جامعة بيرزيت
BIRZEIT UNIVERSITY

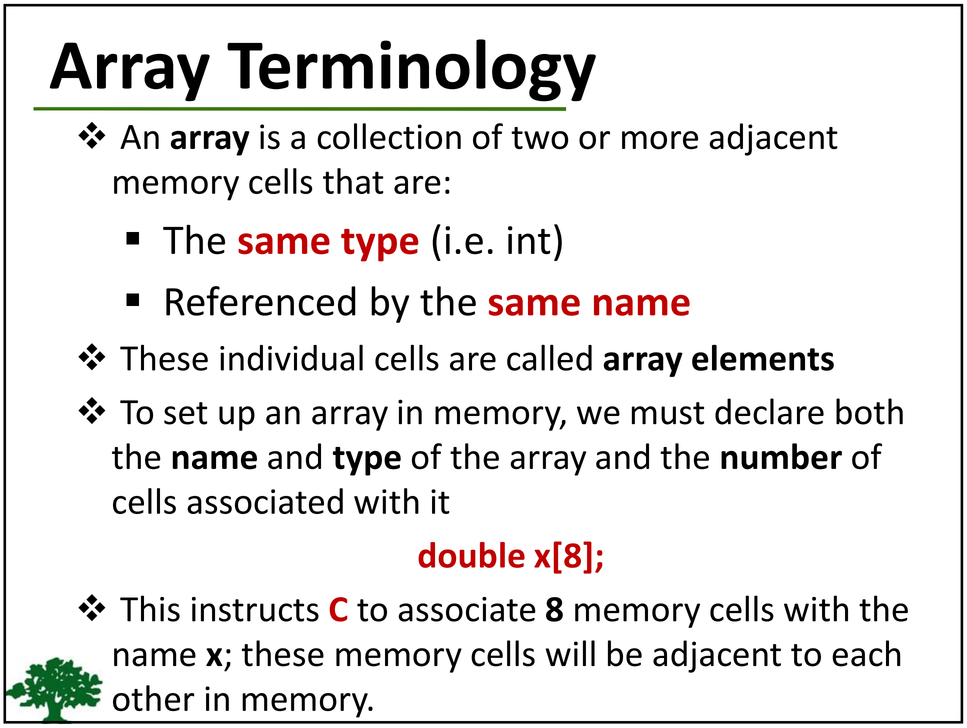
Arrays

PROBLEM SOLVING AND PROGRAM DESIGN In C
7th EDITION
Jeri R. Hanly, Elliot B. Koffman

By: Mamoun Nawahdah (PhD)
2013/2014

What is an Array?

- **Scalar data** types use a **single memory unit** to store a **single value**.
- For many problems you need to group data items together.
- A program that processes exam scores for a class, for example, would be easier to write if all the scores were stored in one area of memory and were able to be accessed as a group.
- **C** allows a programmer to group such related data items together into a single composite data structure.
- In this chapter, we look at one such data structure: the **Array**.

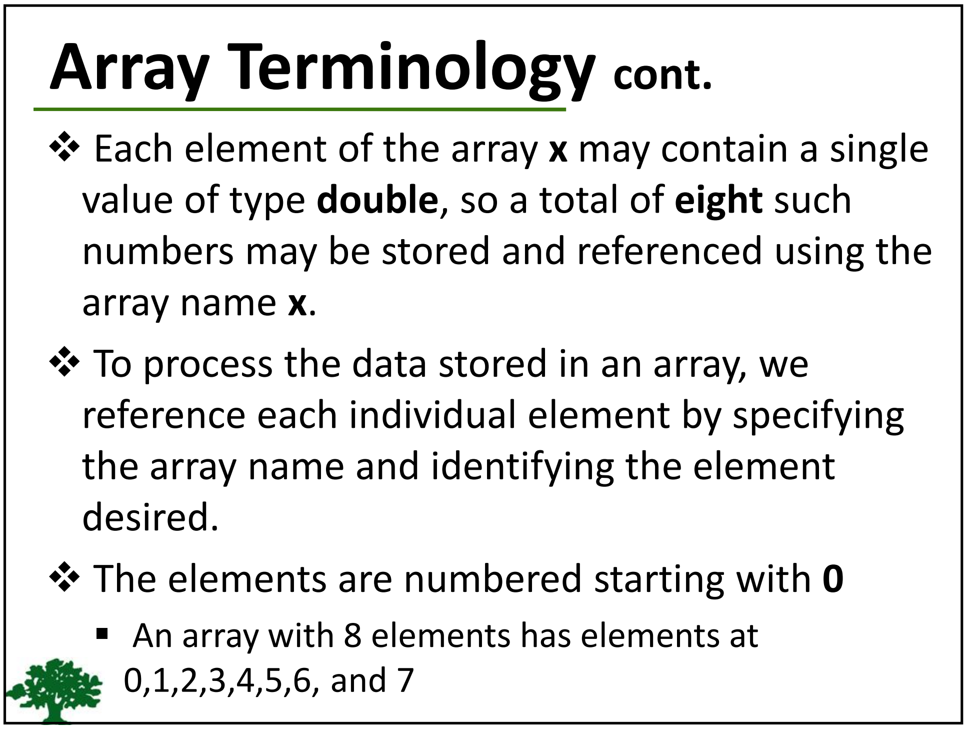


Array Terminology

- An **array** is a collection of two or more adjacent memory cells that are:
  - The **same type** (i.e. int)
  - Referenced by the **same name**
- These individual cells are called **array elements**
- To set up an array in memory, we must declare both the **name** and **type** of the array and the **number** of cells associated with it

```
double x[8];
```

- This instructs **C** to associate **8** memory cells with the name **x**; these memory cells will be adjacent to each other in memory.



Array Terminology cont.

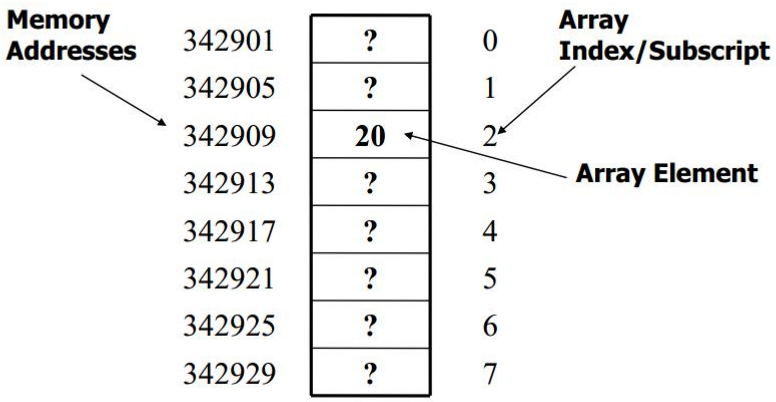
- Each element of the array **x** may contain a single value of type **double**, so a total of **eight** such numbers may be stored and referenced using the array name **x**.
- To process the data stored in an array, we reference each individual element by specifying the array name and identifying the element desired.
- The elements are numbered starting with **0**
  - An array with 8 elements has elements at 0,1,2,3,4,5,6, and 7

Array Terminology cont.

