Introduction to Computers & Programming

Comp 1330/ First Semester 2024/2025

Instructor: Saif Harbia

Faculty of Engineering and Technology
Department of Computer Science

Chapter 07

Arrays

Chapter Objectives:

1. Declare and use arrays for **storing collections of values of the same type**.
2. Use a subscript to **reference the individual values in an array**.
3. **Process the elements of an array** in sequential order using loops
4. **Search** and **sort** an array
5. Multidimensional arrays
6. Declare and use your own data types

And more….

ARRAYS

- To solve many programming problems, it is more efficient to group data items together in main memory than to allocate an individual memory cell for each variable. (1)

- C allows a programmer to group such related data items together into a single composite **data structure** .

- We look at one such data structure: the **array**

7.1 DECLARING AND REFERENCING ARRAYS

- An **array** is a collection of two or more adjacent memory cells, called **array elements** , that are associated with a particular symbolic name.

- To set up an array in memory, we must declare both the name of the array and the number of cells associated with it.

- The declaration: **double x[8];**

  - Instructs the compiler to associate eight memory cells with the name **x**.(1)
  - Each element of array x may contain a single type double value (2)

7.1 DECLARING AND REFERENCING ARRAYS

- **The Subscript Variable**

  - Reference each individual element by specifying the array name and identifying the element desired.

  - The subscripted variable **x[0]** (1) may be used to reference the initial or **0th** element of the array **x**.

  - **x[1]** the next element, and **x[7]** the last element.

  - The **integer** enclosed in brackets is the **array subscript** , and its value must be in the range from 0 to one less than the number of memory cells in the array.

7.1 DECLARING AND REFERENCING ARRAYS

TABLE 7.1 Statements That Manipulate Array x

Statement	Explanation
`printf("%.1f", x[0]);`	Displays the value of `x[0]`, which is `16.0`.
`x[3] = 25.0;`	Stores the value `25.0` in `x[3]`.
`sum = x[0] + x[1];`	Stores the sum of `x[0]` and `x[1]`, which is `28.0` in the variable `sum`.
`sum += x[2];`	Adds `x[2]` to `sum`. The new `sum` is `34.0`.
`x[3] += 1.0;`	Adds `1.0` to `x[3]`. The new `x[3]` is `26.0`.
`x[2] = x[0] + x[1];`	Stores the sum of `x[0]` and `x[1]` in `x[2]`. The new `x[2]` is `28.0`.

Array x

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
16.0	12.0	6.0	8.0	2.5	12.0	14.0	–54.5

=>

Array x

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
16.0	12.0	28.0	26.0	2.5	12.0	14.0	–54.5

Before executing the statements in Table 7.1

After executing the statements in Table 7.1

7.1 DECLARING AND REFERENCING ARRAYS

- You can declare more than one array in a single type declaration.

```
double students[5], instructors, courses[6];
int factor[12], n, index;
```

ARRAY INITIALIZATION

- We can initialize an array in its declaration. (1)
- We can omit the size of an array that is being fully initialized. (2)
- For example, we initialize a 25-element array with the prime numbers less than 100.

```
int prime_100[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97};
```

- Array element **prime_100[24]** is **97**.

ARRAY DECLARATION

SYNTAX: element-type aname [size]; /* *uninitialized* */

element-type aname [size **] = { initialization list };** /* *initialized* */

EXAMPLE:

```
#define A_SIZE 5
. . .
double a[A_SIZE];
char vowels[] = {'A', 'E', 'I', 'O', 'U'};
```

- The general uninitialized array declaration allocates storage space for array **aname** consisting of size memory cells.

- Each memory cell can store one data item whose data type is specified by element-type (i.e., **double** , **int** , or **char**)

ARRAY DECLARATION

- The individual array elements are referenced by the subscripted variables **aname [0] , aname [1] , . . . , aname [size −1].**
- A constant expression of type **int** is used to specify an array's size.
- The *initialization list* consists of constant expressions of the appropriate *element-type* separated by commas.
- Element **0** of the array being initialized is set to the first entry in the initialization list , element **1** to the second, and so forth.
- In the initialized array declaration shown, the size shown in brackets is optional since the array's size can also be indicated by the length of the *initialization list* .

STORING A STRING IN AN ARRAY OF CHARACTERS

- You can store individual characters in an array by writing each character in the *initialization list*.

- If the list is long, you can do this more easily by using a string instead of an *initialization list.*

```
char vowels[] = "This is a long string";
```

- **vowels[0]** stores the character **'T'**
- **vowels[1]** stores the character **'h'** ,

and so on.

7.2 ARRAY SUBSCRIPTS

- We can use any expression of type **int** as an array subscript.

- To create a valid reference, the value of this subscript must lie between 0 and one less than the declared size of the array.

- **Table 7.2 p.380** lists some sample statements involving the array **x** below.

Array x

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
16.0	12.0	6.0	8.0	2.5	12.0	14.0	–54.5

- It is the programmer's responsibility to verify that the subscript is within the declared range. (1)

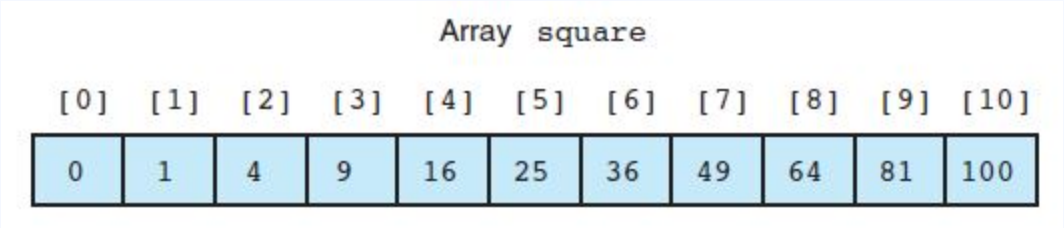
7.3 USING FOR LOOPS FOR SEQUENTIAL ACCESS

- In C, we can process elements of an array easily using an **indexed for loop.**
- **A counting loop** whose loop control variable runs from 0 to one less than the array size.
- Using the loop counter as an array index (subscript) gives access to each array element in turn.

```
#define SIZE 11;
int square[SIZE], i;

for (i = 0; i < SIZE; ++i)
    square[i] = i * i;
```

=>



STATISTICAL COMPUTATIONS USING ARRAYS

- **Figure 7.2, p.382.**
- **TABLE 7.3 p.385:** Partial Trace of Computing for Loop
- In this example, The variable **i** is used as the loop control variable and array subscript in each loop.

