

A list is simply a sequence of values, with a sense of position. That is, there is a definite first element and a definite last element, and each element except the first has a unique preceding element (predecessor) and each element except the last has a unique succeeding element (successor).

It is very common for a problem to require storing and manipulating lists of data values.

For example, we may have a list of temperatures from a sensor, a list of names of friends, a list of prices from an order database.

We may also have lists whose elements are complex entities, such as a list of books, where a book consists of a call number, a title, author name(s), year of publication, etc.

For now we will consider only lists of simple values.

We may manipulate lists in many ways. We may insert new elements and delete old ones. We may search the list for a particular element. We may sort the list into some particular order. We may replace an element with a new value.

The need to store lists leads to the notion of a data structure, which is simply a collection of components which may be viewed as a whole but whose individual parts are individually accessible in some manner.

C++ provides support for a number of structured types, including:

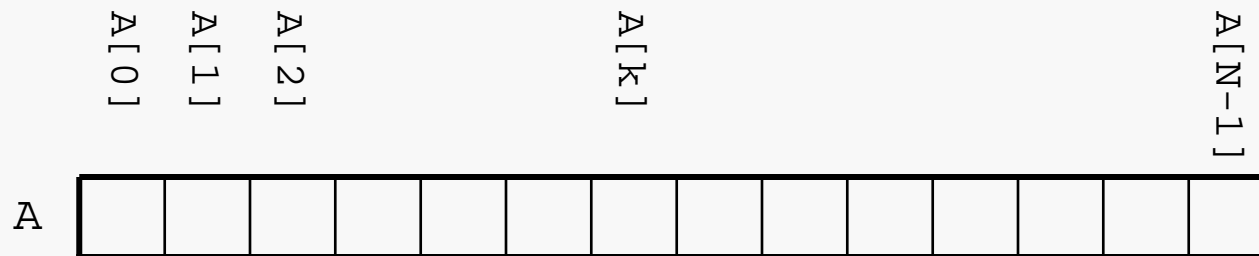
- array
- a sequential list of data values that are all of the same type (homogeneous)
 - individual elements are accessed by their position (1st, 3rd, 72nd, etc.)
 - has a fixed capacity, called the dimension, and can only hold up to that many elements at once
 - fundamental C++ mechanism for storing lists of data
- structure
- a collection of data values that may be of differing types (heterogeneous)
 - studied in a later chapter

In mathematics, a list of values is often denoted by using a name (for the list) and attaching a subscript to indicate the particular position being discussed.

For example we might have a 3-dimensional vector X , in which the three elements of X could be denoted by X_0 , X_1 and X_2 .

Since subscripts aren't supported in most text editors, C++ uses a slightly different notation to represent the same idea. If Z is a 3-dimensional array in C++ then the three elements of Z are denoted by $Z[0]$, $Z[1]$ and $Z[2]$.

In C++ the positions in an array are always numbered sequentially, starting with zero:



Here we have an array named A with dimension N . Notice that the cells (positions) are numbered 0 through $N-1$.

An array is a variable. An array has parts and it is structured, but it is still just a variable.

Every C++ array has a name, and as always that identifier must be declared before it may be used.

The declaration of an array must specify the type of element the array stores and the dimension (number of cells) the array will have.

The elements may be of any type at all, but the dimension must be a positive integer constant or an expression that evaluates to an integer constant.

```
const int BUFFERSIZE = 256;
const int DICESUMS   = 11;

char    Buffer[BUFFERSIZE];           // constant integer dimension
int     DiceFreq[DICESUMS + 1];     // constant integer expression,
                                     //      used as dimension
int     numItems = 10000;            // integer variable
string  Inventory[numItems];        // NOT valid - numItems is not
                                     //      a constant
```

We refer to the individual positions in an array as cells or elements. Each cell is named by giving the name of the array, followed by an integer indicating a particular position within the array; the integer is enclosed in square brackets. The integer is often called the index of the element.

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]	A[12]	A[13]
A	13	7	43	5	3	19	2	23	29	31	11	17	47	41

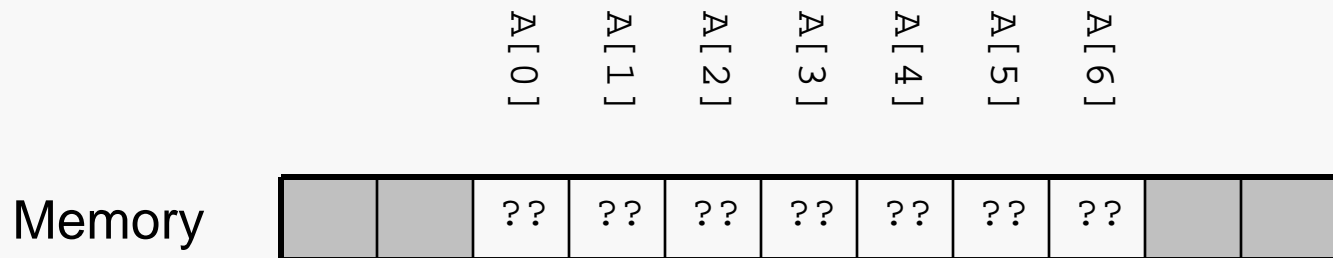
Here: `A[3] == 5` and `A[7] == 23`.

