

CMPE 252

C PROGRAMMING

SPRING 2022

WEEK 4 & 5 & 6

POINTERS AND MODULAR PROGRAMMING

CHAPTER 6

Problem Solving & Program Design in C

Eighth Edition

Global Edition

Jeri R. Hanly & Elliot B. Koffman

Chapter Objectives

- To learn about pointers and indirect addressing / indirect reference
- To see how to access external data files in a program and to be able to read from input file and write to output files using file pointers
- To learn how to return function results through a function's arguments
- To understand the differences between call-by-value and call-by-reference

Chapter Objectives

- To understand the distinction between input, inout, and output parameters and when to use each kind
- To learn how to modularize a program system and pass information between system modules (functions)
- To learn how to write functions that can return multiple output
- To understand how to document the flow of information using structure charts
- To learn testing and debugging techniques appropriate for a program system with several modules

Pointers

- pointer (pointer variable)
 - a memory cell that stores the address of a data item
 - syntax: *type *variable*

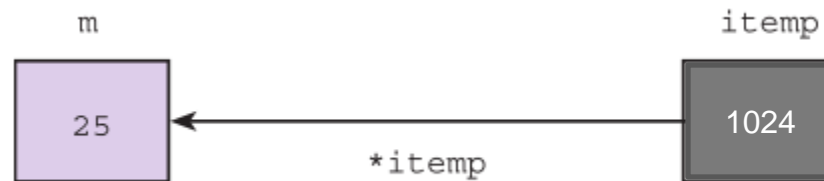
```
int m = 25;  
int *itemp;      /* a pointer to an integer */
```

- How can we store the memory address of **m** in pointer **itemp**?
- Using unary address-of operator &
 - **itemp = &m;**

Indirection

- indirect reference
 - accessing the contents of a memory cell through a pointer variable that stores its address

* is unary
indirection
operator



Assume that variable `m` is associated with memory cell 1024

TABLE 6.1 References with Pointers

Reference	Cell Referenced	Cell Type (Value)
<code>itemp</code>	gray shaded cell	pointer (1024)
<code>*itemp</code>	cell in color	int (25)

NULL Pointer

- Pointers should be initialized when they're defined or they can be assigned a value.
- A pointer may be initialized to NULL, 0 or an address.

```
int * pInt = NULL;
```

- A pointer with the value NULL points to nothing.
- NULL is a symbolic constant defined in the `<stddef.h>` header (and several other headers, such as `<stdio.h>`).

NULL Pointer

- Initializing a pointer to 0 is equivalent to initializing a pointer to NULL, but NULL is preferred.
- When 0 is assigned, it's first converted to a pointer of the appropriate type.
- The value 0 is the only integer value that can be assigned directly to a pointer variable.

NULL Pointer

- NULL pointer is different from uninitialized and dangling pointer.
- All dangling or NULL pointers are invalid but NULL is a specific invalid pointer which is mentioned in C standard and has specific purposes.
- Uninitialized and dangling pointers are invalid but they can point to some memory address which is dangerous.

NULL Pointer – Where to Use

- NULL is useful when we want to dereference.
Dereference pointer variable only if it's not NULL.

```
if(pInt != NULL) //We can also use if(pInt)
{ /*Some code*/}
else
{ /*Some code*/}
```

- NULL is also used to pass a null pointer to a function argument when we don't want to pass any valid memory address.

```
int fun(int *ptr)
{
    /*Fun specific stuff is done with ptr here*/
    return 10;
}
fun(NULL);
```

Quick Check

```
#include <stdio.h>
int main(void)
{
    int *i, *j;
    int *ii = NULL, *jj = NULL;
    if(i == j)
    {
        printf("This might get printed if both i and j are same by chance.");
    }
    if(ii == jj)
    {
        printf("This is always printed because ii and jj are same.");
    }

    return 0;
}
```

Output: ??

This is always printed because ii and jj are same.

Pointers Quick Check

Output?

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int m = 25;
7      int *itemp;
8      itemp = &m;
9
10     printf("%d\n", itemp);
11     printf("%d\n", *itemp);
12
13     *itemp = 35;
14     printf("%d\n", m);
15
16     *itemp = 2 * (*itemp);
17     printf("%d\n", m);
18     printf("%d\n", *itemp);
19     printf("%d\n", itemp);
20
21     *itemp = 3 * m;
22     printf("%d\n", m);
23     printf("%d\n", *itemp);
24     printf("%d\n", itemp);
25 }
26


```

2686744
 25
 35
 70
 70
 2686744
 210
 210
 2686744

Pointers Quick Check

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int m = 10, n = 5;
7      int *mp, *np;
8      mp = &m;
9      np = &n;
10
11      *mp = *mp + *np;
12      *np = *mp - *np;
13      printf("m      *mp      n      *np \n%d%4d%5d%6d", m, *mp, n, *np);
14  }
```

Output?



m	*mp	n	*np
15	15	10	10

- What do I do if I want two pointers both of which point at the same cell?
 - `int* p1;`
 - `int* p2;`
 - `int a;`
 - `p1 = &a;`
 - `p2 = p1;`

Pointers Quick Check

```
int a = 7;
int *aPtr = &a;

printf("The address of a is: %p"
      "\nThe value of aPtr is %p", &a, aPtr);

printf("\n\nThe value of a is: %d"
      "\nThe value of *aPtr is: %d", a, *aPtr);

printf("\n\nShowing that * and & are complements of each other\n"
      "&*aPtr = %p\n*&aPtr = %p\n", &*aPtr, *&aPtr);
```

```
The address of a is: 0028FF1C
The value of aPtr is 0028FF1C
```

```
The value of a is: 7
The value of *aPtr is: 7
```

```
Showing that * and & are complements of each other
&*aPtr = 0028FF1C
*&aPtr = 0028FF1C
```

%p %x %X

```
int a = 7;  
int *aPtr = &a;  
  
printf("The address of a using p is: %p"  
       "\nThe address of a using x is: %x"  
       "\nThe address of a using X is: %X\n\n", aPtr, aPtr, aPtr);
```

```
The address of a using p is: 0028FF1C  
The address of a using x is: 28ff1c  
The address of a using X is: 28FF1C
```

"p" serves to output a pointer. It may differ depending upon the compiler and platform but in our case it gives hexadecimal number.

"x" and "X" serve to output a hexadecimal number. "x" stands for lower case letters (abcdef) while "X" for capital letters (ABCDEF).

There is no uppercase p


Why to use pointers over normal variables?

- Pointers are to be used where you can't use anything else. It is either because the lack of appropriate functionality, missing data types or for pure performance.
- Where/When?
 - In C you don't have any support for complex datatypes such as a string.
 - There is also no way of passing a variable "by reference" to a function.
 - Also you can have them to point at virtually anything, linked lists, members of structs and so on.

More examples

- `int *ip=NULL;`
- `int i =5;`
- `int k=0;`
- `ip=&i;`
- `k = ++ (*ip);`
versus?
- `k=++i;`

Operator Precedence (more)

Operator	Precedence
function calls (), postfix increment (i++), postfix decrement (i--)	highest (evaluated first)
! + - (unary operator), prefix increment (++i), prefix decrement (--i), & address of, * indirection [RIGHT TO LEFT]	
* / %	
+ -	
< <= >= >	
== !=	
&&	
	lowest (evaluated last)
=	

More examples

Operator	Precedence
function calls (), postfix increment (i++), postfix decrement (i--)	highest (evaluated first)
! + - (unary operator), prefix increment (++i), prefix decrement (--i), & address of, * indirection [RIGHT TO LEFT]	
* / %	
+ -	
< <= >= >	
== !=	
&&	
=	lowest (evaluated last)

- `char c = 5;`
- `char *cp = &c;`
- `c = *++cp;` → `c = *(++cp)` : first increment `cp`, then fetch character it points to.
- `c = *cp++;` → `c = *(cp++)` : get initial `cp`, fetch character it points to, then increment `cp`.
- `c = ++*cp;` → `c = ++(*cp)` fetch value pointed by `cp`, increment, assign it to `c`. (`cp` remains unchanged)

Pass-by-reference vs Pass-by-value

pass by reference

cup = 

fillCup()

pass by value

cup = 

fillCup()

Pass-by-value

```
1  #include <stdio.h>
2
3  void f(int a)
4  {
5      a = 3;
6  }
7
8  int main()
9  {
10     int x = 1;
11
12     printf("Before calling function f\n");
13     printf("x = %d\n", x);
14     f(x);
15     printf("After calling function f\n");
16     printf("x = %d\n", x);
17
18     return 0;
19 }
20
```

```
Before calling function f
x = 1
After calling function f
x = 1
```

Simulation of Pass-by-reference in C

```
1  #include <stdio.h>
2
3  void g(int *aPtr)
4  {
5      *aPtr = 3;
6  }
7
8  int main()
9  {
10     int x = 1;
11
12     printf("Before calling function g\n");
13     printf("x = %d\n", x);
14     g(&x);
15     printf("After calling function g\n");
16     printf("x = %d\n", x);
17
18     return 0;
19 }
20
```

```
Before calling function g
x = 1
After calling function g
x = 3
```

Pass-by-value vs Pass-by-reference

- In C, arguments are passed by value, however, some cases can occur where either:
 - functions require to modify the variables in caller
 - or receive a pointer to a large data since receiving large objects by value is a serious overhead
 - making a copy of an object consumes time and memory

Pass-by-value Example

```
1  #include <stdio.h>
2  #include <math.h>
3  double powerFourByValue(double n);
4
5  int main(void)
6  {
7      double number = 5.0; // initialize number
8
9      printf("The original value of number is %.2f", number);
10
11     // pass number by value to powerFourByValue
12     number = powerFourByValue(number);
13
14     printf("\nThe new value of number is %.2f\n", number);
15 }
16
17 // calculate and return cube of integer argument
18 double powerFourByValue(double n)
19 {
20     return pow(n,4);
21 }
```

Pass-by-reference Example

```
1  #include <stdio.h>
2  #include <math.h>
3  void powerFourByReference(double *nPtr);
4
5  int main(void)
6  {
7      double number = 7.2; // initialize number
8
9      printf("The original value of number is %.2f", number);
10
11     // pass number by reference to powerFourByReference
12     powerFourByReference(&number);
13
14     printf("\nThe new value of number is %.2f\n", number);
15 }
16
17 // calculate and return cube of integer argument
18 void powerFourByReference(double *nPtr)
19 {
20     *nPtr = pow(*nPtr, 4);
21 }
```

