1 Templates

- The compiler is smart enough that sometimes, we don't have to specify the type of the templated function.

```
template <typename T>T min (T x, T y) {
    return x < y ? x : y;
}

int f() {
    int x = 1, y = 2;
    int z = min(x, y); //This will work w/o stating it's an int.
    int a = min<int>(x, y); //This will also work.
    char w = min('a', 'c'); //This works, T is a char,
    auto f = min(3.0, 5.0); //This works, where T and f will be a double.
}
```

- Note this only works for function templates. NOT CLASSES.
- Consider the example:

```
void foreach(AbstractIterator start, AbstractIterator finish, int(*f)(int
    while (start != finish) {
        f(*start);
        ++start;
    }
}

template<typename Iter, typename Func>
void for_each(Iter start, Iter finish, Func f) {
    while (start != finish) {
        f(*start);
        ++start;
    }
}

void f(int n) {cout << n << endl;}
int a[] = {1, 2, 3, 4, 5};
for_each (a, a+5, f); //Valid.
```

- The first version will possibly fail in a few situations. If the AbstractIterator doesn't support ++, !=, or *(dereferencing), then it will not work. If the function we pass isn't callable (no idea how the hell that would happen), it will also not work.
- We look at the second example. That's valid! We don't have to explicitly claim the type of each typename, as above, due to this being a function template.
- We could change all the names of Iter and Func and it would still work.

2 Algorithms

- From the ¡algorithm¿ library.
- It contains a suite of template functions, many of which work over iterators.
- We can use these on the final/assignment, but not needed at all.

```
for_each() {
    //seen above.
}

template<typename Iter, typename T>
Iter find(Iter first, Iter last, const T &val) {
    //returns iterator to first element in first (inclusive) to last (exc
    //if val is not found, returns last.
}

int count(Iter first, Iter last, const T &val) {
    //returns the number of occurances of val.
}

template<typename InIter, typename OutIter>
OutIter copy (InIter first, InIter last, OutIter result) {
    // copies the values from first (inc) to last (exc)
    // starting from result.
}

OutIter transform(InIter first, InIter last, OutIter result, Func f) {
    //Basically for_each and copy combined.
}
```

3 Lambda

- Recall we really only used lambda so we could avoid defining functions that we only really used once, so we didn't need a name.
- We can do that in C++ as well. This is useful for template algorithms, as we may only need the required function once.
- We can achieve this like so:

```
vector<int> v {...};
bool even (int n) {return n % 2 == 0;}
int x = count_if(v.begin(), v.end(), even); //Obviously valid

//OR
```

```
int y = count_if(v.begin(), v.end(), [](int n){return n % 2 == 0;}); //Al
```

- Yes. "[]" is our "lambda". Note we don't have to specify the return type; assuming we aren't idiots, the compiler will be able to guess what our return type is.
- If you want to SAVE this lambda, what you can do is say auto x = []...;

¡++¿