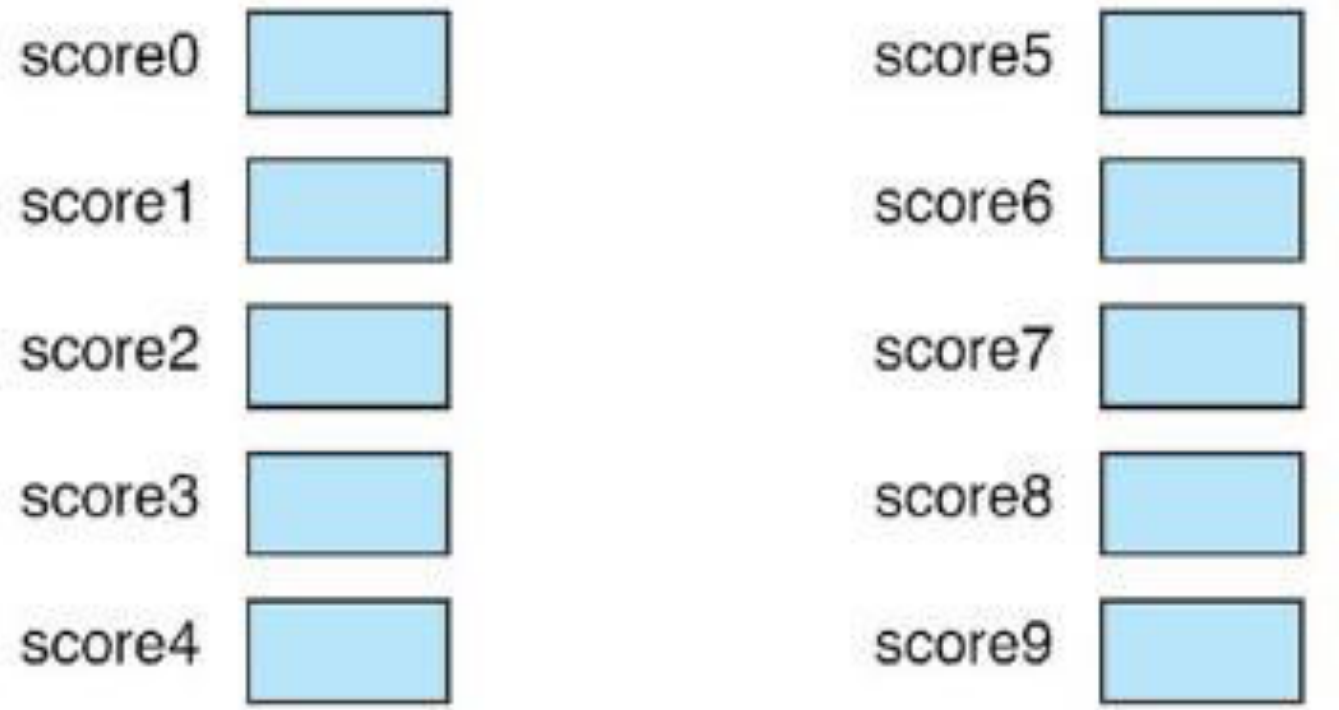
ARRAYS

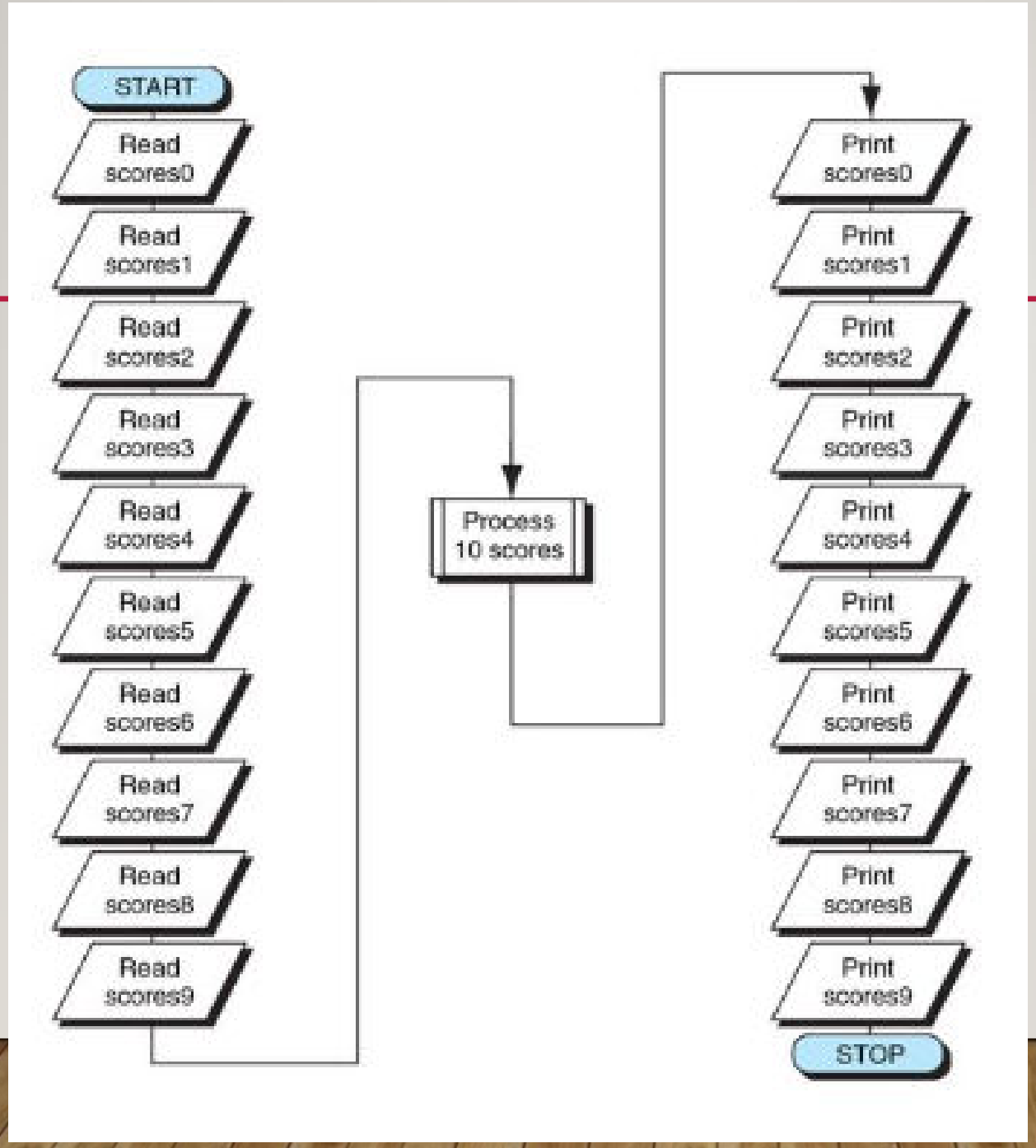
Imagine we have a problem that requires us to read, process, and print a large number of integers. We must also keep the integers in memory for the duration of the program.

To process large amounts of data we need a powerful data structure, the array. An array is a collection of elements of the same data type.

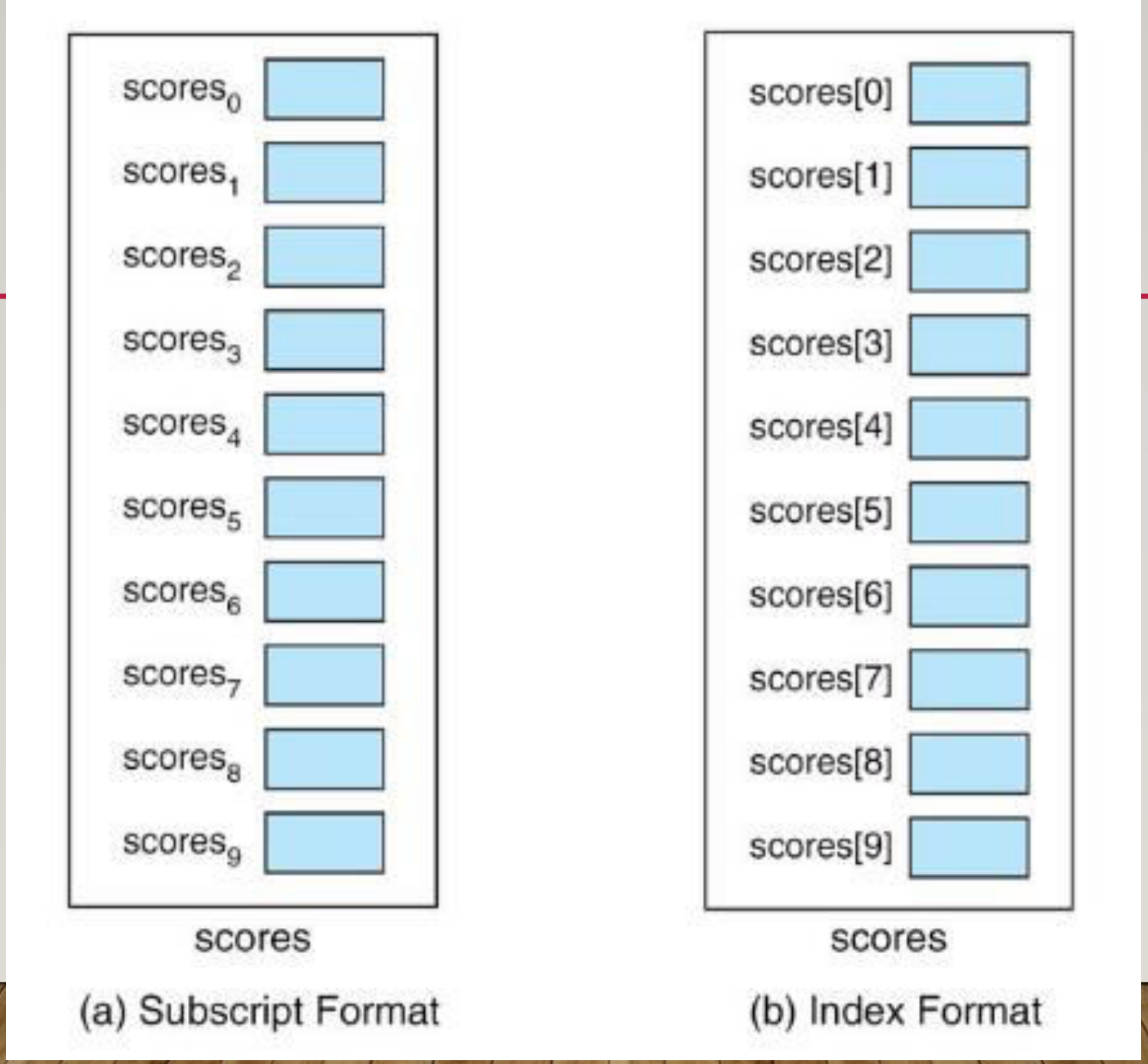
Since an array is a sequenced collection, we can refer to the elements in the array as the first element, the second element, and so forth until we get to the last element.

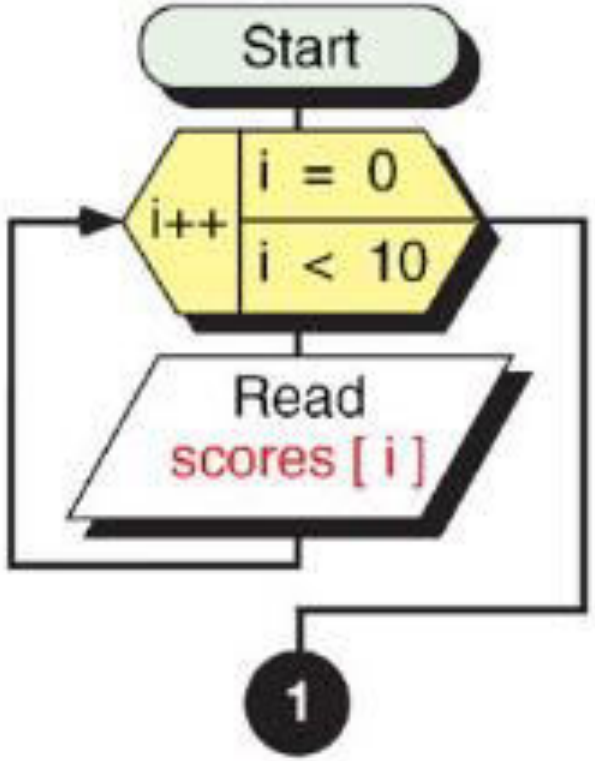
TEN VARIABLES

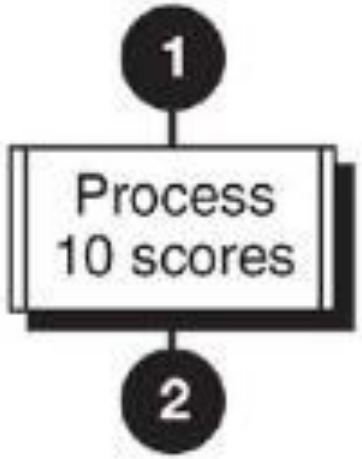



START
Read scores0
Read scores1
Read scores2
Read scores3
Read scores4
Read scores5
Read scores6
Read scores7
Read scores8
Read scores9
Process 10 scores
Print scores0
Print scores1
Print scores2
Print scores3
Print scores4
Print scores5
Print scores6
Print scores7
Print scores8
Print scores9
STOP

ARRAY OF SCORES







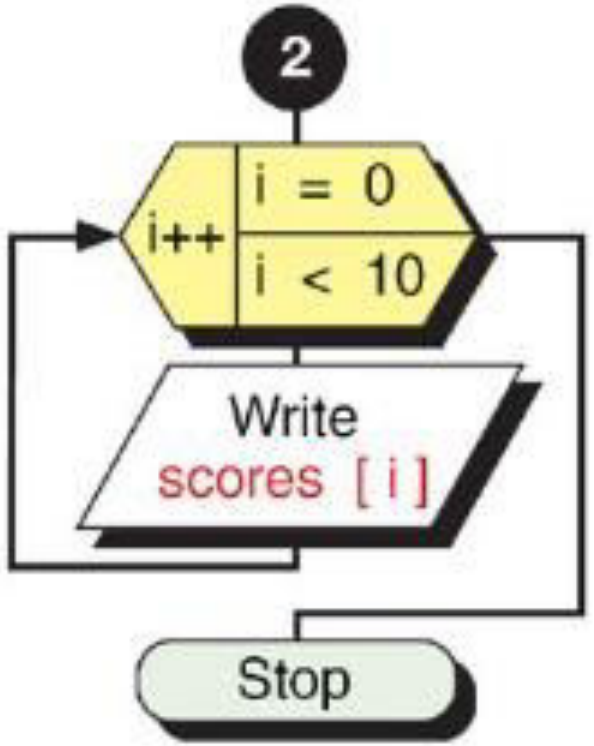


FIGURE 8-5 **Loop for 10 Scores**

ARRAY

- An array is a fixed-size sequential collection of elements of same data types that share a common name.
- It is simply a group of data types.
- An array is a derived data type.
- An array is used to represent a list of numbers , or a list of names.

- For Example :

1. List of employees in an organization.
2. Test scores of a class of students.
3. List of customers and their telephone numbers.
4. List of students in the college.

- For Example, to represent 100 students in college , can be written as

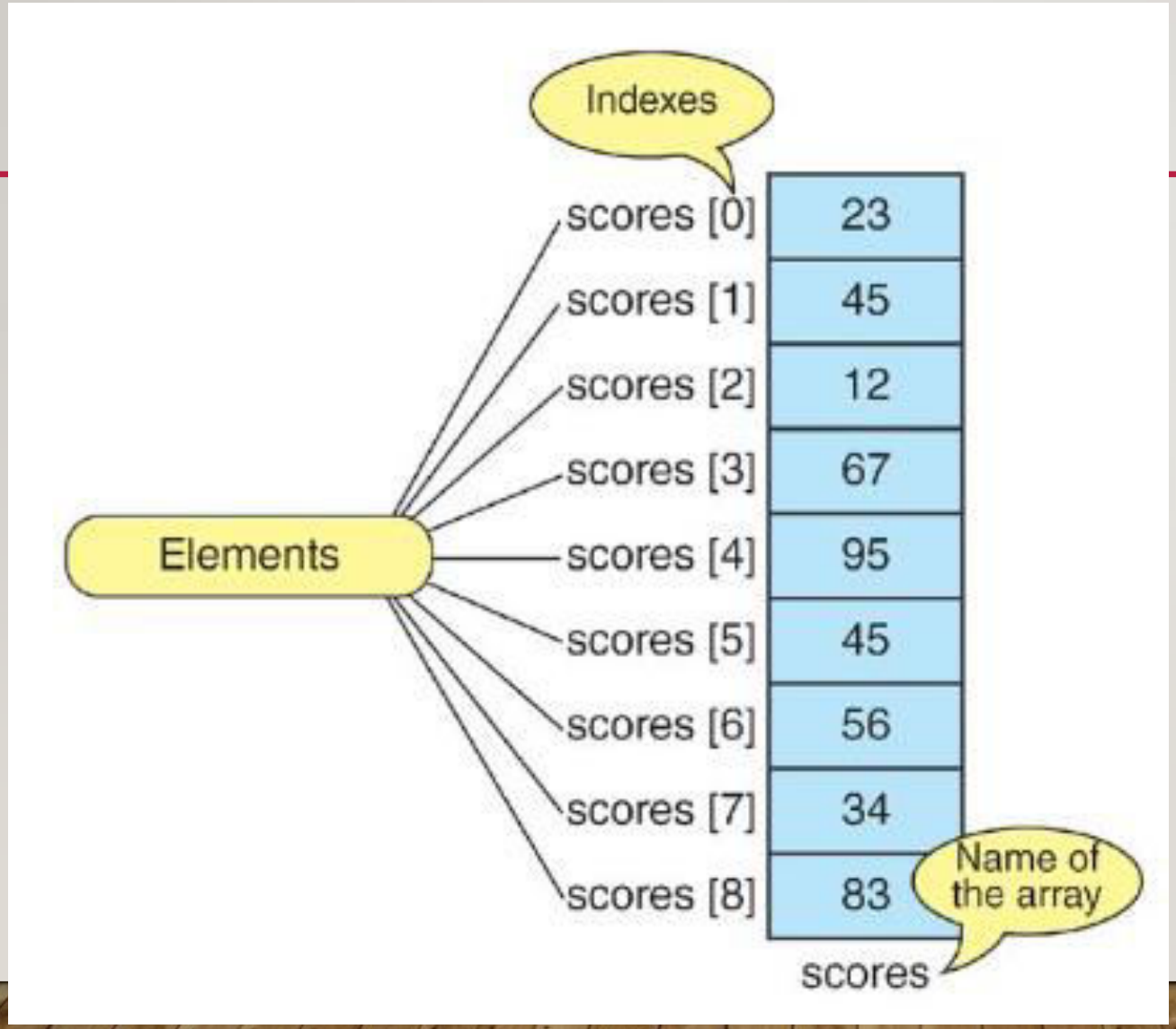
student [100]

- Here student is a array name and [100] is called index or subscript.

Types of Array :

1. One-dimensional arrays
2. Two-dimensional arrays
3. Multidimensional arrays

USING ARRAYS IN C



ONE DIMENSIONAL ARRAY

- A variable which represent the list of items using only one index (subscript) is called one-dimensional array.