- The variable **i** , which is the counter of each indexed for loop, determines which array element is manipulated during each loop repetition.

- The use of the loop control variable as an array subscript is common, because it allows the programmer to specify easily the sequence in which the elements of an array are to be manipulated. (1)

7.4 USING ARRAY ELEMENTS AS FUNCTION ARGUMENTS

- The code below uses array element **x[i]** as an input argument to function **printf**. (1)

```
printf("%3d%4c%9.2f%5c%9.2f\n", i, ' ', x[i], ' ',x[i] - mean);
```

- The code below uses array element **x[i]** as an output argument of **scanf** (2)

```
scanf("%lf", &x[i]);
```

- You can also pass array nctions that you write.
- Each array element must correspond to a formal parameter that is the same simple type as the array element.

7.4 USING ARRAY ELEMENTS AS FUNCTION ARGUMENTS

- EXAMPLE 7.6, p.386

```
void do_it (double arg_1, double *arg2_p, double *arg3_p){
*arg2_p = 5.0;
*arg3_p = 6.5;
```

do_it() function definition

```
do_it (n, &q, &r);
```

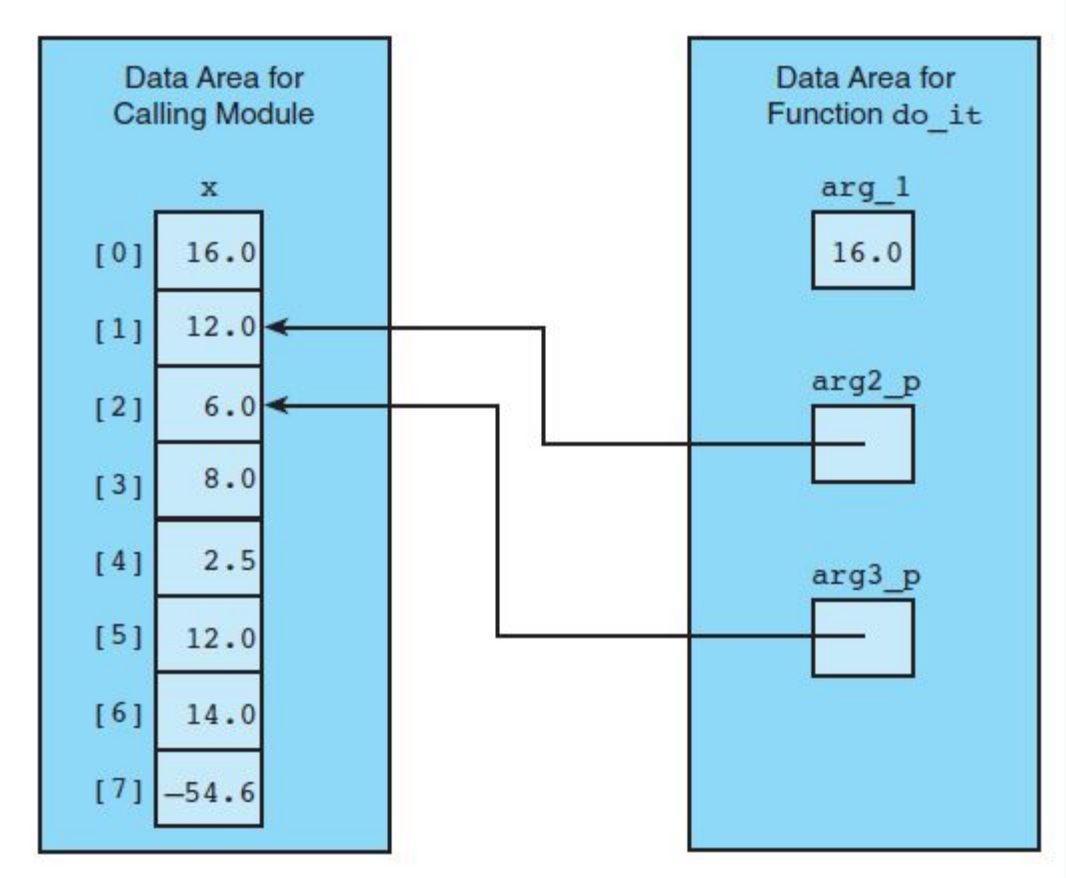
do_it() function call

```
do_it(x[0], &x[1], &x[2]);
```

Another do_it() function call

7.4 USING ARRAY ELEMENTS AS FUNCTION ARGUMENTS

- Figure 7.3



7.5 ARRAY ARGUMENTS

- We can write functions that have **arrays** as arguments. (1)
- **Formal Array Parameters:**
- When an array name with no subscript appears in the argument list of a function call, what is actually stored in the function's corresponding formal parameter is the address of the initial array element.
- We can use subscripts with the formal parameter to access the array's elements. (2)
- The function manipulates the original array, not its own personal copy (3)

7.5 ARRAY ARGUMENTS

- **EXAMPLE 7.7**

FIGURE 7.4 Function fill_array

```
1.  /*
2.   * Sets all elements of its array parameter to in_value.
3.   * Pre: n and in_value are defined.
4.   * Post: list[i] = in_value, for 0 <= i < n.
5.   */
6.  void
7.  fill_array (int list[],    /* output - list of n integers          */
8.              int n,         /* input - number of list elements      */
9.              int in_value)  /* input - initial value                */
10. {
11.
12.       int i;              /* array subscript and loop control     */
13.
14.       for  (i = 0; i < n; ++i)
15.           list[i] = in_value;
16. }
```

7.5 ARRAY ARGUMENTS

- **EXAMPLE 7.7:**

- In function **fill_array** , the array parameter is declared as
  int list[]

- Notice that the parameter declaration does not indicate how many elements are in list .

- Because C does not allocate space in memory for a copy of the actual array, the compiler does not need to know the size of the array parameter.

- In fact, since we do not provide the size, we have the flexibility to pass to the function an array of any number of integers.

