

# Chapter 8

## *Arrays*

# *OBJECTIVES*

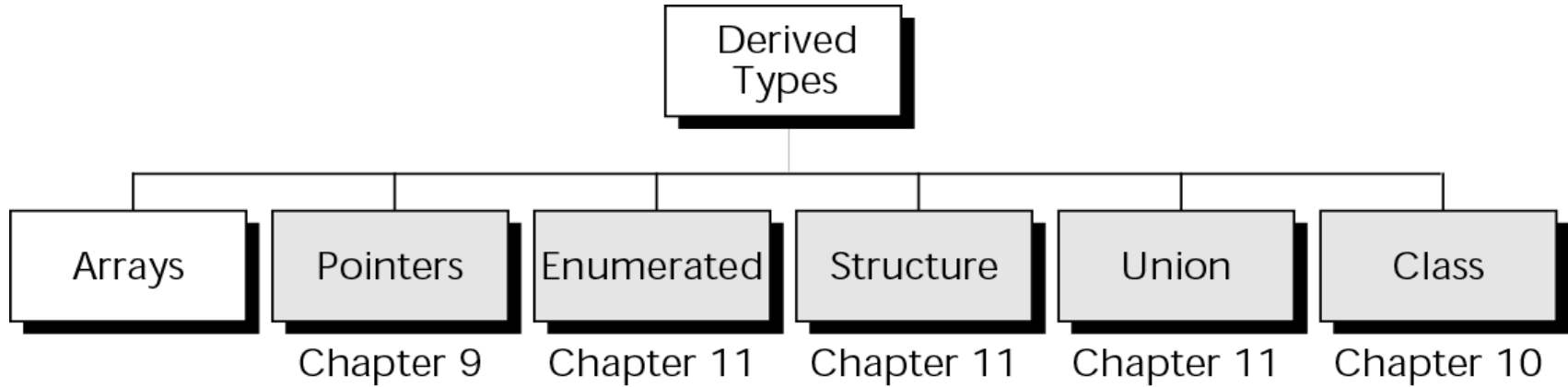
---

*After studying this chapter you will be able to:*

- ☐ Understand and use one, two, and three dimensional arrays in C++.
- ☐ Understand how to initialize arrays.
- ☐ Read data into an array or write data from an array.
- ☐ Understand that array range checking is the responsibility of the programmer.
- ☐ Write programs that pass arrays or array elements to functions.
- ☐ Sort data in an array using selection, bubble, or insertion sorting.
- ☐ Search an array using sequential or binary searches.
- ☐ Analyze the efficiency of sorting and searching algorithms.

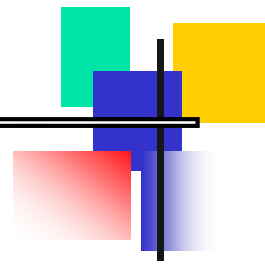


## Figure 8-1 Derived types



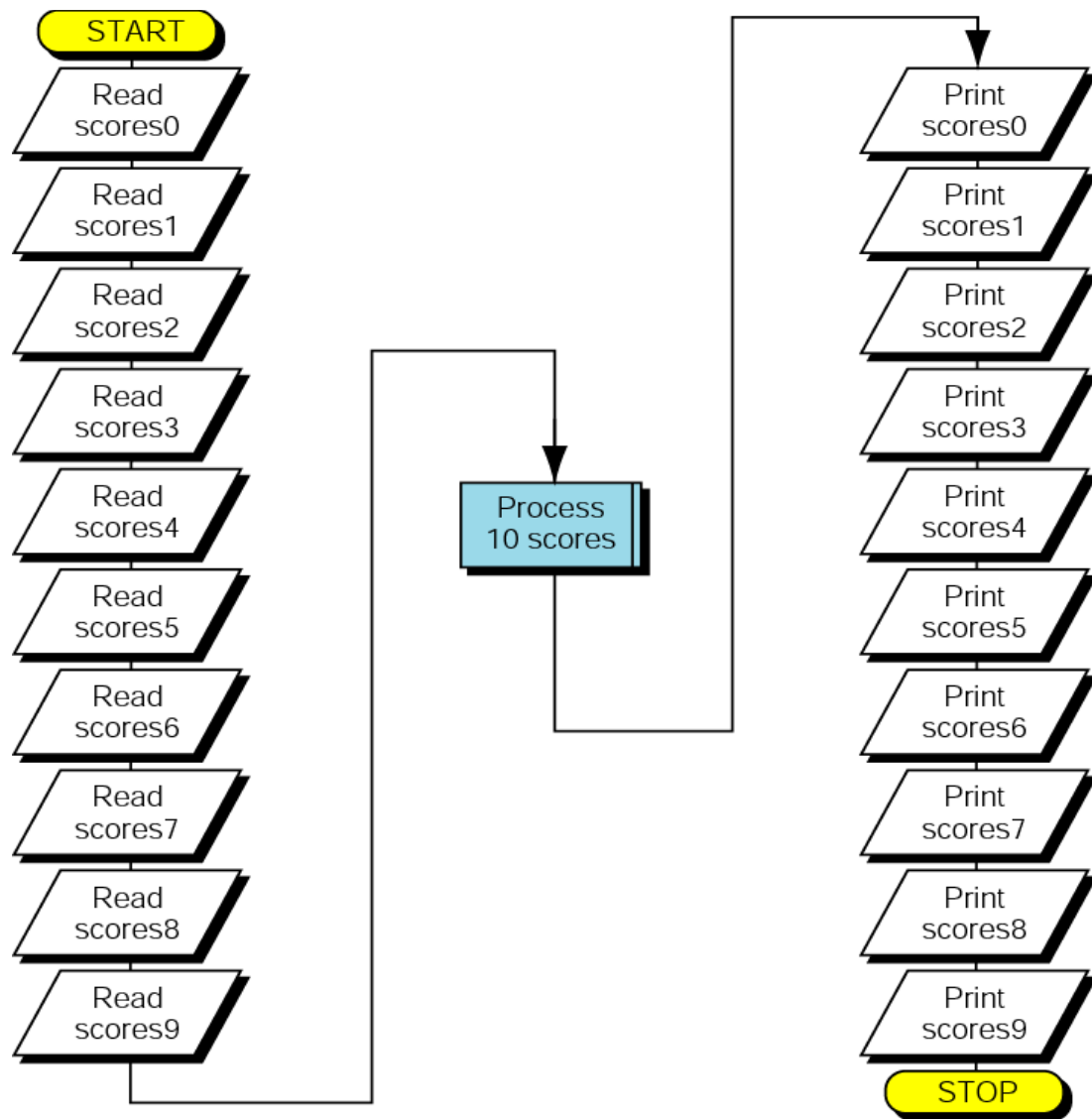
## CONCEPTS

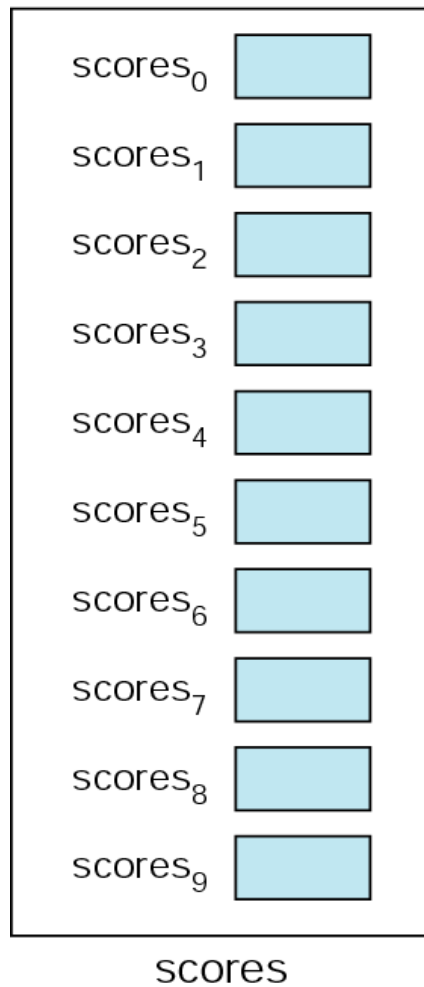
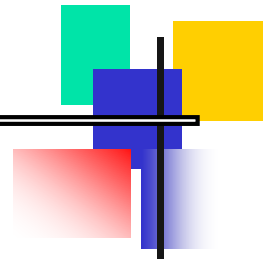
## Figure 8-2 Ten variables



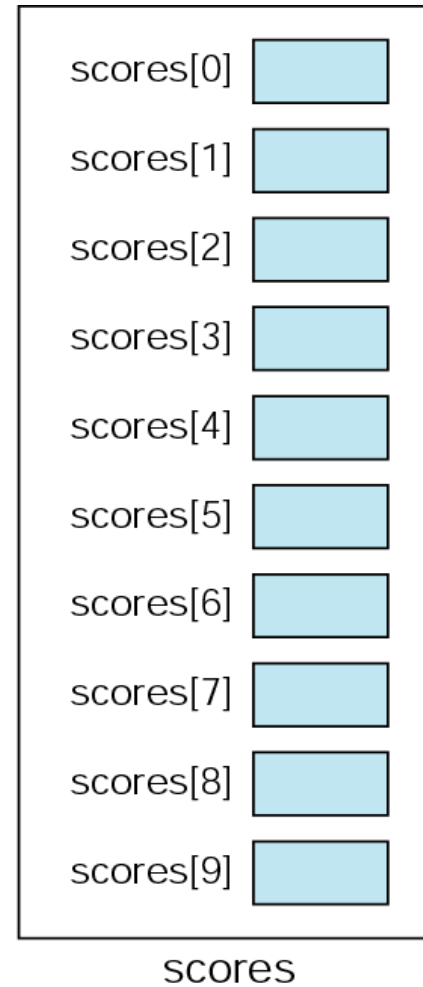
score0	<input type="text"/>	score5	<input type="text"/>
score1	<input type="text"/>	score6	<input type="text"/>
score2	<input type="text"/>	score7	<input type="text"/>
score3	<input type="text"/>	score8	<input type="text"/>
score4	<input type="text"/>	score9	<input type="text"/>

