
# Pointers and Modular Programming

**Mirza Mohammad Lutfe Elahi**

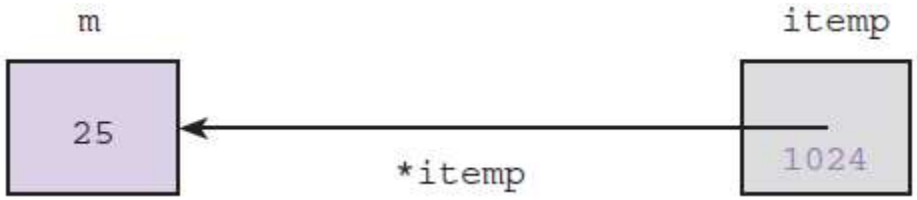
# Outline

- Pointer Variables
- Address Operator and Indirect Reference
- Functions with Output Parameters
- Multiple Calls to a Function
- Scope of Names
- File Input and Output
- Common Programming Errors

# Address Operator

- How to initialize a pointer variable?
- We can use the address operator **&**
- **Example:**

```
int m = 25;
int *itemp;  /* pointer variable */
itemp = &m;  /* Store address of m in pointer
itemp */
```



# Indirect Reference (De-Reference)

We can access and modify a variable:

1. Either directly using the variable name

2. Or indirectly, using a pointer to the variable

- **Example:**

```
int m = 25;
int *itemp;  /* pointer variable */
itemp = &m;  /* Store address of m in pointer
itemp */
*itemp = 35; /* m = 35 */
printf("%d", *itemp);
```

# Triple Use of * (Asterisk)

1. As a multiplication operator:

```
z  = x * y ; /* z = x times y */
```

2. To declare pointer variables:

```
char ch;      /* ch is a character */
char *p;      /* p is pointer to char */
```

3. As an indirection operator:

```
 p = &ch;     /* p = address of ch */
*p = 'A';     /* ch = 'A' */
*p = *p + 1; /* ch = 'A' + 1 = 'B' */
```

# Example


```
#include <stdio.h>
int main(void)  {
  double d = 13.5;
  double *p;  /* p is a pointer to double */
  p = &d;     /* p = address of d */
  printf("Value of  d = %.2f\n", d);
  printf("Value of &d = %d\n", &d);
  printf("Value of  p = %d\n", p);
  printf("Value of *p = %.2f\n", *p);
  printf("Value of &p = %d\n", &p);
  *p = -5.3;  /* d = -5.3 */
  printf("Value of  d = %.2f\n", d);
  return 0;
}
```

```
Value of  d = 13.50
Value of &d = 2293320
Value of  p = 2293320
Value of *p = 13.50
Value of &p = 2293312
Value of  d = -5.30

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

# Pointer Summary


- Using a pointer variable **p**, one can access:

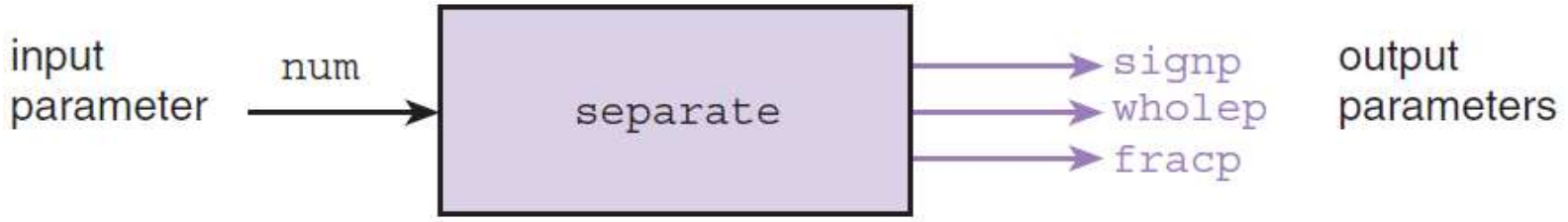
1. Its **direct value**: the value of pointer variable **p**
   - In the example, the value of **p** is **2293320**
   - It is the **address** of variable **d (&d is 2293320)**
2. Its **indirect value**: using the indirection operator ***
   - In the example, ***p** is the value of **d**, which is **13.5**
3. Its **address value**: using the address operator **&**
   - In the example, **&p** is **2293312**

# Function with Output Parameter

- So far, we know how to:
  - Pass **input parameters** to a function
  - Use the `return` statement to return one function result
- Functions can also have **output parameters**
  - To return **multiple results** from a function
- Output parameters are **pointer** variables
  - The caller passes the **addresses** of variables in memory
  - The function uses **indirect reference** to modify variables in the calling function (for output results)

# Example: Function `separate`

- Write a function that separates a number into a sign, a whole number magnitude, and a fractional part.