- For Example , if we want to represent a set of five numbers say(35,40,20,57,19), by an array variable number, then number is declared as follows

```
int number [5] ;
```

- and the computer store these numbers as shown below :

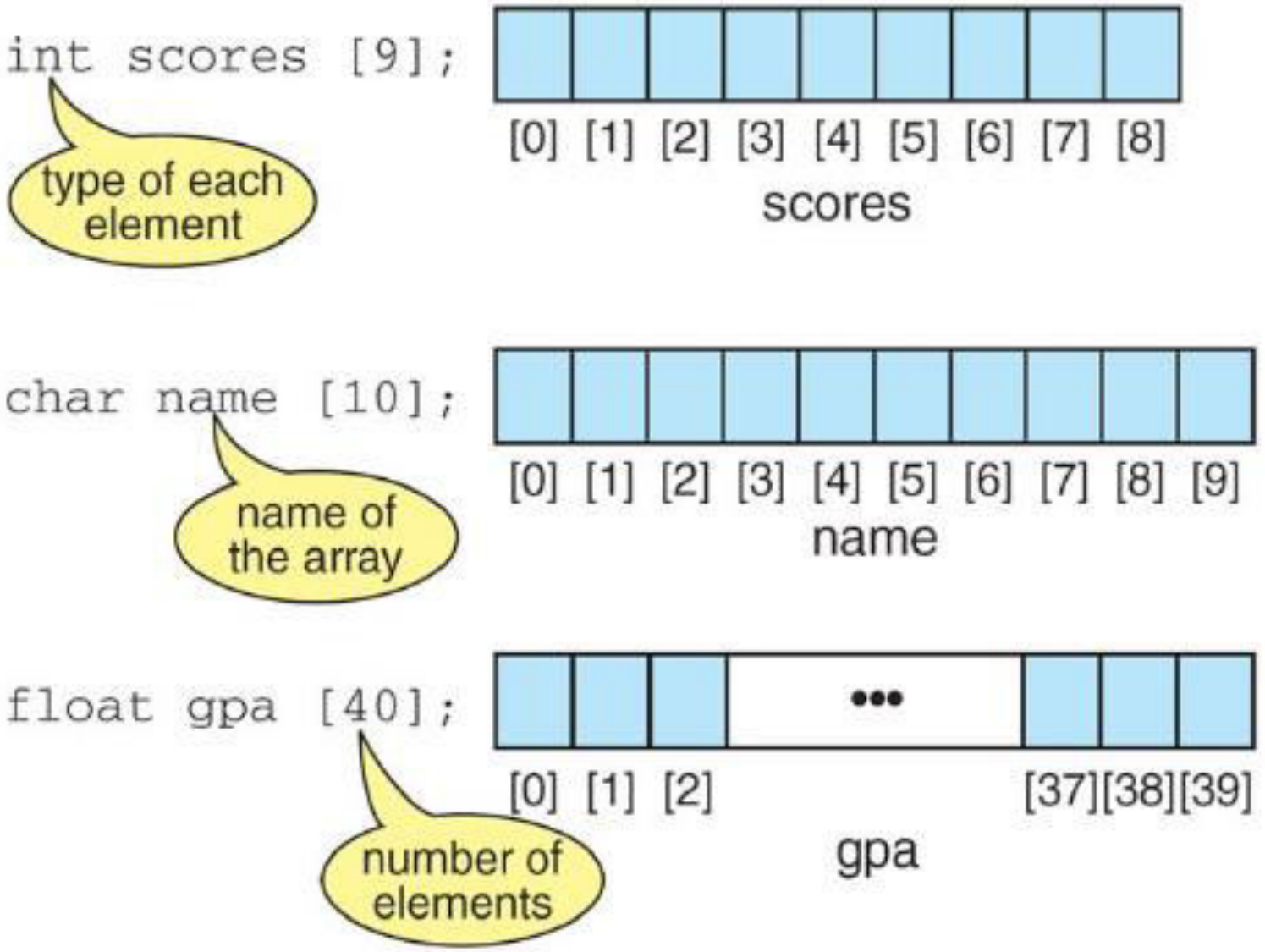
number [0]
number [1]
number [2]
number [3]
number [4]

- The values can be assigned to the array as follows :

number [0] = 35;
number [1] = 40;
number [2] = 20;
number [3] = 57;
number [4] = 19;

DECLARING AND DEFINING THE ARRAY

- Syntax:
- <data-type> <name of array> [<array size>]
- The array size must be an integer constant, which is greater than zero.
- The type of elements can be any C data type.
- The array size is indicated in square brackets '[ ]'.



URE 8-7 Declaring and Defining Arrays

Note

Only fixed-length arrays can be initialized when they are defined. Variable length arrays must be initialized by inputting or assigning the values.

Note

One array cannot be copied to another using assignment.

INITIALIZING ARRAYS

- The initialization of an array in C program can be either one at a time or by using a single statement.
  - Single statement of fixed length array
  - Variable length array (initialization without size)
  - Partial initialization of an array
  - Initialization to all zeros

(a) Basic Initialization

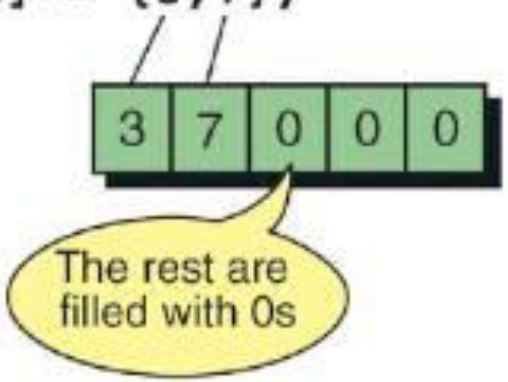
```
int numbers[5] = {3,7,12,24,45};
```

(b) Initialization without Size

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

(c) Partial Initialization

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



(d) Initialization to All Zeros

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

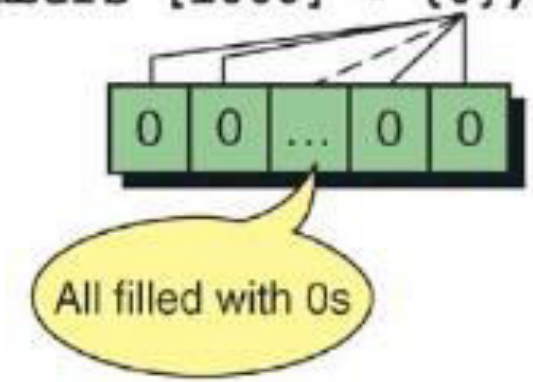


FIGURE 8-8 Initializing Arrays

Compile time initialization

Example :

```
int number[ ] = {1,2,3,4};
```

- The character array can be initialized as follows :

```
char name[ ] = {'j','o','h','n','\0'};
```

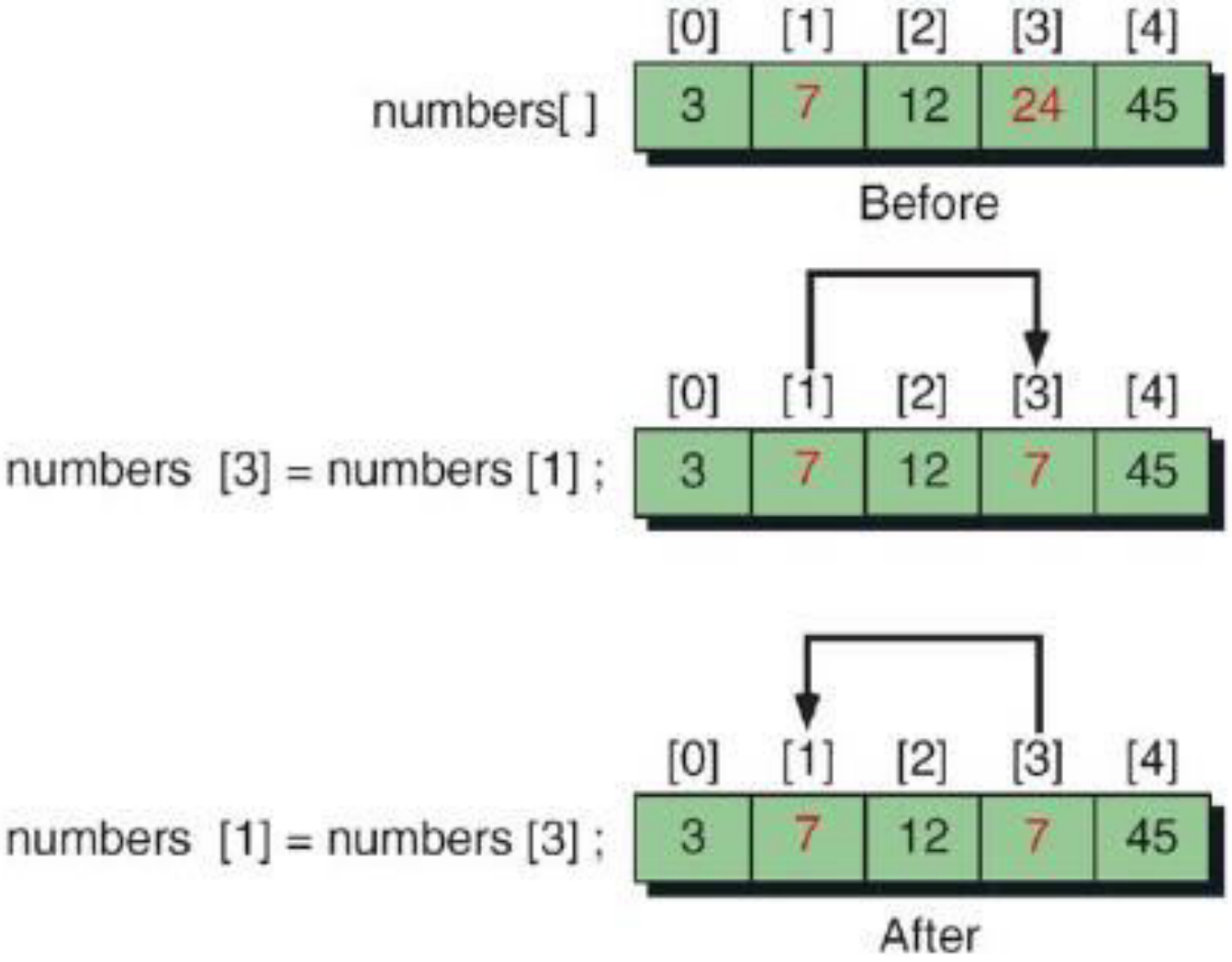
- The character array can also be initialized as follows :

```
char name[ ] = "john";
```

Run time initialization :

- In run time initialization, the array is explicitly initialize at run time.
- This concept generally used for initializing large arrays.
- Example:

```
for(i=0; i < 100; i++)
{
        if( i < 50)
        sum[i] = 0.0;
        else
        sum[i] = 1.0;
}
```



IGURE 8-9 Exchanging Scores—the Wrong Way

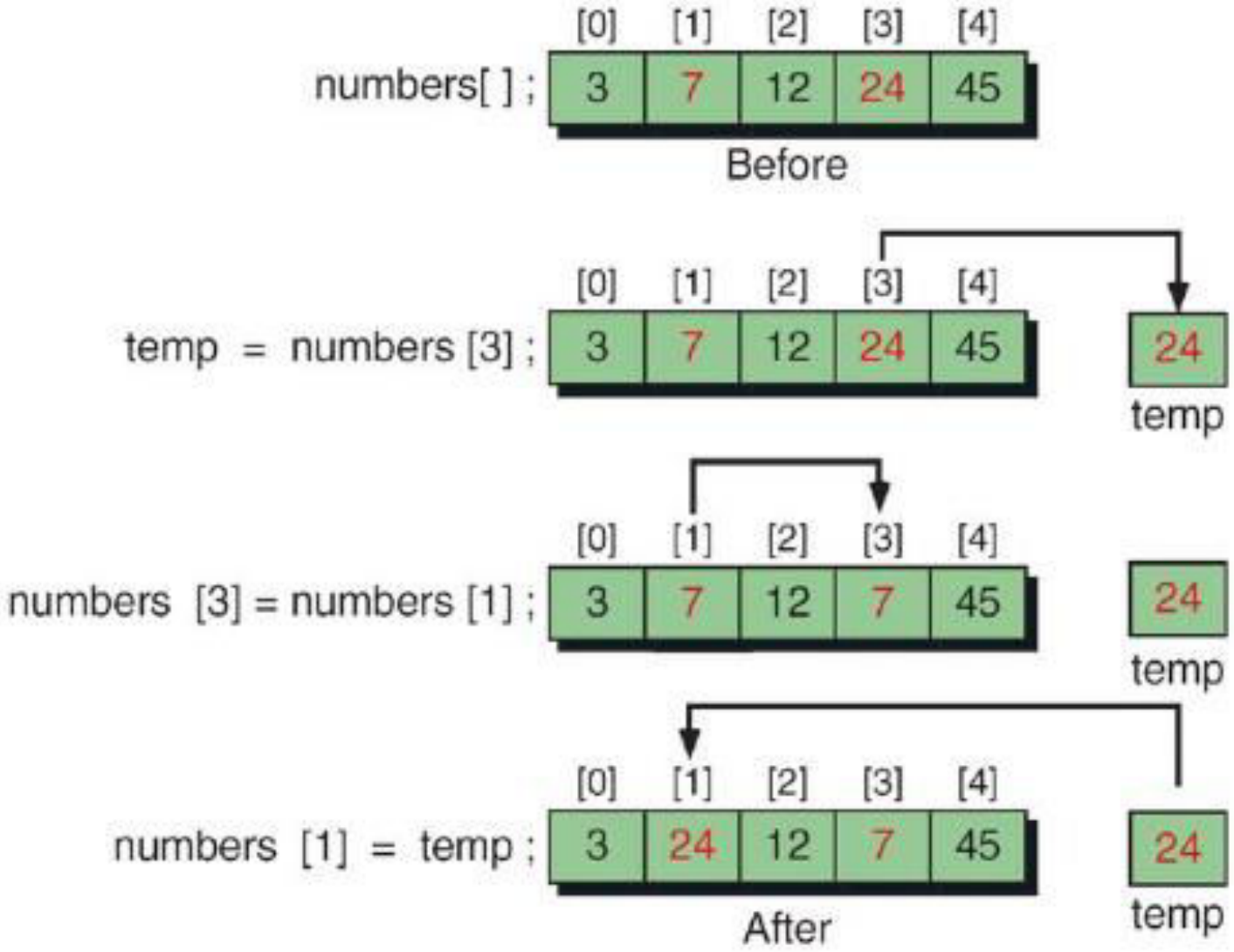


FIGURE 8-10 Exchanging Scores with Temporary Variable

PRINT 10 NUMBERS PER LINE

```
// a program fragment
#define MAX_SIZE 25

// Local Declarations
   int list [MAX_SIZE];

// Statements
   …
   numPrinted = 0;
   for (int i = 0; i < MAX_SIZE; i++)
       {
        printf("%3d", list[i]);
        if (numPrinted < 9)
            numPrinted++;
        else
            {
            printf("\n");
            numPrinted = 0;
            } // else
       } // for
```

```c
/* Initialize array with square of index and print it.
      Written by:
      Date:
*/
#include <stdio.h>
#define ARY_SIZE 5

int main (void)
{
// Local Declarations
   int sqrAry[ARY_SIZE];

// Statements
   for (int i = 0; i < ARY_SIZE; i++)
      sqrAry[i] = i * i;

   printf("Element\tSquare\n");
   printf("=======\t======\n");
```

```
    for (int i = 0; i < ARY_SIZE; i++)
        printf("%5d\t%4d\n", i, sqrAry[i]);
    return 0;
}   // main
```

```
Results:
Element Square
======= ======
      0      0
      1      1
      2      4
      3      9
      4     16
```

```
/* Read a number series and print it reversed.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int readNum;
   int numbers[50];

// Statements
   printf("You may enter up to 50 integers:\n");
   printf("How many would you like to enter? ");
   scanf ("%d", &readNum);

   if (readNum > 50)
       readNum = 50;
```

```
// Fill the array
printf("\nEnter your numbers: \n");
for (int i = 0; i < readNum; i++)
      scanf("%d", &numbers[i]);

// Print the array
printf("\nYour numbers reversed are: \n");
for (int i = readNum - 1, numPrinted = 0;
          i >= 0;
          i--)
    {
     printf("%3d", numbers[i]);
     if (numPrinted < 9)
         numPrinted++;
     else
        {
         printf("\n");
         numPrinted = 0;
        } // else
```

```
        } // for
    return 0;
}   // main
```

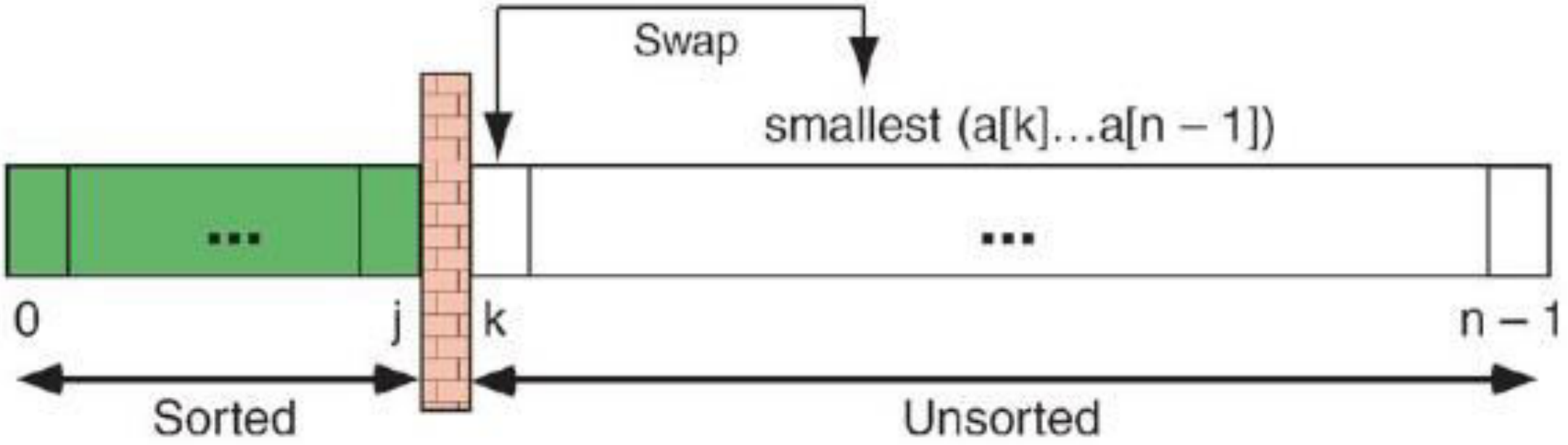
```
Results:
You may enter up to 50 integers:
How many would you like to enter? 12

Enter your numbers:
1 2 3 4 5 6 7 8 9 10 11 12

Your numbers reversed are:
 12 11 10  9  8  7  6  5  4  3
  2  1
```

SORTING

One of the most common applications in computer science is sorting—the process through which data are arranged according to their values. We are surrounded by data. If the data are not ordered, we would spend hours trying to find a single piece of information.