# Figure 8-3 Process 10 variables



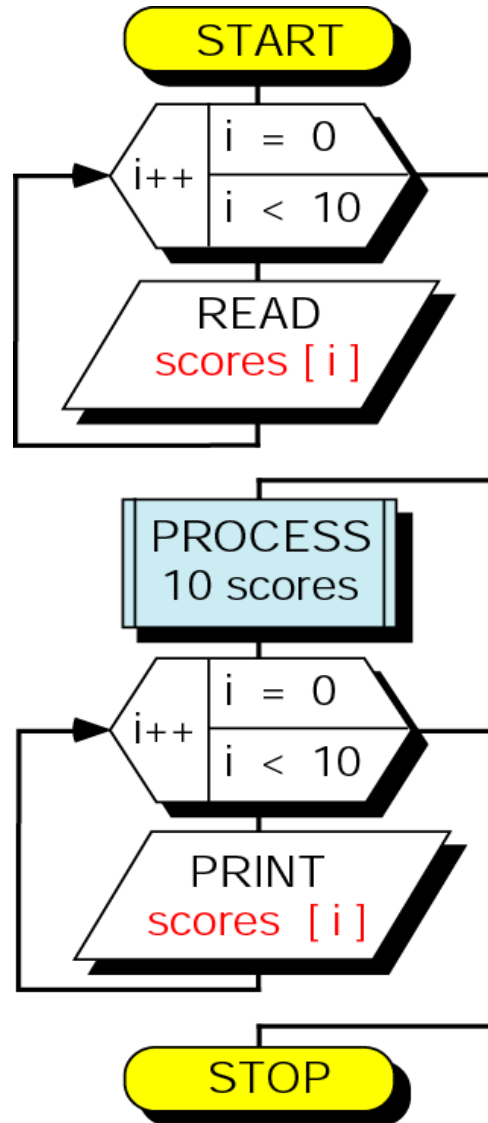


(a) Subscript format



(b) Index format

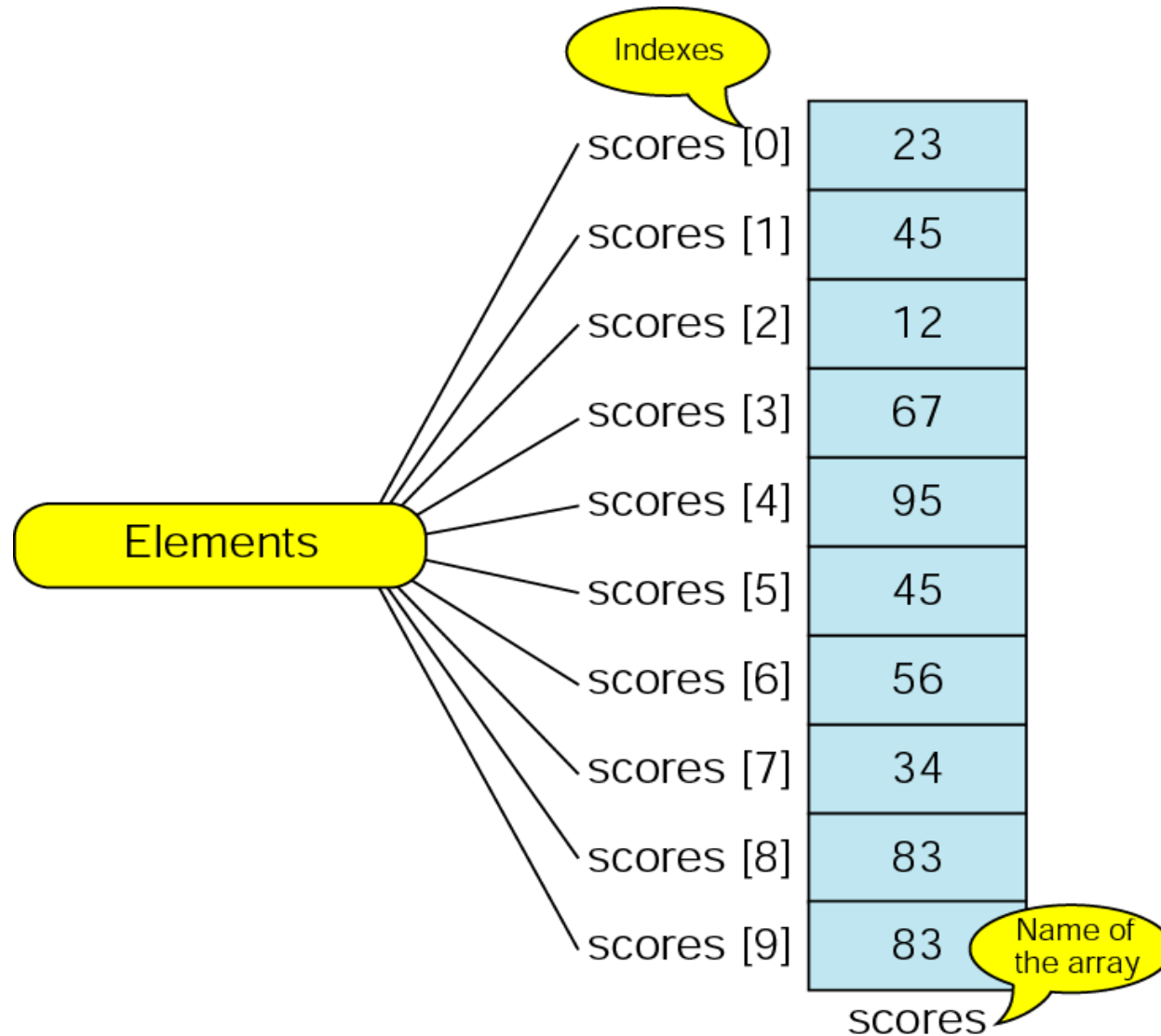
## Figure 8-5 Loop for 10 scores





## USING ARRAYS IN C++

## Figure 8-6 The scores array



## Figure 8-7 Declaring and defining arrays

int scores [9];

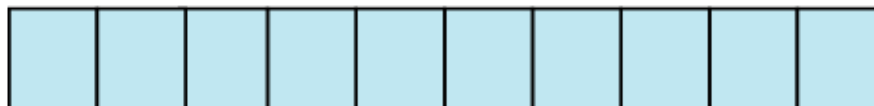


[0] [1] [2] [3] [4] [5] [6] [7] [8]

*scores*

type of each  
element

char name [10];

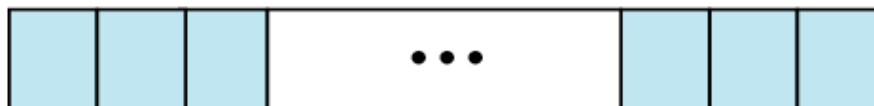


[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]

*name*

name of  
the array

float gpa [40];



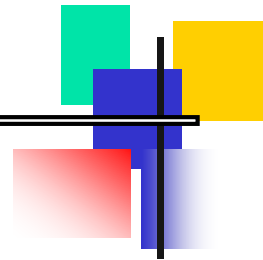
[0] [1] [2] ... [37] [38] [39]

*gpa*

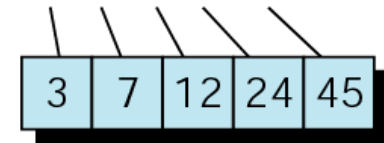
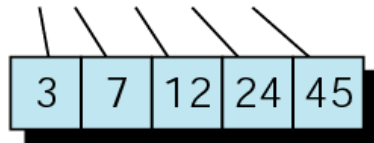
number of  
elements



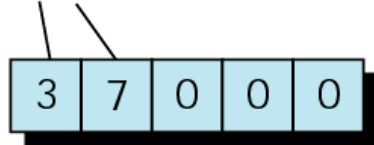
## Figure 8-8 Initializing arrays



`int numbers [5] = { 3,7,12,24,45 } ;`    `int numbers [ ] = { 3,7,12,24,45 } ;`



`int numbers [5] = { 3,7 } ;`



The rest are  
filled with 0s

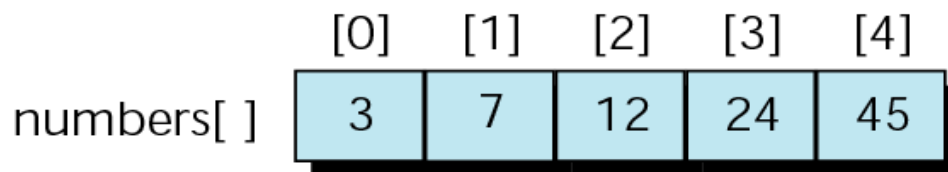
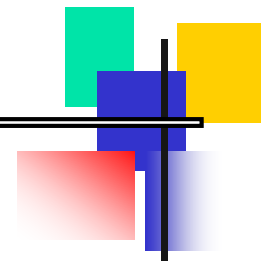
`int lotsOfNumbers [1000] = { 0 } ;`



