# CMPE 252
# C PROGRAMMING

SPRING 2021

WEEK 4-5

# POINTERS AND MODULAR PROGRAMMING

CHAPTER 6

*Problem Solving & Program Design in C*

---

*Eighth Edition*

*Global Edition*

*Jeri R. Hanly & Elliot B. Koffman*

# Pointers

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

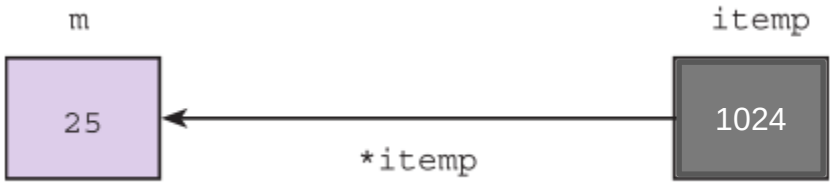
```
int  m  =  25;
int  *itemp;          /* a pointer to an integer */
```

  - 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) |
|---|---|---|
| itemp | gray shaded cell | pointer (1024) |
| *itemp | 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

```c
#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

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int m = 25;
    int *itemp;
    itemp = &m;

    printf("%d\n", itemp);
    printf("%d\n", *itemp);

    *itemp = 35;
    printf("%d\n", m);

    *itemp = 2 * (*itemp);
    printf("%d\n", m);
    printf("%d\n", *itemp);
    printf("%d\n", itemp);

    *itemp = 3 * m;
    printf("%d\n", m);
    printf("%d\n", *itemp);
    printf("%d\n", itemp);
}
```

Output?

2686744
25
35
70
70
2686744
210
210
2686744

# Pointers Quick Check

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int m = 10, n = 5;
    int *mp, *np;
    mp = &m;
    np = &n;

    *mp = *mp + *np;
    *np = *mp - *np;
    printf("m   *mp   n   *np \n%d%4d%5d%6d", m, *mp, n, *np);
}
```

Output?

```
m   *mp   n   *np
15  15   10    10
```

- What do I do if I want two pointers to 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

# More examples

- int *ip=NULL;
- int i =5;
- int k=0;

- ip=&i;

- k = ++ (*ip);

versus?

- k=++i;

# Operator Precedence (more)

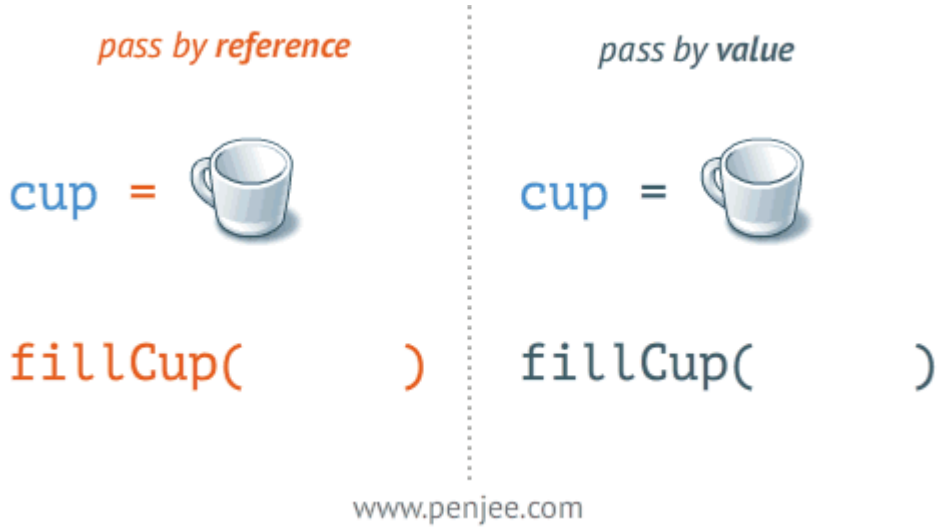
| Operator | Precedence |
|---|---|
| function calls (), postfix increment (i++), postfix decrement (i--) | highest (evaluated first) |
| ! + - (unary operator), prefix increment (++i), prefix decrement (--i) , & adress of, * indirection [RIGHT TO LEFT] | |
| * / % | |
| + - | |
| < <= >= > | |
| ==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

