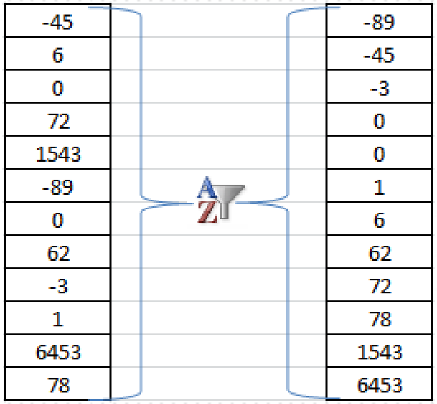
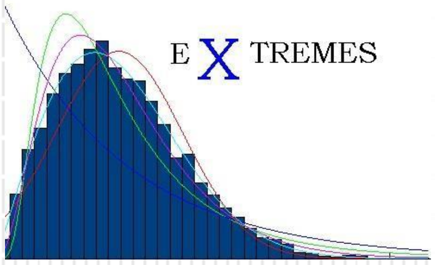
Programming in C

Array Subtasks



-45	-89
6	-45
0	-3
72	0
1543	0
-89	1
0	6
62	62
-3	72
1	78
6453	1543
78	6453

$$\Sigma$$



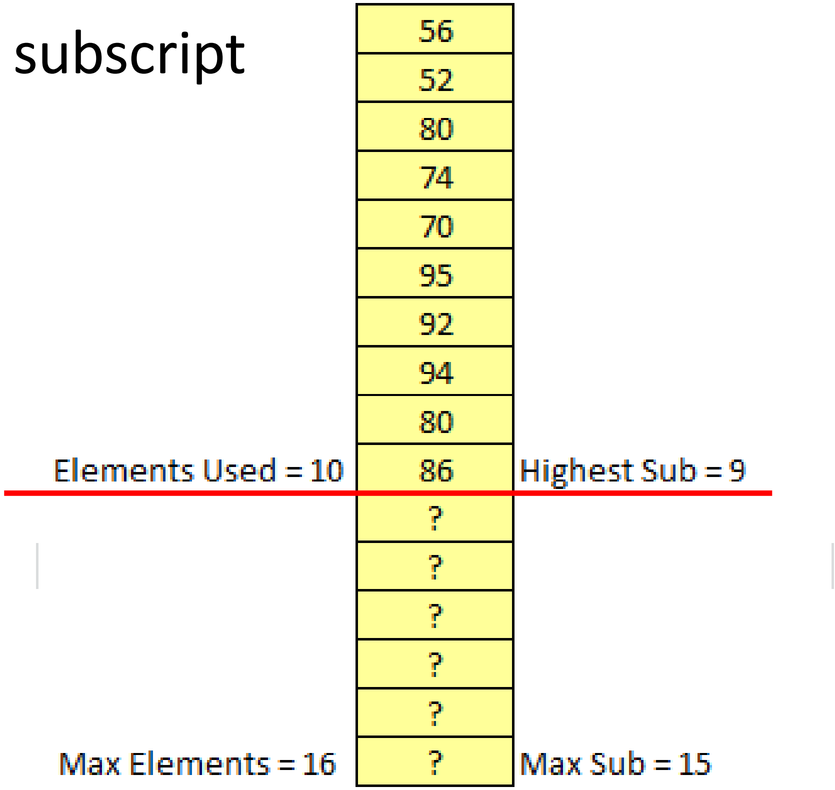
Programming with Arrays

- Subtasks
  - Partially-filled arrays
  - Loading
  - Searching
  - Sorting
  - Sum, average
  - Extremes

Partially-filled Arrays (Common Case)

- Must be declared some maximum size
- Program must maintain
  - How many elements are being used

    and/or
  - Highest subscript



56

52

80

74

70

95

92

94

80

Elements Used = 10 86 Highest Sub = 9

?

?

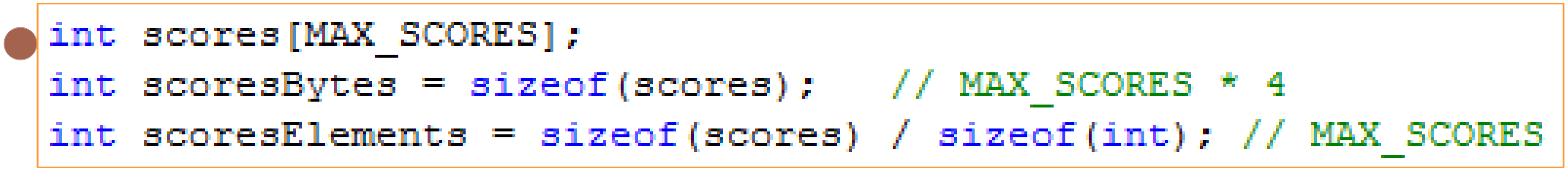
?

?

?

Max Elements = 16 ? Max Sub = 15

Sizeof and Arrays

- Operator sizeof returns the total bytes in the argument
  - Total elements = sizeof(array) / sizeof(data-type)
  - 