URE 8-18 Selection Sort Concept

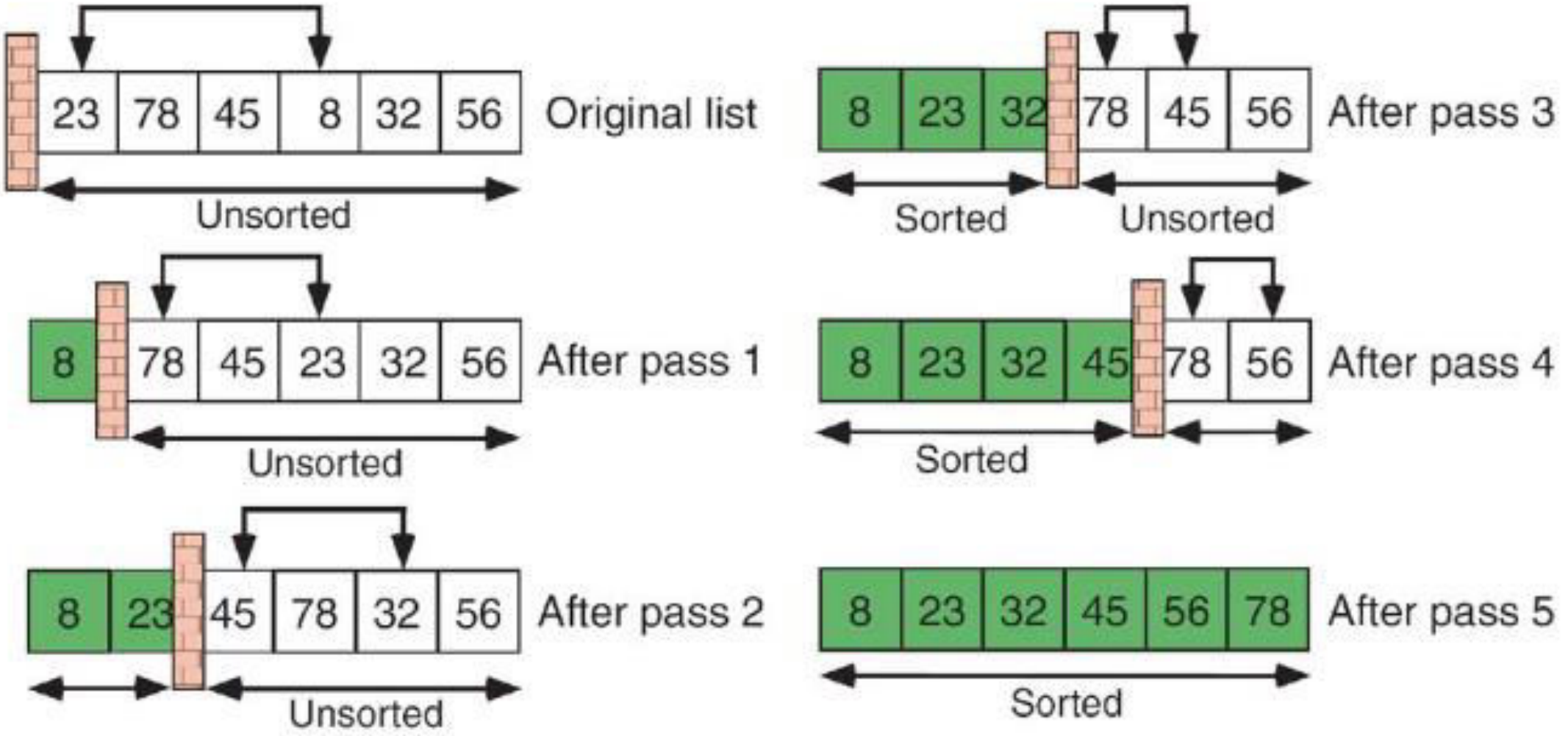
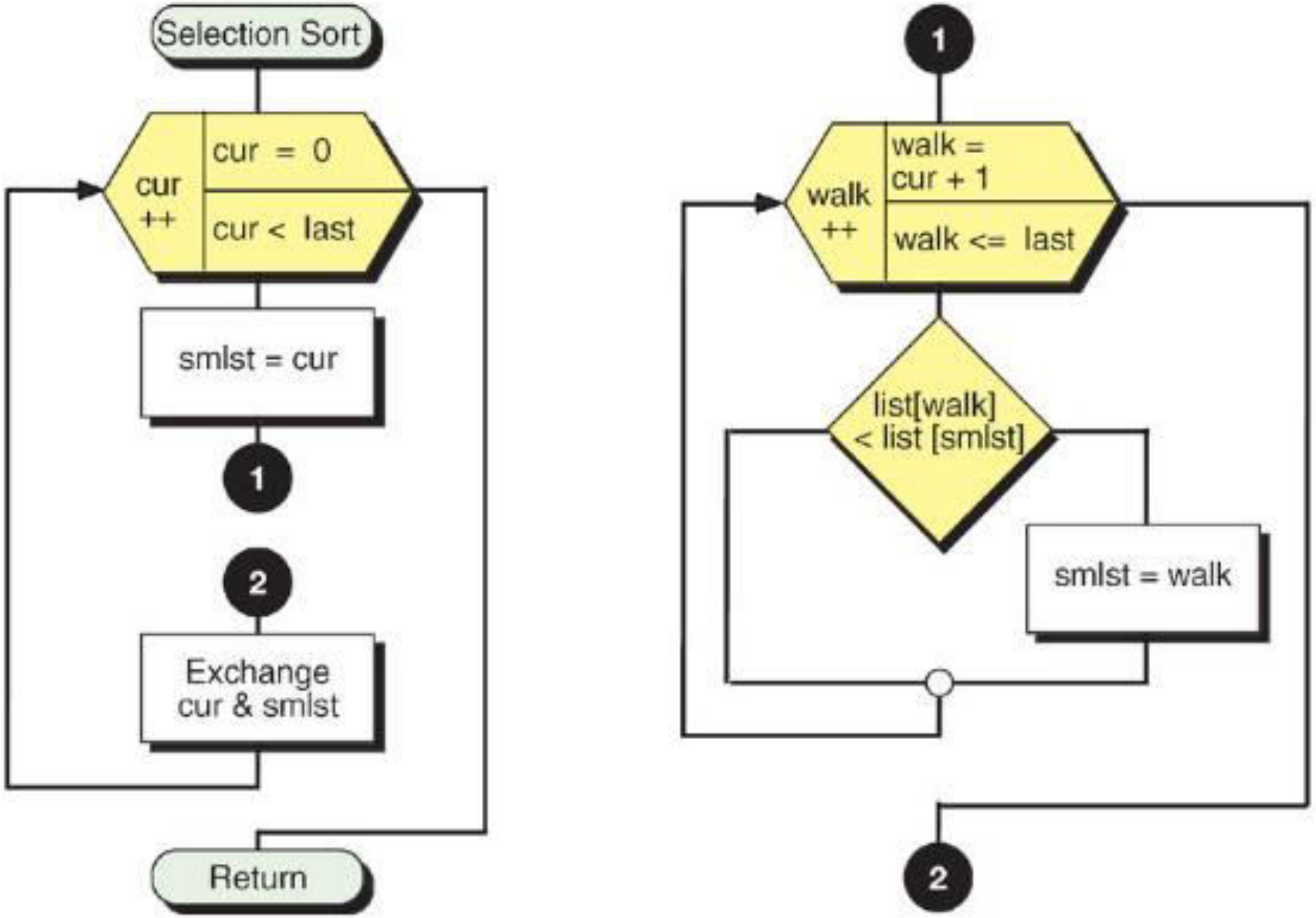


FIGURE 8-19 Selection Sort Example



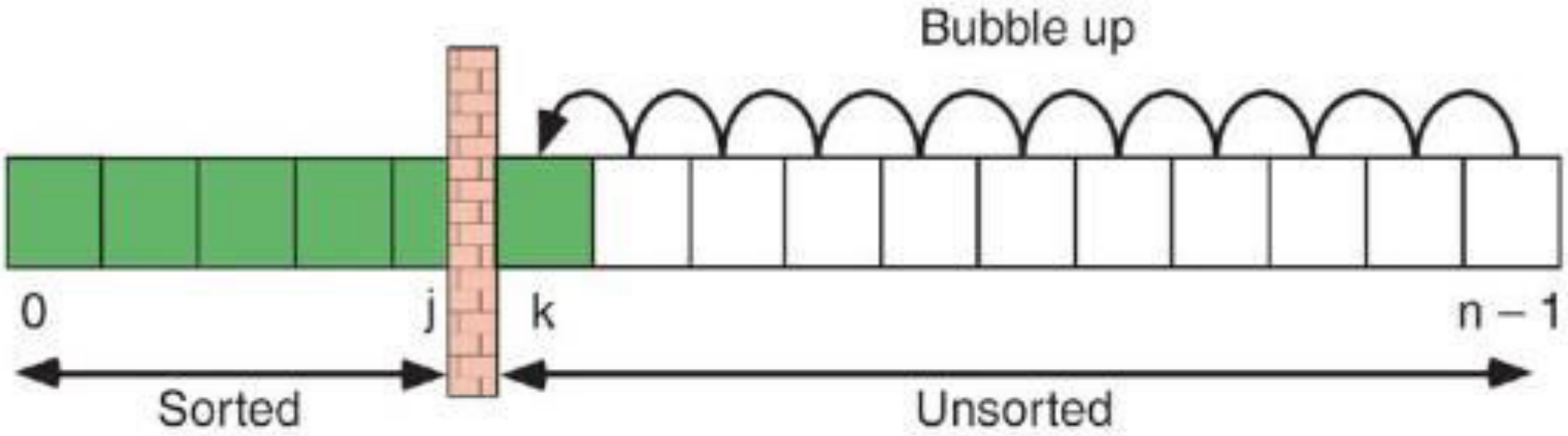
E 8-20 Design for Selection Sort

SELECTION SORT

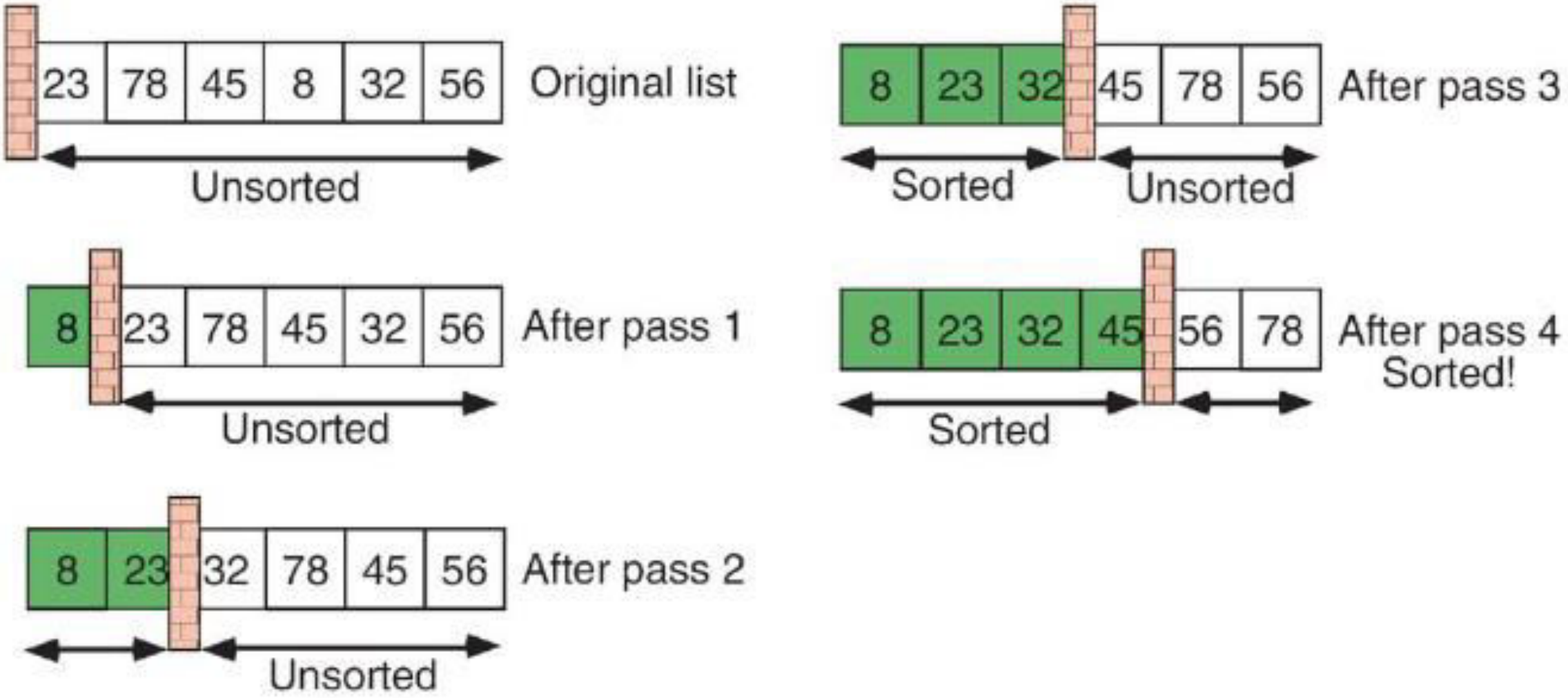
```
// Local Declarations
   int smallest;
   int tempData;

// Statements
   // Outer Loop
   for (int current = 0; current < last; current++)
        {
         smallest = current;
```

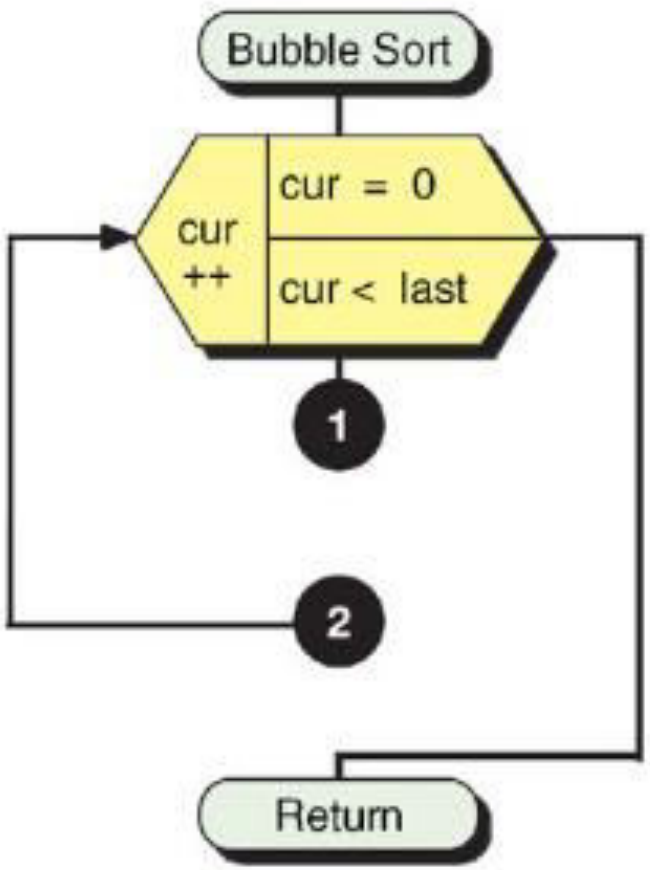
```
        // Inner Loop: One sort pass each loop
        for (int walk = current + 1;
                 walk <= last;
                 walk++)
            if (list[walk] < list[smallest])
                smallest = walk;
    // Smallest selected: exchange with current
    tempData         = list[current];
    list[current]    = list[smallest];
    list[smallest] = tempData;
    } // for current
return;
```