- ❖ The subscripted variable **x[0]** (read as x **sub** zero) refers to the initial or 0th element of the array x, **x[1]** is the next element in the array, and so on.
- ❖ The **integer** enclosed in brackets is the array subscript or **index** and its value must be in the range from **zero** to one less than the array size.

Visual Representation of an Array

```
int x[8];
x[2] = 20;
```



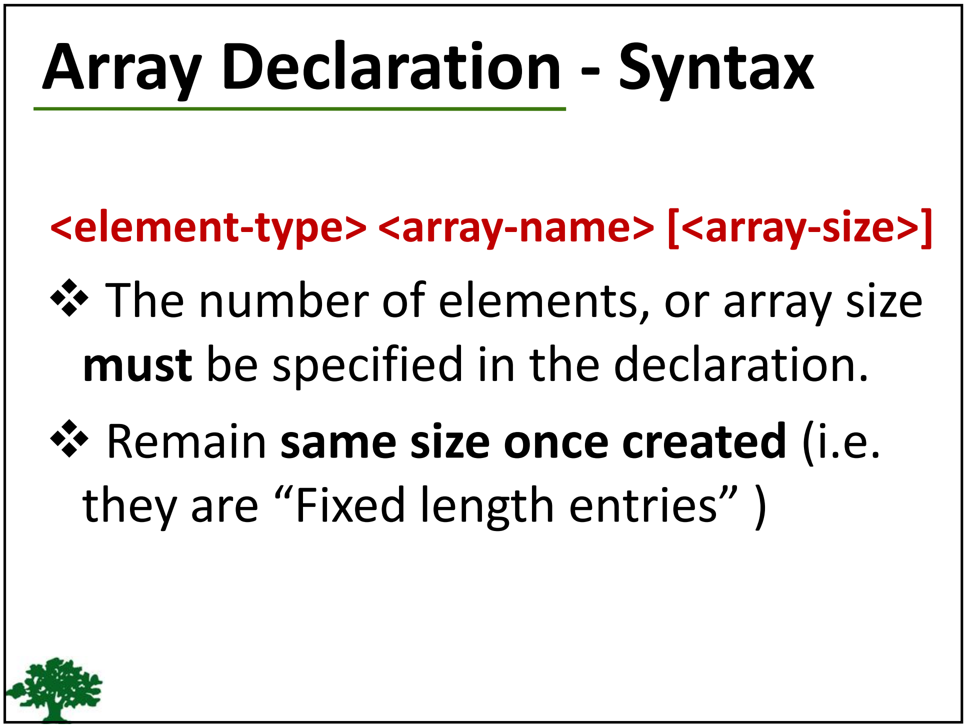
Memory Addresses

Array Index/Subscript

Array Element

Memory Addresses	Array Element	Array Index/Subscript
342901	?	0
342905	?	1
342909	20	2
342913	?	3
342917	?	4
342921	?	5
342925	?	6
342929	?	7

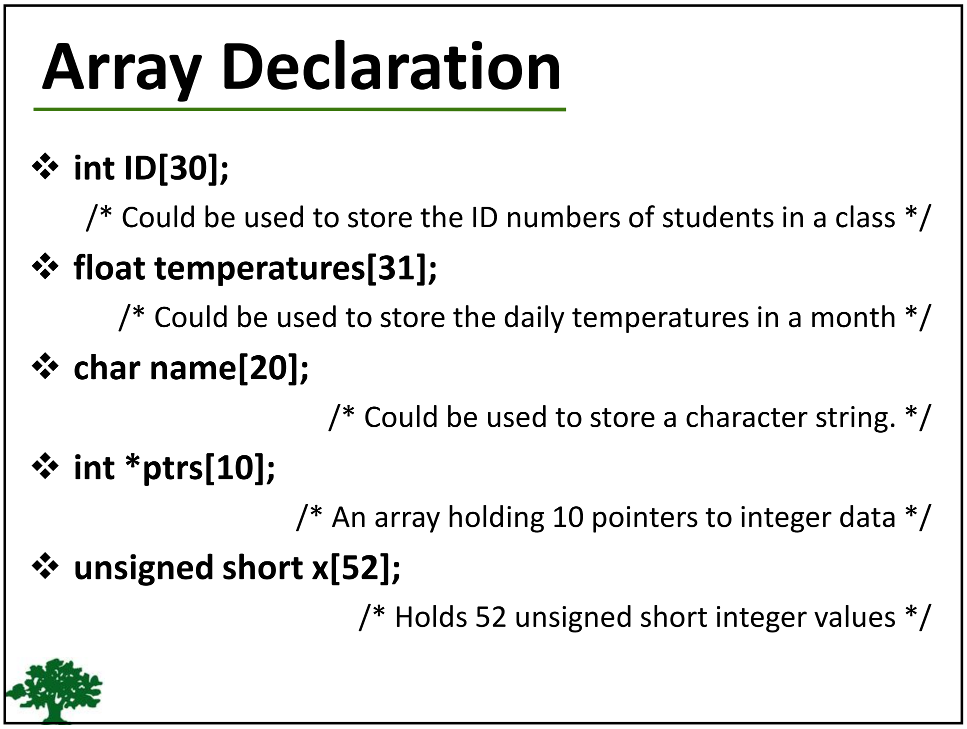
Note: Index starts with 0, not with 1



Array Declaration - Syntax

<element-type> <array-name> [<array-size>]

- The number of elements, or array size **must** be specified in the declaration.
- Remain **same size once created** (i.e. they are “Fixed length entries”)



Array Declaration

- **int ID[30];**

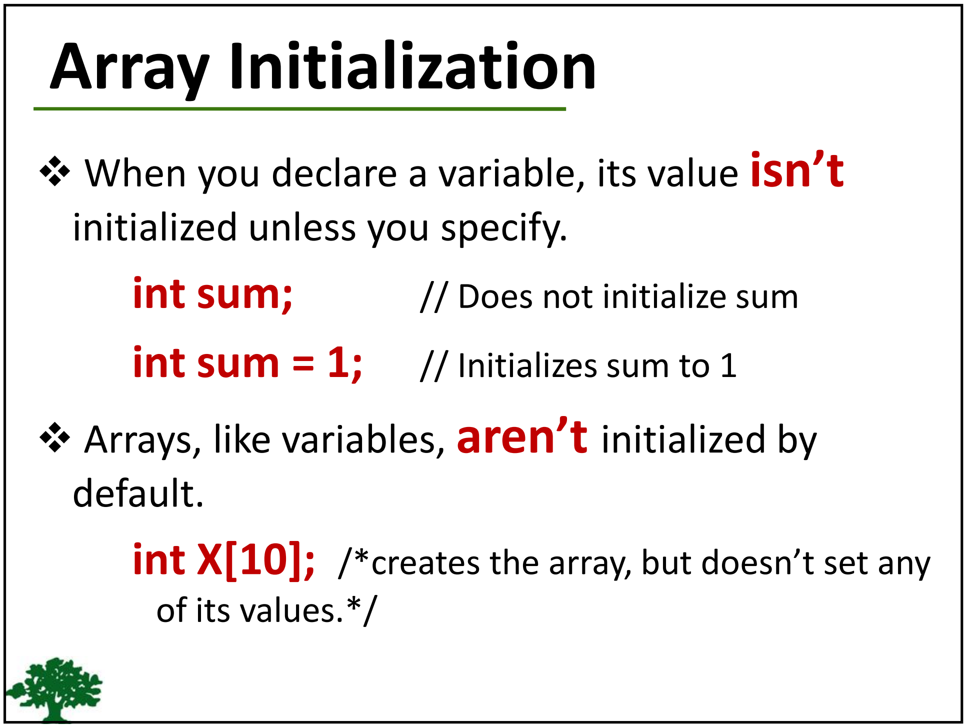
  /* Could be used to store the ID numbers of students in a class */
- **float temperatures[31];**

