BLG 102E
Introduction to Scientific Computing and Engineering

SPRING 2025

WEEK 8

İTÜ

ISTANBUL TECHNICAL UNIVERSITY

Arrays

Slides credit: Evan Gallagher

Arrays &
Functions

Arrays as Parameters in Functions

Recall that when we work with arrays we use a companion variable.

The same concept applies when using arrays as parameters:

You must pass the size to the function so it will know how many elements to work with.

©

Arrays as Parameters in Functions

Here is the `sum` function with an array parameter:
Notice that to pass one array, it takes two parameters.

```
double sum(double data[], int size)
{
   double total = 0;
   for (int i = 0; i < size; i++) {
      total = total + data[i];
   }
   return total;
}
```

©

Arrays as Parameters in Functions

empty pair of square brackets

```
double sum(double data[], int size)
{
   double total = 0;
   for (int i = 0; i < size; i++) {
      total = total + data[i];
   }
   return total;
}
```

©

Arrays as Parameters in Functions

You use an empty pair of square brackets *after* the parameter variable's name to indicate you are passing an array.

```
double sum(double data[], int size)
```

THIS IS AN ARRAY.

AND THIS IS ITS SIZE

©

Arrays as Parameters in Functions

When you call the function,
supply both the name of the array and the size:

```
double NUMBER_OF_SCORES = 10;
double scores[NUMBER_OF_SCORES]
   = { 32, 54, 67.5, 29, 34.5, 80, 115, 44.5, 100, 65 };
double total_score = sum(scores, NUMBER_OF_SCORES);
```

You can also pass a smaller size to the function:

```
double partial_score = sum(scores, 5);
```

This will sum over only the first five `double`s in the array.

©

Arrays as Parameters in Functions

When you pass an array into a function,
the contents of the array can *always* be changed:

```
void multiply(double values[], int size, double factor)
{
    for (int i = 0; i < size; i++) {
        values[i] = values[i] * factor;
    }
}
```

©

Passing Arrays to Functions

- Function prototype

```
void modifyArray( int b[], int arraySize );
```

  - Parameter names are optional in prototype
    - `int b[]` could be written `int []`
    - `int arraySize` could be simply `int`

```
void modifyArray(int [], int);
```

Example: Passing entire array to a function

Part 1 of 3

```
/*  Passing an array to a function */
#include <stdio.h>
#define SIZE 5

// function prototype
void modifyArray( int b[], int size );

// function main begins program execution
int main()
{
   int a[ SIZE ] = { 0, 1, 2, 3, 4 }; // initialize a
   int i; // counter

   printf( "Effects of passing entire array by reference:\n\nThe "
          "values of the original array are:\n" );

   // output original array
   for ( i = 0; i < SIZE; i++ ) {
      printf( "%3d", a[i] );
   }

   printf( "\n" );
```

Example: Passing entire array to a function

Part 2 of 3

```
   // pass array a to modifyArray by reference
   modifyArray( a, SIZE );

   printf( "The values of the modified array are:\n" );

   // output modified array
   for ( i = 0; i < SIZE; i++ ) {
      printf( "%3d", a[i] );
   }


} // end main
```

Part 3 of 3

```
/* in function modifyArray, "b" points to the original array "a"
   in memory */
void modifyArray( int b[], int size )
{
   int j; // counter

   // multiply each array element by 2
   for ( j = 0; j < size; j++ ) {
      b[ j ] *= 2;
   }

} // end function modifyArray
```

Program Output

```
Effects of passing entire array By Reference:

The values of the original array are:
   0   1   2   3   4

The values of the modified array are:
   0   2   4   6   8
```

The original array values changed

Sorting Arrays

- Sorting data is an important computing application.

- **Bubble Sort** method
  - Several passes through the array
  - Successive pairs of elements are compared
    - If increasing order (or identical), no change
    - If decreasing order, elements exchanged
  - Repeat above

- Small elements "bubble" to the top,
  (array is sorted from smallest to biggest.)

Example: Bubble Sort passes

Passes

1

4	75	74	2	54
4	75	74	2	54
4	74	75	2	54
4	74	2	75	54
4	74	2	54	75

2

4	74	2	54	75
4	74	2	54	75
4	2	74	54	75
4	2	54	74	75

3

4	2	54	74	75
2	4	54	74	75
2	4	54	74	75

4

2	4	54	74	75
2	4	54	74	75

5

2	4	54	74	75

Example: Bubble Sort Method

```c
/* fig06_15.c
   This program sorts an array's values into ascending order */
#include <stdio.h>

#define SIZE 10

// function main begins program execution
int main()
{
   // initialize a
   int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
   int pass; // passes counter
   int i;    // comparisons counter
   int hold; // temporary variable used to swap array elements

   printf( "Data items in original order\n" );

   // output original array
   for ( i = 0; i < SIZE; i++ ) {
      printf( "%4d", a[ i ] );
   }
```

Part 1 of 2

```c
    // Bubble Sort
    // loop to control number of passes
    for ( pass = 1; pass < SIZE; pass++ ) {

       // loop to control number of comparisons per pass
       for ( i = 0; i < SIZE - 1; i++ ) {

          /* compare adjacent elements and swap them if first
          element is greater than second element */
          if ( a[ i ] > a[ i + 1 ] ) {
             hold = a[ i ];
             a[ i ] = a[ i + 1 ];
             a[ i + 1 ] = hold;
          } // end if

       } // end inner for

    } // end outer for

    printf( "\nData items in ascending order\n" );
    // output sorted array
    for ( i = 0; i < SIZE; i++ ) {
       printf( "%4d", a[ i ] );
    }
 } // end main
```

Program Output

```
Data items in original order

   2   6   4   8  10  12  89  68  45  37



Data items in ascending order

   2   4   6   8  10  12  37  45  68  89
```

Searching Arrays: Linear Search

- Search an array for a *Key Value*
- Linear search
  - Simple
  - Compare each element of array with the given key value
  - Useful for small and unsorted arrays

Example: Linear search

Part 1 of 2

```
/* fig06_18.c
   Linear search of an array */
#include <stdio.h>

#define SIZE 100

// function prototype
int linearSearch(int array[], int key, int size );

int main()
{
   int a[ SIZE ]; // define array a
   int x; // counter for initializing elements 0-99 of array a
   int searchKey; // value to locate in array a
   int element_loc; // variable to hold location of searchKey or -1

   // create data
   for ( x = 0; x < SIZE; x++ ) {
      a[ x ] = 2 * x;
   }

   printf( "Enter integer search key:\n" );
   scanf( "%d", &searchKey );
```

```
   // attempt to locate searchKey in array a
   element_loc = linearSearch( a, searchKey, SIZE );

   // display results
   if ( element_loc != -1 ) {
      printf( "Found value in element %d\n", element_loc );
   }
   else {
      printf( "Value not found\n" );
   }

} // end main
```

```
/* compare key to every element of array until the location is found
   or until the end of array is reached; return subscript of element
   if key or -1 if key is not found */
int linearSearch(int array[], int key, int size ) {
   int i; // counter

   // loop through array
   for ( i = 0; i < size; i++ ) {
      if ( array[ i ] == key ) {
         return i; // return location of key
      } // end if
   } // end for

   return -1; // key not found
} // end function linearSearch
```

Program Outputs

```
Enter integer search key:
36

Found value in element 18
```

```
Enter integer search key:
37

Value not found
```

Arrays as Parameters in Functions

You can pass an array into a function

but

you cannot return an array.

©

Arrays as Parameters in Functions

If you cannot return an array, how can the caller get the data?

```
??? squares(int n)
{
    int result[];
    for (int i = 0; i < n; i++) {
        result[i] = i * i;
    }
    return result; // ERROR
}
```

©

Arrays as Parameters in Functions

The caller must provide an array to be used:

```
void squares(int n, int result[])
{
    for (int i = 0; i < n; i++) {
        result[i] = i * i;
    }
}
```

©

Arrays as Parameters in Functions

A function can change the size of an array.

It should let the caller know of any change by returning the new size.

```
int read_inputs(double inputs[], int capacity)
{
   int current_size = 0;
   double input;
   scanf("%lf", &input);
   while (input > 0) {
      if (current_size < capacity) {
         inputs[current_size] = input;
         current_size++;
      }
      scanf("%lf", &input);
   }
   return current_size;
}
```

©

Arrays as Parameters in Functions

Here is a call to the function:

```
const int MAXIMUM_NUMBER_OF_VALUES = 1000;
double values[MAXIMUM_NUMBER_OF_VALUES];
int current_size =
       read_inputs(values, MAXIMUM_NUMBER_OF_VALUES);
```

After the call,
the `current_size` variable
specifies how many were added.

©

Arrays as Parameters in Functions

The following program uses the preceding functions to read values from standard input, double them, and print the result.

- The `read_inputs` function fills an array with the input values. It returns the number of elements that were read.
- The `multiply` function modifies the contents of the array that it receives, demonstrating that arrays can be changed inside the function to which they are passed.
- The `print` function does not modify the contents of the array that it receives.

©

Arrays as Parameters in Functions

```
#include <stdio.h>
#include <stdlib.h>



/**
 * Reads a sequence of floating-point numbers.
 *
 * @param inputs an array containing the numbers
 * @param capacity the capacity of that array
 * @return the number of inputs stored in the array
 */
int read_inputs(double inputs[], int capacity)
{
```

©

Arrays as Parameters in Functions

```
    int current_size = 0;
    printf("Please enter values, 0 to quit:\n");
    bool more = true;
    while (more) {
       double input;
        scanf("%lf", &input);
        if (input <= 0) {
           more = false;
        } else if (current_size < capacity) {
           inputs[current_size] = input;
           current_size++;
        }
    }
    return current_size;
}
```

©

Arrays as Parameters in Functions

```
/**
 * Multiplies all elements of an array by a factor.
 *
 * @param values a partially filled array
 * @param size the number of elements in values
 * @param factor the value with which each element is
   multiplied
 */
void multiply(double values[], int size,
              double factor)
{
   for (int i = 0; i < size; i++) {
      values[i] = values[i] * factor;
   }
}
```

©

Arrays as Parameters in Functions

```
/**
 * Prints the elements of an array, separated by
   commas.
 *
 * @param values a partially filled array
 * @param size the number of elements in values
 */
void print(double values[], int size)
{
   for (int i = 0; i < size; i++) {
      if (i > 0) {
         printf(", ");
      }
      printf("%f”, values[i]);
   }
   printf("\n");
}
```

©

Arrays as Parameters in Functions

```
int main()
{
    const int CAPACITY = 1000;
    double values[CAPACITY];
    int size = read_inputs(values, CAPACITY);
    multiply(values, size, 2);
    print(values, size);

    return EXIT_SUCCESS;
}
```

©

Problem Solving: Adapting Algorithms

You can try to use algorithms you already know
to produce a new algorithm that will solve this problem.

