Module 2: Generic Programming and Policy-based Design

For course "Distance Learning C++, Programming Models, Libraries and Parallel Computation"

Summary

The exercises are grouped into three main categories:

- Basic templates: creating and debugging template code
- Intermediate: using inheritance and composition with templates
- Advanced (Josuttis/Vanderoorde, code JVD); CRTP patterns, traits, policy-based design, generic patterns

A. Basic Templates

1. (My first Template Class)

Create a template class that model one-dimensional intervals and ranges. The interface is:

```
template <class Type = double> class Range
{

private:

	Type lo;
	Type hi;

public:
	// Constructors
	Range();                                    // Default constructor
	Range(const Type& low, const Type& high); // Low and high value
	Range(const Range<Type>& ran2);            // Copy constructor

	// Destructor
	virtual ~Range();

	// Modifier functions
	void low(const Type& t1);      // Sets low value current range
	void high(const Type& t1);     // Sets high value current range

	//Accessing functions
	Type low() const;              // Lowest value in range
	Type high() const;             // Highest value in the range

	Type spread() const;           // High - Low value

	// Boolean functions
	bool left(const Type& value) const;       // Is value left of range?
	bool right(const Type& value) const;      // Value right of  range?
	bool contains(const Type& value) const;   // Range contains value?

	// Operator overloading
	Range<Type>& operator = (const Range<Type>& ran2);

};
```

Write the source code for these member functions. Create a test program to check if your code is OK.

In particular, address the following issues:

a) Each member function should be created and tested
b) Test your code with `int`, `double` and `string`. Which ones work and which one does not?
c) The accessing functions `low()` and `high()` return copies of the range's internal state. Modify the code so that they return `const` references to the internal state. What are the advantages?
d) Document the requirements that the underlying type `T` in `Range<T>` should satisfy so that clients will know for which specific types this class will function. In other words, what are the *constraints* on T?

2. (Composing ranges)

We now wish to create a class called `Box<T1, T2>` that models a pair of `Range<T>` instances:

```
// Box.hpp
//
// A two-dimensional domain consisting of two ranges
// Common functionality; template specialisations may add
// extra member functions.
//
// (C) Datasim Education BV 2009
//

template <typename T1=double, typename T2= double>
      class Box
{ // A box is composed of two ranges

private:
            Range<T1> dir1;   // first direction
            Range<T2> dir2;   // second direction
public:

            // Constructors

            // Accessing functions

            // Properties
            bool InBox(const T1& first, const T2& second) const;
};
```

Answer the following questions:

a) Create the code for constructors (default, copy and two ranges as arguments), accessing functions for the internal data and testing if a point is in a box
b) For certain instantiations of `T1` and `T2` (for example, `T1` = `T2` = `double`) we wish to add a number of new member functions, namely:
   `area()` : the area of the box
   `cog()` : the centre of gravity of box
c) For the `cog()` function you will need to add a new member function to `Range<T>`. What is it called?
d) In all cases, the `Box<T1, T2>` box class should delegate to the interface of `Range<T1>` and `Range<T2>`.

e) For initialization and specialization of class templates you will need the JVD, pages 27-33 and page 88.

3. (Template Arguments and Template Parameters, page 90 JVD)
C++ supports both typenames and integral types as template parameters. This feature is useful when creating compile-time (fixed-size) arrays, usage in traits classes and other applications.
In this exercise we wish to create a class that models fixed-sized mathematical arrays. We have constructors, accessing operators and mathematical operations. The class interface is:

```
template<typename Type, int N> class VectorSpace
{
private:

	Type arr[N];

public:
	// Constructors & destructor
	VectorSpace();
	VectorSpace(const Type& value);		// All elements get this value
	VectorSpace(const VectorSpace<Type, N>& source);
	virtual ~VectorSpace();

	// Selectors
	int Size() const;
	int MinIndex() const;	// First index
	int MaxIndex() const;	// Last index

	// Some properties
	// Inner product
	Type innerProduct (const VectorSpace<Type, N>& p2) const;
	Type Norm() const; // The l Infinity norm, max value in array
	Type componentProduct() const;		// The product of all components

	// Numeric operations
	VectorSpace<Type, N> operator - () const; // The negative of a vector

	VectorSpace<Type, N> operator + (const VectorSpace<Type, N>& v2) const;
	VectorSpace<Type, N> operator - (const VectorSpace<Type, N>& v2) const;

	// Add and subtrct offsets to each coord
	VectorSpace<Type, N> operator + (const Type& offset) const;
	VectorSpace<Type, N> operator - (const Type& offset) const;

	// ** Template member functions ** Premultiplication by a field value
	template <typename F> VectorSpace<Type, N>
				friend operator * (const F& scalar, const
						VectorSpace<Type, N>& pt);

	// Operators
	Type& operator[](int index);  // Index operator for non const
	const Type& operator[](int index) const;  // Index operator for const
	VectorSpace<Type, N>& operator = (const VectorSpace<Type, N>& source);

};
```