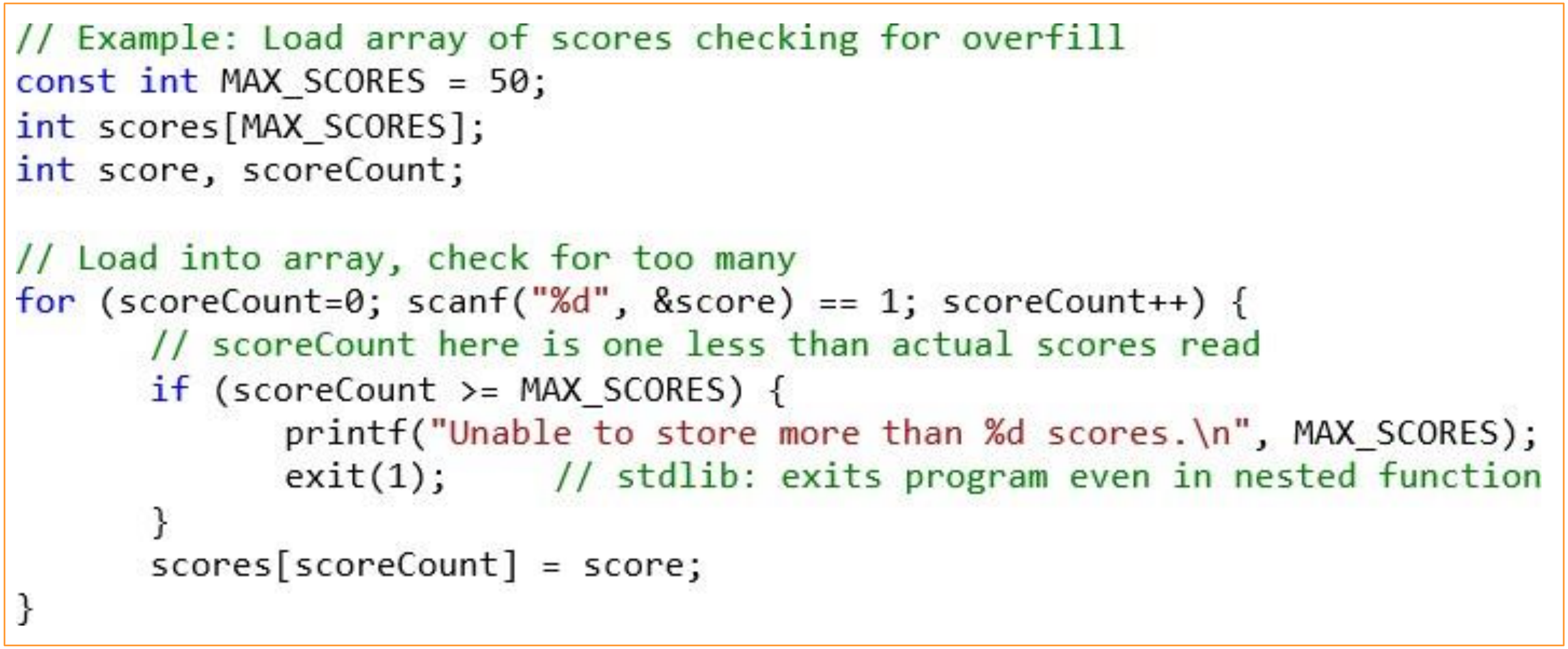
```
int scores[MAX_SCORES];
int scoresBytes = sizeof(scores);    // MAX_SCORES * 4
int scoresElements = sizeof(scores) / sizeof(int); // MAX_SCORES
```

- Sizeof does not return total bytes being used

You cannot use sizeof to determine the number of elements being used in a partially filled array

