Lists of Data

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.

©1995-2001 Barnette ND & McQuain WD

Structured Types

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

©1995-2001 Barnette ND & McQuain WD

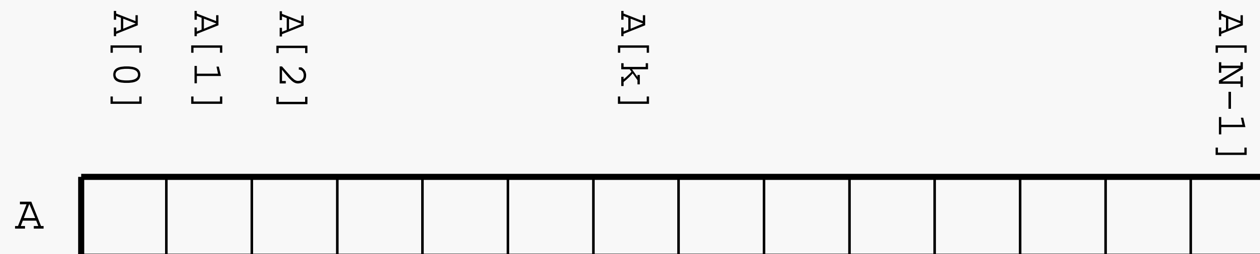
Terminology and Notation

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.

©1995-2001 Barnette ND & McQuain WD

Declaring an Array Variable

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.

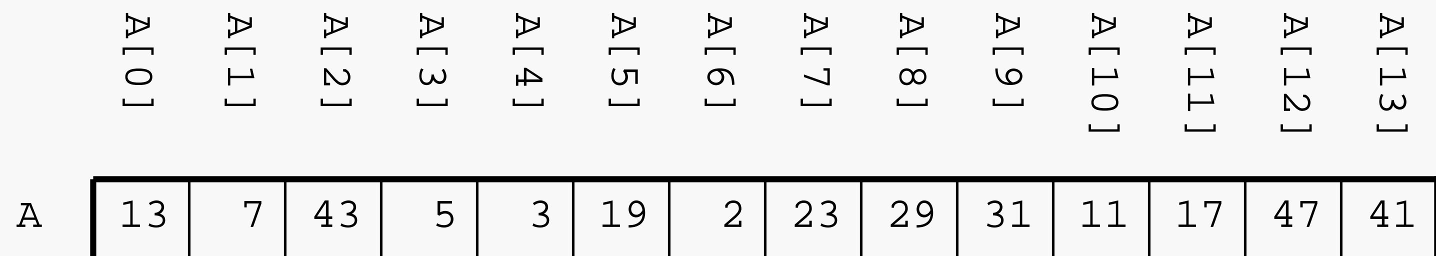
```
const int BUFFERSIZE = 256;
const int DICESUMS   =  11;

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
```


©1995-2001 Barnette ND & McQuain WD

Accessing the Individual Cells of an Array

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.



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

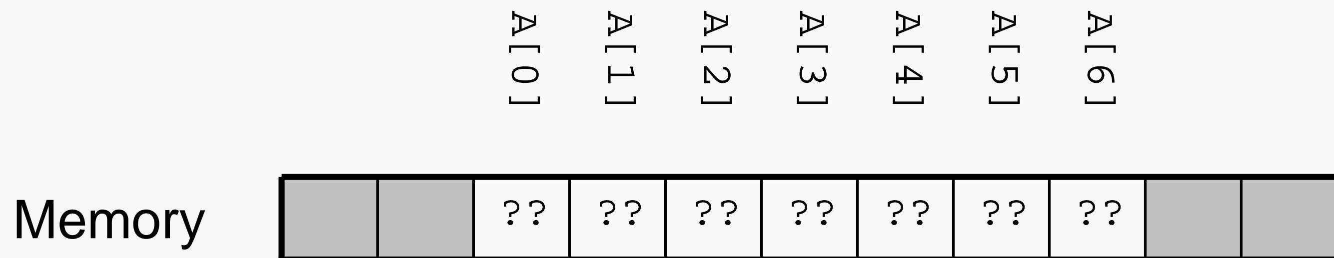
The individual elements of an array can be used in any way that a simple variable of that type may be used. So we could write:

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

Array Declarations and Memory

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.