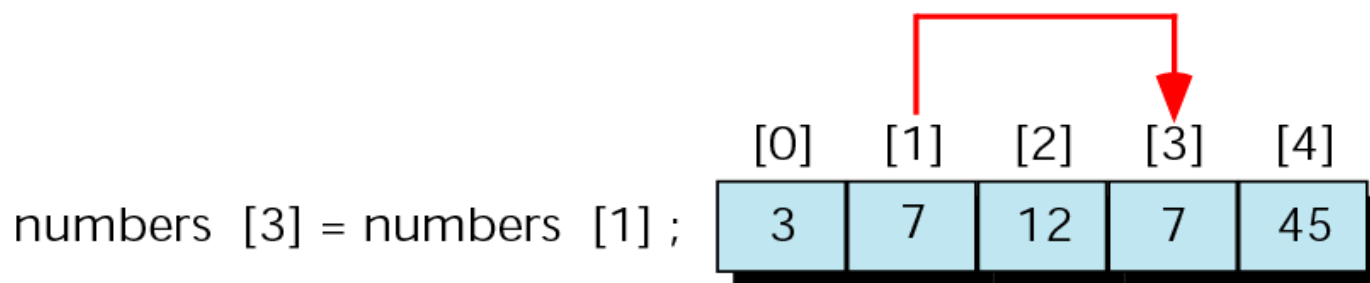
All filled with 0s

**Figure 8-9**

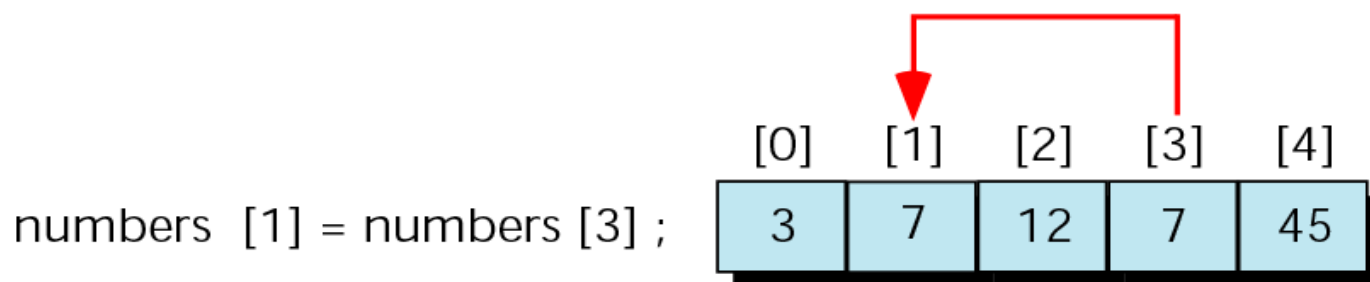
## Exchanging scores—the wrong way



Before



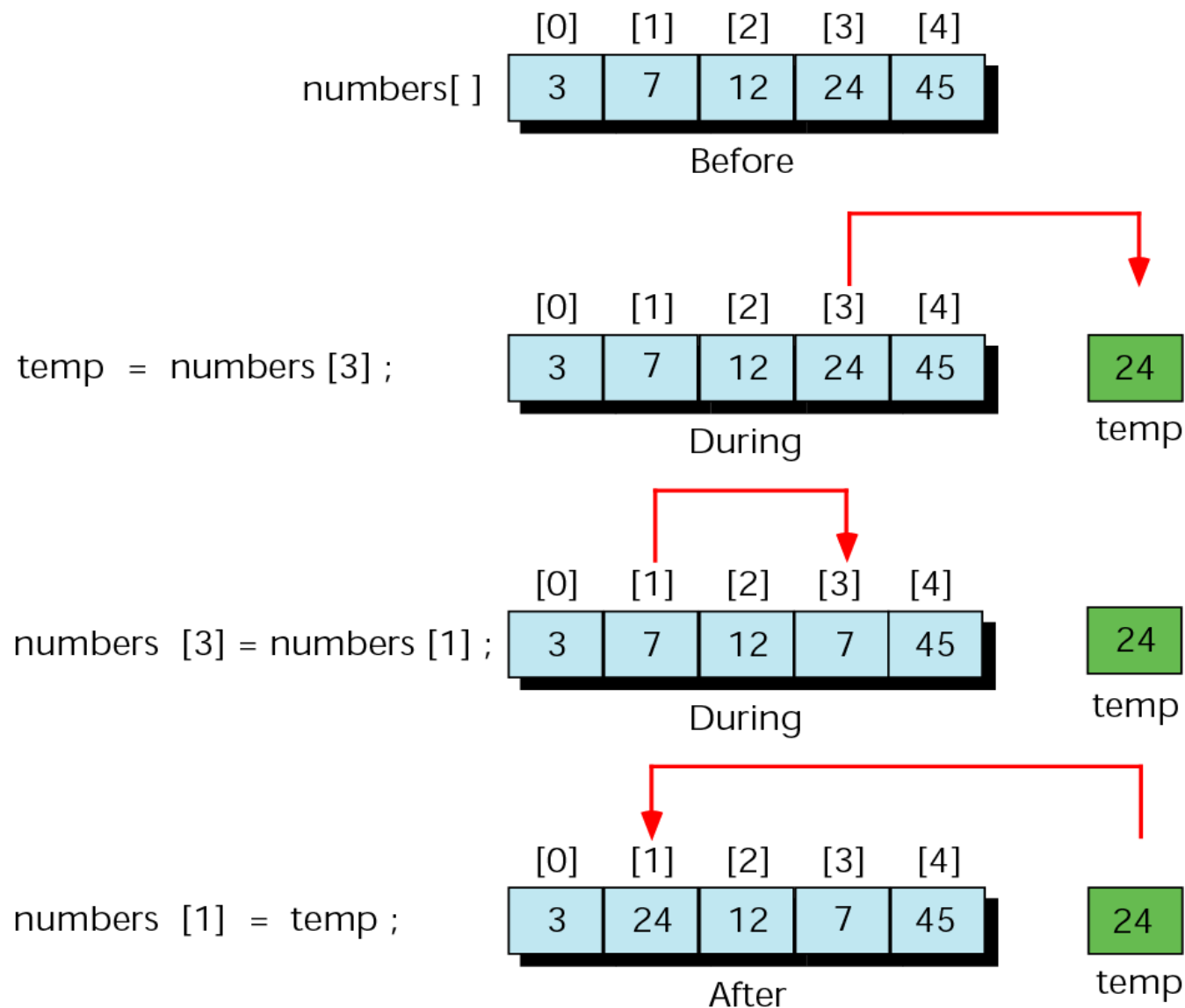
During



After

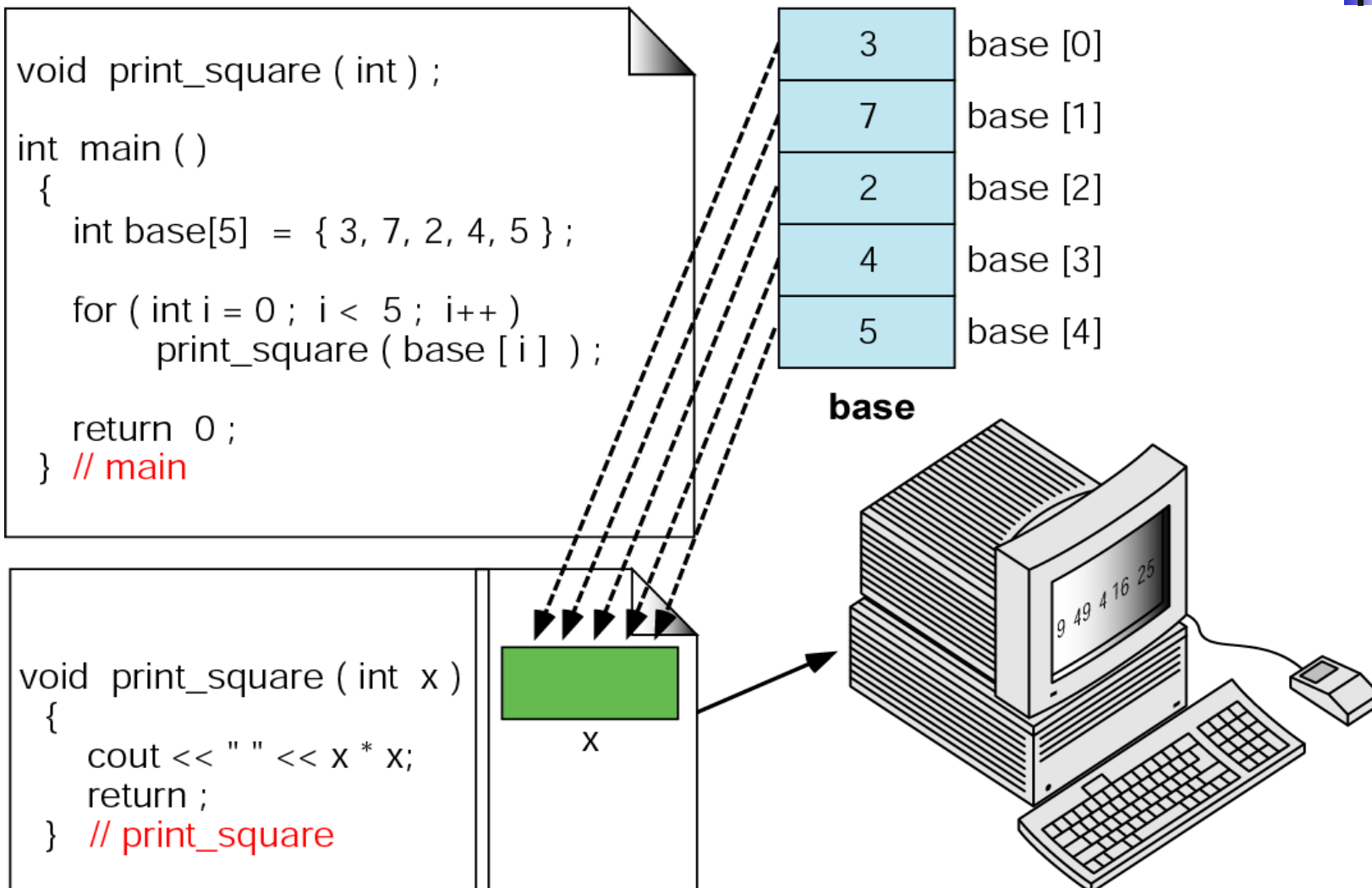
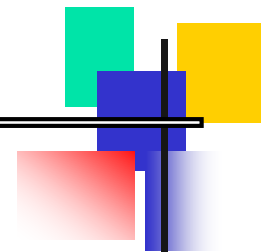


