Introduction to C++

Templates

Templates

- **C++ implements genericity with templates**
  - Resolved at compile time
  - No runtime checks
- **Write a class or function once, for use with a variety of types**
  - Average, largest, smallest and many more
  - Type safe collections – and algorithms that work on them
  - Often rely on operator overloads
- **Much of the Standard Library is template-based**
  - Old name: Standard Template Library, STL

Template Functions

- **Write the function with a placeholder**

```
template <class T>
T max(T& t1, T& t2)
{
return t1 < t2? t2: t1;
}
```

- **When using the function, compiler may deduce the type you're using**

```
max(33, 44)
max(s1,s2)
max(p1,p2)
max<double>(33, 2.0) //will return double
```

Template Classes

- **Write the class with a placeholder**

```
template <class T>
class Accum
{
private:
   T total;
public:
   Accum(T start): total(start) {};
   T operator+=(const T& t){return total = total + t;};
   T GetTotal() {return total;}
};
```

- **When using the class, specify the type**

```
Accum<int> integers(0);
Accum<string> strings("");
```

Template Specialization

- **Sometimes a template won't work for a particular class**
  - Operator or function is missing (and you can't add it)
  - Logic in the operator won't work for this case
- **First choice: add the operator or function with the right logic**
- **Second choice: specialize the template**

Summary

- **Templates add tremendous power to C++**
  - Compile time checks mean no runtime hit
- **Author of code that uses templates must ensure that types are compatible with the template chosen**
- **Template Specializations let you handle special cases**
- **Many C++ developers never write a template**
- **All good C++developers should use them**
  - Save development time
  - Error checking and edge cases aren't forgotten
  - Flexibility in the face of future enhancements