, complex<float/double>. Then use this value for enable_if<like_it<T>, return type>
    - I think we can use enable_if for the base types + SRank stuff?

4/26

He just said "tl;dr" but actually "too long didn't read"

Spring 2015 - Lots of substitution failure

MetaProgramming 3 today.

Metaprogramming = communicate through types

Two types - true_type and false_type, that's what our meta programs return. Chase makes a struct for TRUE and FALSE. They can just be empty to indicate what they want. But it's probably best to put a static constexpr bool = true/false so that you're able to easily convert from TRUE type to true value (TRUE::value, or standard C++ true_type)

Compile time decision - do you have a specific method (operator[], foo())? Using substitution failure relating to function overloading resolution. It can't build the signature b/c the type doesn't exist in the return type or param type or template type list default value. Doesn't look at body of the function until it's selected. So typos in the body won't get checked, uh-oh
Ex: template <typename T, typename test=decltype(declval<T>()*operator/function*)>

Enable_if is for if you already have a predicate, at compile time you can get a true or false. Therefore easy for only allowing certain types, hard for detecting if a type has certain methods.

Now if both overloads are available, how do you get the compiler to select the one you want?
Variadic argument version is less specific/more general = compiler makes it a lower priority
Template <typename T, typename… not_used> // the variadic args aren't used
foo(not_used…) // Since it's not used, it's an empty set of arguments which is less specific than the other foo(void)

Problem:
typename test=decltype(declval<T>().foo()) // Only looks for foo(void), not foo(int)
decltype(&T::foo) // worked with something that had only one foo(). But when foo was overloaded in T's class(foo(void) and foo(int)), the address of foo was ambiguous and it didn't work

Tuple - variadic template (void might be silly though) with two or more types. standard Pair = old version with two types. Used for some complicated type (grouping of types) but not important enough to make a class for it. Functions are returning tuples and you catch them with auto.

```
template <typename... T>
struct Tuple;

template <typename T>
struct Tuple<T> {
   T val;
};

template <typename T, typename... OtherArgs>
struct Tuple<T, OtherArgs...> : public Tuple<OtherArgs...> {
   T val;
};
```

Omg, he has Tuple inheriting from Tuple… A recursively defined whatever. The base case is a specialization when there's exactly 1 template argument. Could do when zero arguments but that's kind of weird. But accessing the individual elements (for multiple arguments) is a pain, for now at least.

Now he's adding the loop struct. Its integer argument index - keeps track if we're at the base case. Index = 0 is the base case as shown by the second version of loop struct which is used when index = 0. The typename T is the actual tuple

```
template <int index, typename T>
struct loop {
   static constexpr int value = 1 + loop<index - 1, T>::value;
};

template <typename T> struct loop<0, T> {
   static constexpr int value = 0;
};
```

loop<7,Whatever>::value is a compile time constant. Cool. No runtime overhead!

We also have if/else logic compile time with the compile time resolution happening

Now want access to nth field in tuple. I knew we'd get here. Need to go up N base classes. This is insane. Tuple_loop and tuple now work in concert. Wanted tuple_loop to be specialized with <index, Type<T, Tail…>> where Tail is the last template argument w/ … that way short_tuple = Tuple<Tail…> . Instead he had to write tail_tuple meta stuff to find it for him

Here extract is the recursive method that does the n-1 times typecast and return the reference to the value(@ compile time), it’s not a const ref, so it’s also assignable :)

"I'm going to wave the victory flag and give up" - The great Chase

4/28

Not Chase today. Today we're talk MKS

Qi is using XCode - "It's for pussies"
Real coders use sublime! --Nima

```
Template <typename ...Args>
Void print (Args… x) // How to print all the args?

Template <typename T, typename… Args>
Void print (T const& t, Args const & … args)
{
	Cout << t << endl;
	print<Args…>(args…); //This will call this same method until you get down the the last
```
element, which you'd just print with a *void print(T const & t)* method
```
}
```

What if you specifically want to retrieve the nth argument when calling print? Gotta do something like Chase did on Tuesday

```
template<int index, typename T, typename… Args>
Struct retrieve_index {
	Using type = retrieve_index<index-1, Args…>::type;
};

template<int index, typename T, typename… Args>
Struct retrieve_index<0, T, Args...?> {
	Using type = T;
};

//But this is silly b/c then you'd have to do it for all the possible indices
template<int index, typename T, typename R, typename… Args>
Struct retrieve_index<1, T, R, Args...?> {
	//Using type = R; //This is assuming at least two args
	Using type = typename retrieve_index<1-1, R, Args…>::type; //How more general looks
};
```

Now MKS - adding distances = OK, time + volume = not OK

Exponents:
ME: meter exponent
SE: second //
GE: gram //
Example: 1 meter - 1 ME, 0 SE, 0 GE
	Gravity 9.81 m/s^2 - 1 ME, -2 SE, 0 GE

Ratio of two integers for scalars -> Num/Den, used to determine between meter, kilometer, inches, feet, etc

But what is T? I don't know… Precision? Double

3b - Substitution failure is not a requirement

5/3

Project 3b - due Wednesday night, tests run on everyone. Then really due Friday night, officially. Then release test code to us, and we can submit code up until Sunday night. Grading script runs Wednesday & Monday. Manual regrade only if you submit by Wednesday

More MKS stuff today (MKS2). Talking about all the base units and how they get all the derived units.

"I don't even remember what work is" - Chase

Integral constant = in symbol table
Floating point & string literal = stored in memory

One pain in the ass is that we have to remember the order of integer exponents and have them in the right place. We want to address this problem now:

Said specifically to look at MKS comments in prep for exam.