Analysis of Pass-by-value

Step 1 – before calling powerFourByValue function

```
int main(void)
{
    double number = 5.0; // initialize number
    number = powerFourByValue(number);
}
```

number

5.0

```
double powerFourByValue(double n)
{
    return pow(n,4);
}
```

n

undefined

Step 2 – after calling powerFourByValue function

```
int main(void)
{
    double number = 5.0; // initialize number
    number = powerFourByValue(number);
}
```

number

5.0

```
double powerFourByValue(double n)
{
    return pow(n,4);
}
```

n

5.0

Analysis of Pass-by-value

Step 3 – after powerFourByValue computes power and before returns the value

```
int main(void)
{
    double number = 5.0; // initialize number
    number = powerFourByValue(number);
}
```

number

5.0

```
double powerFourByValue(double n)
{
    return pow(n,4);
}
```

625.0

n

5.0

Step 4 – after powerFourByValue returns and before it is assigned to number

```
int main(void)
{
    double number = 5.0; // initialize number
    number = powerFourByValue(number);
}
```

625.0

number

5.0

```
double powerFourByValue(double n)
{
    return pow(n,4);
}
```

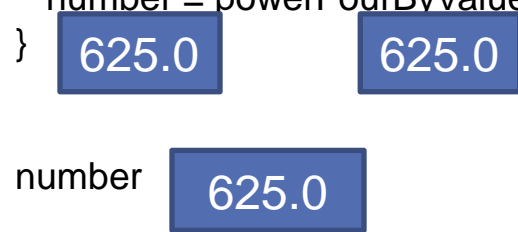
n

undefined

Analysis of Pass-by-value

Step 5 – after main completes the assignment

```
int main(void)
{
    double number = 5.0; // initialize number
    number = powerFourByValue(number);
}
```



```
double powerFourByValue(double n)
{
    return pow(n,4);
}
```

n

undefined

Analysis of Pass-by-reference

Step 1 – before calling powerFourByReference function

```
int main(void)
{
    double number = 5.0;
    powerFourByReference(&number);
}
```

number

5.0

```
Void powerFourByReference(double *nPtr)
{
    *nPtr = pow(*nPtr,4);
}
```

nPtr

undefined

Step 2 – after calling powerFourByValue function

```
int main(void)
{
    double number = 5.0;
    powerFourByReference(&number);
}
```

number

5.0

```
Void powerFourByReference(double *nPtr)
{
    *nPtr = pow(*nPtr,4);
}
```

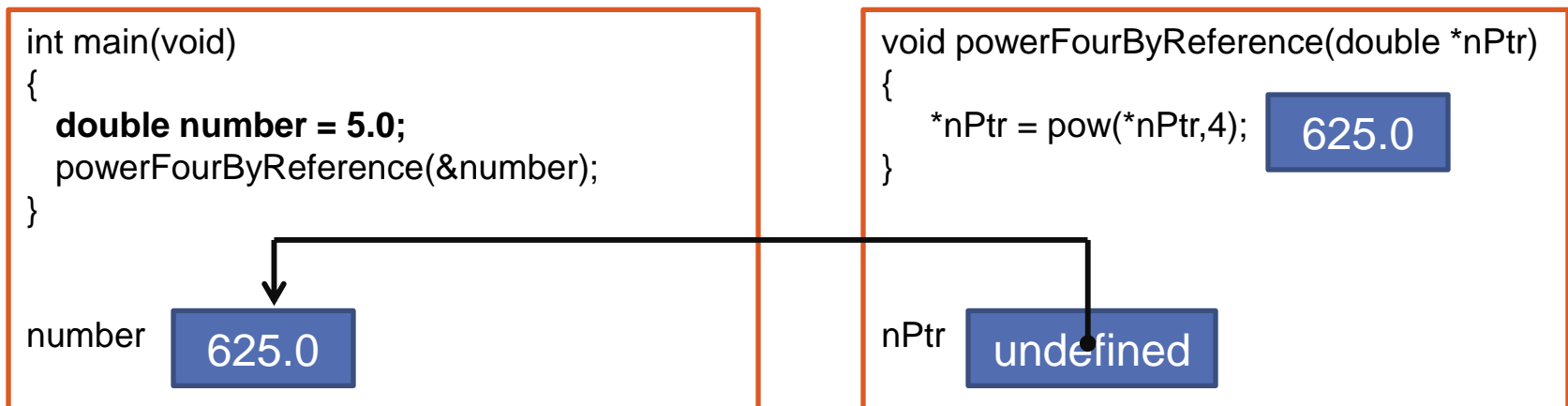
nPtr

•



Analysis of Pass-by-reference

Step 3 – after powerFourByReference computes power and *before returning to main (value is already changed)*



The following are some typical causes of a segmentation fault:

- **Call a function by value which waits for reference!!**
 - many compilers will give error while some of them causes segmentation fault
- Dereferencing null pointers – this is special-cased by memory management hardware
- Attempting to access a nonexistent memory address (outside process's address space)
- Attempting to access memory the program does not have rights to (such as kernel structures in process context)
- Attempting to write read-only memory (such as code segment)
- These in turn are often caused by programming errors that result in invalid memory access:
 - Dereferencing or assigning to an uninitialized pointer (wild pointer, which points to a random memory address)
 - Dereferencing or assigning to a freed pointer (dangling pointer, which points to memory that has been freed/deallocated/deleted)
 - A buffer overflow
 - A stack overflow
 - Attempting to execute a program that does not compile correctly.

Functions with Output Parameters

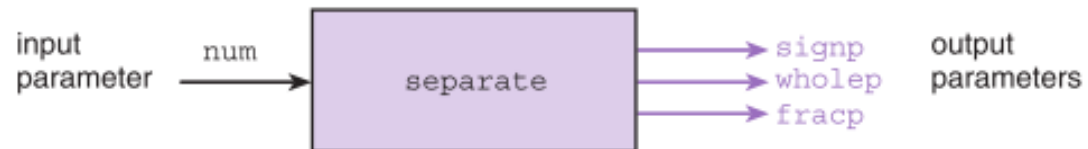
- We've used the return statement to send back one result value from a function.
- We can also use output parameters to return multiple results from a function.
- How??
 - We can use address of operator to store actual parameter's address which is a pointer instead of actual value. Therefore we can manipulate the content of memory address.

Functions with Output Parameters

- Lets write a function which finds the sign, whole number magnitude and fractional parts of its first parameter.
- `void separate (double num, char *signp, int *wholep, double *fracp)`

FIGURE 6.4

Diagram of
Function separate
with Multiple
Results



```
Enter a value to analyze> -345.56  
Parts of -345.5600  
  sign: -  
  whole number magnitude: 345  
  fractional part: 0.5600
```

```

27  /*
28  * Separates a number into three parts: a sign (+, -, or blank),
29  * a whole number magnitude, and a fractional part.
30  * Pre: num is defined; signp, wholep, and fracp contain addresses of memory
31  *       cells where results are to be stored
32  * Post: function results are stored in cells pointed to by signp, wholep, and
33  *       fracp
34  */
35  void separate(double num,      /* input - value to be split          */
36               char  *signp,   /* output - sign of num          */
37               int   *wholep,  /* output - whole number magnitude of num */
38               double *fracp)  /* output - fractional part of num */
39  {
40     double magnitude; /* local variable - magnitude of num */
41     /* Determines sign of num */
42     if (num < 0)
43         *signp = '-';
44     else if (num == 0)
45         *signp = ' ';
46     else
47         *signp = '+';
48
49     /* Finds magnitude of num (its absolute value) and separates it into
50        whole and fractional parts
51        magnitude = fabs(num);
52        *wholep = floor(magnitude);
53        *fracp = magnitude - *wholep;
54  }

```

indirect referencing

indirect referencing

How can we call such a function from our driver function?

```

1  #include <stdio.h>
2  #include <math.h>
3  void separate(double num, char *signp, int *wholep, double *fracp);
4
5  int main(void)
6  {
7      double value; /* input - number to analyze */
8      char sn;      /* output - sign of value */
9      int whl;      /* output - whole number magnitude of value */
10     double fr;     /* output - fractional part of value */
11
12     /* Gets data */
13     printf("Enter a value to analyze> ");
14     scanf("%lf", &value);
15
16     /* Separates data value into three parts */
17     separate(value, &sn, &whl, &fr);
18
19     /* Prints results */
20     printf("Parts of %.4f\n sign: %c\n", value, sn);
21     printf(" whole number magnitude: %d\n", whl);
22     printf(" fractional part: %.4f\n", fr);
23
24     return (0);
25 }

```

What if you forget putting address of operators while calling separate?