```
void separate          /* function separate */
  (double num,         /* input number */
   char *signp,        /* sign pointer */
   int *wholep,        /* whole number pointer */
   double *fracp);     /* fraction pointer */
```

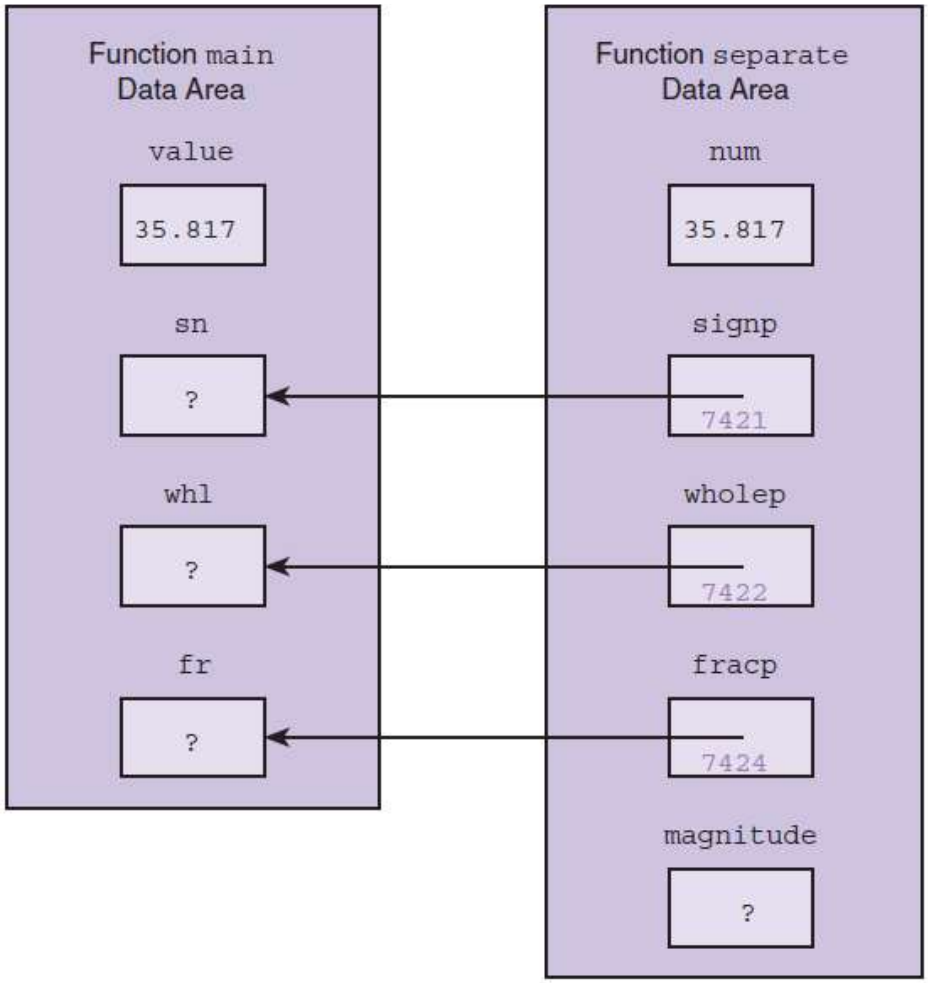
```
7.  void separate(double num, char *signp, int *wholep, double *fracp);
8.
9.  int
10. main(void)
11. {
12.       double value; /* input - number to analyze                    */
13.       char sn;      /* output - sign of value                       */
14.       int whl;      /* output - whole number magnitude of value     */
15.       double fr;    /* output - fractional part of value            */
16.
17.       /* Gets data                                                   */
18.       printf("Enter a value to analyze> ");
19.       scanf("%lf", &value);
20.
21.       /* Separates data value into three parts                       */
22.       separate(value, &sn, &whl, &fr);
23.
24.       /* Prints results                                              */
25.       printf("Parts of %.4f\n sign: %c\n", value, sn);
26.       printf(" whole number magnitude: %d\n", whl);
27.       printf(" fractional part: %.4f\n", fr);
28.
29.       return (0);
30. }
```

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