 ©1995-2001 Barnette ND & McQuain WD

Initializing an Array

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.

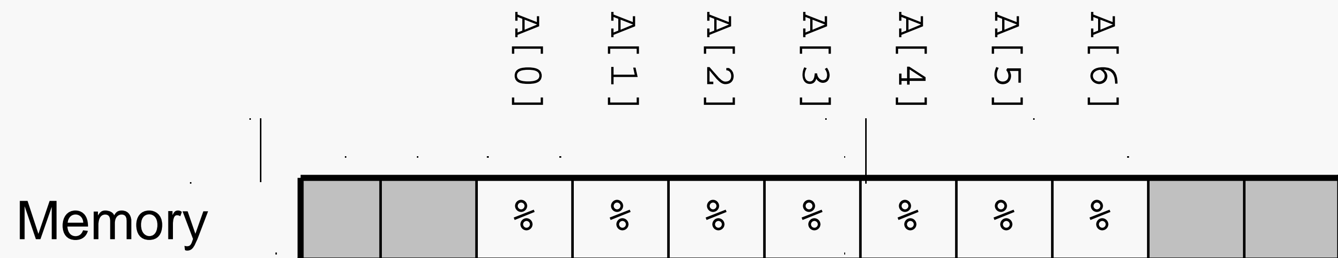
The loop terminates when `Idx` equals the dimension of the array.

```
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.



©1995-2001 Barnette ND & McQuain WD

The Usage of an Array

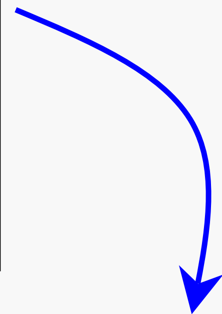
Usually only some of the cells of an array are actually storing meaningful data:

```
. . .
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++;
}
```

```
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.

©1995-2001 Barnette ND & McQuain WD

More Array Access Examples

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;
```


©1995-2001 Barnette ND & McQuain WD

Array Elements as Parameters

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, Second& a2) {

   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`.

 ©1995-2001 Barnette ND & McQuain WD

Aggregate Operations

An aggregate operation is an operation that is performed on a data structure, such as an array, as a whole rather than performed on an individual element.

Assume the following declarations:

```
const int SIZE = 100;
int X[SIZE];
int Y[SIZE];
```

Assuming that both X and Y have been initialized, consider the following statements and expressions involving aggregate operations:

```
X = Y;          // _____ assigning one array to another

X == Y;         // _____ comparing two arrays with a relational operator

cout << X;      // _____ inserting an array to a stream

X + Y           // _____ adding two numeric arrays

return X;       // _____ using an array as the return value from a function

Foo(X);         // _____ passing an entire array as a parameter
```

Of course, the operations that are not supported by default may still be implemented via user-defined functions.

©1995-2001 Barnette ND & McQuain WD

Implementing Aggregate Operations

The operations discussed on the previous slide that are not supported automatically may still be implemented by user-defined code. For example, an equality test could be written as:

```
bool areEqual = true;
int Idx;
for (Idx = 0; ( Idx < Size && areEqual ); Idx ++) {
    if ( X[Idx] != Y[Idx] )
       areEqual = false;
}
```

As we noted earlier, the most common way to process an array is to write a `for` loop and use the loop counter as an index into the array; as the loop iterates, the index "walks" down the array, allowing you to process the array elements in sequence.

Note the use of the Boolean variable in the `for` loop header. This causes the `for` loop to exit as soon as two unequal elements have been found, improving efficiency at run-time.

©1995-2001 Barnette ND & McQuain WD

Entire Arrays as Parameters

Entire arrays can also be passed as function parameters.

Note well that when passing an entire array to a function:

- the actual parameter is the array name, without an index.
- the formal parameter is an identifier, followed by an empty pair of brackets.

The function to which the array is passed has no way of knowing either the dimension or the usage of the array unless they are global or are also passed to the function. It's often an indication of a logical error if an array is passed to a function but it's usage is not.

By default, if an array name is used as a parameter, the array is passed-by-reference.

Preceding a formal array parameter by the keyword `const` results in the array being passed by constant reference, the effect being that the actual array parameter cannot be modified by the called function.

©1995-2001 Barnette ND & McQuain WD

Array Parameters Example

Here is an illustration of passing an array parameter to a function:

```
// Elements of A and B with subscripts ranging from 0 to Size - 1
// are summed element by element, results being stored in C.
//
// Pre:  A[i] and B[i] (0 <= i <= Size -1) are defined
// Post: C[i] = A[i] + B[i] (0 <= i <= Size - 1)
void addArray(int Size, const int A[], const int B[], int C[]) {

   int Idx;
   for (Idx = 0; Idx < Size; Idx++)
        C[Idx] = A[Idx] + B[Idx];       // add and store result
}
```

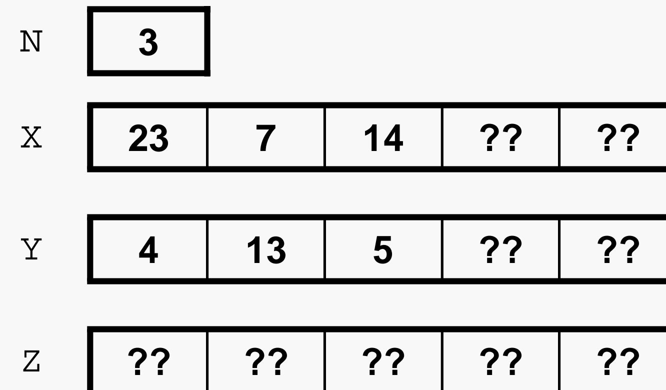
Invoke `addArray()` as follows:

```
const int SIZE = 5;
int X[SIZE], Y[SIZE], Z[SIZE];
// store some data into X and Y
addArray(N, X, Y, Z);
```

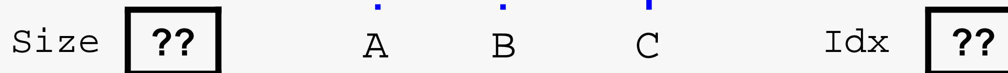

©1995-2001 Barnette ND & McQuain WD

Memory Use and Trace

Calling function memory area:



`addArray` memory area:



Note that the fact that the arrays are passed by reference reduces the total memory usage by the program considerably.

©1995-2001 Barnette ND & McQuain WD

Aggregate Operation as a Function

Here is an illustration of implementing an aggregate copy operation for arrays:

```
// The elements of Source at indices 0 through Size - 1 are copied
// to the corresponding locations of Target.
// Parameters:
//   Target   array of dimension >= Size
//   Source   array storing at least Size values
//   Size     # of elements to be copied from Source to Target
//
// Pre:  Source[i] (0 <= i <= Size -1) are defined
// Post: Target[i] == Source[i] (0 <= i <= Size - 1)
void Copy(int Target[], const int Source[], int Size) {

   int Idx;
   for (Idx = 0; Idx < Size; Idx++)
        Target[Idx] = Source[Idx];        // copy cells
}
```

Should the third actual parameter to this function be the dimension or the usage of the array `Source`?

 ©1995-2001 Barnette ND & McQuain WD

Cautionary Note

The equality test code presented earlier can be incorporated into a function, as shown below. Note that there is no way for the function to determine whether the array indices are within logically correct bounds.

If the function is passed a value for `NumCells` that is larger than the dimension of either of the actual array parameters, then memory locations that exist outside the array boundaries will be compared. This type of logical error is extremely difficult to debug.