ARGUMENT CORRESPONDENCE FOR ARRAY PARAMETERS

- If **y** is an array with ten type **int** elements, the following function call stores the value of **num** in the ten elements of array **y**

```
fill_array(y, 10, num);
```

- If **x** is a five-element array of type **int** values,

```
fill_array(x, 5, 1); (1)
```

```
fill_array(&x[0], 5, 1);(2)
```

- However, the call on the right may lead the reader of the code to expect that **fill_array** may be using only the array element **x[0]** as an output argument. (3)

ARGUMENT CORRESPONDENCE FOR ARRAY PARAMETERS

fill_array(x, 5, 1);

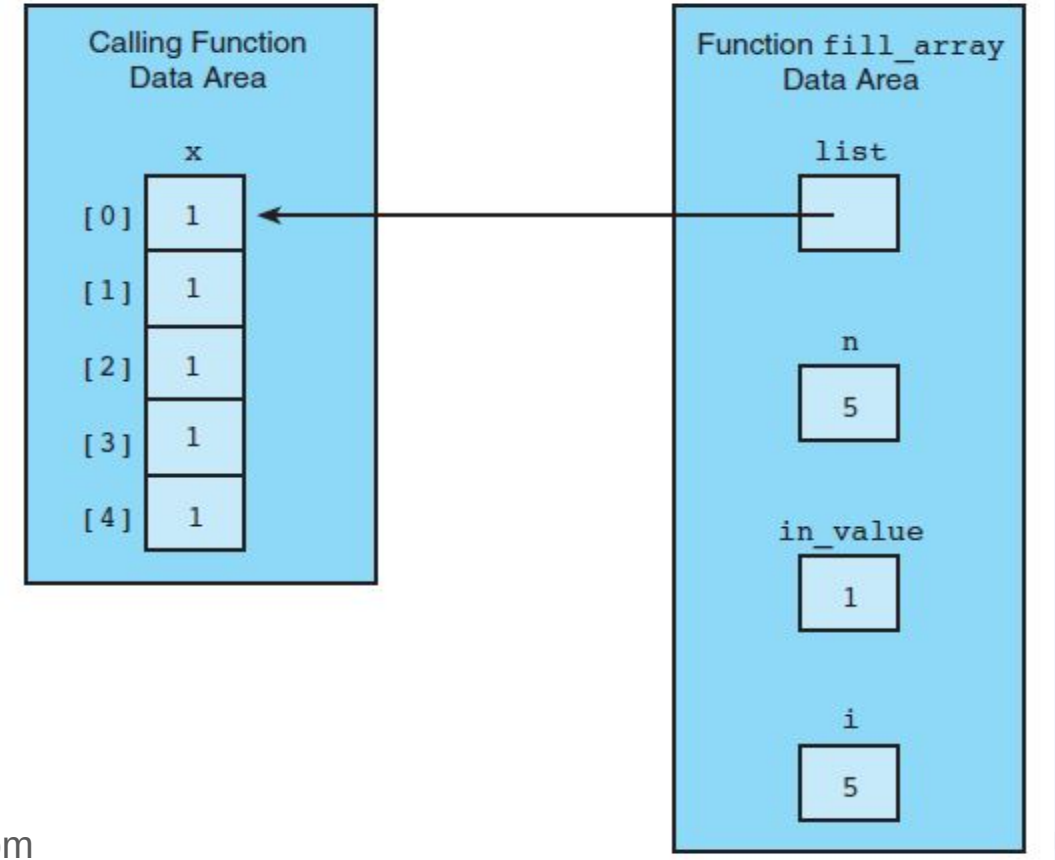


Figure 7.5

ARGUMENT CORRESPONDENCE FOR ARRAY PARAMETERS

- **Use of *list Instead of list[] in a Formal Parameter List:**

- In the declaration for function **fill_array** , we can use either parameter declaration:
(1)
- **int list[]**
- **int *list**

- Because C passes an array argument by passing the address of its initial element, the second declaration would be equally valid for an integer array parameter.

- Remember that a formal parameter of the form *type1 *param*
is compatible with an actual argument that is an array of ***type1*** values.

ARRAYS AS INPUT ARGUMENTS

- We can use a **qualifier** that we can include in the declaration of the array formal parameter in order to notify the *C compiler* that the array is only an input to the function and that the function does not intend to modify the array. (1)

- In a formal parameter list, the reserved word **const** indicates that the array variable declared is strictly an input parameter and will not be modified by the function. (2)

- **EXAMPLE 7.8, p.391,** [figure 7.6]:

  - Formal parameter list actually contains the address of the type **int** variable **x[0]** . (3)

RETURNING AN ARRAY RESULT

- In C, it is not legal for a function's return type to be an array; therefore, defining such a function requires use of an **output parameter** to send the result array back to the calling module.

- A function returning an array result depends on its caller to provide an array variable into which the result can be stored.