Answer the following questions:

a) Implement he .hpp and .cpp files for this class (you need to type the above function declarations, it' s good practice)

b) Create a program to test all the functions in the class (you also need to create a corresponding print() function):

```
int main()
{
	const int N = 10;
	VectorSpace<double, N> myArray;

	for (int j = myArray.MinIndex(); j <= myArray.MaxIndex(); j++)
	{

		myArray[j] = double (j);

	}

	print(myArray);

	VectorSpace<double, N> myArray2 = myArray; // No other size works!!

	VectorSpace<double, N> myArray3 = myArray2 - myArray;

	print(myArray2);
	print(myArray3);

	double factor = 0.5;
	VectorSpace<double, N> myArray4 = factor * myArray3;
	print(myArray4);

	double ip = myArray.innerProduct(myArray2);


	return 0;
}
```

c) Pay particular attention to the *template member function* in this class as it is a useful feature (see also JVD page 46, using Stack as an example)

4. ((Global) Template Functions)

It is possible to define C-style non-member functions having template arguments. In some cases we can group these functions in a namespace because they logically belong together and we also do not wish to implement as member functions. These kinds of functions are common in STL and boost.

Two examples are:

```
		template <typename V, int N>
			void print(const VectorSpace<V,N>& vec);

		// D = A + bB + cC; A,B,C, D are vectors, b,c scalars
		template <typename V, int N>
		void TripleSum(VectorSpace<V,N>& D,
			const VectorSpace<V,N>& A, const VectorSpace<V,N>& B,
```

```
                    const VectorSpace<V,N>& C, V a, V b);
```

Answer the following questions:

a) Implement these functions (create separate `.hpp` and `.cpp` files and use a namespace to hold these functions together)

b) What is the performance difference when comparing calls to `TripleSum()` against a naïve application of operator overloading:

```
D = A + b*B + c*C;
```

Investigate the reasons for the discrepancy (e.g. are temporary objects being constrcted and destructed?)

5. (CRTP Pattern, JVD page 295-299)

The objective in this exercise is to model *static polymorphism* using the CRTP pattern and to compare the performance improvements when compared with *dynamic polymorphism* and virtual function in classic OOP. To this end, we wish to define flexible algorithms in a class hierarchy where some of the code is invariant in the base class and some code needs to be implemented in derived classes. Instead of the usual approach, we define a CRTP hierarchy:

```
template <typename D>
            struct Base
{
      double algorithm(double x)
      { // Template method pattern

            // Variant part I
            double y = FuncV(x);

            // Invariant part
            y += 2.0;
            if ( y <= 21.0)
                  y = 3.3;
            else
                  y = 3.4;

            // Variant part II
            double z = FuncVII(x);

            // Postcondition
            return y * z;

      }

      inline double FuncV(double x)
      {
            return static_cast<D*> (this) -> FuncV(x);
      }

      inline double FuncVII(double x)
      {
            return static_cast<D*> (this) -> FuncVII(x);
      }
```

```
	virtual ~Base() {}
};
```

This is only an example but what is special is how the base class 'uses' its derived classes as template parameters. We also note how the base delegates to its derived classes and that it is compile-time because there are no virtual functions used.