  /* Could be used to store the daily temperatures in a month */
- **char name[20];**

  /* Could be used to store a character string. */
- **int *ptrs[10];**

  /* An array holding 10 pointers to integer data */
- **unsigned short x[52];**

  /* Holds 52 unsigned short integer values */



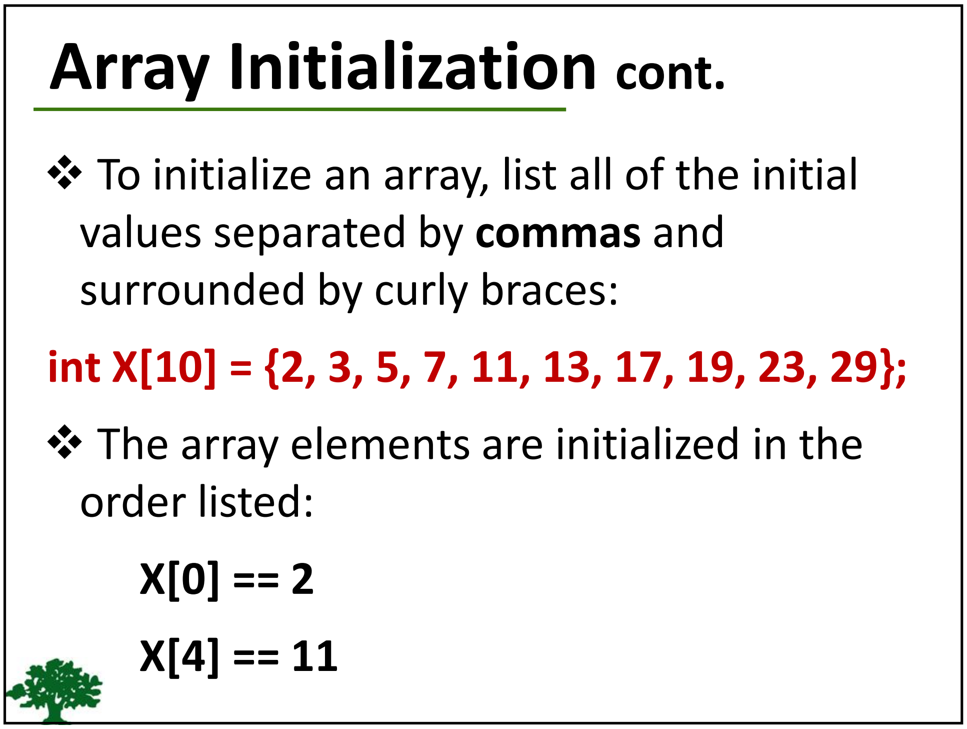
Array Initialization

- When you declare a variable, its value **isn't** initialized unless you specify.

```
int sum;          // Does not initialize sum
int sum = 1;      // Initializes sum to 1
```

- Arrays, like variables, **aren't** initialized by default.

```
int X[10];  /*creates the array, but doesn't set any
   of its values.*/
```



Array Initialization cont.

- To initialize an array, list all of the initial values separated by **commas** and surrounded by curly braces:

```
int X[10] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
```

- The array elements are initialized in the order listed:

```
X[0] == 2
X[4] == 11
```

Array Initialization cont.

❖ If there are values in the initialization block, but not enough to fill the array, all the elements in the array without values are initialized to **0** in the case of **double** or **int**, and **NULL** in the case of **char**.

```
int scores[20] = {0};  /* all 20 elements are initialized to 0 */

int scores[20] = {1, 2, 3};  /* First 3 elements are initialized to
    1, 2, 3 and the rest are   initialized to 0 */
```

Array Initialization cont.

❖ If there are values in the initialization block, an explicit size for the array does not need to be specified. Only an empty array element is sufficient, C will count the size of the array for you.

```
int scores[] = {20, 10, 25, 30, 40};  /* size of the
    array score is automatically calculated as 5 */
```

Good Practice

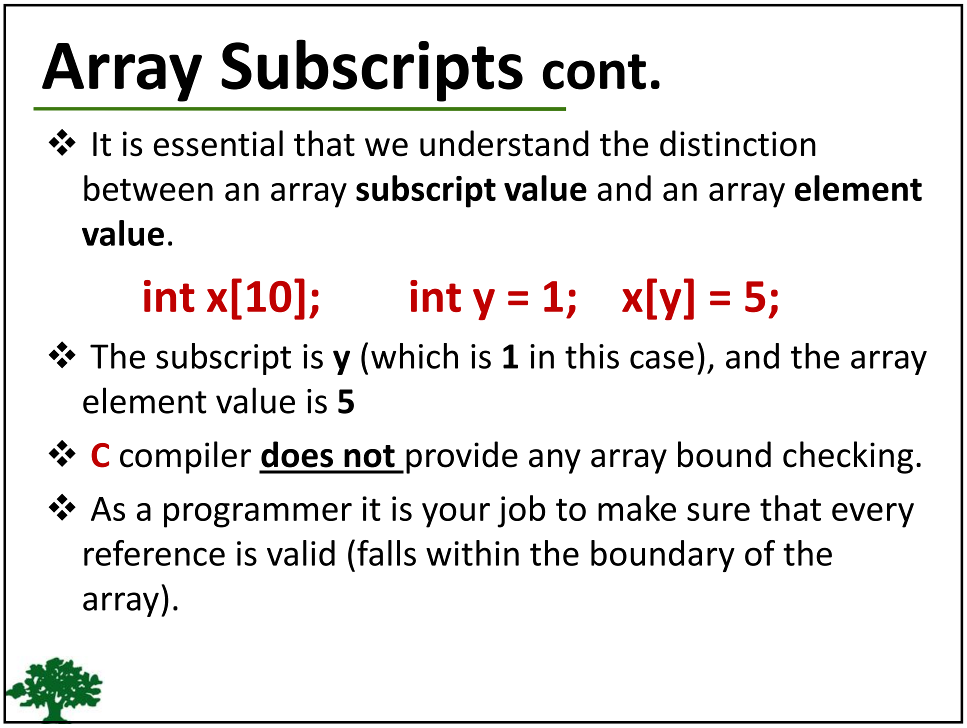
```
const int AarraySize  = 12;
int myArray[ArraySize];
```

OR

```
#define ARRAY_SIZE 12
int myArray[ARRAY_SIZE];
```

Array Subscripts

- We use subscripts/indices to differentiate between the individual array elements.
- We can use any expression of type int as an array subscript.
- However, to create a valid reference, the value of this subscript **must** lie between **0** and one less than array size.