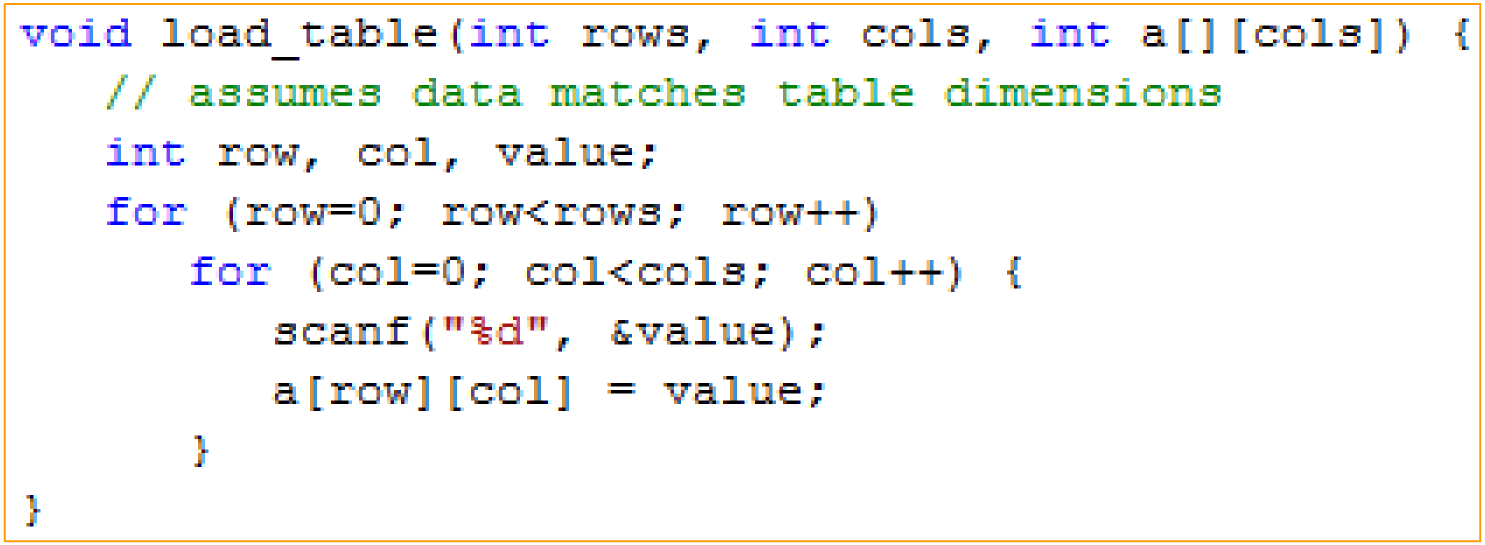
Loading an Array

- Be careful not to overfill
  - Do not read directly into array elements



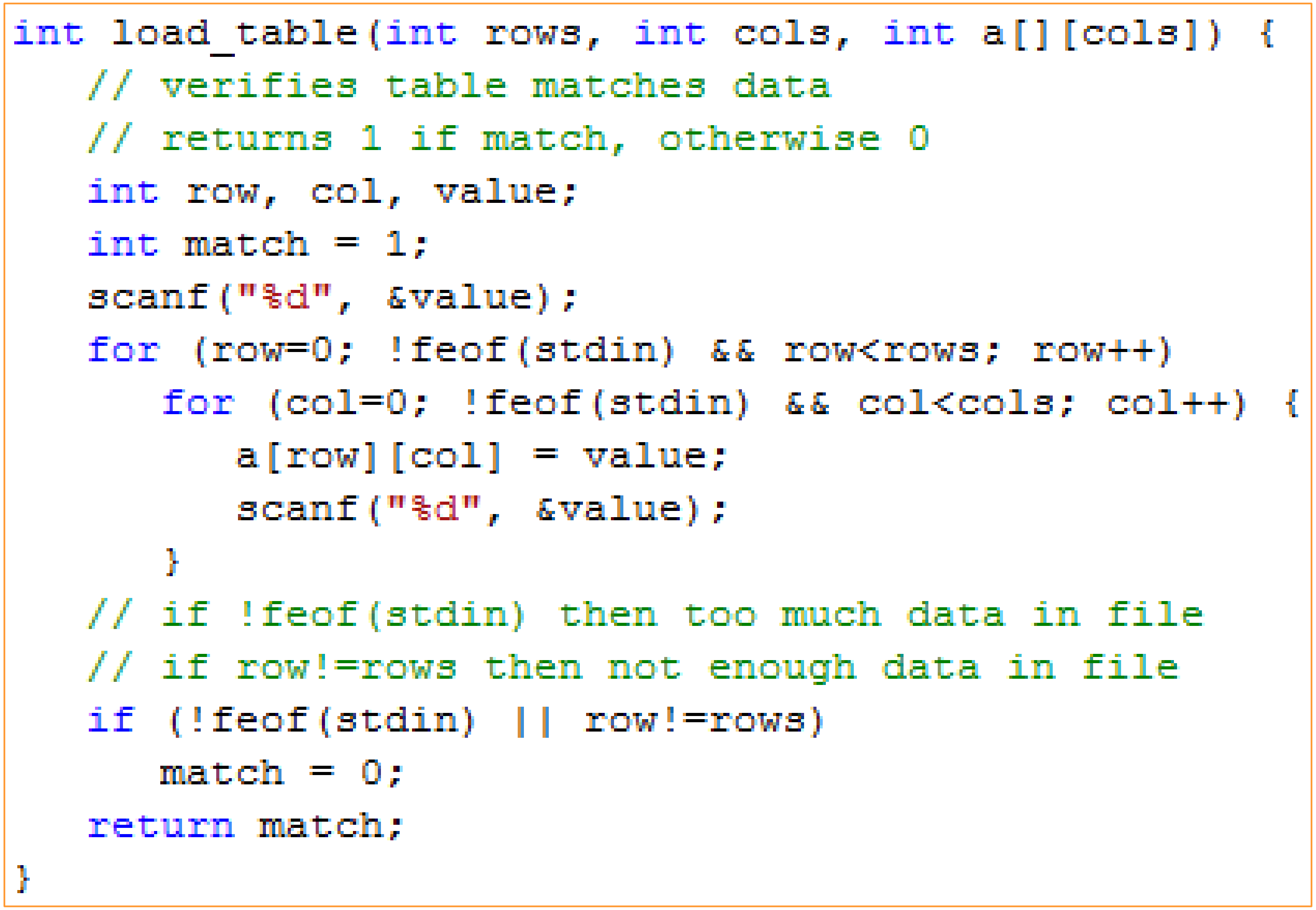
```
// Example: Load array of scores checking for overfill
const int MAX_SCORES = 50;
int scores[MAX_SCORES];
int score, scoreCount;

// Load into array, check for too many
for (scoreCount=0; scanf("%d", &score) == 1; scoreCount++) {
        // scoreCount here is one less than actual scores read
        if (scoreCount >= MAX_SCORES) {
                printf("Unable to store more than %d scores.\n", MAX_SCORES);
                exit(1);        // stdlib: exits program even in nested function
        }
        scores[scoreCount] = score;
}
```

Loading a Two-dimensional Array



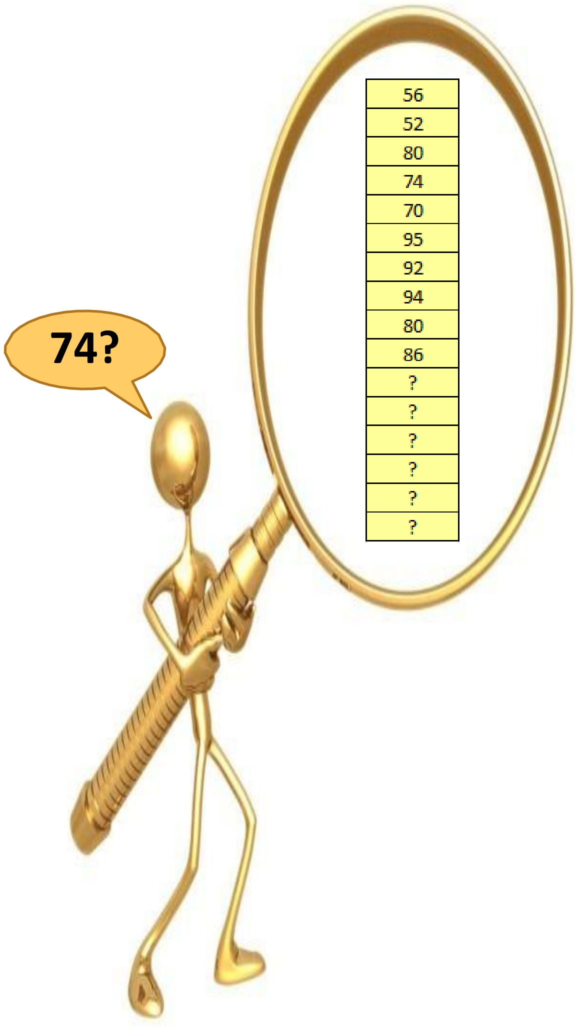
```
void load_table(int rows, int cols, int a[][cols]) {
   // assumes data matches table dimensions
   int row, col, value;
   for (row=0; row<rows; row++)
      for (col=0; col<cols; col++) {
         scanf("%d", &value);
         a[row][col] = value;
      }
}
```