```
55.         /* Finds magnitude of num (its absolute value) and separates it into
56.            whole and fractional parts                                        */
57.         magnitude = fabs(num);
58.         *wholep = floor(magnitude);
59.         *fracp = magnitude - *wholep;
60.  }

     Enter a value to analyze> 35.817
     Parts of 35.8170
       sign: +
       whole number magnitude: 35
       fractional part: 0.8170
```

Parameter Passing for Function **separate**



```c
/*
 * Tests function order by ordering three numbers
 */
#include <stdio.h>

void order(double *smp, double *lgp);

int
main(void)
{
        double num1, num2, num3; /* three numbers to put in order        */

        /* Gets test data                                                 */
        printf("Enter three numbers separated by blanks> ");
        scanf("%lf%lf%lf", &num1, &num2, &num3);

        /* Orders the three numbers                                       */
        order(&num1, &num2);
        order(&num1, &num3);
        order(&num2, &num3);

        /* Displays results                                               */
        printf("The numbers in ascending order are: %.2f %.2f %.2f\n",
               num1, num2, num3);

        return (0);
}
```

**Sort 3 Numbers**

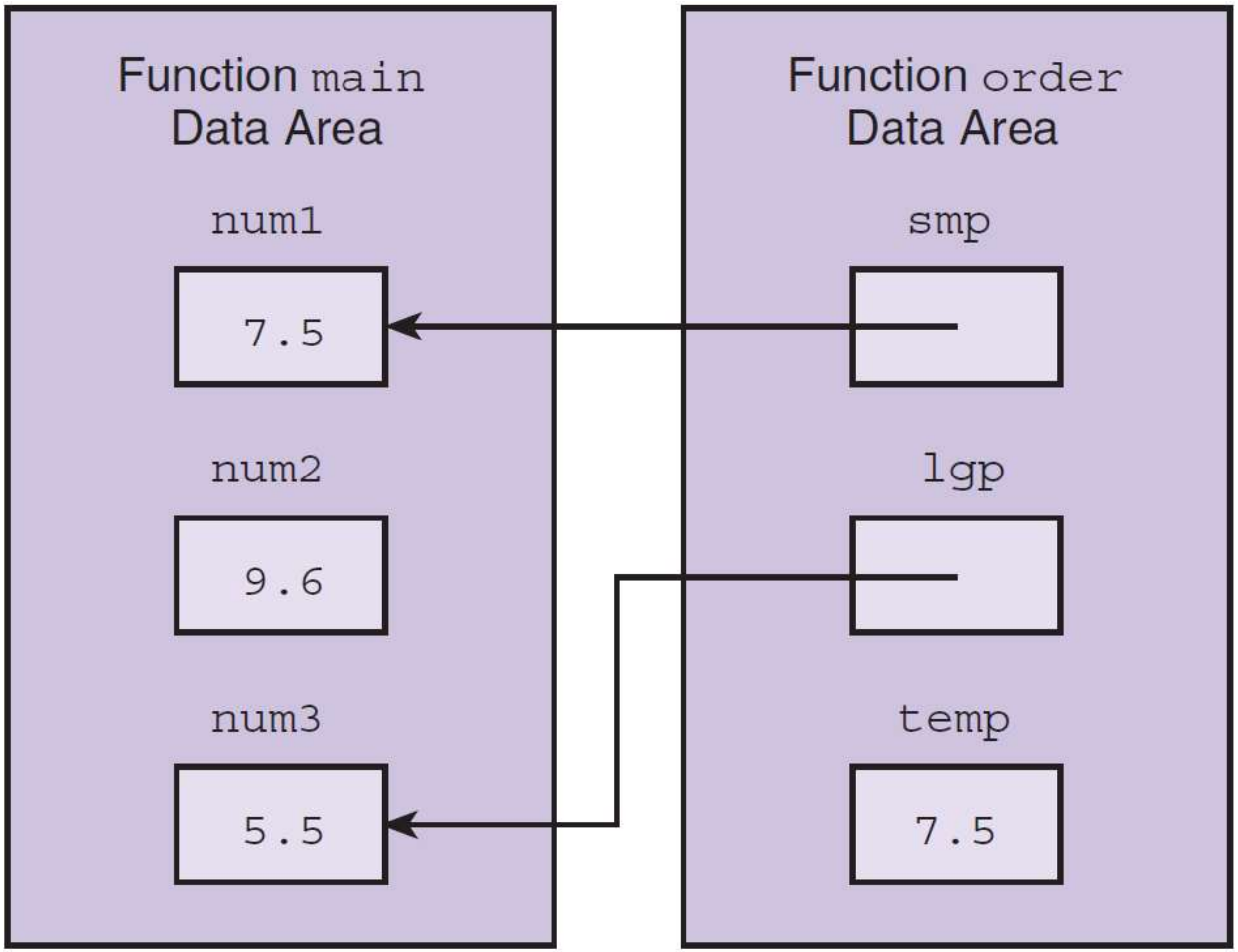
```
28.
29.  /*
30.   * Arranges arguments in ascending order.
31.   * Pre:   smp and lgp are addresses of defined type double variables
32.   * Post:  variable pointed to by smp contains the smaller of the type