Parameter Correspondance for separate

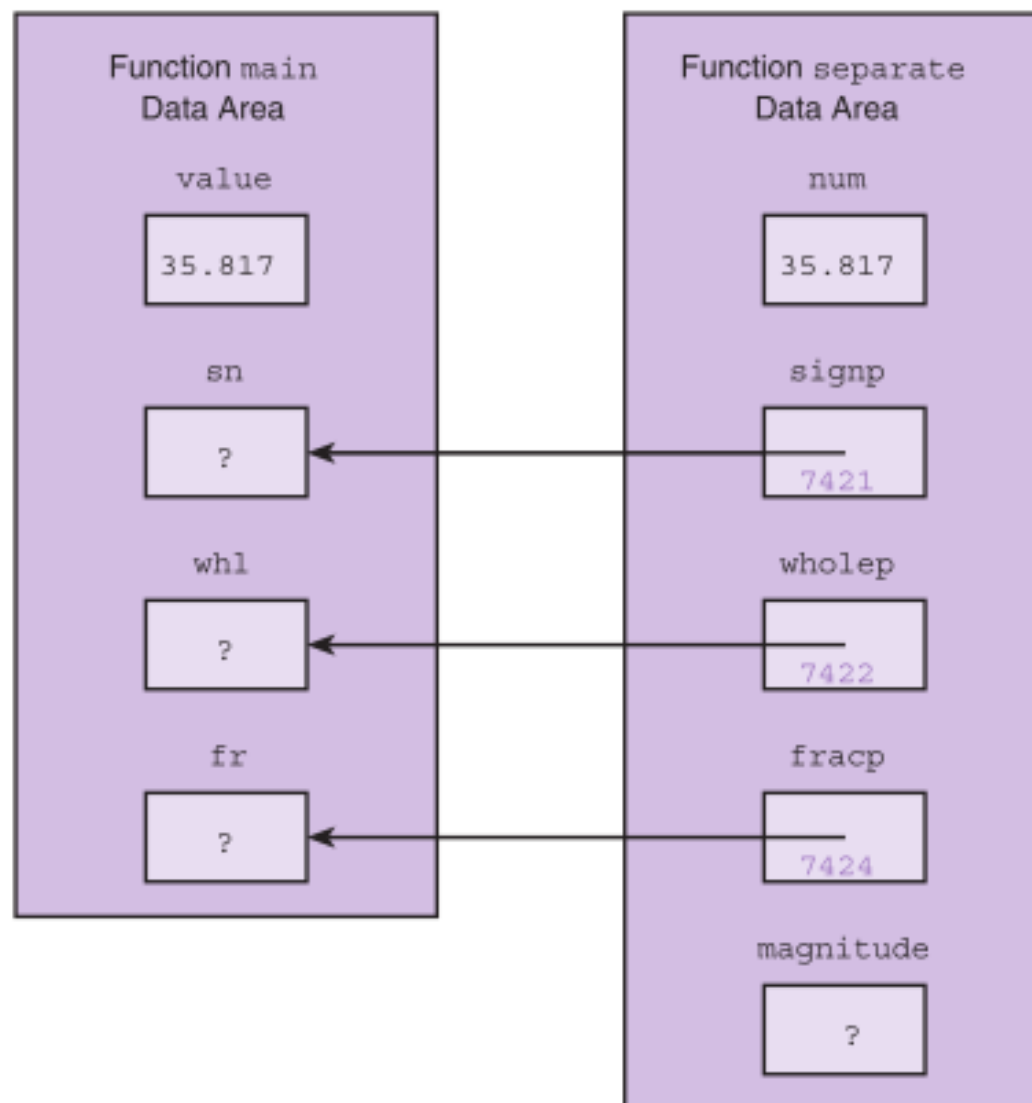
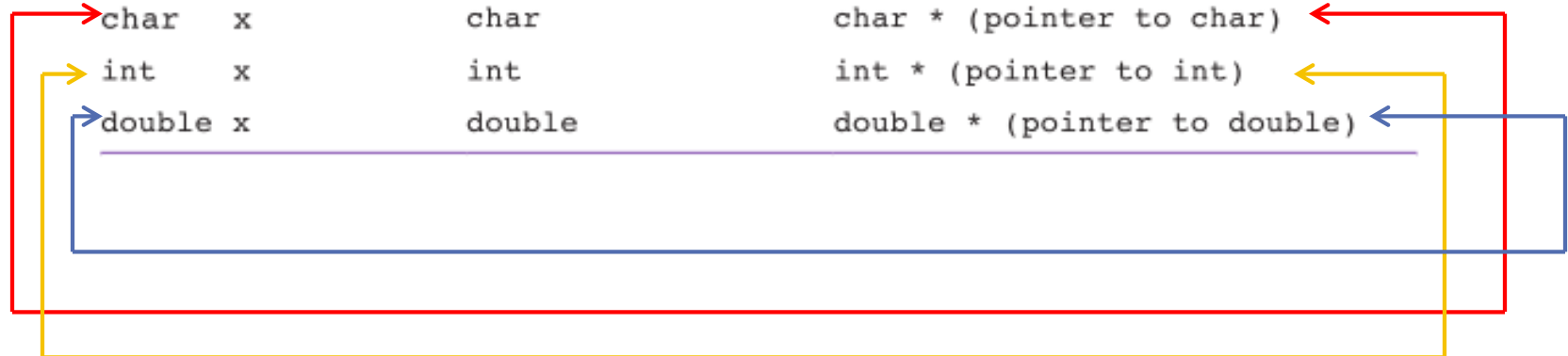


TABLE 6.2 Effect of & Operator on the Data Type of a Reference

Declaration	Data Type of x	Data Type of &x
char x	char	char * (pointer to char)
int x	int	int * (pointer to int)
double x	double	double * (pointer to double)



Types should match

Meaning of Symbol *

- binary operator for multiplication
- “pointer to” when used when declaring a function’s formal parameters
- unary indirection operator in a function body

HOA

- Write a function `sum_n_avg` that has three type double input parameters and two output parameters. The function computes sum and average of its three input arguments and relays its results through two output parameters.
- Complete the function call statement below:

```
int main (void)
{
    double one, two, three;
    double sum, avg;

    printf ("Enter 3 double numbers: ");
    scanf("%lf%lf%lf", &one, &two,&three);
    sum_n_avg(____);
    printf("Sum is %f, average is %f", sum, avg);

    return 0;
}
```

```
Enter 3 double numbers: 34.56 67.78 45
Sum is 147.340000, average is 49.113333
```

```
1  #include <stdio.h>
2
3  void sum_n_avg(double n1, double n2, double n3, double *sump, double *avgp);
4
5  int main (void)
6  {
7      double one, two, three;
8      double sum, avg;
9
10     printf ("Enter 3 double numbers: ");
11     scanf ("%lf%lf%lf", &one, &two, &three);
12     sum_n_avg(one, two, three, &sum, &avg);
13     printf ("Sum is %f, average is %f", sum, avg);
14
15     return 0;
16 }
17
18 void sum_n_avg(double n1, double n2, double n3, double *sump, double *avgp)
19 {
20     *sump = n1+n2+n3;
21     *avgp = *sump/3.0;
22 }
```

MULTIPLE CALLS TO A FUNCTION WITH INPUT/OUTPUT PARAMETERS

An example of sorting data

```
1  #include <stdio.h>
2
3  void order(double *smp, double *lgp);
4
5  int main(void)
6  {
7      double num1, num2, num3; /* three numbers to put in order */
8
9      /* Gets test data */
10     printf("Enter three numbers separated by blanks> ");
11     scanf("%lf%lf%lf", &num1, &num2, &num3);
12
13     /* Orders the three numbers */
14     order(&num1, &num2);
15     order(&num1, &num3);
16     order(&num2, &num3);
17
18     /* Displays results */
19     printf("The numbers in ascending order are: %.2f %.2f %.2f\n",
20           num1, num2, num3);
21
22     return (0);
23 }
```

```

25  /*
26  * Arranges arguments in ascending order.
27  * Pre:  smp and lgp are addresses of defined type double variables
28  * Post: variable pointed to by smp contains the smaller of the type
29  *       double values; variable pointed to by lgp contains the larger
30  */
31  void order(double *smp, double *lgp)    /* input/output */
32  {
33      double temp; /* temporary variable to hold one number during swap */
34      /* Compares values pointed to by smp and lgp and switches if necessary */
35      if (*smp > *lgp) {
36          temp = *smp;
37          *smp = *lgp;
38          *lgp = temp;
39      }
40  }
41  /*
42  Enter three numbers separated by blanks> 7.5 9.6 5.5
43  The numbers in ascending order are: 5.50 7.50 9.60
44  */

```

TABLE 6.3 Trace of Program to Sort Three Numbers

Statement	num1	num2	num3	Effect
<code>scanf("...", &num1, &num2, &num3);</code>	7.5	9.6	5.5	Enters data
<code>order(&num1, &num2);</code>				No change
<code>order(&num1, &num3);</code>	5.5	9.6	7.5	Switches num1 and num3
<code>order(&num2, &num3);</code>	5.5	7.5	9.6	Switches num2 and num3
<code>printf("...", num1, num2, num3);</code>				Displays 5.5 7.5 9.6

FIGURE 6.8

Data Areas After
`temp = *smp;`
 During Call
`order(&num1,
 &num3);`

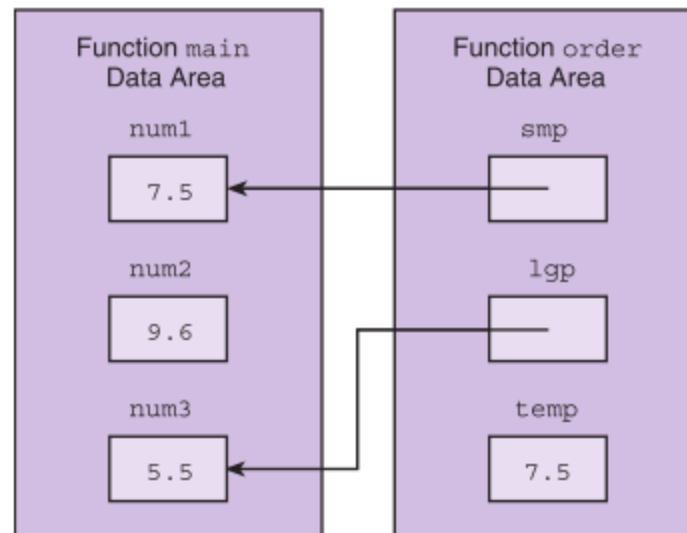


TABLE 6.4 Different Kinds of Function Subprograms

Purpose	Function Type	Parameters	To Return Result
To compute or obtain as input a single numeric or character value.	Same as type of value to be computed or obtained.	Input parameters hold copies of data provided by calling function.	Function code includes a return statement with an expression whose value is the result.
To produce printed output containing values of numeric or character arguments.	void	Input parameters hold copies of data provided by calling function.	No result is returned.
To compute multiple numeric or character results.	void	Input parameters hold copies of data provided by calling function. Output parameters are pointers to actual arguments.	Results are stored in the calling function's data area by indirect assignment through output parameters. No return statement is required.
To modify argument values.	void	Input/output parameters are pointers to actual arguments. Input data is accessed by indirect reference through parameters.	Results are stored in the calling function's data area by indirect assignment through output parameters. No return statement is required.

Pointers to Files

- C allows a program to explicitly name a file for input or output.