Safer 2D Load

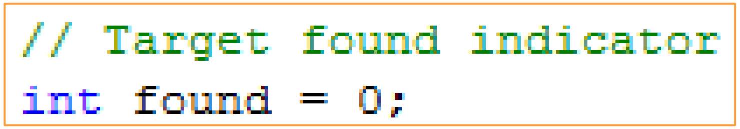


```
int load_table(int rows, int cols, int a[][cols]) {
   // verifies table matches data
   // returns 1 if match, otherwise 0
   int row, col, value;
   int match = 1;
   scanf("%d", &value);
   for (row=0; !feof(stdin) && row<rows; row++)
      for (col=0; !feof(stdin) && col<cols; col++) {
         a[row][col] = value;
         scanf("%d", &value);
      }
   // if !feof(stdin) then too much data in file
   // if row!=rows then not enough data in file
   if (!feof(stdin) || row!=rows)
      match = 0;
   return match;
}
```

Searching an Array

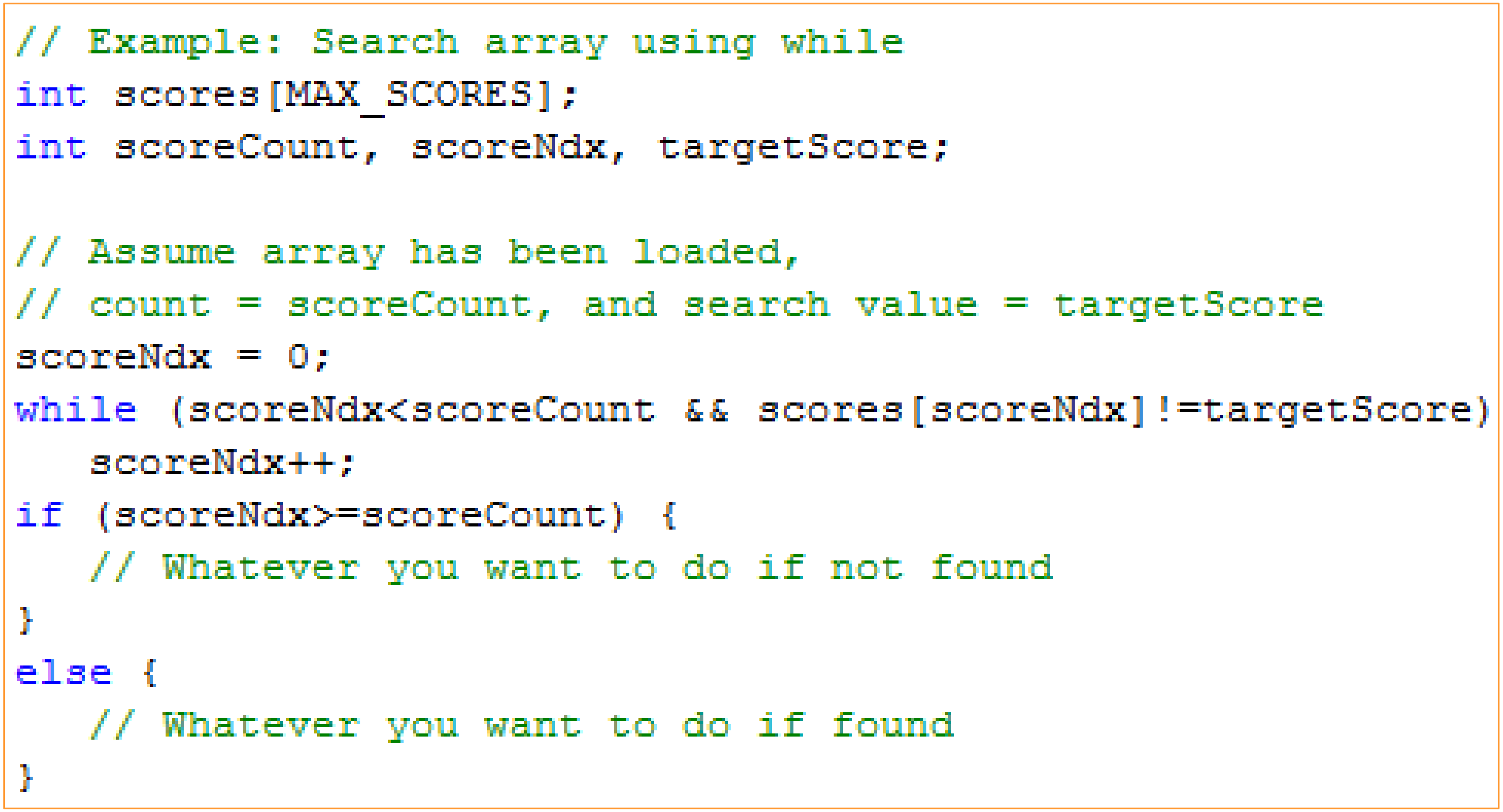


- Linear search
  - Simple
- Binary search
  - Requires sorted array
  - Generally faster for large arrays
- May require the use of an indicator to denote found or not found