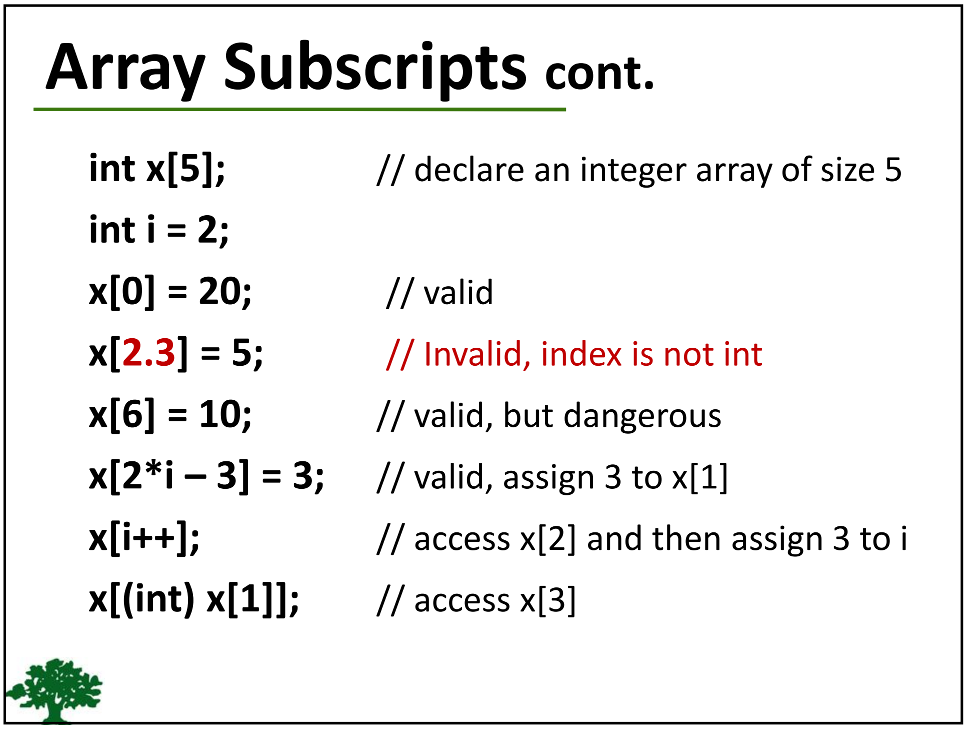


Array Subscripts cont.

- It is essential that we understand the distinction between an array **subscript value** and an array **element value**.

```
int x[10];     int y = 1;   x[y] = 5;
```

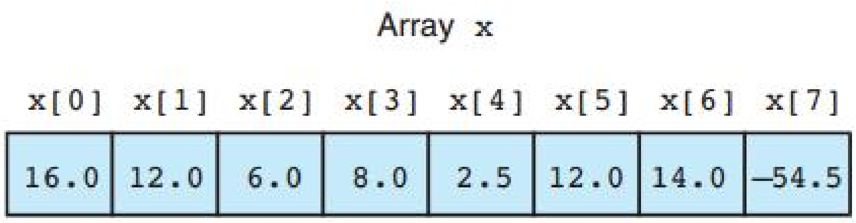
- The subscript is **y** (which is **1** in this case), and the array element value is **5**
- C compiler **does not** provide any array bound checking.
- As a programmer it is your job to make sure that every reference is valid (falls within the boundary of the array).



Array Subscripts cont.

```
int x[5];           // declare an integer array of size 5
int i = 2;
x[0] = 20;           // valid
x[2.3] = 5;          // Invalid, index is not int
x[6] = 10;          // valid, but dangerous
x[2*i – 3] = 3;     // valid, assign 3 to x[1]
x[i++];             // access x[2] and then assign 3 to i
x[(int) x[1]];      // access x[3]
```

Array Manipulation



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

- ❖ Each array element can be manipulated like any simple variable. The array element values can be displayed (use the given values for array **x**):

  printf("%.1f", x[0]); /* output: 16.0 */

Array Manipulation cont.

- ❖ or can be assigned a value,

  x[1] = 125.6; /* stores 125.6 to second cell overwriting any existing value */

- ❖ or can be used with **scanf** ,

  scanf("%lf",&x[2]); /* allows keyboard entry for the third cell's value */

- ❖ or can be used in any arithmetic operation if possible,

  x[2] = x[4] + 5.0;