```c
#include <stdio.h>

void f(int a)
{
    a = 3;
}

int main()
{
    int x = 1;

    printf("Before calling function f\n");
    printf("x = %d\n",x);
    f(x);
    printf("After calling function f\n");
    printf("x = %d\n",x);

    return 0;
}
```

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

# Simulation of Pass-by-reference in C

```c
#include <stdio.h>

void g(int *aPtr)
{
    *aPtr = 3;
}

int main()
{
    int x = 1;

    printf("Before calling function g\n");
    printf("x = %d\n",x);
    g(&x);
    printf("After calling function g\n");
    printf("x = %d\n",x);

    return 0;
}
```

```
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

```c
#include <stdio.h>
#include <math.h>
double powerFourByValue(double n);

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

    printf("The original value of number is %.2f", number);

    // pass number by value to powerFourByValue
    number = powerFourByValue(number);

    printf("\nThe new value of number is %.2f\n", number);
}

// calculate and return cube of integer argument
double powerFourByValue(double n)
{
    return pow(n,4);
}
```

# Pass-by-reference Example

```c
#include <stdio.h>
#include <math.h>
void powerFourByReference(double *nPtr);

int main(void)
{
    double number = 7.2; // initialize number

    printf("The original value of number is %.2f", number);

    // pass number by reference to powerFourByReference
    powerFourByReference(&number);

    printf("\nThe new value of number is %.2f\n", number);
}

// calculate and return cube of integer argument
void powerFourByReference(double *nPtr)
{
    *nPtr = pow(*nPtr,4);
}
```

# 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);
}
```

625.0 625.0

number 625.0

```
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)***

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

number 625.0

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

625.0

nPtr undefined

# 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

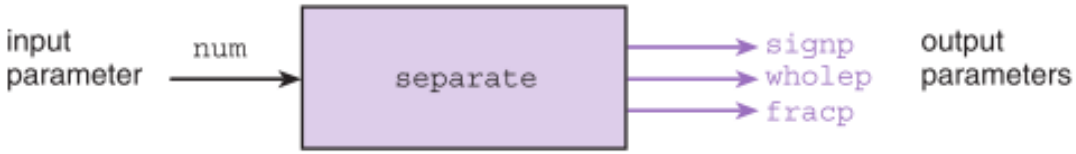
- We've used the return statement to send back one result value from a function.
- We can also use output parameters to return multiple results from a function.
- How??
  - We can use address of operator to store actual parameter's address which is a pointer instead of actual value. Therefore we can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

```c
/*
 * Separates a number into three parts: a sign (+, -, or blank),
 * a whole number magnitude, and a fractional part.
 * Pre: num is defined; signp, wholep, and fracp contain addresses of memory
 *      cells where results are to be stored
 * Post: function results are stored in cells pointed to by signp, wholep, and
 *       fracp
 */
void separate(double num,     /* input - value to be split                    */
              char   *signp,  /* output - sign of num                         */
              int    *wholep, /* output - whole number magnitude of num       */
              double *fracp)  /* output - fractional part of num              */
{
      double magnitude; /* local variable - magnitude of num                */
      /* Determines sign of num */
      if (num < 0)
            *signp = '-';
      else if (num == 0)
            *signp = ' ';
      else
            *signp = '+';

      /* Finds magnitude of num (its absolute value) and separates it into
         whole and fractional parts                                          */
      magnitude = fabs(num);
      *wholep = floor(magnitude);
      *fracp = magnitude - *wholep;
}
```

indirect referencing

indirect referencing

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

```c
#include <stdio.h>
#include <math.h>
void separate(double num, char *signp, int *wholep, double *fracp);

int main(void)
{
        double value; /* input - number to analyze                          */
        char sn;      /* output - sign of value                             */
        int whl;      /* output - whole number magnitude of value           */
        double fr;    /* output - fractional part of value                  */

        /* Gets data                                                        */
        printf("Enter a value to analyze> ");
        scanf("%lf", &value);

        /* Separates data value into three parts                            */
        separate(value, &sn, &whl, &fr);

        /* Prints results                                                   */
        printf("Parts of %.4f\n sign: %c\n", value, sn);
        printf(" whole number magnitude: %d\n", whl);
        printf(" fractional part: %.4f\n", fr);

        return (0);
}
```