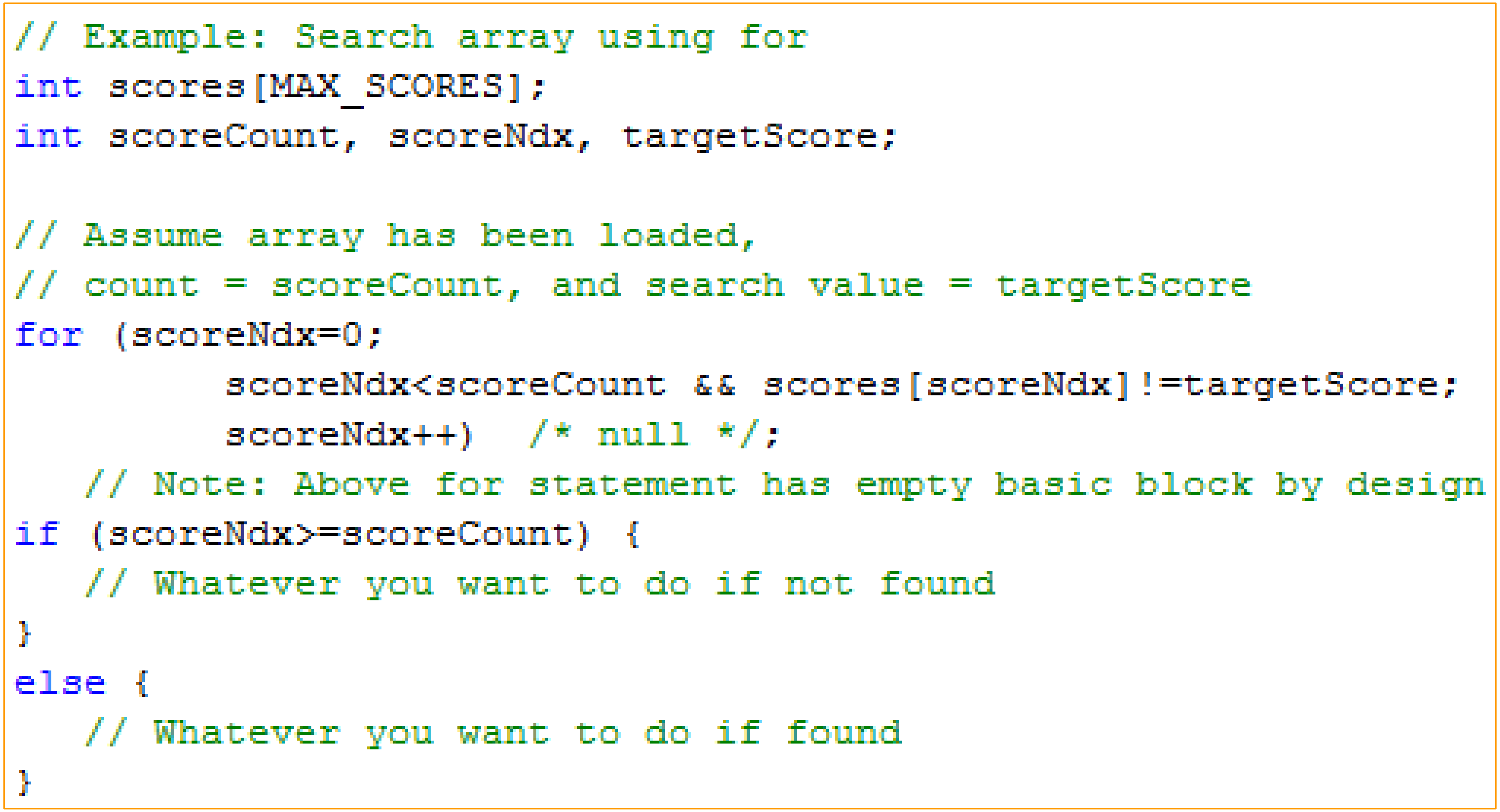
```
// Target found indicator
int found = 0;
```

Linear Search Example Using While



```
// Example: Search array using while
int scores[MAX_SCORES];
int scoreCount, scoreNdx, targetScore;

// Assume array has been loaded,
// count = scoreCount, and search value = targetScore
scoreNdx = 0;
while (scoreNdx<scoreCount && scores[scoreNdx]!=targetScore)
    scoreNdx++;
if (scoreNdx>=scoreCount) {
    // Whatever you want to do if not found
}
else {
    // Whatever you want to do if found
}
```

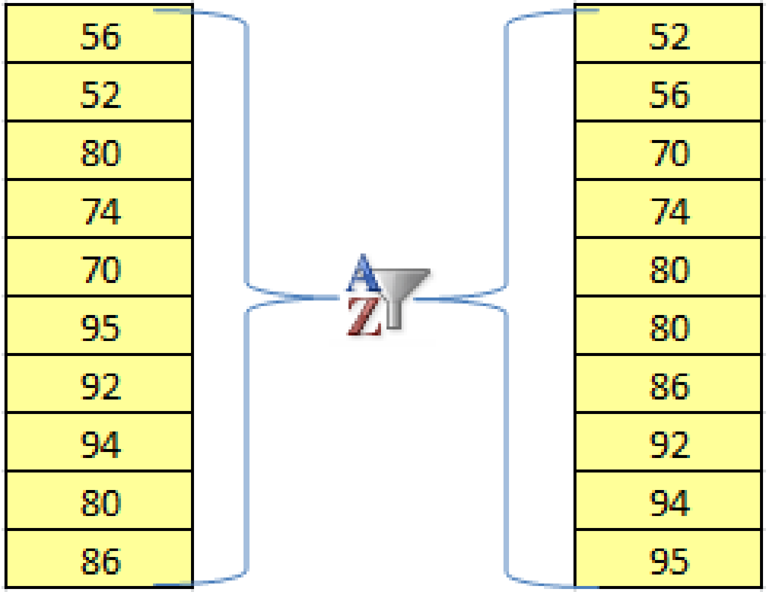
Linear Search Example Using For



```
// Example: Search array using for
int scores[MAX_SCORES];
int scoreCount, scoreNdx, targetScore;

// Assume array has been loaded,
// count = scoreCount, and search value = targetScore
for (scoreNdx=0;
         scoreNdx<scoreCount && scores[scoreNdx]!=targetScore;
         scoreNdx++)  /* null */;
   // Note: Above for statement has empty basic block by design
if (scoreNdx>=scoreCount) {
   // Whatever you want to do if not found
}
else {
   // Whatever you want to do if found
}
```