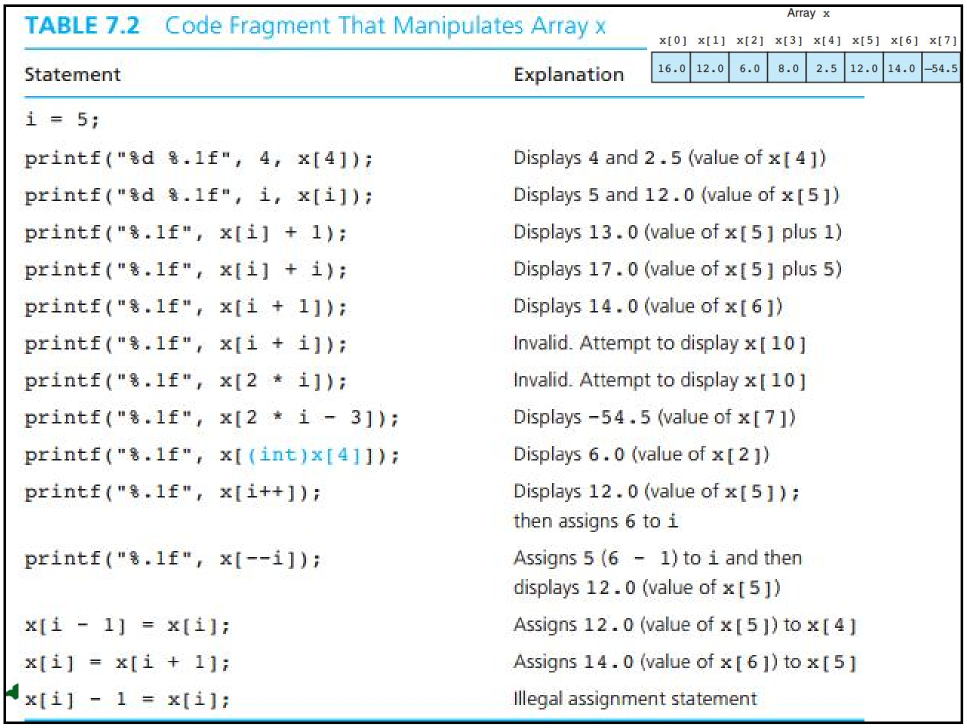


TABLE 7.2 Code Fragment That Manipulates Array x

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

Statement	Explanation
`i = 5;`	
`printf("%d %.1f", 4, x[4]);`	Displays 4 and 2.5 (value of x[4])
`printf("%d %.1f", i, x[i]);`	Displays 5 and 12.0 (value of x[5])
`printf("%.1f", x[i] + 1);`	Displays 13.0 (value of x[5] plus 1)
`printf("%.1f", x[i] + i);`	Displays 17.0 (value of x[5] plus 5)
`printf("%.1f", x[i + 1]);`	Displays 14.0 (value of x[6])
`printf("%.1f", x[i + i]);`	Invalid. Attempt to display x[10]
`printf("%.1f", x[2 * i]);`	Invalid. Attempt to display x[10]
`printf("%.1f", x[2 * i - 3]);`	Displays -54.5 (value of x[7])
`printf("%.1f", x[(int)x[4]]);`	Displays 6.0 (value of x[2])
`printf("%.1f", x[i++]);`	Displays 12.0 (value of x[5]); then assigns 6 to i
`printf("%.1f", x[--i]);`	Assigns 5 (6 - 1) to i and then displays 12.0 (value of x[5])
`x[i - 1] = x[i];`	Assigns 12.0 (value of x[5]) to x[4]
`x[i] = x[i + 1];`	Assigns 14.0 (value of x[6]) to x[5]
`x[i] - 1 = x[i];`	Illegal assignment statement

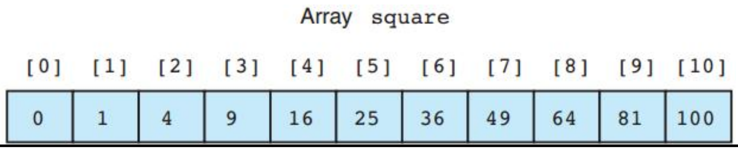
Using Loops for Sequential Access

- Very often, we wish to process the elements of an array in sequence, starting with element **0**.
  - Example: scanning data into the array or printing its contents.
- In **C**, we can accomplish this by using indexed **for** loop whose control variable runs from **0** to one less than the array size.

Using Loops for Sequential Access

❖ The following array **square** will be used to store the squares of the integers from **0** to **10**.

```
const  int  SIZE = 11;
int   square[SIZE],  i;
//The for loop
for (i=0 ;   i < SIZE ;   i++) {
          square[i] = i * i;
}
```



Array square

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
0	1	4	9	16	25	36	49	64	81	100

What's the output??

```
#include <stdio.h>
int main(){
     int square[100],  i, k;
     // Calculate the squares
     for (i = 0;  i < 100;  i++) {        // i runs from 0 to 99
          k = i + 1;                       //k runs from 1 to 100
          square[i] = k * k;
          printf("The square of %d is %d\n", k, square[i] );
     }
}
```

Access

- ❖ Want to process all of the elements of an array?
- ❖ Example: Adding the values of all array elements. Two alternative style for loops:

```
for ( i = 0; i < arraySize; i++)
  sum += a[i];
```

```
for ( i = 0; i <= arraySize-1; i++)
  sum += a[i];
```

Arrays and Pointers in C

- ❖ Arrays can also be accessed with pointers in **C**.
- ❖ Pointers do not have to point to single/scalar variables. They can also point at individual array elements:

```
    int * ptr;                int arr[10];
                 ptr = &arr[2];
```

- ❖ Pointers can be manipulated by “**+**” and “**-**”.
- ❖ The pointer **ptr-1** points to **arr[1]** and **ptr+2** points to **arr[4]**. Pointer **ptr++** points at **arr[3]**.

Arrays and Pointers in C cont.

- The name of the array is the address of the **1st** element of the array.
- In other words, a name of the array is actually a pointer to the element of the array that has index equal to **0**:

```
int * ptr;          int arr[10];

        ptr = arr; // same as ptr = &arr[0]
```

- Note: the name of the array ("**arr**") is a constant. We can't force this pointer to point at something else.
  - **arr+1** must be the same as **&arr[1]**
  - **arr+2** must be the same as **&arr[2]**

Arrays and Pointers in C cont.

```
int x[8], *aptr;
aptr = x;

printf("%d\n", x[5]);
printf("%d\n", *(x+5));
printf("%d\n", aptr[5]);
printf("%d\n", *(aptr+5));
```

10	0
20	1
30	2
40	3
50	4
60	5
70	6
80	7

The output is 60 in every case

Quick Review

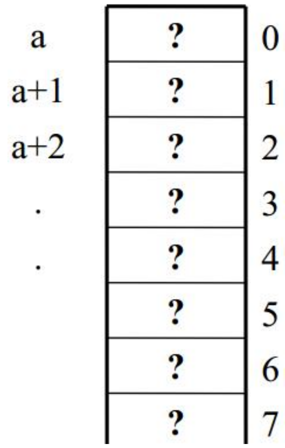
int * ptr1, * ptr2;

int a[10];

ptr1 = &a[2];

ptr2 = a; // equivalent to **ptr2 = &a[0];**

- ❖ An array variable is actually a pointer to the 1st element of the array.
- ❖ **ptr2** points to the 1st element of the array and get others by offset.
- ❖ Referring **a[i]** is same as referring ***(a+i)**.



a	?	0
a+1	?	1
a+2	?	2
.	?	3
.	?	4
	?	5
	?	6
	?	7

Searching an Array

```
1. Assume target has not been found (flag = false).
2. Start with the initial array element (index = 0).
3. If the target is not found and there are more:
    4. if the current element matches array element:
        5. set flag true.
        6. remember array index.
    7. else:
        8. advance to next array element (index++) , go to 3.
9. If flag equal true :
    10. return the array index.
11. else:
    12. return -1 to indicate not found.
```

Searching an Array

```
int  found = 0, i = 0, index = -1,  arr[10];
while ( !found   &&   (i <10)   ) {
     if ( arr[i] == target ) {
        found = 1;
        index = i;
     }
     i++;
}
if ( found )             return index;
else                     return -1;
```

Array Elements as Function Arguments

- If we want to print the **i**th element of the array **x[i]**, then we can do the following:
  - The call **printf(”%d\n”, x[i]);** uses array element **x[i]** as input argument to **printf** .
  - The call **scanf(”%d”, &x[i]);** uses the array element **x[i]** as output argument of **scanf** .
    - If **i** is **4**, the address of array element **x[4]** is passed to **scanf**, and **scanf** stores the value scanned in element **x[4]**.