MassPL (placeholder) for each unit type you have AKA the exponents

Why is it not the best to use std::tuple in our context? (think about it) He created his own thing called TypeSequence

Want to keep it flexible enough to add other units besides distance, time, and mass like temperature super easily without having to depend on the specific order of the units. (the index of exponents become relative, which makes it maintainable!)

Write function to get length of tuple with meta programming, compile time const. He has get_length_TypeSequence in MKS2.h, that'll be helpful. Recursive, of course. Length is one longer than the length of it's tail until you get down to the last one(base case), that'll return length = 1.

len(list) = 1 + len(list[1:]) # python syntax…
len([]) = 0 # base case

Now find, to get which exponent thing we want, find_TypeSequence. Tell what you're looking for and pluck off the head until you get to the exponent you want is the first template argument in the TypeSequence. Specialization #2 (the base case of find_TypeSequence) is more specific because it doesn't use Head AKA has less template arguments but still fits. Will always finds what he wants because compile time errors will stop people from looking for exponents he hasn't used.

So much stuff. increment_ExponentIndex -> for when you need to increase the exponent value for a specific unit (distance, time, mass). ExponentSequence is just a <> list thing of ints/exponents, like a completely int tuple. Another recursive thing

So cool! Line 212 of the code. Lets say <0, **0, 0, 0**> should become <0, **1, 0, 0**>. We strip it off to get <**0, 0, 0**> to <**1, 0, 0**> and then we go back to the original domain, which it becomes <0, 1, 0, 0>.

buildSequence - application, create unitless type - exponent sequence with the right number of zeros. Need to build up from nothing <> to <0, 0, 0>. Adds a zero to the beginning of a sequence.

Create arbitrary sequence of specified length and types (which only have to be specified in one place). And everything else has to work with it so that you only need to change one line.

- Const for immutable
- Constexpr for actual constants

Easy to do the multiplying of stuff together - just add the exponents! Division = subtract exponents.

We are now waving the victory flag. Need variadic template functions working both on types and integer sequences to get it all together.

Made MKS overall a nice layered thing. So instead of 8 template arguments, there's only 2.

Inches * milliseconds -> probably shouldn't just have the result units be inches/millisecond. Maybe should be something better, like the standard scale. Build list of standard scales (like meter/second), and when we do a calculation, pick the standard scale (sort of like casting) but no if there's no standard scale, just return the naively created one. (FINAL EXAM). This is what he wants us to think more about.

Need to make sure we can extract exponent and scale properly. You can add things that have different scales as long as they have the same exponent

Operators are extremely uninteresting in new version, same as old MKS. Need to KNOW tuple loop for final

5/5

Today is "Thing" in examples folder…

Boost - Any: runtime polymorphic data type

"More than I expected but less than I hoped" - Chase

Dynamically the type of x changes, use void* to keep it generic within the class. Make the assignment operator a member template to take any type and the operator will know the type. Then you can run the constructor

```
Template <typename T>
Thing& operator=(T const & obj) { ptr = new T{obj}; return *this; }
```

How to print? The print method doesn't have that type T so you don't know how to print it. Use virtual functions to solve this. Want to remember type of object from assignment operator to the print function. Need a pointer to the root of an inheritance hierarchy where the subclasses know how to print or know what T is. Then delegate through interface/ThingHelperInterface to print.

ThingHelperInterface - needs to know how to print and clean up, virtually. Make ThingHelper to inherit from it and it'll remember the type T by just storing the obj.

```
Template <typename T>
Thing& operator=(T const & obj) { ptr = new ThingHelper<T>{obj}; return *this; }
```

The copy of the object stores the copy of the obj along with the Helper instead of just making the obj all by itself. Now we need to change void* to ThingHelperInterface*, yay. Now printing is easy: ptr->print(out); Reminder - const goes on print methods

Clean-up: make sure there's no memory leak. Make sure that when a Thing, x, goes out of scope and the destructor gets rid of the ThingHelper on the heap. ThingHelper has an object and the destructor will get that. Just got to make sure we have our virtual destructors and Thing deletes our pointer

What'd we do? Wrote template functions where we used compile time type deduction to capture a type.

Int y = x.get<int>(); // Return an int if an int was the last thing assigned to x. Otherwise throw an exception
Int y = (int) x; // Chase liked this one more, some type conversion. If int was the last thing assigned to x, it'd return the value. Otherwise it would default construct the type and return it.

```
Operator int(void) //Only for conversion to int
{
        ThingHelp<int>* cast_ptr = dynamic_cast<ThingHelper<int>*>(ptr);
        If (cast_ptr)
                //Have to figure out if last assigned is int, no template meta programming
                //Typeid works but is lame.
                //Does ThingHelper point to an int? ThingHelper<int>? Dyanmic cast
        {
                Return (int) cast_ptr->obj; // Have to type cast it to access object anyways
        }
        Else
        {
                Return int{};
        }
}
```

Template <typename T>
Operator T (void) //Now can convert to anything
{ //As above, just replace all *int* with *T*
}

Final exam
Expression Template

Template Meta-programming

- If then else (like that, conditionals)
- MKS original, templates that had ints as arguments
- ChooseType
- SFINAE
- Variadic templates
- Recursive instantiation of templates

From first half -> today's problem. Is static polymorphism a possibility? Or should I use dynamic

Test length - 2-ish hrs

Thing, when assigning to an int you are allocating an object on the heap. If your type is able to be stored in <8 bytes, you shouldn't need to put it on the heap. How can you change the design so if the object being assigned is small, nothing is placed on the heap. Involves some low level manipulation of memory, C type casts. Now think about a function acting differently if the argument is < a certain size (Great on the final)