33.   *        double values; variable pointed to by lgp contains the larger
34.   */
35.  void
36.  order(double *smp, double *lgp)    /* input/output */
37.  {
38.          double temp; /* temporary variable to hold one number during swap    */
39.          /* Compares values pointed to by smp and lgp and switches if necessary    */
40.          if (*smp > *lgp) {
41.                  temp = *smp;
42.                  *smp = *lgp;
43.                  *lgp = temp;
44.          }
45.  }

Enter three numbers separated by blanks> 7.5 9.6 5.5
The numbers in ascending order are: 5.50 7.50 9.60
```

# Tracing Program: Sort 3 Numbers

| Statement | num1 | num2 | num3 | Effect |
|---|---|---|---|---|
| `scanf(. . .);` | 7.5 | 9.6 | 5.5 | Input Data |
| `order(&num1, &num2);` | 7.5 | 9.6 | 5.5 | No change |
| `order(&num1, &num3);` | 5.5 | 9.6 | 7.5 | swap num1, num3 |
| `order(&num2, &num3);` | 5.5 | 7.5 | 9.6 | swap num2, num3 |
| `printf(. . .);` | | | | 5.50 7.50 9.60 |

# Trace: `order($num1, &num3)`




**Data area after:** `temp = *smp;`

# Scope of a Name

- Region of program where a name is visible
- Region of program where a name can be referenced
- Scope of: **`#define NAME value`**
  - From the definition line until the end of file
  - Visible to all functions that appear after **`#define`**
- Scope of a function prototype
  - Visible to all functions defined after the prototype
- Scope of a parameter and a local variable
  - Visible only inside the function where it is defined
  - Same name can be re-declared in different functions

