Intro

Suppose we want to create our own generic classes and cuntions? A (function template) describes a family of functions.

```
template<typename T>
int compare(const int &v1, const int &v2) {
  if(v1 < v2) return -1;
  if(v1 > v2) return 1;
  return 0;
}

\\Client code
compare(1,3) //compare <int>
compare(3.14, 2.7) //compare<double>
```

EX:

```
short s = 5;
compare (1, s);
compare(s, s);

template<typename T1, typename T2> //allows the comparison of two different data types
int compare (const T1& a, const T2& b) {
  if(a < b) return -1;
  if(a > b) return 1;
  return 0;
}

int compare(double, double);
int compare(string, string);
int i = compare ( 2.0, 3.0); //this sees an exact match for double double and thus calls it
int i = compare(s, s); //no perfect match for short short, the compiler might try to
    convert it to int int, but if it doesnt it will try to match it to the template it does
    this by trying to create a short short compare function from the template
```

Instantiation

Explicit ARguments

We can explicitly state template parameters argument types (important for specifying return types).

```
template <typename T1, typename T2, typename T3>
T1 sum (const &T2 a, const T3 &b) {
  return a + b;
}

//Client code
float f = sum<float>(10, 3.14);
```

```
//we could try it another way
tempalte <typename T1, typename T2, typename T3>
T1 sum(const T2& a, const T1 &b) {
  ...
}

float f = sum<float>(2.0, 3); //this returns a compiler error
float g = sum<int, float, float>(2.0, 3); //this compiles because the compiler doesnt need
    to infer things
```

Class Templates

Define a generic class lie a containter with an element type specified by parameter. The original purpose was to create parameterized class definitions.

```
template<typename T>
class Stack {
public:
  Stack();
  void push(const T&);
  T top() { return items_[top_]; }
  T pop();
private:
  T items_[STACK_SIZE];
  int top_;
}

template<typename T>
void Stack<T>::push(const T &elem) {
  top +=1;
  items_[top_] = elem;
}
...
```