(Then you'll have yet another algorithm.)

©

Problem Solving: Adapting Algorithms

Consider this problem:

Compute the final quiz score from a set of quiz scores,

but be nice:

drop the lowest score.

©

Problem Solving: Adapting Algorithms

What do I know how to do?

©

Problem Solving: Adapting Algorithms

Calculate the sum:

```
double total = 0;
for (int i = 0; i < size of values; i++) {
   total = total + values[i];
}
```

©

Problem Solving: Adapting Algorithms

Find the minimum:

```
double smallest = values[0];
for (int i = 1; i < size of values; i++) {
    if (values[i] < smallest) {
        smallest = values[i];
    }
}
```

©

Remove an element:

```
values[pos] = values[current_size - 1];
current_size--;
```

Problem Solving: Adapting Algorithms

Here is the algorithm:

1. *Find the minimum*
2. *Remove it from the array*
3. *Calculate the sum*
   (will be without the lowest score)
4. *Calculate the final score*

©

WAIT!

©

Problem Solving: Adapting Algorithms

```
values[pos] = values[current_size - 1];
current_size--;
```

This algorithm removes by knowing
the position
of the element to remove…

…but…

```
double smallest = values[0];
for (int i = 1; i < size of values; i++) {
    if (values[i] < smallest) {
        smallest = values[i];
    }
}
```

That’s not the *position* of the smallest –
it IS the smallest.

©

Here's another algorithm I know that *does* find the position:

```
int pos = 0;
bool found = false;
while (pos < size of values && !found) {
    if (values[pos] == 100) {
        found = true;
    } else {
        pos++;
    }
}
```

©

Here is the algorithm:

1. *Find the minimum*
2. *Find the position of the minimum*
   → the one I just searched for!!!
3. *Remove it from the array*
4. *Calculate the sum*
   (will be without the lowest score)
5. *Calculate the final score*

©

Problem Solving: Adapting Algorithms

But I’m repeating myself.

I searched

for the minimum

and then

I searched

for the position…

…of the minimum!

©

Problem Solving: Adapting Algorithms

I wonder if I can *adapt* the algorithm that finds the minimum so that it finds the position of the minimum?

©

Problem Solving: Adapting Algorithms

Start with this:

```
double smallest = values[0];
for (int i = 1; i < size of values; i++) {
    if (values[i] < smallest) {
        smallest = values[i];
    }
}
```

©

Problem Solving: Adapting Algorithms

What is it about the minimum value
and where the minimum value is?

```
double smallest = values[0];
for (int i = 1; i < size of values; i++) {
    if (values[i] < smallest) {
        smallest = values[i];
    }
}
```

Problem Solving: Adapting Algorithms

What is it about the minimum value
and where the minimum value is?

```
int smallest_pos = 0;
for (int i = 1; i < size of values; i++) {
   if (values[i] < values[smallest_pos]) {
      smallest_pos = i;
   }
}
```

©

Problem Solving: Adapting Algorithms

Finally:

1. *Find the* position *of the minimum*
2. *Remove it from the array*
3. *Calculate the sum*
   (will be without the lowest score)
4. *Calculate the final score*

©

There is a technique that you can use called:

MANIPULATING PHYSICAL OBJECTS

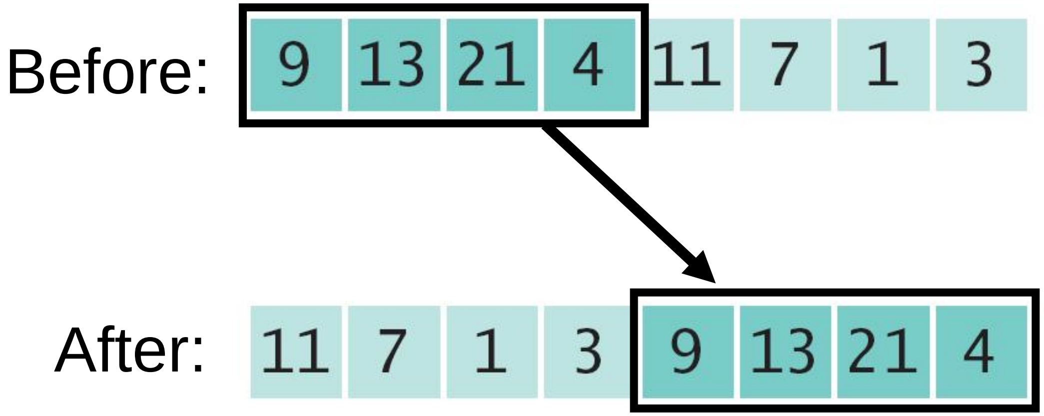
better know as:

playing around with things.

©

Here is a problem:

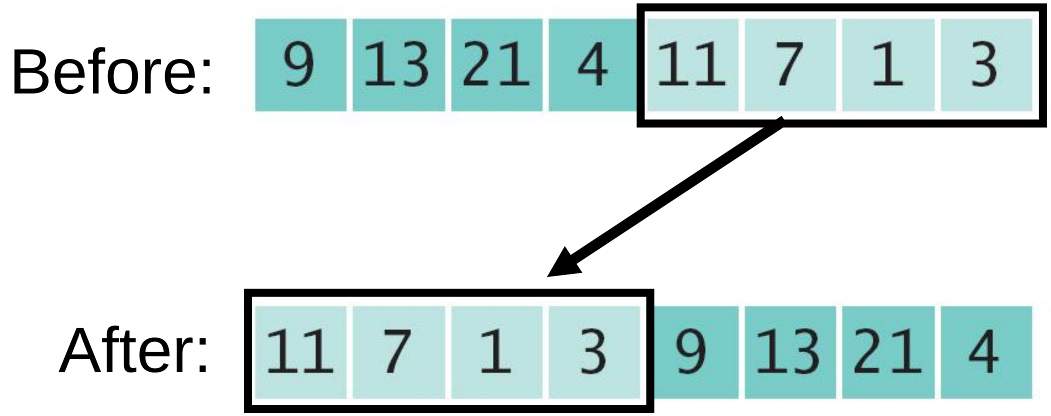
You are given an array whose size is an even number.
You are to switch the first and the second half.



Discovering Algorithms by Manipulating Physical Objects

Here is a problem:

You are given an array whose size is an even number.
You are to switch the first and the second half.



©

To learn this *Manipulating Physical Objects* technique,
let’s play with some coins
and review some algorithms you already know.

©

Discovering Algorithms by Manipulating Physical Objects

What algorithms do you know
that allow you to rearrange a set of coins?

©

Discovering Algorithms by Manipulating Physical Objects

You know how to remove a coin.

©

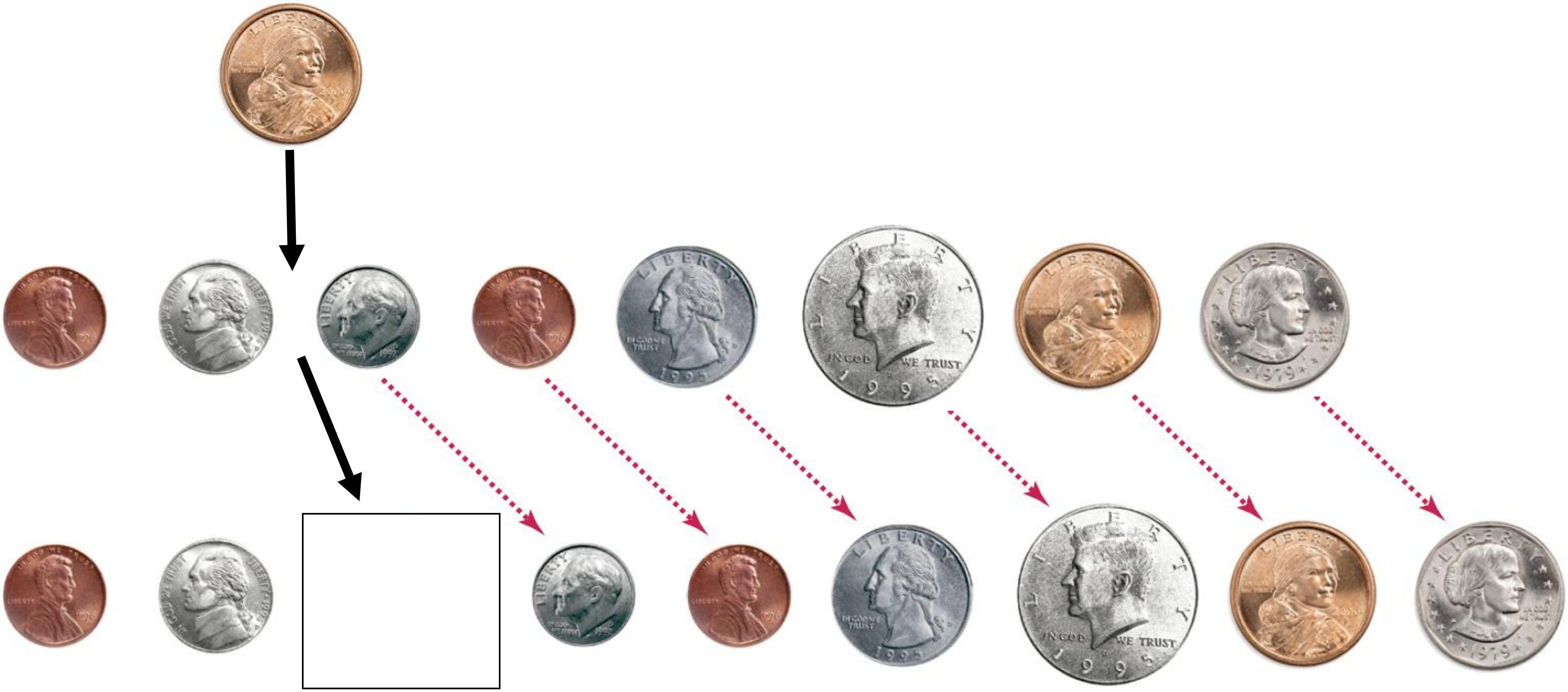
You know how to remove a coin.

