A Book Chapter

By Narendra Kumar Chahar

on **Vectors, Arrays and Union**

Index:

Chapter 3

Vectors, Arrays and Union

Vectors:

In computing, sequence containers refer to a group of container class templates in the standard library of the C++ programming language that implement storage of data elements. Being templates, they can be used to store arbitrary elements, such as integers or custom classes. One common property of all sequential containers is that the elements can be accessed sequentially. Like all other standard library components, they reside in namespace std.

The following containers are defined in the current revision of the C++ standard: array, vector, list, forward_list, deque. Each of these containers implement a different algorithms for data storage, which means that they have different speed guarantees for different operations.

- array implements a compile-time non-resizable array.
- vector implements an array with fast random access and an ability to automatically resize when appending elements.
- Deque implements a double-ended queue with comparatively fast random access.
- list implements a doubly linked list.
- forward_list implements a singly linked list.

Since each of the containers needs to be able to copy its elements in order to function properly,the type of the elements must fulfillCopyConstructible and Assignable requirements. For a given container all elements must belong to the same type. For instance, one cannot store data in the form of both char and int within the same container instance.

Properties

array, vector and deque all support fast random access to the elements. list supports only bidirectional iteration, whereas forward_list supports only unidirectional iteration.

array does not support element insertion or removal. vector supports fast element insertion or removal at the end. Any insertion or removal of an element not at the end of the vector needs elements between the insertion position and the end of the vector to be copied. The iterators to the affected elements are thus invalidated. In fact, any insertion can potentially invalidate all iterators. Also, if the allocated storage in the vector is too small to insert elements, a new array is allocated, all elements are copied or moved to the new array, and the old array is freed. deque, list and forward_list all support fast insertion or removal of elements anywhere in the container. list and forward_listpreserves validity of iterators on such operation, whereas deque invalidates all of them.

Vector

The elements of a vector are stored contiguously. Like all dynamic array implementations, vectors have low memory usage and good locality of reference and data cache utilization. Unlike other STL containers, such as deques and lists, vectors allow the user to denote an initial capacity for the container.

Vectors allow random access; that is, an element of a vector may be referenced in the same manner as elements of arrays (by array indices). Linked-lists and sets, on the other hand, do not support random access or pointer arithmetic.

The vector data structure is able to quickly and easily allocate the necessary memory needed for specific data storage. This is particularly useful for storing data in lists whose length may not be known prior to setting up the list but where removal (other than, perhaps, at the end) is rare. Erasing elements from a vector or even clearing the vector entirely does not necessarily free any of the memory associated with that element.

Capacity and reallocation

A typical vector implementation consists, internally, of a pointer to a dynamically allocated array, and possibly data members holding the capacity and size of the vector. The size of the vector refers to the actual number of elements, while the capacity refers to the size of the internal array.

When new elements are inserted, if the new size of the vector becomes larger than its capacity, reallocation occurs. This typically causes the vector to allocate a new region of storage, move the previously held elements to the new region of storage, and free the old region.

Because the addresses of the elements change during this process, any references or iterators to elements in the vector become invalidated. Using an invalidated reference causes undefined behaviour.

The reserve() operation may be used to prevent unnecessary reallocations. After a call to reserve(n), the vector's capacity is guaranteed to be at least n.

The vector maintains a certain order of its elements, so that when a new element is inserted at the beginning or in the middle of the vector, subsequent elements are moved backwards in terms of their assignment operator or copy constructor. Consequently, references and iterators to elements after the insertion point become invalidated.

C++ vectors do not support in-place reallocation of memory, by design; i.e., upon reallocation of a vector, the memory it held will always be copied to a new block of memory using its elements' copy constructor, and then released. This is inefficient for cases where the vector holds plain old data and additional contiguous space beyond the held block of memory is available for allocation.

