Declaration of Statically-Allocated Arrays

In C, an array is simply a fixed-sized aggregation of a list of cells, each of which can hold a single values (objects).

The number of cells in an array is called its *dimension*.

The number of values that are actually stored in an array is called its *usage*.

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

double X[1000];                // literal constant dimension
char   Buffer[BUFFERSIZE];     // define'd constant dimension
int    DiceFreq[DICESUMS + 1]; // constant integer expression,
                               //     used as dimension
int    numItems = 10000;       // integer variable
int    List[numItems];         // NOT valid - numItems is not
                               //     a constant
```

The dimension must* be a constant expression (known at compile-time).

The dimension and usage are separate values, with no association as far as the language is concerned with the array itself.

***but see VLAs**

Limitations

There is no way to alter the dimension of an array once it is declared.

Access to individual cells uses the same syntax as Java; however, there is no run-time check to be sure that the specified index is actually valid.

There are no automatic aggregate operations for arrays in C.

- = does not copy the contents one array into another
- == is not supported for arrays; at least not the way you'd like…
- arrays cannot be passed by value to a function (although array names can)

When an array is passed to a function, its dimension and/or usage must generally be passed as well... otherwise the function will have to way to determine where the array ends.

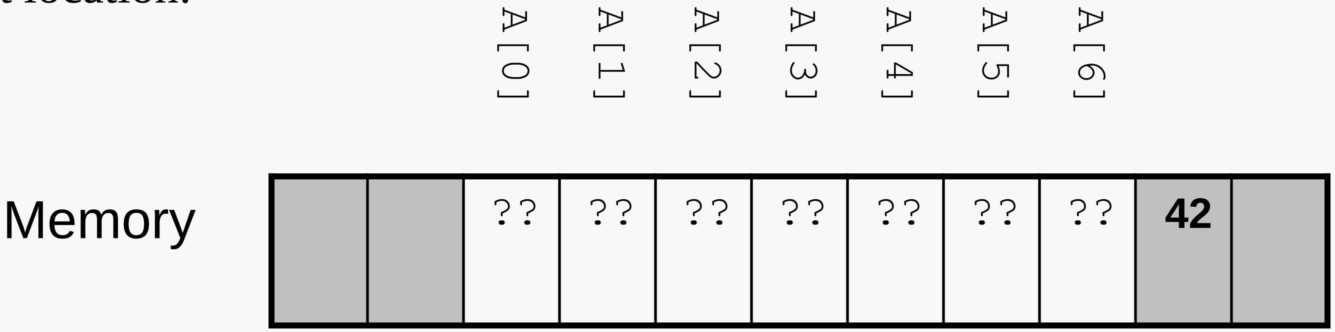
Out-of-Bounds Array Indices

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:

```
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…

A Warning

You may see examples like this that purport to show a way to determine the dimension of an array:

```
void f(int A[]) {

   int dimension = sizeof(A) / sizeof(A[0]);
   ...
}
```

Be aware that **this does not work** if applied to an array passed to a function, or an array that's allocated dynamically. Try it.

The `sizeof()` trick works if used in the same scope as the declaration of the array, in which case it is hardly needed.

There are many fairly stupid discussions of this available online, and even in some textbooks.

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/Me 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/XP or UNIX, will detect that a memory access violation has been attempted and suspend or kill your program.

Array Initialization

As with all variables in C, array cells are not automatically initialized when an array is created:

```c
int Primes[5];                     // Primes[0:4] are unknown

int Evens[5] = {0, 2, 4, 6, 8};   // Evens[0:4] are known

int Odds[5] = {1, 3, 5};          // Odds[0:2] are as shown;
                                   //  rest are 0!

int Zeros[10000] = {0};           // Zeros[0:9999] are all 0

int Bads[5] = {1, 3, 5, 7, 9, 11}; // too many initializers!
```

Of course, for arrays of interesting sizes you'll usually initialize via a loop…

Arrays as Parameters

Arrays may be passed as parameters in function calls, and the effect is pass-by-reference:

```
#define SZ 5
void fillPrimes( int Primes[] );

int main() {

   int Primes[SZ];

   fillPrimes(Primes);

   for (int i = 0; i < SZ; i++) {
      printf("%3d:%5d\n", i, Primes[i]);
   }

   return 0;
}

void fillPrimes( int Primes[] ) {

   for (int i = 0; i < SZ; i++) {
      Primes[i] = i * i + i + 41;
   }
}
```

Note parameter declaration syntax.