©

You know how to insert a coin at a specific position.

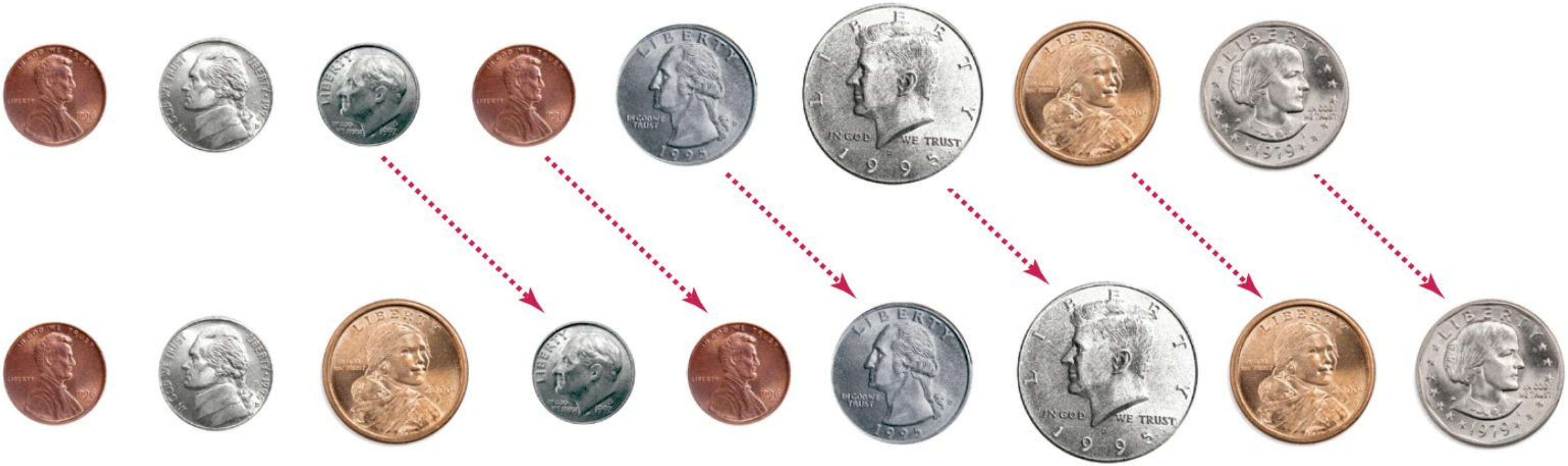
Discovering Algorithms by Manipulating Physical Objects

You know how to insert a coin at a specific position.



©

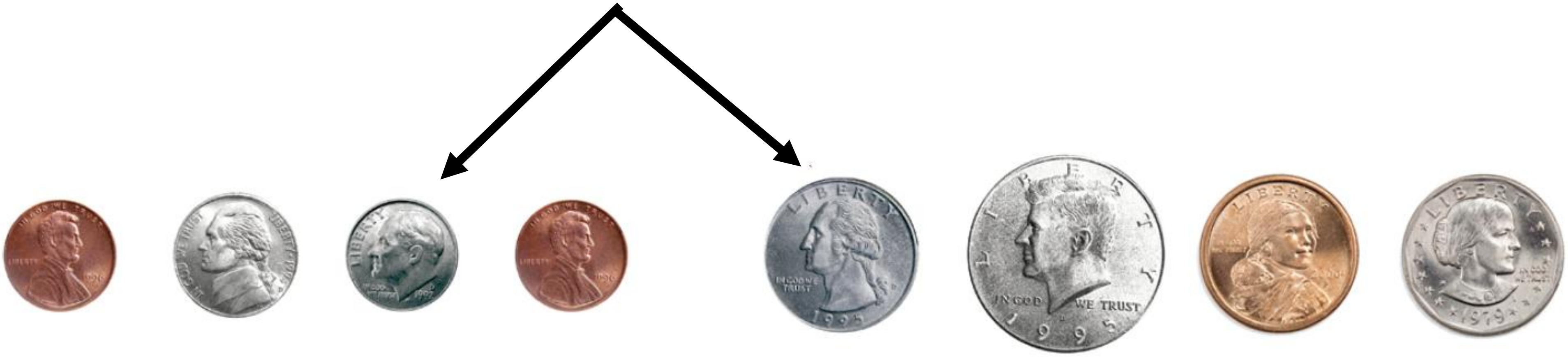
You know how to insert a coin at a specific position.



©

Discovering Algorithms by Manipulating Physical Objects

And you know how to swap two elements.



©

And you know how to swap two elements.

©

Discovering Algorithms by Manipulating Physical Objects

And you know how to swap two elements.

©

Discovering Algorithms by Manipulating Physical Objects

Swapping.

©

Discovering Algorithms by Manipulating Physical Objects

Swapping any two.

©

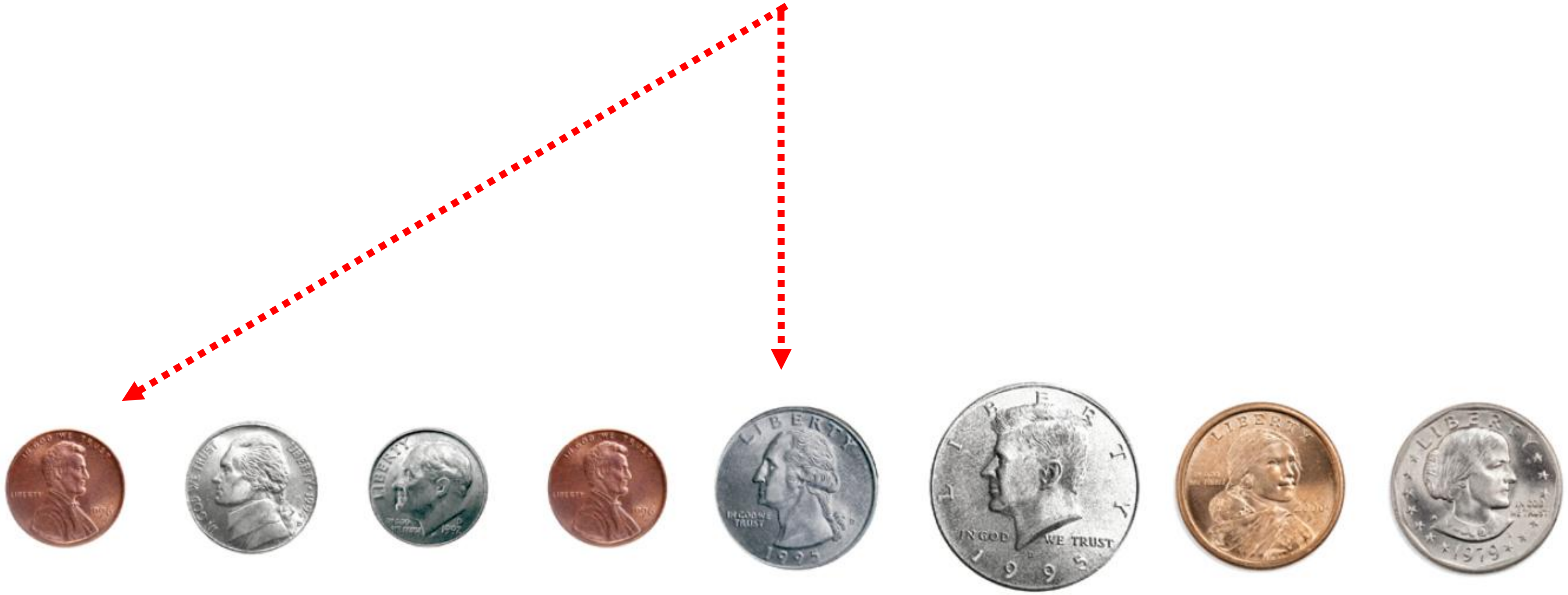
Discovering Algorithms by Manipulating Physical Objects

Any two.

©

Discovering Algorithms by Manipulating Physical Objects

Hm.



©

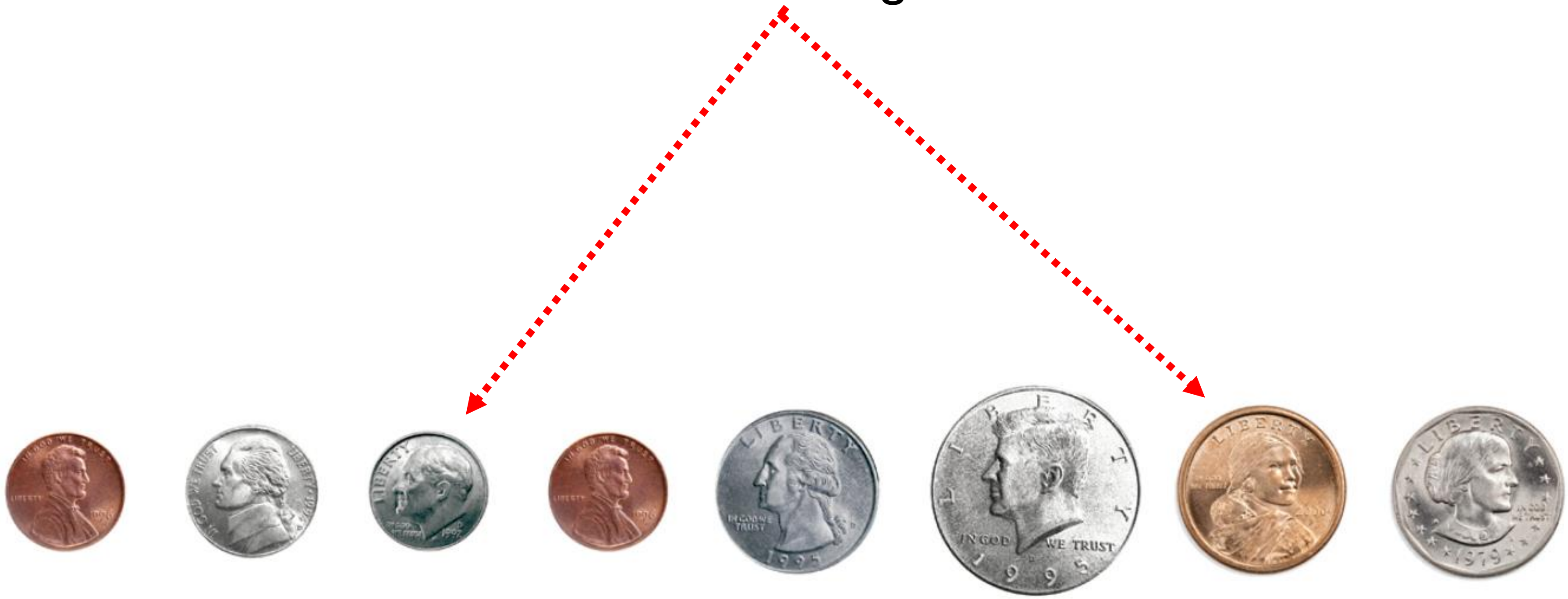
Discovering Algorithms by Manipulating Physical Objects

And hm.