Specialization for bool

The Standard Library defines a specialization of the vector template for bool. The description of this specialization indicates that the implementation should pack the elements so that every bool only uses one bit of memory. This is widely considered a mistake. vector<bool> does not meet the requirements for a C++ Standard Librarycontainer. For instance, a container<T>::reference must be a true lvalue of type T. This is not the case with vector<bool>::reference, which is a proxy classconvertible to bool.[11] Similarly, the vector<bool>::iterator does not yield a bool& when dereferenced. There is a general consensus among the C++ Standard Committee and the Library Working Group that vector<bool> should be deprecated and subsequently removed from the standard library, while the functionality will be reintroduced under a different name.

Array:

In computer science, array programming languages (also known as vector or multidimensional languages) generalize operations on scalars to apply transparently to vectors, matrices, and higher dimensional arrays.

Array programming primitives concisely express broad ideas about data manipulation. The level of conciseness can be dramatic in certain cases: it is not uncommon to find array programming language one-liners that require more than a couple of pages of Java code.[1]

APL, designed by Ken Iverson, was the first programming language to provide array programming capabilities. The mnemonic APL refers to the title of his seminal book "A Programming Language" and not to arrays per se. Iverson's contribution to rigor and clarity was probably more important than the simple extension of dimensions to functions.

Concepts

The fundamental idea behind array programming is that operations apply at once to an entire set of values. This makes it a high-level programming model as it allows the programmer to think and operate on whole aggregates of data, without having to resort to explicit loops of individual scalar operations.

Iverson described the rationale behind array programming (actually referring to APL) as follows:

most programming languages are decidedly inferior to mathematical notation and are little used as tools of thought in ways that would be considered significant by, say, an applied mathematician.

The thesis is that the advantages of executability and universality found in programming languages can be effectively combined, in a single coherent language, with the advantages offered by mathematical notation. It is important to distinguish the difficulty of describing and of learning a piece of notation from the difficulty of mastering its implications. For example, learning the rules for computing a matrix product is easy, but a mastery of its implications (such as its associativity, its distributivity over addition, and its ability to represent linear functions and geometric operations) is a different and much more difficult matter.

Indeed, the very suggestiveness of a notation may make it seem harder to learn because of the many properties it suggests for explorations.

Users of computers and programming languages are often concerned primarily with the efficiency of execution of algorithms, and might, therefore, summarily dismiss many of the algorithms presented here. Such dismissal would be short-sighted, since a clear statement of an algorithm can usually be used as a basis from which one may easily derive more efficient algorithm.

The basis behind array programming and thinking is to find and exploit the properties of data where individual elements are similar and/or adjacent. Unlike object orientation which implicitly breaks down data to its constituent parts (or scalar quantities), array orientation looks to group data and apply a uniform handling.

Function rank is an important concept to array programming languages in general, by analogy to tensor rank in mathematics: functions that operate on data may be classified by the number of dimensions they act on. Ordinary multiplication, for example, is a scalar ranked function because it operates on zero-dimensional data (individual numbers). The cross product operation is an example of a vector rank function because it operates on vectors, not scalars. Matrix multiplication is an example of a 2-rank function, because it operates on 2-dimensional objects (matrices). Collapse operators reduce the dimensionality of an input data array by one or more dimensions. For example, summing over elements collapses the input array by 1 dimension.

Languages

The canonical examples of array programming languages are APL, J, and Fortran. Others include: A+, IDL, K, Q, Mathematica, MATLAB, MOLSF, NumPy, GNU
Octave, PDL, R,S-Lang, SAC, Nial and ZPL.

Scalar languages

In scalar languages like C, C# and Pascal, etc. operations apply only to single values, so a+b expresses the addition of two numbers. In such languages adding two arrays requires indexing and looping, which is tedious and error prone.

```
for (i = 0; i < n; i++)
```

```
 for (j = 0; j < n; j++)
     a[i][j] += b[i][j];
```

Union:

Unions Types

•A union is a type whose variables are allowed to store different type values at different times during execution

•Design issues

–Should type checking be required?

–Should unions be embedded in records?

Discriminated vs. Free Unions

•Fortran, C, and C++ provide union constructs in which there is no language support for type checking; the union in these languages is called free union

•Type checking of unions require that each union include a type indicator called a discriminated.