- Declare file pointers:


- `FILE *inp; /* pointer to input file */`
- `FILE *outp; /* pointer to output file */`

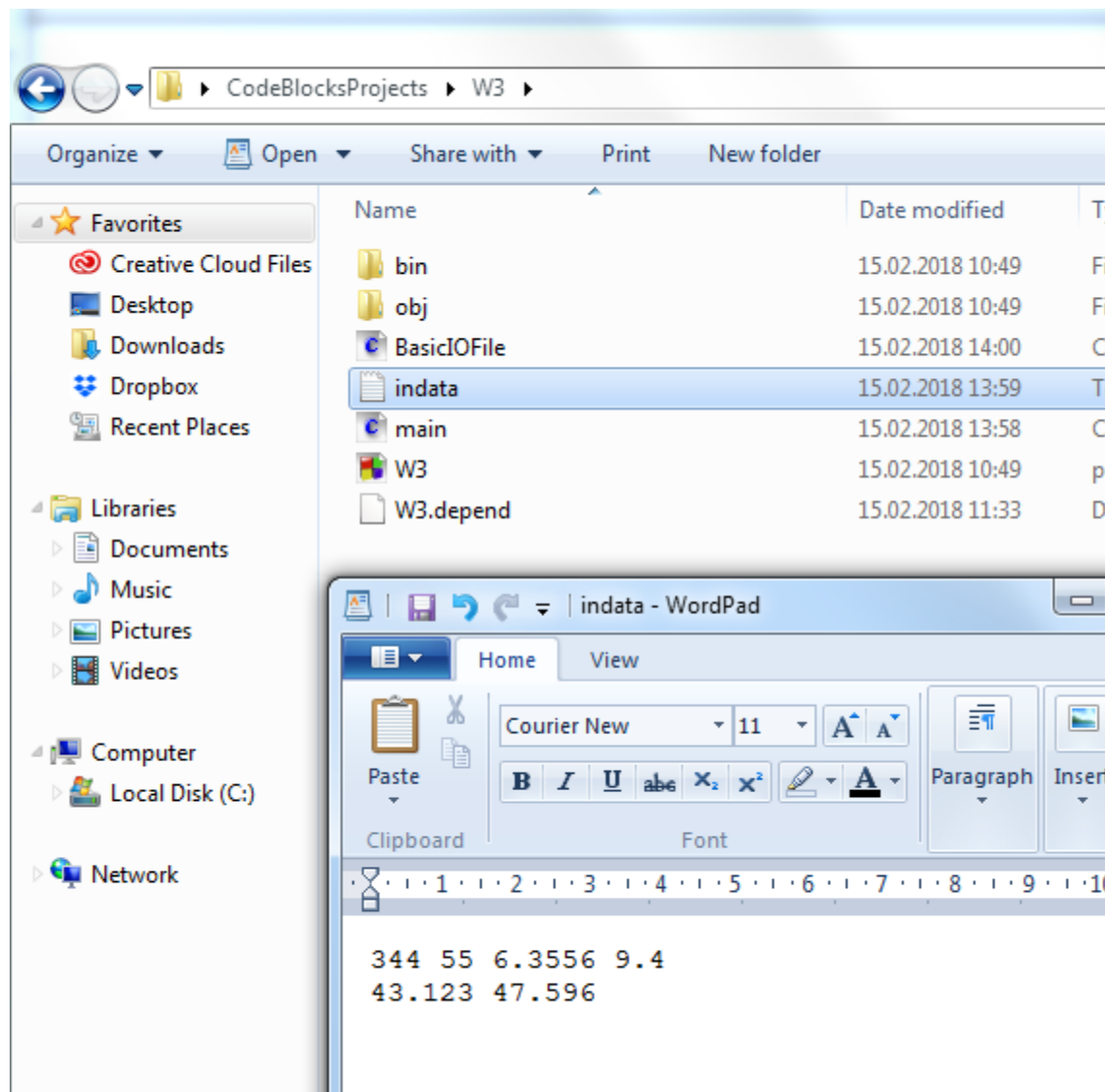
- Prepare for input or output before permitting access:

- `inp = fopen("infile.txt", "r");`
- `outp = fopen("outfile.txt", "w");`

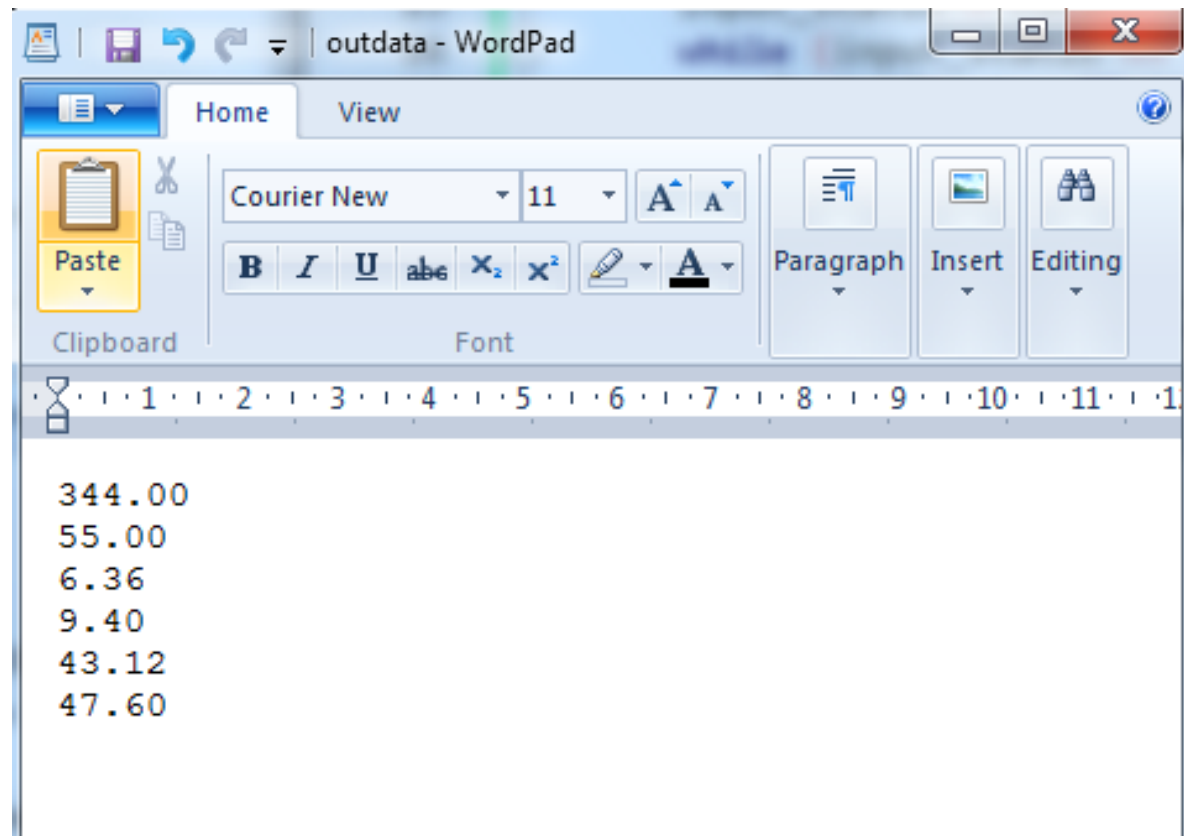
- infile is the source of input
- OS stores the necessary access value in the file pointer variable inp
- r is read
- outfile is where we write
- OS stores the necessary access value in the file pointer variable outp.
- w is write

Pointers to Files

- `fscanf`
 - file equivalent of `scanf`
 - `fscanf(inp, "%lf", &item);`
- `fprintf`
 - file equivalent of `printf`
 - `fprintf(outp, "%.2f\n", item);`
- closing a file when done 
 - `fclose(inp);`
 - `fclose(outp);`



```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      FILE *inp;          /* pointer to input file */
6      FILE *outp;         /* pointer to output file */
7      double item;
8      int input_status;   /* status value returned by fscanf */
9
10     /* Prepare files for input or output */
11     inp = fopen("indata.txt", "r");
12     outp = fopen("outdata.txt", "w");
13
14     /* Input each item, format it, and write it */
15     input_status = fscanf(inp, "%lf", &item);
16     while (input_status == 1) {
17         fprintf(outp, "%.2f\n", item);
18         input_status = fscanf(inp, "%lf", &item);
19     }
20
21     /* Close the files */
22     fclose(inp);
23     fclose(outp);
24
25     return (0);
26 }
```



File access mode string	Meaning	Explanation	Action if file already exists	Action if file does not exist
"r"	read	Open a file for reading	read from start	failure to open
"w"	write	Create a file for writing	destroy contents	create new
"a"	append	Append to a file	write to end	create new
"r+"	read extended	Open a file for read/write	read from start	error
"w+"	write extended	Create a file for read/write	destroy contents	create new
"a+"	append extended	Open a file for read/write	write to end	create new

[2]

r+ = read/write mode – used for update

doesn't delete the content of the file and doesn't create a new file if such file doesn't exist

the new stream is positioned at the beginning of the file

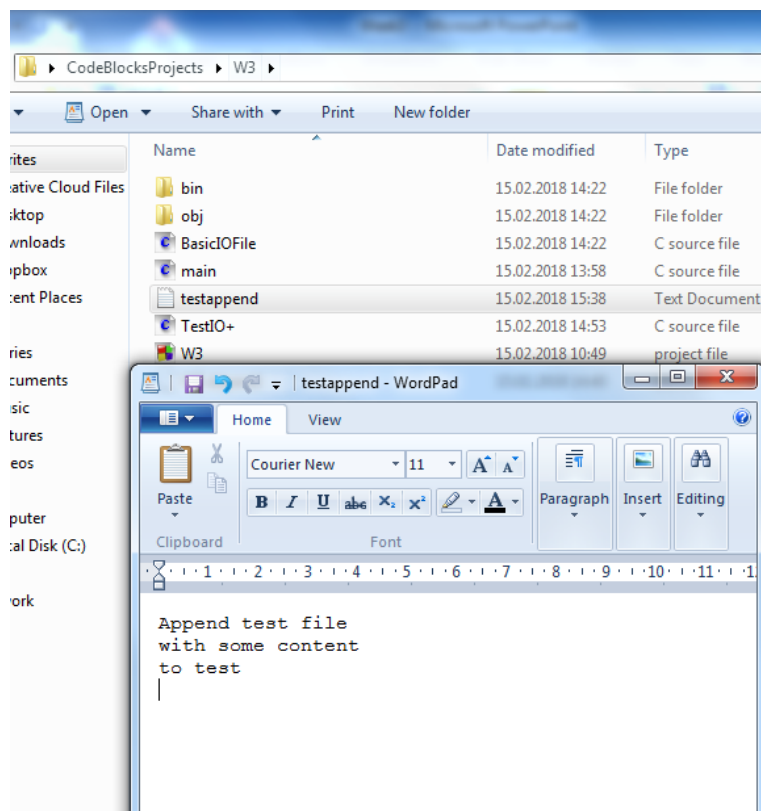
w+ = read/write mode – used for update

deletes the content of the file and creates it if it doesn't exist

a+ = Open a file for reading and appending. All writing operations are performed at the end of the file, protecting the previous content to be overwritten. The file is created if it does not exist.