Having Arrays as Function Arguments

- ❖ Besides passing individual array elements to functions, we can write functions that take **entire** arrays as arguments.
- ❖ There are several ways of passing arrays to functions but in each case we only **pass the address** of the array.
- ❖ This is very similar to what we did during "**passing variables by reference**"

Having Arrays as Function Arguments

- ❖ As we are not passing a copy of the array, any changes to the array made within the function will also effect the original array.
- ❖ 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 parameter is the address of the array.
- ❖ Example:

```
int a[10];
foo(a);
foo(&a[0]); // same as above
```

Using Arrays in Formal Parameter List

void function1(int x[10]); // **sized** array

- Store the address of the corresponding array argument to variable **x** and remember it as an array of 10 items.

void function1(int x[]); // **unsized** array

- The length of the array is not specified. Since it is not a copy, the compiler does not need to allocate space for the array and therefore does not need to know the size of the array.
- With this, we can pass an array of any size to function.

void function1(int *x); // array pointer

- This function can take any integer array as argument.

Example

```
void function2 (double,  double *,  double *);
main () {
    double x[8];
    double p, q, r;
    function2(p,  &q,  &r);
    function2( x[0],  &x[1],  &x[2]);
}
void function2 (double arg1,  double *arg2,  double *arg3){
    //statements;
}
```

- ❖ The statement (function call) passes the value of **p** to **function2** and returns the function results to variable **q** and **r**.
- ❖ **x[0]** is the input argument and **x[1]** and **x[2]** are output argument.
- ❖ Use ***arg2** and ***arg3** to return values to the calling function.
- ❖ Function parameters **arg2** and **agr3** contains the addresses of array elements **x[1]** and **x[2]**

C code Example

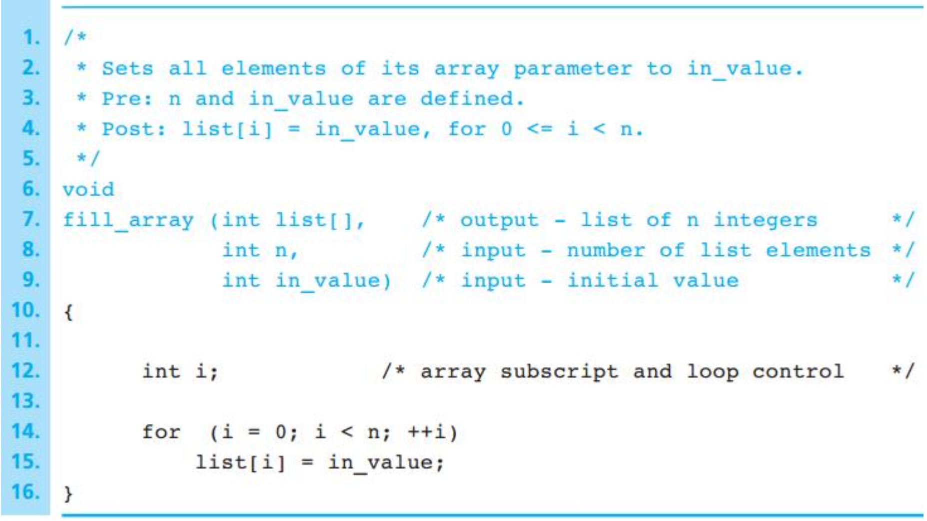
```
int main(void){
   int a[5]={1,2,3,4,5};
   int i;

   clear1(a,5);
   clear2(a,5);
   for(i=0; i<5; i=i+1)
     printf("%d ", a[i]);
   return 0;
}
```

```
void clear1(int x[], int size){
    int i;
    for(i=0; i<size; i=i+1)
      x[i] = 0;
}

void clear2(int *x, int size){
    int *p;
    for(p=x; p<(x+size); p=p+1)
    // for(p=&x[0]; p<&x[size]; p=p+1)
      *p = 0;
}
```

Arrays Arguments Example

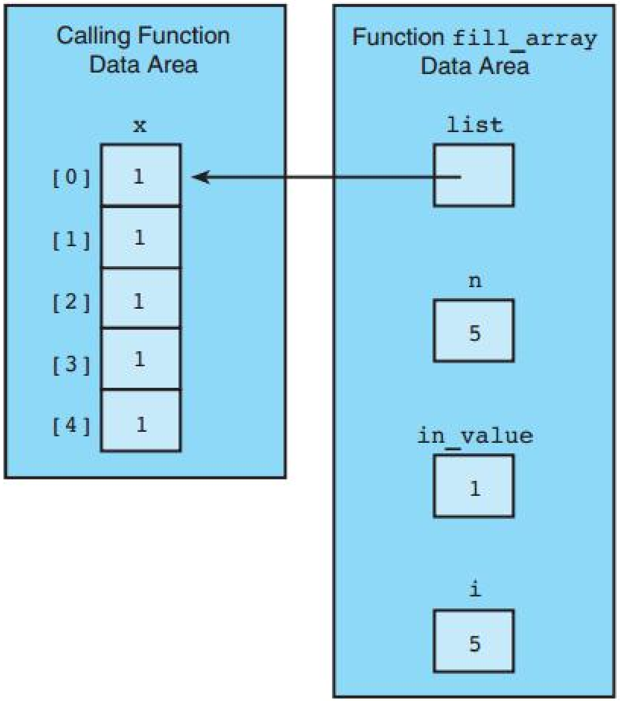


```
/*
 * Sets all elements of its array parameter to in_value.
 * Pre: n and in_value are defined.
 * Post: list[i] = in_value, for 0 <= i < n.
 */
void
fill_array (int list[],     /* output - list of n integers     */
            int n,          /* input - number of list elements */
            int in_value)   /* input - initial value           */
{

      int i;            /* array subscript and loop control    */

      for  (i = 0; i < n; ++i)
          list[i] = in_value;
}
```