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

# Parameter Correspondance for separate

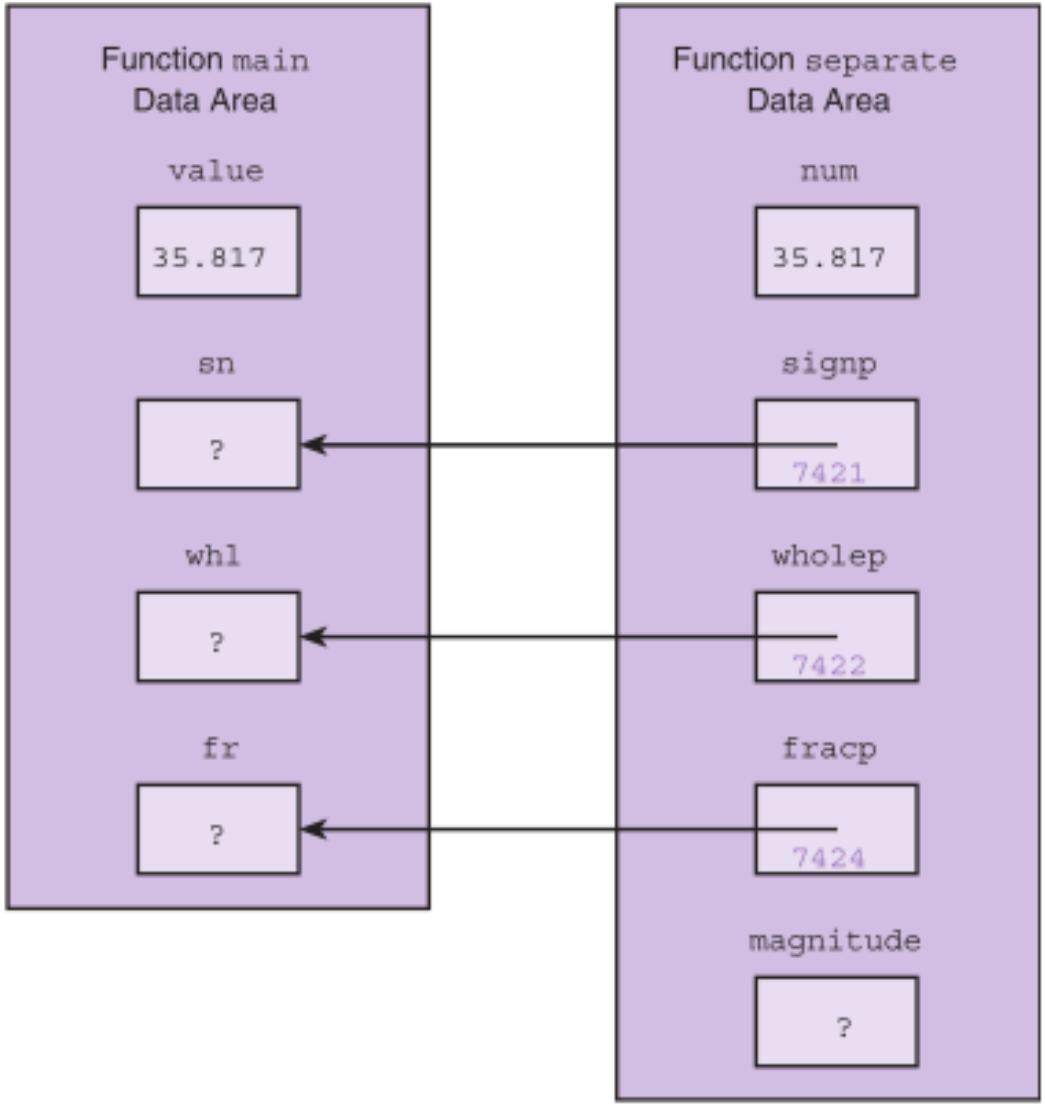


**TABLE 6.2** Effect of & Operator on the Data Type of a Reference

| 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
```

```c
#include <stdio.h>

void sum_n_avg(double n1, double n2, double n3, double *sump, double *avgp);