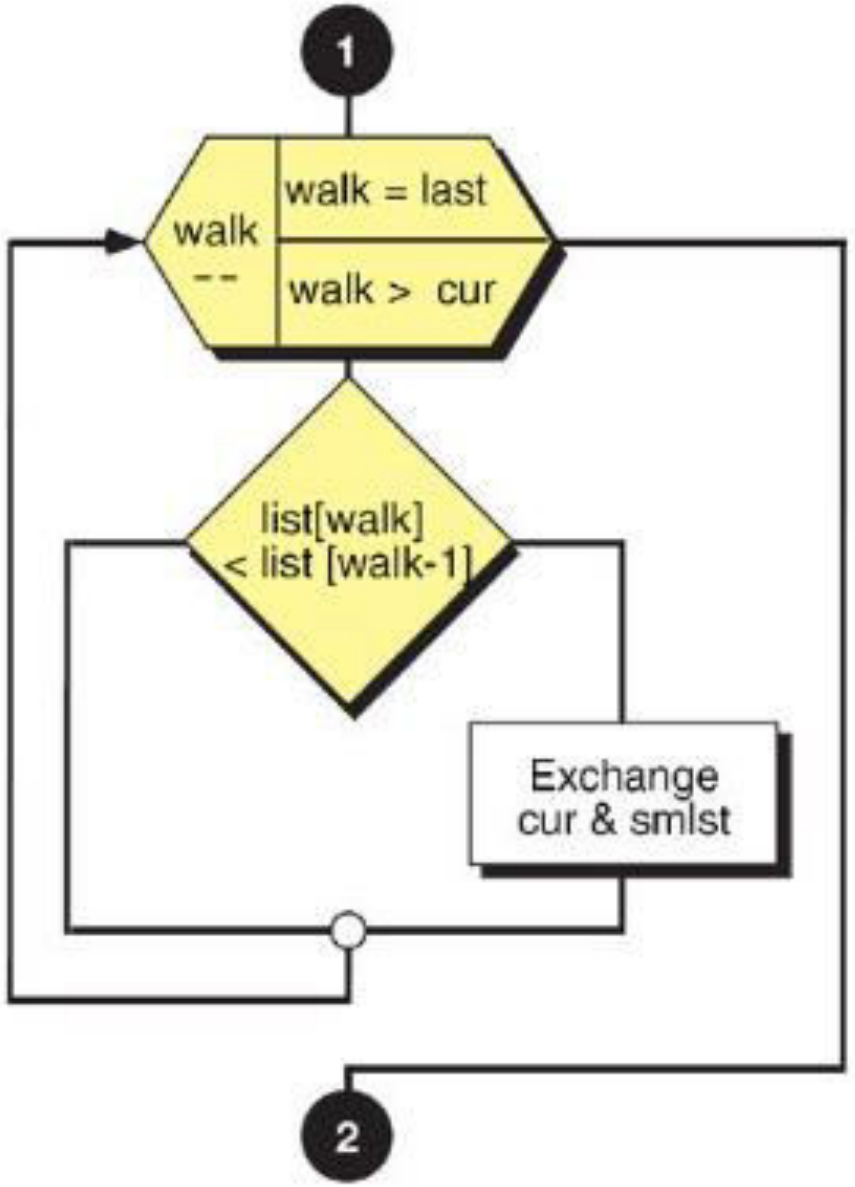


8-21 Bubble Sort Concept



GURE 8-22 Bubble Sort Example





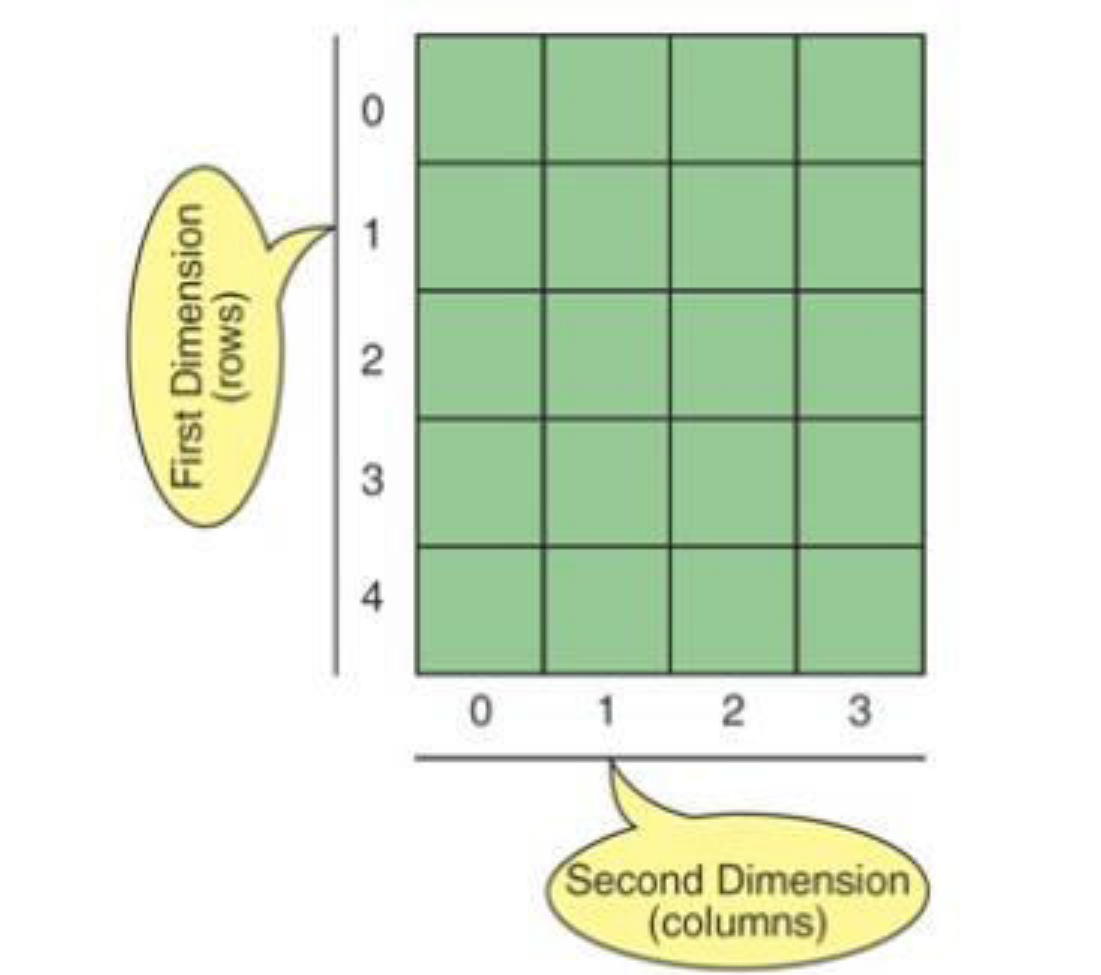
RE 8-23 Bubble Sort Design

```
// Local Declarations
   int temp;

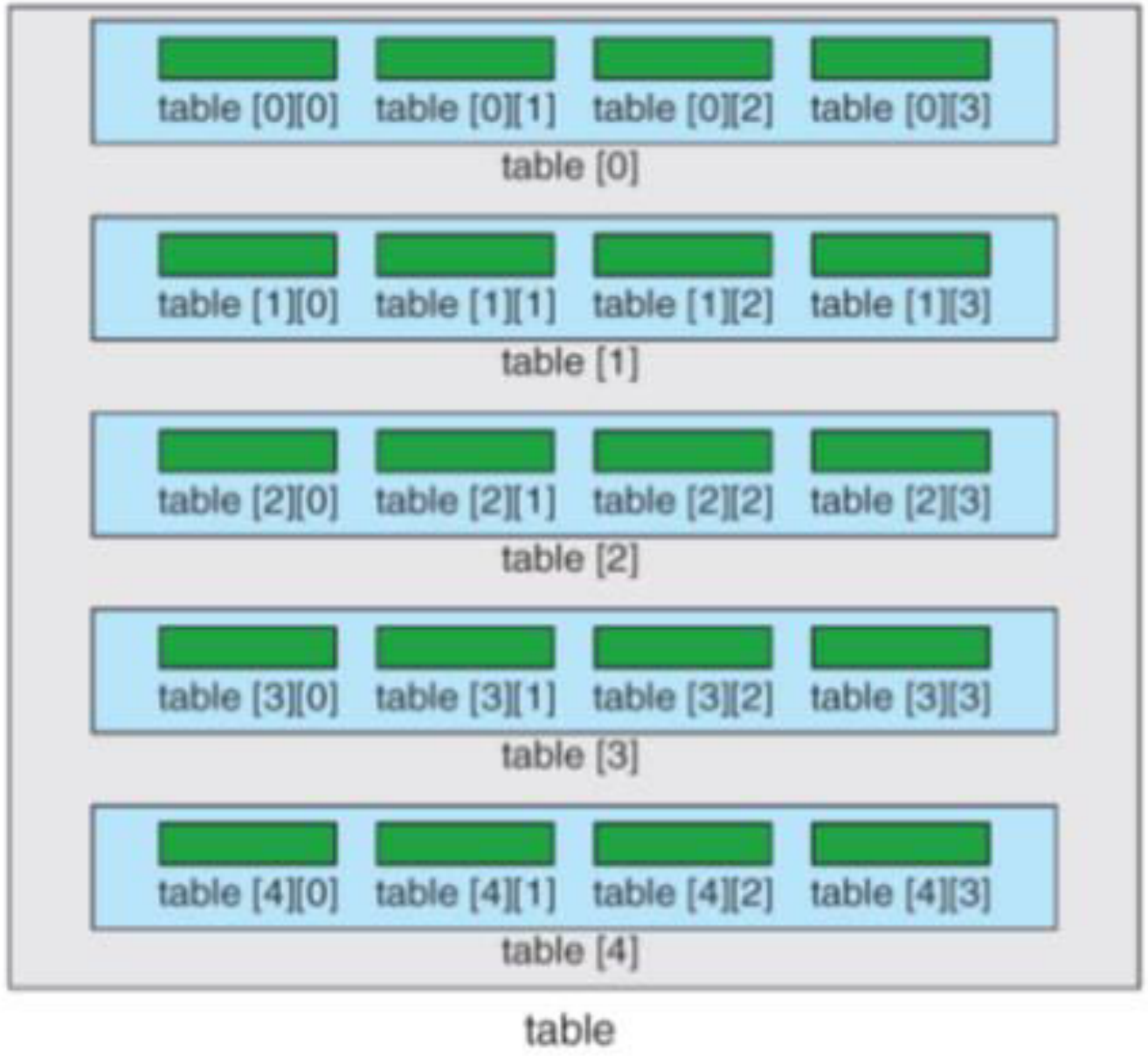
// Statements
   // Outer loop
   for(int current = 0; current < last; current++)
      {
       // Inner loop: Bubble up one element each pass
```

```
        for (int walker = last;
                 walker > current;
                 walker--)
            if (list[walker]  < list[walker - 1])
               {
                temp                = list[walker];
                list[walker]        = list[walker - 1];
                list[walker - 1] = temp;
               } // if
    } // for current
  return;
} // bubbleSort
```

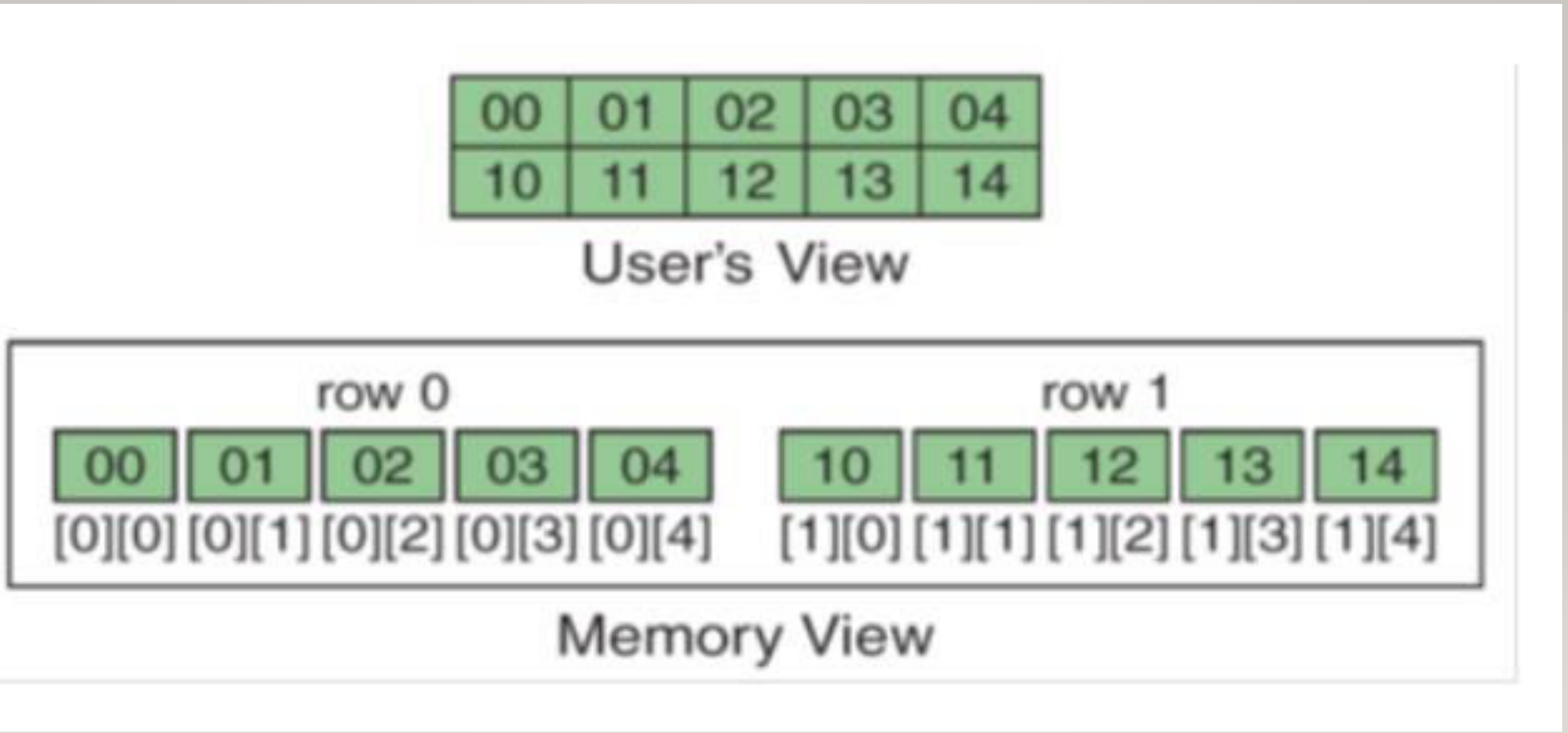
TWO DIMENSIONAL ARRAY



Two-dimensional Array



Array Of Arrays


00 01 02 03 04
10 11 12 13 14
User's View
row 0
row 1
00 01 02 03 04
10 11 12 13 14
[0][0] [0][1] [0][2] [0][3] [0][4]
[1][0] [1][1] [1][2] [1][3] [1][4]
Memory View

Two-dimensional Arrays

- A variable which represent the list of items using two index (subscript) is called two-dimensional array.
- In Two dimensional arrays, the data is stored in rows and columns format.
- For example:

```
int table[2][3];
```

DECLARATION OF TWO-DIMENSIONAL ARRAYS :

- The general form of two dimensional array declaration is :

  type *array-name[row_size][column_size];*

- Here the type specifies the data type of elements contained in the array, such as int, float, or char.
- The size should be either a numeric constant or a symbolic constant.

INITIALIZATION OF TWO-DIMENSIONAL ARRAYS :

- The general form of initializing two-dimensional array is :

type array-name[row_size][column_size] = {list

of values};

- Example :

int table[2][3] = {0,0,0,1,1,1};

- Here the elements of first row initializes to zero and the elements of second row initializes to one.
- This above statement can be written as :

int table[2][3] = {{0,0,0}, {1,1,1}};

- In two-dimensional array the row_size can be omitted.

INITIALIZATION OF TWO-DIMENSIONAL ARRAYS :

- Example :

```
int table[ ][3] = {{0,0,0}, {1,1,1}};
```

- If the values are missing in an initializer, they are automatically set to zero.

- Example :

```
int table[2][3] = {1,1,2};
```

- Here first row initialize to 1,1 and 2, and second row initialize to 0,0 and 0 automatically.

Memory Layout of Two-dimensional array :

- In Two dimensional arrays, the data is stored in rows and columns format.
- For example:

```
int table[2][3] = {1,2,3,4,5,6};
```

- The memory layout of above example :

```
table[0][0] = 1;
table[0][1] = 2
table[0][2] = 3;
 table[1][0] = 4;
table[1][1] = 5;
 table[1][2] = 6;
```

```
/* This program changes a two-dimensional array to the
   corresponding one-dimensional array.
      Written by:
      Date:
*/
#include <stdio.h>
#define ROWS 2
#define COLS 5

int main (void)
{
// Local Declarations
   int table [ROWS] [COLS] =
      {
       {00, 01, 02, 03, 04},
       {10, 11, 12, 13, 14}
      }; // table
   int line [ROWS * COLS];
```

```
// Statements
    for (int row = 0; row < ROWS; row++)
        for (int column = 0; column < COLS; column++)
            line[row * COLS + column] = table[row][column];

    for (int row = 0; row < ROWS * COLS; row++)
        printf(" %02d ", line[row]);

    return 0;
}   // main
```

```
Results:
00   01   02   03   04   10   11   12   13   14
```

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

```
/* This program fills the diagonal of a matrix (square
   array) with 0, the lower left triangle with -1, and
   the upper right triangle with 1.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int table [6][6];

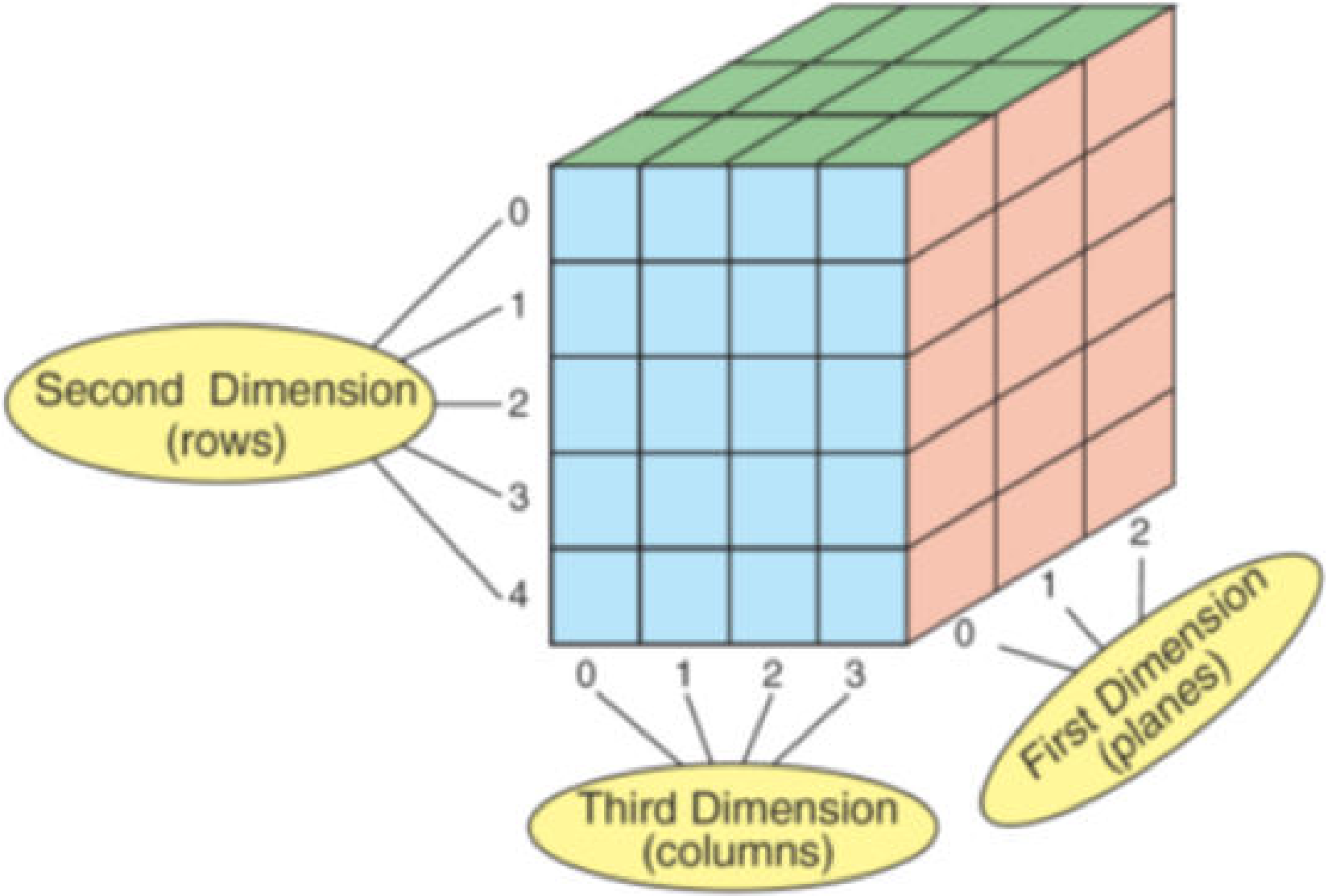
// Statements
   for (int row = 0; row < 6; row++)
      for (int column = 0; column < 6; column++)
```

```
        if (row ==  column)
            table [row][column] = 0;
        else if (row > column)
            table [row][column] = -1;
        else
            table [row][column] = 1;

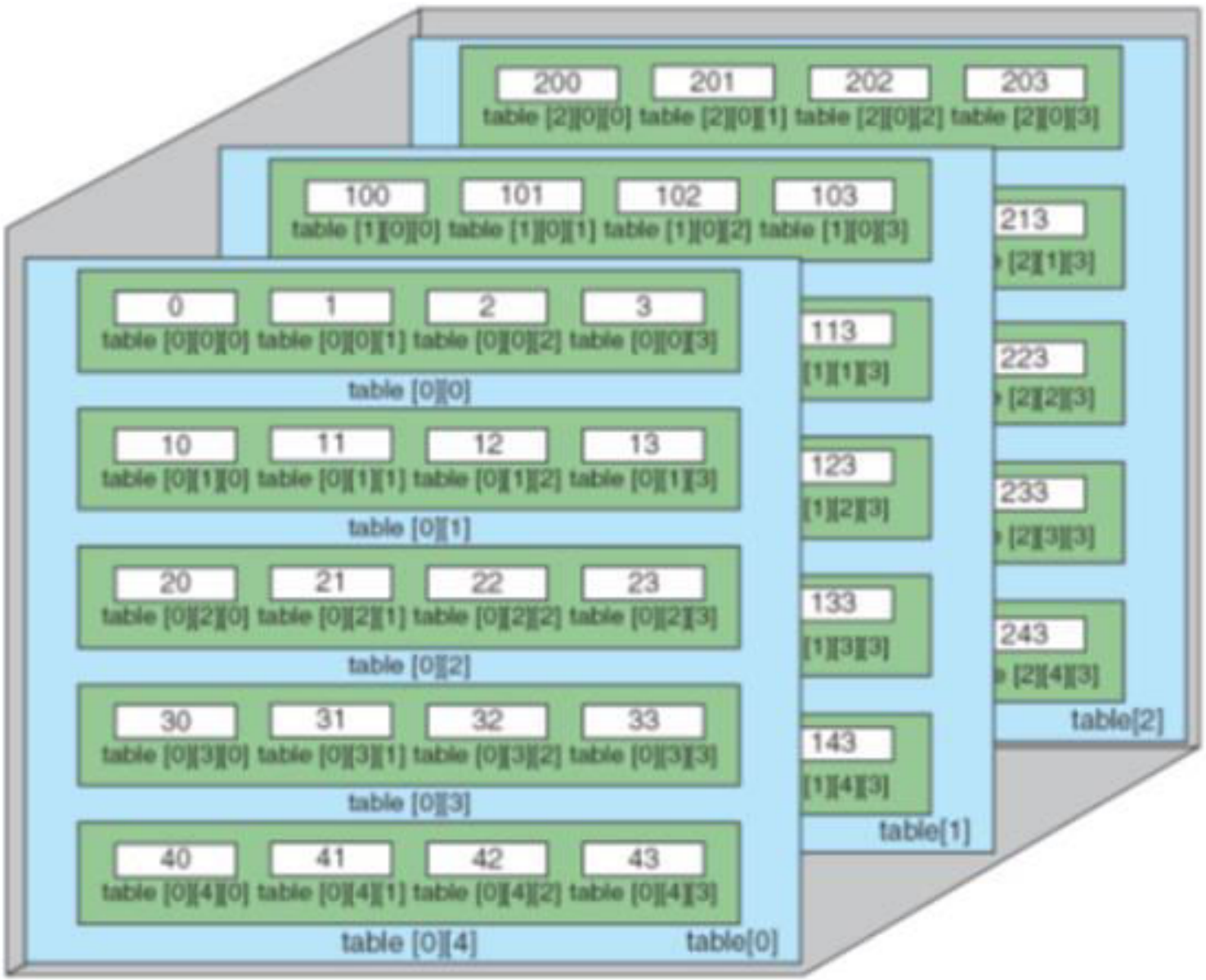
    for (int row = 0; row < 6; row++)
       {
        for (int column = 0; column < 6; column++)
           printf("%3d", table[row][column]);
        printf("\n");
       } // for row
    return 0;
} // main
```

8-8 Multidimensional Arrays

Multidimensional arrays can have three, four, or more dimensions. The first dimension is called a plane, which consists of rows and columns. The C language considers the three-dimensional array to be an array of two-dimensional arrays.



E 8-40 A Three-dimensional Array (3 x 5 x 4)



8-41 C View of Three-dimensional Array

multi-dimensional Arrays

- A variable which represent the list of items using more than two index (subscript) is called multi-dimensional array.