```
1.  #define MAX 950
2.  #define LIMIT 200
3.
4.  void one(int anarg, double second);     /* prototype 1 */
5.
6.  int fun_two(int one, char anarg);       /* prototype 2 */
7.
8.  int
9.  main(void)
10. {
11.          int localvar;
12.         . . .
13. } /* end main */
14.
15.
16. void
17. one(int anarg, double second)          /* header 1     */
18. {
19.         int onelocal;                  /* local 1      */
20.         . . .
21. } /* end one */
22.
23.
24. int
25. fun_two(int one, char anarg)           /* header 2     */
26. {
27.         int localvar;                  /* local 2      */
28.         . . .
29. } /* end fun_two */
```

**MAX and LIMIT are visible to all functions**

**prototypes are typically visible to all functions**

**function one is not visible to fun_two: has parameter one**

**localvar is visible inside main only**

**anarg, second, and onelocal are visible inside function one only**

**one, anarg, and localvar are visible inside fun_two only**

# Why Data Files?

- So far, all our examples obtained their input from the keyboard and displayed their output on the screen
- However, the input data can be large that it will be inconvenient to enter the input from the keyboard
  - Example: processing large number of employees data
- Similarly, there are applications where the output will be more useful if it is stored in a file
- The good news is that C allows the programmer to use data files, both for input and output

# Using Data Files

- The process of using data files for input/output involves four steps as follows:
  1. Declare pointer variables of type `FILE *`
  2. Open the files for reading/writing using `fopen` function
  3. Read/write the files using `fscanf` and `fprintf`
  4. Close the files after processing the data using `fclose`
- In what follows, we explain each of these steps

# Declaring `FILE` Pointer Variables

- Declare pointer variables to files as follows:

```
FILE *inp;  /* pointer to input file */

FILE *outp; /* pointer to output file */
```

- Note that the type **`FILE`** is in upper case
  - The type **`FILE`** stores information about an open file
- Also note the use of **`*`** before a pointer variable
  - **`inp`** and **`outp`** are pointer variables
  - Recall that pointer variables store memory addresses

# Opening Data Files for I/O

- The second step is to open a file for reading or writing
- Suppose our input data exists in file: `"data.txt"`
- To open a file for reading, write the following:

```
inp = fopen("data.txt", "r");
```

- The `"r"` indicates the purpose of reading from a file
- Suppose we want to output data to: `"results.txt"`
- To open a file for writing, write the following:

```
outp = fopen("results.txt", "w");
```

- The `"w"` indicates the purpose of writing to a file

# Handling File not Found Error

- `inp = fopen("data.txt", "r");`
- If the above **`fopen`** operation succeeds:
  - It returns the address of the open **`FILE`** in **`inp`**
  - The **`inp`** pointer can be used in all file read operations
- If the above **`fopen`** operation fails:
  - For example, if the file **`data.txt`** is not found on disk
  - It returns the **NULL** pointer value and assign it to **`inp`**
- Check the pointer **`inp`** immediately after **`fopen`**

```
if (inp == NULL)
  printf("Cannot open file: data.txt\n");
```

# Creating a File for Writing

- `outp = fopen("results.txt", "w");`
- If the above `fopen` operation succeeds:
  - It returns the address of the open `FILE` in `outp`
  - The `outp` pointer can be used in all file write operations
- If file `results.txt` does not exist on the disk
  - The OS typically creates a new file `results.txt` on disk
- If file `results.txt` already exists on the disk
  - The OS typically clears its content to make it a new file
- If `fopen` fails to create a new file for writing, it returns the **NULL** pointer in `outp`

# Input from & Output to Data Files

- The third step is to scan data from an input file and to print results into an output file
- To input a double value from file **`data.txt`**, use:

```
fscanf(inp, "%lf", &data);
```

- The **`fscanf`** function works the same way as **`scanf`**
  - Except that its first argument is an input **`FILE`** pointer
- To output a double value to **`results.txt`**, use:

```
fprintf(outp, "%f", data);
```

- Again, **`fprintf`** works similar to **`printf`**
  - Except that its first argument is an output **`FILE`** pointer

# Closing Input and Output Files

- The final step in using data files is to close the files after you finish using them
- The `fclose` function is used to close both input and output files as shown below:

```
fclose(inp);
fclose(outp);
```

- **Warning:** Do not forget to close files, especially output files. This is necessary if you want to re-open a file for reading after writing data to it. The OS might delay writing data to a file until closed.