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(one, two, three, &sum, &avg);
    printf("Sum is %f, average is %f", sum, avg);

    return 0;
}

void sum_n_avg(double n1, double n2, double n3, double *sump, double *avgp)
{
    *sump = n1+n2+n3;
    *avgp = *sump/3.0;
}
```

# MULTIPLE CALLS TO A FUNCTION WITH INPUT/OUTPUT PARAMETERS

An example of sorting data

```c
#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);
}
```

```
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 |
|---|---|---|---|---|
| `scanf("...", &num1, &num2, &num3);` | 7.5 | 9.6 | 5.5 | Enters data |
| `order(&num1, &num2);` | | | | No change |
| `order(&num1, &num3);` | 5.5 | 9.6 | 7.5 | Switches num1 and num3 |
| `order(&num2, &num3);` | 5.5 | 7.5 | 9.6 | Switches num2 and num3 |
| `printf("...", num1, num2, num3);` | | | | Displays 5.5 7.5 9.6 |

**FIGURE 6.8**

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

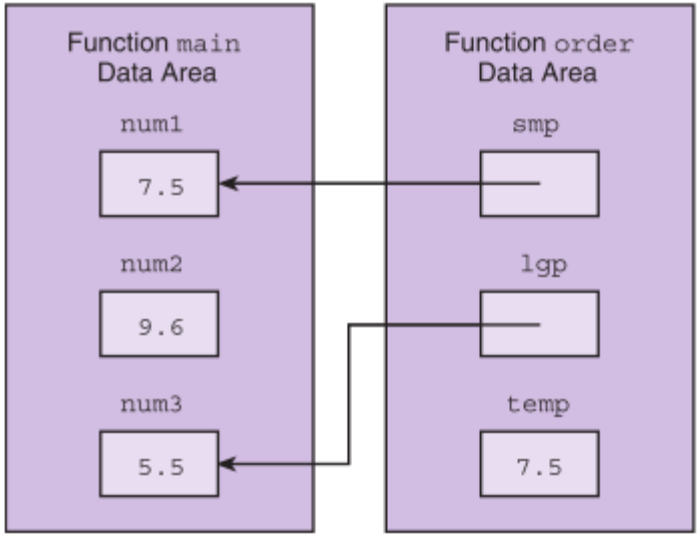


**TABLE 6.4** Different Kinds of Function Subprograms

| 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.<br>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

