CMPE 252
C PROGRAMMING

SPRING 2021

WEEK 5-6

ARRAY POINTERS
CHAPTER 7

Problem Solving & Program Design in C

Eighth Edition
Global Edition

Jeri R. Hanly & Elliot B. Koffman

Basic Terminology

- data structure
  - a composite of related data items stored under the same name
- array
  - a collection of data items of the same type

Declaring and Referencing Arrays

- array element
  - a data item that is part of an array
- subscripted variable
  - a variable followed by a subscript in brackets, designating an array element
- array subscript
  - a value or expression enclosed in brackets after the array name, specifying which array element to access

FIGURE 7.1

The Eight Elements of Array x

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

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

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	28.0	26.0	2.5	12.0	14.0	-54.5

Declaration

- You can declare more than one array in a single type declaration
- double cactus[5], needle, pins[6];
- int factor[12], n, index;

Array Initialization

- In declaration time, e.g.
- int prime_lt_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}

- Size is deduced from the initialization list!!

- int x[5] = {10,20,30};

10	20	30	?	?

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'};
```

INTERPRETATION: 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`). 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*.

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*. 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.

Storing a String in a Character Array

- char vowels[] = “hello”;

=

- char vowels[] = {‘h’,’e’,’l’,’l’,’o’,’\0’};

- Escape sequence is string termination null character \0
- Details will come in the following weeks about characters and strings!!

Array Subscripts

- Syntax:

  aname [subscript]

- Examples:

  x[3]

  x[i + 1]

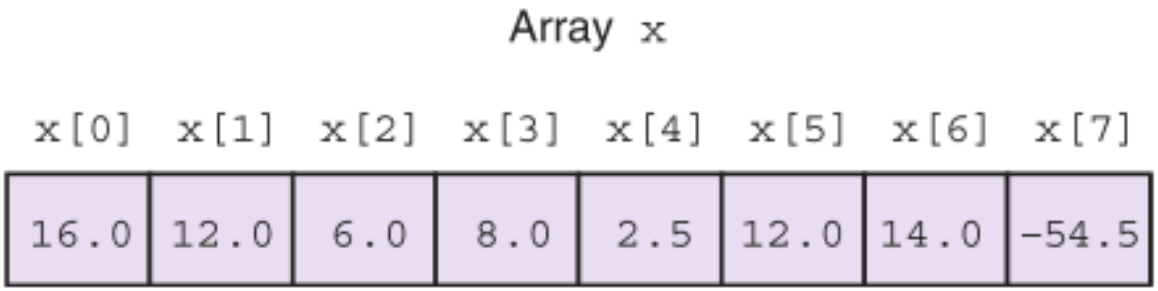
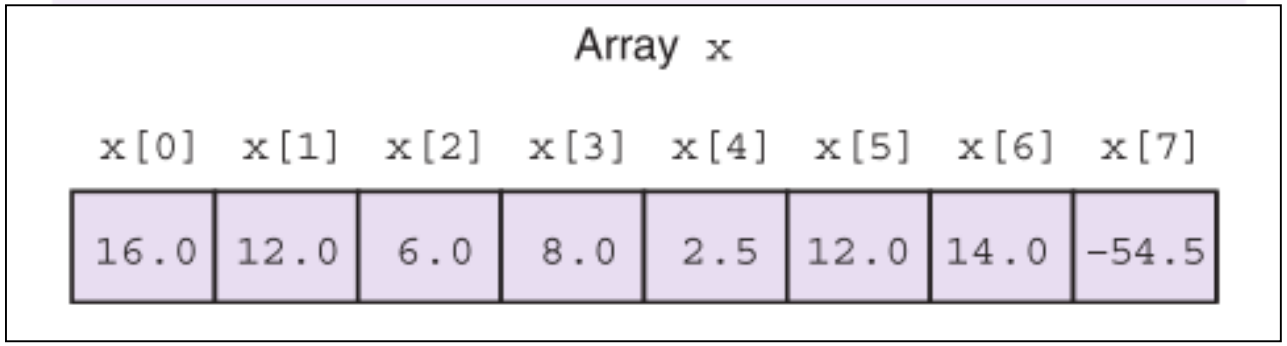


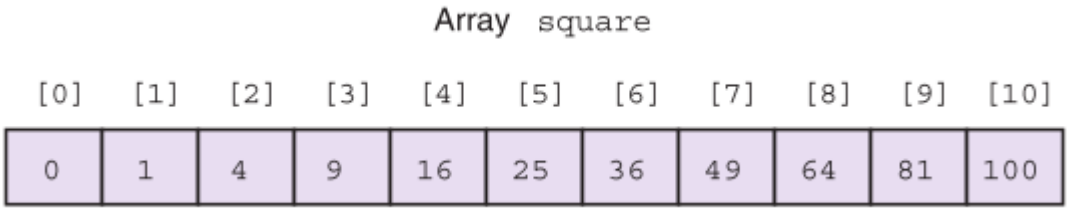
TABLE 7.2 Code Fragment That Manipulates Array x

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

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



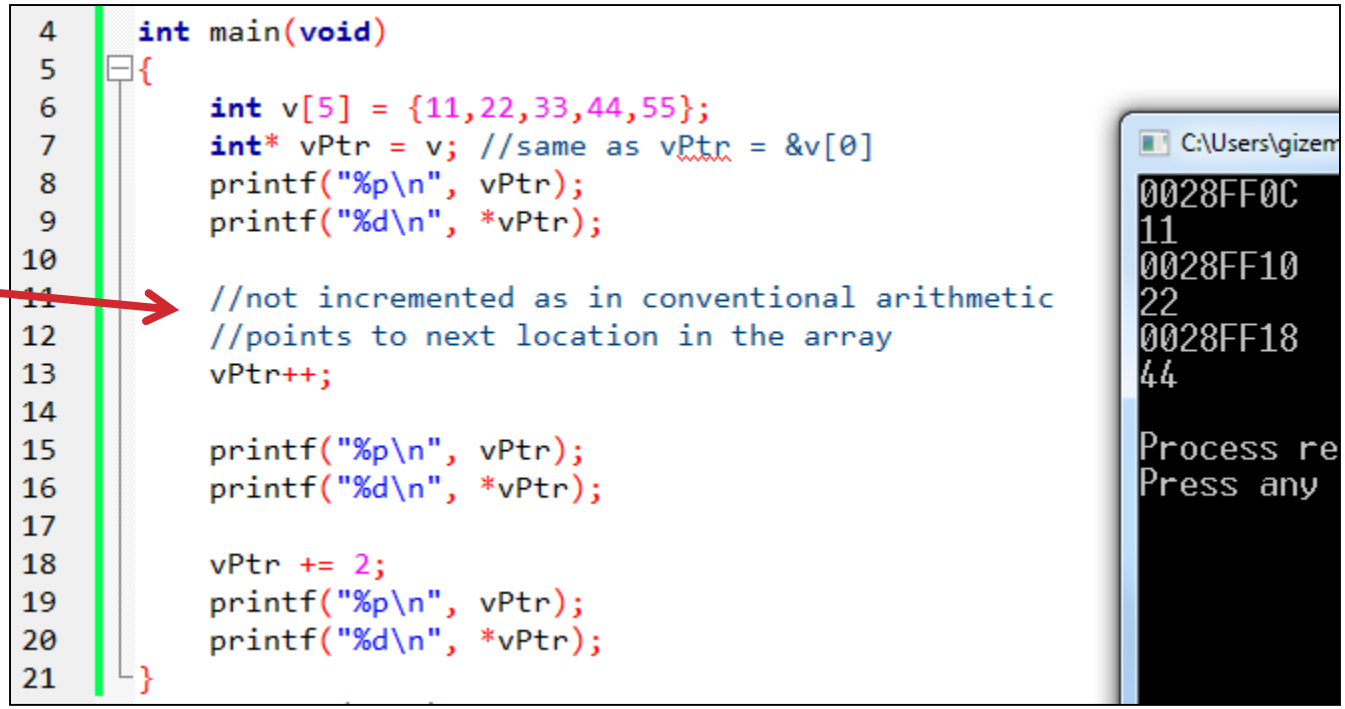
Arrays and Memory

- When we declare an array
  int arr[size]

space is reserved in the memory of the computer for the array.

The elements of the array are stored in these memory locations. The important thing about arrays is that **array elements are always stored in consecutive memory locations.**

Pointers as Array Iterators



Pointers as Array Iterators

- Pointers can be incremented, which make them a natural choice for iterating an array.