Data Areas Before Return from `fill_array (x, 5, 1);`



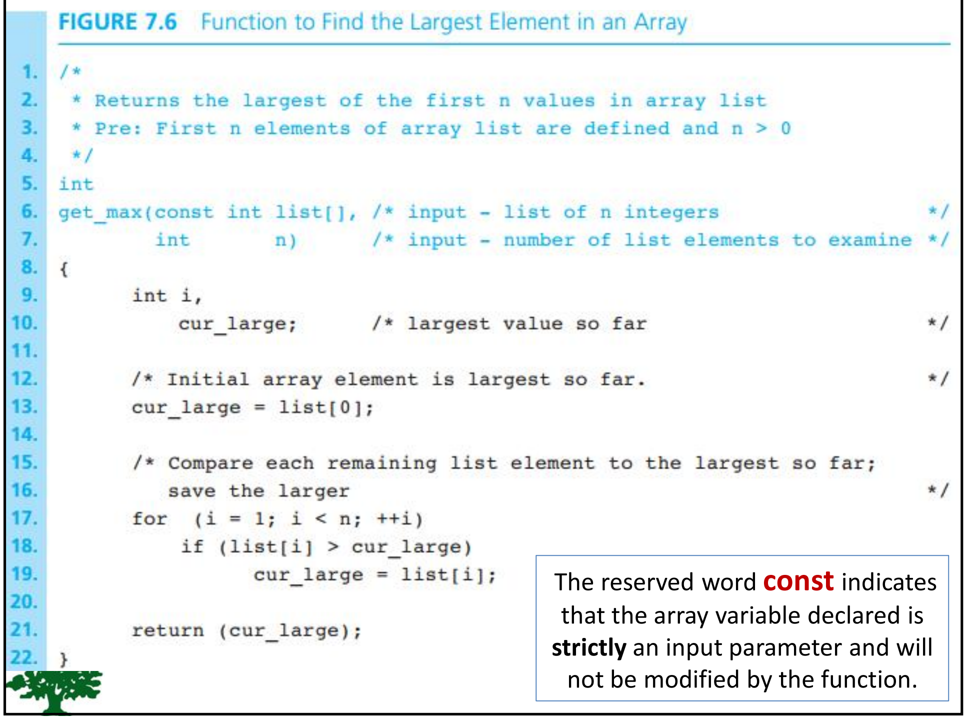


FIGURE 7.6 Function to Find the Largest Element in an Array

```
/*
 * Returns the largest of the first n values in array list
 * Pre: First n elements of array list are defined and n > 0
 */
int
get_max(const int list[], /* input - list of n integers             */
        int       n)      /* input - number of list elements to examine */
{
      int i,
          cur_large;      /* largest value so far                   */

      /* Initial array element is largest so far.                   */
      cur_large = list[0];

      /* Compare each remaining list element to the largest so far;
         save the larger                                            */
      for  (i = 1; i < n; ++i)
          if (list[i] > cur_large)
                cur_large = list[i];

      return (cur_large);
}
```

The reserved word **const** indicates that the array variable declared is **strictly** an input parameter and will not be modified by the function.

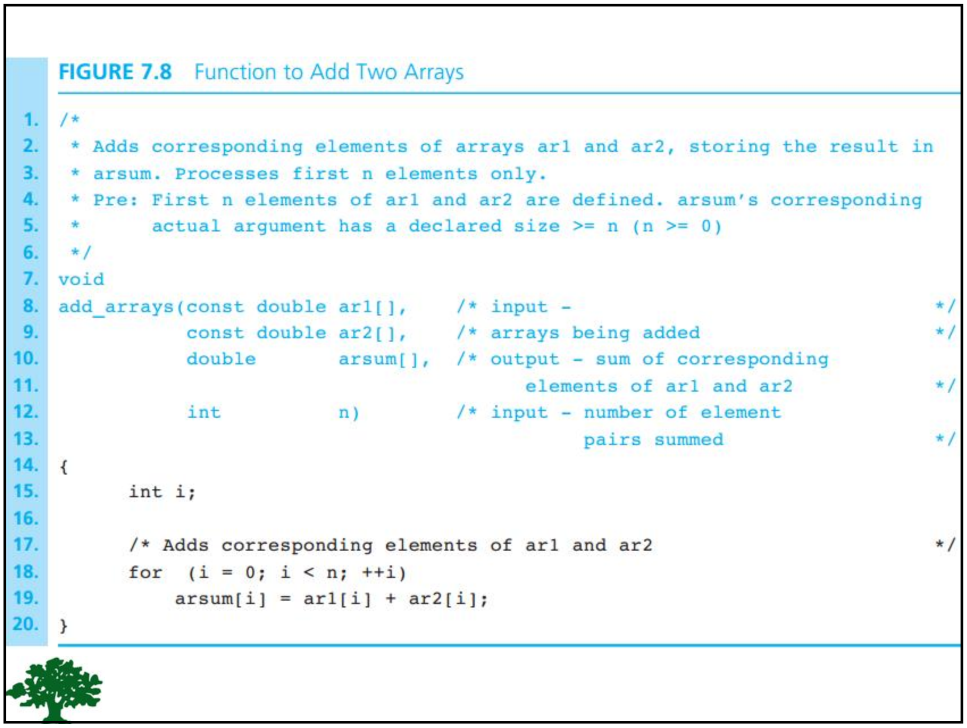


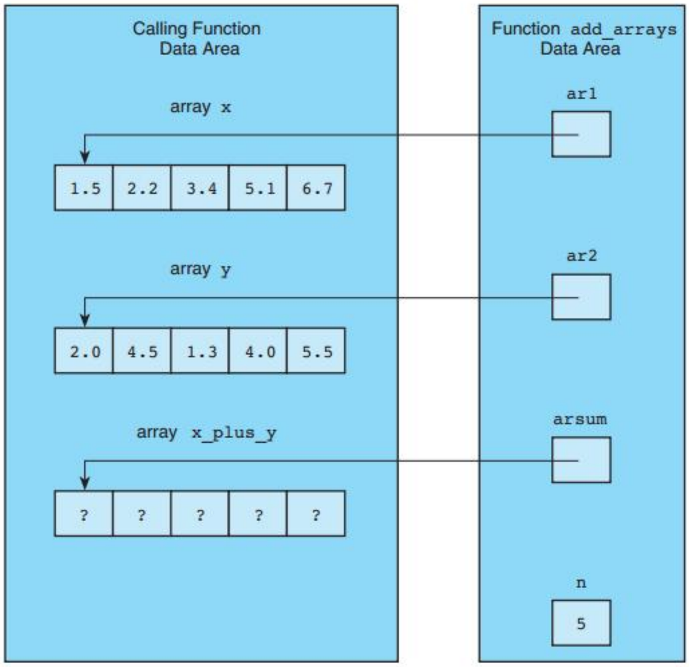
FIGURE 7.8 Function to Add Two Arrays

```
/*
 * Adds corresponding elements of arrays ar1 and ar2, storing the result in
 * arsum. Processes first n elements only.
 * Pre: First n elements of ar1 and ar2 are defined. arsum's corresponding
 *      actual argument has a declared size >= n (n >= 0)
 */
void
add_arrays(const double ar1[],     /* input -                                  */
           const double ar2[],     /* arrays being added                       */
           double       arsum[],   /* output - sum of corresponding
                                          elements of ar1 and ar2              */
           int          n)         /* input - number of element
                                               pairs summed                    */
{
      int i;

      /* Adds corresponding elements of ar1 and ar2                            */
      for  (i = 0; i < n; ++i)
          arsum[i] = ar1[i] + ar2[i];
}
```

FIGURE 7.9

Function Data Areas for `add_arrays(x, y, x_plus_y, 5);`



Returning an Array from a Function

- Returning arrays should work too - **right?**

```
int a[5];
a = foo();
.....
int* foo() {
  int c[5] = {1,2,3,4,5};
  return c;
}
```

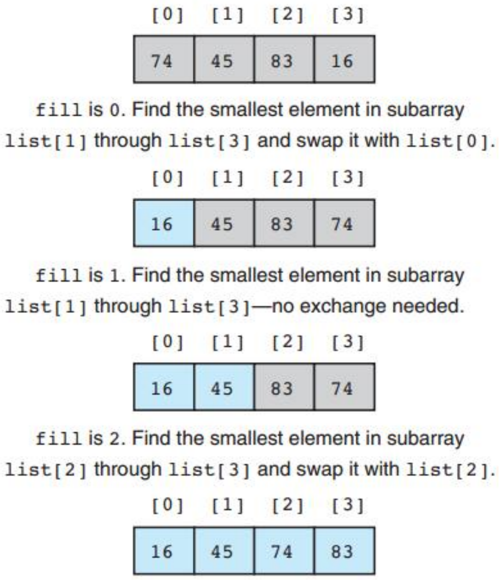
❌

- **Wrong!** In **C**, it is not legal for a function’s return type to be an array.