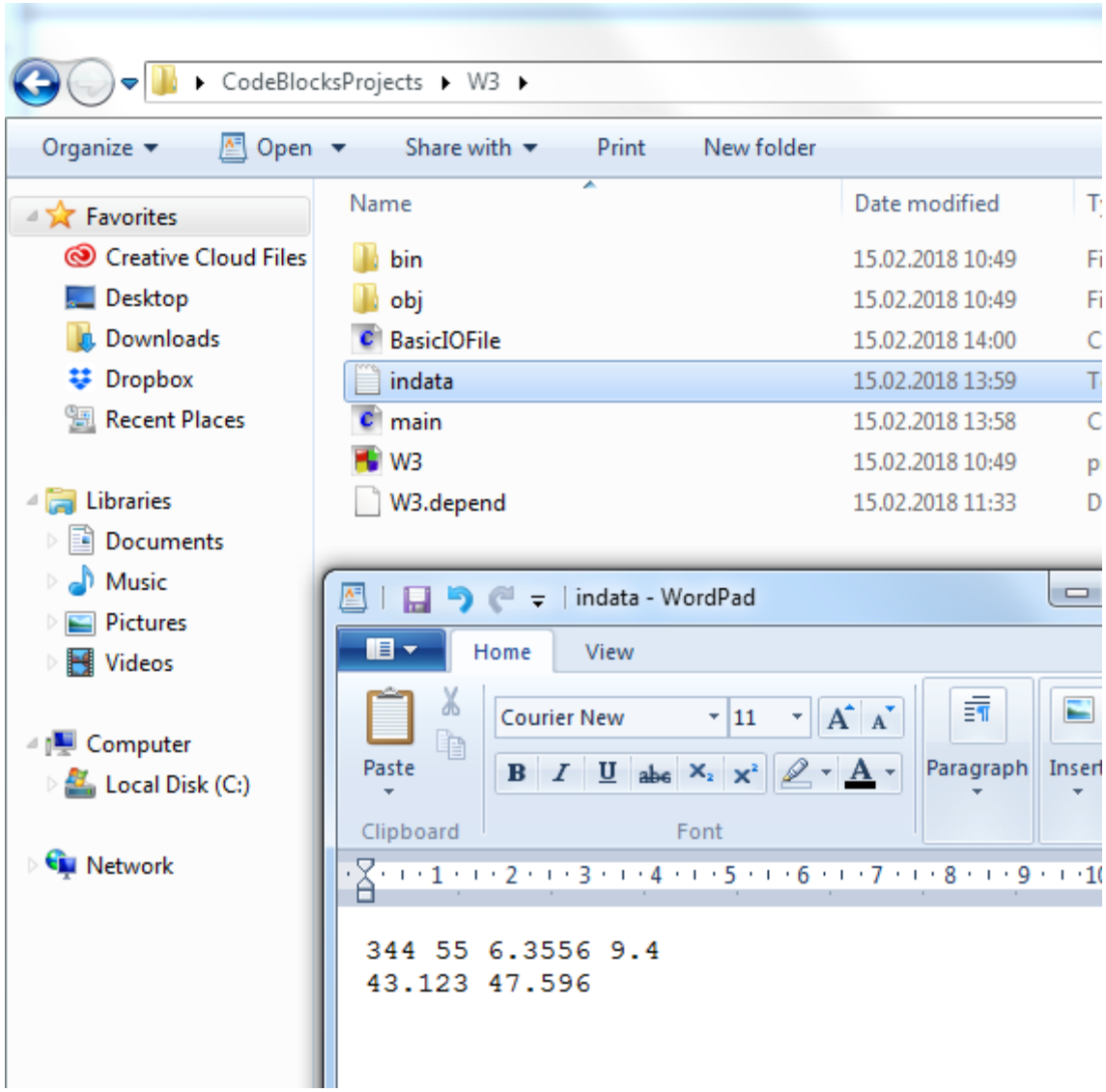
- C allows a program to explicitly name a file for input or output.
- Declare file pointers:
  - FILE *inp; /* pointer to input file */
  - FILE *outp; /* pointer to output file */
- Prepare for input or output before permitting access:
  - inp = fopen("infile.txt", "r");
  - outp = fopen("outfile.txt", "w");

- infile is the source of input
- OS stores the necessary access value in the file pointer variable inp
- r is read

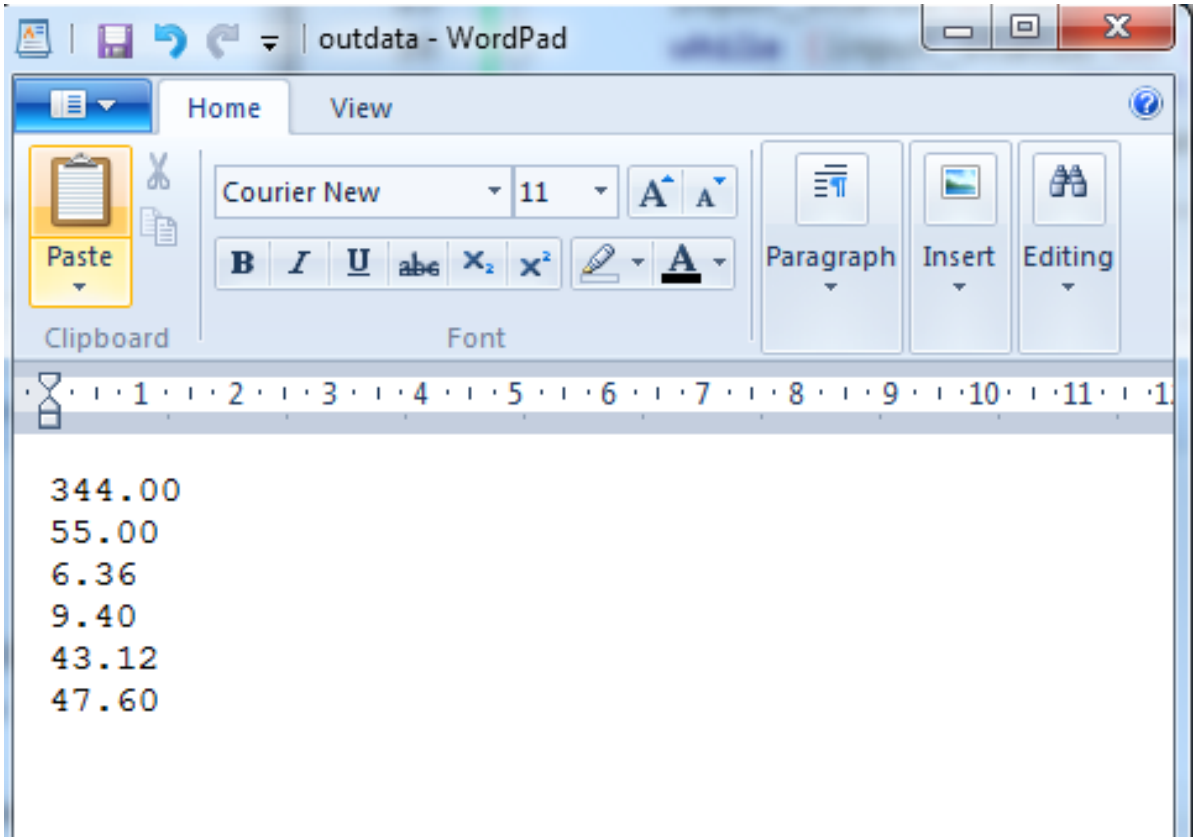
- outfile is where we write
- OS stores the necessary access value in the file pointer variable outp.
- w is write