```
bool areEqual(const int A[], const int B[], int Usage) {

  for (int Index = 0; Index < Usage; Index++) {
      if (A[Index] != B[Index])
         return false;           // exit if mismatch is found
  }
  return true;
}
```

This represents the classic source of programming errors when arrays are used.

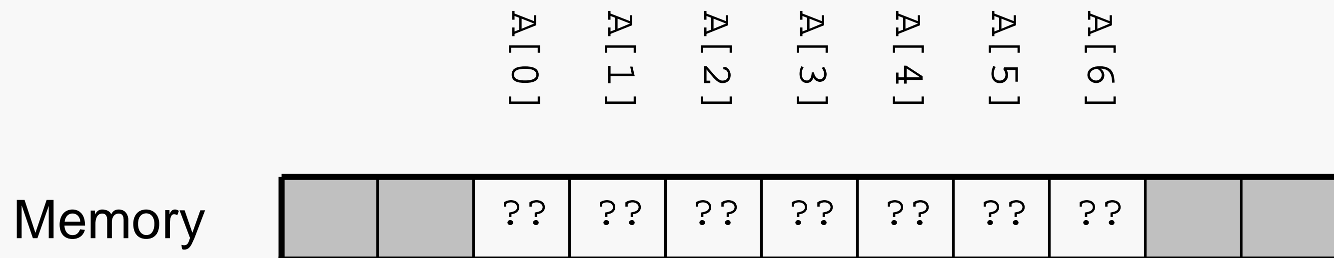
The effects of out-of-bounds index values range from incorrect results to subtle crashes to spectacular crashes.

 ©1995-2001 Barnette ND & McQuain WD

Array Indices and Memory

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).

The memory locations before A[0] and after A[6] are NOT allocated for the array, in fact there is no reason to believe they are even allocated to the program containing the array declaration. Consider the following statement:

```
A[7] = 42;
```


©1995-2001 Barnette ND & McQuain WD

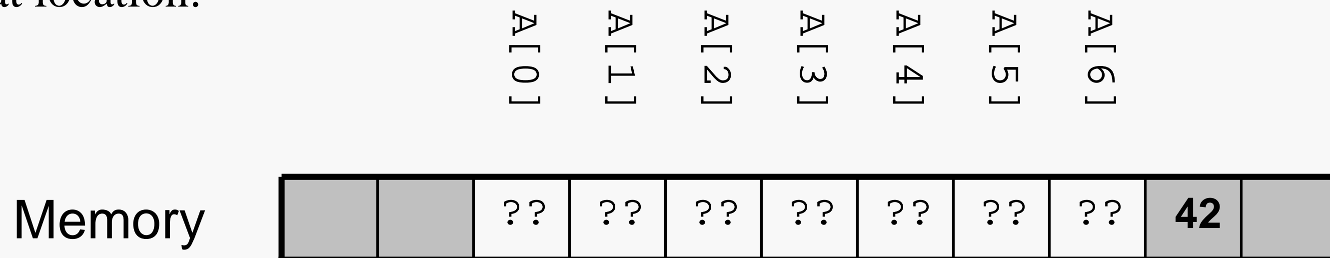
Out-of-Bounds Array Indices

What happens when a statement uses an array index that is out of bounds?

First, there is no automatic checking of array index values at run-time (some languages do provide for this). Consider the C++ code from the previous slide:

```
int A[7];
A[7] = 42;
```

Logically `A[7]` does not exist. Physically `A[7]` refers to the `int`-sized chunk of memory immediately after `A[6]`. The effect of the assignment statement will be to store the value 42 at that location:



Clearly this is undesirable. What actually happens as a result depends upon what this location is being used for…

©1995-2001 Barnette ND & McQuain WD

Memory Access Errors

Consider the possibilities. The memory location `A[7]` may:

- store a variable declared in your program
- store an instruction that is part of your program (unlikely on modern machines)
- not be allocated for the use of your program

In the first case, the error shown on the previous slide would cause the value of that variable to be altered. Since there is no statement that directly assigns a value to that variable, this effect seems very mysterious when debugging.

In the second case, if the altered instruction is ever executed it will have been replaced by a nonsense instruction code. This will (if you are lucky) result in the system killing your program for attempting to execute an illegal instruction.

In the third case, the result depends on the operating system you are using. Some operating systems, such as Windows 95/98 do not carefully monitor memory accesses and so your program may corrupt a value that actually belongs to another program (or even the operating system itself). Other operating systems, such as Windows NT/2000 or UNIX, will detect that a memory access violation has been attempted and suspend or kill your program.

©1995-2001 Barnette ND & McQuain WD

Using `assert()` for Safety

So what can a programmer do to help detect array indices that are out-of-bounds?

We may "guard" each array access with logic that will trigger diagnostic output if an out-of-bounds index is found?

One way to do this is to use the C language `assert()` function.

If the dimensions of A and B are global constants, we may modify the equality test function as follows:

```
bool areEqual(const int A[], const int B[], int Usage) {
   assert(Usage <= A_DIM && Usage <= B_DIM);

   for (int Index = 0; Index < Usage; Index++) {
      if (A[Index] != B[Index])
         return false;           // exit if mismatch is found
   }
   return true;
}
```

`assert()` takes a Boolean expression as its only parameter. If the Boolean expression is false when the `assert()` call is reached, the program will be automatically terminated.

©1995-2001 Barnette ND & McQuain WD

`assert()` Issues

The prototype for `assert()` is in the header file `<cassert>`.

`assert()` calls are only enabled in debug builds of a program. While that is the default when using the Visual C++ IDE (and some others), there are good reasons to avoid a debug build when packaging a program for release. In such a case, the `assert()` calls will be disabled.

`assert()` calls also only produce limited diagnostic information. There may be some output to the OS command shell, including some information about the location of the `assert()` call that was triggered, such as the line number or the specific function involved, but no more.

While `assert()` is useful for reporting such errors, there are better approaches.

In a more advanced course, we cover the use of the C++ *exception* mechanism for this purpose.

At this level, we will cover the alternative of adding diagnostic tests and output to the existing code.

©1995-2001 Barnette ND & McQuain WD

Instrumenting Code

We may also "instrument" code to provide more precise and useful diagnostic output when errors are detected.

Consider:

```
bool areEqual(const int A[], const int B[], int Usage) {
   if ( Usage <= A_DIM ) (
      cout << "Parameter error in areEqual" << endl;
      cout << "Value of Usage exceeds dimension of A." << endl;
      exit(1);
   }
   else if ( Usage <= B_DIM ) {
      cout << "Parameter error in areEqual" << endl;
      cout << "Value of Usage exceeds dimension of B." << endl;
      exit(1);
   }
   . . .
}
```

This will report precisely where and what went wrong, and then abort the program.

Also, this is not disabled in non-debug builds.

©1995-2001 Barnette ND & McQuain WD

Organizing Related Data

Consider the problem of organizing and manipulating the following data:

Origin	Destination	Miles	Time
Blacksburg, VA	Knoxville, TN	244	3:25
Knoxville, TN	Nashville, TN	178	2:35
Nashville, TN	Memphis, TN	210	3:17
Memphis, TN	Little Rock, AR	137	2:05
Little Rock, AR	Texarkana, TX	141	2:10

Suppose that we need to calculate some values for each trip, such as the average speed. Also suppose that we need to be able to look up all trips with a given origin and report the data for those trips. How can we accomplish this?

Clearly we must read in all of the trip data and store it in some way; we cannot just process this data line by line because reading data from a file on disk is too slow for a useful application.

However, we cannot store all the data in a single array since an array can hold data of only one type, and we must deal with strings and integers and decimal numbers.

The solution does involve using arrays… note that the data is naturally organized as a table, where each row represents a particular trip and all the values in each column are of the same type.

©1995-2001 Barnette ND & McQuain WD

Parallel Arrays

We may organize a table of related data of differing types by using a collection of arrays, thinking of each array as representing a column of the table.

The values from each row of the table would be stored across the various arrays, but at the same index value in each.

By declaring appropriate arrays, we may read and store the given data as shown below, and then calculate the speed for each trip and store those values:

```
const int MAXTRIPS = 500;
string Origin[MAXTRIPS];
string Destination[MAXTRIPS];
int    Miles[MAXTRIPS];
int    Minutes[MAXTRIPS];
double MPH[MAXTRIPS];
```

	Origin	Destination	Miles	Minutes	MPH
0	Blacksburg, VA	Knoxville, TN	244	205	??
1	Knoxville, TN	Nashville, TN	178	155	??
2	Nashville, TN	Memphis, TN	210	197	??
3	. . .	. . .	. . .	. . .	...
4	Little Rock, AR	Texarkana, AR	141	160	??
	. . .	. . .	. . .	. . .	...

©1995-2001 Barnette ND & McQuain WD

Processing with Parallel Arrays

The only special issue when using parallel arrays is to be careful to always access each array at the same index with storing or retrieving values, in order to reference corresponding data locations:

```
void printTrips(ofstream& Out, const int numTrips, const string Origin[],
                const string Destination[], const int Miles[],
                const int Minutes[], const double MPH[]) {

   int Idx;
   for (Idx = 0; Idx < numTrips; Idx++) {
      Out << left << setw(MAXNAMELENGTH + 1) << Origin[Idx]
          << setw(MAXNAMELENGTH + 1) << Destination[Idx]
          << right << setw(10) << Miles[Idx]
          << setw(10) << (Minutes[Idx] / MINSPERHOUR) << ':'
          << setfill('0') << setw(2) << (Minutes[Idx] % MINSPERHOUR)
          << setfill(' ') << setw(10) << setprecision(1)
          << MPH[Idx]
          << endl;
   }
}
```


©1995-2001 Barnette ND & McQuain WD

Multidimensional Arrays

C++ also allows an array to have more than one dimension.

For example, a two-dimensional array consists of a certain number of rows and columns:

```
const int NUMROWS = 3;
const int NUMCOLS = 7;
int Array[NUMROWS][NUMCOLS];
```

	0	1	2	3	4	5	6
0	4	18	9	3	-4	6	0
1	12	45	74	15	0	98	0
2	84	87	75	67	81	85	79

`Array[2][5]` 3rd value in 6th column

`Array[0][4]` 1st value in 5th column

The declaration must specify the number of rows and the number of columns, and both must be constants.

©1995-2001 Barnette ND & McQuain WD

Processing a 2-D Array

A one-dimensional array is usually processed via a for loop. Similarly, a two-dimensional array may be processed with a nested for loop:

```
for (int Row = 0; Row < NUMROWS; Row++) {
    for (int Col = 0; Col < NUMCOLS; Col++) {
        Array[Row][Col] = 0;
    }
}
```

Each pass through the inner for loop will initialize all the elements of the current row to 0.

The outer for loop drives the inner loop to process each of the array's rows.

©1995-2001 Barnette ND & McQuain WD

Initializing in Declarations

```
int Array1[2][3]  = { {1, 2, 3} , {4, 5, 6} };
int Array2[2][3]  = { 1, 2, 3, 4, 5 };
int Array3[2][3]  = { {1, 2} , {4 } };
```

If we printed these arrays by rows, we would find the following initializations had taken place:

Rows of Array1:

```
1 2 3
4 5 6
```

Rows of Array2:

```
1 2 3
4 5 0
```

Rows of Array3:

```
1 2 0
4 0 0
```

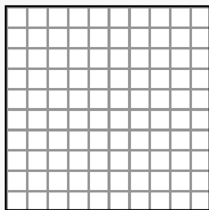
```
for (int row = 0; row < 2; row++) {
    for (int col = 0; col < 3; col++) {
        cout << setw(3)
             << Array1[row][col];
    }
    cout << endl;
}
```


©1995-2001 Barnette ND & McQuain WD

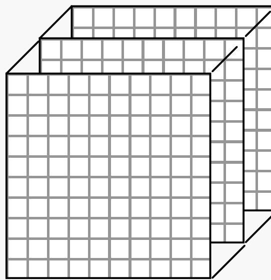
Higher-Dimensional Arrays

An array can be declared with multiple dimensions.

2 Dimensional

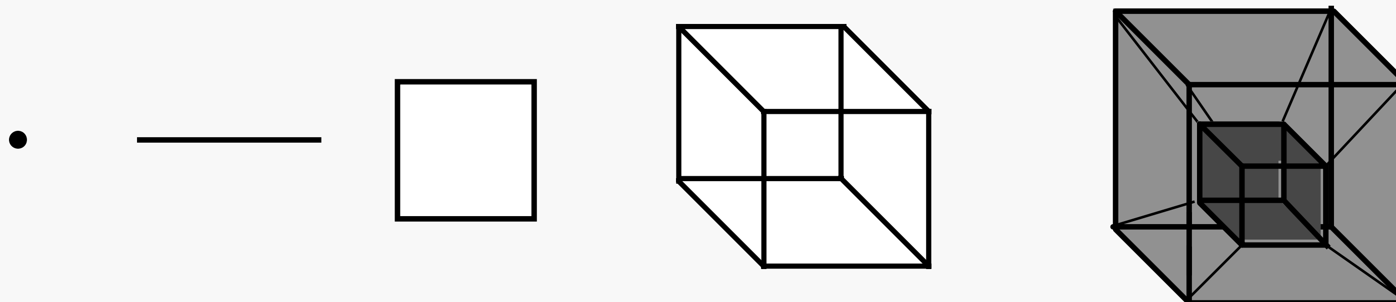


3 Dimensional



```
double Coord[100][100][100];
```

Multiple dimensions get difficult to visualize graphically.



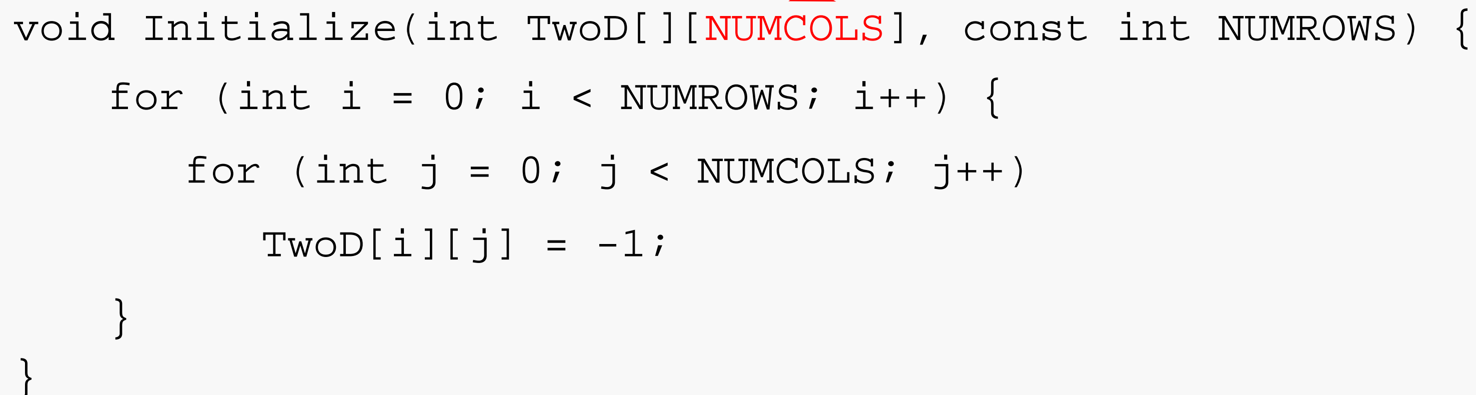
©1995-2001 Barnette ND & McQuain WD

2-D Arrays as Parameters

When passing a two-dimensional array as a parameter, the base address is passed, as is the case with one-dimensional arrays.

But now the number of columns in the array parameter must be specified. This is because arrays are stored in row-major order, and the number of columns must be known in order to calculate the location at which each row begins in memory:

address of element (r, c) = base address of array
+ r*(number of elements in a row)*(size of an element)
+ c*(size of an element)



```
void Initialize(int TwoD[][NUMCOLS], const int NUMROWS) {
    for (int i = 0; i < NUMROWS; i++) {
        for (int j = 0; j < NUMCOLS; j++)
            TwoD[i][j] = -1;
    }
}
```


©1995-2001 Barnette ND & McQuain WD