```c
#include <stdio.h>

int main (void)
{
    printf("\n");
    int a[5] = {0,2,5,8,11};
    int *ptr = &a[0];
    for (int i = 0; i < 5; i++)
    {
            printf("  %p: %d\n", ptr, *ptr);
            ptr++;
    }

    return 0;
}
```

```
0028FF0C: 0
0028FF10: 2
0028FF14: 5
0028FF18: 8
0028FF1C: 11
```

Pointers as Array Iterators

- If ptr is a pointer, what does ptr[i] mean?

  ptr[i]= *(ptr+i)

```
int v[2] = {2,4};
int* ptr = v;
printf("%p**%d",ptr,ptr[1]);

0028FF18**4
```

Static Local Arrays

- A static local variable exists for the duration of the program but visible only in the function body.
- If an array is declared as static in a function, this array is not created and initialized each time the function is called and therefore, not destroyed each time the function is exited.

```
18  // function to demonstrate a static local array
19  void staticArrayInit(void)
20  {
21     // initializes elements to 0 before the function is called
22     static int array1[3];
23
24     puts("\nValues on entering staticArrayInit:");
25
26     // output contents of array1
27     for (size_t i = 0; i <= 2; ++i) {
28        printf("array1[%u] = %d  ", i, array1[i]);
29     }
30
31     puts("\nValues on exiting staticArrayInit:");
32
33     // modify and output contents of array1
34     for (size_t i = 0; i <= 2; ++i) {
35        printf("array1[%u] = %d  ", i, array1[i] += 5);
36     }
37  }
38
39  //// function to demonstrate an automatic local array
40  void automaticArrayInit(void)
41  {
42     // initializes elements each time function is called
43     int array2[3] = { 1, 2, 3 };
44
45     puts("\n\nValues on entering automaticArrayInit:");
46
47     // output contents of array2
48     for (size_t i = 0; i <= 2; ++i) {
49        printf("array2[%u] = %d  ", i, array2[i]);
50  }
51
52     puts("\nValues on exiting automaticArrayInit:");
53
54     // modify and output contents of array2
55     for (size_t i = 0; i <= 2; ++i) {
56        printf("array2[%u] = %d  ", i, array2[i] += 5);
57     }
58  }
```

```c
#include <stdio.h>

void staticArrayInit(void); // function prototype
void automaticArrayInit(void); // function prototype

// function main begins program execution
int main(void)
{
   puts("First call to each function:");
   staticArrayInit();
   automaticArrayInit();

   puts("\n\nSecond call to each function:");
   staticArrayInit();
   automaticArrayInit();
}
```

```
Values on entering staticArrayInit:
array1[0] = 0  array1[1] = 0  array1[2] = 0
Values on exiting staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5

Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8

Second call to each function:

Values on entering staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5
Values on exiting staticArrayInit:
array1[0] = 10  array1[1] = 10  array1[2] = 10

Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8
```

Using Array Elements as Function Arguments

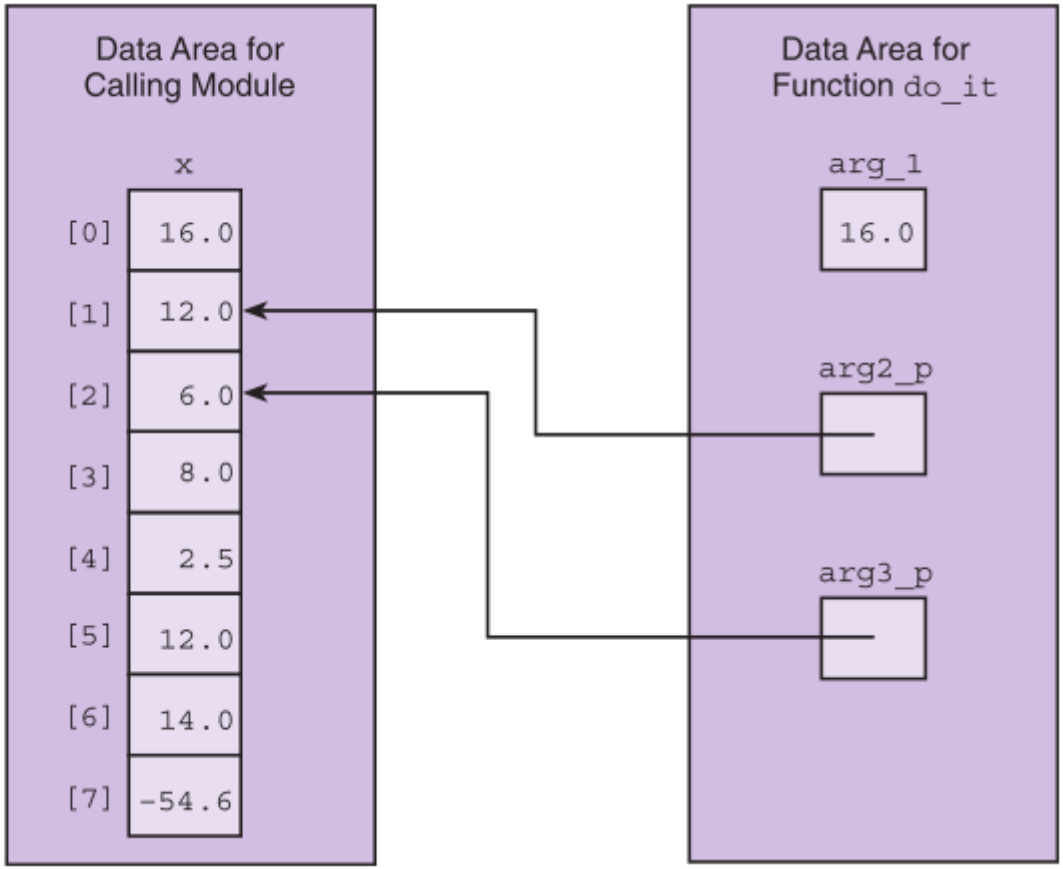
- The call *scanf("%lf", &x[i]);* uses array element x[i] as an output argument of scanf.
- When i=4, a pointer to array element x[4] is passed to scanf, and scanf stores the scanned value in x[4].

- void do_it (double arg_1, double* arg2_p, double* arg3_p)
  {*arg2_p=...., ...}

- double x[3] = {...,...,...}
  - do_it(x[0], &x[1], &x[2]);

FIGURE 7.3

Data Area for Calling Module and Function do_it



Arrays as Arguments

- We can write functions that have arrays as arguments.
- Such functions can manipulate some, or all, of the elements corresponding to an actual array argument.

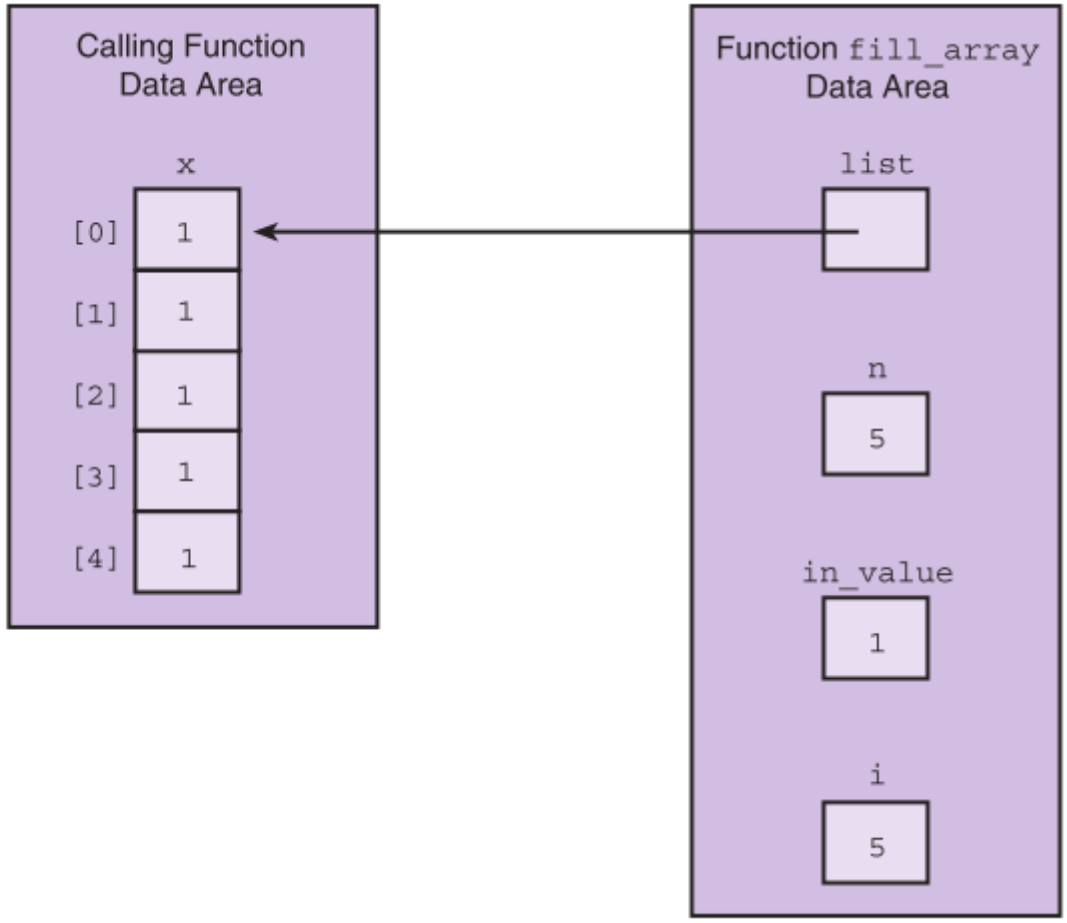
```c
#include <stdio.h>

int main(void)
{
    int arr[5];
    fill_array(arr,5,3);
    for(int i = 0; i < 5; i++)
        printf("Element %d is: %d\n",i, arr[i]);
}
void fill_array (int list[], int n, int in_value)
{
    int i;

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

Notice that it is an output argument

In C, **array name represents address** and when we pass an array, we actually pass address and the parameter receiving function always accepts them as pointers.

