## Introduction

The design and structure of a program is more clearly understood in terms of abstract data types, such as queues and stacks. A parameterized type, or template, is a linguistic mechanism for implementing an abstract data type. A scientifically prepared library of templates is an asset for reusing well-understood abstract data types.

The C++ template mechanism for parameterized types is extended so one can specify requirements for arguments that are considered permissible to instantiate the parameters. The Z++ mechanism for this specification is called pattern.

## Pattern

The syntax for a pattern is similar to class. The pattern construct is used for defining other forms of specification. For a template, a pattern can specify types, or methods. Let us illustrate these with examples.

```
pattern template<T> Type_Pattern
      T : int, long;
      T : short*;
end;
```

In the above pattern T is template parameter and Type_Pattern is the name of the pattern. In this pattern we are specifying three types that can be used to instantiate a template parameter. On each line the sequence of types is separated with commas. The number of lines is optional. That is, all the intended types can be listed on one single line.

Now consider the following pattern.

```
pattern template<T> Method_Pattern
      long fun(int, double);
      int ?(long, short);    //function name is wild
      void operator+(int);
      const T& ?(int);    //function name is wild, return is template
      operator int(void); //conversion operator
end;
```

The above pattern lists the methods that the type instantiating a template parameter must define. These must be public methods of the instantiating type.

In a pattern, a question mark for function name means that we are only interested in the signature of the method, but not how it is named. The question mark will match any public method with the same signature, including overloaded operators.

The template parameter T in a pattern corresponds to the template parameter that the pattern will be attached to. This will become clear from following examples.

Let us now use the two patterns defined above in specifying conditions for the types that will instantiate template parameters. Here is a simple example.

```
template<type U : Type_Pattern, type V : Method_Pattern>
struct patterned_template
      U i;
      V d;
end;
```

That is, each template parameter that we wish to pattern is followed with a colon and the name of the pattern. In the above example both parameters are patterned. The pattern for parameter U is Type_Pattern. That means, only the types listed in Type_Pattern can be used to instantiate U. On the other hand, a type for instantiating the parameter V must have the (public) methods specified in Method_Pattern. For instance, the following is an example of a type that satisfies the conditions of Method_Pattern.

```
class patternType
      short s;
public:
      patternType(void);

      long fun(int, double);
      void operator+(int);
      int wild(long, short);
      const patternType& wild(int);
      operator int(void);
end;
```

## Conclusion

Z++ extends the usefulness of generic programming in several directions. For instance the definition of a template class can be separated from its implementation. Indeed, the Z++ template library incorporates the Z++ extended form of namespaces.

The pattern construct for specifying the arguments that may be used to instantiate a template parameter is essential in avoiding many obscure software defects. The construct is quite simple with intuitive semantics, and yet quite expressive in defining all desirable conditions that one would like to specify.