Sorting

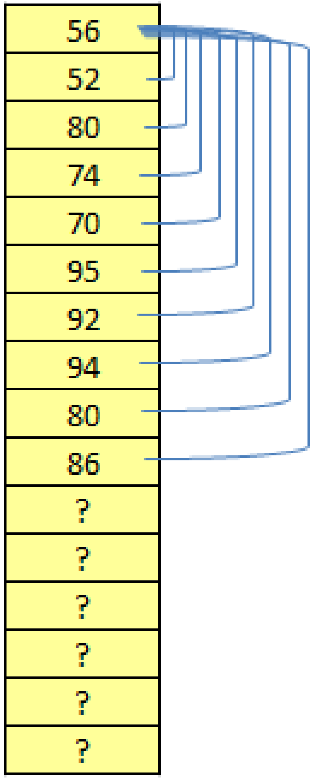
- Place array into some order
  - Ascending or descending
- Many types
  - Simple: Selection
  - More intelligent: Bubble, selection, insertion, shell, comb, merge, heap, quick, counting, bucket, radix, distribution, timsort, gnome, cocktail, library, cycle, binary tree, bogo, pigeonhole, spread, bead, pancake, …

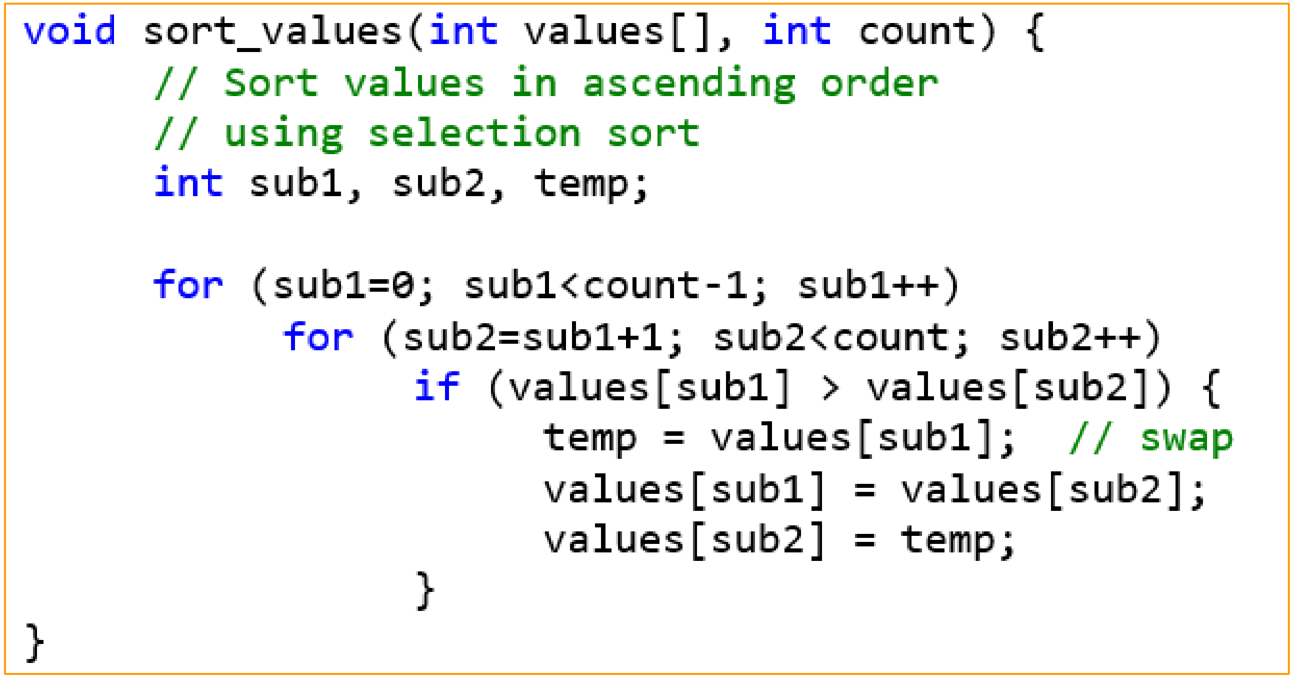


56	52
52	56
80	70
74	74
70	80
95	80
92	86
94	92
80	94
86	95

Selection Sort

- Compare element to all elements below and then move to next element, swap when appropriate