©

Discovering Algorithms by Manipulating Physical Objects

Then hm again.



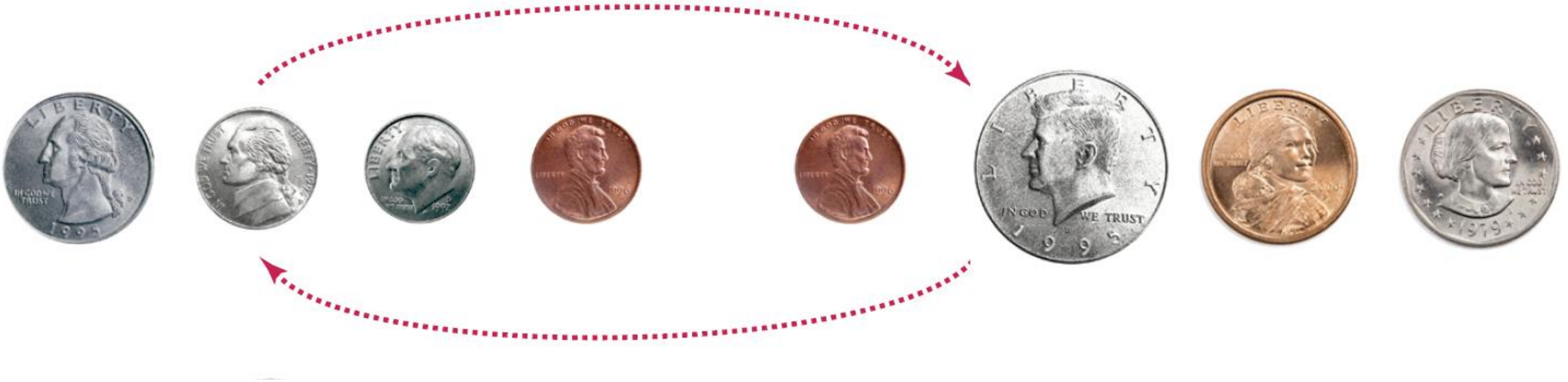
©

And finally…

Discovering Algorithms by Manipulating Physical Objects

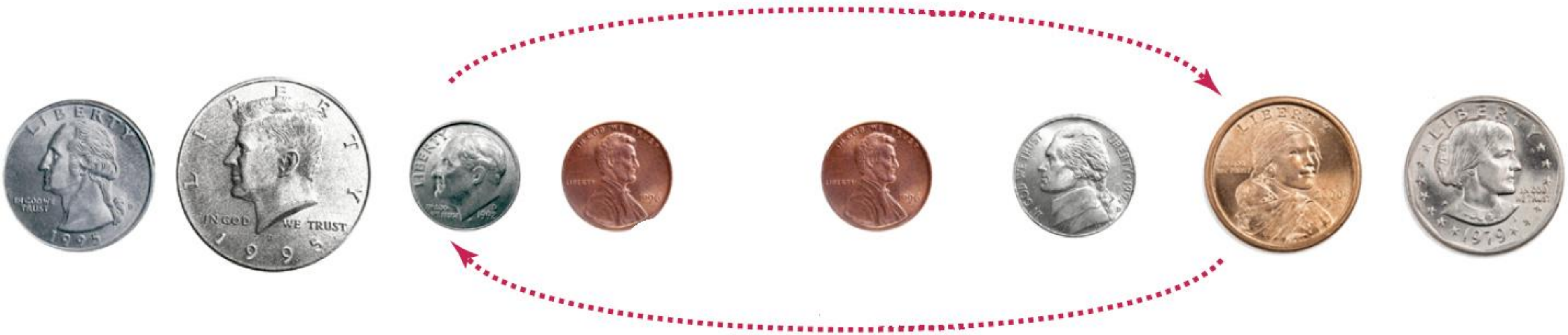
©

Discovering Algorithms by Manipulating Physical Objects



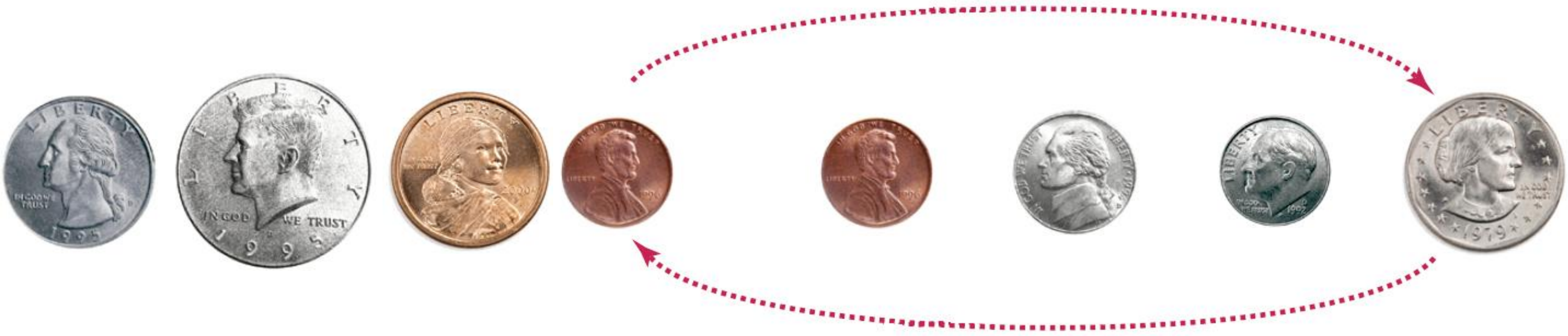
©

Discovering Algorithms by Manipulating Physical Objects



©

Discovering Algorithms by Manipulating Physical Objects

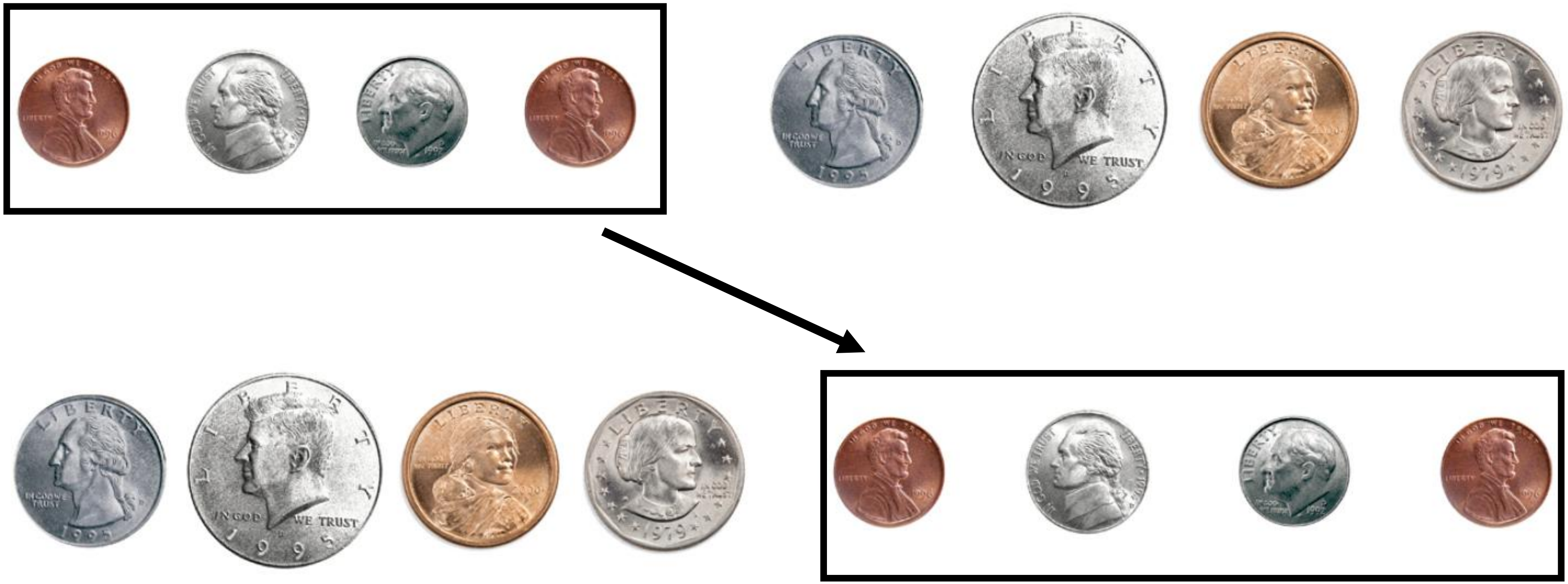


©

Discovering Algorithms by Manipulating Physical Objects

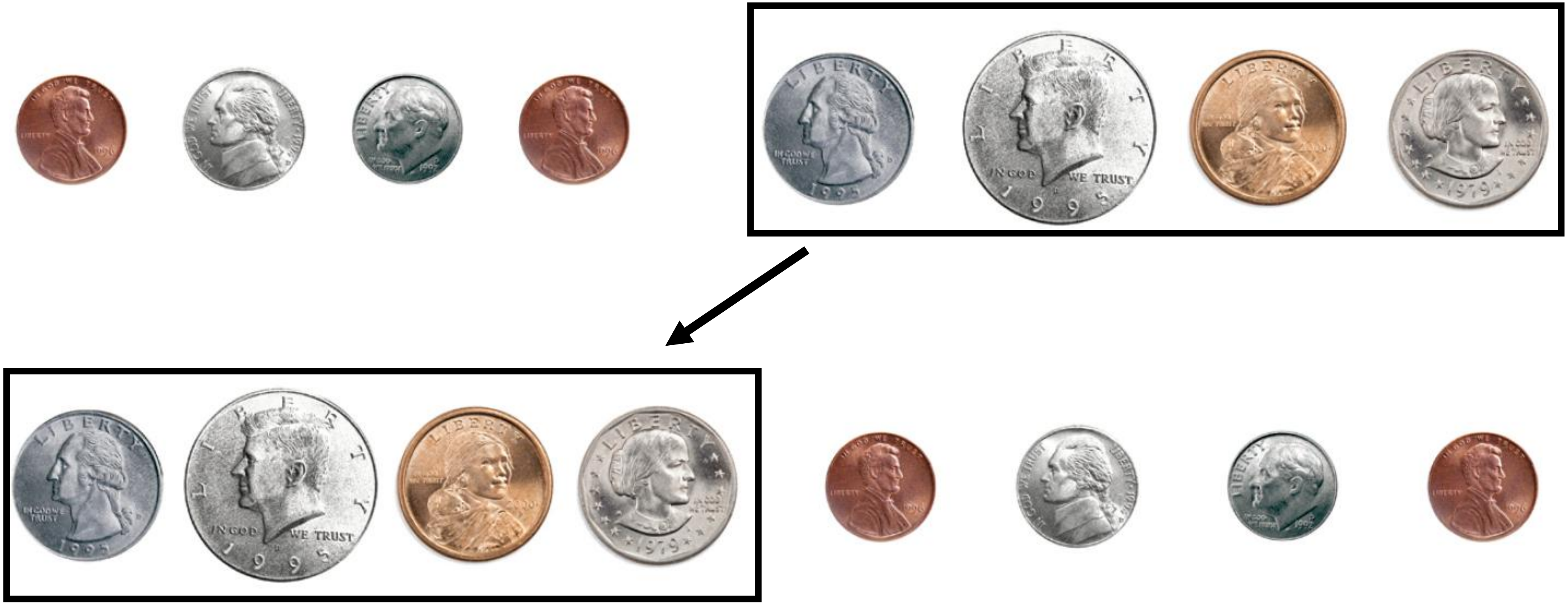
©

Discovering Algorithms by Manipulating Physical Objects



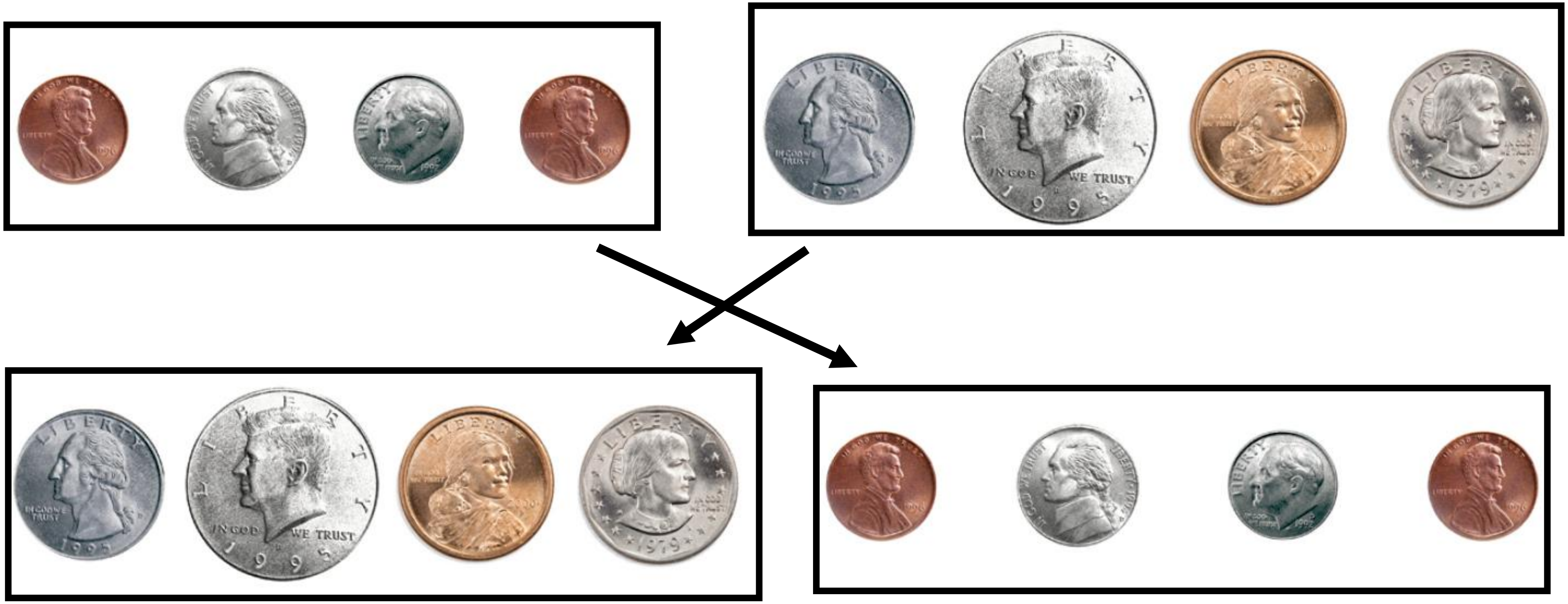
©

Discovering Algorithms by Manipulating Physical Objects



©

Discovering Algorithms by Manipulating Physical Objects



©

Discovering Algorithms by Manipulating Physical Objects

```
i =
j =
```

Two indices means we need two variables.

©

Discovering Algorithms by Manipulating Physical Objects

```
i =
j =
```

Initialization?

©

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j =
```

OK.

©

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j =
```

©

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = ?
```

Where does that index start?

©

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = ?
```

We swap the leftmost with somewhere in the middle.

©

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = ?
```

The middle!
That's it – half way
into the array.

©

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = size / 2;
```

Now we will loop…

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = size / 2;
while ( ??? )
```

…but…
for how long?
until when?

©

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = size / 2;
while ( ??? )
```

Let's think
about that later.

©

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = size / 2;
while ( ??? )
```

For certain we will be swapping the elements at the indices…

©

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = size / 2;
while ( ??? )