- The general form of multi dimensional array is :

  type *array-name[s1][s2][s3].....[sn];*

multi-dimensional Arrays

- Where S is the size. Some examples are :

```
int  survey[3][5][6];

float  table[5][4][5][3];
```

FUNCTIONS

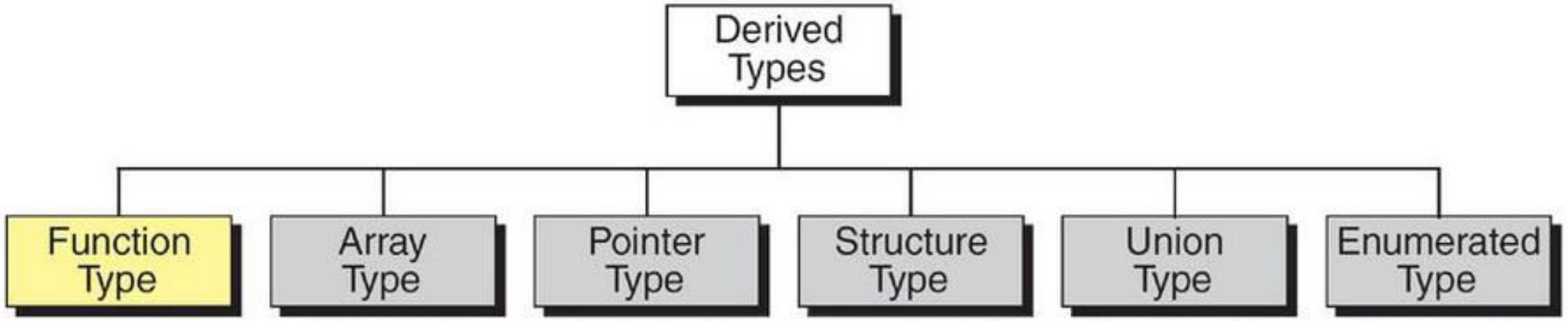
UNIT-3: Functions

Objectives

- To understand the software engineering principles of Structured programming and modularity (top-down development)
- To understand the function declaration, function call, and function definition
- To understand inter- function communication through parameters
- To understand the four basic function designs
- To understand how function communicate through parameters
- To understand Standard functions
- To understand the differences between global and local scope

Derived data types

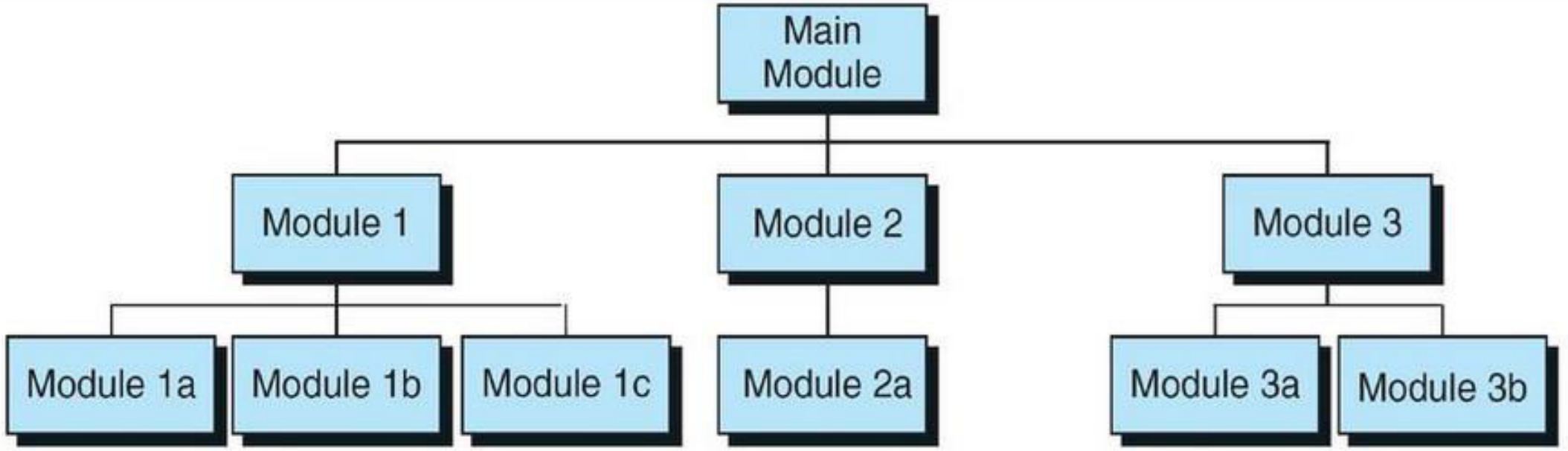
- Limited number of problems can be solved by using primary data types
- Derived data types are needed to solve complex problems.
- Complex data types are derived from primary data types
- Derived data types available in C language are given below:



Designing Structured Programs

- Simple programs studied so far can be solved and understood without too much effort.
- Large programs need to be reduced into elementary parts for better understanding.
- The principles of top-down design and structured programming dictate that a program should have a main module and its related modules.
- Each module can be further divided in to sub modules.
- Breaking a complex problem into smaller parts is known as *factoring.*

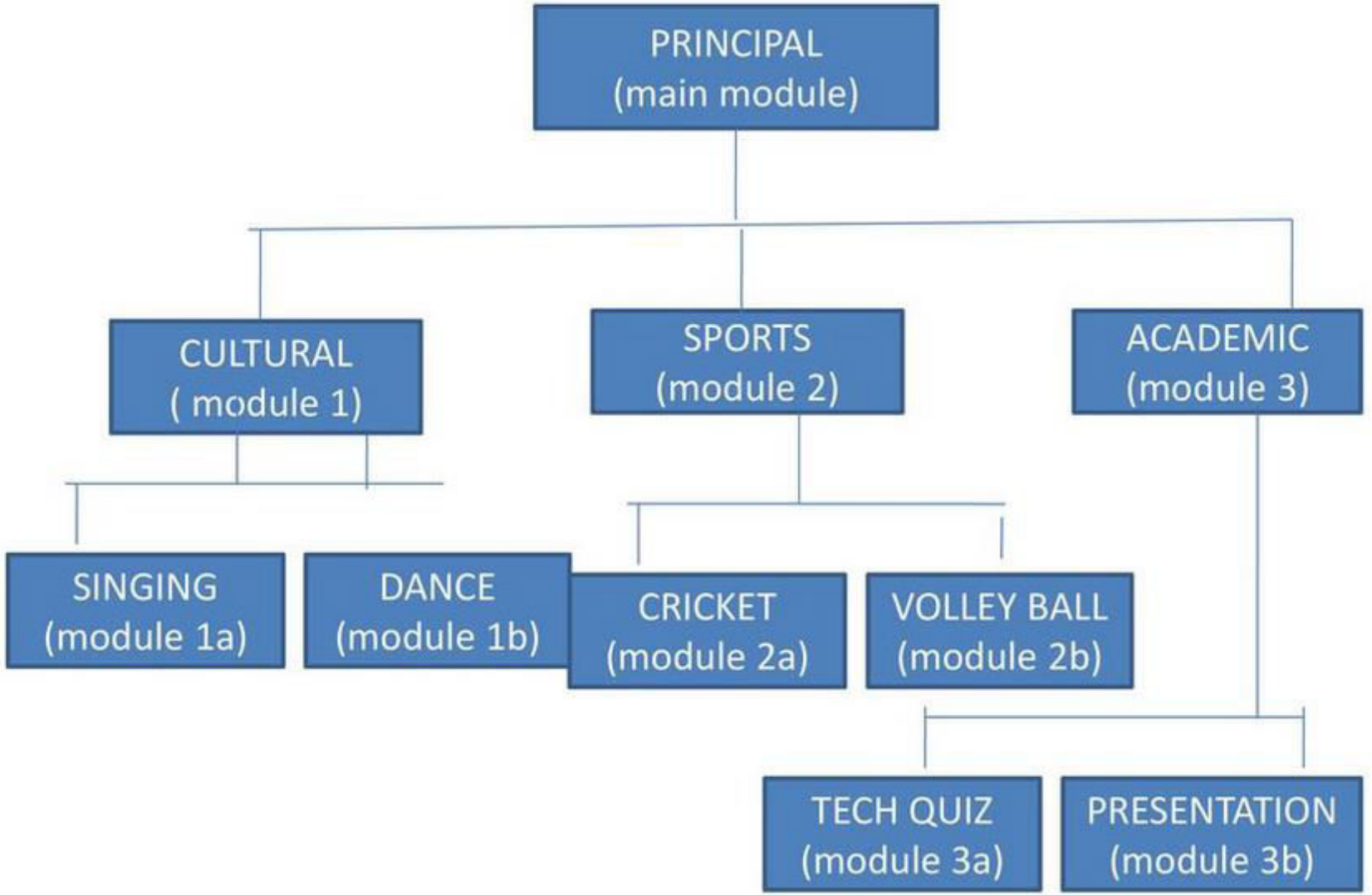
Hierarchy of modularization



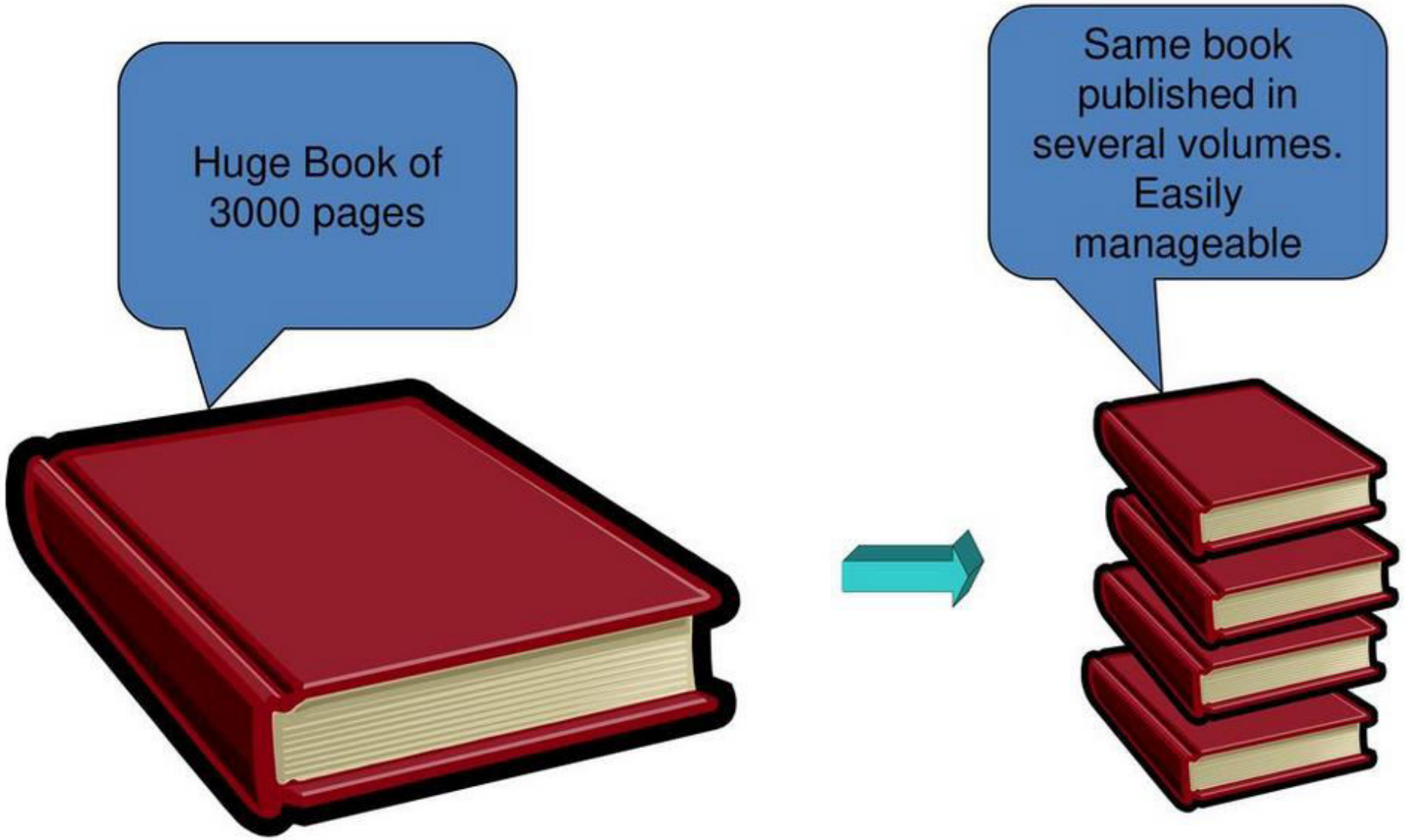
- Module 1,2,3 are sub modules of main module
- Module 1a,1b,1c are sub modules of module1
- Module 2a is sub module of module 2
- Module 3a,3b are sub modules of module 3

Modularization

Example:



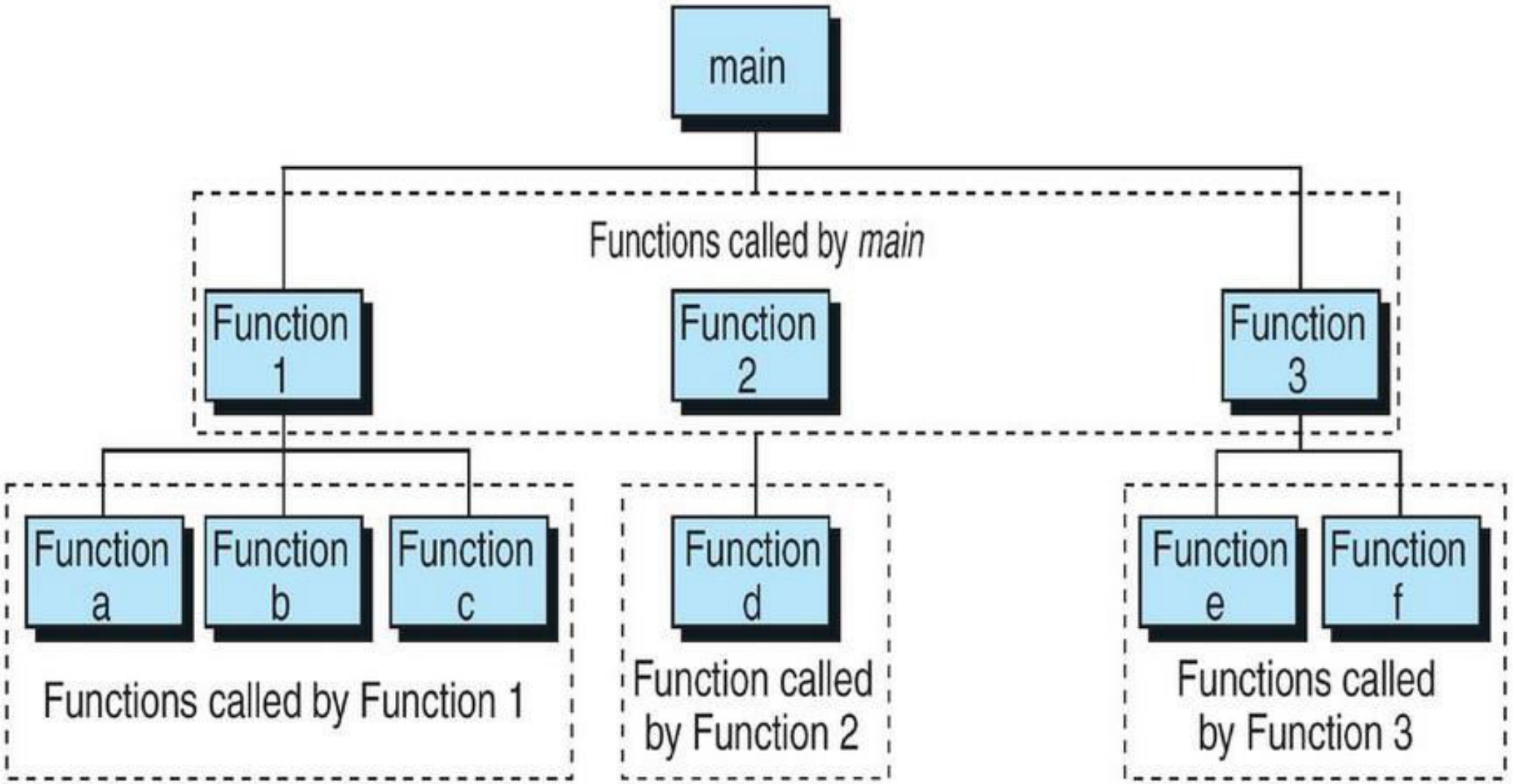
Modular Programming - Manageable



Functions in C

- Top-down design is implemented by using Functions in C
- A Program in C is made up of one or more functions, one and only one of which must be named as main.
- The execution of the program always starts and ends with main, but it can call other functions to do specific task.
- main ( ) is called by the OS and control returns to the OS after completion of main function.

Structure Chart for a C Program



Functions in C

- A function is a section of a program that performs a specific task
- Solving a problem using different functions makes programming much simpler with fewer defects
- A function receives zero or more pieces of data, operate on them and return at most one piece of data

Advantages of Functions (1 of 2)

- Problems can be factored in to understandable and manageable steps(easy to code and debug).
- Functions can be developed by different people and can be combined together as one application.
- Functions support reusability. That is, once a function is written, it can be called from any other module without having to rewrite the same. This saves time in rewriting the same code.

Intro

- Advantages of using functions
  - *avoids the need for redundant (repeated) programming of the same instructions.*
  - *easier to write and easier to debug.*
  - *logical structure is more apparent than programs which lack this type of structure, especially true of lengthy, complicated programs.*
  - *avoids repetitive programming between programs*

Function Overview

- Every C program consists of one or more functions.
  - *One of these functions must be called* main.
- A function will carry out its intended action whenever it is **accessed** or **called** from some other portion of the program.
- The same function can be accessed from several different places within a program.
- Once the function has carried out its intended action, control will be returned to the point from which the function was accessed.

Function Overview

- Two type of functions
  - ➢ *Library functions*
    - used to carry out a number of commonly used operations such as *printf(…)*, *getchar()*, and *putchar()*.
  - ➢ *User/Programmer-defined functions*
    - used to define their own functions for carrying out various individual tasks.
- Information is passed to the function via special identifiers called **arguments** (also called parameters), and returned via the return statement.

Library Functions

- 4 categories:
  - *Library functions that carry out standard I/O operations (e.g., read and write characters, and numbers, etc.).*
  - *Functions that perform operations on characters (e.g., convert from lower- to uppercase, test for end of file, etc.).*
  - *Functions that perform operations on strings (e.g., copy a strings, compare strings, concatenate strings, etc.).*
  - *Functions that carry out various mathematical calculations (e.g. evaluate trigonometric, logarithmic and exponential functions,, etc.).*

Advantages of Functions (2 of 2)

James working on enquiry module

Susan working on cash withdrawal module

George working on cash deposit module

Code Integration

Application Development

Function Definition (1 of 4)

- A function is always associated with following three steps
  - Function definition
  - Function declaration
  - Function call
- Function definition contains the code for a function
- Function definition is made up two parts
  - Function Header
  - Function body (Compound statement within opening and closing braces)

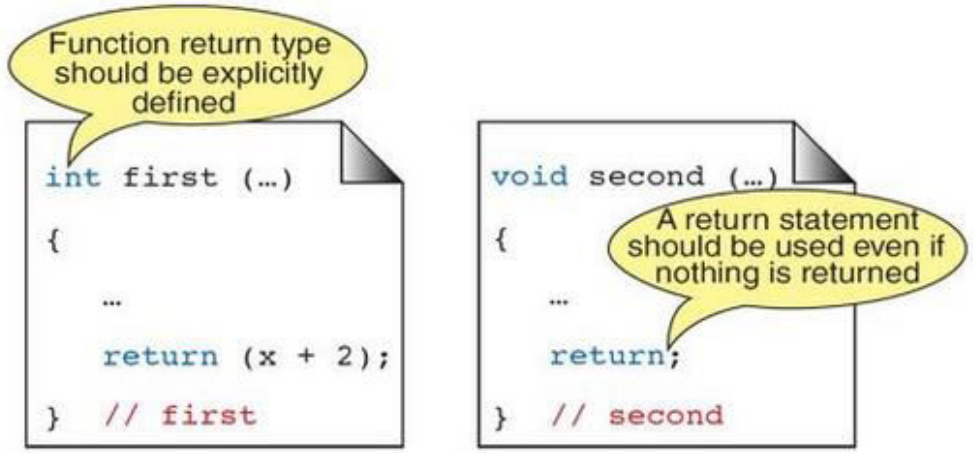
Function Definition (2 of 4)

Function Header

- A function header consist of 3 parts
  - return type
  - function name
  - formal parameter list
- A semicolon is not used at end of function definition header

Function Body

- The function body contains local declarations and function statements in between braces.
- Local declarations specify the variables needed by the function.
- The function statement terminated by a return statement
- If the function return type is void, a return statement is optional.