A class that can be used as a template argument in `Base<D>` is defined as:

```
struct DerivedCRTP : Base<DerivedCRTP>
{
	inline double FuncV(double x) { return exp(-x*x) * log(x);}
	inline double FuncVII(double x) { return tanh(x) * sinh(x);}
};
```

An example of use is:

```
double x = 3.1415;
Base<DerivedCRTP> d;

for (long j = 1; j <= NBig; j++)
{
	x = x * (100000); temp = d.algorithm(x);

}
```

Answer the following questions:

a) Under which circumstances is it better to use CRTP rather than dynamic polymorphism, and vice versa? Focus on performance, how often the functions are called and how easy it is to maintain the code

b) Choose a test case of a class hierarchy that you have created and create a CRTP version. Compare the performance of two solutions (in some cases CRTP is [8, 12] times faster but the speedup depends on the problem and its formulation)

6. (Templates and Inheritance, JVD Chapter 16)
It is possible to derive a template class or a specialized template class from another template class or even from a non-template class. In this exercise we wish to create a generic composite pattern that models recursive data structures. To this end, we implement the recursive data structure using an STL `list` and the class interface is given by:

```
#include <list>

template <typename T> class GenericComposite: public T
{
private:
	// The element list using the STL list
	std::list<T*> sl;
	void Copy(const GenericComposite<T>& source);
```

```
public:
	// User can use the STL iterator
	typedef typename std::list<T*>::iterator iterator;
	typedef typename std::list<T*>::const_iterator const_iterator;

	// Constructors and destructor
	GenericComposite();              // Default constructor
	GenericComposite(const GenericComposite& source);// Copy constructor

	virtual ~GenericComposite();   // Destructor

	// Iterator functions
	iterator begin();                // Return iterator at begin of composite
	const_iterator begin() const; // const iterator at begin of composite
	iterator end();                  // Return iterator after end of composite
	const_iterator end() const;    // const iterator after end of composite
	// Selectors
	virtual int Count() const;     // The number of elements in the list

	// Add functions
	void push_front(T* s);   // Add element at the beginning of Genericlist.
	void push_back(T* s);    // Add element at the end of Genericlist.

	// Remove functions
	void RemoveFirst();      // Remove first element
	void RemoveLast();       // Remove last element
	void RemoveAll();        // Remove all elements from the list
	void Remove(T* t);       // Delete pointer and remove from list

	// Prototype pattern
	virtual T* Copy() const;

	// Operators
	GenericComposite& operator = (const GenericComposite& source);
};
```

Answer the following questions:

a) Implement the bodies of the above member function declarations. You will need to determine a memory management policy such as where memory is created and by whom and when is it deleted (in later sections we show how to alleviate this chore by using *smart pointers*).

b) What are the constraints on the type `T` in `GenericComposite<T>`? In other words, which functions must instances of `T` implement in order for the composite class to work?

c) Create a simple class hierarchy with one base class `B` and two derived classes `D1` and `D2` (for the moment they can have minimal functionality but a polymorphic `print()` function would be useful as well as having print statements in all destructors)

d) What are the advantages of generic composites? What things do you have to watch out for in the current implementation?

7. (Template Template Parameters, JVD pages 50, 102, 111)

In exercise 6 we use STL `list` to implement the `GenericComposite<T>` class. But we now wish to choose other data structures as well, for example `vector`. To this end, we employ the template template parameter mechanism. The syntax is a bit terse and for this reason we motivate it by a scoped-down example of a collection. First, the data structure is:

```
template <typename T, template <typename ELEM, typename Alloc> class
Container, typename TAlloc = allocator<T>  >
                  class GenericComposite: public T
{ // Admittedly, a bit tricky syntax
private:
      // The element list using the STL list
      Container<T*, TAlloc> elements;

public:
      GenericComposite()
      {
            elements = Container<T*, TAlloc>();
      }

      void add(T* t)
      {
            elements.push_back(t);
      }

      void print() const
      {

            cout << "Size: " << elements.size() << endl;
      }

      //
};
```

We how use the following stripped-down class hierarchy whose instances can be composites:

```
struct Shape
{

};

struct Point : Shape
{

};
```

Finally, we need a test program to show how to use the new code:

```
int main()
{

      GenericComposite<Shape, vector> myComposite;
      GenericComposite<Shape, list> myComposite2;

      Shape* s = new Point;

      myComposite.add(s);
      myComposite2.add(s);

      myComposite.print();
      myComposite2.print();
```

```
    delete s;

    return 0;
}
```

Answer the following question:

a) Use this example as a template on how to implement the template template mechanism for `GenericComposite<T>`
b) Test your new class with the program code from exercise 6