```c
/* Inputs each number from an input file and writes it
 * rounded to 2 decimal places on a line of an output file.
 */
#include <stdio.h>

int
main(void)
{
     FILE *inp;          /* pointer to input file */
     FILE *outp;         /* pointer to ouput file */
     double item;
     int input_status;  /* status value returned by fscanf */

     /* Prepare files for input or output */
     inp = fopen("indata.txt", "r");
     outp = fopen("outdata.txt", "w");

     /* Input each item, format it, and write it */
     input_status = fscanf(inp, "%lf", &item);
     while (input_status == 1) {
         fprintf(outp, "%.2f\n", item);
         input_status = fscanf(inp, "%lf", &item);
     }

     /* Close the files */
     fclose(inp);
     fclose(outp);

     return (0);
}
```

# Sample Run

- File: **indata.txt**

```
344 55 6.3556 9.4
43.123 47.596
```

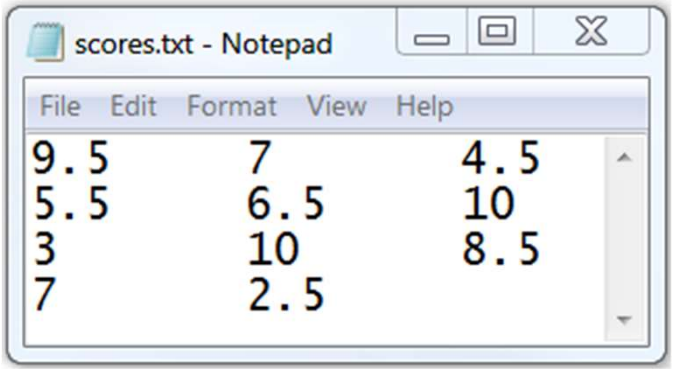
- File: **outdata.txt**

```
344.00
55.00
6.36
9.40
43.12
47.60
```

# End-of-File Controlled Loop

- When reading input from a data file, the program does not know how many data items to read
- Example: finding class average from student grades
- The grades are read from an input file one at a time in a loop, until the end of file is reached
- The question here is how to detect the end of file?
- The good news is that `fscanf` returns a special value, named **EOF**, when it encounters **End-Of-File**
- We can take advantage of this by using **EOF** as a condition to control the termination of a loop

```
/* This program computes the average score of a class
   The scores are read from an input file, scores.txt */
#include <stdio.h>
int main (void) {
   FILE *infile;
   double score, sum=0, average;
   int count=0, status;
   infile = fopen("scores.txt", "r");
   status = fscanf(infile, "%lf", &score);
   while (status != EOF) {
      printf("%5.1f\n", score);
      sum += score;
      count++;
      status = fscanf(infile, "%lf", &score);
   }
   average = sum / count;
   printf("\nSum of scores is %.1f\n", sum);
   printf("Average score is %.2f\n", average);
   fclose(infile);
   return 0;
}
```




```
  9.5
  7.0
  4.5
  5.5
  6.5
 10.0
  3.0
 10.0
  8.5
  7.0
  2.5

Sum of scores is 74.0
Average score is 6.73
```

# Common Programming Errors

- Be careful when using pointer variables
  - A pointer should be initialized to a valid address before use
  - De-referencing an invalid/NULL pointer is a runtime error
- Calling functions with output parameters
  - Remember that output parameters are pointers
  - Pass the address of a variable to a pointer parameter
- Do not reference names outside their scope
- Create a file before reading it in a program
  - Remember that `fopen` prepares a file for input/output
  - The result of `fopen` should not be a `NULL` pointer
  - Check the status of `fscanf` to ensure correct input
  - Remember to use `fclose` to close a file, when done