FIGURE 7.8 Function to Add Two Arrays

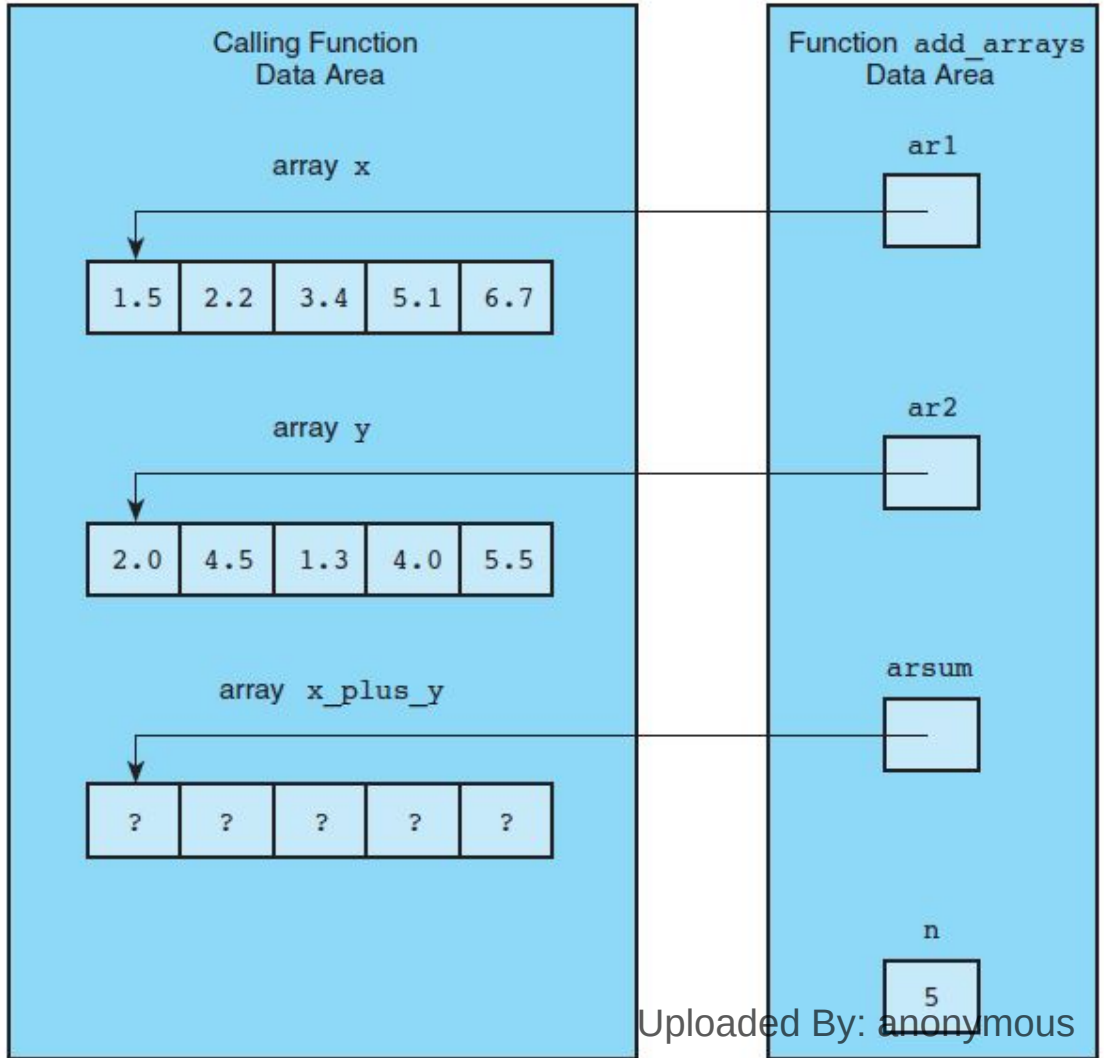
```
1.  /*
2.   * Adds corresponding elements of arrays ar1 and ar2, storing the result in
3.   * arsum. Processes first n elements only.
4.   * Pre: First n elements of ar1 and ar2 are defined. arsum's corresponding
5.   *      actual argument has a declared size >= n (n >= 0)
6.   */
7.  void
8.  add_arrays(const double ar1[],    /* input -                              */
9.             const double ar2[],    /* arrays being added                   */
10.            double       arsum[],  /* output - sum of corresponding
11.                                       elements of ar1 and ar2             */
12.            int          n)        /* input - number of element
13.                                       pairs summed                        */
14. {
15.       int i;
16.
17.       /* Adds corresponding elements of ar1 and ar2                       */
18.       for  (i = 0; i < n; ++i)
19.           arsum[i] = ar1[i] + ar2[i];
20. }
```

RETURNING AN ARRAY RESULT

- If we assume this calling function:

```
add_arrays(x, y, x_plus_y, 5);
```

=>



x_plus_y after call to add_arrays

3.5	6.7	4.7	9.1	12.2

Figure 7.9

RETURNING AN ARRAY RESULT

- Note that Address-of Operator is Not Used.

- The & (address-of) operator is not applied to the name of the output array argument

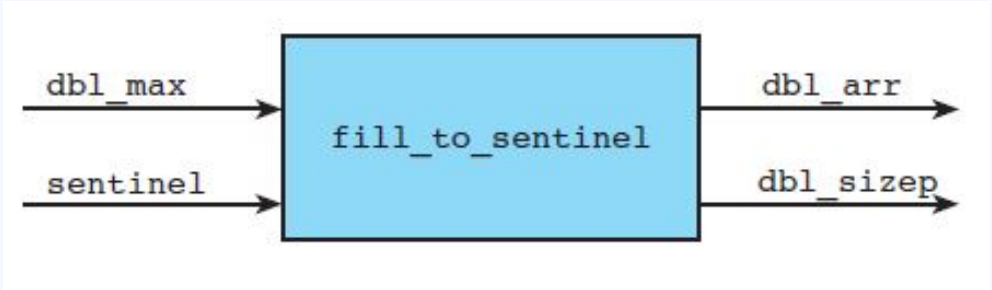
- Since the output parameter **arsum** is declared with no **const** qualifier, function **add_arrays** automatically has access and authority to change the corresponding actual array argument

PARTIALLY FILLED ARRAYS

- Frequently, a program will need to process many lists of similar data; these lists may not all be the same length.

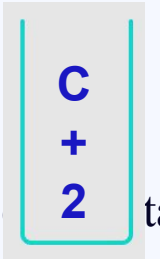
- In order to reuse an array for processing more than one data set, the programmer often declares an array large enough to hold the largest data set anticipated. (1).

- **EXAMPLE 7.10**



STACKS

- A **stack** is a data structure in which only the top element can be accessed. (1)
- A stack of three characters:

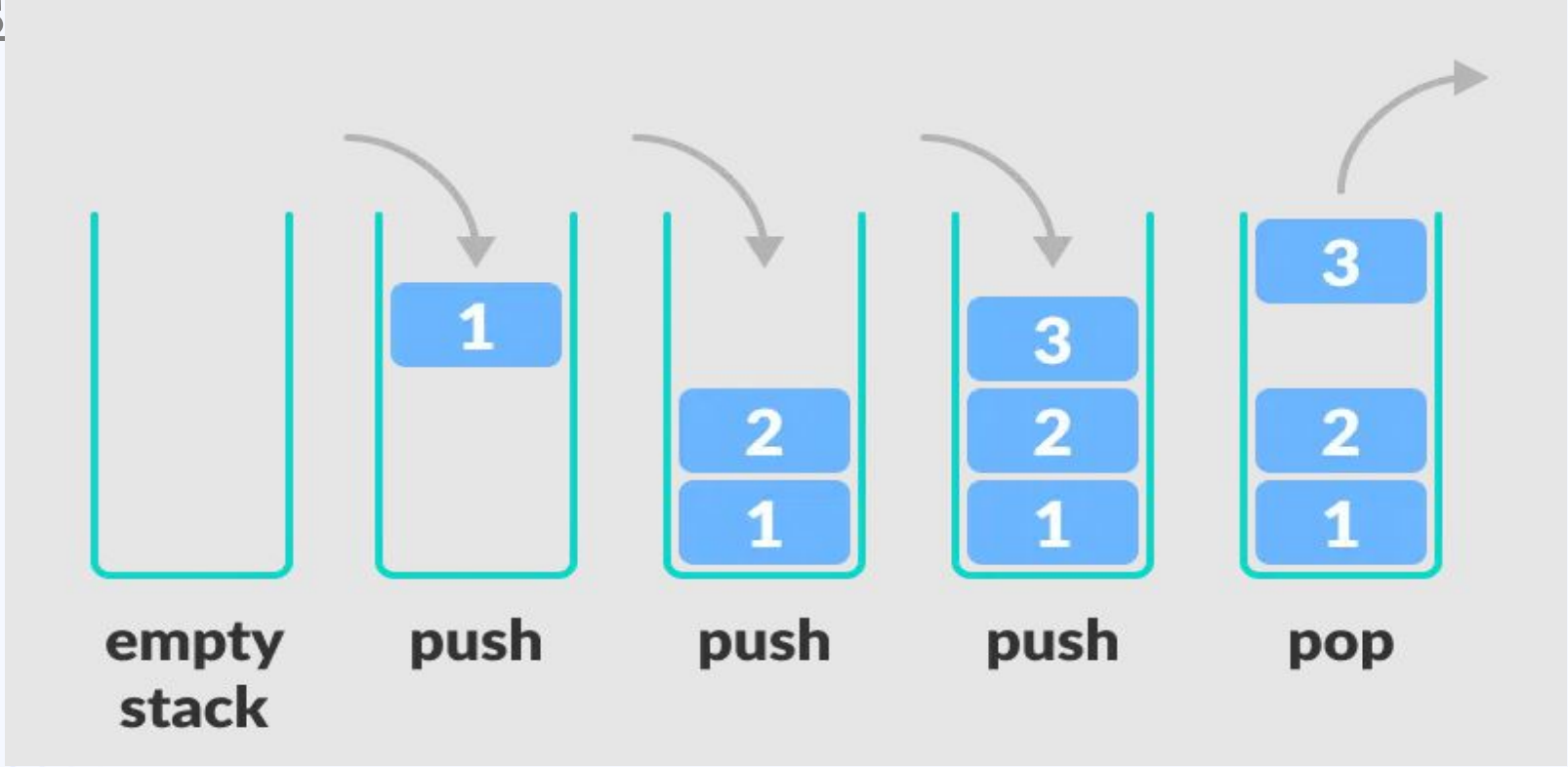


- The letter C, the character at the top of the stack, is the only one we can access. (2)

- **popping the stack**: Removing a value from a stack.
- **pushing it onto the stack**: storing an item in a stack

STACK

S



➢ **Figure 7.13**

7.6 SEARCHING AND SORTING AN ARRAY

- As an example of an array search, we might want to search an array of student exam scores to determine which student, if any, got a particular score.

- An example of an array sort would be rearranging the array elements so that they are in increasing order by score.

ARRAY SEARCH

- In order to search an array, we need to know the array element value we are seeking, or the **search target.**

- Then, we can perform the search by examining in turn each array element using a loop and by testing whether the element matches the target.

- The search loop should be exited when the target value is found; this process is called ***a linear search .***

ARRAY SEARCH

➢ The following algorithm for ***linear search*** sets a flag (for loop control) when the element being tested matches the target:

1. Assume the target has not been found.
2. Start with the **initial** array element.
3. repeat while the target is not found and there are more array elements
4. **if** the current element matches the target
5. Set a **flag** to indicate that the target has been found.

else

6. Advance to the **next array element**.
7. **if** the target was found
8. Return the **target index** as the search result.

else

9. Return **−1** as the search result.

ARRAY SEARCH

- **Figure 7.14**

➢ The type **int** variable ***found*** is used to represent the logical concept of whether the target has been found yet and is tested in the loop repetition condition (1)

➢ After found becomes true or the entire array has been searched, the loop is exited, and the decision statement following the loop defines the value returned.

➢ If array **arr** and **target** are declared in the calling function, the assignment statement: **index = search(arr, target, SIZE);**

calls function **search** to search the first **SIZE** elements of array **arr** for the **target** (**i.e. target = 3**):(2)

SORTING AN ARRAY

- Many programs execute more efficiently if the data they process are sorted before processing begins.