FIGURE 7.5

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



Call

Preferred to prevent confusion

```
int arr[5];
fill_array(arr,5,3);
```

The same

```
int arr[5];
fill_array(&arr[0],5,3);
```

```
void fill_array (int list[], int n, int in_value)
{
      int i;

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

How about writing formal parameter as a pointer

```
void fill_array (int list[], int n, int in_value)
{
      int i;

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

Preferred to prevent confusion

The same

```
void fill_array (int* list, int n, int in_value)
{
      int i;

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

Quick Check

- How do you find the length of an array?

```
2
3   int main(void)
4   {
5       int arr[5];
6       fill_array(arr,5,3);
7       for(int i = 0; i < 5; i++)
8           printf("Element %d is: %d\n",i, arr[i]);
9
10      size_t size = sizeof arr / sizeof arr[0];
11      printf("Length is: %d", size);
12  }
```

Output is 5

- When *sizeof()* is used with the data types such as int, float, char… it simply returns the amount of memory allocated to that data types. Cannot be negative therefore declared as size_t.

sizeof

```c
#include <stdio.h>

int main(void)
{
    char c;
    short s;
    int i;
    long l;
    long long ll;
    float f;
    double d;
    long double ld;
    int array[20]; // create array of 20 int elements
    int *ptr = array; // create pointer to array

    printf("     sizeof c = %u\tsizeof(char)  = %u"
           "\n     sizeof s = %u\tsizeof(short) = %u"
           "\n     sizeof i = %u\tsizeof(int) = %u"
           "\n     sizeof l = %u\tsizeof(long) = %u"
           "\n    sizeof ll = %u\tsizeof(long long) = %u"
           "\n     sizeof f = %u\tsizeof(float) = %u"
           "\n     sizeof d = %u\tsizeof(double) = %u"
           "\n    sizeof ld = %u\tsizeof(long double) = %u"
           "\nsizeof array = %u"
           "\n   sizeof ptr = %u\n",
           sizeof c, sizeof(char), sizeof s, sizeof(short), sizeof i
           sizeof(int), sizeof l, sizeof(long), sizeof ll,
           sizeof(long long), sizeof f, sizeof(float), sizeof d,
           sizeof(double), sizeof ld, sizeof(long double),
           sizeof array, sizeof ptr);
}
```

```
    sizeof c = 1        sizeof(char)  = 1
    sizeof s = 2        sizeof(short) = 2
    sizeof i = 4        sizeof(int) = 4
    sizeof l = 4        sizeof(long) = 4
   sizeof ll = 8        sizeof(long long) = 8
    sizeof f = 4        sizeof(float) = 4
    sizeof d = 8        sizeof(double) = 8
   sizeof ld = 12       sizeof(long double) = 12
sizeof array = 80
  sizeof ptr = 4
```

Arrays as Input Arguments

- ANSI C provides 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 the function does not intend to modify the array.
- The qualifier const allows the compiler to mark as an error any attempt to change an array element within the function.

```c
#include <stdio.h>

int main(void)
{
    int largest;
    int arr[] = {34, 56, 12, 4, 89, 45, 78, 21};
    get_max(arr,8,&largest);
    printf("Largest is: %d", largest);
}


void get_max(const int list [], /* input - list of n integers
         int n, int* largest)
{
      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];

    *largest = cur_large;
}
```

Edit Attempt to Const Array

```
12  void get_max(const int list [], /* input - list of n integers
13          int n, int* largest)
14  {
15        int i,cur_large;       /* largest value so far
16
17        /* Initial array element is largest so far.                    */
18        cur_large = list[0];
19
20        /* Compare each remaining list element to the largest so far;
21           save the larger                                              */
22        for  (i = 1; i < n; ++i)
23            if (list [i] > cur_large)
24                  cur_large = list [i];
25
26     list[i] = 5;
27     *largest = cur_large;
28  }
```

& others

Code::Blocks | Search results | Cccc | Build log | **Build messages** | CppCheck/Vera++

e	Line	Message
		=== Build: Release in W4 (compiler: GNU GCC Compiler) ===
.Users\gizem...		In function 'main':
.Users\gizem...	7	*warning: implicit declaration of function 'get_max' [-Wimplicit-function-declaration]*
.Users\gizem...	12	*warning: conflicting types for 'get_max'*
.Users\gizem...	7	note: previous implicit declaration of 'get_max' was here
.Users\gizem...		In function 'get_max':
.Users\gizem...	26	error: assignment of read-only location '*(list + (sizetype)((unsigned int)i * 4u))'
		=== Build failed: 1 error(s), 2 warning(s) (0 minute(s), 0 second(s)) ===

const

- const keyword prevents you from modifying the value of a particular variable, not only arrays

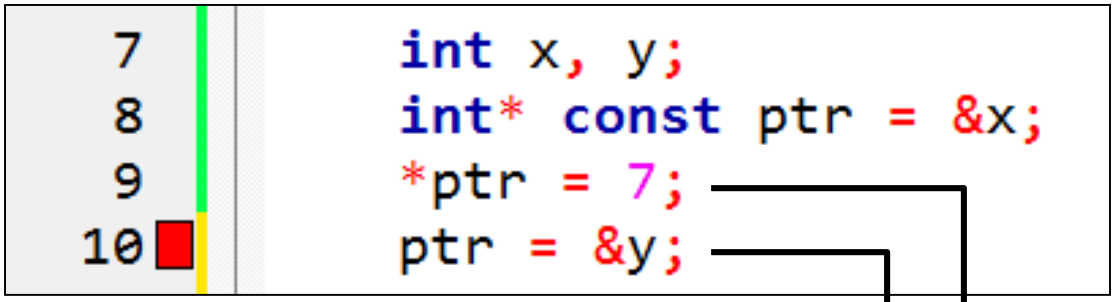
```
int const x = 5;
x = 7;
```