Pass array to fn by name.

Fn can modify array passed to it by the caller…

Note idiomatic, but perhaps questionable use of #define here.

Example: Some Array Manipulations

```
#include <stdio.h>
#include <time.h>                         // for time()
#include <stdlib.h>                       // for srand(), rand()
#define SZ 10                             // constant for array dimension

void fillArray( int List[], unsigned int Sz);    # fn declarations
void Sort(int List[] , unsigned int Usage);

int main() {

   int A[SZ];                             // allocate space for array

   fillArray(A, SZ);                      // fill array with random values

   for (int i = 0; i < SZ; i++) {         // print original array
      printf("%3d:%5d\n", i, A[i]);
   }

   Sort(A, SZ);                           // sort the array

   for (int i = 0; i < SZ; i++) {         // print the sorted array
      printf("%3d:%5d\n", i, A[i]);
   }
   return 0;
}
                                          // more to come . . .
```

Example: Initializing an Array

```
// Writes Size random integer values in [0, 1000) into List[]
//
// Pre:
//        List[] has dimension >= Size
// Post:
//        List[0:Size-1] have been set to random values in
//        the range 0 to 999, inclusive.
//
void fillArray(int List[], unsigned int Size) {

   srand( (unsigned int) time(NULL) );      // initialize random
                                            //    generator

   for (int pos = 0; pos < Size; pos++) {

      List[pos] = rand() % 1000;
   }
}
                                            // more to come . . .
```

Example: Sorting an Array

```c
// Uses insertionsort algorithm to put elements of List[] into
// ascending order.
//
void Sort(int List[], unsigned int Usage) {

   int unsortedFront = 1;

   while ( unsortedFront < Usage ) {

      int currElement = List[unsortedFront];
      int probeLocation = unsortedFront;

      while ( probeLocation > 0   &&
              List[probeLocation-1] > currElement) {

         List[probeLocation] = List[probeLocation-1];
         probeLocation--;
      }

      List[probeLocation] = currElement;
      unsortedFront++;
   }
}
```

Example: Squeezing out Odd Values

Problem: take an array of integer values and eliminate all the odd values from the array, leaving only the even values, and keeping them in their original relative order, but leaving no "gaps" within the array

For example, we would transform the first array below into the second:

484	**501**	**122**	**777**	**29**	**24**	**543**	**204**

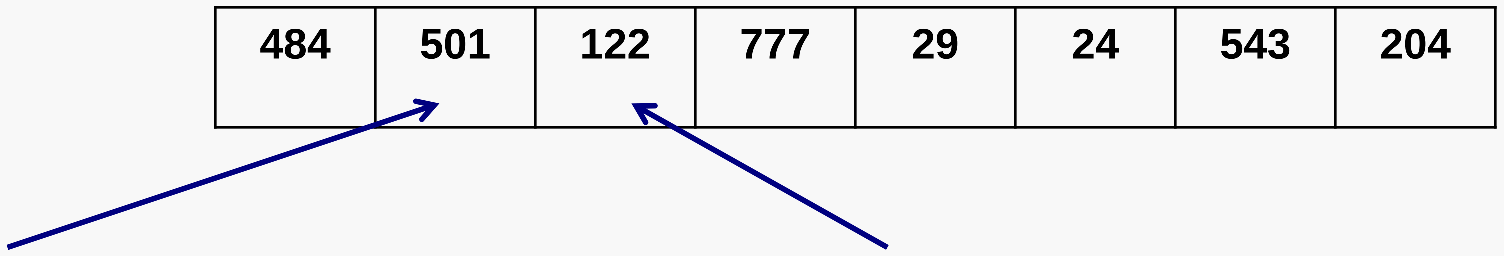
484	**122**	**24**	**204**				

Obviously, the algorithm must report the number of elements in the modified array, since that will likely be smaller than the number of elements in the original array.

