Introducing Templates

Template Functions

- C++ allows a function to be parameterised on type as well as values
  - C++ compiler creates a different version of the function for each required type
  - Note: Very different to Java Generic Functions!

Declaration:

```
template <typename T>
T Square( T ToBeSquared );
```

Implementation:

```
template <typename T>
T Square( T ToBeSquared )
{
  return ToBeSquared * ToBeSquared;
}
```

The template statement is followed by one or more type parameters. The word 'typename' is used before the name used for the type parameter, 'T' in this case. The word 'class' may be used instead of typename

The template statement is duplicated before the implementation of the function. In this example the template affects only the following function. To use it for several functions, enclose them in {braces}

Calling Template Functions

- Template functions called as normal functions
  - Compiler generates a new version of the function for each type

```
int n = 3, m = 4;
double d = 5.5;
cout << Square( n ) << endl;
cout << Square( d ) << endl;
cout << Square( m ) << endl;
```

Compiler generates a version of the function using ints

Compiler generates a new version of the function using doubles

Compiler uses the existing int version of the function

Template Functions and Type Safety

- Template functions will enforce type safety

A (trivial) Multiply function:

```
template <typename T>
T Multiply( T x, T y );
```

```
template <typename T>
T Multiply( T x, T y )
{
  return x * y;
}
```

```
int n = 3, m = 4, i;
double d = 5.5, e = 4.5, f;
i = Multiply( n, m );
f = Multiply( d, e );
i = Multiply( n, d );
f = Multiply( d, n );
```

OK

Fails - mixed types. Note that the compiler will not promote the int to a double (as it would in a normal function call)

Using Multiple Type Parameters

- More than one type parameter can be used

An example - A variation on the Multiply function that allows different argument types:

Declaration:

```
template <typename TypeA, typename TypeB>
TypeA Multiply( TypeA x, TypeB y );
```

All type parameters specified in the template statement must actually be used in the function that follows it

Implementation:

```
template <typename TypeA, typename TypeB>
TypeA Multiply( TypeA x, TypeB y )
{
  return x * y;
}
```

Templates and Overloading

- Templates provide an alternative to overloading of functions
  - Function only requires to be written once
  - Template functions only instantiated for required types
- Template functions can be overloaded
  - With other template functions or normal functions

A small disadvantage of templates is that the reason for compiler errors can be less obvious - the template may be shown as the source of an error caused by calling it with invalid arguments

As they are a fairly new feature in C++, the compiler messages for errors may be less informative than for well-established features

Template Classes

- C++ allows classes (also structs) to be templated
  - No inherent overloading of classes
  - Very useful for implementing collection types – STL

```
template <typename T>
class Number
{
private:
 T number;
public:
 void Get()  { cin >> number; }
 void Show();
};
```

The basic template syntax is the same as for functions, although not all the parameter types need to be used

Class member functions implemented in the .CPP file must be prefixed by the template statement, for example:

```
template <typename T>
void Number<T>::Show()
{
 cout << number;
}
```

Using Template Classes

- Specify values for template parameters when instance of template class is created
  - Intrinsic or user-defined types can be used

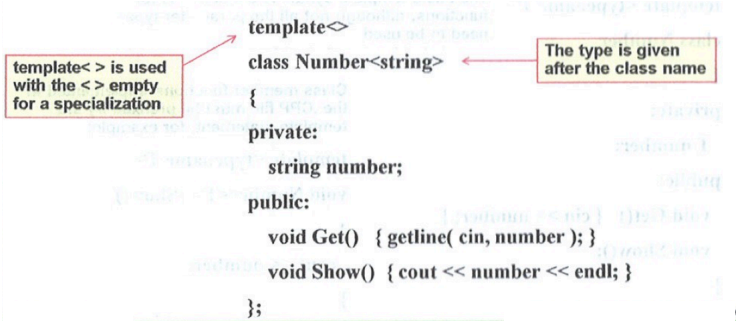
```
Number<int> aNum;
aNum.Get();
aNum.Show();

Number<double> another;
another.Get();
another.Show();
```

The type(s) are listed in <brackets>, separated by ',' if there are more than one

Template Specialisation

- Template class can be explicitly instantiated for a particular type
  - Requires a complete implementation of the class




```
template<>
class Number<string>
{
private:
   string number;
public:
   void Get()   { getline( cin, number ); }
   void Show()  { cout << number << endl; }
};
```

Templates and Constants

- Class can be parameterised using type or constant value

```
template <typename T, int Counter>
class Numbers
{
private:
   T number;
public:
   void Get()   { cin >> number; }
   void Show();
};
template <typename T, int Counter>
void Numbers<T, Counter>::Show()
{
   for ( int n = 0; n < Counter; n++ )
      cout << number;
}
```

```
Numbers<int, 4> aNum;
aNum.Get();
aNum.Show();
Numbers<double, 7> another;
another.Get();
another.Show();
```