## Figure 8-10 Exchanging scores with temporary variable



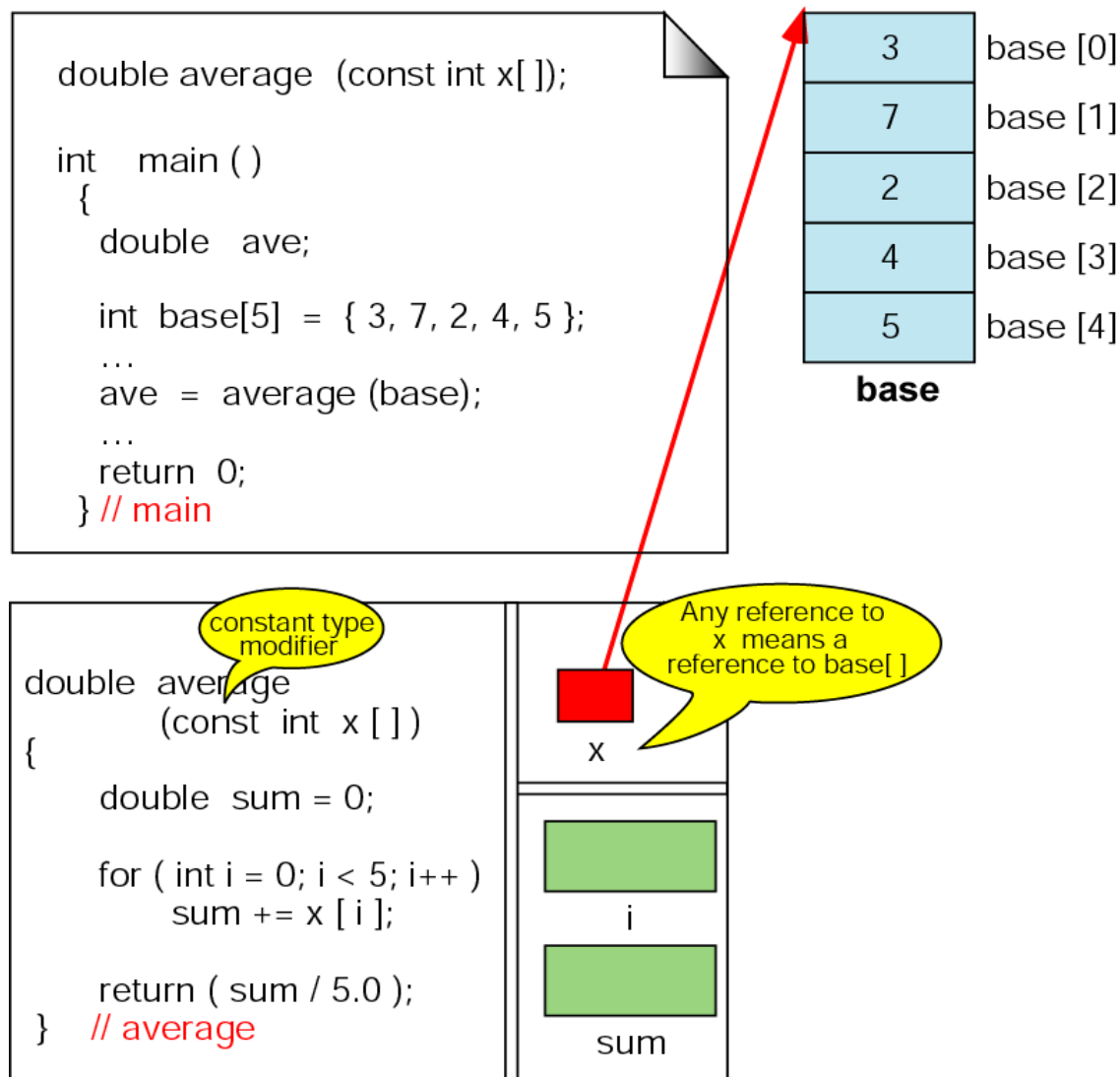
## ARRAYS AND FUNCTIONS

## Figure 8-11 Passing individual elements

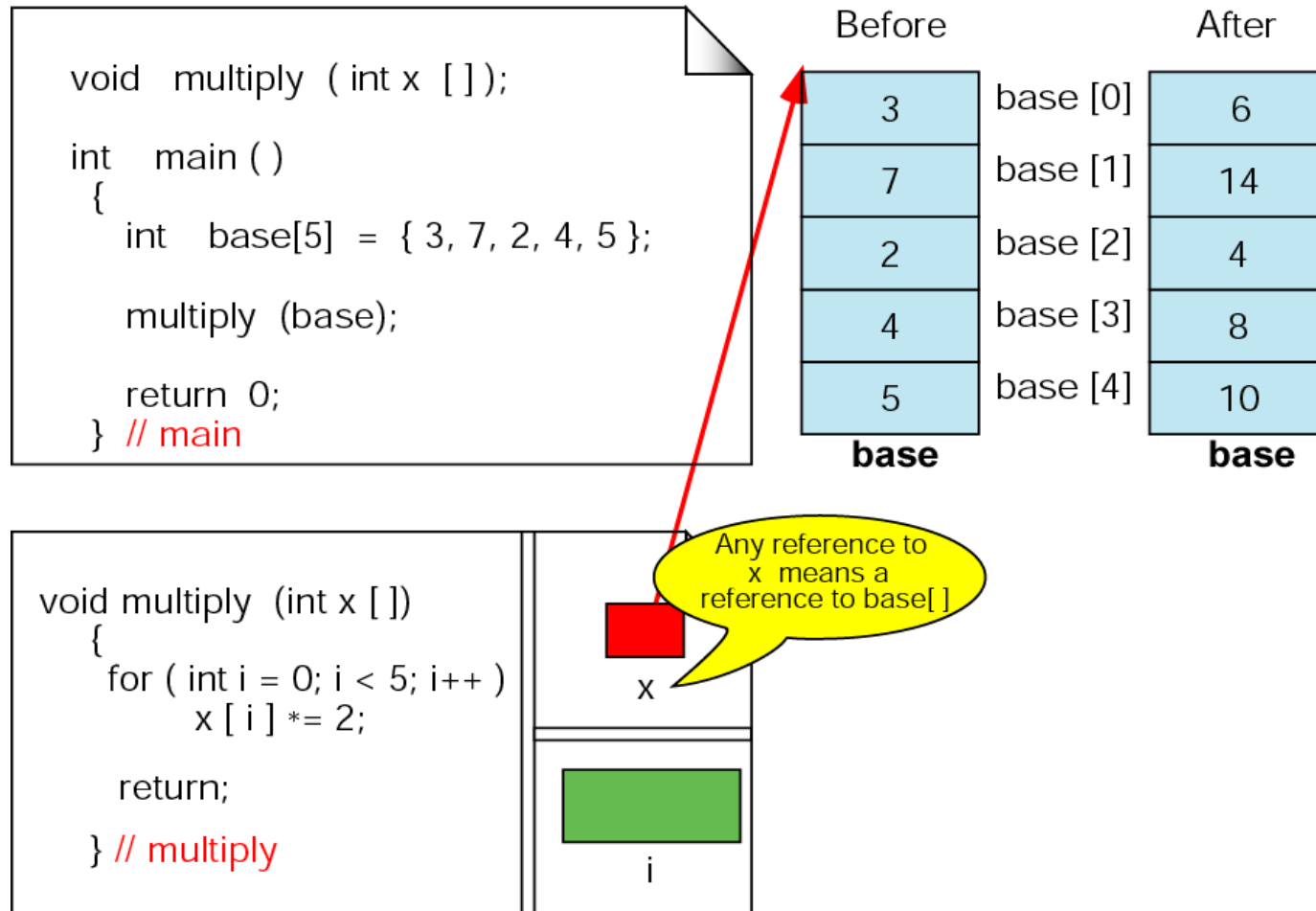




## Figure 8-12 Passing arrays—average

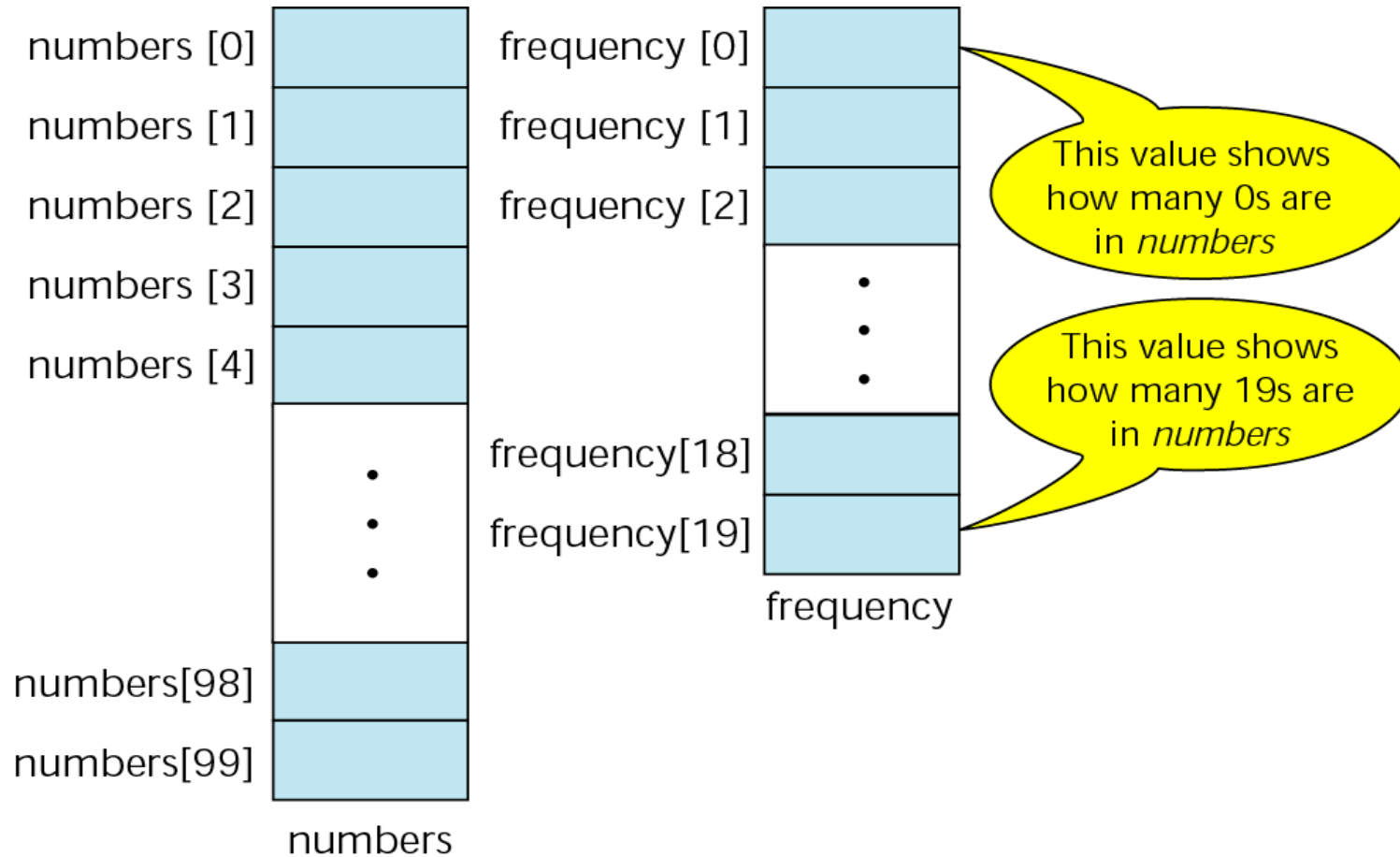
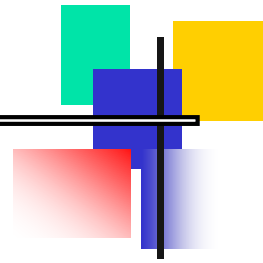


## Figure 8-13 Changing values in arrays

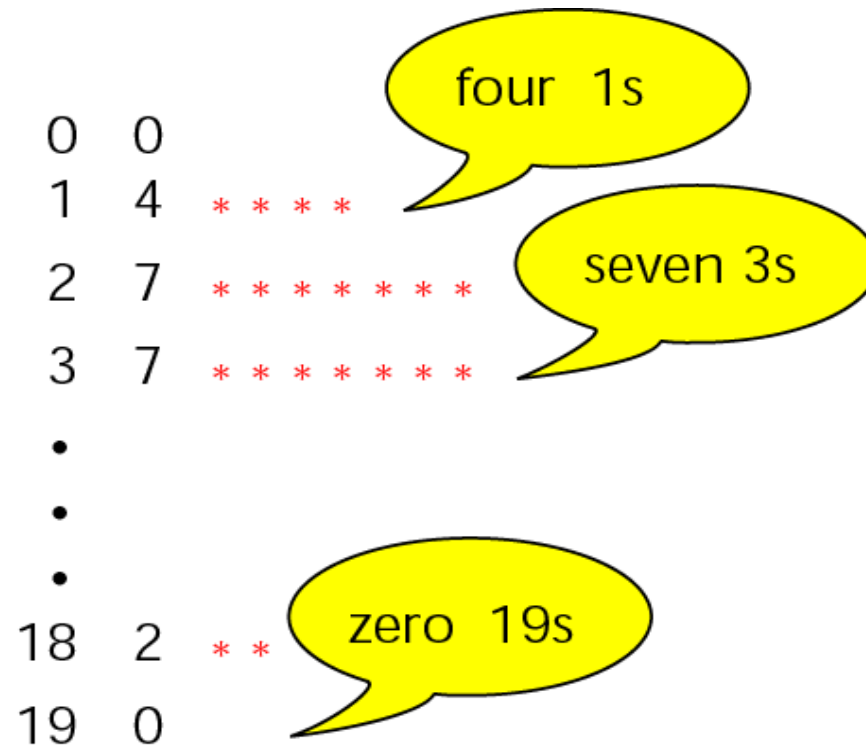


## ARRAY APPLICATIONS

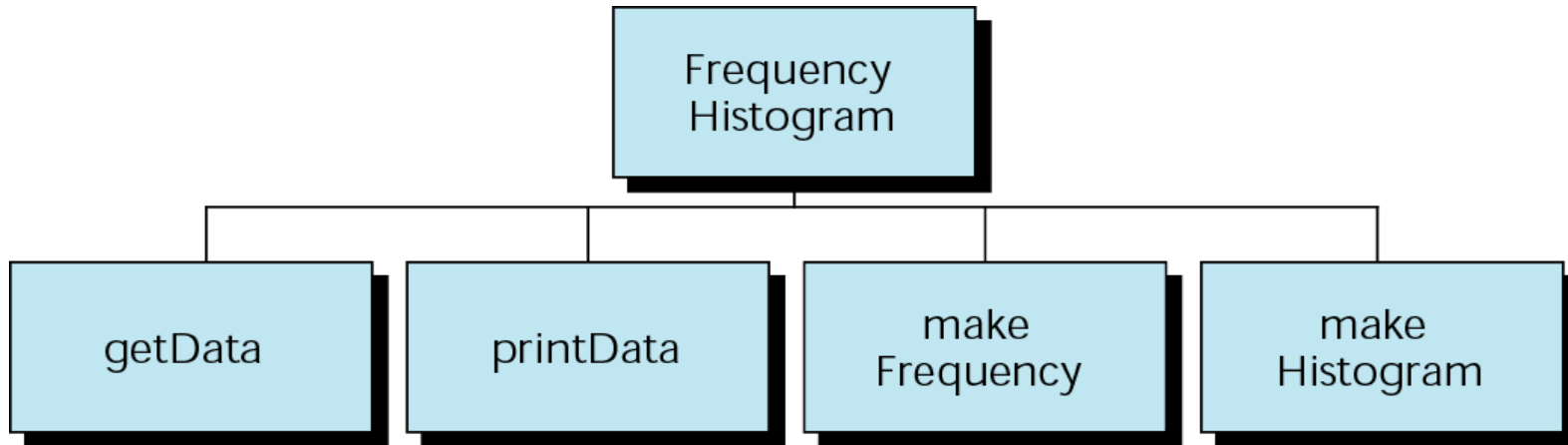
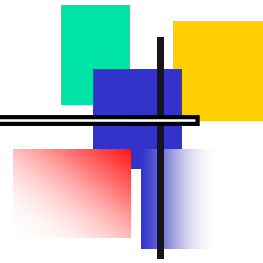
## Figure 8-14 Frequency array



## Figure 8-15 Frequency histogram



## Figure 8-16 Histogram program design



## Figure 8-17 Design for random number permutations

randNos[0]	8	haveRand[0]	0
randNos[1]	3	haveRand[1]	1
randNos[2]	5	haveRand[2]	0
randNos[3]	1	haveRand[3]	1
randNos[4]	7	haveRand[4]	0
randNos[5]		haveRand[5]	1
randNos[6]		haveRand[6]	0
randNos[7]		haveRand[7]	1
randNos[8]		haveRand[8]	1
randNos[9]		haveRand[9]	0

randNos

haveRand

0 means random  
2 not generated

1 means random  
5 generated

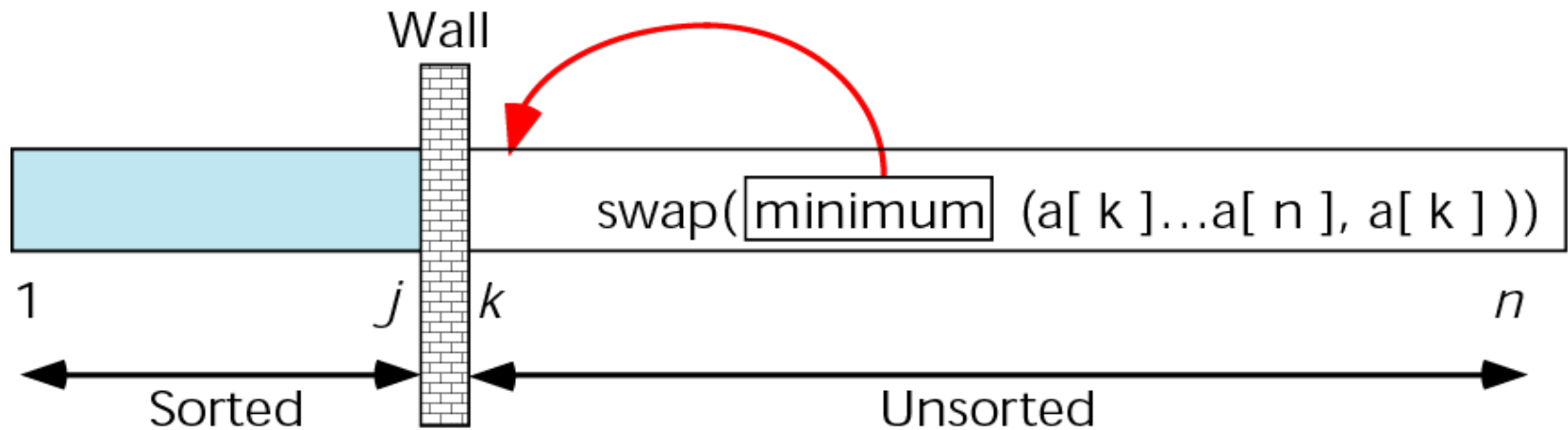
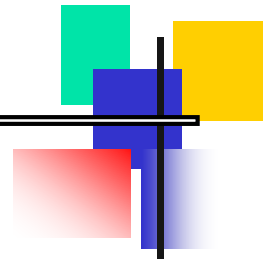
After first five random numbers generated.



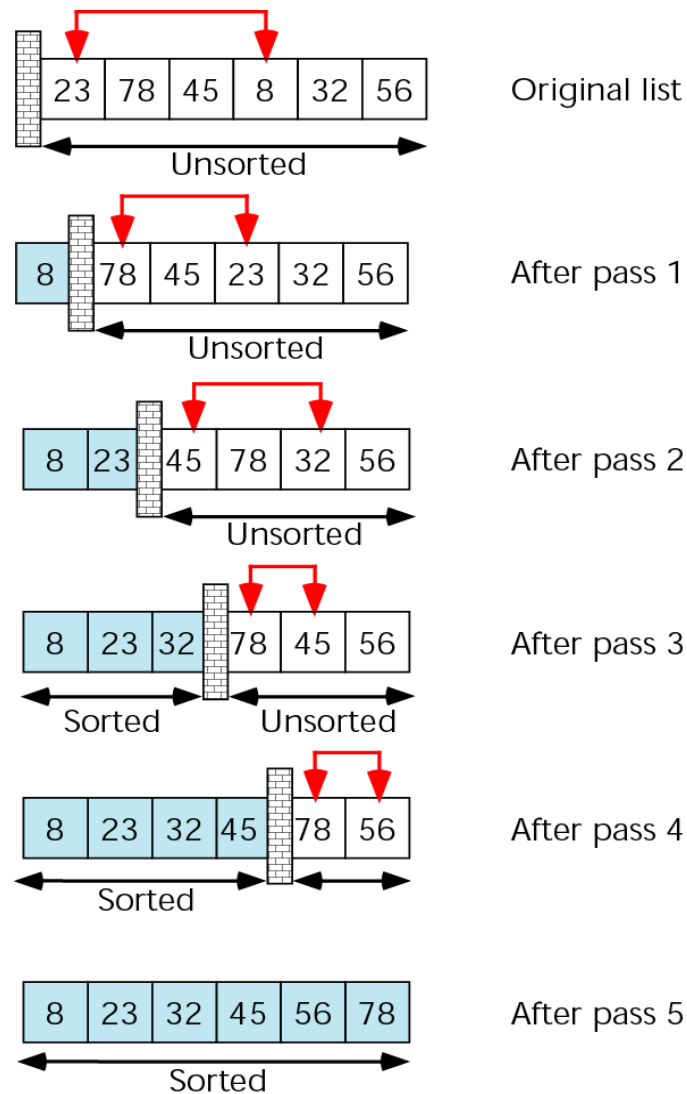
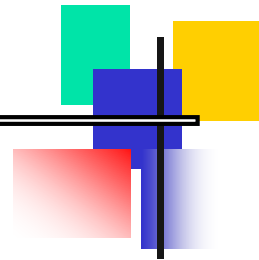
## SORTING