    swap elements at i and j
```

…and then go on to the next pair of indices to swap…

©

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = size / 2;
while ( ??? )
    swap elements at i and j
    i++;
    j++;
```

But when are we finished swapping?

©

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = size / 2;
while ( ??? )
    swap elements at i and j
    i++;
    j++;
```

We only process *half* the array

©

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = size / 2;
while (i < size / 2)
    swap elements at i and j
    i++;
    j++;
```

That's the algorithm!

©

Two-Dimensional Arrays

It often happens that you want to store collections of values that have a two-dimensional layout.

Such data sets commonly occur in financial and scientific applications.

©

Two-Dimensional Arrays

An arrangement consisting of *tabular data*:

rows and columns of values

is called:

a *two-dimensional array*, or a *matrix*.

©

Two Dimensional Arrays

Two-Dimensional Arrays

Consider this data from the 2010 Olympic skating competitions:

	Gold	Silver	Bronze
Canada	1	0	1
China	1	1	0
Germany	0	0	1
Korea	1	0	0
Japan	0	1	1
Russia	0	1	1
United States	1	1	0

©

Defining Two-Dimensional Arrays

C uses an array with *two* subscripts
to store a *two*-dimensional array.

```
const int COUNTRIES = 7;
const int MEDALS = 3;
int counts[COUNTRIES][MEDALS];
```

An array with 7 rows and 3 columns
is suitable for storing our medal count data.

©

Defining Two-Dimensional Arrays – Unchangeable Size

Just as with one-dimensional arrays,
you *cannot* change the size of
a two-dimensional array once it has been defined.

©

Defining Two-Dimensional Arrays – Initializing

But you can initialize a 2-D array:

```
int counts[COUNTRIES][MEDALS] =
    {
        { 1, 0, 1 },
        { 1, 1, 0 },
        { 0, 0, 1 },
        { 1, 0, 0 },
        { 0, 1, 1 },
        { 0, 1, 1 },
        { 1, 1, 0 }
    };
```

©

Defining Two-Dimensional Arrays

SYNTAX 6.3 Two-Dimensional Array Definition

Element type Rows Columns

Optional list of initial values