Partially Filled Arrays

- Frequently, a program will need to process many lists of similar data. These list 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.
- This array can be used for processing shorter lists as well, provided that the program **keeps track of how many array elements are actually in use**.

Sorting an Array – Selection Sort



[0]	[1]	[2]	[3]
74	45	83	16

fill is 0. Find the smallest element in subarray list[1] through list[3] and swap it with list[0].

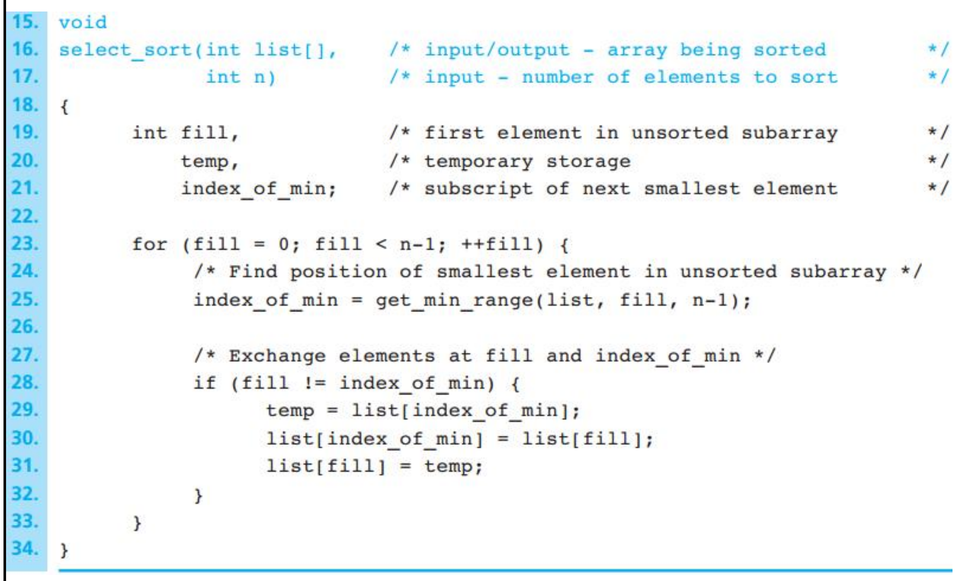
[0]	[1]	[2]	[3]
16	45	83	74

fill is 1. Find the smallest element in subarray list[1] through list[3]—no exchange needed.

[0]	[1]	[2]	[3]
16	45	83	74

fill is 2. Find the smallest element in subarray list[2] through list[3] and swap it with list[2].

[0]	[1]	[2]	[3]
16	45	74	83



```c
void
select_sort(int list[],     /* input/output - array being sorted          */
            int n)          /* input - number of elements to sort         */
{
      int fill,             /* first element in unsorted subarray         */
          temp,             /* temporary storage                          */
          index_of_min;     /* subscript of next smallest element         */

      for (fill = 0; fill < n-1; ++fill) {
           /* Find position of smallest element in unsorted subarray */
           index_of_min = get_min_range(list, fill, n-1);

           /* Exchange elements at fill and index_of_min */
           if (fill != index_of_min) {
                 temp = list[index_of_min];
                 list[index_of_min] = list[fill];
                 list[fill] = temp;
           }
      }
}
```

Multidimensional Arrays

- A multidimensional array is an array with two or more dimensions.
- Thus **int x[3][3]** would define a **3** by **3** matrix that holds integers.
- Initialization is bit different:

  int x[3][3] = { {1,2,3} , {4,5,6} , {7,8,9} };

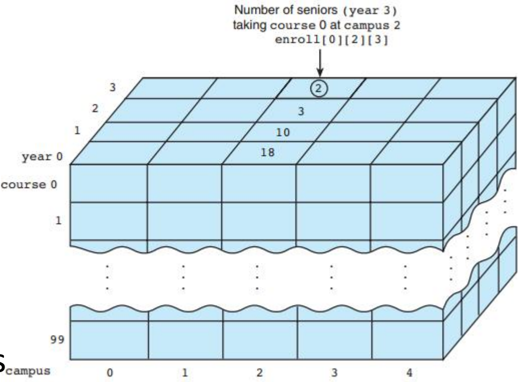
- Both indices starts at **0**.
- Think of it as **x[rows][cols]**.

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

3 Dimensional Array

int enroll[100][5][4];

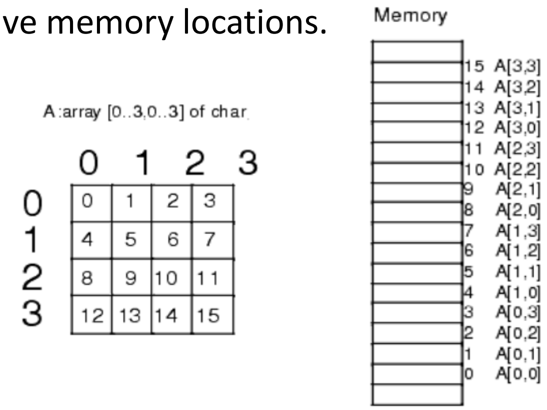
- The 1st dimension stores the course number. 100 of 2D matrices put one under another.
- The 2nd dimension stores campus id. 5 vectors, each 4-element long.
- The last dimension stores the year number.



Number of seniors (year 3) taking course 0 at campus 2 enroll[0][2][3]

2D Array in Address Space

- **How to store a multi-dimensional array into a one-dimensional memory space??**
- Row major ordering assigns successive elements, moving across the rows and then down the columns, to successive memory locations.



A :array [0..3,0..3] of char

	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

Common Programming Errors

- The most common error in using arrays is a **subscript range error**.
- An out-of-range reference occurs when the subscript value is outside the range specified by the array declaration.
  - In some situations, no run-time error message will be produced – the program will simply produce incorrect results.
  - Other times, you may get a runtime error like “segmentation fault” or “bus error”
- Remember how to pass arrays to functions.
- Remember that the first index of the array is **0.**