Introduction to Templates

Object-Oriented Programming in C++

Our Task

- Write a function that swaps two values:

```
void swap( int& v1, int& v2 ) {
    int tmp = v1;
    v1 = v2;
    v2 = tmp;
}
```

- OK, works for integers, but what about?
  - other numeric types, like float, double, …
  - other types, like char, string
  - user defined types (classes)

Use Templates

- Template is a function or a class with the type parameterized.
- Seen use:
  - `vector<int> V;                  // explicit`
  - `max(5, 6); max(5.6,2.7); // implicit`
  - `...`
- Template declaration:

Use **typename** or **class** ?

```
template <typename T>
void foo( T v ) { . . . }
```

```
template <class T>
void foo( T v ) { . . . }
```

Function Templates

- Type independent code (well, kind of):

```
template <typename T>
void swap( T& v1, T& v2 ) {
    T tmp = v1;
    v1 = v2;
    v2 = tmp;
}
```

Why kind-of?
Swap makes
requirements
about the type.

- Use:

```
int main() {
    int    ix=5,   iy=6;
    double dx=5.0, dy=6.0;
    swap<int>(ix, iy);     // Specify type, like STL
    swap<double>(dx, dy);
    swap(ix, iy);          // This works too! Compiler
    swap(dx, dy);          // figures out the type for us.
}
```

Type Substitute

- Requirements about parameterized types are called template concept.
  - e.g. that type must define a copy constructor and assignment operator, as in the swap template.
- Types that meet these requirements are called a model of that concept.
- Can substitute parameter type with any type that is a model of the template concept.
  - e.g. works for most (basic) types in swap.

Template Specialization

- Can write different template functions depending on parameter (type)

```
template <typename T>
void swap( T& v1, T& v2 ) {
    T tmp = v1;
    v1 = v2;
    v2 = tmp;
}

void swap( string& v1, string& v2 ) {
   // More efficient swap for strings.
    v1.swap( v2 );
}
```

Template Parameters

- Multiple parameters:

```
template <typename T, typename U>
foo( T v, U u ) {
   U var;
   . . .
}
```

Must be integrals, pointers, or references.

- Non-type parameters

```
template <typename T, int maxSize>
Void foo( ) {
   T arr[maxSize];
   . . .
}
 . . .
foo<char,100>();
```

Can even use defaults!
…, int maxSize = 10

Class Templates

- Classes (and structs) can be made templates:

```
template <typename T>
class Stack {
    void push( T elem );
    // ...
private:
    T *m_data;
};

// Definition (must also be in header file ☹)
template <typename T>
Stack<T>::push( T elem ) {
    // ...
}
```

Aha! A new type for each parameterization!

Templated Class Methods

- Classes can have templated methods:

```
class Foo {
public:
    template <typename T>
    bool isValid(T elem) { return sizeof(elem) <= sizeof(int); }
    // ...
private:
    // ...
};

int main() {
    Foo f;
    cout << f.isValid( 1 )   << '\n';
    cout << f.isValid( 1.0 ) << '\n';
}
```

Although cannot be **virtual**.

Template Meta-Programming

```
template <int N>
struct Factorial {
    enum { value = N * Factorial<N - 1>::value };
};

template <>
struct Factorial<0> {
    enum { value = 1 };
};

// Factorial<4>::value == 24
// Factorial<0>::value == 1
const int x = Factorial<4>::value; // == 24
const int y = Factorial<0>::value; // == 1
```

Example from Wikipedia: http://en.wikipedia.org/wiki/Template_metaprogramming

Hands On Example

Summary

- Template allows us to parameterize types
- Function templates
- Class templates
- Templated class methods
- Template Meta-Programming
- Further reading
  - C++ FAQ Lite (http://yosefk.com/c++fqa/templates.html)
  - Book C++ Templates: The Complete Guide.