append - a

- Assume that we have a file called testappend.txt and its content is:

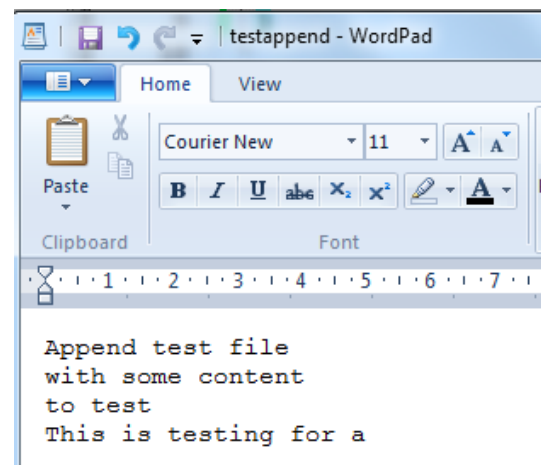


Now run the below code:

```

1  #include <stdio.h>
2  int main()
3  {
4      FILE *fp;
5
6      fp = fopen("testappend.txt", "a");
7      fprintf(fp, "This is testing for a\n");
8      fclose(fp);
9  }

```



r+

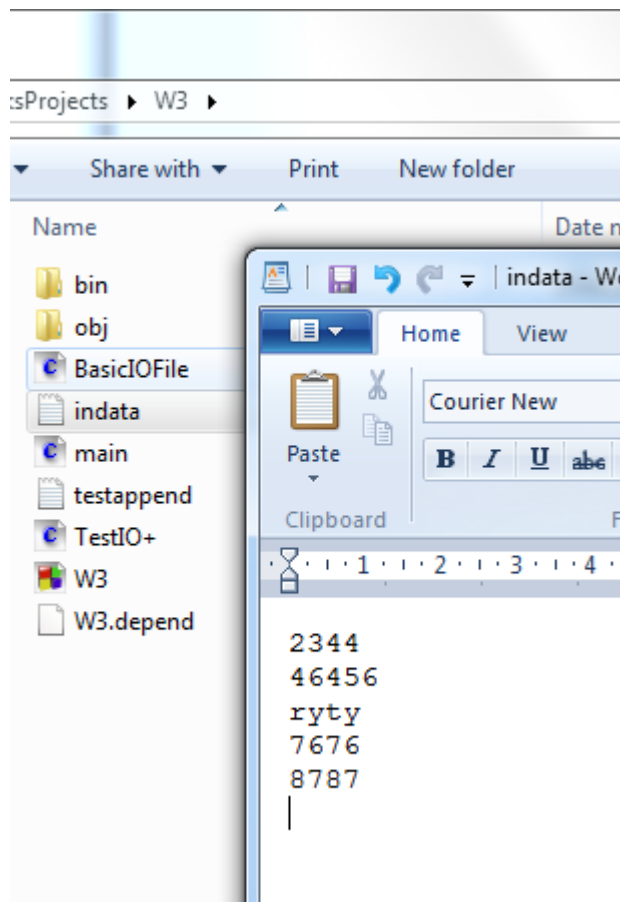
- Assume that we do not have any file named indata.txt beforehand
- Now run the below code:

```
1  #include <stdio.h>
2  int main()
3  {
4      FILE *fp;
5
6      fp = fopen("indata.txt", "r+");
7      fprintf(fp, "This is testing for r+\n");
8      fclose(fp);
9  }
```

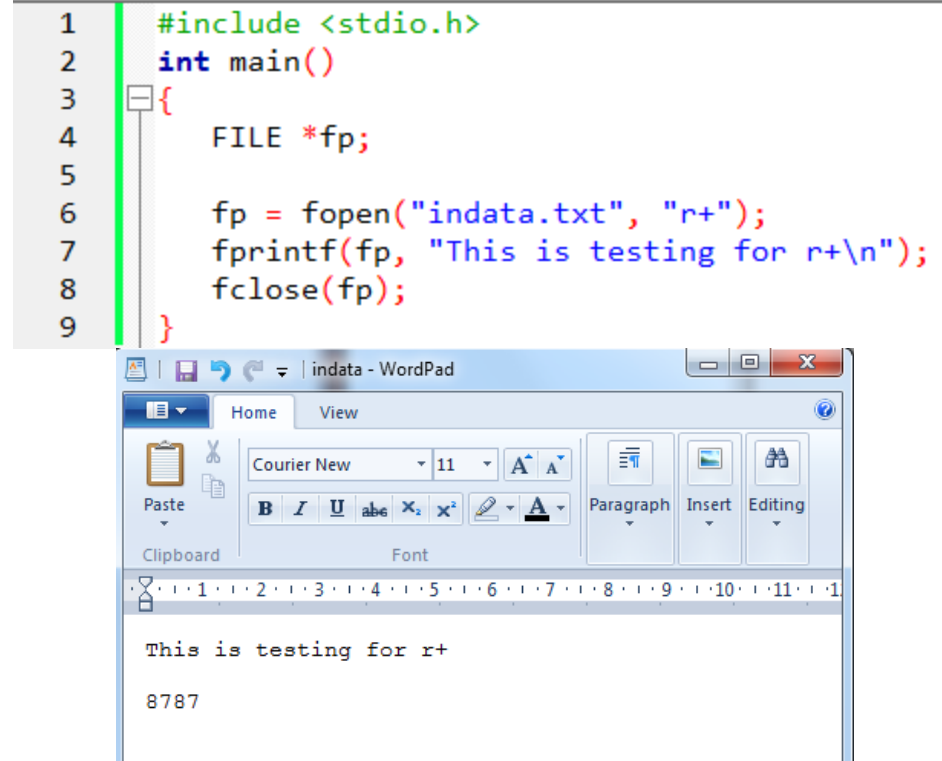
- No file will be created

r+

- Now create indata.txt and write some lines inside it, e.g.



Now run the code again:

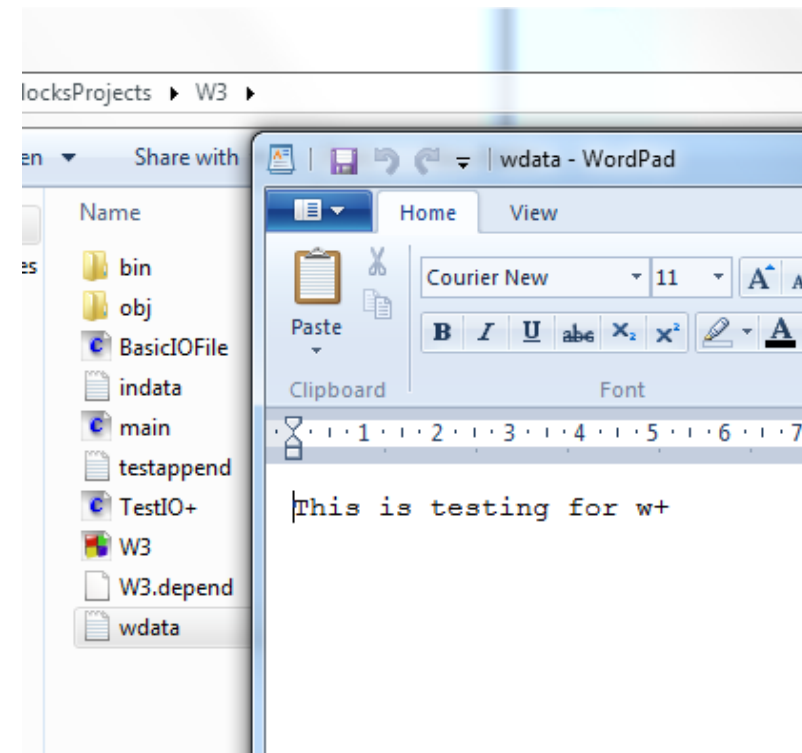


W+

- Do not create any file
- Run the below code

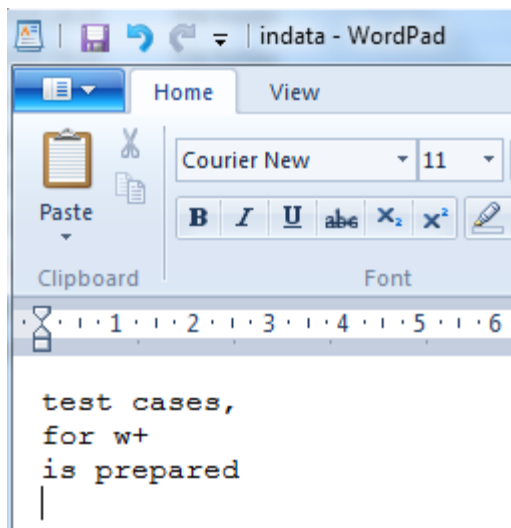
```
1  #include <stdio.h>
2  int main()
3  {
4      FILE *fp;
5
6      fp = fopen("wdata.txt", "w+");
7      fprintf(fp, "This is testing for w+\n");
8      fclose(fp);
9  }
```

- File is created with given content



W+

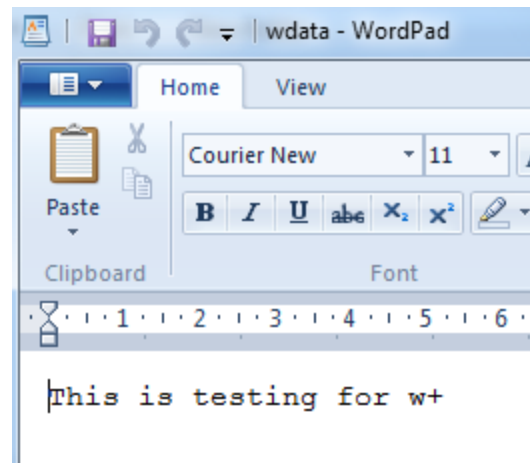
- Now create a file called wdata.txt and put some content inside, e.g.



```

1  #include <stdio.h>
2  int main()
3  {
4      FILE *fp;
5
6      fp = fopen("wdata.txt", "w+");
7      fprintf(fp, "This is testing for w+\n");
8      fclose(fp);
9  }