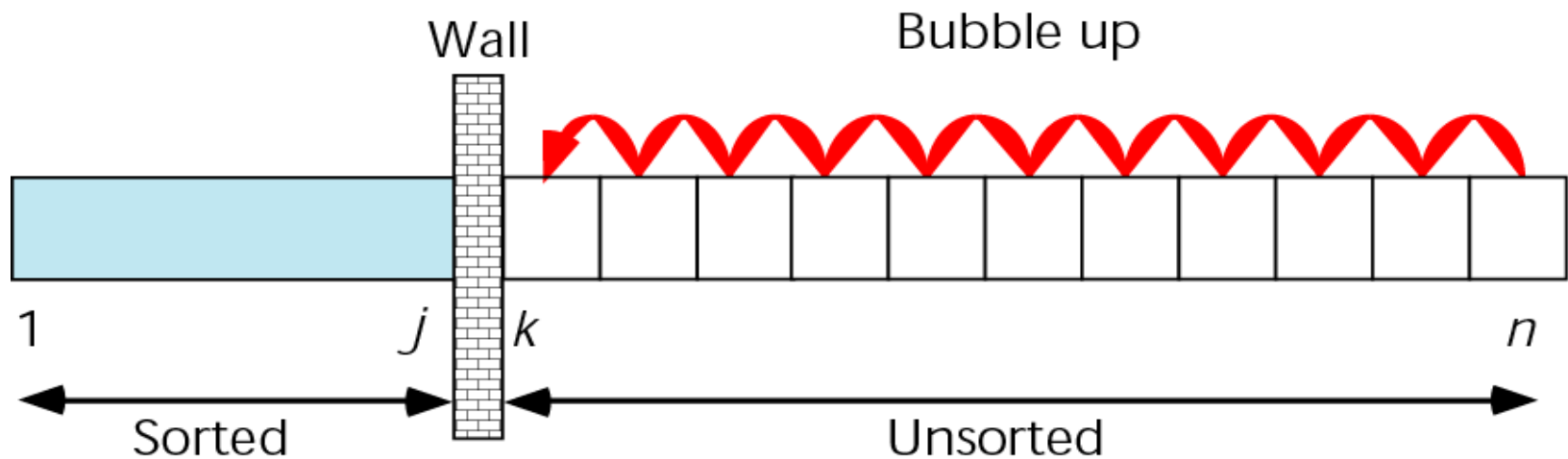
## Figure 8-18 Selection sort concept



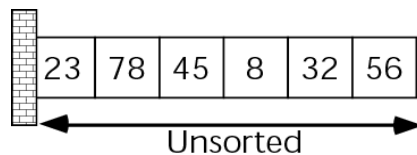
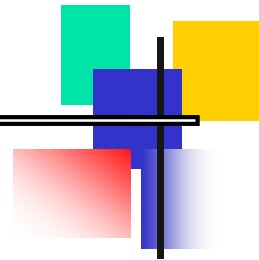
## Figure 8-19 Selection sort example



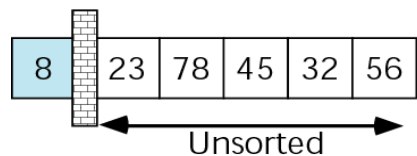
## Figure 8-20 Bubble sort concept



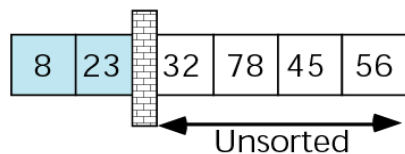
## Figure 8-21 Bubble sort example



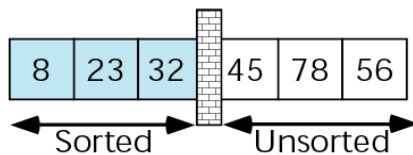
Original list



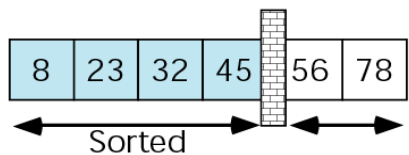
After pass 1



After pass 2



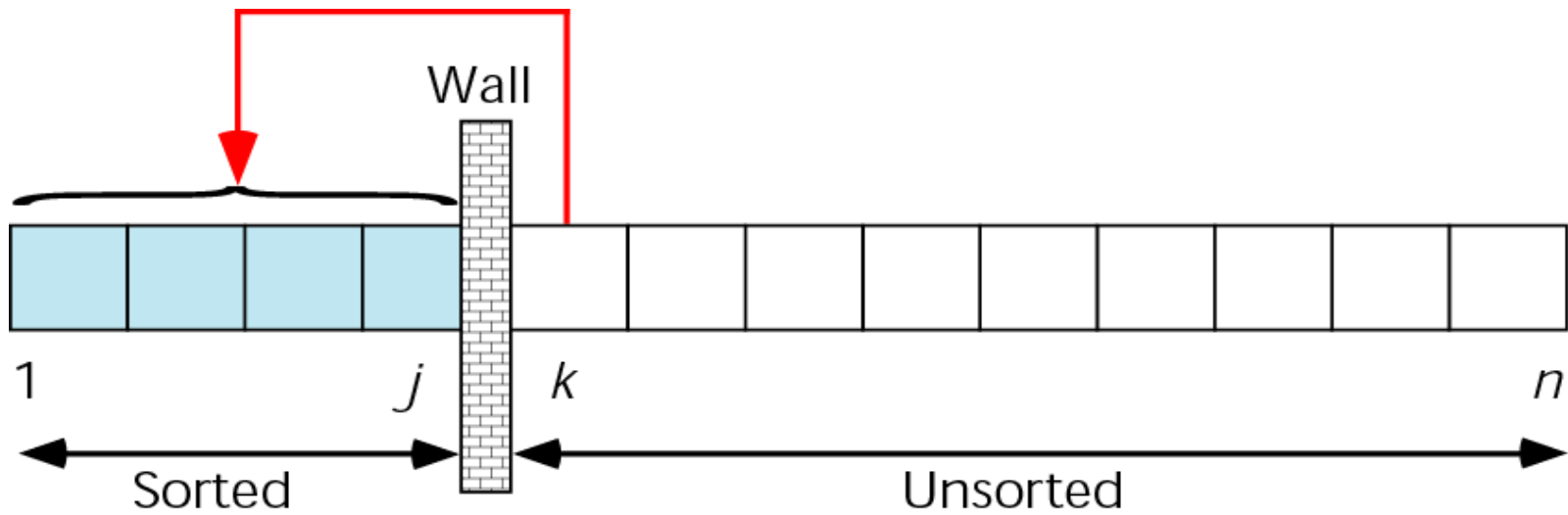
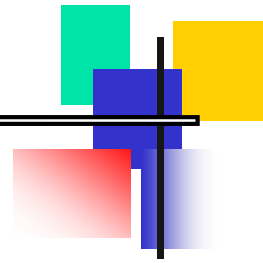
After pass 3



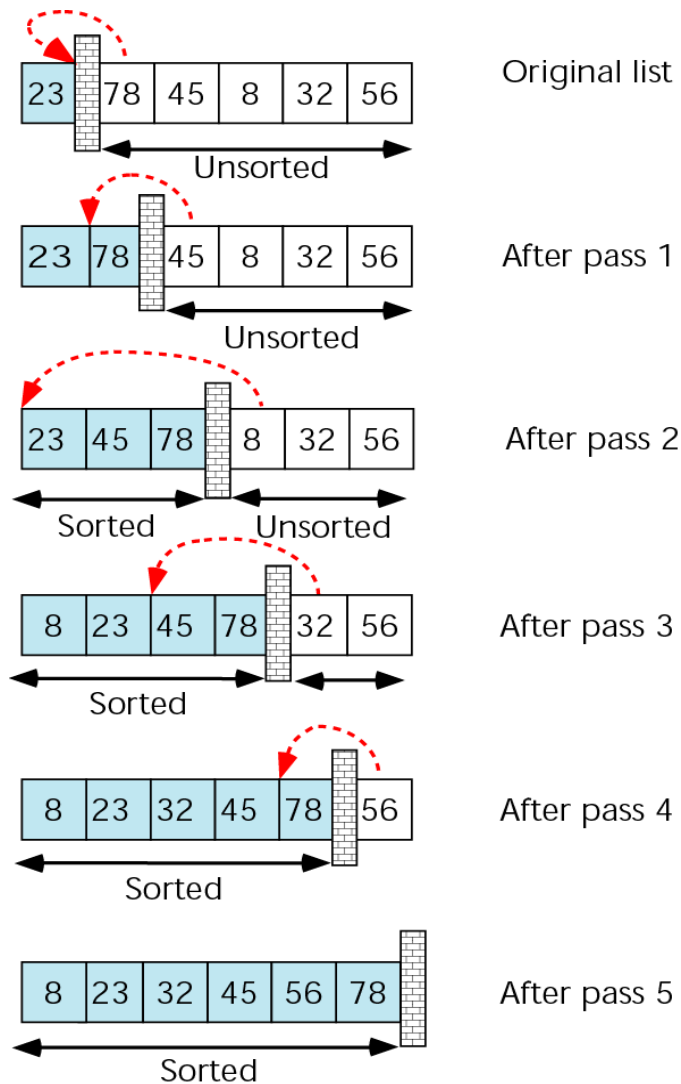
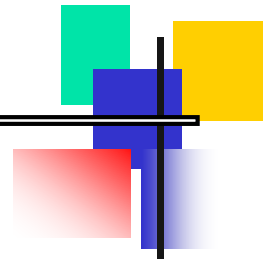
After pass 4  
Sorted!



## Figure 8-22 Insertion sort concept

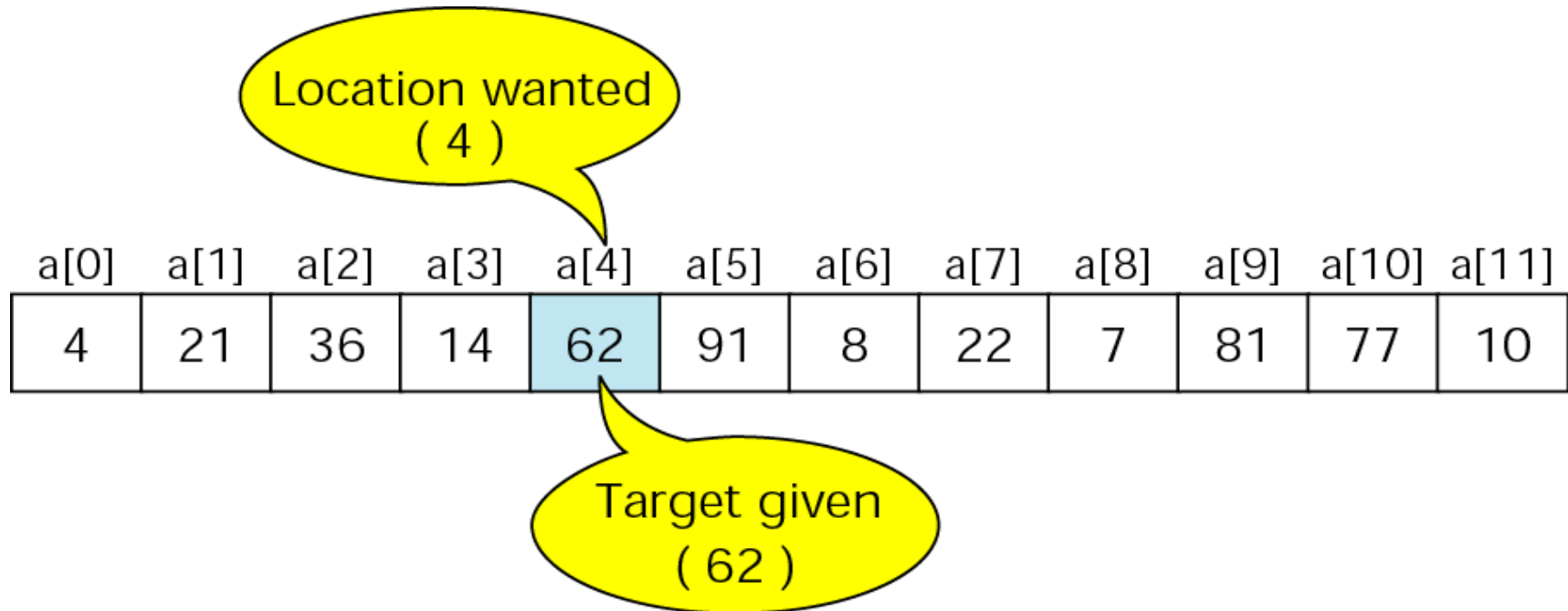
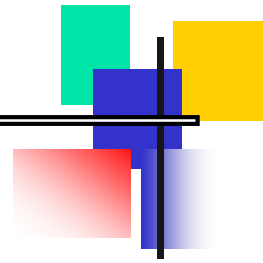