# 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);


CodeBlocksProjects ▸ W3 ▸
Organize
Open
Share with
Print
New folder
Favorites
Creative Cloud Files
Desktop
Downloads
Dropbox
Recent Places
Libraries
Documents
Music
Pictures
Videos
Computer
Local Disk (C:)
Network
Name
Date modified
bin 15.02.2018 10:49
obj 15.02.2018 10:49
BasicIOFile 15.02.2018 14:00
indata 15.02.2018 13:59
main 15.02.2018 13:58
W3 15.02.2018 10:49
W3.depend 15.02.2018 11:33
indata - WordPad
Home
View
Courier New
11
Paste
Clipboard
Font
Paragraph
Insert
344 55 6.3556 9.4
43.123 47.596

```c
#include <stdio.h>

int main(void)
{
        FILE *inp;          /* pointer to input file */
        FILE *outp;         /* pointer to output 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);
}
```


outdata - WordPad
Home
View
Paste
Clipboard
Courier New
11
Font
Paragraph
Insert
Editing
344.00
55.00
6.36
9.40
43.12
47.60

| 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]

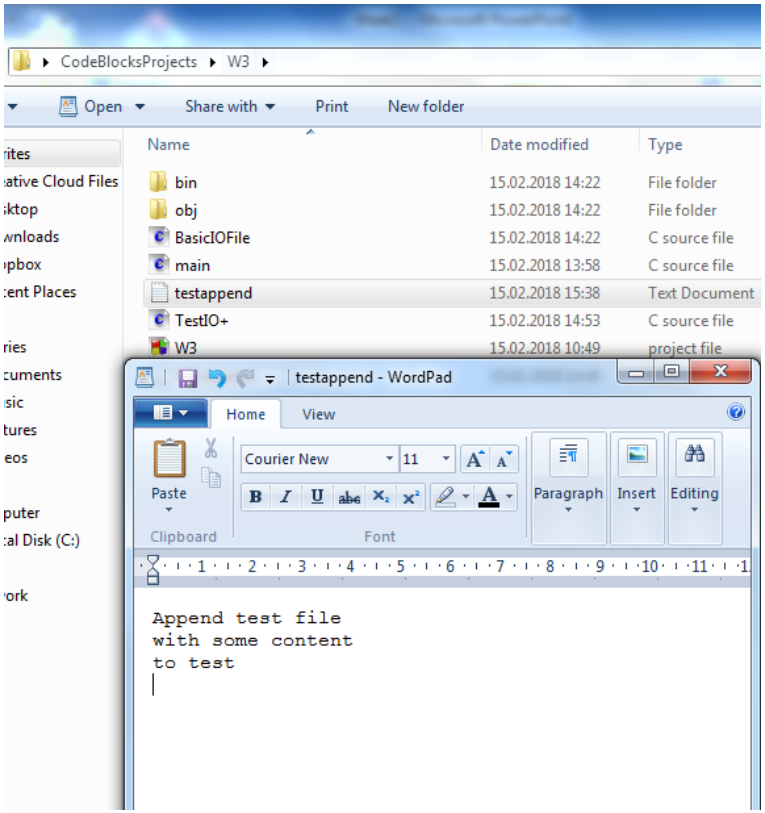
r+ = read/write mode – used for update
doesn't delete the content of the file and doesn't create a new file if such file doesn't exist
the new stream is positioned at the beginning of the file