```
const int x = 5;
x = 7;
```

Both create constant integer x
2nd assignments (x=7) give error in both cases
«error: assignment of read-only variable ‘x’»

const pointer

- How to declare a constant pointer that cannot be changed to point to something else:



const pointer

```
int a = 5, b = 7;
int *p1, *p2;
int* const p3 = &a;

p1 = &b;
p2 = p1;

printf("p1: %p, p2: %p, p3: %p\n",p1,p2,p3);
printf("\n*p1: %d, *p2: %d, *p3: %d\n",*p1,*p2,*p3);

//p3 = &b;
```

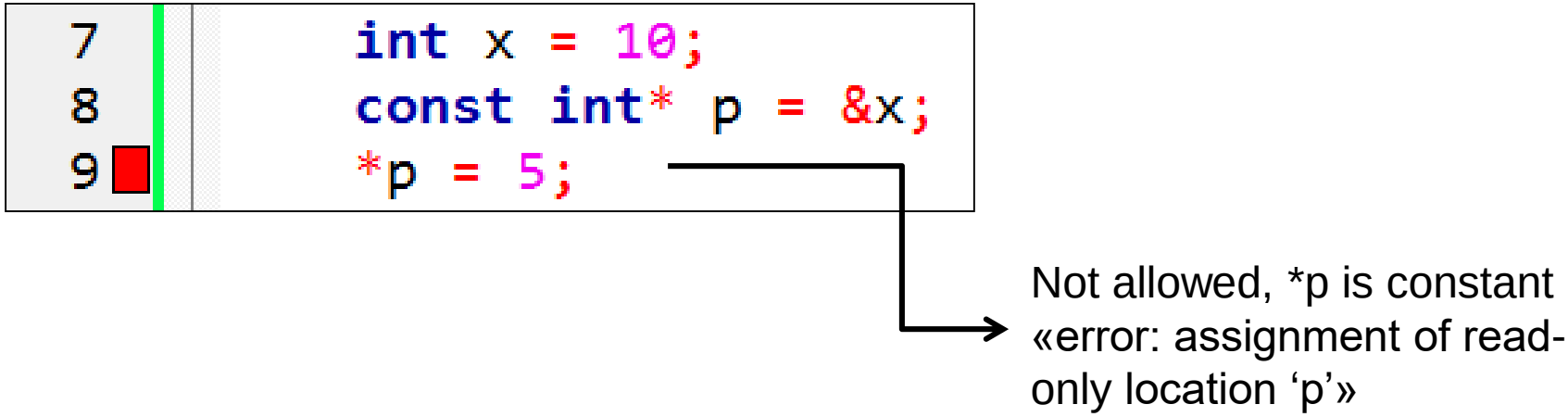
P2 and p1 points to the same location

```
p1: 0028FF1C, p2: 0028FF1C, p3: 0028FF18

*p1: 7, *p2: 7, *p3: 5
```

error

const data

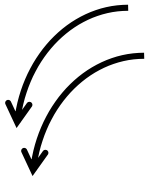
- How to declare a pointer whose *data* cannot be changed:



const data and pointer

- How to grant the least access privilege?
  - const pointer to const data
    - ptr always points to the same location
    - the integer at the location can never be changed