## Figure 8-23 Insertion sort example



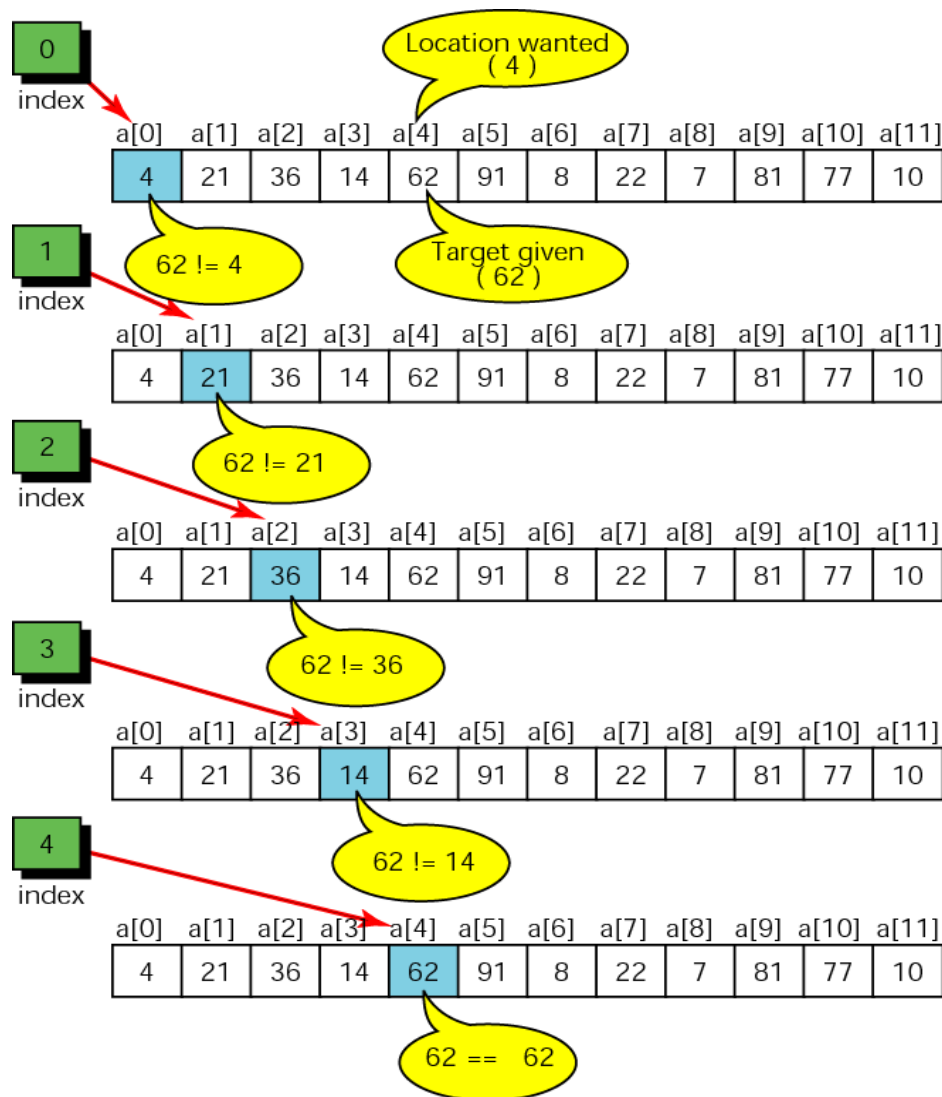
## SEARCHING

## Figure 8-24 Search concept

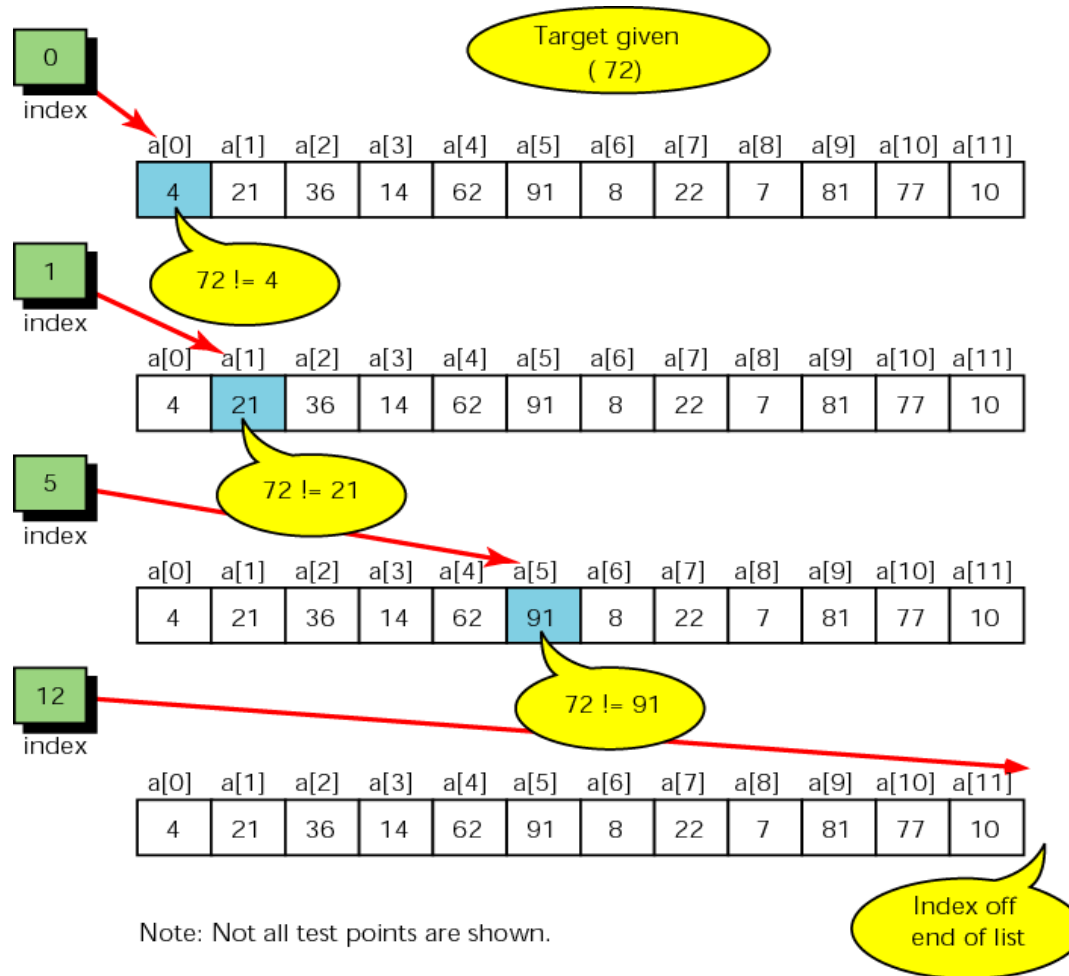




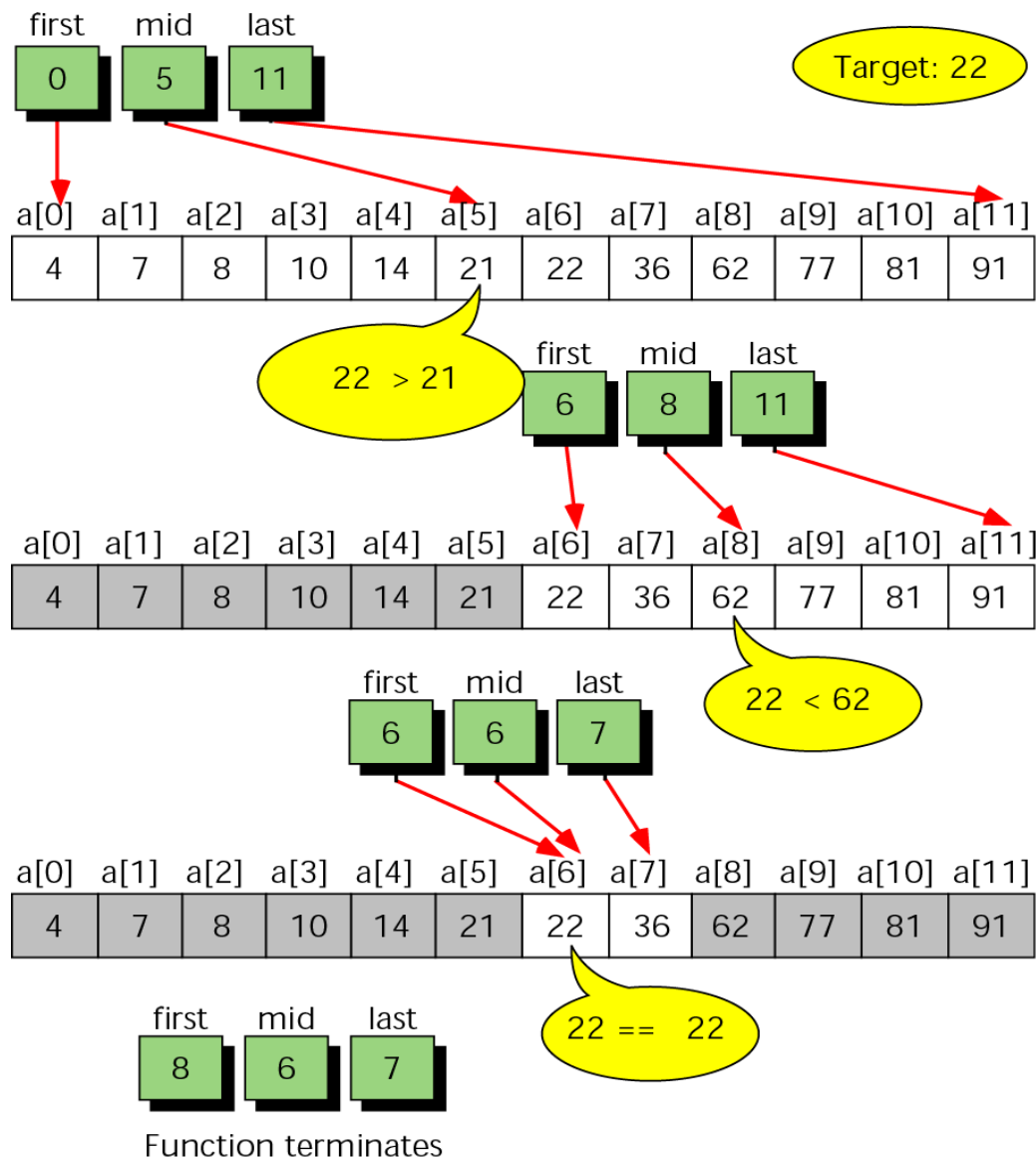
## Figure 8-25 Locating data in an unordered list



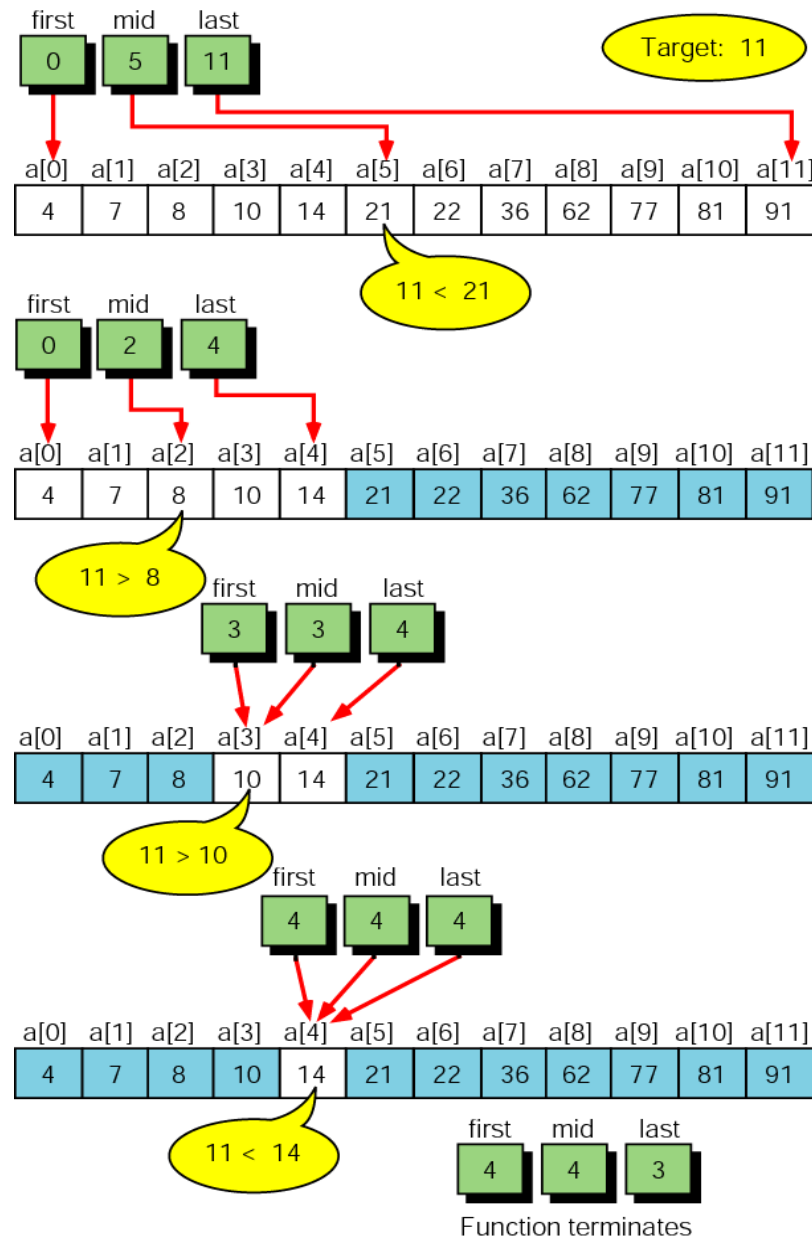
## Figure 8-26 Unsuccessful search in an unordered list