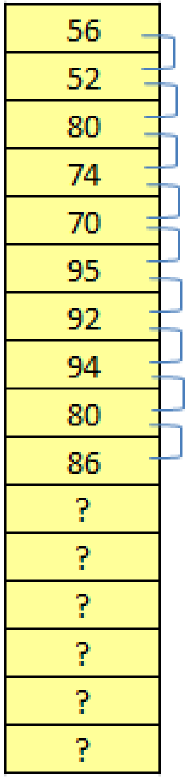




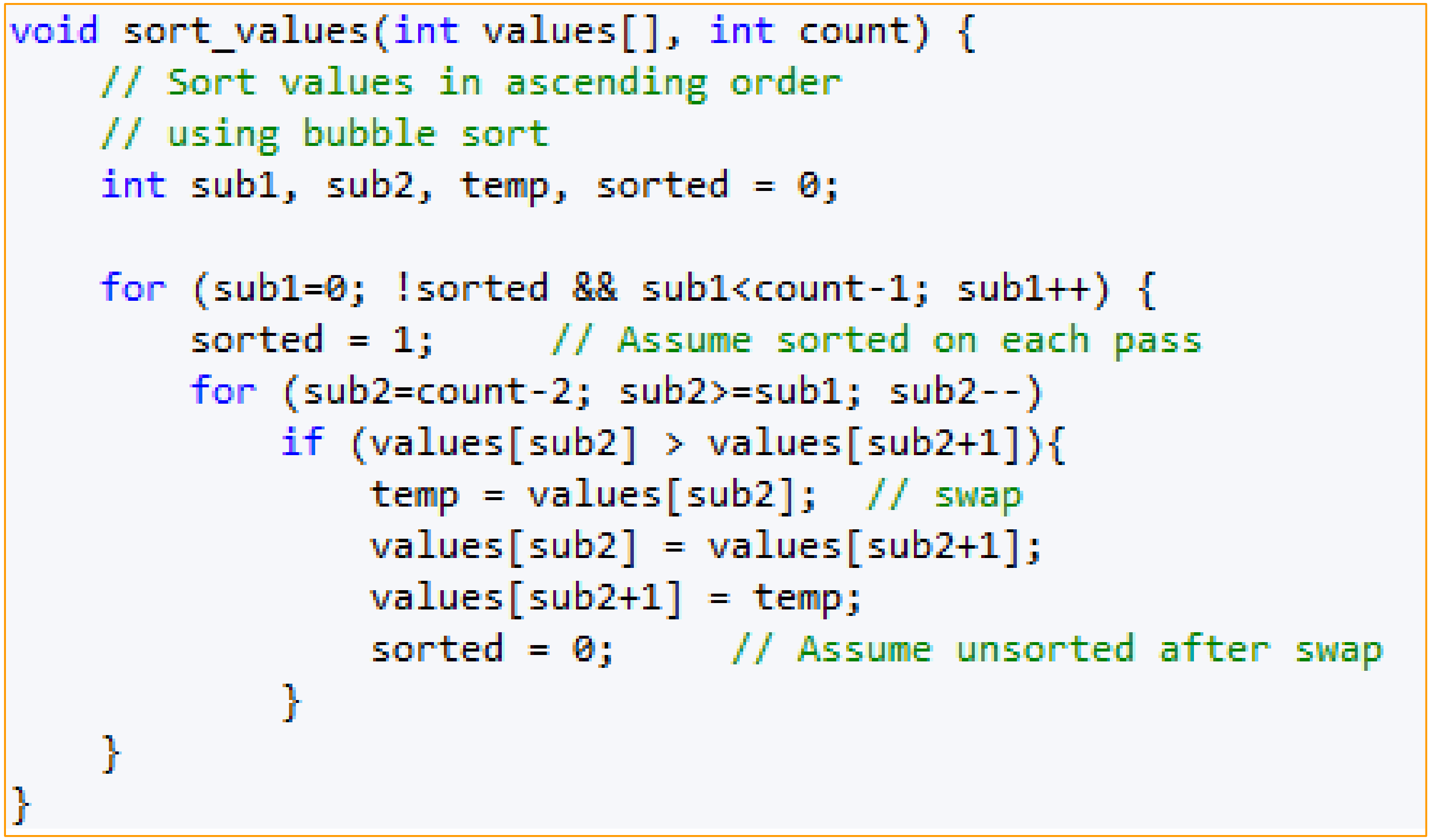
```
void sort_values(int values[], int count) {
        // Sort values in ascending order
        // using selection sort
        int sub1, sub2, temp;

        for (sub1=0; sub1<count-1; sub1++)
                for (sub2=sub1+1; sub2<count; sub2++)
                        if (values[sub1] > values[sub2]) {
                                temp = values[sub1];  // swap
                                values[sub1] = values[sub2];
                                values[sub2] = temp;
                        }
}
```

Bubble/Sinking Sort

- Compare adjacent elements, swap when appropriate
- Stop if no swaps on a pass



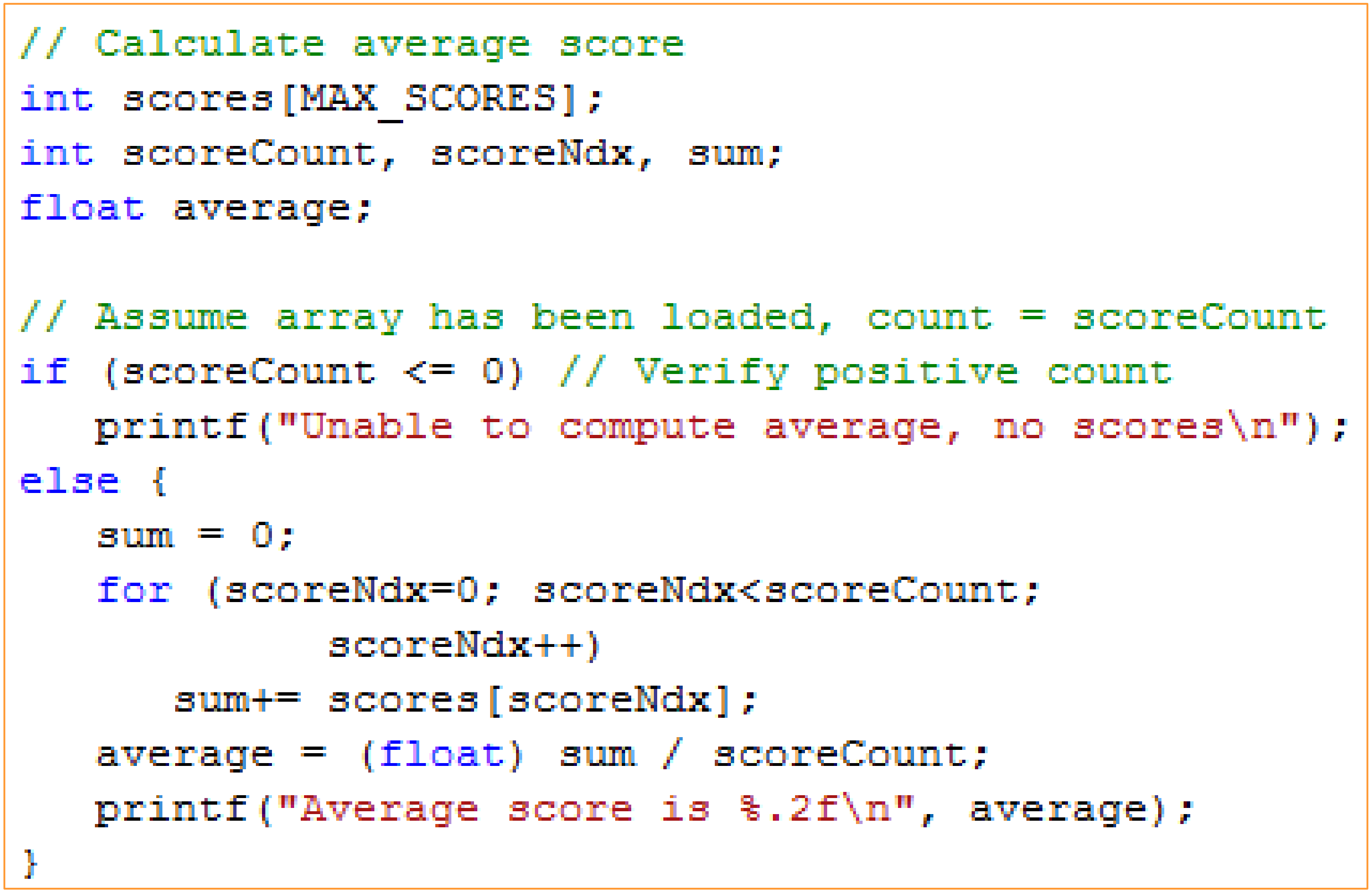
56
52
80
74
70
95
92
94
80
86
?
?
?
?
?
?



```c
void sort_values(int values[], int count) {
    // Sort values in ascending order
    // using bubble sort
    int sub1, sub2, temp, sorted = 0;

    for (sub1=0; !sorted && sub1<count-1; sub1++) {
        sorted = 1;     // Assume sorted on each pass
        for (sub2=count-2; sub2>=sub1; sub2--)
            if (values[sub2] > values[sub2+1]){
                temp = values[sub2];  // swap
                values[sub2] = values[sub2+1];
                values[sub2+1] = temp;
                sorted = 0;     // Assume unsorted after swap
            }
    }
}
```