Function Definition (3 of 4)

Parameters

- Formal parameters are variables that are declared in the header of the function definition
- Actual parameters are used in the calling statement or sometimes they may be an expression
- Formal and actual parameters must match exactly in type, order, and number. Their names, however, no need to match.

Function Definition (4 of 4)

Parameters

Formal and actual parameters must match exactly in type, order, and number. Their names, however, need not match.

```
double average(int x,int y);
void main()
{
double sum1;
sum1=average(10,20);// calling function
}
```

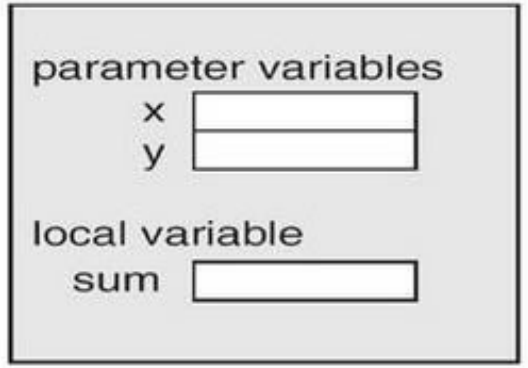
actual parameters

x,y formal parameters

Two values received from calling function

```
double average (int x,int y)
{
    double sum;
    sum = x + y;
    return (sum / 2);
}   // average
```

One value returned to calling function



Execute

Function Declaration (1 of 2)

- Function declaration consists only of a function header (no code)
- Function declaration header consists of three parts: the return type, the function name and formal parameters
- Function declarations are terminated with a semicolon
- Declarations are placed in global declaration section before the main function
- Function declaration is also known as function prototype

Function Header

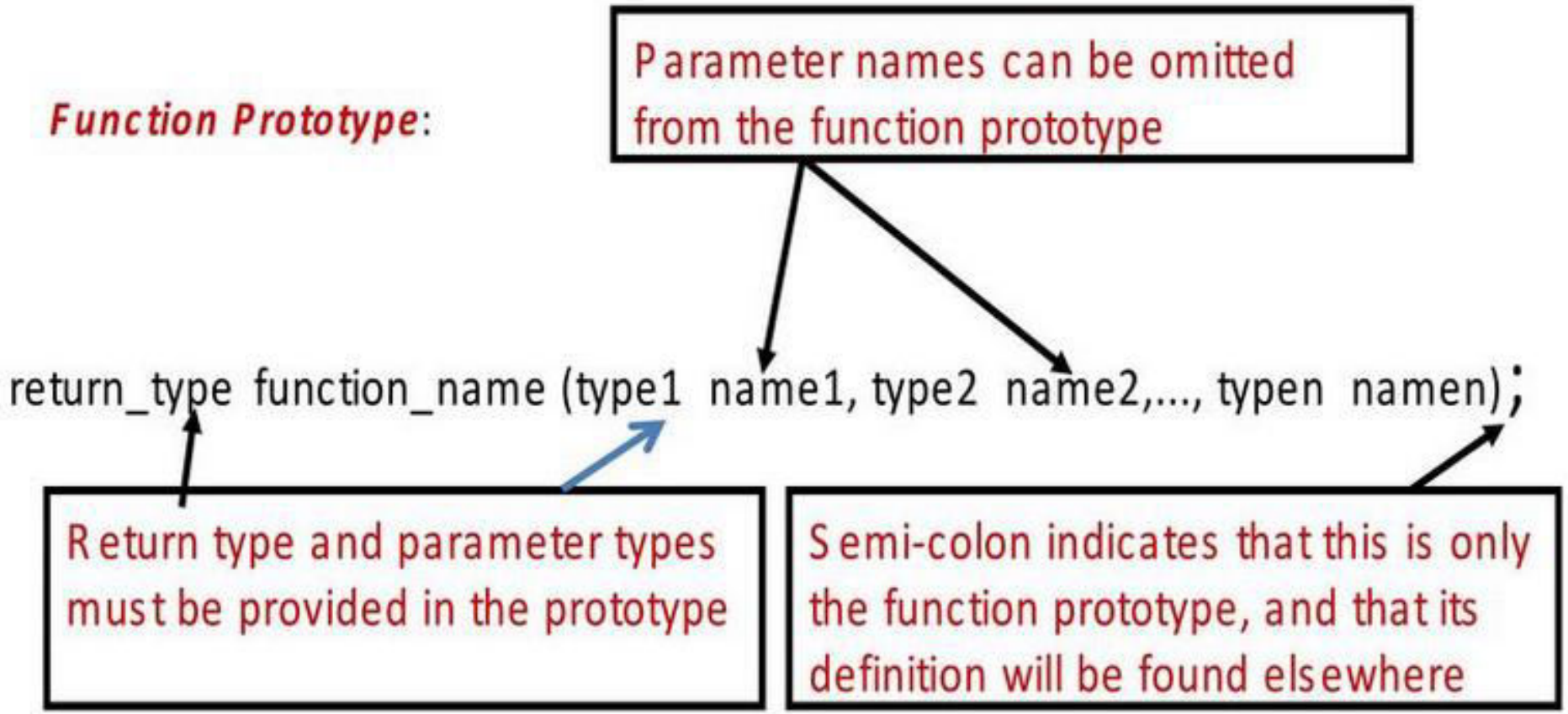
```
return_type function_name (formal parameter list)
```

```
{
// Local Declarations
   …
// Statements
   …
}   // function_name
```

Function Body

Function Declaration (2 of 2)

The general form of a function Prototype or Declaration

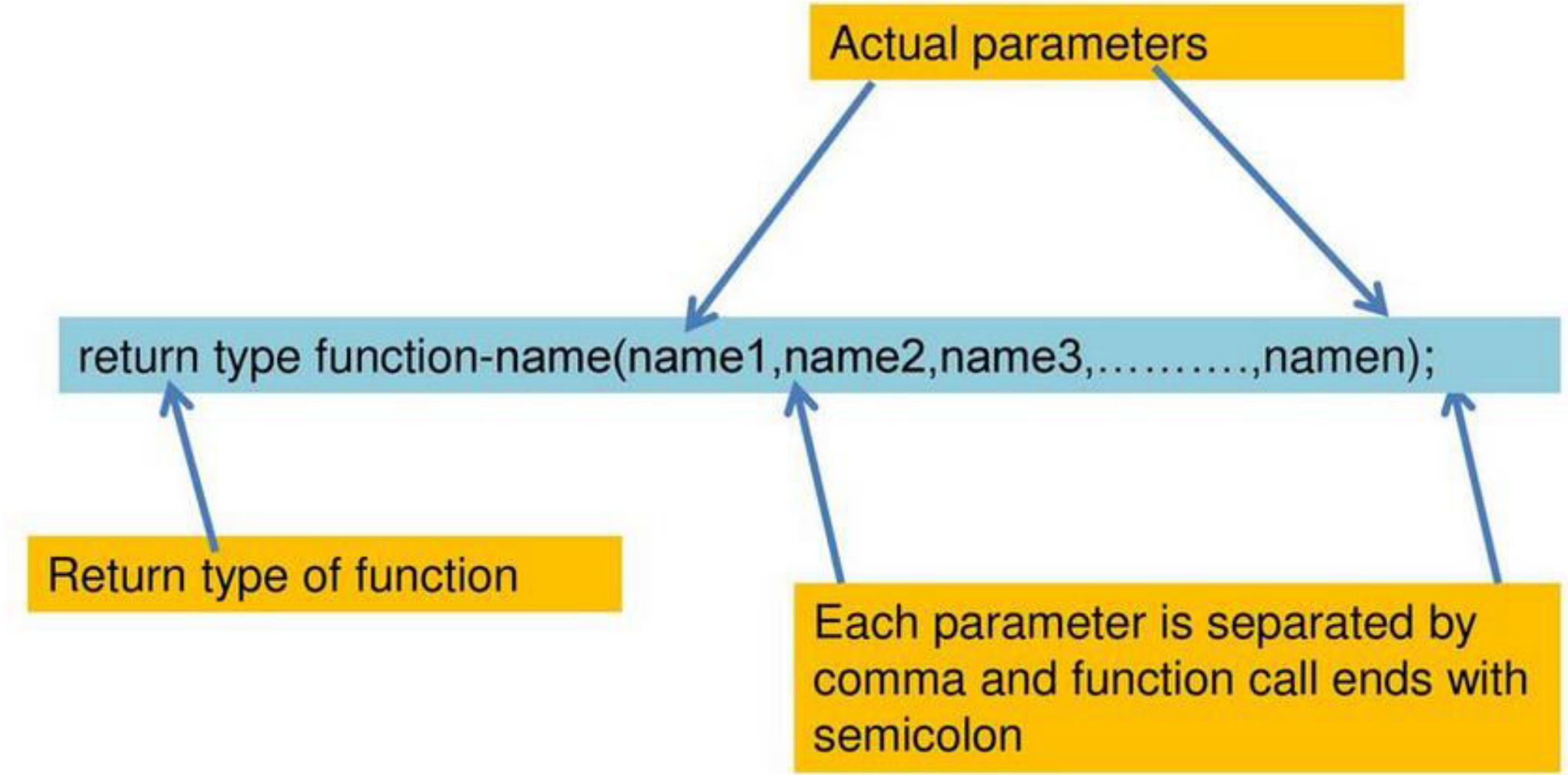


Function Call (1 of 4)

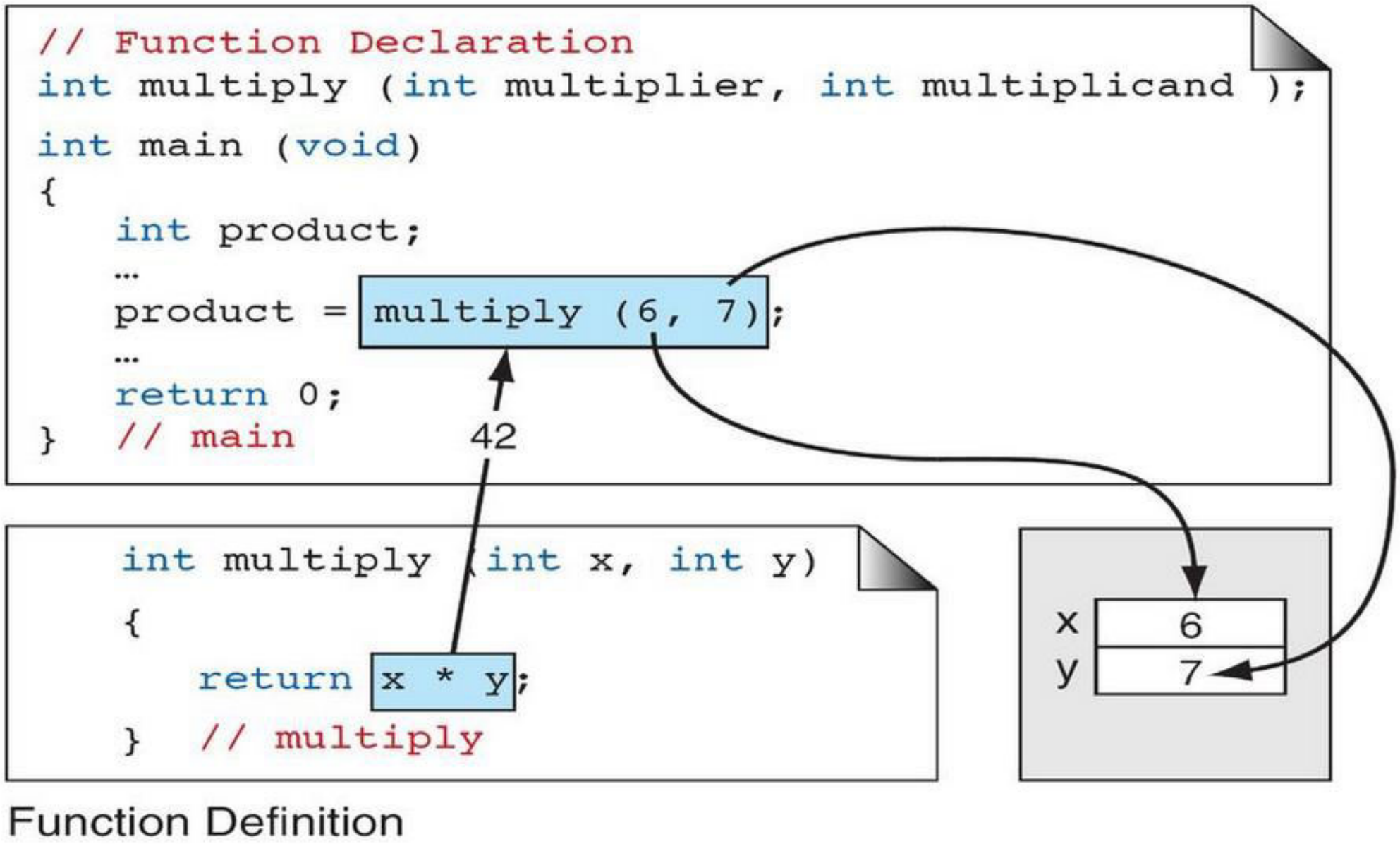
- Function call is a postfix expression
- The operand in a function call is the function name; the operator is the parentheses set (…), which contains actual parameters.
- Actual parameters identify the values that are to be sent to called function and must match the formal parameters in type and order
- Multiple actual parameters are separated by comma
- Function call transfers the control to the function definition (called function)
- After execution of the function, it returns the result to the calling function
- Function having a return type void can be used only as a stand-alone statement

Function Call (2 of 4)

General form



Function Call (3 of 4)



Function Definition

Function Call (4 of 4)

Examples of Function Calls

multiply (6, 7)

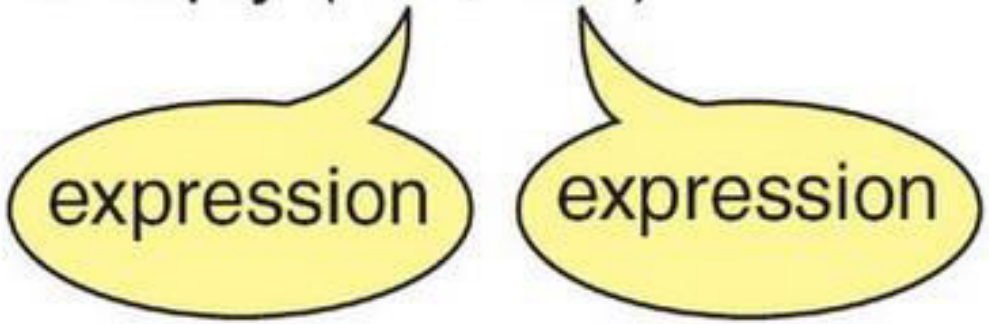
multiply (6, b)

multiply (multiply (a, b), 7)

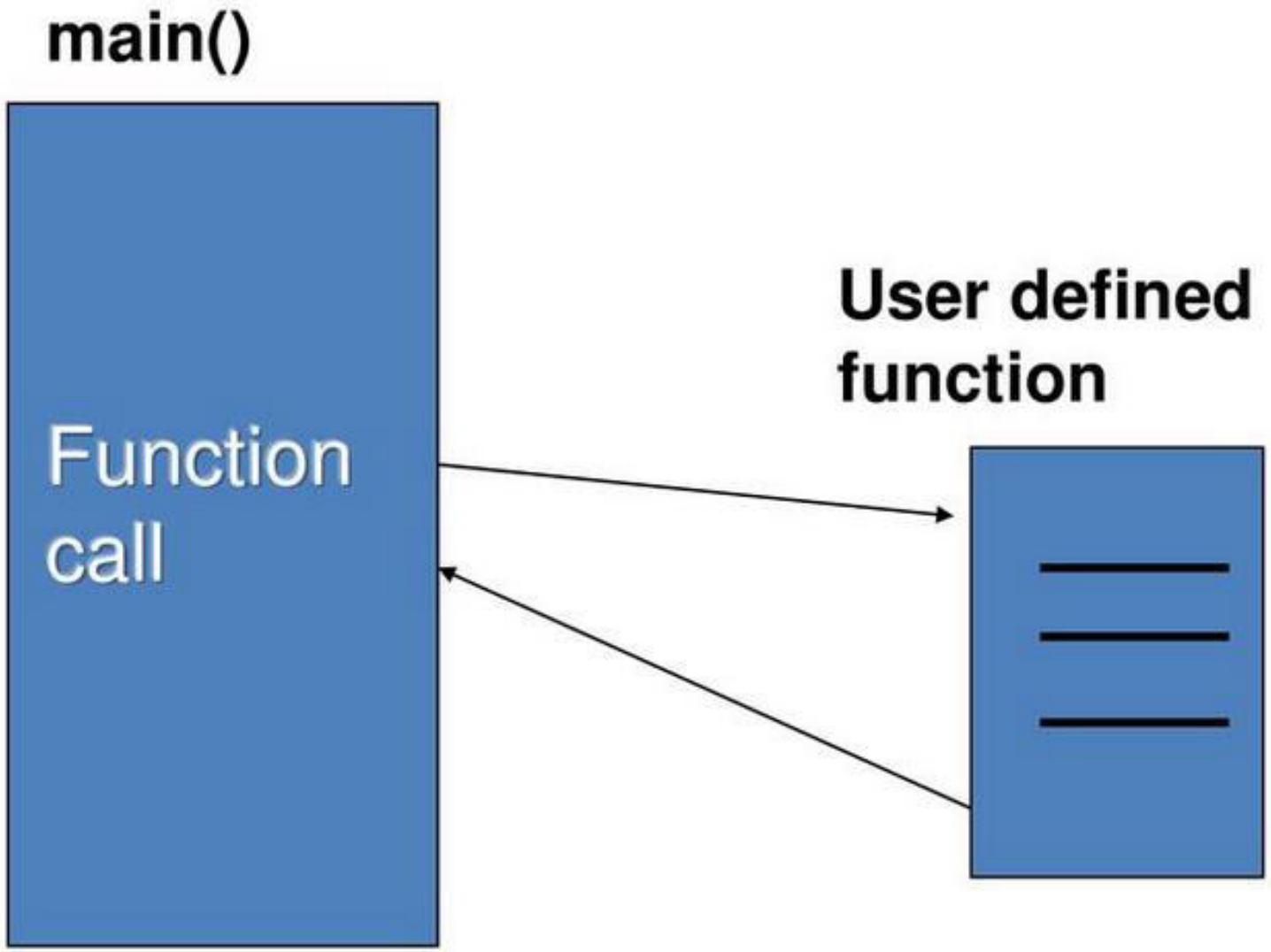
multiply (a, 7)

multiply (a + 6, 7)

multiply (… , …)



How Functions Work?



