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.

©1995-2009 Barnette ND, McQuain WD & Kennan MA

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.

©1995-2009 Barnette ND, McQuain WD & Kennan MA

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 (poor practice), 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.

©1995-2009 Barnette ND, McQuain WD & Kennan MA

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
}
```

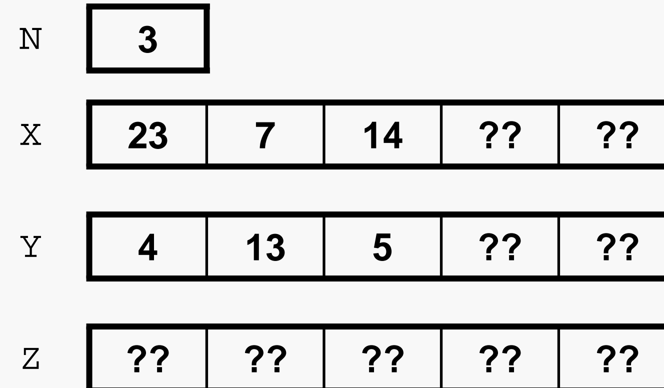
Invoke `addArray()` as follows:

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


©1995-2009 Barnette ND, McQuain WD & Kennan MA

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.

©1995-2009 Barnette ND, McQuain WD & Kennan MA

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`?

©1995-2009 Barnette ND, McQuain WD & Kennan MA

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.

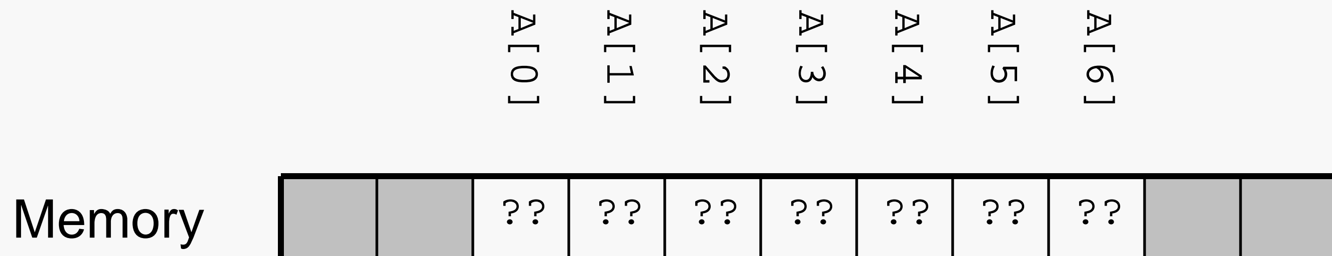
The effects of out-of-bounds index values range from incorrect results to subtle crashes to spectacular crashes.

©1995-2009 Barnette ND, McQuain WD & Kennan MA

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;
```


©1995-2009 Barnette ND, McQuain WD & Kennan MA

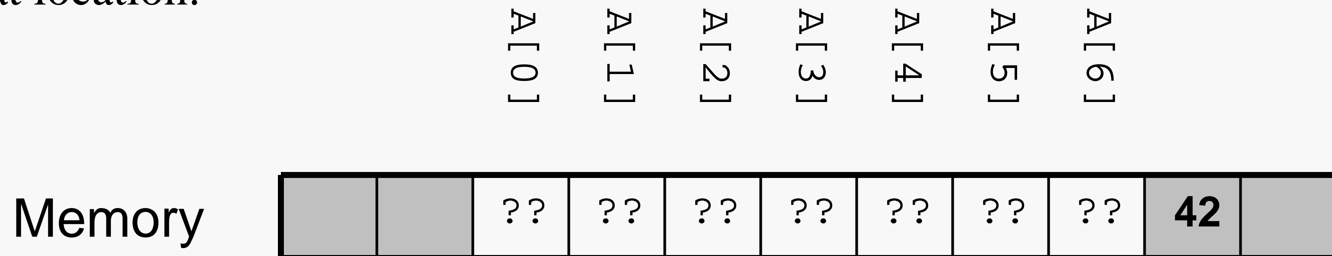
Out-of-Bounds Array Indices

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…

©1995-2009 Barnette ND, McQuain WD & Kennan MA

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.

©1995-2009 Barnette ND, McQuain WD & Kennan MA