Introduction

The simple go-to mechanism can simulate all control structures. However, a program (as an encoding of a solution) written in terms of control structures is more maintainable. The measure of expressiveness of a language is the degree of its richness in well-defined coherent mechanisms for encoding solutions.

A programming language based on types needs an adequate supply of type construction mechanisms. Using simpler mechanisms to simulate well-tested ones results in obscure encoding. The simpler the set of type constructors, the less expressive the language, and therefore the more obscure and lengthy the encoding.

The enumeration type is indispensable for writing meaningful code. In this paper we present the several directions in which Z++ extends enumeration. One can associate any type to an enumeration literal rather than just the numeric type int. It is also possible to extend an enumeration type in a manner similar to derivation mechanism for classes.

Extension mechanism

Quite frequently we need to include more literals for an already defined enumeration type. We would like to do so in a way that the new type would have some connection to the existing type, somewhat like the notion of inheritance with regard to classes.

The following example illustrates the simplicity of extending enumerations in Z++.

```
enum basicFigures {_square, _rectangle, _triangle};
enum moreFigures : basicFigures {_parallelogram, _trapezoid};
enum mostFigures : moreFigures {_circle, _ellipse};
```

In this example, `moreFigures` extends `basicFigures`, and then `mostFigures` further extends the type `moreFigures`. That means an instance of `mostFigures` can take on values from `basicFigures`.

Z++ treats the literals of an enumeration type private to the type. Thus, the same literal can be reused in defining other enumerations. It is a Z++ convention that enumeration literals must be preceded with an underscore, and that no identifier can have a leading underscore. This makes it easy to distinguish enumeration literals from other strings. Ambiguities reported by the compiler can be resolved in the usual way, as in `basicFigures::_square`, and `mostFigures::_square`.

Operators

The C++ increment and decrement operators furnish the successor and predecessor functions for an instance of an enumeration type, as shown below.

```
mostFigures figure = _trapezoid;
figure++; // now figure is _circle
```

There are two bracket functions associated with an enumeration type, which retrieve the first and the last literals of the type. Below, the single brackets retrieves the first value, in this case the literal `_square`, and the double brackets retrieves the last literal `_ellipse`.

```
for (mostFigures figureCounter = [mostFigures];
     figureCounter <= [[mostFigures]];
     figureCounter++)

// loop body

endfor;
```

The bracket function can also be applied to an enumeration instance to retrieve its associated integer value. For example, consider the instance `figure` defined in previous example. Then the expression `[figure]` will evaluate to an integer instance with value of 4, for `_trapezoid`.

Collections

The type constructor collection is a generalization of enumeration allowing any type to be associated with literals of an enumeration. Traditionally, int is the type associated with an enumeration literal. To avoid confusion, a new type constructor, called collection is used for the generalized mechanism.

Consider the following definition for `Square`.

```
class Square : FigureBaseType
      //members
public:
      //constructors, methods, etc
end;
```

Suppose we have defined the classes `Rectangle`, `Triangle` etc, analogously. The `FigureBaseType` is the traditional abstract class for polymorphism. The following illustrates the type constructor collection. We shall discuss the semantics shortly.

```
collection basicFiguresType<basicFigures> {
                                _square<Square>,
                                _rectangle<Rectangle>,
                                _triangle<Triangle>

      basicFiguresType(void);
};
```

A collection generalizes a previously defined enumeration. The compiler will enforce the rule that a collection must use all the literals of its associated enumeration. For instance, we cannot leave out the literal `_rectangle` in the above definition.

An instance of a collection will consist of instances of types specified for the values of its literals. That is, the constructor for the above collection will call the constructors for `Square`, `Rectangle` and `Triangle`. The rules are the same as for classes. The default constructor will call default constructors, and a user-defined constructor for the collection will call the constructors listed by the user. This allows any desirable initialization for the values of a collection.

Reaching the Tag and individual Values

A collection is somewhat similar to a tagged union. The tag is internal to the collection instance, but it can be accessed and modified. The bracket function for an instance of a collection returns a reference to its tag. For instance, in preceding section we declared the instance `initialFigurs` using the default constructor. This sets the tag to the lowest value of `_square`. Consider the following lines of code. Recall that basicFigures is an enumeration type defined earlier.

```
[initialFigures] = _rectangle; //change collection tag
basicFigures someFigure = [initialFigures];
```

On the second line, the instance `someFigure` will be initialized with `_rectangle`. The increment and decrement operators also work the same way for collections as they do for enumeration. In this case, the value of the tag is incremented or decremented.

Collection can have it own methods for manipulating individual values. Furthermore, reaching individual values is needed for using collections in selection control structures. The syntax for reaching values is the same as reaching array cells. The following illustrates a possible use of collections in a switch statement. In each case we are invoking a different method on the object reached. For instance, `squareMethod()` is invoked on the instance of type `Square`, reached at `initialFigures[_square]`.

```
switch([initialFigures]) //switch on current tag
      case _square:
            initialFigures[_square].squareMethod();
      case _rectangle:
            initialFigures[_rectangle].rectangleMethod();
      case _triangle:
            initialFigures[_triangle].triangleMethod();
endswitch;
```

Derivation

Derivation for collections is similar to extending enumerations, except a collection can have methods, and the methods can be redefined. Thus, derivation for collections is more like single (public) inheritance. The methods of a collection are public by default, but can also be protected or private.

In order to extend a collection, first we must extend its associated enumeration type. Earlier, we extended the enumeration `basicFigures` to the enumeration `moreFigures`. Let us now extend the collection `basicFiguresType` to `convexFiguresType`.

```
collection convexFiguresType<moreFigures> : basicFiguresType {
                                _ parallelogram <Parallelogram>,
                                _ trapezoid <Trapezoid>

                  convexFiguresType(void);
};
```

Conclusion

Z++ extends enumeration in several useful directions facilitating more natural encoding. In addition, Z++ collection type constructor allows associating any type to enumeration literals.

Collection admits further extensions that are currently under research. Among the extensions is the notion of shared methods. There are times that a particular method is shared among types that do not lend themselves to inheritance without distorting abstraction. Each of these types could belong to a different derivation hierarchy, with that method cutting across them. In such cases, a shared method of a collection will apply the method to all of its values, furnishing an elegant way to express a commonality.