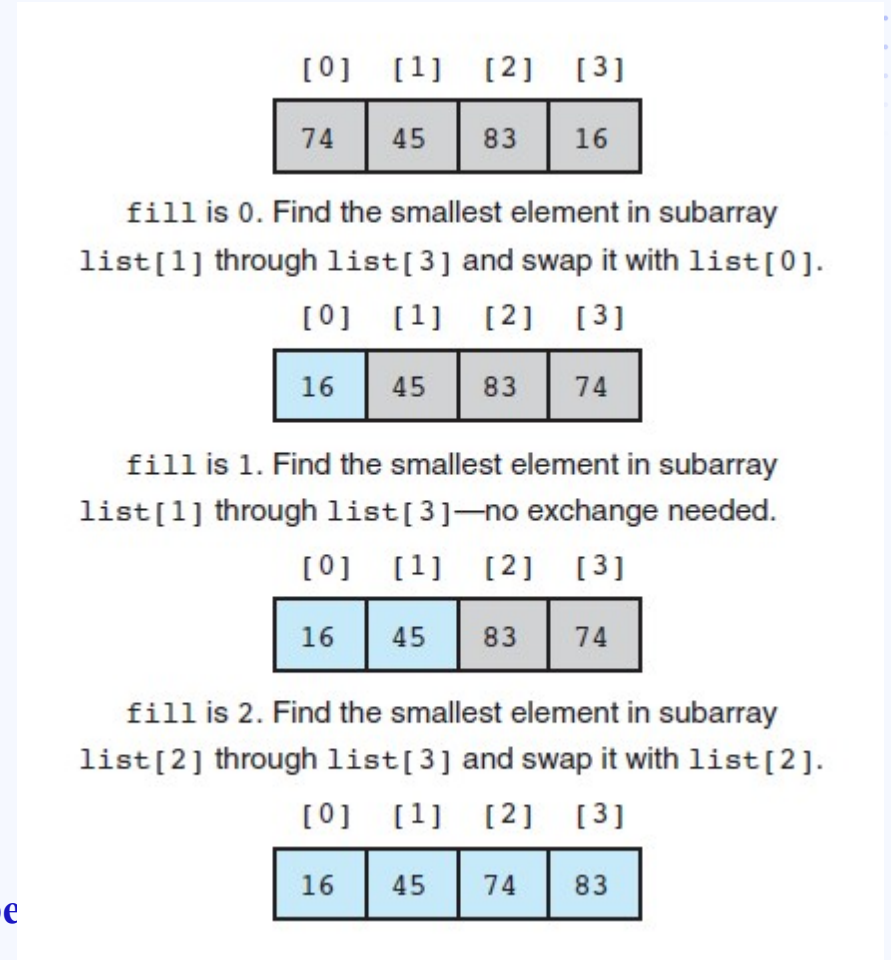
- For example, your university might want students grade report sorted by student ID number.

- **ALGORITHM FOR SELECTION SORT:**

1. for each value of **index** from **0 to n-2**
2. Find **index_of_min** , the index of the smallest element in the unsorted subarray **list[index]** through **list[n-1] .**
3. if **index** is not the position of the smallest element (**index_of_min**)
4. Exchange the smallest element with the one at position **index** .

SORTING AN ARRAY

- Figure 7.15



- See **Figure 7.16 p.405** for a program example

- **at most, n-1 exchanges will be**

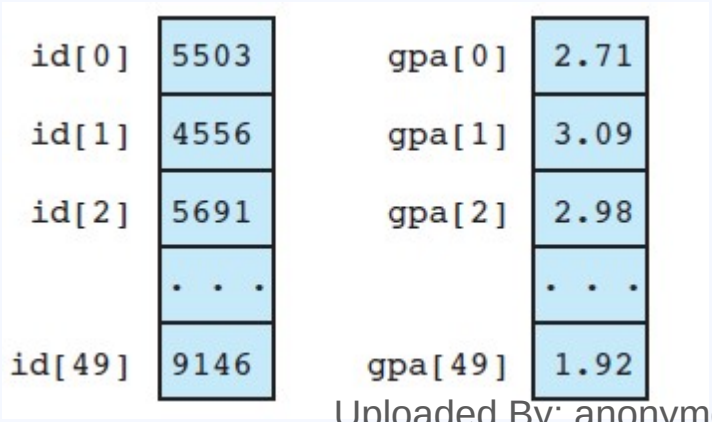
7.7 PARALLEL ARRAYS AND ENUMERATED TYPES

➢ **Parallel arrays:** arrays that have the same number of elements:

1. If there are **n -elements**, these parallel arrays contain data for **n -objects** of the same kind. [not type]
2. Further, all array elements at subscript **i** contain data for the **i-th** object in this group of **n -objects**.

- **EXAMPLE 7.11 , Figure 7.17** two parallel arrays for a student records program (1)

```
#define NUM_STUDENTS 50
..
int id[NUM_STUDENTS];
double gpa[NUM_STUDENTS];
```



ENUMERATED TYPES

- Good solutions to many programming problems require new data types (1)
- C allows you to associate a *numeric code* with each category by creating an enumerated type that has its own list of meaningful values.

(3)

```
typedef enum      (2)
{entertainment, rent, utilities, food, clothing,
automobile, insurance, miscellaneous}
expense_t;
```

Entertainment (0)
Rent (1)
Utilities (2)
…

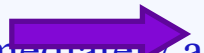
To declare:

```
expense_t expense_kind; (4)
```

ENUMERATED TYPES

➢ **Figure 7.18**

- The scope rules for identifiers apply to *enumerated types* and *enumeration constants*.
- Enumeration constants must be identifiers; they cannot be numeric, character, or string literals (e.g., "entertainment" cannot be a value for an enum
- We recommend that you place type definitions immediately afte
#include directives. (1)
- The reserved word **typedef** can be used to name many varieties of user-defined types.



ENUMERATED TYPES

```
typedef enum
{monday, tuesday, wednesday, thursday,friday, saturday, sunday}
day_t;
```

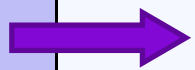
- An **identifier** cannot appear in more than one enumerated type example, the **weekday_t** definition could not be used with the ty

```
typedef enum
{monday, tuesday, wednesday, thursday, friday}
weekday_t;
```

ENUMERATED TYPES

- **Relational, assignment,** and even **arithmetic operators** can be used with enumerated types, just as with other integers.
- The following for type **day_t** are true:

```
sunday < monday
wednesday != friday
tuesday >= sunday
```

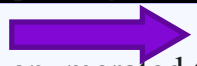


- We can combine the use of **arithmetic operators** and **casts** to find enumeration constants that follow and precede a current value.

ENUMERATED TYPES

➢ **EXAMPLE 7.12:** If today and tomorrow are type **day_t** variables, the following if statement assigns the value of tomorrow based on the value of today : (1)

```
if (today == sunday)
  tomorrow = monday;
else
  tomorrow = (day_t)(today + 1);
```



- C provides no range checking to verify that the value stored in an enumerated type

For example, this assignment statement will not cause a run-time error even though it i

```
today = saturday + 3;
```

ENUMERATED TYPES

- enumerated type variable can also be used as a loop counter:

```
for (today = monday; today <= friday; ++today) {
. . .
}
```