```
int x = 10, y;
const int* const ptr = &x;
*ptr = 7;
ptr = &y;
```



error: assignment of read-only location ‘*ptr’

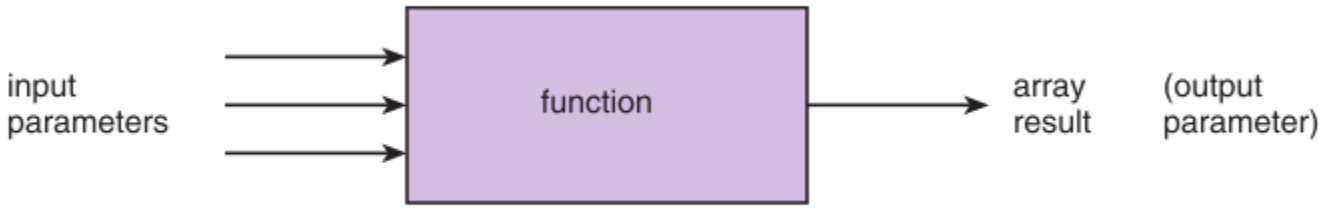
error: assignment of read-only variable ‘ptr’

Returning an Array Result

- In C, it is not legal for a function's return type to be an array.
- You need to use an output parameter to send your array back to the calling module.

FIGURE 7.7

Diagram of a Function That Computes an Array Result

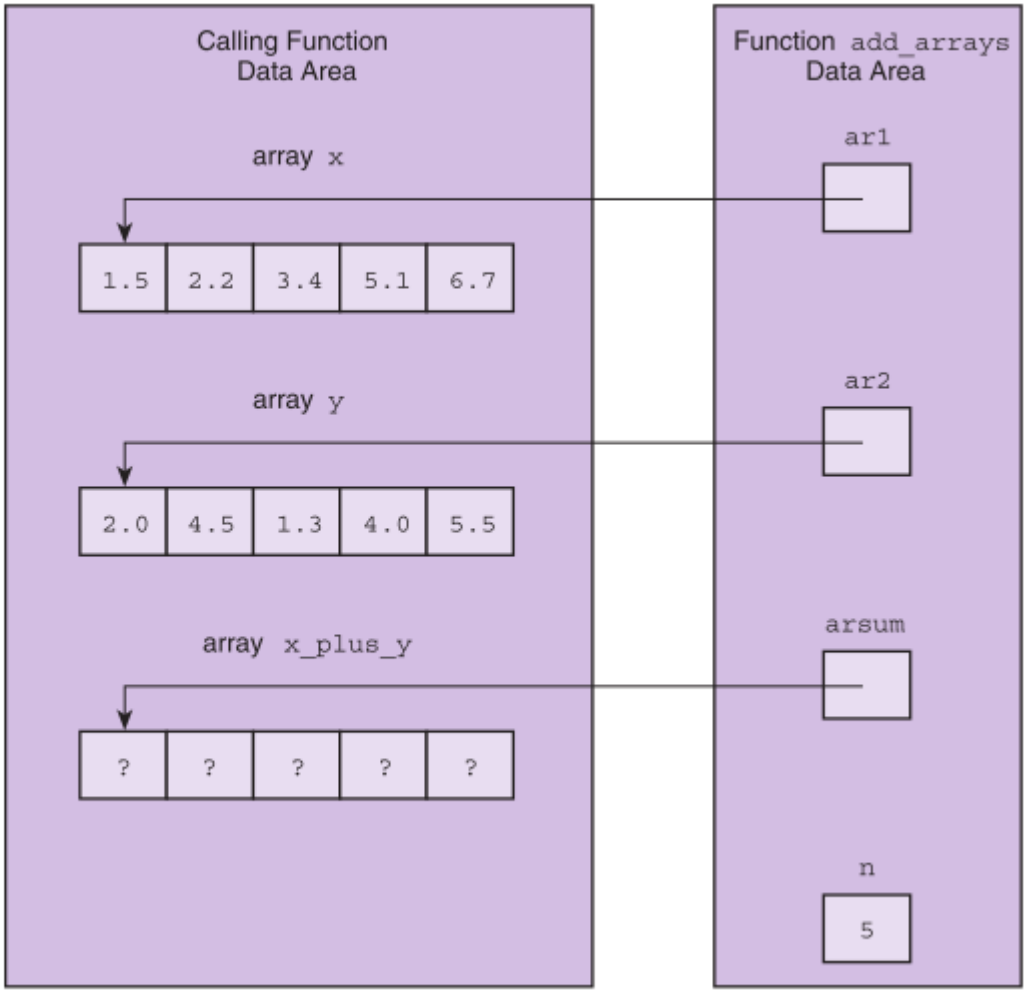


```c
#include<stdio.h>

int main(void)
{
    double x[5]={1.5,2.2,3.4,5.1,6.7};
    double y[5]={2.0,4.5,1.3,4.0,5.5};
    double x_plus_y[5];
    add_arrays(x,y,x_plus_y,5);
    for  (int i = 0; i < 5; ++i)
        printf("%.2f\n",x_plus_y[i]);
}

void add_arrays(const double ar1[], const double ar2[],
                double arsum[],  int n)

{
      for  (int i = 0; i < n; ++i)
          arsum[i] = ar1[i] + ar2[i];
}
```

No addres of operator needed during call of output parameter

FIGURE 7.9

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



Does the array name also take a space in memory in C?

Consider the following array definition:

int arr[3];

The array name arr is just a symbol that represent the location of the first byte of the allocated memory.

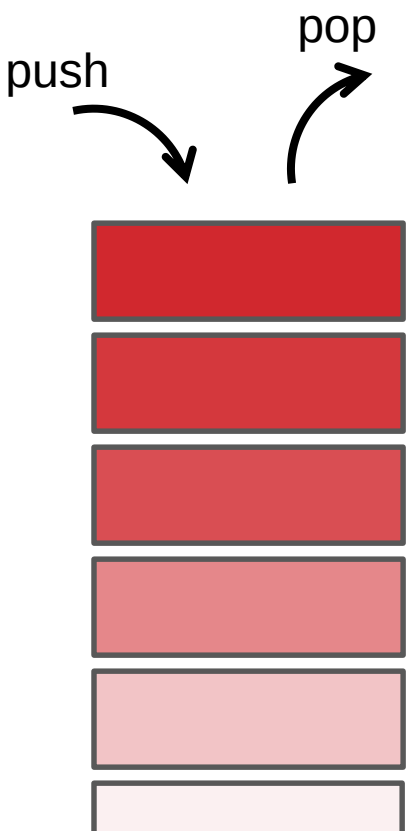
→ The name of an array does not hold extra space in memory.

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int arr[3] = {10,20,30};
6      int x = 5;
7      int *ptr = &x;
8      arr = &x;
9      return 0;
10 }
```

Produces compile time error.

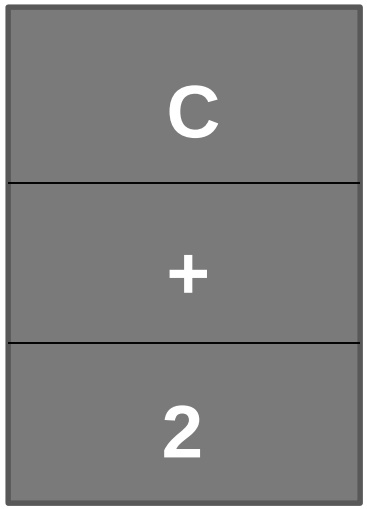
Stack

- A stack is a data structure in which only the top element can be accessed.
- pop
  - remove the top element of a stack
- push
  - insert a new element at the top of the stack



Stack Example

- Push characters 2, +, C in order



- The only character that we can access is:
  - C
- If we want to access +
  - We have to remove C