```



Overwrites
everything

Scope of Names

- The scope of a name is the region in a program where a particular meaning of a name is visible.

```
1  | #define MAX 950
2  | #define LIMIT 200
3  |
4  | void one(int anarg, double second);
5  | int fun_two(int one, char anarg);
6  |
7  | int main(void)
8  | {
9  |     int localvar;
10 | }
11 |
12 | void one(int anarg, double second)
13 | {
14 |     int onelocal;
15 | }
16 |
17 | int fun_two(int one, char anarg)
18 | {
19 |     int localvar;
20 |
21 | }
```


TABLE 6.5 Scope of Names in Fig. 6.9

Name	Visible in one	Visible in fun_two	Visible in main
MAX	yes	yes	yes
LIMIT	yes	yes	yes
main	yes	yes	yes
localvar (in main)	no	no	yes
one (the function)	yes	no	yes
anarg (int)	yes	no	no
second	yes	no	no
onelocal	yes	no	no
fun_two	yes	yes	yes
one (formal parameter)	no	yes	no
anarg (char)	no	yes	no
localvar (in fun_two)	no	yes	no

```

1  #define MAX 950
2  #define LIMIT 200
3
4  void one(int anarg, double second);
5  int fun_two(int one, char anarg);
6
7  int main(void)
8  {
9      int localvar;
10 }
11
12 void one(int anarg, double second)
13 {
14     fun_two(5, 'a');
15     int onelocal;
16 }
17
18 int fun_two(int one, char anarg)
19 {
20     int localvar;
21
22 }


```



```

1  #define MAX 950
2  #define LIMIT 200
3
4  void one(int anarg, double second);
5  int fun_two(int one, char anarg);
6
7  int main(void)
8  {
9      int localvar;
10 }
11
12 void one(int anarg, double second)
13 {
14     int onelocal;
15 }
16
17 int fun_two(int one, char anarg)
18 {
19     one(2,3);
20     int localvar;
21
22 }

```



others

Line	Message
=== Build: Release in W3 (compiler: GNU GCC Compiler) ===	
	In function 'main':
9	warning: unused variable 'localvar' [-Wunused-variable]
	In function 'one':
14	warning: unused variable 'onelocal' [-Wunused-variable]
	In function 'fun_two':
19	error: called object 'one' is not a function or function pointer
17	note: declared here
20	warning: unused variable 'localvar' [-Wunused-variable]
22	warning: control reaches end of non-void function [-Wreturn-type]
=== Build failed: 1 error(s), 4 warning(s) (0 minute(s), 0 second)	

Wrap Up

- a program can declare pointers to variables of a specified type
- C allows a program to explicitly name a file for input or output
- parameters enable a programmer to pass data to functions and to return multiple results from functions
- a function can use parameters declared as pointers to return values
- the scope of an identifier dictates where it can be referenced

References

1. Problem Solving & Program Design in C, Jeri R. Hanly & Elliot B. Koffman, Pearson 8. Edition, Global Edition
2. <http://en.cppreference.com/w/cpp/io/c/fopen>

ARRAY POINTERS

CHAPTER 7

Problem Solving & Program Design in C

Eighth Edition

Global Edition

Jeri R. Hanly & Elliot B. Koffman

Chapter Objectives

- To learn how to declare and use arrays for storing collections of values of the same type
- **To understand that C implements arrays as pointers**
- To understand how to use a subscript to reference the individual values in an array
- To learn how to process the elements of an array in sequential order using loops

Chapter Objectives

- To understand how to pass individual array elements and entire arrays through function arguments
- To learn a method for searching an array
- To learn a method for sorting an array
- To learn how to use multidimensional arrays for storing tables of data

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*

<i>x</i> [0]	<i>x</i> [1]	<i>x</i> [2]	<i>x</i> [3]	<i>x</i> [4]	<i>x</i> [5]	<i>x</i> [6]	<i>x</i> [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
<code>printf("%.1f", x[0]);</code>	Displays the value of <i>x</i> [0], which is 16.0.
<code>x[3] = 25.0;</code>	Stores the value 25.0 in <i>x</i> [3].
<code>sum = x[0] + x[1];</code>	Stores the sum of <i>x</i> [0] and <i>x</i> [1], which is 28.0 in the variable <i>sum</i> .
<code>sum += x[2];</code>	Adds <i>x</i> [2] to <i>sum</i> . The new <i>sum</i> is 34.0.
<code>x[3] += 1.0;</code>	Adds 1.0 to <i>x</i> [3]. The new <i>x</i> [3] is 26.0.
<code>x[2] = x[0] + x[1];</code>	Stores the sum of <i>x</i> [0] and <i>x</i> [1] in <i>x</i> [2]. The new <i>x</i> [2] is 28.0.

Array *x*

<i>x</i> [0]	<i>x</i> [1]	<i>x</i> [2]	<i>x</i> [3]	<i>x</i> [4]	<i>x</i> [5]	<i>x</i> [6]	<i>x</i> [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:

$x[3]$

$x[i + 1]$

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 7.2 Code Fragment That Manipulates Array *x*

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

Array *x*

<code>x[0]</code>	<code>x[1]</code>	<code>x[2]</code>	<code>x[3]</code>	<code>x[4]</code>	<code>x[5]</code>	<code>x[6]</code>	<code>x[7]</code>
16.0	12.0	6.0	8.0	2.5	12.0	14.0	-54.5

Using for Loops for Sequential Access

```
for (int 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

Arrays and Memory

- When we declare an array

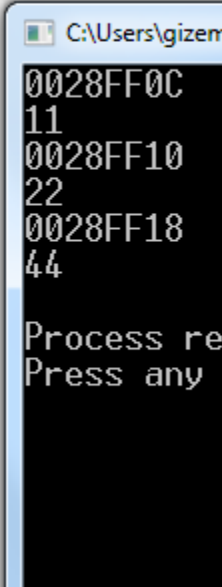
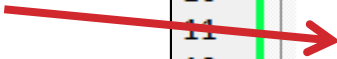
int arr[size]

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

```
4  int main(void)
5  {
6      int v[5] = {11,22,33,44,55};
7      int* vPtr = v; //same as vPtr = &v[0]
8      printf("%p\n", vPtr);
9      printf("%d\n", *vPtr);
10
11     //not incremented as in conventional arithmetic
12     //points to next location in the array
13     vPtr++;
14
15     printf("%p\n", vPtr);
16     printf("%d\n", *vPtr);
17
18     vPtr += 2;
19     printf("%p\n", vPtr);
20     printf("%d\n", *vPtr);
21 }
```



0028FF0C
11
0028FF10
22
0028FF18
44
Process re
Press any

Pointers as Array Iterators

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

```
1  #include <stdio.h>
2
3  int main (void)
4  {
5      printf("\n");
6      int a[5] = {0,2,5,8,11};
7      int *ptr = &a[0];
8      for (int i = 0; i < 5; i++)
9      {
10         printf("  %p: %d\n", ptr, *ptr);
11         ptr++;
12     }
13
14     return 0;
15 }
```