- This loop will execute for each value of **today** from **monday** th

ARRAY WITH ENUMERATED TYPE SUBSCRIPT

EXAMPLE 7.13

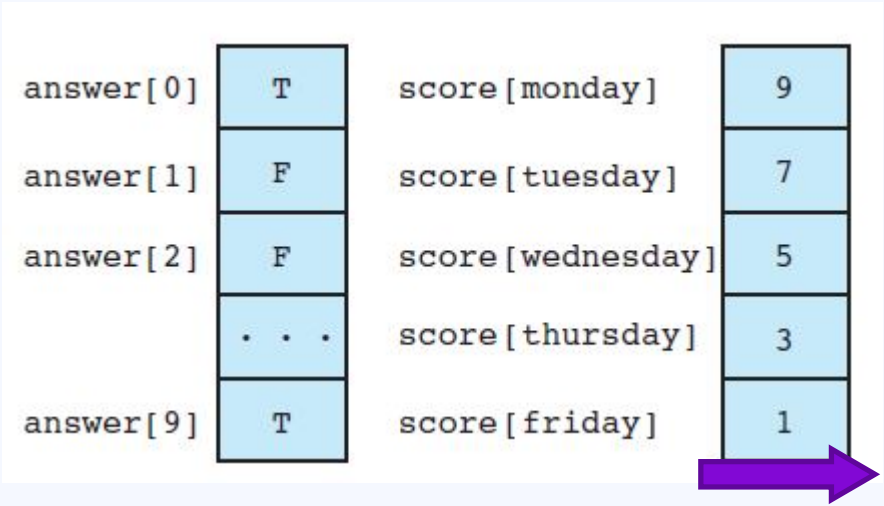
```
#define NUM_QUEST 10                    /* number of questions on daily quiz */
#define NUM_CLASS_DAYS 5                /* number of days in a week of class */
typedef enum
{monday, tuesday, wednesday, thursday, friday}
week_day_t;
. . .
char answer[NUM_QUEST];                 /* correct answers for one quiz */
int score[NUM_CLASS_DAYS];              /* one student's quiz scores for each day */

score[monday] = 9;
score[tuesday] = 7;
score[wednesday] = 5;
score[thursday] = 3;
score[friday] = 1;
```

Countinue…

ARRAY WITH ENUMERATED TYPE SUBSCRIPT

EXAMPLE 7.13




```
ascore = 9;
for (today = monday; today <= friday; ++today) {
score[today] = ascore;
ascore -= 2;
}
```

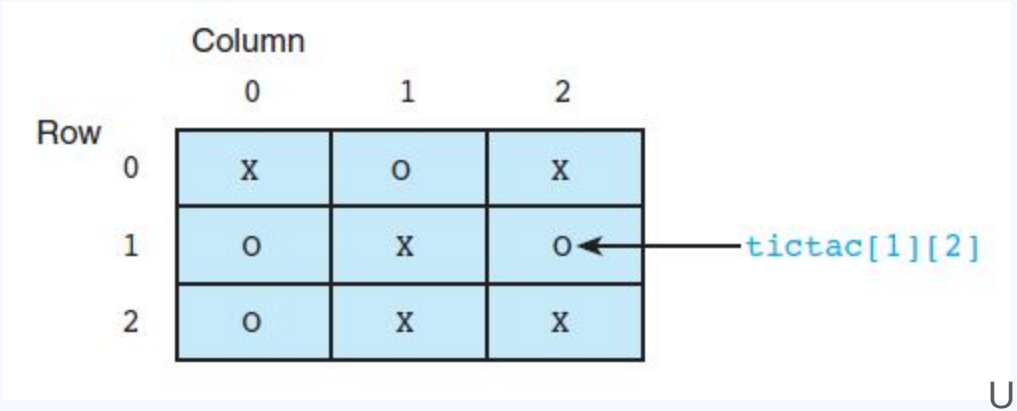
7.8 MULTIDIMENSIONAL ARRAYS

- **Multidimensional arrays:** arrays with two or more dimensions.

- The array declaration: **char tictac[3][3];**
  allocates storage for a two-dimensional array (**tictac**) with three rows and three columns.

- Each array element contains a character value

- This array has nine elements, each of which must be referenced by specifying a **row subscript** (0 , 1 , or 2) and a **column subscript** (0 , 1 , or 2).



7.8 MULTIDIMENSIONAL ARRAYS

- A function that takes a tic-tac-toe board as a parameter will have a declaration similar to this in its prototype:

```
char tictac[][3]
```

- In the declaration of a multidimensional array parameter, only the first dimension, the number of rows, can be omitted.

- Including both dimensions is also permissible. (1)

```
char tictac[3][3]
```

- The following statement would declare a tic-tac-toe board and initialize its contents to blanks. (2)

```
char tictac[3][3] = { {' ', ' ', ' '}, {' ', ' ', ' '}, {' ', ' ', ' '} };
```

7.8 MULTIDIMENSIONAL ARRAYS

- **EXAMPLE 7.14**

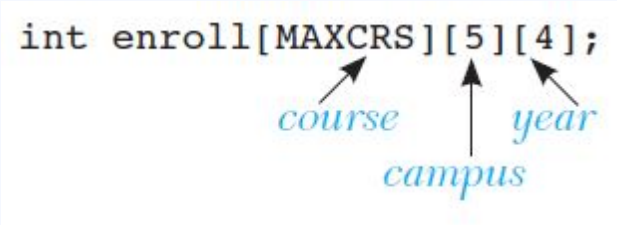
- This array table consists of three dimensions:
  double table[7][5][6];

- The first subscript may take on values from 0 to 6 ; the second, from 0 to 4 ; and the third, from 0 to 5 .

- A total of 7 × 5 × 6 , or 210 , type double values may be stored in the array table.

- All three subscripts must be specified in each reference to array table in order to access a single number (e.g., **table[2][3][4]**).

