Good things to know

- Templates takes practice and patience

Suggested Coding Practice

- Write self documenting code
- Use very detailed variable names
- Have variable names start with lower case letters
- Class member names (**private** variables) should start with and underscore (_)
- **Public** class variables should be captialized
- Indent your code 2–3 spaces within functions and loops

Templates

Templating allows one to have variable data types for functions and classes. For example let's say we wanted to find the infinity norm of a vector. We would have to have multiple functions that took in different kinds of vectors.

OverLoaded Functions Doing the Exact Same Thing

```
1  int
2  infinityNorm(vector<int> inputVector){
3    int largestValue = inputVector[0]; // just setting a value
4    for (unsigned int index = 0; index < inputVector.size(); index++){
5         if (inputVector[index] > largestValue){
6           largestValue = inputVector[index];
7         }
8    }
9    return largestValue;
10 };
11
12 double
13 infinityNorm(vector<double> inputVector){
14   double largestValue = inputVector[0]; // just setting a value
15   for (unsigned int index = 0; index < inputVector.size(); index++){
16        if (inputVector[index] > largestValue){
17          largestValue = inputVector[index];
18        }
19   }
20   return largestValue;
21 };
```

Well that just seems unnecessary ... if only there was a better way ... OH WAIT .. there is ... templating is the way. With a single templated function we can now handle unsigned int, int, float, double.

Templating a Function ... One Function To Rule Them All

```
1  template <class InputType>
2  InputType
3  infinityNorm(vector<InputType> inputVector){
4    InputType largestValue = inputVector[0];
5    for (unsigned int index = 0; index < inputVector.size(); index++){
6      if (inputVector[index] > largestValue){
7        largestValue = inputVector[index];
8      }
9    }
10   return largestValue;
11 }
12
13 int main(){
14   unsigned int numberOfValuesToInitializeVectorWith = 4;
15   int valueToFillIntVectorWith = 1;
16   double valueToFillDoubleVectorWith = M_PI;
17   vector<int> inputVectorInt(numberOfValuesToInitializeVectorWith,
18                              valueToFillIntVectorWith);
19   vector<double> inputVectorDouble(numberOfValuesToInitializeVectorWith,
20                              valueToFillDoubleVectorWith);
21
22    // actually specify template type for completeness
23   int largestValueInVectorSpec = infinityNorm<int>(inputVectorInt);
24   double largestValueInDoubleVectorSpec = infinityNorm<double>(inputVectorDouble);
25
26   // didn't specify template type because compiler could deduce the type
27   int largestValueInVectorNonSpec = infinityNorm(inputVectorInt);
28   double largestValueInDoubleVectorNonSpec = infinityNorm(inputVectorDouble);
29 return 0;
30 }
```

Let's consider an example that is a little more heavy on the templating ...

Structs for Use in Next Example

```
1  struct Tiger{
2    typedef int FavoritePrecision;
3    static const unsigned int NumberOfLegs = 4;
4
5    Tiger( unsigned int numberOfStripes):
6    _numberOfStripes(numberOfStripes){
7    }
8    FavoritePrecision
9    topSpeed(double angleOfInclinationInRadians){
10      return 1;
11   }
12
13   unsigned int _numberOfStripes;
14 };
15
16 struct Turkey{
17  typedef double FavoritePrecision;
18   static const unsigned int NumberOfLegs = 2;
19
20   Turkey(unsigned int numberOfFeathers):
21   _numberOfFeathers(numberOfFeathers){
22   }
23   FavoritePrecision
24   topSpeed(double angleOfInclinationInRadians){
25     return cos(angleOfInclinationInRadians);
26   }
27
28   unsigned int _numberOfFeathers;
29
30 };
```

A class templated on two input classes

```
template<class Animal1, class Animal2>
class TwoAnimalRace{
public:
  TwoAnimalRace(Animal1 animal1,
                Animal2 animal2):
                _animal1(animal1),
                _animal2(animal2){
  }
  // simply renaming data types for use inside this class
  typedef typename Animal1::FavoritePrecision Animal1Precision;
  typedef typename Animal2::FavoritePrecision Animal2Precision;

  // these must be static const so that the compiler is GUARANTEED they will not change
  // also they are capitalized because they are public variables
  static const unsigned int Animal1NumberOfLegs = Animal1::NumberOfLegs;
  static const unsigned int Animal2NumberOfLegs = Animal2::NumberOfLegs;

  void
  runRace(double angleOfRaceTrackRadians){
    Animal1Precision animal1Speed =
      _animal1.topSpeed(angleOfRaceTrackRadians);
    Animal2Precision animal2Speed =
      _animal2.topSpeed(angleOfRaceTrackRadians);
    if (double(animal1Speed) > double(animal2Speed)){
      printf("Animal 1 won!\n");
    }
    else{
      printf("Animal 2 won!\n");
    }
  }

 // this is the shorter version of using the templated number
 // because we used the public variables up top
  array<Animal1Precision,Animal1NumberOfLegs>
  maxSpeedOfAnimal1Legs(double angleOfRaceTrackRadians){
    array<Animal1Precision,Animal1NumberOfLegs> legArray;
    for (unsigned int legIndex = 0; legIndex < Animal1NumberOfLegs; legIndex++){
      legArray[legIndex] = 2 * _animal1.topSpeed(angleOfRaceTrackRadians);
    }
    return legArray;
  }

  // this is the longer version
     // template argument 1) note that typename is required if you
     // ask a template datatype what
```

```
76      // it's favorite precision is
77      // template argument 2) note that typename is not required here
78      // for the number of legs because the templated argument is only a number
79    array<typename Animal2::FavoritePrecision,Animal2::NumberOfLegs>
80    maxSpeedOfAnimal2Legs(double \addtolength{?}{?}
81    ngleOfRaceTrackRadians){
82      array<double, Animal2NumberOfLegs> legArray;
83      for (unsigned int legIndex = 0; legIndex < Animal2NumberOfLegs; legIndex++){
84        legArray[legIndex] = 2 * _animal2.topSpeed(angleOfRaceTrackRadians);
85      }
86      return legArray;
87    }
88
89  private:
90    Animal1 _animal1;
91    Animal2 _animal2;
92  };
93
94  int main(){
95   // let's make a race between animals
96   unsigned int numberOfStripesOnTiger = 10;
97   Tiger tigger(numberOfStripesOnTiger);
98
99
100  unsigned int numberOfFeathers = 1000;
101  Turkey shakeNBake(numberOfFeathers);
102
103  TwoAnimalRace<Tiger,Turkey> twoAnimalRace(tigger, shakeNBake);
104  twoAnimalRace.runRace(M_PI/4.);
105
106 }
```

Namespace

Namespaces are used to define scope of where functions, classes, structs, etc reside in. Scope helps you distinguish between two Element classes that may have the same name and take in the same arguments with the same datatypes such as the small strain bar element and the small strain with damage bar element. Both have the same exact names and constructors, but the internal workings are different.

Also usually libraries will have their own namespace such as std:: or Eigen:: .

```
namespace Elements{
  namespace SmallStrain{
    class BarElement{
      BarElement(Vector<double,2,1> nodePositions,
                 double stiffness)
      // filled with public variables, constructor, methods, private variables
    }
  }
  namespace SmallStrainWithDamage{
    class BarElement{
      BarElement(Vector<double,2,1> nodePositions,
                 double stiffness)
    // filled with public variables, constructor, methods, private variables
    }
  }
  namespace FiniteStrain{
    class BarElement{
      BarElement(Vector<double,2,1> nodePositions,
                 double stiffness)
    // filled with public variables, constructor, methods, private variables
    }
  }

}
```