–Supported by Ada

Ada Union Types

```
type Shape is (Circle, Triangle, Rectangle);

type Colors is (Red, Green, Blue);

type Figure (Form: Shape) is record

Filled: Boolean;

Color: Colors;

case Form is

when Circle => Diameter: Float;

when Triangle =>

Leftside, Rightside: Integer;
```

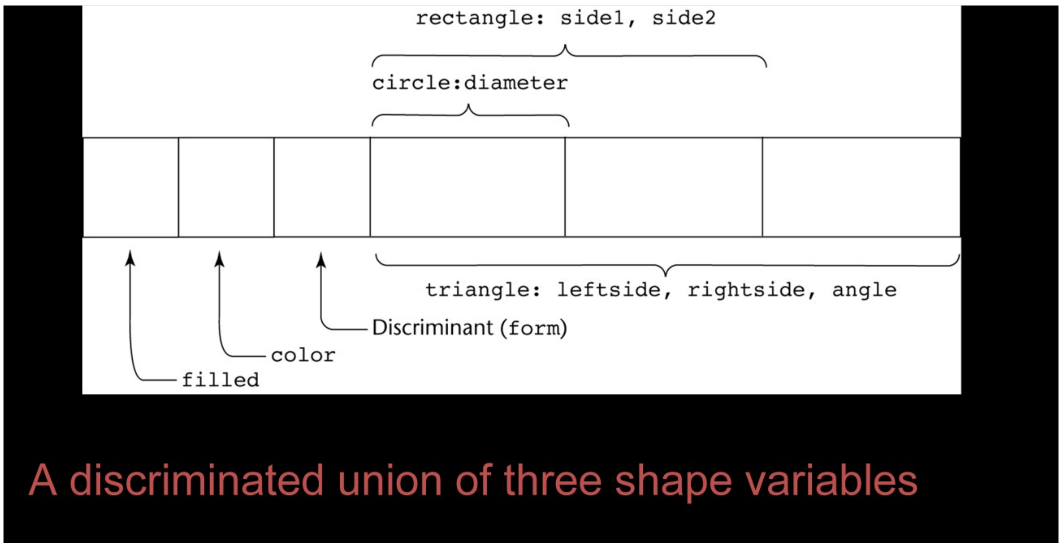
Angle: Float;

when Rectangle => Side1, Side2: Integer;

end case;

end record;

Ada Union Type Illustrated


rectangle: side1, side2
circle:diameter
triangle: leftside, rightside, angle
Discriminant (form)
color
filled
A discriminated union of three shape variables

Evaluation of Unions

•Free unions are unsafe

–Do not allow type checking

•Java and C# do not support unions

–Reflective of growing concerns for safety in programming language •Ada‘s discriminated unions are safe.

References:

[1]. Swierstra, Wouter. "Data types à la carte." *Journal of functional programming* 18.4 (2008): 423-436.
[2]. Guttag, John V., and James J. Horning. "The algebraic specification of abstract data types." *Acta informatica* 10.1 (1978): 27-52.
[3]. Pavlidis, Paul, et al. "Learning gene functional classifications from multiple data types." *Journal of computational biology* 9.2 (2002): 401-411.
[4]. Bruce, Kim B. "Safe type checking in a statically-typed object-oriented programming language." *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1993.
[5]. Cardelli, Luca. "Basic polymorphic typechecking." *Science of computer programming* 8.2 (1987): 147-172.
[6]. Mössenböck, Hanspeter, and Niklaus Wirth. "The programming language Oberon-2." *Structured Programming* 12.4 (1991): 179-196.
[7]. Igarashi, Atsushi, and Hideshi Nagira. "Union types for object-oriented programming." *Proceedings of the 2006 ACM symposium on Applied computing*. 2006.
[8]. Griesemer, Robert. *A Programming Language for Vector Computers*. Eidgenössische Technische Hochschule [ETH] Zürich, 1993.