Sum & Average Example

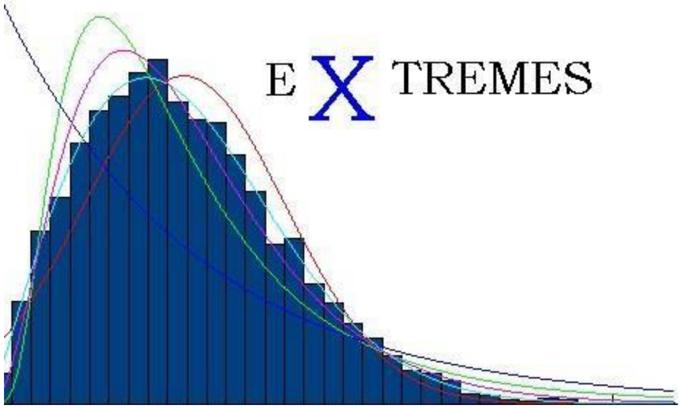
- Verify positive count before computing average
  - Protects against division by zero



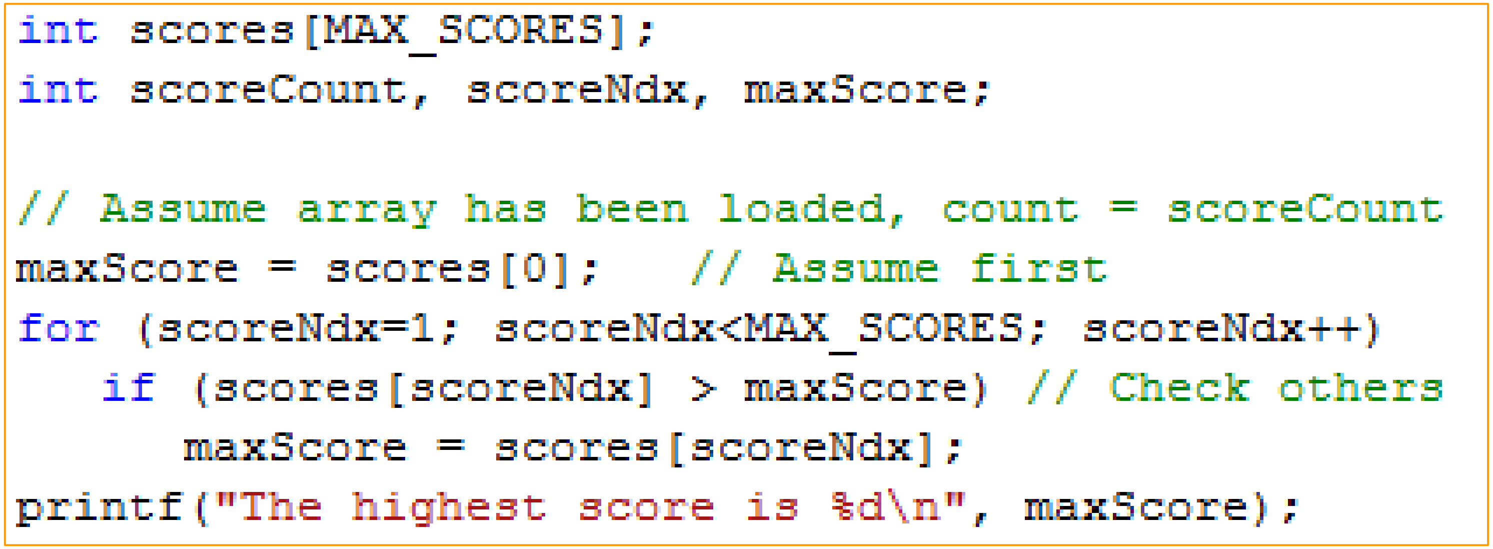
```
// Calculate average score
int scores[MAX_SCORES];
int scoreCount, scoreNdx, sum;
float average;

// Assume array has been loaded, count = scoreCount
if (scoreCount <= 0) // Verify positive count
   printf("Unable to compute average, no scores\n");
else {
   sum = 0;
   for (scoreNdx=0; scoreNdx<scoreCount;
            scoreNdx++)
      sum+= scores[scoreNdx];
   average = (float) sum / scoreCount;
   printf("Average score is %.2f\n", average);
}
```

Extremes

- Same techniques as chapter 5 – best:
  - Assume first is extreme
  - Compare others to current extreme
  - Replace extreme when finding new extreme



Extremes: Find Maximum Example



```
int scores[MAX_SCORES];
int scoreCount, scoreNdx, maxScore;

// Assume array has been loaded, count = scoreCount
maxScore = scores[0];   // Assume first
for (scoreNdx=1; scoreNdx<MAX_SCORES; scoreNdx++)
   if (scores[scoreNdx] > maxScore) // Check others
      maxScore = scores[scoreNdx];
printf("The highest score is %d\n", maxScore);
```

Programming in C

Array Subtasks

THE END