```
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 }
```

```
1  #include <stdio.h>
2
3  void staticArrayInit(void); // function prototype
4  void automaticArrayInit(void); // function prototype
5
6  // function main begins program execution
7  int main(void)
8  {
9      puts("First call to each function:");
10     staticArrayInit();
11     automaticArrayInit();
12
13     puts("\n\nSecond call to each function:");
14     staticArrayInit();
15     automaticArrayInit();
16 }
```

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

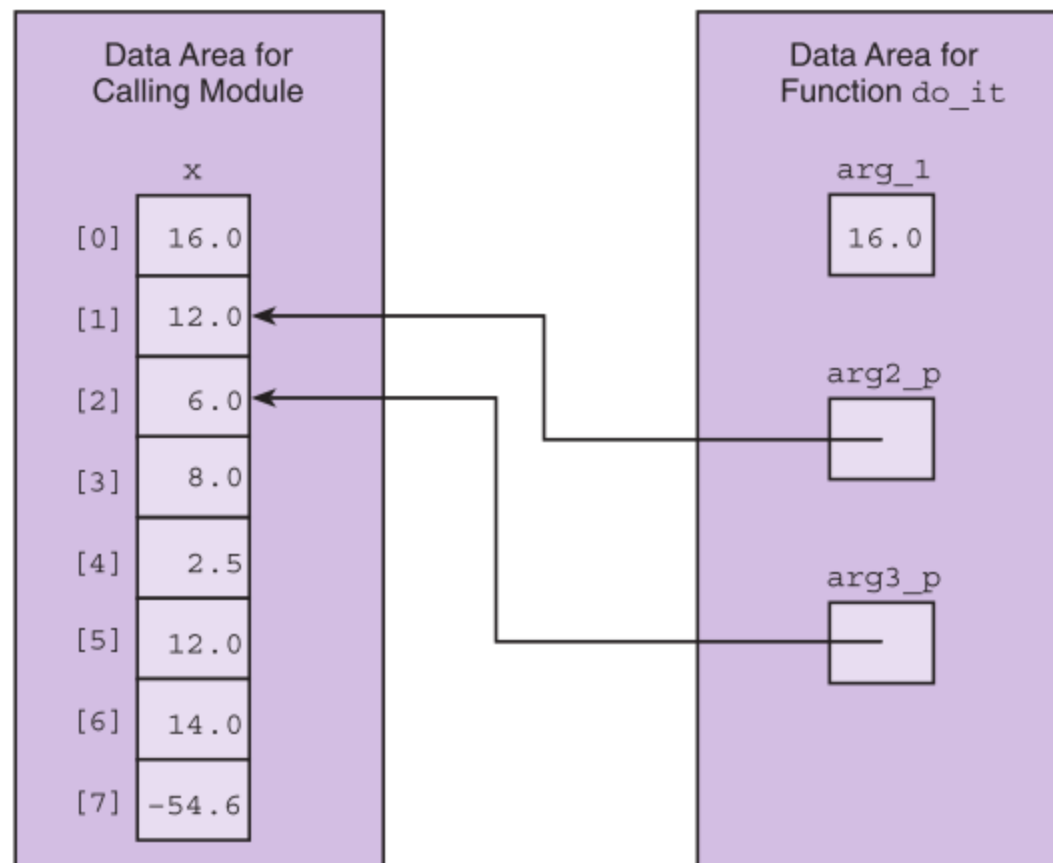
Values on exiting automaticArrayInit:
array2[0] = 6 array2[1] = 7 array2[2] = 8

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.

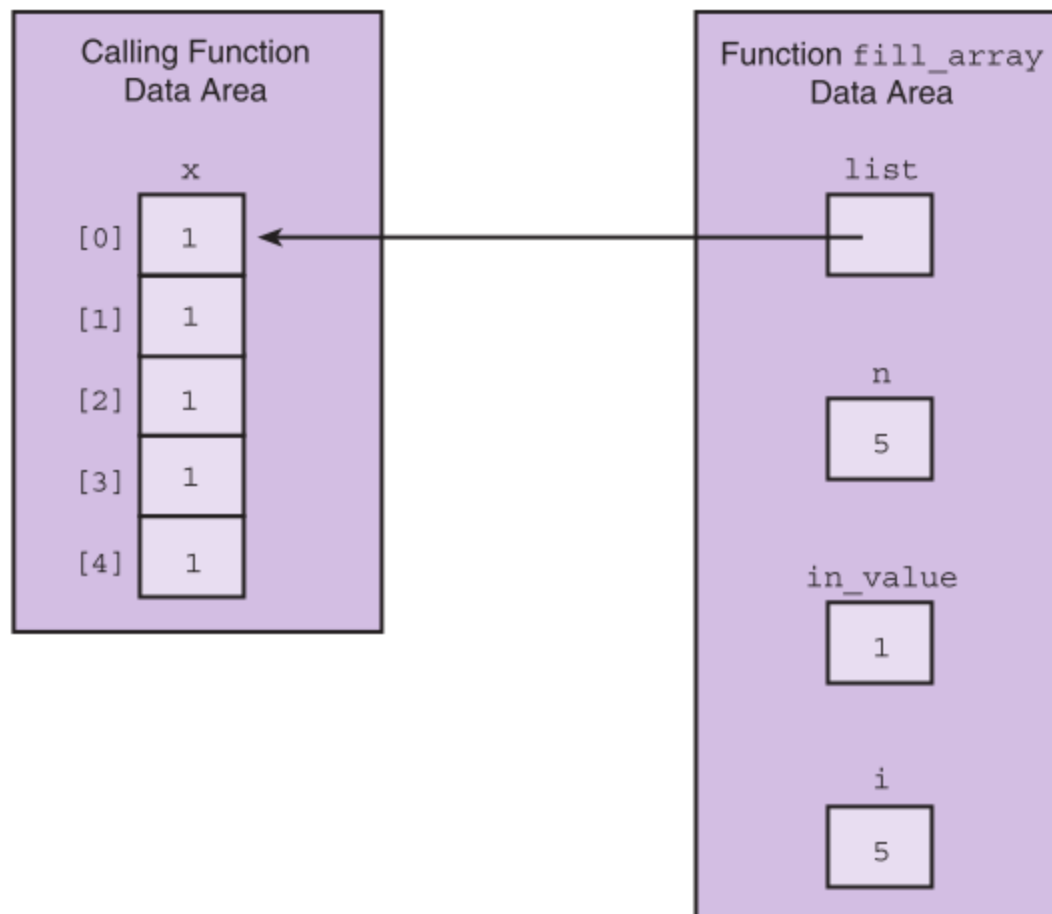

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int arr[5];
6      fill_array(arr,5,3);
7      for(int i = 0; i < 5; i++)
8          printf("Element %d is: %d\n",i, arr[i]);
9  }
10 void fill_array (int list[], int n, int in_value)
11 {
12     int i;
13
14     for (i = 0; i < n; ++i)
15         list[i] = in_value;
16 }
```

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);
```

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

The same



```
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

```

4  #include <stdio.h>
5
6  int main(void)
7  {
8      char c;
9      short s;
10     int i;
11     long l;
12     long long ll;
13     float f;
14     double d;
15     long double ld;
16     int array[20]; // create array of 20 int elements
17     int *ptr = array; // create pointer to array
18
19     printf("    sizeof c = %u\tsizeof(char) = %u"
20           "\n    sizeof s = %u\tsizeof(short) = %u"
21           "\n    sizeof i = %u\tsizeof(int) = %u"
22           "\n    sizeof l = %u\tsizeof(long) = %u"
23           "\n    sizeof ll = %u\tsizeof(long long) = %u"
24           "\n    sizeof f = %u\tsizeof(float) = %u"
25           "\n    sizeof d = %u\tsizeof(double) = %u"
26           "\n    sizeof ld = %u\tsizeof(long double) = %u"
27           "\n    sizeof array = %u"
28           "\n    sizeof ptr = %u\n",
29           sizeof c, sizeof(char), sizeof s, sizeof(short), sizeof i,
30           sizeof(int), sizeof l, sizeof(long), sizeof ll,
31           sizeof(long long), sizeof f, sizeof(float), sizeof d,
32           sizeof(double), sizeof ld, sizeof(long double),
33           sizeof array, sizeof ptr);
34 }


```

```
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.


```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int largest;
6      int arr[] = {34, 56, 12, 4, 89, 45, 78, 21};
7      get_max(arr, 8, &largest);
8      printf("Largest is: %d", largest);
9  }
10
11
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      *largest = cur_large;
27  }
```

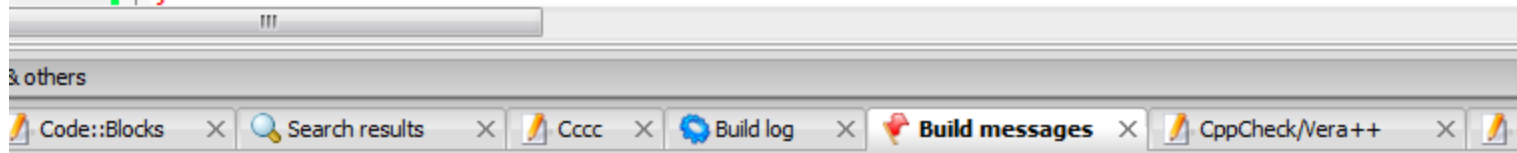


Edit Attempt to Const Array

```

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 }

```



.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:

7		<code>int x, y;</code>
8		<code>int* const ptr = &x;</code>
9		<code>*ptr = 7;</code>
10	■	<code>ptr = &y;</code>

Allowed, *ptr is not constant

Not allowed, ptr is constant
«error: assignment of read-only variable 'ptr'»

const pointer

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

```
p1 = &b;
```

```
p2 = p1;
```

→ P2 and p1 points to the same location

```
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;
```

```
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:


7		<code>int x = 10;</code>
8		<code>const int* p = &x;</code>
9	■	<code>*p = 5;</code>

Not allowed, *p is constant
«error: assignment of read-only location 'p'»

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’

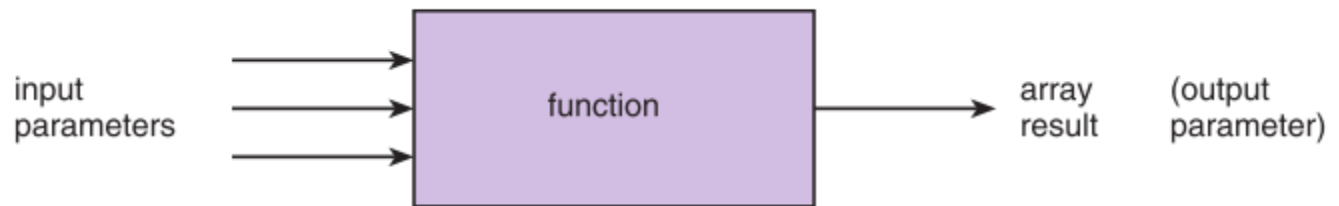
error: assignment of read-only variable ‘ptr’

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

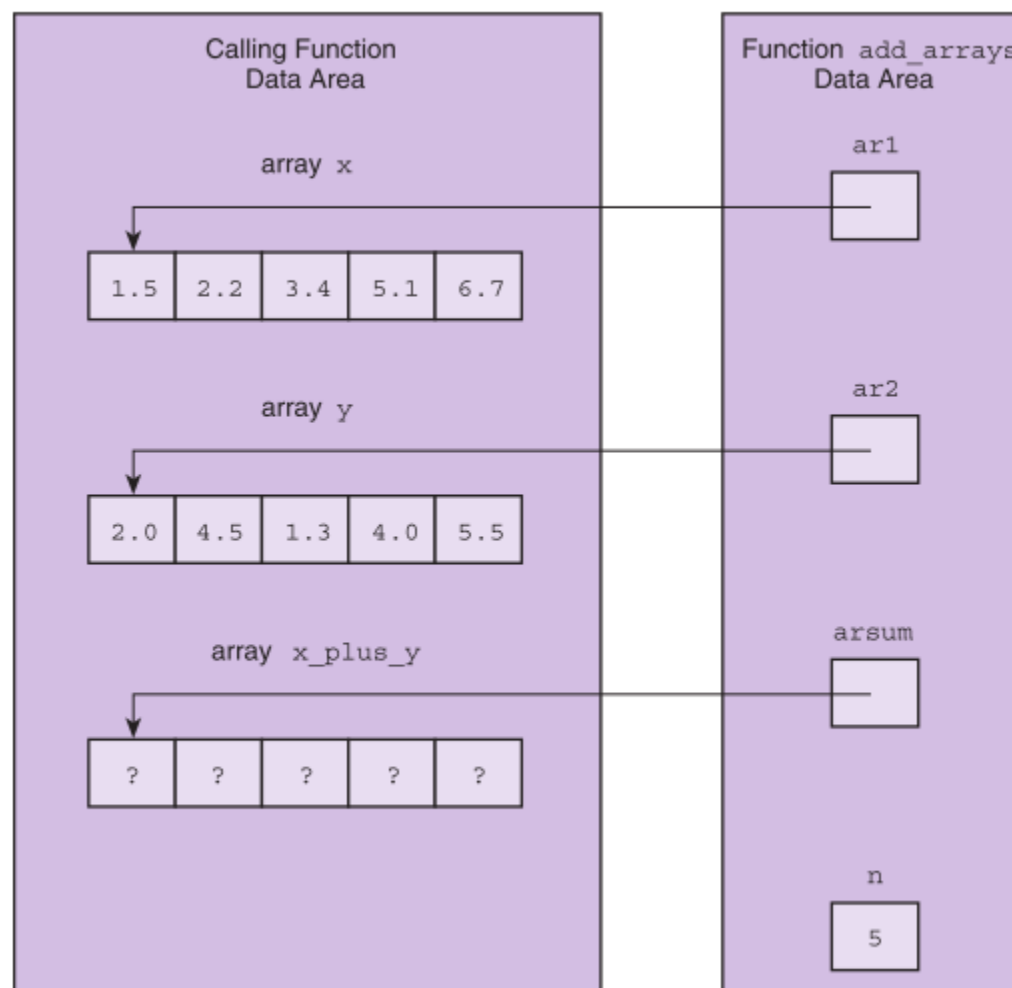



```
1  #include<stdio.h>
2
3  int main(void)
4  {
5      double x[5]={1.5,2.2,3.4,5.1,6.7};
6      double y[5]={2.0,4.5,1.3,4.0,5.5};
7      double x_plus_y[5];
8      add_arrays(x,y,x_plus_y,5);
9      for (int i = 0; i < 5; ++i)
10         printf("%.2f\n",x_plus_y[i]);
11 }
12
13 void add_arrays(const double ar1[], const double ar2[],
14                double arsum[], int n)
15
16 {
17     for (int i = 0; i < n; ++i)
18         arsum[i] = ar1[i] + ar2[i];
19 }
20
```

