Templates

Kate Gregory

@gregcons www.gregcons.com/kateblog

C++ Implements Genericity with Templates

Resolved at compile time

No runtime checks

Write a Class or Function Once

Average

Largest

Smallest

Type safe collections

Algorithms that work on them

Often rely on operator overloads

Much of the Standard Library Is Template-based

Collections

Sorting

Searching

Standard Template Library

Template Functions

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

- Write the function with a placeholder type

```
max(33, 44)
max(x,0)
max(s1,s2)
max(p1,p2)
max<double>(33, 2.0)
      //will return double
```

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

Template Classes

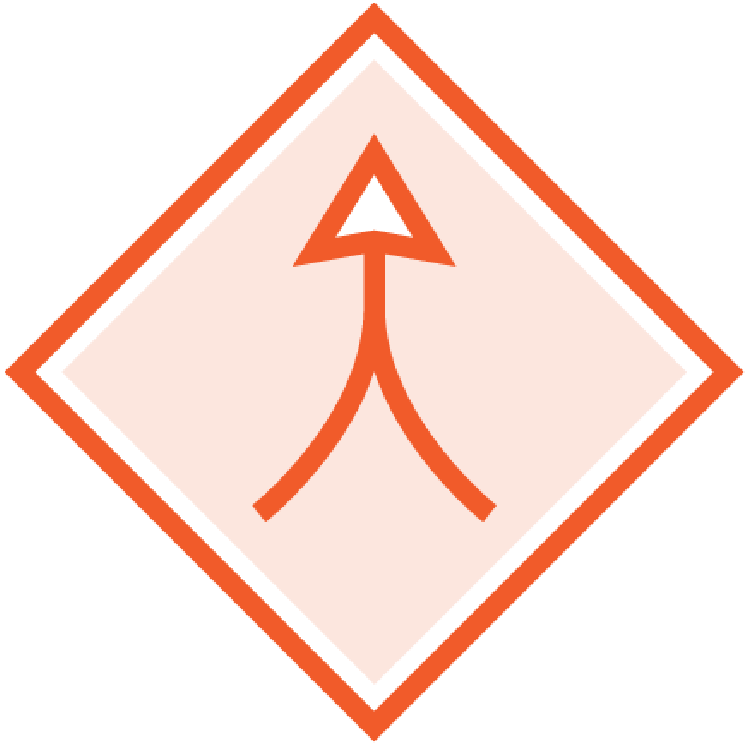
```
template <class T>
class Accum
{
private:
   T total;
public:
   Accum(T start): total(start) {};
   T operator+=(T const& t)
       {return total = total + t;};
   T GetTotal() const {return total;}
};
```

- Write the class with a placeholder type

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

- When using the class, specify the type
- C++ 17 template deduction may allow you to omit the type hint here

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