Functors in C++

By Swaminathan Bhaskar
11/04/2008

An object that can be called like a function is known as a **Function Object** or a **Functor**.

In C++, an object of a class can be invoked like a function if it overloads the function call operator by defining the ***operator()*** member function.

The following is a simple example:

```
#include <iostream>

using namespace std;

class Simple {
	public:
		void operator()() {
			cout << "Welcome to Functor in C++" << endl;
		}
};

// ----- Main -----

int main() {
	Simple s;
	s();
}
```

In the main() function above, the object s of type Simple is invoked like a function producing the output:

```
Welcome to Functor in C++
```

When the function call s() is made, the compiler actually makes a call to the function **s.operator()()**. Cool, ain't it !!!

The following is another example that allows one to pass arguments:

```
#include <iostream>
#include <string>

using namespace std;

class Print {
	public:
		void operator()(const string& arg) {
			cout << arg << endl;
		}
};

// ----- Main -----
```

```
int main() {
	Print p;
	p("Functor With Arguments !!!");
}
```

A **Functor** can be passed as an argument to another object for callback. In C, this would have been accomplished through Function Pointers, while in C++ its more elegantly done through Functors.

The following example illustrates the function callback ability using Functor:

```
#include <iostream>

using namespace std;

class Double {
	public:
		int operator()(int arg) {
			return 2*arg;
		}
};

template <typename T>
void double_ints(T op) {
	for (int i = 1; i <= 5; i++) {
		cout << op(i) << " "; // Callback into T
	}
	cout << endl;
}

// ----- Main -----

int main() {
	double_ints(Double()); // A temporary object is passed
}
```

The above could have also been achieved using a global function instead of a Functor.

The following code exhibits the same behavior using a global function:

```
#include <iostream>

using namespace std;

int Double(int arg) {
	return 2*arg;
}

template <typename T>
void double_ints(T op) {
	for (int i = 1; i <= 5; i++) {
		cout << op(i) << " "; // Callback into T
	}
	cout << endl;
}

// ----- Main -----
```

```
int main() {
	double_ints(Double); // Global function Double is passed
}
```

The main advantage of using a Functor over a global function is that a Functor can maintain state between calls since it is an object. Instead of multiplying by 2, what if we wanted to multiply by an arbitrary value ?

The following example depicts this scenario:

```
#include <iostream>

using namespace std;

class Multiply {
	private:
		int _i;

	public:
		Multiply(int i) : _i(i) {}

		int operator()(int arg) {
			return _i*arg;
		}
};

template <typename T>
void multiply_ints(T op) {
	for (int i = 1; i <= 5; i++) {
		cout << op(i) << " "; // Callback into T
	}
	cout << endl;
}

// ----- Main -----

int main() {
	multiply_ints(Multiply(5)); // A temporary object is passed
}
```

This would be messy to achieve with a global function. It would also not be thread-safe !!!

The C++ Standard Templates Library (STL) uses Functors extensively. Examples of some of the functors from STL are: **less**, **negate**, **plus**, **greater**, etc. The STL built-in functors are defined in the header **<functional>**.

The following example shows a simple example using STL functor **less**:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;

int main() {
```

```
	vector<int> vec;

	// Add numbers 1 to 5 in random order
	vec.push_back(2);
	vec.push_back(5);
	vec.push_back(1);
	vec.push_back(4);
	vec.push_back(3);

	// Display elements in the vector
	cout << "Unsorted: ";
	for (int i = 0; i < vec.size(); i++) {
		cout << vec[i] << ' ';
	}
	cout << endl;

	// Sort the list in ascending order using functor less
	sort(vec.begin(), vec.end(), less<int>());

	// Display elements in the vector (sorted)
	cout << "Sorted: ";
	for (int i = 0; i < vec.size(); i++) {
		cout << vec[i] << ' ';
	}
	cout << endl;
}
```