**Data Types– Part 2**

**Slide 1**

In part two of this presentation on data types we will discuss array types.

**Slide 2**

We begin by examining how languages differ with regard to specifying array bounds.

Both C and C++ treat array names as pointers to the type of the elements contained in the array. The two are interchangeable. An array element can be accessed both with an array subscript and by dereferencing a pointer.

In C and C++ array sizes are not stored, so checking whether an array subscript is out of bounds is not possible. By contrast, Java and Ada both generate run-time errors when a subscript is out of bounds.

Another feature that varies is whether the programmer can specifies both the lower and upper bounds of an array. Both Pascal and Ada allow both to be specified. In the C family of languages, the lower bound is always 0 and what is specified by the programmer is the array size.

Pascal coupled the bounds of an array with the array type, which made it difficult to write a subprogram with an array type that could handle arrays of different sizes.

Ada introduced the unconstrained array type to solve this problem. In the C family of languages, the size of an array is not considered part of the type, so this was never an issue.

In Java, like all objects, all arrays are dynamically allocated on the heap. So Java treats array names as references. As a result, array assignments produce shallow copies and array parameters are a copy of the array argument reference.

**Slide 3**

As our first example, we will examine an Ada program that contains an unconstrained array type.

Here is an example of a declaration of an unconstrained array type. Notice that the type of the subscripts is specified but not the bounds. The reason that the subscript type needs to be specified is that Ada allows subscripts that are any discrete type. For example, an array could be declared that had subscripts of an enumerated type.

When an actual array variable is declared whose type is an unconstrained array type, the bounds must be specified. The compiler needs that information in order to know how much space to reserve for that array. Keep in mind that all arrays are allocated on the compiler’s run-time stack in Ada.

This function accepts an array of percentages as a parameter and returns the maximum value in that array. Notice that the type of an array parameter can be an unconstrained array type, which allows arrays of different sizes to be passed to the function.

Ada saves the bounds of array variables so that they can be accessed at run-time. The attributes FIRST and LAST refer to the lower and upper bounds. Again notice that attributes are referenced using the single quote.

In Ada, array aggregates, as they are called, specify the values of an array. Such aggregates can be assigned to array variables, which is what this assignment accomplishes.

Here is the call to the above function. An array with specified bounds must be passed as the argument.

**Slide 4**

When using subprograms with array parameters that are intended to be called with arrays of different lengths, like the one we just examined, there is variation among languages on how to determine the array length.

In C and C++ when an array with an unspecified length is passed as a parameter, it is necessary to pass a second integer parameter that contains its length because these languages do not provide any mechanism to determine the length of the array argument.

By contrast, in Ada, as we saw in the previous example, there are attributes that provide this information. In addition to FIRST and LAST, which were used in the previous example, there is also an attribute LENGTH. Additionally the attribute RANGE can be used when the entire range between FIRST and LAST is required.

In Java, the technique is to append the array name with .length, which looks like an instance variable, but because arrays are not objects of any class, it really isn’t one. Language design choices of this kind make a language irregular.

**Slide 5**

When array variables are allocated on the stack, which is true in C, C++ and Ada, unless pointers are used, the array size usually must be a constant, so that the compiler can determine that amount of space needed at compile time. In Ada, there is one exception, which are dynamic arrays. The bounds of a dynamic array can be a variable, provided that the variable has been initialized before the array declaration is elaborated.

The variable Size is declared at the procedure level, but is not yet initialized.

It is subsequently initialized by allowing the user to supply its value.

The array Dynamic\_Array, whose upper bound is Size, is declared in an inner block, so it is not elaborated until this inner block is encountered. At that point, the variable Size has been initialized.

The program then reads in the appropriate number of integers needed to fill the array and then outputs them in reverse order.

**Slide 6**

Finally, let’s consider arrays with more than one dimension.

Both C and C++ have true multidimensional arrays, which allocate one contiguous block of memory for the entire array.

The compiler must calculate the array element position at compile time using the array bounds. For that reason an array parameter can only leave the first dimension unspecified.

Ada supports both true multidimensional arrays and arrays of arrays, that is, an array whose elements are arrays.

Java supports only arrays of arrays. The benefit of arrays of arrays is that ragged arrays are possible. The drawback is that accessing an array required multiple steps, so it is slightly slower.

**Slide 7**

Let’s examine a C++ program that contains a function with a two-dimensional array as the parameter.

Notice that the array parameter must specify the number of columns. The reason is that the number of columns is needed by the compiler to calculate the position of an array element. Only the number of rows is passed as a parameter.

The array argument is initialized here. Notice that C++ allows array constants to be defined for multidimensional arrays. The nested braces are optional. The array will be filled in row major order regardless.

When the function is called, only the number of rows is passed. Using a pointer parameter instead, it is possible to write a function that accepts a two dimensional array where both the number of rows and columns are passed as parameters. In that case the programmer must code the calculation to locate the position of the array elements.