Stack Example

- Let’s assume we have a character array called stack_s
- How can we write push, pop and retrieve functions?
  - Push: pushes item and increments the subscript of the element at top of the stack
  - Pop: removes top item and decrements the subscript of the element at top of the stack
  - Retrieve: Accesses the element at the top of the stack without removing it.
  - HOA

```c
#include <stdio.h>
#define MAXSIZE 10
#define STACK_EMPTY 0

void push(char stack[], char item, int  *top, int  max_size);
char pop(char stack[], int *top);
char retrieve(const char stack[], int top);
void print_char_stack(const char stack[], int s_top);

int main(void)
{
    char stack_s[MAXSIZE];
    int s_top = -1; // subscript of the element at top of the sta
    char ch1,ch2;

    push(stack_s,'2',&s_top,MAXSIZE);
    push(stack_s,'+',&s_top,MAXSIZE);
    push(stack_s,'C',&s_top,MAXSIZE);

    printf("Before pop, stack looks like:\n");
    print_char_stack(stack_s, s_top);

    ch1 = pop(stack_s,&s_top);
    printf("\nAfter pop, stack looks like:\n");
    print_char_stack(stack_s, s_top);
    printf("\nch1 is: %c\n", ch1);

    ch2 = retrieve(stack_s,s_top);
    printf("\nch2 is: %c\n", ch2);
    printf("\nAfter retrieve, stack looks like:\n");
    print_char_stack(stack_s, s_top);

    return 0;
}
```

```
36  void push(char stack[],    /* input/output - the stack */
37       char item,        /* input - data being pushed onto the stack */
38       int  *top,        /* input/output - pointer to top of stack */
39       int  max_size)    /* input - maximum size of stack */
40  {
41       if (*top < max_size-1) {
42           ++(*top);
43           stack[*top] = item;
44       }
45  }
46
47  char pop(char stack[],    /* input/output - the stack */
48      int *top)         /* input/output - pointer to top of stack */
49  {
50      char item;        /* value popped off the stack */
51
52      if (*top >= 0) {
53          item = stack[*top];
54          --(*top);
55      } else {
56           item = STACK_EMPTY;
57      }
58
59      return (item);
60  }
```

```
62  char retrieve(const char stack[], /* input - the stack */
63                int       top)     /* input - stack top subscript */
64      {
65          char item;
66
67          if (top >= 0)
68              item = stack[top];
69          else
70              item = STACK_EMPTY;
71
72          return (item);
73      }
74
75  void print_char_stack(const char stack[], int s_top)
76  {
77      for(int i = s_top; i > -1; i--)
78          printf("%c\n", stack[i]);
79  }
```

Array Search - Linear

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.

```c
#define NOT_FOUND -1

/*
 * Searches for target item in first n elements of array arr
 * Returns index of target or NOT_FOUND
 * Pre:  target and first n elements of array arr are defined and n>=0
 */
int search(const int arr[],  /* input - array to search */
           int target, /* input - value searched for */
           int n)      /* input - number of elements to search */
{
      int i,
          found = 0,     /* whether or not target has been found */
          where;         /* index where target found or NOT_FOUND */

      /* Compares each element to target*/
      i = 0;
      while (!found && i < n) {
          if (arr[i] == target)
                found = 1;
          else
                ++i;
      }

      /* Returns index of element matching target or NOT_FOUND */
      if (found)
            where = i;
      else
            where = NOT_FOUND;

      return (where);
}
```

Bubble Sort – 1

Repeatedly swaps the adjacent elements if they are in wrong order.

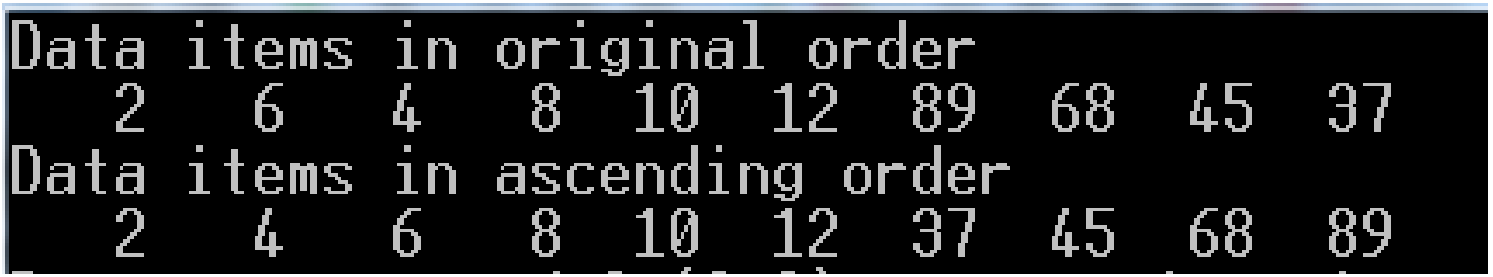
Example:
First Pass:
(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:
(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.
Third Pass:
(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

How do you implement bubble sort?

```c
#include <stdio.h>
#define SIZE 10

int main(void)
{
   // initialize a
   int a[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};

   puts("Data items in original order");

   // output original array
   for (size_t i = 0; i < SIZE; ++i)
      printf("%4d", a[i]);

   for (int pass = 1; pass < SIZE; ++pass)
   {
      for (int i = 0; i < SIZE - 1; ++i)
      {
         if (a[i] > a[i + 1])
         {
            int hold = a[i];
            a[i] = a[i + 1];
            a[i + 1] = hold;
         }
      }
   }

   puts("\nData items in ascending order");

   // output sorted array
   for (size_t i = 0; i < SIZE; ++i)
      printf("%4d", a[i]);

   return 0;
}
```


Data items in original order
2 6 4 8 10 12 89 68 45 37
Data items in ascending order
2 4 6 8 10 12 37 45 68 89

Bubble Sort – 2 (pass by reference)

- Write the bubble sort program again using two functions:
  - bubble_sort (int* const array, const size_t size)
  - swap (int* el1ptr, int* el2ptr)
  - see main function on next slide