No address 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.

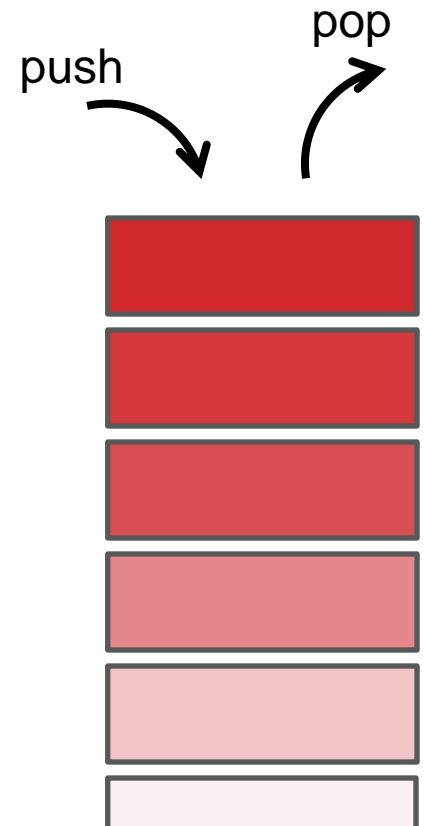
→ The name of an array does not hold extra space in memory.

```
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.

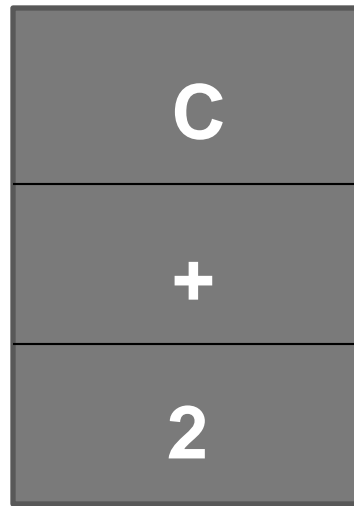
Stack

- 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

```

1  #include <stdio.h>
2  #define MAXSIZE 10
3  #define STACK_EMPTY 0
4
5  void push(char stack[], char item, int *top, int max_size);
6  char pop(char stack[], int *top);
7  char retrieve(const char stack[], int top);
8  void print_char_stack(const char stack[], int s_top);
9
10 int main(void)
11 {
12     char stack_s[MAXSIZE];
13     int s_top = -1; // subscript of the element at top of the stack
14     char ch1, ch2;
15
16     push(stack_s, '2', &s_top, MAXSIZE);
17     push(stack_s, '+', &s_top, MAXSIZE);
18     push(stack_s, 'C', &s_top, MAXSIZE);
19
20     printf("Before pop, stack looks like:\n");
21     print_char_stack(stack_s, s_top);
22
23     ch1 = pop(stack_s, &s_top);
24     printf("\nAfter pop, stack looks like:\n");
25     print_char_stack(stack_s, s_top);
26     printf("\nch1 is: %c\n", ch1);
27
28     ch2 = retrieve(stack_s, s_top);
29     printf("\nch2 is: %c\n", ch2);
30     printf("\nAfter retrieve, stack looks like:\n");
31     print_char_stack(stack_s, s_top);
32
33     return 0;
34 }

```

```
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.

```

1 | #define NOT_FOUND -1
2 |
3 | /*
4 |  * Searches for target item in first n elements of array arr
5 |  * Returns index of target or NOT_FOUND
6 |  * Pre: target and first n elements of array arr are defined and n>=0
7 |  */
8 | int search(const int arr[], /* input - array to search */
9 |           int target, /* input - value searched for */
10 |          int n) /* input - number of elements to search */
11 | {
12 |     int i,
13 |         found = 0, /* whether or not target has been found */
14 |         where; /* index where target found or NOT_FOUND */
15 |
16 |     /* Compares each element to target*/
17 |     i = 0;
18 |     while (!found && i < n) {
19 |         if (arr[i] == target)
20 |             found = 1;
21 |         else
22 |             ++i;
23 |     }
24 |
25 |     /* Returns index of element matching target or NOT_FOUND */
26 |     if (found)
27 |         where = i;
28 |     else
29 |         where = NOT_FOUND;
30 |
31 |     return (where);
32 | }

```

Bubble Sort – 1

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

Example:

First Pass:

(**5** 1 4 2 8) \rightarrow (**1** **5** 4 2 8), Here, algorithm compares the first two elements, and swaps since $5 > 1$.
(1 **5** 4 2 8) \rightarrow (1 **4** **5** 2 8), Swap since $5 > 4$
(1 4 **5** 2 8) \rightarrow (1 4 **2** **5** 8), Swap since $5 > 2$
(1 4 2 **5** **8**) \rightarrow (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) \rightarrow (**1** **4** 2 5 8)
(1 **4** **2** 5 8) \rightarrow (1 **2** **4** 5 8), Swap since $4 > 2$
(1 2 **4** **5** 8) \rightarrow (1 2 **4** **5** 8)
(1 2 4 **5** **8**) \rightarrow (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) \rightarrow (**1** **2** 4 5 8)
(1 **2** **4** 5 8) \rightarrow (1 **2** **4** 5 8)
(1 2 **4** **5** 8) \rightarrow (1 2 **4** **5** 8)
(1 2 4 **5** **8**) \rightarrow (1 2 4 **5** **8**)

How do you implement bubble sort?

```
1  #include <stdio.h>
2  #define SIZE 10
3
4  int main(void)
5  {
6      // initialize a
7      int a[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};
8
9      puts("Data items in original order");
10
11     // output original array
12     for (size_t i = 0; i < SIZE; ++i)
13         printf("%4d", a[i]);
14
15     for (int pass = 1; pass < SIZE; ++pass)
16     {
17         for (int i = 0; i < SIZE - 1; ++i)
18         {
19             if (a[i] > a[i + 1])
20             {
21                 int hold = a[i];
22                 a[i] = a[i + 1];
23                 a[i + 1] = hold;
24             }
25         }
26     }
27
28     puts("\nData items in ascending order");
29
30     // output sorted array
31     for (size_t i = 0; i < SIZE; ++i)
32         printf("%4d", a[i]);
33
34     return 0;
35 }
```

```
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)

```
1  #include <stdio.h>
2  #define SIZE 10
3
4  void bubble_sort(int * const array, const size_t size);
5  void swap(int *element1Ptr, int *element2Ptr);
6
7  int main(void)
8  {
9      int a[SIZE] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
10
11     puts("Data items in original order");
12
13     for (size_t i = 0; i < SIZE; ++i)
14         printf("%4d", a[i]);
15
16     bubble_sort(a, SIZE);
17
18     puts("\nData items in ascending order");
19
20     // loop through array a
21     for (size_t i = 0; i < SIZE; ++i)
22         printf("%4d", a[i]);
23
24     return 1;
25 }
```



```
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`.

[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

```

1  int get_min_range(int list[], int first, int last);
2
3  void select_sort(int list[], int n)
4  {
5      int fill,
6          temp,
7          index_of_min;
8
9      for (fill = 0; fill < n-1; ++fill)
10     {
11         /* Find position of smallest element in unsorted subarray */
12         index_of_min = get_min_range(list, fill, n-1);
13
14         /* Exchange elements at fill and index_of_min */
15         if (fill != index_of_min)
16         {
17             temp = list[index_of_min];
18             list[index_of_min] = list[fill];
19             list[fill] = temp;
20         }
21     }
22 }
23
24 int get_min_range(int list[], int first, int last)
25 {
26     int i,          /* Loop Control Variable (LCV)          */
27     small_sub;      /* subscript of smallest value so far */
28
29     small_sub = first; /* Assume first element is smallest */
30
31     for (i = first + 1; i <= last; ++i)
32         if (list[i] < list[small_sub])
33             small_sub = i;
34
35     return (small_sub);
36 }

```

Introduction to Multidimensional Arrays

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

		Column		
		0	1	2
Row	0	X	O	X
	1	O	X	O ← tictac[1][2]
	2	O	X	X

row_id column_id

The diagram illustrates the indexing of a 2D array. A 3x3 grid represents the array 'tictac'. The rows are indexed 0, 1, and 2, and the columns are indexed 0, 1, and 2. The cell at row 1, column 2 contains the character 'O'. An arrow points from the text 'tictac[1][2]' to this cell. Below the text, two arrows point upwards: one from 'row_id' pointing to the '1' in the index, and another from 'column_id' pointing to the '2' in the index.

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/>