```
int data[4][4] = {
      { 16, 3, 2, 13 },
      { 5, 10, 11, 8 },
      { 9, 6, 7, 12 },
      { 4, 15, 14, 1 },
   };
```

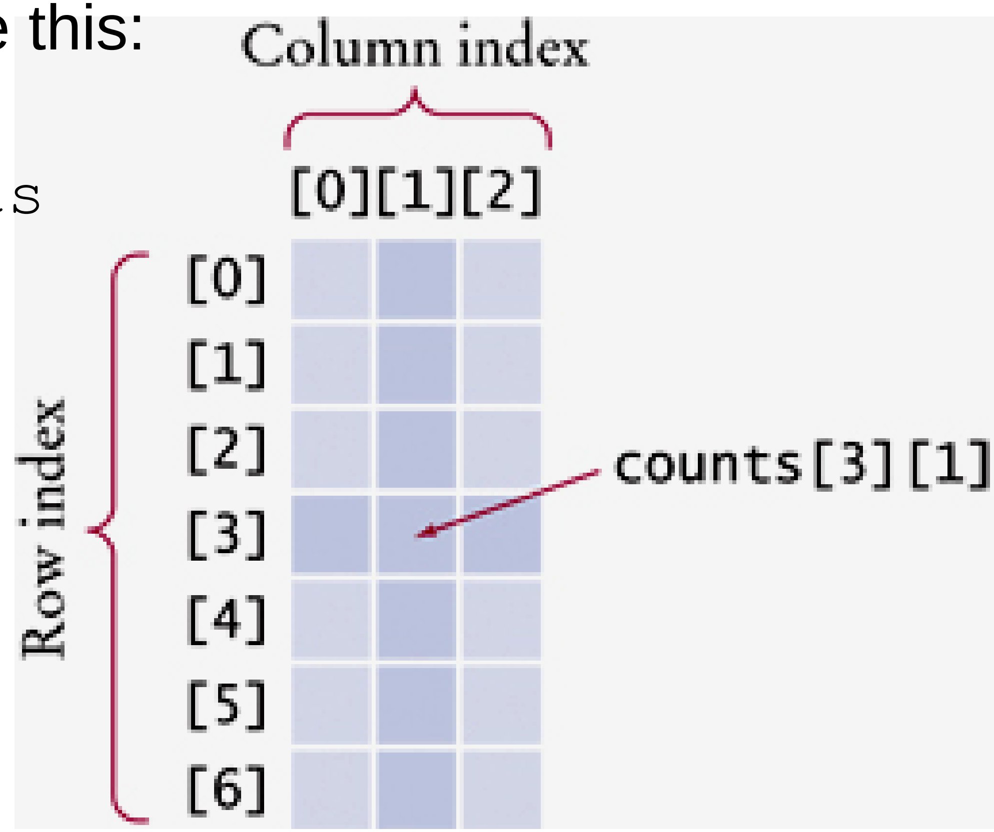
Name

©

Defining Two-Dimensional Arrays – Accessing Elements

The Olympic array looks like this:

`counts`

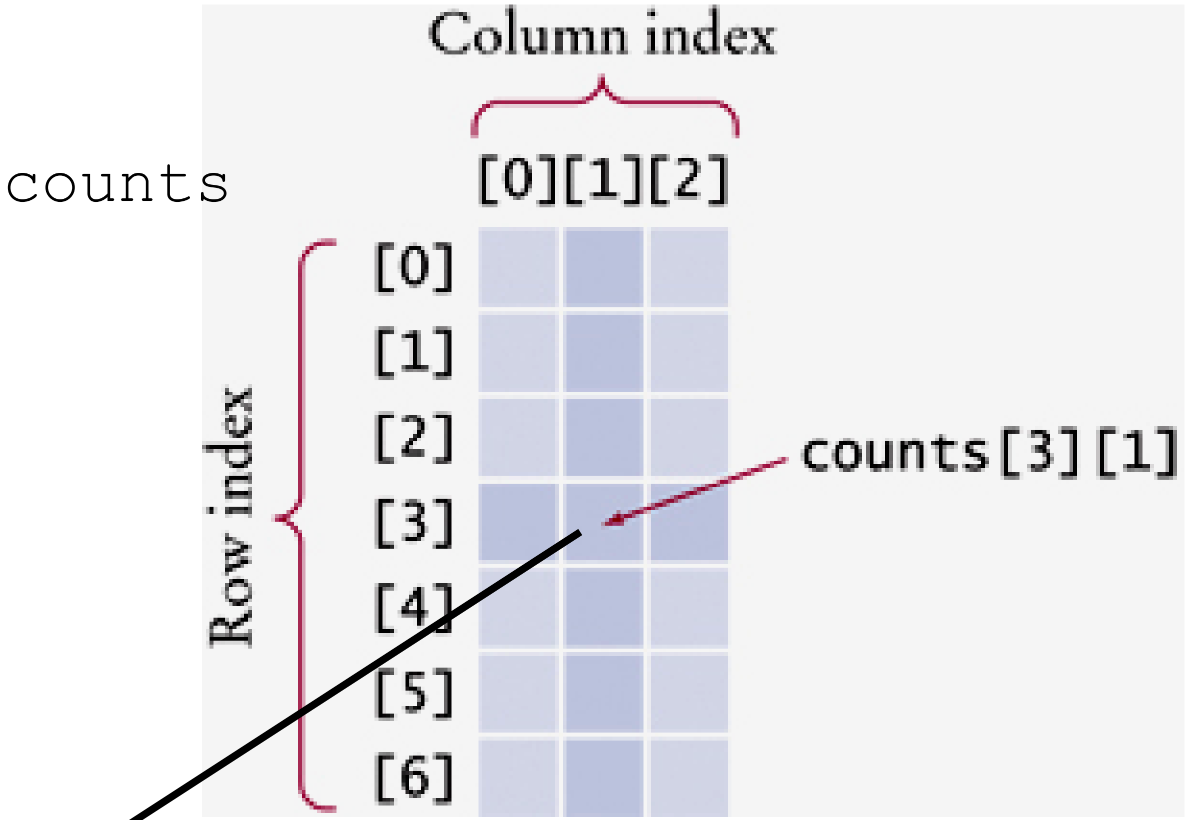


Access to the second element in the fourth row is:

`counts[3][1]`

©

Defining Two-Dimensional Arrays – Accessing Elements

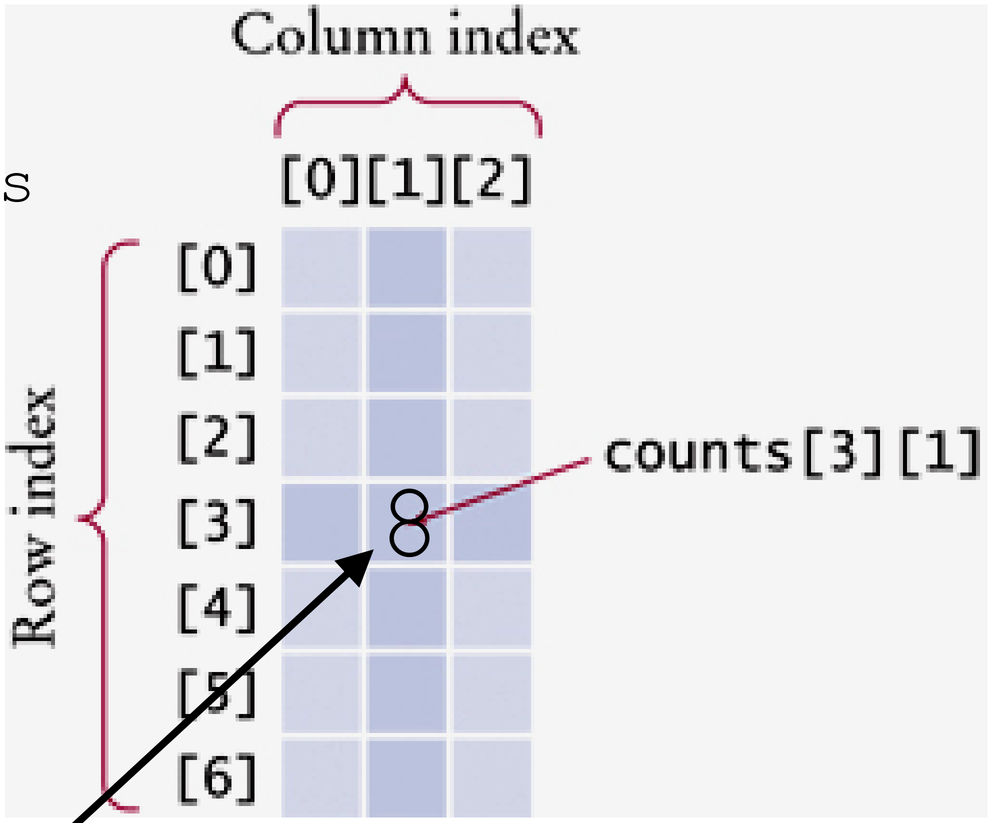



```
// set value to what is currently
// stored in the array at [3][1]
int value = counts[3][1];
```

©

Defining Two-Dimensional Arrays – Accessing Elements

counts




```
// set that position in the array to 8
counts[3][1] = 8;
```

©

Two-Dimensional Arrays

```
for (int i = 0; i < COUNTRIES; i++) {
    // Process the ith row
    for (int j = 0; j < MEDALS; j++) {
        // Process the jth column in the ith row
        printf("%8d", counts[i][j]);
    }
    // Start a new line at the end of the row
    printf("\n");
}
```

©

Computing Row and Column Totals

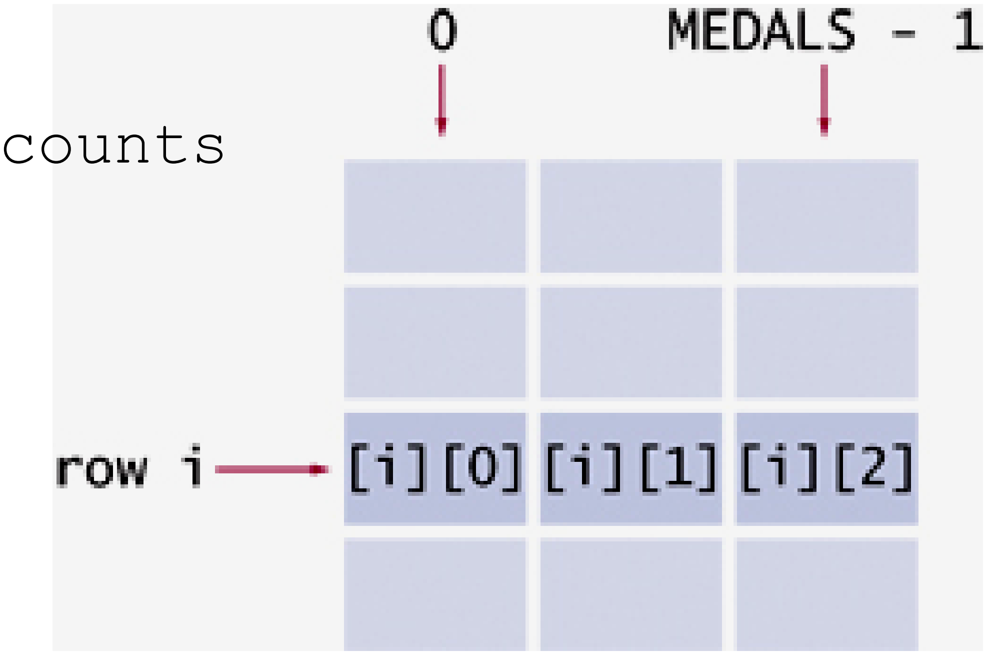
A common task is to compute row or column totals.

In our example,
the row totals give us the total number
of medals won by a particular country.

©

Computing Row and Column Totals

We must be careful to get the right indices.



For each row `i`, we must use the column indices:

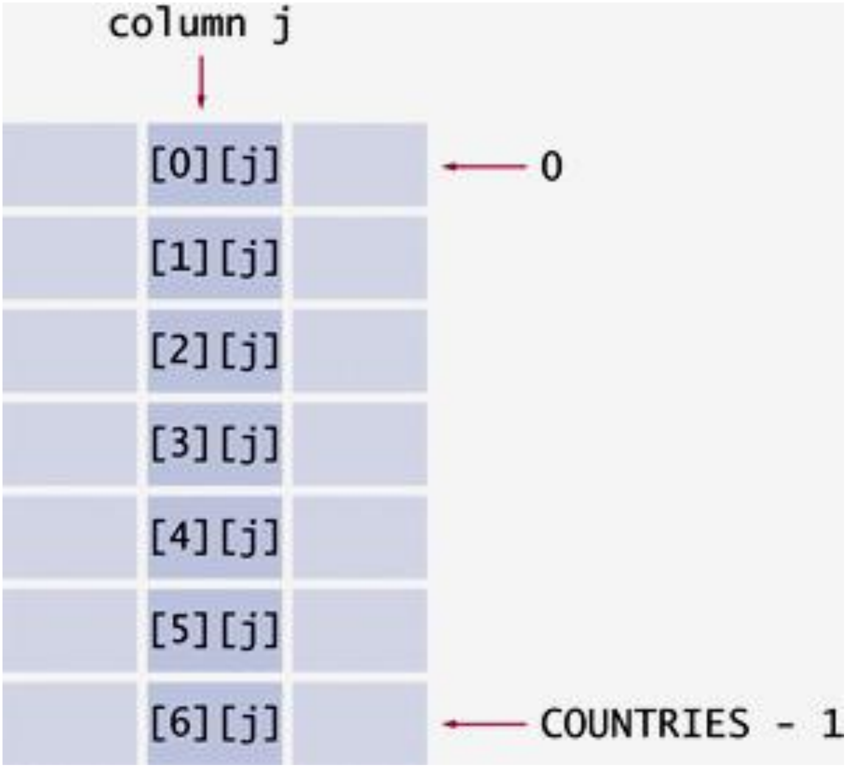
`0, 1, … (MEDALS -1)`

©

Computing Row and Column Totals

How many of each kind of medal was won by the set of these particular countries?

counts



That would be a column total.

Let `j` be the silver column:

```
int total = 0;
for (int i = 0; i < COUNTRIES; i++) {
   total = total + counts[i][j];
}
```

©

Two-Dimensional Array Parameters

When passing a two-dimensional array to a function, you must specify the number of columns *as a constant* when you write the parameter type.

```
table[][COLUMNS]
```

©

Two-Dimensional Array Parameters

This function computes the total of a given row.

```
const int COLUMNS = 3;
int row_total(int table[][COLUMNS], int row)
{
   int total = 0;
   for (int j = 0; j < COLUMNS; j++) {
      total = total + table[row][j];
   }
   return total;
}
```

©

Two-Dimensional Array Parameters

```
int row_total(int table[][COLUMNS], int row)
```

In this function, to find the element `table[row][j]`

the compiler generates code

by computing the offset

```
(row * COLUMNS) + j
```

©

Two-Dimensional Array Parameters

That function works for only arrays of 3 columns.

If you need to process an array
with a different number of columns, like 4,

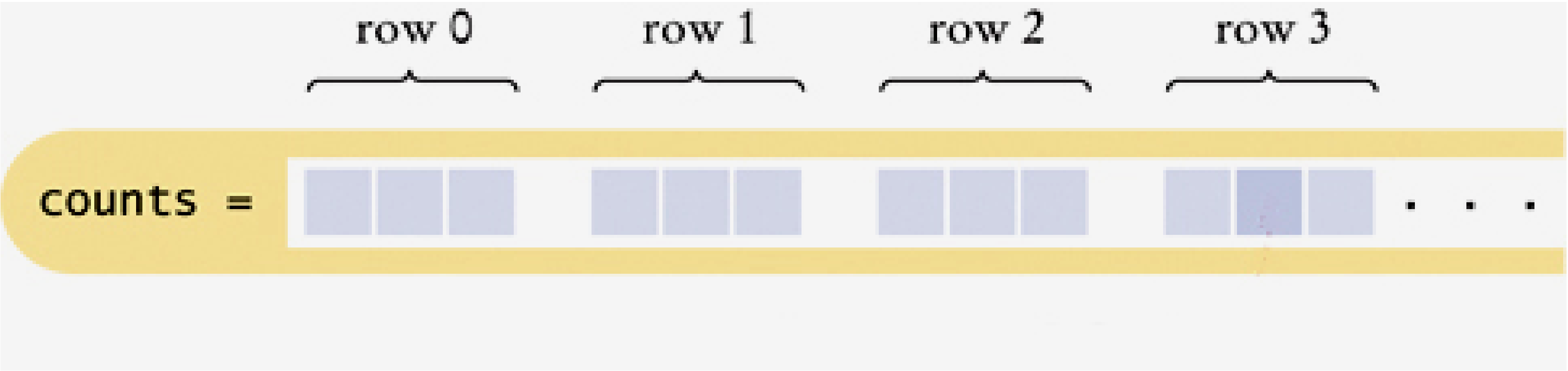
you would have to write
a different function
that has 4 as the parameter.

©

Two-Dimensional Array Parameters

What's the reason behind this?

Although the array appears to be two-dimensional, the elements are still stored as a linear sequence.



©

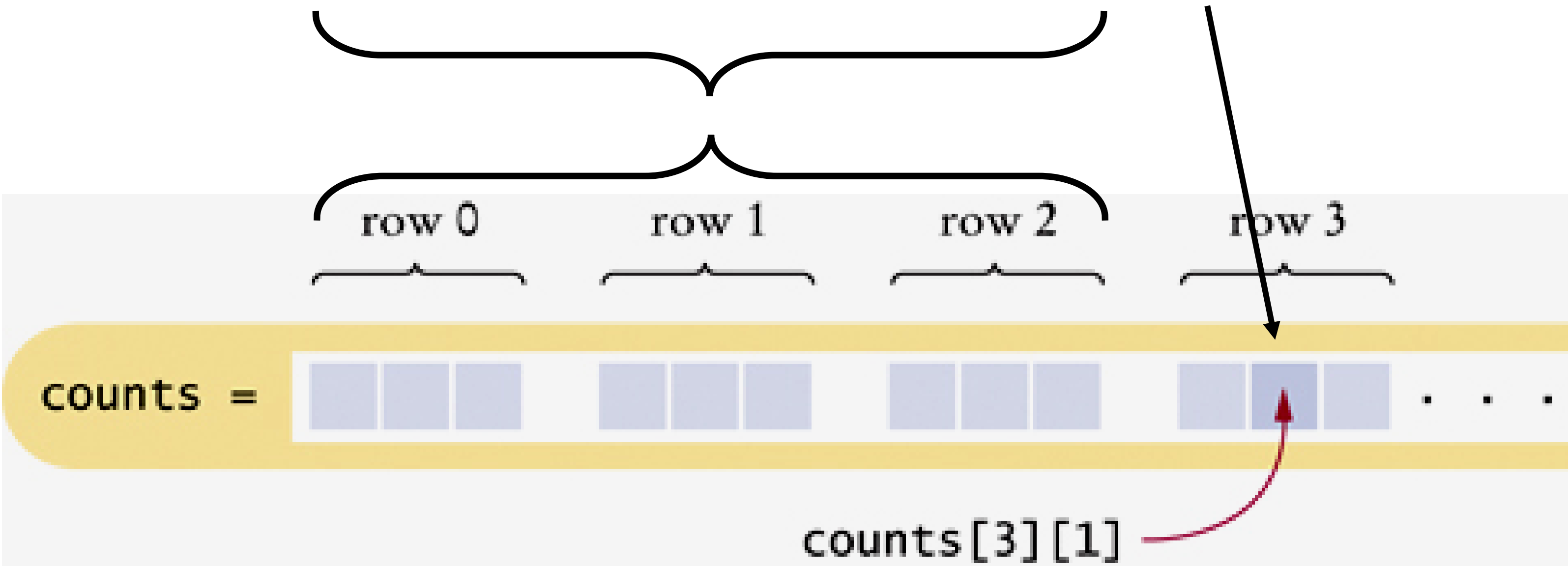
Two-Dimensional Array Parameters

`counts` is stored as a sequence of rows, each 3 long.

So where is `counts[3][1]`?

The offset from the start of the array is

`3 x` *`number of columns`* `+ 1`



©

Two-Dimensional Array Parameters

```
int row_total(int table[][COLUMNS], int row)
```

`table[]` looks like a normal 1D array.

Notice the empty square brackets.

©

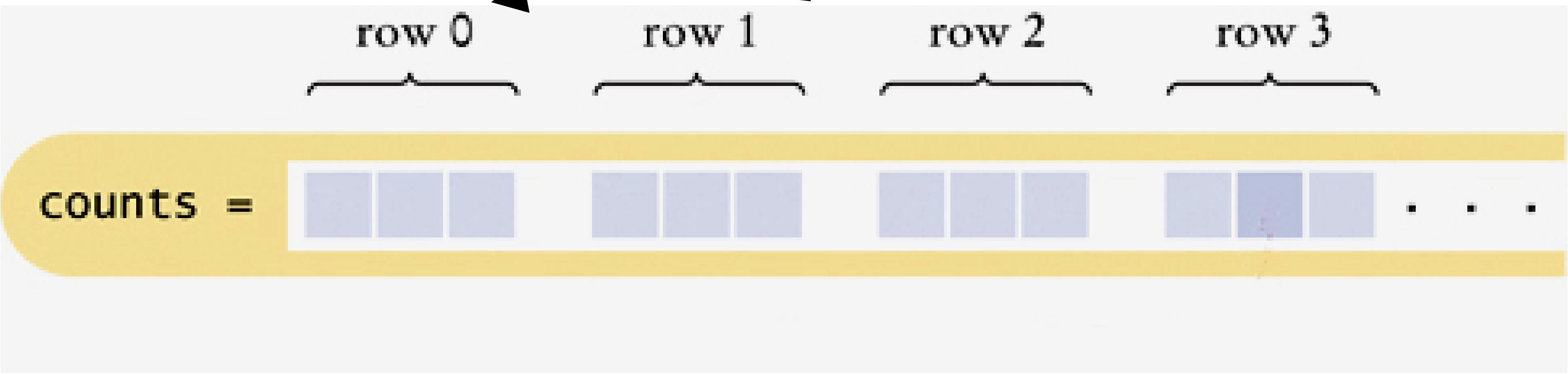
Two-Dimensional Array Parameters

```
int row_total(int table[][COLUMNS], int row)
```

`table[]` looks like a normal 1D array.

It is!

Each element is `COLUMNS` `int`s long.



©

Two-Dimensional Array Parameters

The `row_total` function did not need to know the number of rows of the array.

If the number of rows is required, pass it in:

```
int column_total(int table[][COLUMNS], int rows, int col)
{
    int total = 0;
    for (int i = 0; i < rows; i++) {
        total = total + table[i][col];
    }
    return total;
}
```

©

Two-Dimensional Array Parameters – Common Error

Leaving out the columns value is a very common error.

```
int row_total(int table[][], int row)
...
```

The compiler doesn't know how "long" each row is!

©

Two-Dimensional Array Parameters – Not an Error

Putting a value for the rows is not an error.