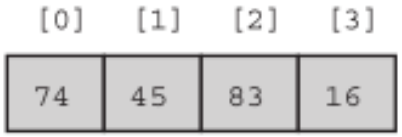
Bubble Sort – 2 (pass by reference)

```c
#include <stdio.h>
#define SIZE 10

void bubble_sort(int * const array, const size_t size);
void swap(int *element1Ptr, int *element2Ptr);

int main(void)
{
   int a[SIZE] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

   puts("Data items in original order");

   for (size_t i = 0; i < SIZE; ++i)
      printf("%4d", a[i]);

   bubble_sort(a, SIZE);

   puts("\nData items in ascending order");

   // loop through array a
   for (size_t i = 0; i < SIZE; ++i)
      printf("%4d", a[i]);

   return 1;
}
```

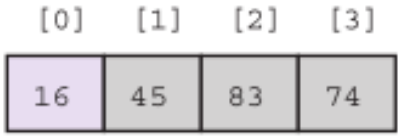
```
26
27  void bubble_sort(int * const array, const size_t size)
28  {
29     for (unsigned int pass = 0; pass < size - 1; ++pass)
30        for (size_t j = 0; j < size - 1; ++j)
31           if (array[j] > array[j + 1])
32              swap(&array[j], &array[j + 1]);
33  }
34
35  void swap(int *element1Ptr, int *element2Ptr)
36  {
37     int temp = *element1Ptr;
38     *element1Ptr = *element2Ptr;
39     *element2Ptr = temp;
40  }
```

Selection Sort

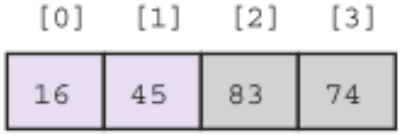
1. for each value of fill from 0 to n-2
    2. Find index_of_min, the index of the smallest element in the unsorted subarray list[fill] through list[n-1]
    3. if fill is not the position of the smallest element (index_of_min)
        4. Exchange the smallest element with the one at position fill.



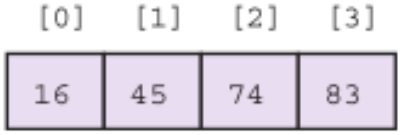
`fill` is `0`. Find the smallest element in subarray `list[1]` through `list[3]` and swap it with `list[0]`.



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



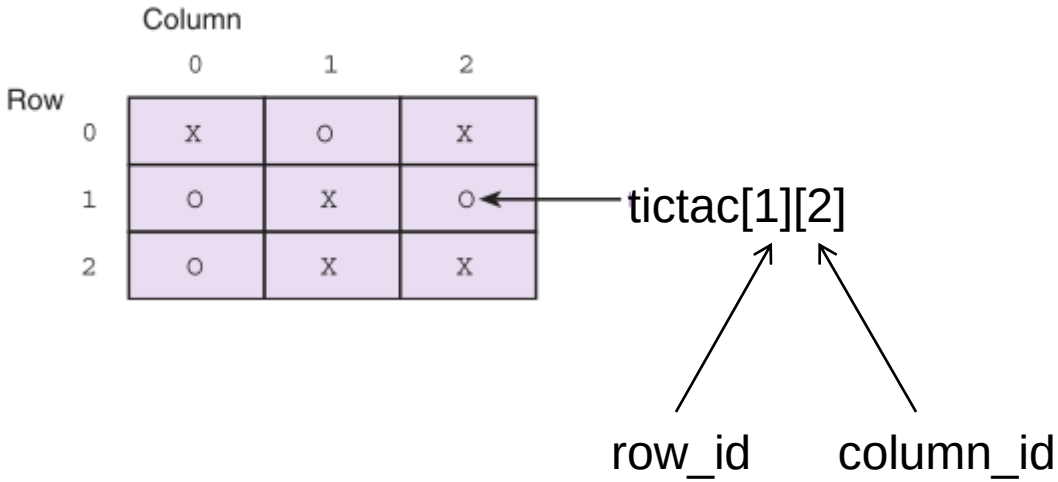
`fill` is `2`. Find the smallest element in subarray `list[2]` through `list[3]` and swap it with `list[2]`.



```c
int get_min_range(int list[], int first, int last);

void select_sort(int list[], int n)
{
     int fill,
         temp,
         index_of_min;

     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;
          }
     }
}

int get_min_range(int list[], int first, int last)
{
       int i,           /* Loop Control Variable (LCV)        */
       small_sub;       /* subscript of smallest value so far */

       small_sub = first;  /* Assume first element is smallest   */

       for (i = first + 1; i <= last; ++i)
          if (list[i] < list[small_sub])
             small_sub = i;

       return (small_sub);
}
```

Introduction to Multidimensional Arrays

- multidimensional array
  - an array with two or more dimensions
- multidimensional array parameter declaration
  - char tictac[3][3];



Multidimensional Array as Argument

```
21  int ttt_filled(char ttt_brd[3][3])
22  {
23        int r, c, /* row and column subscripts  */
24            ans; /* whether or not board filled */
25
26        /* Assumes board is filled until blank is found*/
27        ans = 1;
28
29        /* Resets ans to zero if a blank is found */
30        for (r = 0; r < 3; ++r)
31           for (c = 0; c < 3; ++c)
32              if (ttt_brd[r][c] == ' ')
33                    ans = 0;
34
35        return (ans);
36  }
```

Multidimensional Array as Argument

```
1  #include <stdio.h>
2  int ttt_filled(char ttt_brd[3][3]);
3
4  int main(void)
5  {
6      // initialize during declaration
7      // group by rows
8      char ttt[3][3] = { {'x','o',' '},
9                         {'x','o','x'},
10                        {'o','x','x'},
11                    };
12
13     int filled = ttt_filled(ttt);
14     printf("%d",filled);
15
16     return 0;
17 }
```

Output: 0

- HOA

Wrap Up

- A data structure is a grouping of related data items in memory.

- An array is a data structure used to store a collection of data items of the same type.

References

1. Problem Solving & Program Design in C, Jeri R. Hanly & Elliot B. Koffman, Pearson 8. Edition, Global Edition
2. C How to Program, Paul Deitel, Harvey Deitel. Pearson 8th Edition, Global Edition.
3. https://www.geeksforgeeks.org/bubble-sort/