The individual elements of an array can be used in any way that a simple variable of that type may be used. So we could write:

```
A[3] = 51;           // stores the value 51 in cell #3 of A
if (A[0] < A[5]) {  // compares element #0 and element #5
    . . .
}
```

If we have the declaration: `char A[7];`

then at runtime memory will be allocated for the array variable A. Since A has 7 cells, each of type char, and a char is stored in one byte of memory, A will require 7 bytes of memory. These will be allocated contiguously (as a single chunk) and the cells will then be stored in the order:



Logically, the valid index values range from 0 to 6 (the dimension minus 1).

As with any variable, declaring an array does not cause any automatic initialization of the memory allocated for it. (That's why the cells above are filled with question marks.)

One of the Deadly Sins of Programming is failing to initialize variables before they are used. That is especially important with array variables. We will return to this issue shortly.

An array may be initialized with a simple for loop:

The for loop counter, `Idx`, is used as the array index within the loop body.

`Idx` is started at 0 and increased by 1 on each pass.

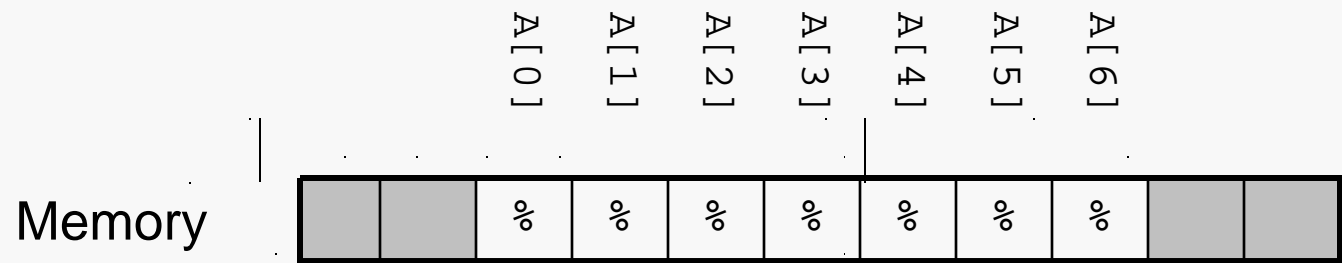
The loop terminates when `Idx` equals the dimension of the array.

```
const int SIZE = 7;
char A[SIZE];

int Idx;
for (Idx = 0; Idx < SIZE; Idx++) {
    A[Idx] = '%';
}
```

The loop design guarantees that each cell of the array will be accessed, in turn, and that the loop will stop before an invalid array index is reached.

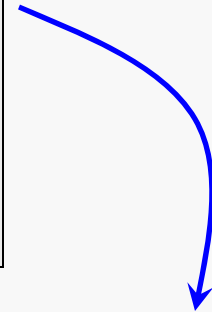
This is a standard pattern for array processing. Be sure you understand it.



Usually only some of the cells of an array are actually storing meaningful data:

```
. . .  
const int SIZE = 10;  
string Name[SIZE];  
  
int Idx;  
for (Idx = 0; Idx < SIZE; Idx++) {  
    Name[Idx] = "Unused";  
}  
  
Idx = 0;  
string Temp;  
getline(In, Temp);  
while ( In && Idx < SIZE ) {  
    Name[Idx] = Temp;  
    Idx++;  
    getline(In, Temp);  
}
```

Socrates
Plato
Aristotle
Epimenides
Epictetis
Parmenides
Zeno



0	Socrates
1	Plato
2	Aristotle
3	Epimenides
4	Epictetis
5	Parmenides
6	Zeno
7	Unused
8	Unused
9	Unused

The number of cells of an array that store meaningful data is called the usage.

Here is an example showing array access logic:

```
const int MAXSTUDENTS = 100;
int Test[MAXSTUDENTS];
int numStudents = 0;
. . .
// code that reads and counts # of data values given
. . .
int maxScore = INT_MIN;
int idxOfMax = -1;
int Idx;
for (Idx = 0; Idx < numStudents; Idx++) {
    if ( maxScore < Test[Idx] ) {
        maxScore = Test[Idx];
        idxOfMax = Idx;
    }
}
if (idxOfMax >= 0)
    cout << "Student " << idxOfMax << " achieved the highest"
        << " score: " << maxScore;
```

A single array element can be passed as an actual parameter to a function.

```
const int SIZE = 100;
int X[SIZE];
. . .
// read data into X[]
. . .
swapInts(X[13], X[42]);
```

This function uses simple `int` (reference) parameters. It neither knows nor cares that the actual parameters are elements of an array.

```
void swapInts(int& First, int& Second) {

    int Temp = First;
    First = Second;
    Second = Temp;

}
```

A single array element can be treated just as any simple variable of its type. So the receiving function treats the parameter just as a simple variable.

Note well: when passing a single array element, the actual parameter is the array name, **with an index attached**.

In other words, `X` refers to the entire array while `X[k]` refers to the single element at index `k`.