```
int row_total(int table[17][COLUMNS], int row)
...
```

The compiler just ignores whatever you place there.

©

Two-Dimensional Array Parameters – Not an Error

Putting a value for the rows is not an error.

```
int row_total(int table[17][COLUMNS], int row)
...
```

The compiler just ignores whatever you place there.

```
int row_total(int table[][COLUMNS], int row)
...
```

Never mind

©

Two-Dimensional Array Parameters

Here is the complete program for medal and column counts.

```
#include <stdio.h>
#include <stdlib.h>

#define MEDALS 3
#define COLUMNS 3
```

©

Two-Dimensional Array Parameters

```
/**
 * Computes the total of a row in a table.
 *
 * @param table a table with 3 columns
 * @param row the row that needs to be totaled
 * @return the sum of all elements in the given row
 */
double row_total(int table[][COLUMNS], int row)
{
   int total = 0;
   for (int j = 0; j < COLUMNS; j++) {
      total = total + table[row][j];
   }
   return total;
}
```

©

Two-Dimensional Array Parameters

```
int main()
{
   const int COUNTRIES = 7;

   char countries[][15] =
      {
         "Canada",
         "China",
         "Germany",
         "Korea",
         "Japan",
         "Russia",
         "United States"
      };
```

©

Two-Dimensional Array Parameters

```
int counts[][MEDALS] =
    {
        { 1, 0, 1 },
        { 1, 1, 0 },
        { 0, 0, 1 },
        { 1, 0, 0 },
        { 0, 1, 1 },
        { 0, 1, 1 },
        { 1, 1, 0 }
    };
```

©

Two-Dimensional Array Parameters

```
   printf("    Country  Gold  Silver  Bronze   Total\n");

   // Print countries, counts, and row totals
   for (int i = 0; i < COUNTRIES; i++) {
      printf("%15s", countries[i]);
      // Process the ith row
      for (int j = 0; j < MEDALS; j++) {
         printf("%8d", counts[i][j]);
      }
      int total = row_total(counts, i);
      printf("%8d\n", total);
   }
   return EXIT_SUCCESS;
}
```

©

Multiple-Subscripted Arrays

- Initialization
  - `int b[ 2 ][ 2 ] = { { 10, 20 }, { 30, 40 } };`
  - Initializers grouped by row in braces

10	20
30	40

  - If not enough data, unspecified elements set to zero

```
int b[ 2 ][ 2 ] = { { 10}, { 30, 40 } };
```

10	0
30	40

- Referencing elements
  - Specify row, then column

```
printf( "%d", b[ 1 ][ 0 ] ); //Displays 30
```

Example: Matrix initialization

Part 1 of 2

```c
/* fig06_21.c
   Initializing multidimensional arrays */
#include <stdio.h>

void printArray(int a[2][3] ); // function prototype

int main()
{
   // initialize array1, array2, array3
   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 } };

   printf( "Values in array1 by row are:\n" );
   printArray( array1 );

   printf( "Values in array2 by row are:\n" );
   printArray( array2 );

   printf( "Values in array3 by row are:\n" );
   printArray( array3 );

} // end main
```

Part 2 of 2

```
// function to output array with two rows and three columns
void printArray(int a[2][3] )
{
   int i; // row counter
   int j; // column counter

   // loop through rows
   for ( i = 0; i <= 1; i++ ) {

      // output column values
      for ( j = 0; j <= 2; j++ ) {
         printf( "%d ", a[ i ][ j ] );
      } // end inner for

      printf( "\n" ); // start new line of output
   } // end outer for

} // end function printArray
```

Program Output

```
Values in array1 by row are:
1 2 3
4 5 6

Values in array2 by row are:
1 2 3
4 5 0

Values in array3 by row are:
1 2 0
4 0 0
```

Example: Adding Two Matrices

```c
#include <stdio.h>
int main() {
    int a[2][2] = {{10, 15}, {20, 5}};  // Matrix a
    int b[2][2] = {{25,  5}, { 6, 0}};  // Matrix b
    int c[2][2]; // Matrix c
    int i, j;
    printf ("RESULTING ADDITION MATRIX \n\n");

    for(i=0; i<2; i++)  {
        for(j=0; j<2; j++)   {
            c[i][j] = a[i][j] + b[i][j];
            printf ("%d\t", c[i][j]);
        } // end inner for
        printf ("\n"); // new line
     } // end outer for
} // end main
```

Program Output

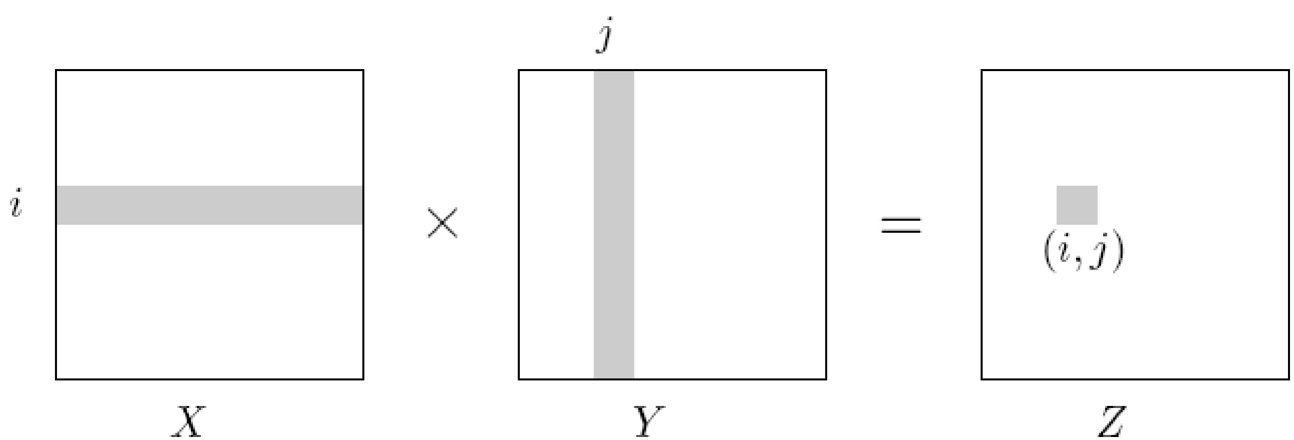
```
RESULTING ADDITION MATRIX

35      20
26      5
```

Example: Multiplying Two Matrices

- The product of two nxn matrices X and Y is a third nxn matrix Z = X.Y , with (i,j)th entry.

- **Linear Algebra** formula:

$$Z_{ij} = \sum_{k=1}^{N} X_{ik} Y_{kj}$$



Example: Multiplying Two Matrices (continued)

- Example: X and Y are two 2x2 matrices.
- Z is also a 2x2 matrix (Z = X . Y)

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

Example: Multiplying Two Matrices

Part 1 of 3

```
#include <stdio.h>
#define N 2
#define M 4
#define L 3

int main()
{
    int a[N][M] = {{ 8,  5, -6,  7},
                   { 0,  2,  1,  4}
                  };

    int b[M][L] = {{ 3, -9,  1},
                   { 2,  5,  8},
                   {-2,  4,  0},
                   { 1,  7,  6}
                  };

    int c[N][L];

    int i,j,k;
```

Example: Multiplying Two Matrices

Part 2 of 3

```
// Compute the multiplication
    for (i = 0; i < N; i++)
    {
        for (j = 0; j < L; j++)
        {
            c[i][j] = 0;
            for (k = 0; k < M; k++)
                c[i][j] += a[i][k] * b[k][j];
        }
    }
```

Example: Multiplying Two Matrices

Part 3 of 3

```c
    // Display the result matrix

    printf ("RESULT OF MULTIPLICATION MATRIX \n\n");

    for(i=0; i < N; i++)
    {
        for (j=0; j < L; j++)
            printf ("%d\t", c[i][j]);

        printf ("\n");
     }

} // end main
```

Program Output

```
RESULT OF MULTIPLICATION MATRIX

53      -22     90
6       42      40
```

Arrays – One Drawback

The size of an array *cannot* be changed after it is created.

You have to get the size right – *before* you define an array.

The compiler has to know the size to build it.
and a function must be told about the number
elements and possibly the capacity.

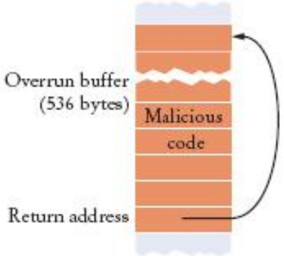
It cannot hold more than it's initial capacity.

©

SUMMARY

Use arrays for collecting values.

- Use an array to collect a sequence of values of the same type.
- Individual elements in an array *values* are accessed by an integer index `i`, using the notation *values*`[i]`.
- An array element can be used like any variable.
- An array index must be at least zero and less than the size of the array.
- A bounds error, which occurs if you supply an invalid array index, can corrupt data or cause your program to terminate.
- With a partially filled array, keep a companion variable for the current size.

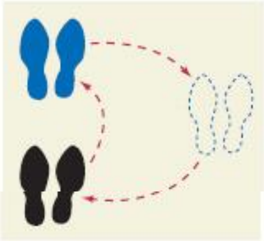


©

SUMMARY

Be able to use common array algorithms.

- To copy an array, use a loop to copy its elements to a new array.
- When separating elements, don't place a separator before the first element.
- A linear search inspects elements in sequence until a match is found.
- Before inserting an element, move elements to the end of the array *starting with the last one*.
- Use a temporary variable when swapping two elements.



Implement functions that process arrays.

- When passing an array to a function, also pass the size of the array.
- Array parameters are always reference parameters.
- A function's return type cannot be an array.
- When a function modifies the size of an array, it needs to tell its caller.
- A function that adds elements to an array needs to know its capacity.

©

SUMMARY

Be able to combine and adapt algorithms for solving a programming problem.

- By combining fundamental algorithms, you can solve complex programming tasks.
- You should be familiar with the implementation of fundamental algorithms so that you can adapt them.

Discover algorithms by manipulating physical objects.

- Use a sequence of coins, playing cards, or toys to visualize an array of values.
- You can use paper clips as position markers or counters.

Use two-dimensional arrays for data that is arranged in rows and columns.

- Use a two-dimensional array to store tabular data.
- Individual elements in a two-dimensional array are accessed by using two subscripts, *array*`[i][j]`.
- A two-dimensional array parameter must have a fixed number of columns.

©