## Figure 8-27 Binary search example

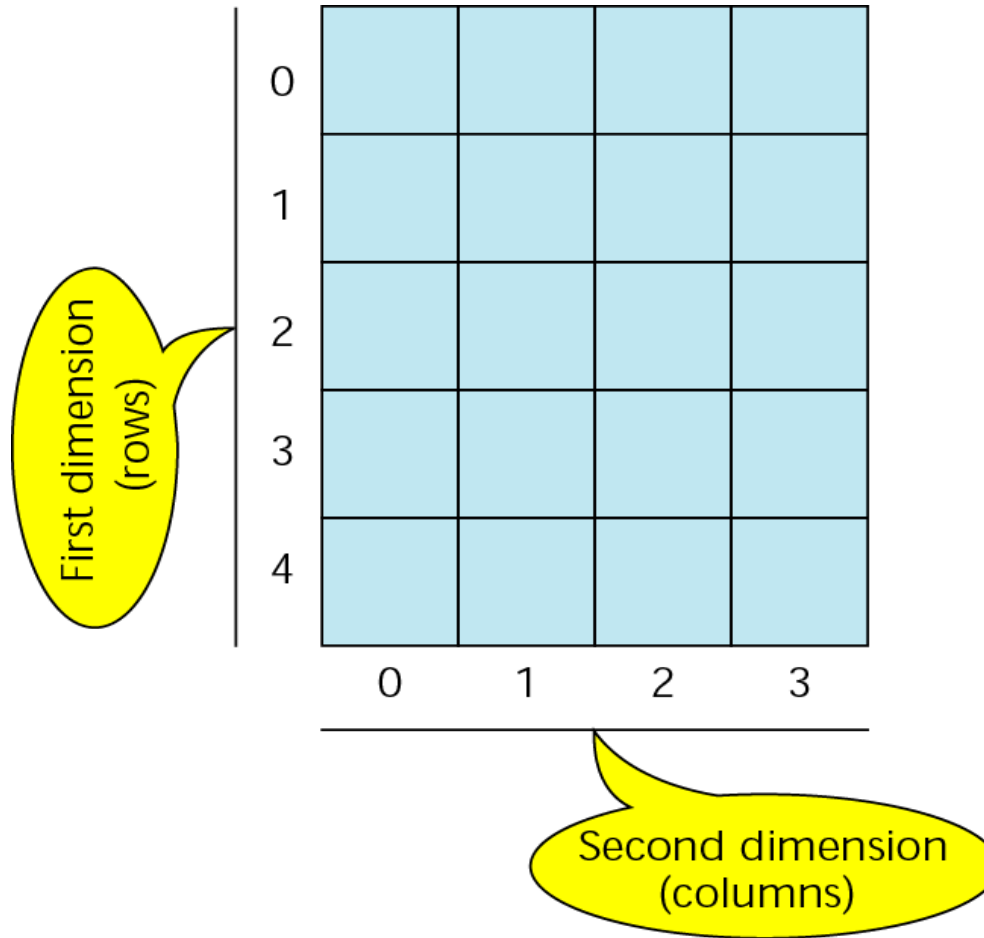
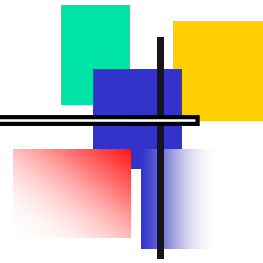


## Figure 8-28 Unsuccessful binary search example

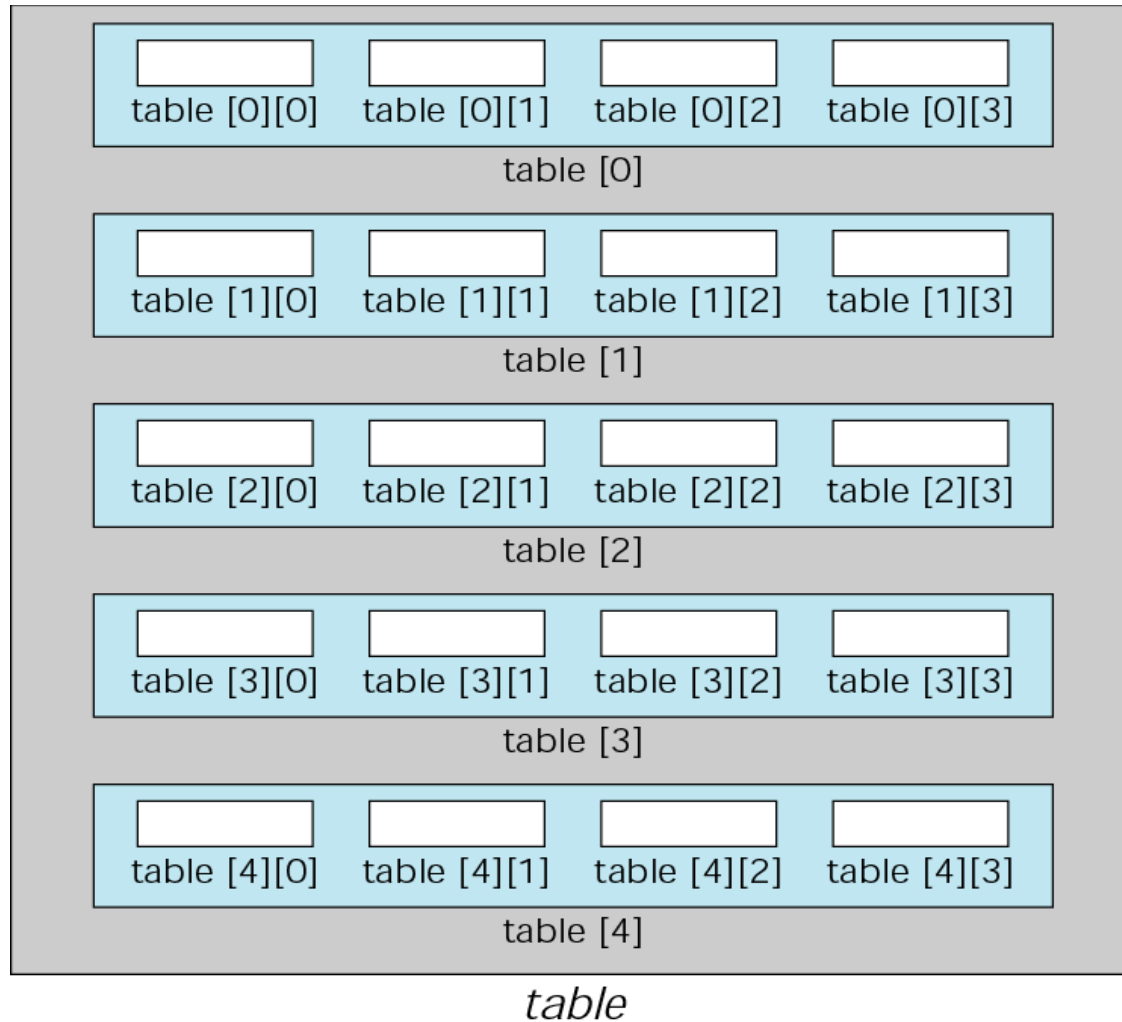
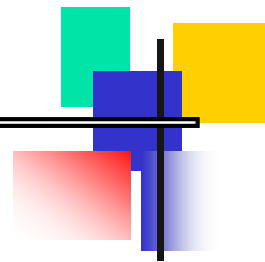


## TWO- DIMENSIONAL ARRAYS

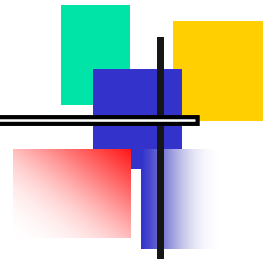
## Figure 8-29 Two-dimensional array



## Figure 8-30 Array of arrays

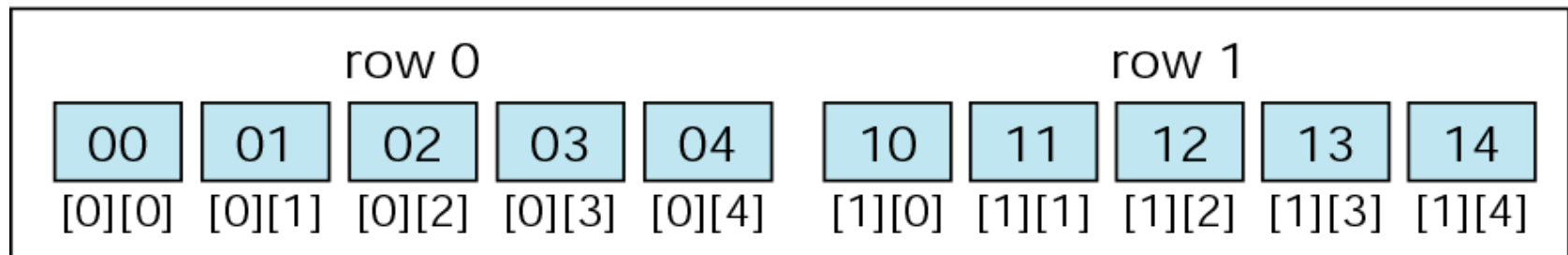


## Figure 8-31 Memory layout



00	01	02	03	04
10	11	12	13	14

User's view



Memory view





## Figure 8-32 Passing a row

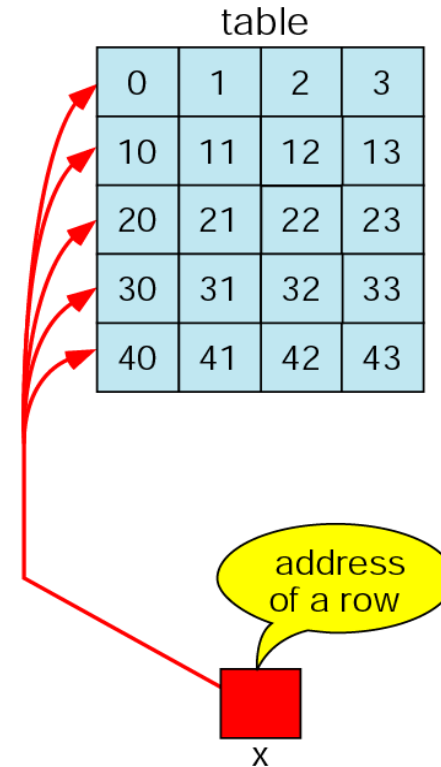
```
const int cMAX_ROWS = 5;
const int cMAX_COLS = 4;

void print_sqr ( const int [ ] );

int main ()
{
    int table [cMAX_ROWS] [cMAX_COLS] =
        {
            { 0, 1, 2, 3 },
            { 10, 11, 12, 13 },
            { 20, 21, 22, 23 },
            { 30, 31, 32, 33 },
            { 40, 41, 42, 43 }
        }; // table

    ...
    for ( int row = 0 ; row < cMAX_ROWS ; row++ )
        print_sqr ( table [ row ] );
    ...
    return 0 ;
} // main
```

```
void print_sqr (const int x [ ] )
{
    for ( int col = 0 ; col < MAX_COLS ; col++ )
        cout << setw(6) << x [col] * x [col] ;
    cout << endl ;
    return ;
} // print_sqr
```



## Figure 8-33 Calculate average of integers in array

```
const int cMAX_ROWS   = 5;
const int cMAX_COLS   = 4;
double average ( int [ ] [cMAX_COLS] ) ;
int main ( )
{
    int
        ave;
    int
        table [ MAX_ROWS ] [cMAX_COLS] =
            {
                { 0, 1, 2, 3 },
                { 10, 11, 12, 13 },
                { 20, 21, 22, 23 },
                { 30, 31, 32, 33 },
                { 40, 41, 42, 43 }
            };
    ...
    ave = average ( table ) ;
    ...
    return 0 ;
} // main
```

table

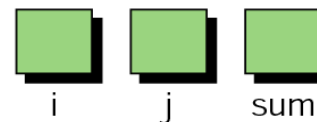
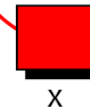
0	1	2	3
10	11	12	13
20	21	22	23
30	31	32	33
40	41	42	43

```
double average ( int x [ ] [cMAX_COLS] )
{
    double sum = 0 ;

    for (int i = 0 ; i < cMAX_ROWS ; i++)
        for (int j = 0 ; j < cMAX_COLS ; j++)
            sum += x [ i ] [ j ] ;

    return( sum / ( cMAX_ROWS * cMAX_COLS ) ;
} // average
```