Sample Program with function

```
1  /* This program demonstrates function calls by calling
2     a small function to multiply two numbers.
3        Written by:
4        Date:
5  */
6  #include <stdio.h>
7
8  // Function Declarations
9  int multiply (int num1, int num2);
10
11 int main (void)
12 {
13 // Local Declarations
14    int multiplier;
15    int multiplicand;
16    int product;
17
```

Sample Program with function

```
18  // Statements
19     printf("Enter two integers: ");
20     scanf ("%d%d", &multiplier, &multiplicand);
21
22     product = multiply (multiplier, multiplicand);
23
24     printf("Product of %d & %d is %d\n",
25                multiplier, multiplicand, product);
26     return 0;
27  }  // main
28
29  /* ================== multiply ====================
30     Multiples two numbers and returns product.
31        Pre   num1 & num2 are values to be multiplied
32        Post  product returned
```

Sample Program with function

```
33  */
34  int multiply (int num1, int num2)
35  {
36  // Statements
37      return (num1 * num2);
38  }  // multiply
```

```
Results:
Enter two integers: 17 21
Product of 17 & 21 is 357
```

User Defined Functions

- Functions must be both declared and defined.
- Function declaration gives the whole picture of the function .
- Function declaration must be placed before the function call.

- Function declaration consists of:
  - name of the function,
  - return type,
  - type and order of parameters

- Function definition contains the code that needs to complete the task

Declaring, Calling and Defining Functions

```
// Function Declaratio
void greeting (void);
int main (void)
{
// Statements
   greeting( );    // call
   return 0;
} // main
```

```
void greeting (void)
{
   printf("Hello World!");
   return;
} // greeting
```

Back to Operating System

Side Effect

Basic Function designs (1 of 5)

- Basic function design is based on their return value and parameter list
- There are four basic designs

1. Void Functions without Parameter.
    - Function with no arguments and no return value.
2. Void Functions with Parameter.
    - Functions with arguments and no return values.
3. Non Void Functions without Parameters
    - Functions with no argument and return values.
4. Non Void Functions with Parameters
    - Functions with argument and return values.

Basic Function designs (1 of 5)

	Do not pass argument	Do pass arguments

No return	`void main(void)`
	`{`
	`  TestFunct();`
	`  ...`
	`}`
	
	`void TestFunct(void)`
	`{`
	`  // receive nothing`
	`  // and nothing to be`
	`  // returned`
	`}`
		`void main(void)`
		`{`
		`  TestFunct(123);`
		`  ...`
		`}`
		
		`void TestFunct(int i)`
		`{`
		`  // receive something and`
		`  // the received/passed`
		`  // value just`
		`  // used here. Nothing`
		`  // to be returned.`
		`}`

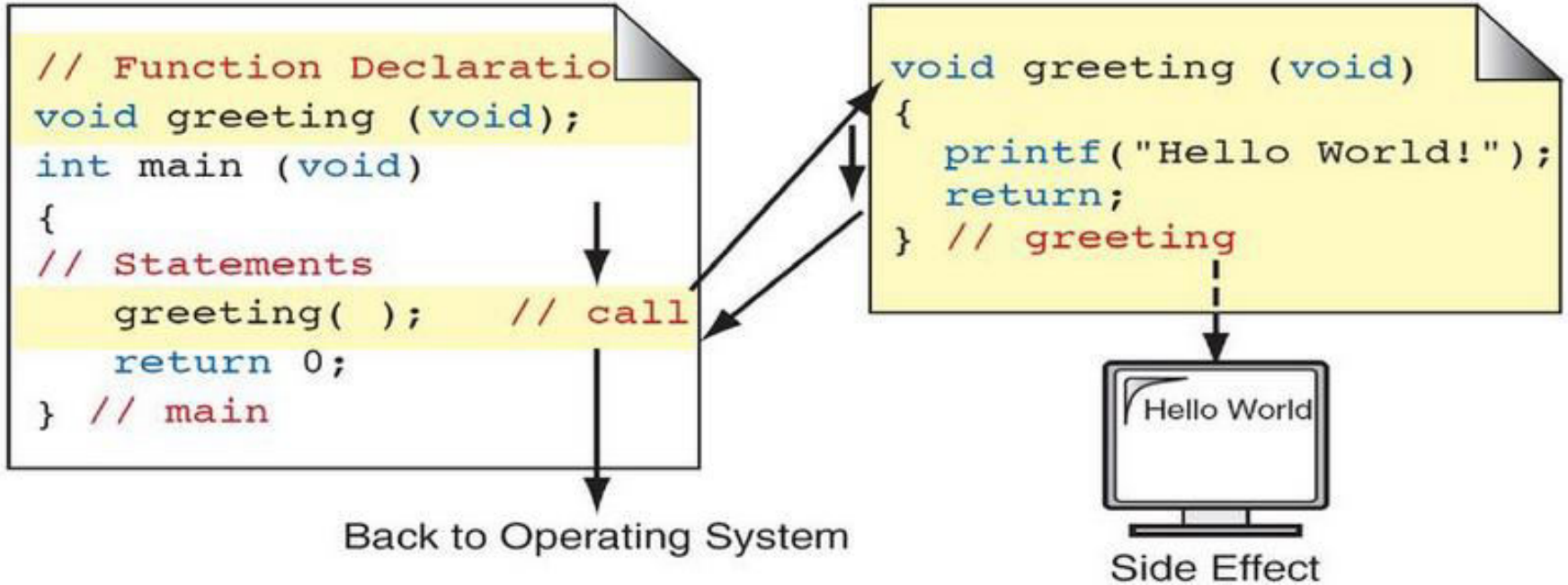
With a return	`void main(void)`
	`{`
	`  x = TestFunct();`
	`  ...`
	`}`
	
	`int TestFunct(void)`
	`{`
	`  // received/passed`
	`  // nothing but need to`
	`  // return something`
	`  return 123;`
	`}`
		`void main(void)`
		`{`
		`  x = TestFunct(123);`
		`  ...`
		`}`
		
		`int TestFunct(int x)`
		`{`
		`  // received/passed something`
		`  // and need to return something`
		`  return (x + x);`
		`}`

Basic Function designs (2 of 5)

1. Void Functions without Parameters

- Void functions without parameters does not receive any parameters and also does not return any value .
- This can be used only as a statement because a Void function does not return a value
- This cannot be used in an expression

  Example : result=greeting(); //error



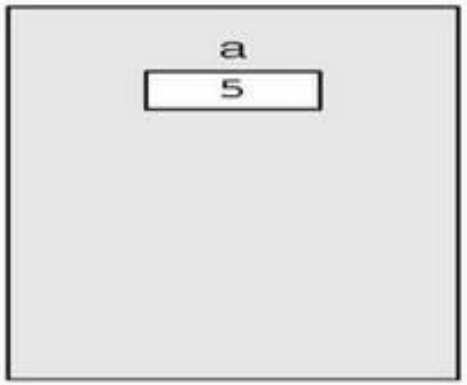
Basic Function designs (3 of 5)

2. Void Functions with Parameter

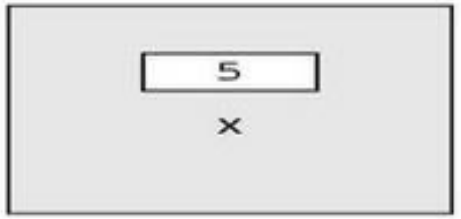
- The function that receives parameter but does not return any value
- This function takes arguments(parameter list)
- It can be used only as statement
- It cannot be used in an expression

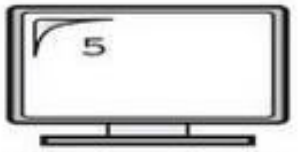
Example: result=greeting(); //error

```
// Function Declaration
void printOne (int x);
int main (void)
{
// Local Declarations
    int   a = 5;
// Statements
    printOne (a);     // call
    return 0;
}   // main
```




```
void printOne (int x)
{
    printf("%d\n", x);
    return;
} // printOne
```





Basic Function designs (4 of 5)

3. Non Void Functions without Parameters

- The function returns a value but does not receive parameters
- It cannot be used as statement
- It can be used in an expression

Example: result=greeting(); //correct

```
// Function Declaration
int getQuantity (void);

int main (void)
{
// Local Declarations
   int amt;

// Statements
   amt = getQuantity ( );
   ...
   return 0;
}  // main
```

```
int getQuantity (void)
{
// Local Declarations
   int qty;

// Statements
   printf("Enter Quantity");
   scanf ("%d", &qty);
   return qty;
}  // getQuantity
```

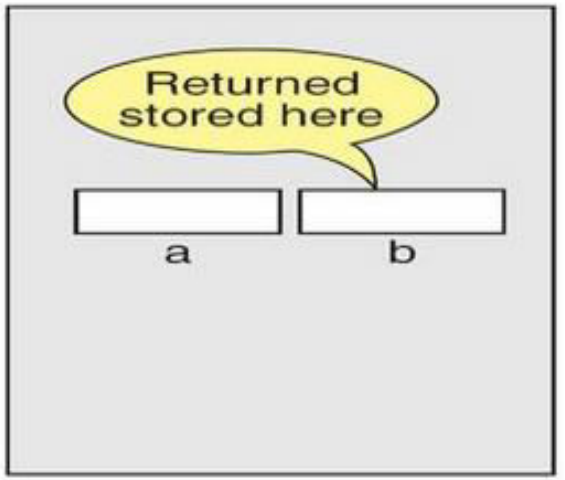
Basic Function designs (5 of 5)

4. Non Void Function with Parameters

- The function receives parameters and return values
- It cannot be used as a statement
- It can be used in an expression

Example : result=greeting(); //correct

```
// Function Declaration
int sqr (int x);
int main (void)
{
// Local Declarations
    int a;
    int b;
// Statements
    scanf("%d", &a);
    b  =  sqr (a);
    printf("%d squared: %d\n", a,  b);
    return 0;
}   // main
```




```
int sqr (int x)
 {
// Statements
    return (x * x);
}   // sqr
```

SCOPE (1 of 4)

- Scope determines the region of the program in which a defined object is visible.
- Scope pertains to any object that can be declared, such as a variable or a function declaration.
- An object can have three levels of scope

    1. Block scope
    2. Function scope
    3. File scope

Scope (2 of 4)

Block Scope

- Variables defined within a block have a local scope. They are visible in that block only. Outside the block they are not visible. Eg:
- ```
  { //outer block starts
          {//inner block starts

          }//inner block ends
  } //outer block ends
  ```
- Variables are in scope from their point of declaration until the end of the block. Block scope is also referred as local scope.

- Variables that are declared in a block are known as local variables.
  ```

Scope (3 of 4)

Function Scope

- Variables defined within a function (including main) are local to this function and no other function has direct access to them.
- Eg: int add(int a, int b) //add() is a function

```
{ //function scope starts


}//function scope ends
```

Scope (4 of 4)

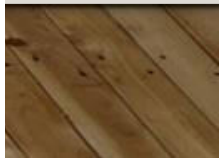
File Scope

- File scope includes the entire source file for a program, including any files that are part of it.
- An object with file scope has visibility through the whole source file in which it is declared.
- Objects within block scope are excluded from file scope unless specifically declared to have file scope; in other words, by default, block scope hides objects from file scope.
- An object with file scope sometimes referred to as a global object and the variables declared inside this scope are referred to as Global Variables.

Extent

- Defines the duration for which the computer allocates memory for the variable.
- It is also known as storage duration.
- Extents are of two types:

1. Automatic Extent: object is created each time when it is declared and it is destroyed each time when the block is exited.
2. Static Extent : object is created when the program is loaded for execution and it is destroyed when the execution stops.



Linkage

- A large application program may be broken into several modules, with each module potentially written by a different programmer. Each module is a separate source file with its own objects.
- We can define two types of linkages:

  -internal

  -external

- Internal Linkage: An object with an internal linkage is declared and visible in one module. Other modules cannot refer to this object.

- External Linkage: An object with an external linkage is declared in one module but is visible in all other modules with a special keyword, extern.

C - Scope Rules

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed.

There are three places where variables can be declared in C programming language –

- Inside a function or a block which is called local variables.
- Outside of all functions which is called global variables.
- In the definition of function parameters which are called formal parameters.

Arrays to functions

- To process arrays in a large program, arrays are required to be passed to functions.

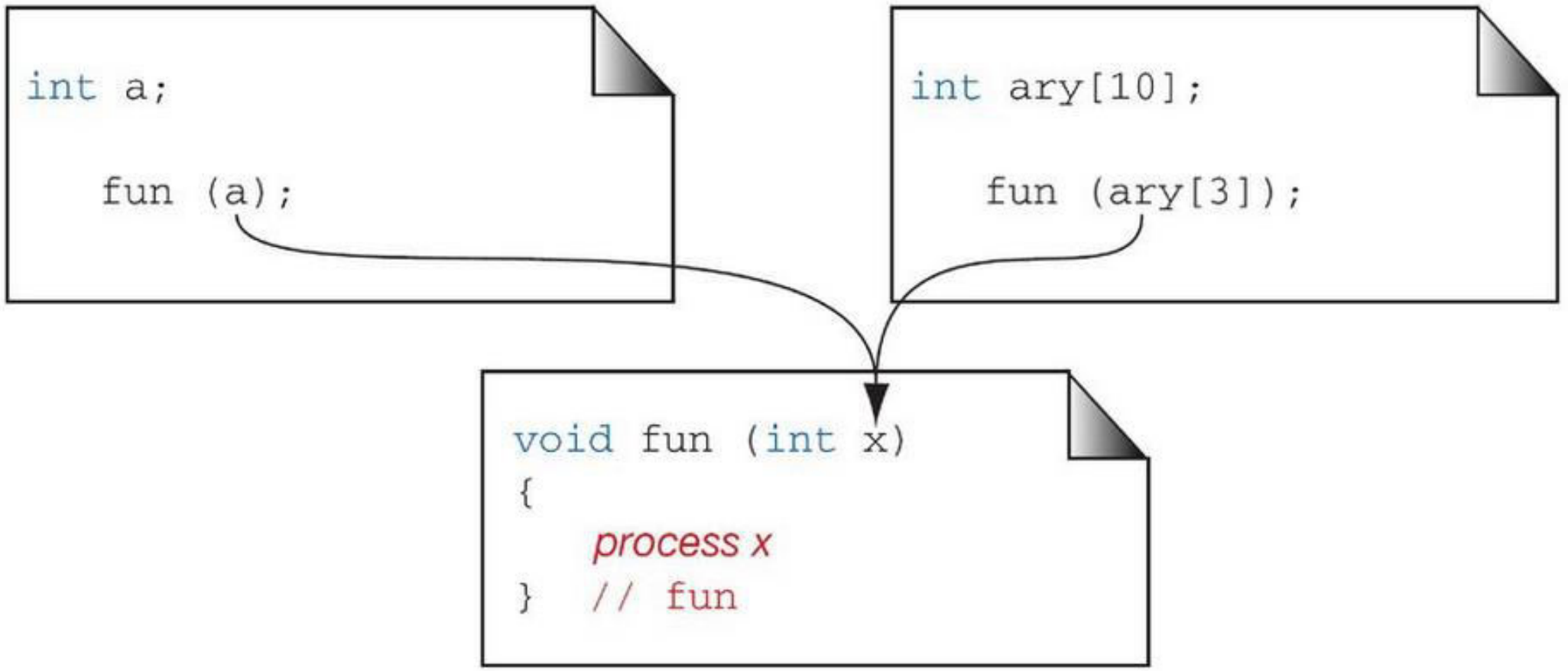
- Arrays can be passed to functions in two ways:

- **Passing Individual Elements**
  - Individual elements can be passed to a function either by passing the data values or by passing the addresses

- **Passing the whole Array**
  - By using this, an Array name can be passed to the called function

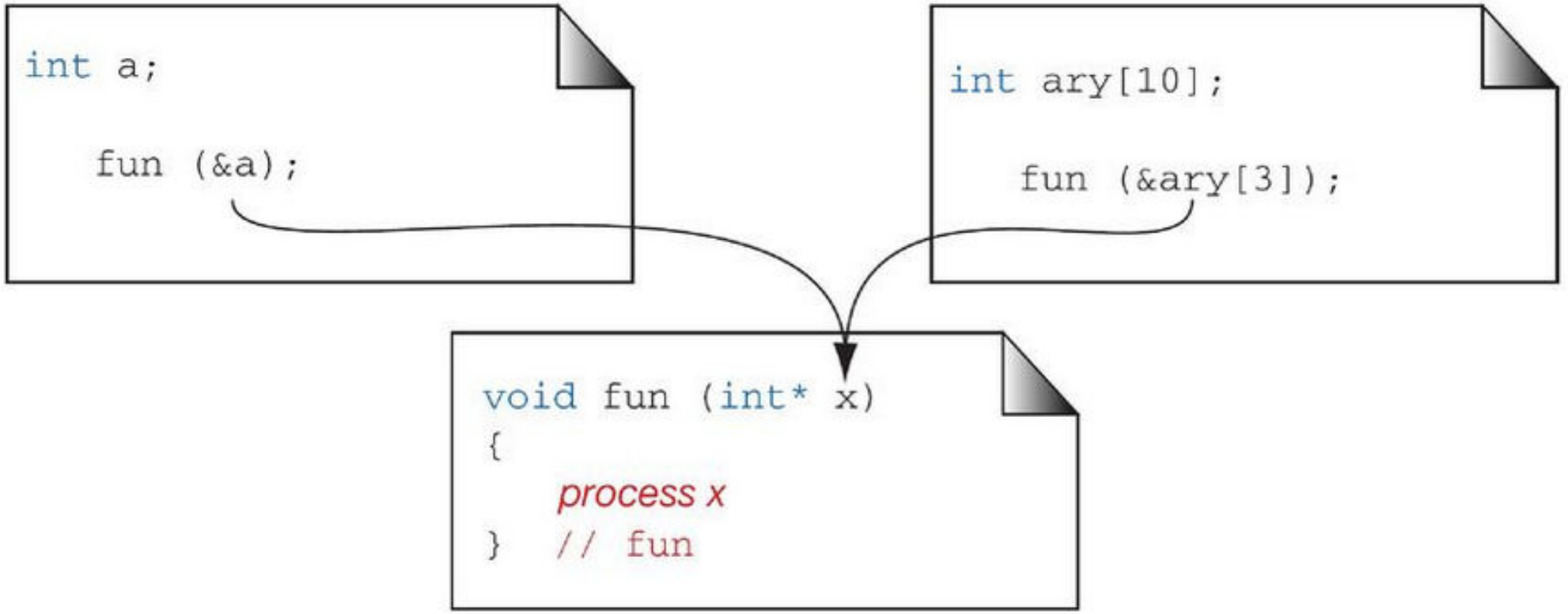
Arrays to functions

Passing Array Elements



Arrays to functions

Passing the Address of an Array Element



Arrays to functions

Passing the Whole Array