We also do not want to make use of a second array; that would waste memory and entail too much extra copying of data.

Example: Squeezing out Odd Values

Here's one approach.



Move `Trailer` to the first odd value.

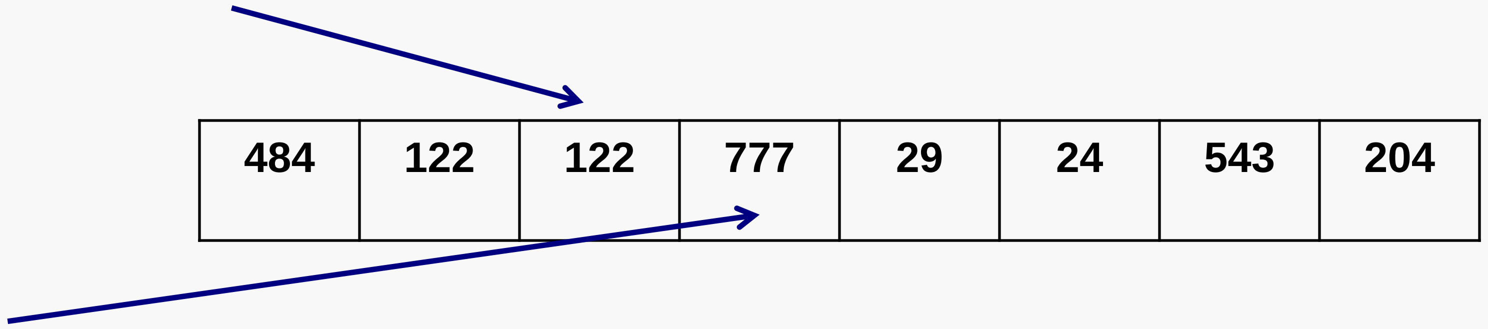
If there isn't one, we are done.

Set `Leader` to the first value after `Trailer`.

