**TEMPLATE CODE:**

```
template <typename T>
   //indicates the following class
   //or function is a template
void function(T a[], T p2)
   //T type must be passed as a
   //parameter!
{
    T total = T(); //see*
    ...
}

void function(int a[], int p2)
{...}  //you can write exceptions the
       //compiler will default to

template <typename T1, T2>
    //multi-type templates work too!
void f2(T1 a[], T2 b[])
```

```
/* In templates, the compiler uses
template argument deduction (checks the
parameters) to figure out what functions
to use. Non-template matches have
priority, then te plate matches. If the
call does not match the template exactly,
there will be a compile time error!*/
```

```
/* Using the term T() allows you to initialize to the "default constructor" of whatever type you use. For numbers, this is 0. Bools are false, strings are empty, chars are the 0 byte. */
```

**ALWAYS PLACE TEMPLATES IN THE HEADER FILE**

```
/* when you have a function that traverses the entire leftover list each time, the algorithm has time complexity O(N^2): N(N+1)/2 = 1/2N^2+1/2N)*/
```

**Template Classes**

```
template <typename T>
class something
{…};

template <typename T>
void something<T>::f1(T a)
{…};
```

**Inline Functions:**

```
/* anything declared inside the class
declaration is automatically inline: the
compiler copies the code wherever you
call the function, speeding up the
program because there's less jumping.
declare external functions inline like
this: */

inline void sclass::f1()
{}

/* setting large functions inline will
greatly increase your exe file size
*/
```

**Runtime Time Complexity**

```
/*written in terms of "Big 'O' Notation" O(some function of N), where N is the number of data terms.
Things to consider if complexity varies:
Best Case Time
Worst Case Time
Average Case Time
Does your data cause you to generate the Best/Worst case often? */
```

```
/* sometimes, for things like sorting, you consider complexity of swaps over comparisons (or some other specific action) because it takes significantly longer. Usually, the longer one is not swaps, because you should SWAP POINTERS */
```

**INFIX TO POSTFIX**

```
Initialize postfix to null
Initialize the operator stack to empty
For each character ch in the infix string
    Switch (ch)
        case operand:
            append ch to end of postfix
            break
         case '(':
            push ch onto the operator stack
            break
        case ')':
             // pop stack until matching '('
            While stack top is not '('
              append the stack top to postfix
              pop the stack
            pop the stack  // remove the '('
            break
         case operator:
            while the stack is not empty and the stack top is not '('
             and precedence(ch) <= precedence(stack top)
                 append the stack top to postfix
                  pop the stack
            push ch onto the stack
            break
While the stack is not empty
    append the stack top to postfix
    pop the stack
```

**Evaluating Postfix**

```
Initialize the operand stack to empty
    For each character ch in the postfix string
        if ch is an operand
            push the value that ch represents onto the operand stack
        else // ch is an operator
            set operand2 to the top of the operand stack
            pop the stack
             set operand1 to the top of the operand stack
             pop the stack
             apply the operation that ch represents to operand1 and operand2,
                and push the result onto the stack
    When the loop is finished, the operand stack will contain one item,
      the result of evaluating the expression
```

```
Passing functions as parameters to functions:
double g(int x);
double integrate(int xlow, int xhigh, double f(int))
{
      double y= (*f)(x) //or f(x);
}

main()
{
      double area = integrate(low, high, g);
}
```