```
int ary[10];

   fun (ary);
```

```
void fun (int fAry[ ])
{
    process x
}   // fun
```

Fixed-size Array

```
int ary[size];

   fun (ary);
```

```
void fun (int fAry[*])
{
    process x
}   // fun
```

Variable-size Array

Recursion

- There are two approaches to write repetitive algorithms: one approach uses loops and the other one uses recursion
- Recursion is a repetitive process, in which a function can call itself.
- Recursive functions are useful in evaluating certain types of mathematical functions.
- Recursion is also a useful way of creating and accessing dynamic data structures such as linked lists or binary trees.

Designing recursive function

- Recursive functions have two steps:
  - each call either solves one part of the problem
  - it reduces the size of the problem
- Every recursive function must have a base case. The statement that solves the problem is known as the base case.
- The rest of the function is known as the general case.
- The recursion will be done until the problem converges to the simplest case.
- This simplest case will be solved by the function which will then return a value.
- Each recursive call reduces the size of the problem

Rules for designing recursive function

- First, determine the base case
- Then, determine the general case
- Finally, combine the base case and general case into a function.

Example: Factorial of a given number

Iterative Definition

- Factorial(n)= $\begin{cases} 1 & \text{if } n=0 \\ n*(n-1)*(n-2)\ldots 3*2*1 & \text{if } n>0 \end{cases}$

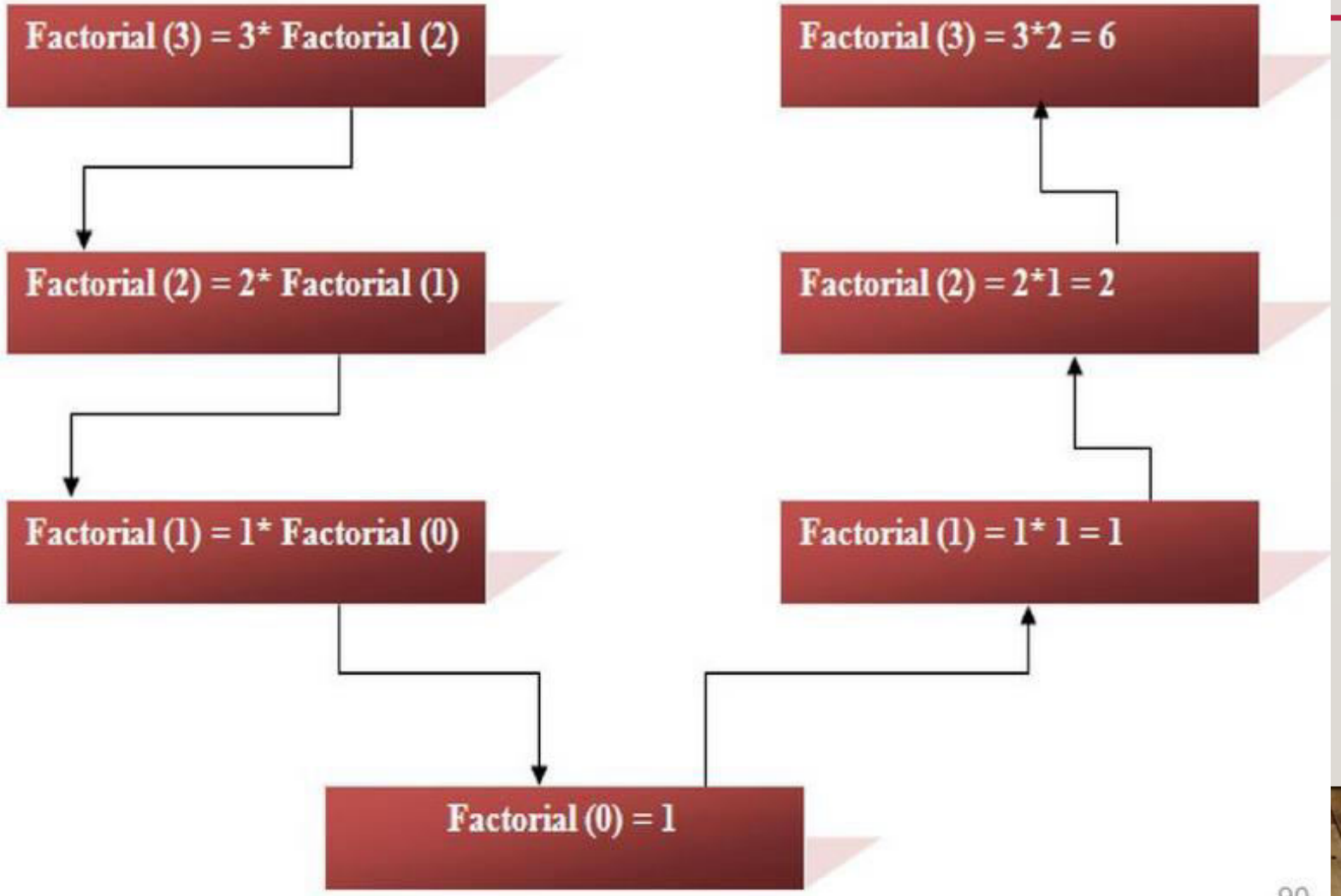
Recursive Definition

Factorial(n) = $\begin{cases} 1 & \text{if } n=0 \\ n*\text{Fcatorial}(n-1) & \text{if } n>0 \end{cases}$

Base Case

General Case

Recursive representation of Factorial(3) (1 of 2)



Recursive representation of Factorial(3) (2 of 2)

```
factorial(0) = 1;
factorial(n) = n*factorial(n-
```

factorial(3) = 3 * factorial(2)

= 3 * (2 * factorial(1))

= 3 * (2 * (1 * factorial(0)))

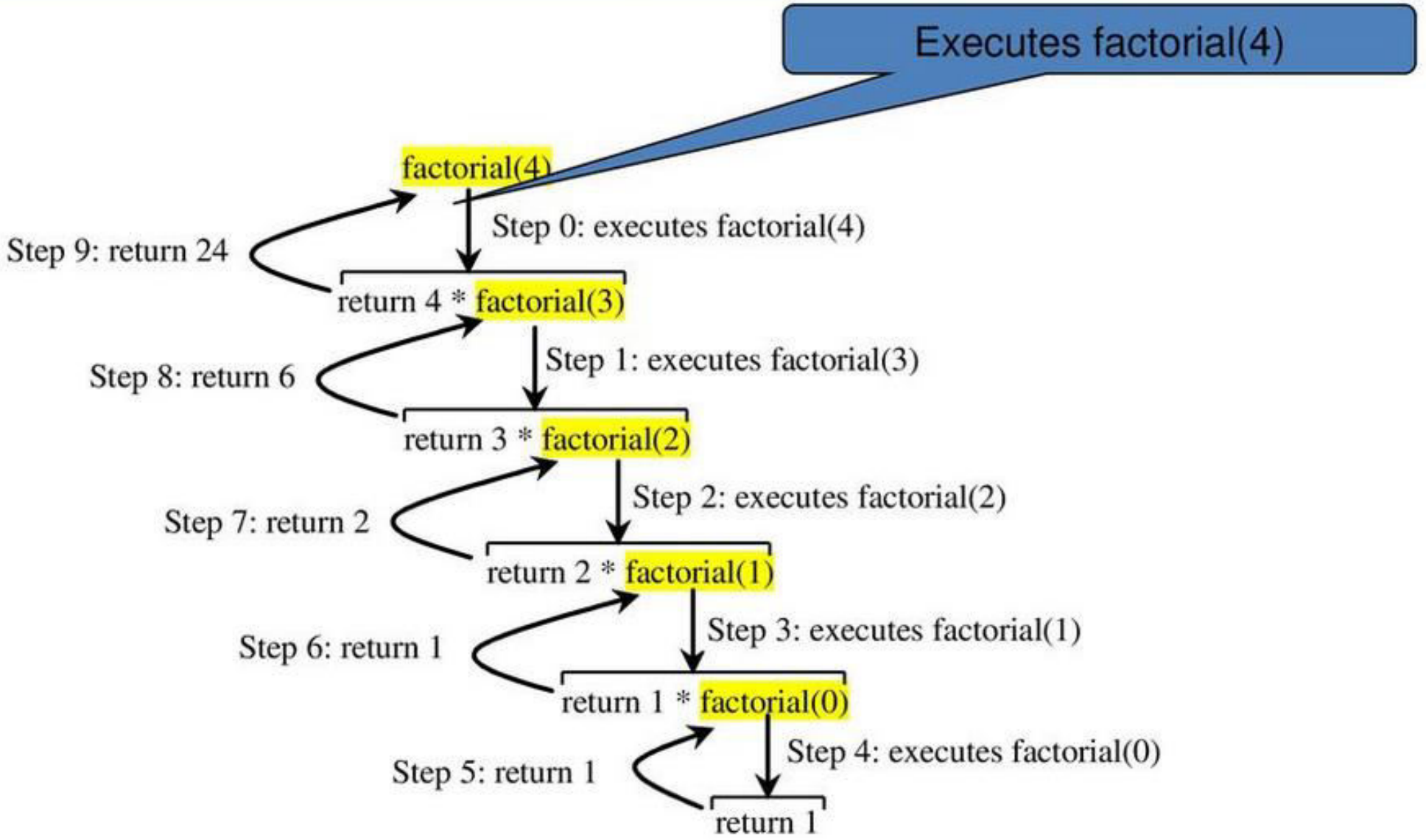
= 3 * (2 * (1 * 1)))

= 3 * (2 * 1)

= 3 * 2

= 6

Recursive representation of Factorial(4) (1 of 11)



```
long factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return(n * factorial(n-1));
}
```

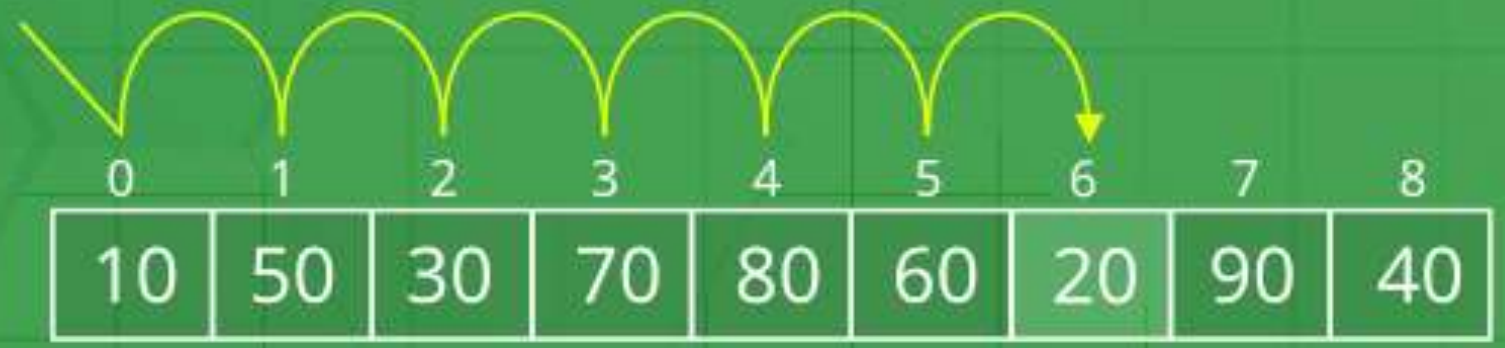
Differences between Iteration and Recursion

ITERATION	RECURSION
Iteration explicitly uses repetition structure.	Recursion achieves repetition by calling the same function repeatedly.
Iteration is terminated when the loop condition fails	Recursion is terminated when base case is satisfied.
May have infinite loop if the loop condition never fails	Recursion is infinite if there is no base case or if base case never reaches.
Iterative functions execute much faster and occupy less memory space.	Recursive functions are slow and takes a lot of memory space compared to iterative functions

Searching Algorithm

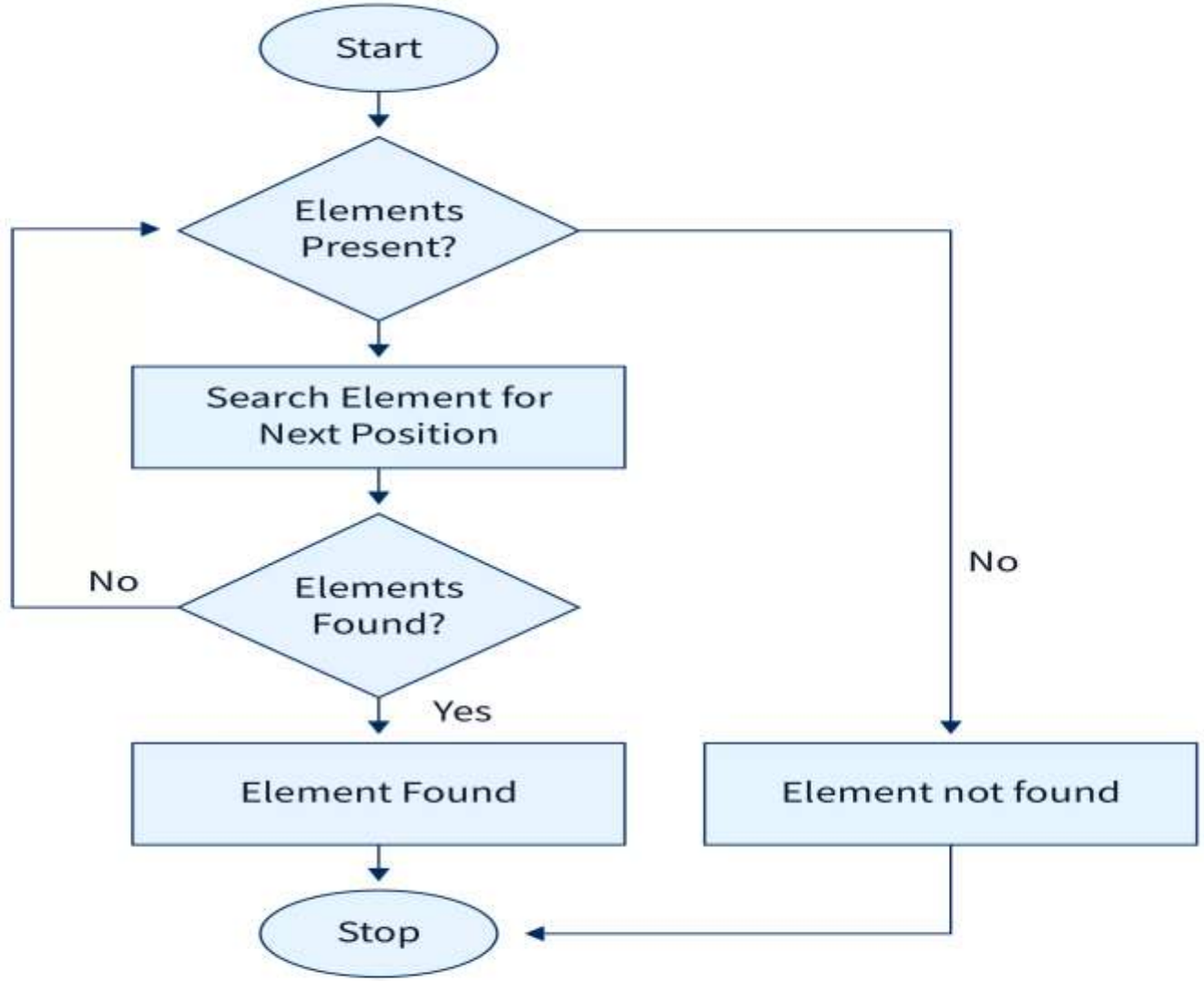
Linear Search

Find '20'



Linear Search

- Set the first element of the array as the current element.
- If the current element is the target element, return its index.
- If the current element is not the target element and if there are more elements in the array, set the current element to the next element and repeat step 2.
- If the current element is not the target element and there are no more elements in the array, return -1 to indicate that the element was not found.



SCALER Topics

```
#include <stdio.h>

int search(int arr[], int N, int x)
{
    int i;
    for (i = 0; i < N; i++)
        if (arr[i] == x)
            return i;
    return -1;
}
```

```
int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int N = sizeof(arr) / sizeof(arr[0]);

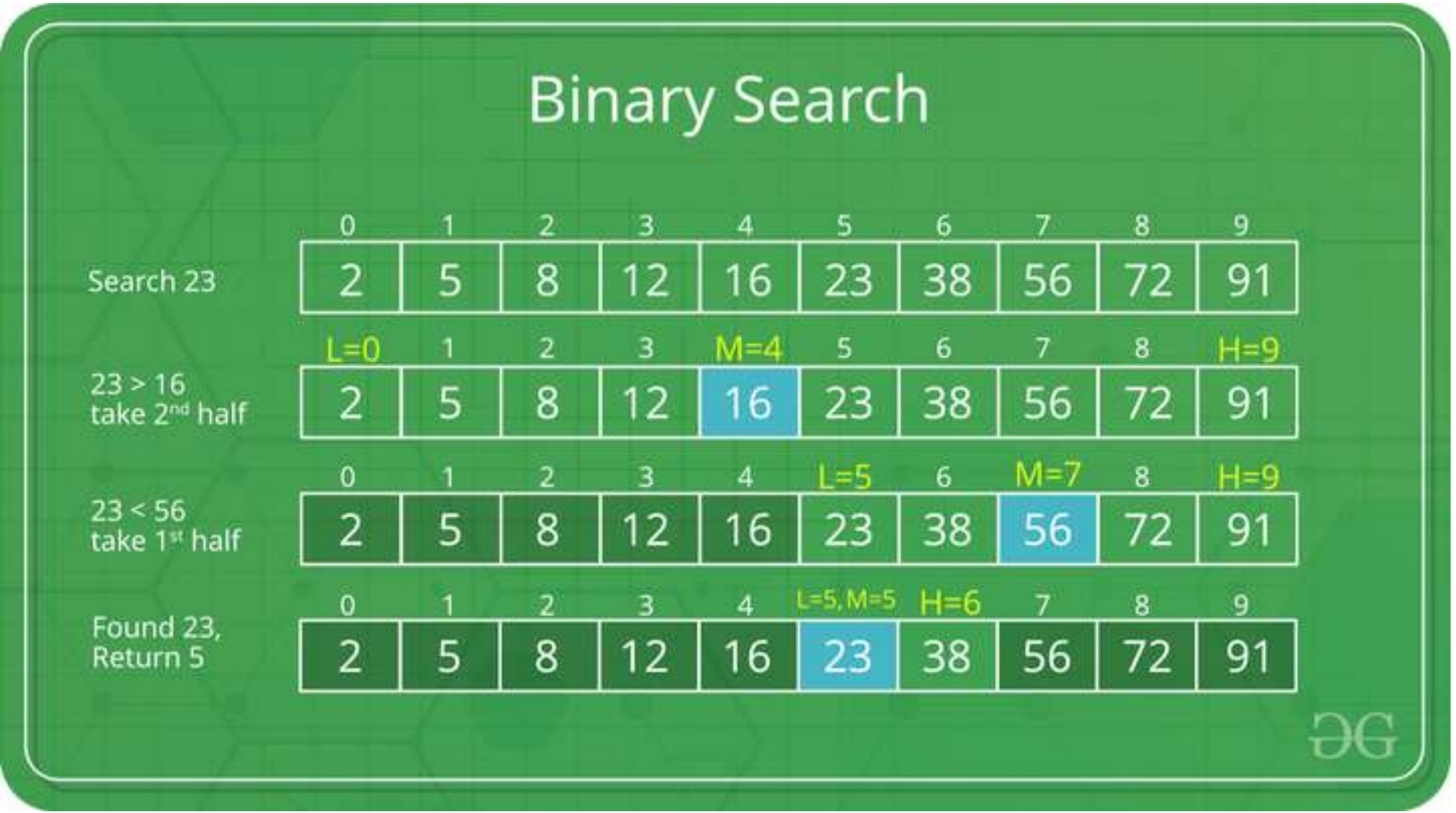
    // Function call
    int result = search(arr, N, x);
    (result == -1)  ? printf("Element is not present in array")
        : printf("Element is present at index %d", result);
    return 0;
}
```

```
int linearsearch(int arr[], int size, int key)
{
    if (size == 0) {
        return -1;
    }
    else if (arr[size - 1] == key) {
        // Return the index of found key.
        return size - 1;
    }
    else {
        int ans = linearsearch(arr, size - 1, key);
        return ans;
    }
}
```

Binary Search

- Sort the array in ascending order.
- Set the low index to the first element of the array and the high index to the last element.
- Set the middle index to the average of the low and high indices.
- If the element at the middle index is the target element, return the middle index.
- If the target element is less than the element at the middle index, set the high index to the middle index – 1.
- If the target element is greater than the element at the middle index, set the low index to the middle index + 1.
- Repeat steps 3-6 until the element is found or it is clear that the element is not present in the array.

Binary Search



```
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;

        // If the element is present at the middle
        // itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);
```

```
            // Else the element can only be present
            // in right subarray
            return binarySearch(arr, mid + 1, r, x);
        }

        // We reach here when element is not
        // present in array
        return -1;
    }
```

```
int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1)
        ? cout << "Element is not present in array"
        : cout << "Element is present at index " << result;
    return 0;
}
```

```c
#include <stdio.h>
int main()
{
int i, low, high, mid, n, key, array[100];
printf("Enter number of elementsn");
scanf("%d",&n);
printf("Enter %d integersn", n);
for(i = 0; i < n; i++)
scanf("%d",&array[i]);
printf("Enter value to findn");
scanf("%d", &key);
```

```
low = 0;
high = n - 1;
mid = (low+high)/2;
while (low <= high) {
if(array[mid] < key)
low = mid + 1;
else if (array[mid] == key) {
printf("%d found at location %d.n", key, mid+1);
break;
}
else
high = mid - 1;
mid = (low + high)/2;
}
```

```
if(low > high)
printf("Not found! %d isn't present in the list.n", key);
return 0;
}
```