If `Leader` points to an even value,

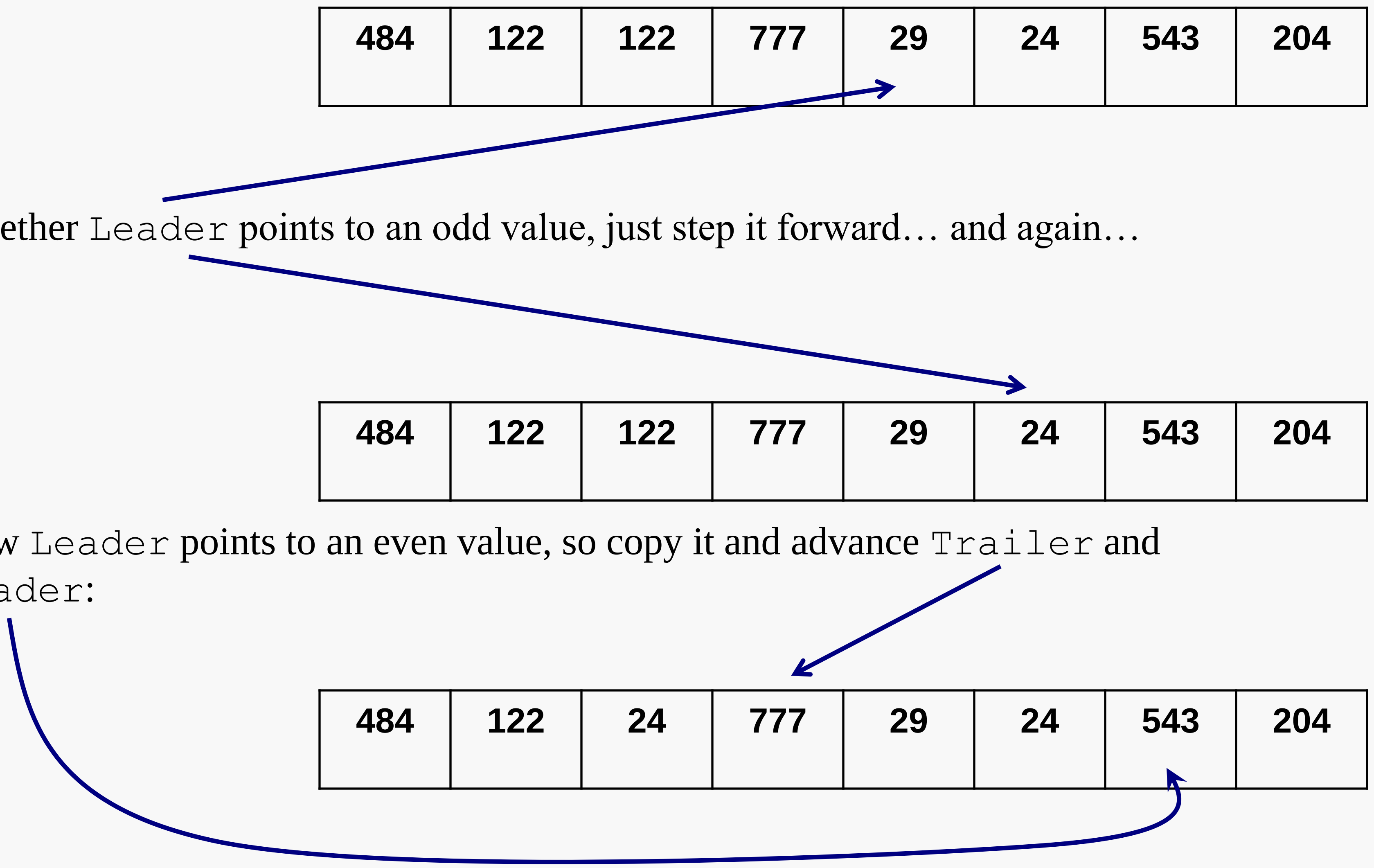
copy that value to `Trailer`'s location

advance `Trailer`



Whether `Leader` points to an even value or not, step Leader ahead one spot.

Example: Squeezing out Odd Values



Whether `Leader` points to an odd value, just step it forward… and again…

Now `Leader` points to an even value, so copy it and advance `Trailer` and `Leader`:

Example: Squeezing out Odd Values

```
// Takes an array of integers and removes all the odd ones,
// without altering the order of any of the even values.
//
// Pre:
//      List[] has dimension >= Usage
// Post:
//      All the even values are listed in successive cells at the
//      beginning of List[], in their original relative order.
// Returns:
//      the number of even values in the list
//
unsigned int squeezeOutOdds(int List[], unsigned int Usage) {

   unsigned int Trailer = 0;
   // Move Trailer to first odd value, if any.
   while ( Trailer < Usage && List[Trailer] % 2 == 0 )
      ++Trailer;
   // Check for case there are no odd values in List[]
   if ( Trailer == Usage )
      return Trailer;
   // . . .
```

Example: Squeezing out Odd Values

```
    unsigned int Leader = Trailer + 1;

    // Walk Leader to end of List[]
    while ( Leader < Usage ) {

       // If Leader is at an even value, move it forward;
       //    advance Trailer
       if ( List[Leader] % 2 == 0 ) {
          List[Trailer] = List[Leader];
          ++Trailer;
       }

       // Always advance Leader
       ++Leader;
    }

    // When done, Trailer is one past the final even value,
    // so it equals the number of even values that are left.
    return Trailer;
}
```

Variable-length Arrays

In C99, it is possible to declare an array whose dimension is not known at compile time:

```
#include <stdio.h>
void F(int n, int A[n]);

int main() {

   int n;
   printf("Enter the size of the desired arrays: ");
   scanf("%d", &n);

   int A[n];

   F(n, A);

   return 0;
}
. . .
```

dimension is not known until run-time

Variable-length Arrays

When passing a variable-sized array as a parameter, the (variable) dimension and the array declaration are associated syntactically:

```
. . .
void F(int n, int A[n]) {

   int B[n];
   for (int i = 0; i < n; i++) {
      A[i] = i * i;
      B[i] = i * i * i;
   }

   for (int j = 0; j < n; j++) {
      printf("%3d:  %5d  %5d\n", j, A[j], B[j]);
   }
}
```

space for B[ ] is allocated when the declaration is reached (at runtime) and deallocated when the surrounding block is exited

This isn't nearly as cool as it may look… although it does automate the deallocation of the array.