Address  
of table



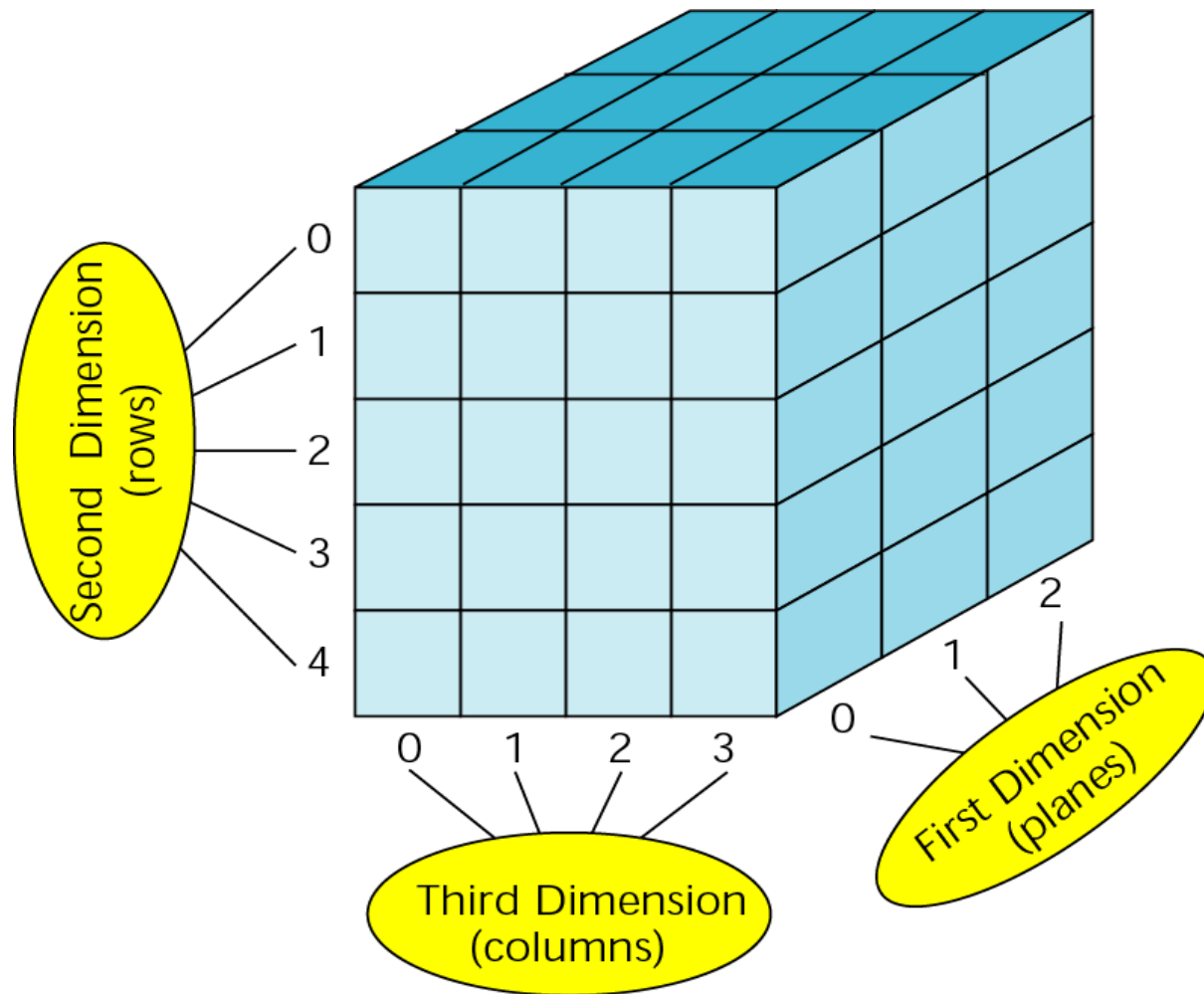
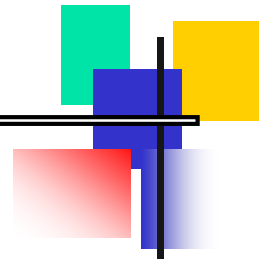
## Figure 8-34 Example of filled matrix



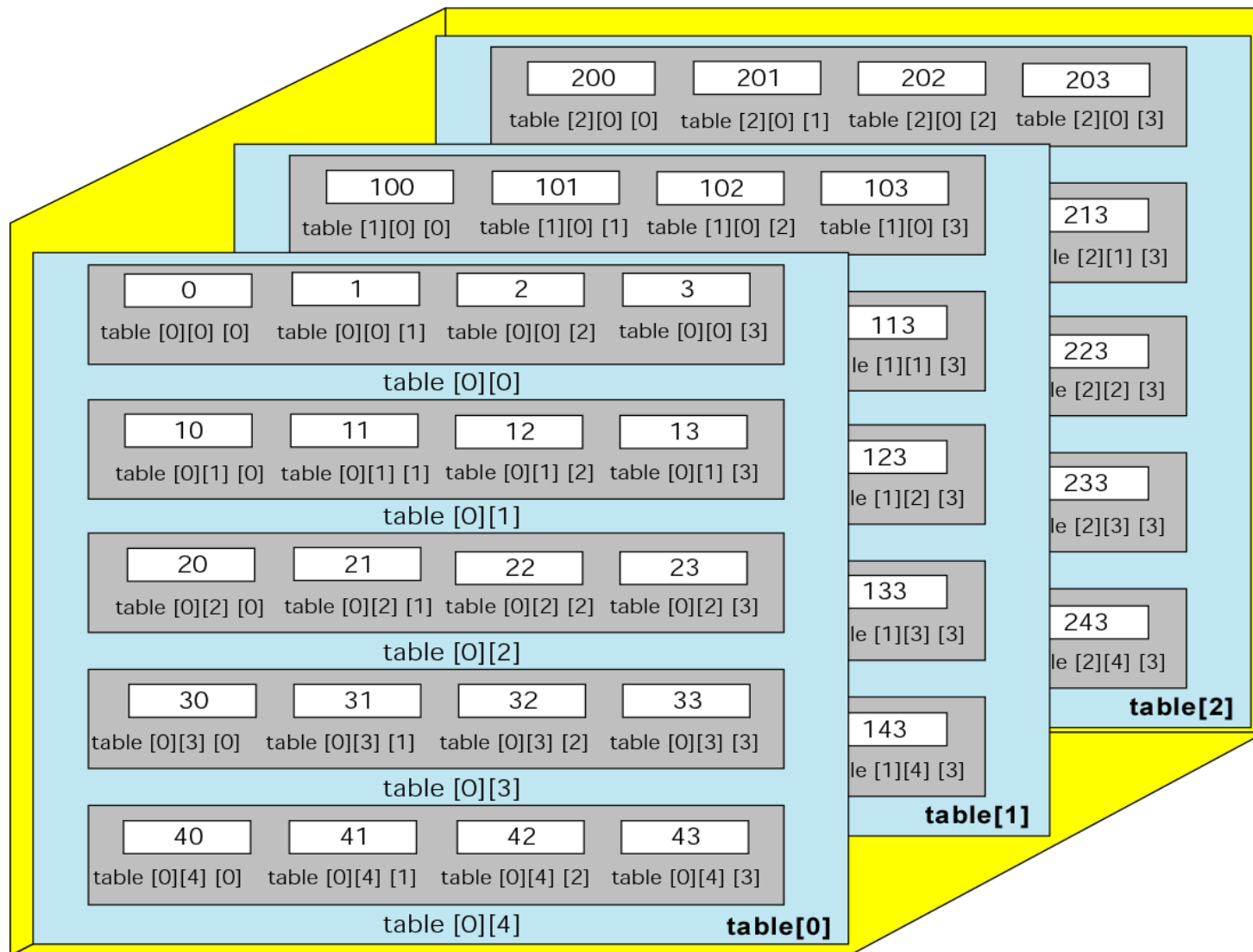
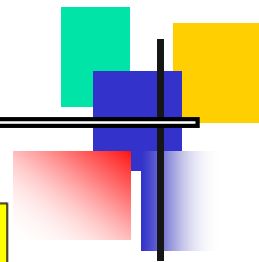
0	1	1	1	1	1
-1	0	1	1	1	1
-1	-1	0	1	1	1
-1	-1	-1	0	1	1
-1	-1	-1	-1	0	1
-1	-1	-1	-1	-1	0

# MULTIDIMENSIONAL ARRAYS

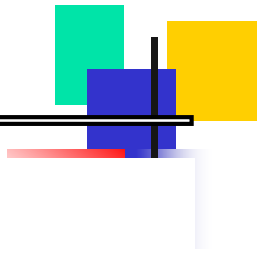
**Figure 8-35** A three-dimensional array (3 x 5 x 4)



# Figure 8-36 C++ view of three-dimensional array



## Figure 8-37 Initializing a three-dimensional array



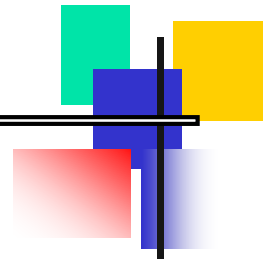
```
int    table[3][5][4] =  
{  
    {    // Plane 0  
        {0, 1, 2, 3},    // Row 0  
        ...  
        {40, 41, 42, 43}    // Row 4  
    },  
    {    // Plane 1  
        {100, 101, 102, 103},    // Row 0  
        ...  
        {140, 141, 142, 143}    // Row 4  
    },  
    {    // Plane 2  
        {200, 201, 202, 203},    // Row 0  
        ...  
        {240, 241, 242, 243}    // Row 4  
    }  
}; // table
```



PROGRAMMING  
EXAMPLE—  
CALCULATE ROW  
AND COLUMN  
AVERAGES



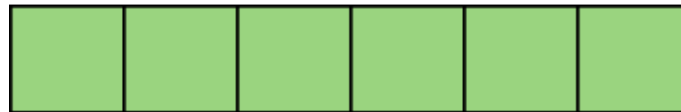
## Figure 8-38 Data structures for row-column averages




table

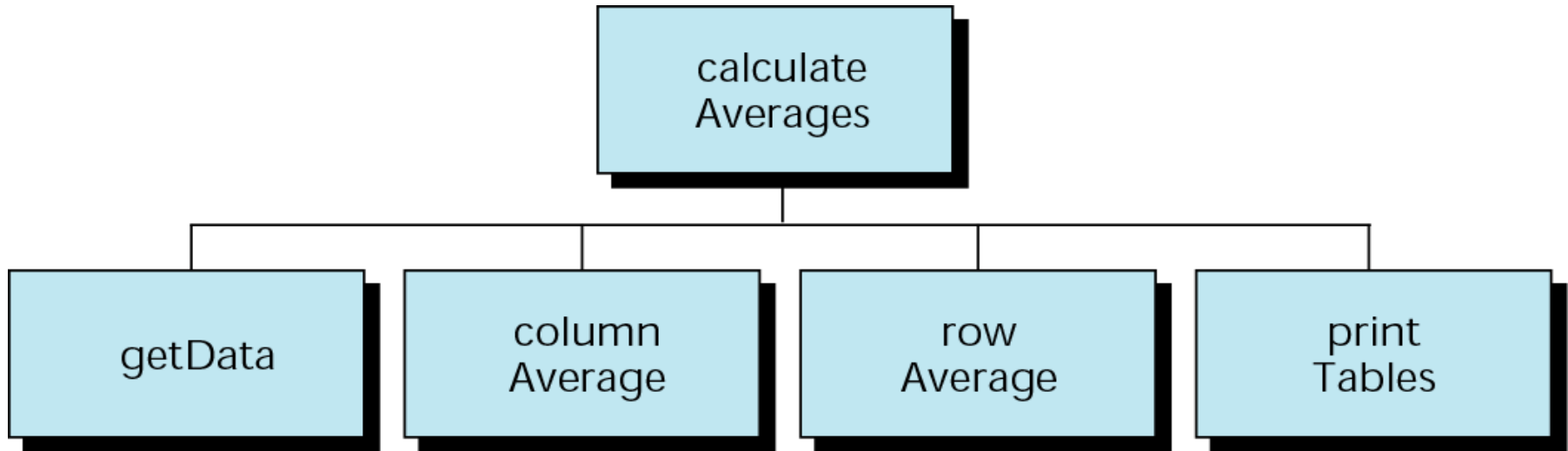
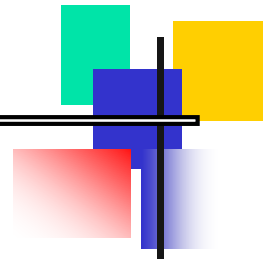


rowAve



columnAve

## Figure 8-39 Calculate row-column average design



# SOFTWARE ENGINEERING AND PROGRAMMING STYLE

Note:

*The efficiency of the bubble sort is  $O(n^2)$ .*

Note:

*The efficiency of the selection sort is  $O(n^2)$ .*

Note:

*The efficiency of the insertion sort is  $O(n^2)$ .*

Note:

*The efficiency of the sequential search  
is  $O(n)$ .*

Note:

*The efficiency of the binary search is  $O(\log_2 n)$ .*