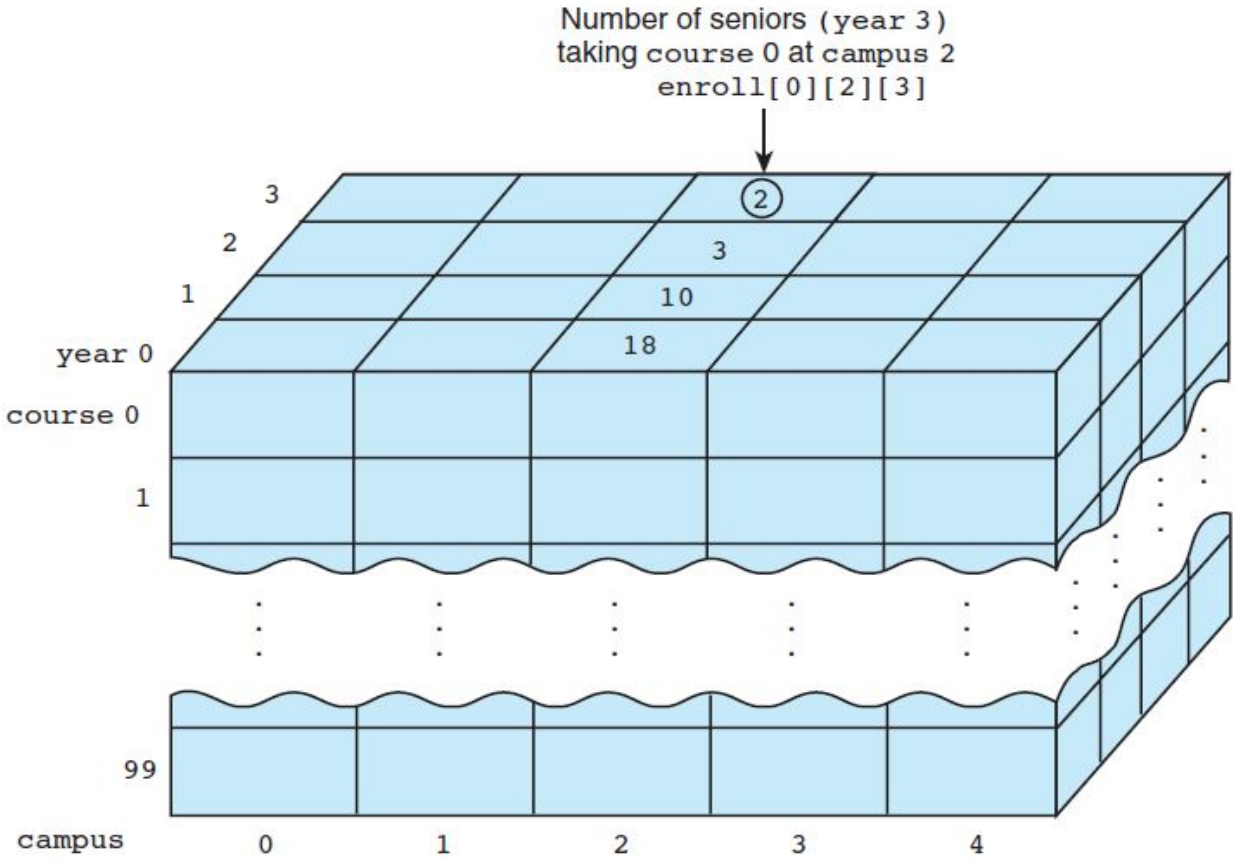
ARRAYS WITH SEVERAL DIMENSIONS:



- We will assume that the college offers 100 (**MAXCRS**) courses at five different campuses.
- Array enroll is composed of a total of 2000 (100 × 5 × 4) elements.
- we will number the freshman year 0 , the sophomore year 1, and so on.
- Thus, **enroll[1][4][3]** represents the number of seniors taking course 1 at campus 4 .
- Memory space can be used up rapidly if several multidimensional arrays are declared in the same program. (2)
- The type of information desired determines the order in which we must reference the array elements. (3)

ARRAYS WITH SEVERAL DIMENSIONS:

➢ Figure 7.22



ARRAYS WITH SEVERAL DIMENSIONS:

➢ **Example 7.16**

```
/* Finds and displays number of students in each course */
for (course = 0; course < MAXCRS; ++course) {
   crs_sum = 0;
   for (campus = 0; campus < 5; ++campus) {
      for (year = 0; year < 4; ++ year) {
         crs_sum += enroll[course][campus][year];
      }
   }
   printf("Number of students in course %d is %d\n", course, crs_sum);
}
```

ARRAYS WITH SEVERAL DIMENSIONS:

➢ Example 7.16

```
/* Finds and displays number of students at each campus */
for (campus = 0; campus < 5; ++campus) {
   campus_sum = 0;
   for (course = 0; course < MAXCRS; ++course) {
      for (year = 0; year < 4; ++ year) {
         campus_sum += enroll[course][campus][year];
      }
   }
   printf("Number of students at campus %d is %d\n", campus, campus_sum);
}
```

CASE STUDY (Homework)
P.419 - 427

Summary of Hospital Revenue

7.11 COMMON PROGRAMMING ERRORS

- **Subscript-range error:** An out-of-range reference occurs when the subscript value used is outside the range specified by the array declaration. (1)

```
int celsius[100];
```

- A **subscript-range error** occurs when **celsius** is used with a subscript that has a value less than 0 or greater than 99,

- i.e. If the value of **i** is 150 **=> error** **access violation at line no. 28** **(2)**

- They are most often caused by:

1. **An incorrect subscript expression**,
2. **A loop counter error**
3. **A nonterminating loop**.

7.11 COMMON PROGRAMMING ERRORS

- **Subscript-range error:**
- If a **subscript-range error** occurs inside an indexed loop, verify that the subscript is in range for both the initial and the fin able (1)
- If a **subscript-range error** occurs in a loop controlled by a variable other than the array subscript, check that the loop control variable is being updated as required. (2)

7.11 COMMON PROGRAMMING ERRORS

- When using arrays as arguments to functions, be careful not to apply the **address-of** operator to the array name even if the array is an output argument.
- However, do remember to us is being passed as an output argument.
- Be sure to use the correct forms for declaring array input and output parameters in function prototypes.
- A parameter declared a could represent a single integer output parameter or an integer array parameter **int *z**
- So, comment your own prototypes carefully and use the alternate declaration form for array parameters to assist readers of your code.

int z[]

7.11 COMMON PROGRAMMING ERRORS

- **Memory access violation:** If you are working on a computer system with very limited memory, you may find that some correct C programs generate run-time error messages indicating an access violation. (1)

- When you define **enumerated types**, remember that only identifiers can appear in the list of values (**enumeration constants**) for the type.

- Be careful not to reuse one of these identifiers in another type or as a variable name in a function that needs your type definition. (2)

Refernces

Problem Solving and Program Design in C, 7th Ed., by Jeri R. Hanly and Elliot B. Koffman