Introduction

The notion of array is as old as programming language itself. Each of the languages of C++, ADA and APL has its own useful flavors of dealing with arrays. However, no language extends object-oriented notions to arrays.

Z++ brings together the features of dealing with arrays from languages mentioned above, in a systematic and coherent manner. It then extends object-oriented notions to arrays.

The concept of arrays is important enough to be brought into the language rather than passing it over to the libraries. In particular, many orthogonal extensions, like operator overloading, require direct language support.

Preliminaries

The syntax for declaring a simple array is same as in C++. When the following statement is elaborated, the compiler does two things. First it makes a new type, if not already there, and then creates an instance of that type.

```
int integerArray[5];
```

The new type is internally named "`int[5]`" for static array of five integers. The term static refers to the index 5 (we are discussing a type as opposed to an instance).

Now consider the following lines.

```
int index = 5;
int dynamicArray[index];
```

In this case the compiler makes a new type and internally names it "`int[]`" for dynamic array of integers. Note that the index is not known at elaboration time. We shall see how to get the value of index at execution time.

The term static is not used in Z++ language. In this paper, static array implies that the compiler knows the sizes of dimensions of the array. On the other hand, dynamic implies that the sizes of dimensions can only be determined at run time.

The compiler makes type names for multi-dimensional arrays in a similar manner. For instance, "`double[5][7]`" and "`string[][]`". If the size of any one dimension cannot be determined at compile time, then the array is considered dynamic.

We use the term base-type to refer to the type of cells of an array. Two dynamic array types are compatible when their base-types are the same, and they have the same number of dimensions. Two array objects are compatible if they are instances of the same type, or instances of compatible types. Specifically two static array objects are compatible exactly when they are instances of the same array type. In other words, the sizes of their corresponding dimensions must be the same.

Static Arrays

A reasonable starting point would be to allow arrays of numeric types to be manipulated as plain numeric types. There is no danger of confusing an array object with an instance of a simple numeric type.

The following examples illustrate the simplicity of manipulating numeric arrays in Z++.

```
int a[5][7];   //two-dimensional static array of integers
a = 97;  //set all cells to 97
long b[5][7] = a;    //initialize b using a
short s[5][7] = 47;  //initialize all cells to 47

// First increment every cell of s. Then, set every cell of b
// equal to the product of corresponding cells of s and a.

b = ++s * a;
b += a;  // apply operator += cell-wise
```

Operator overloading

In previous section we saw that Z++ overloads operators to arrays of numeric base-types. The same direct support is provided for arrays of classes. Indeed, regardless of whether an array is dynamic or static, the methods of its base-type can be applied to the array object itself. This is convenient when the base-type overloads certain operators. For instance, consider the following.

```
struct MyBaseType
    // methods and members
    void operator+(int);
end;

MyBaseType a[19];
a + 23; //apply operator+(23) to every cell of the array
```

In the same vain, compatible arrays can be tested for equality or lack of it. The compiler simply applies operators `==` or `!=` to every corresponding pair of cells of the arrays. However, this is done more efficiently than it sounds. For instance, in case of testing for equality, as soon as two corresponding cells are found different, the test terminates.

Another useful consequence of extending overloading to arrays is the ability to pick the desired constructor. When declaring array objects of classes, any constructor can be used as opposed to the default constructor only. For instance, the following is a valid statement. Presumably, the arguments x, y and z are expected by a constructor of the type `MyClass`.

```
MyClass a[17](x, y, z);
```

The use of typedef

In Z++, dynamic, as well as static arrays can be created on the stack or on the heap (via operator new). The following dynamic array is created on the stack.

```
int DA[m][n];
```

The same (dynamic) array can be created on the heap (dynamically) as follows. Note that in Z++ typedef is reversed. First the new name is given, and then the type.

```
typedef dynIntArray int[][];
dynIntArray* DAP = new int[m][n](97); // initialize all cells to 97
```

Static arrays are created on the heap analogously, as shown below.

```
typedef integerTwenty int[20];
integerTwenty* SAP = new int[20];
```

Z++ is very sensitivity to type matching. For instance, the following statement may be permissible in C++, but it is a type-mismatch in Z++ because the type of object being created is `int[20]`, not `int*`.

```
int* p = new int[20];
```

For proper type checking, we need a means of pointing to dynamic arrays created on the heap. A degenerate pointer type serves just that purpose.

Consider the pointer DAP defined above. DAP is a (two-dimensional) degenerate (array) pointer of base-type int. The pointer GAP, defined below, is another example of a degenerate pointer.

```
typedef degenPointer int[][]*;
degenPointer GAP = DAP;
```

A degenerate pointer is specific to dynamic arrays, and cannot point to static arrays. The following is an error.

```
typedef degIntArrayPointer int[]*;
degIntArrayPointer q = new int[20]; // type mismatch
```

The point is that, dynamic arrays can have different sizes. A degenerate pointer has all the information needed for type checking except the sizes of the dimensions. Thus, a degenerate pointer represents the set of all compatible dynamic arrays to which it can point.

Sizes of Dimensions

When dealing with arrays in Z++ the use of loops is greatly minimized. Nevertheless, there are times that one needs to use an array in a loop. For a static array the sizes of dimensions are known at compile time. For dynamic arrays, we need run-time support.

The operator size used to get the length of a string can also fetch the sizes of dimensions of a dynamic array. This is illustrated below.

```
int DA[m][n]; // m and n are integer objects
size(DA, 0);  // returns size of first dimension, i.e. m
size(DA, 1);  // returns size of second dimension, the value of n
```

Conclusion

Z++ coherently brings together all the desirable features of arrays from successful languages. Furthermore, Z++ provides object-oriented extensions, which simplify method invocations on instances of arrays.