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.

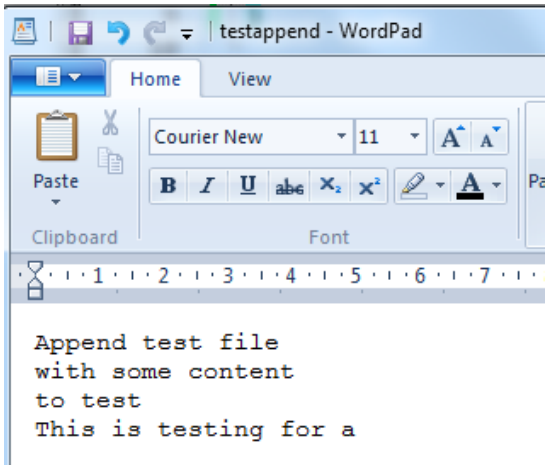
# append - a

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




Now run the below code:

```c
#include <stdio.h>
int main()
{
    FILE *fp;

    fp = fopen("testappend.txt", "a");
    fprintf(fp, "This is testing for a\n");
    fclose(fp);
}
```



# r+

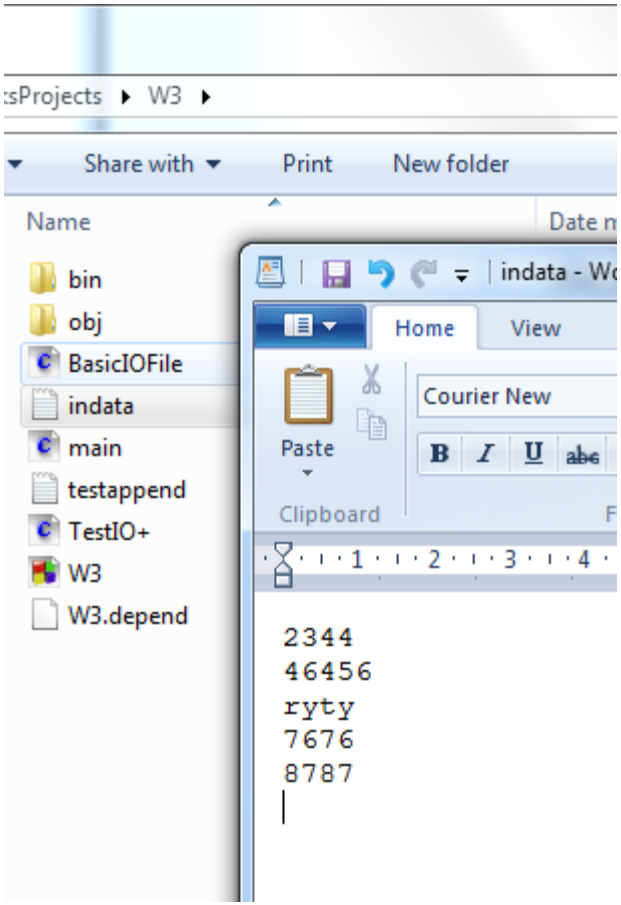
- Assume that we do not have any file named indata.txt beforehand
- Now run the below code:

```c
#include <stdio.h>
int main()
{
    FILE *fp;

    fp = fopen("indata.txt", "r+");
    fprintf(fp, "This is testing for r+\n");
    fclose(fp);
}
```

- No file will be created

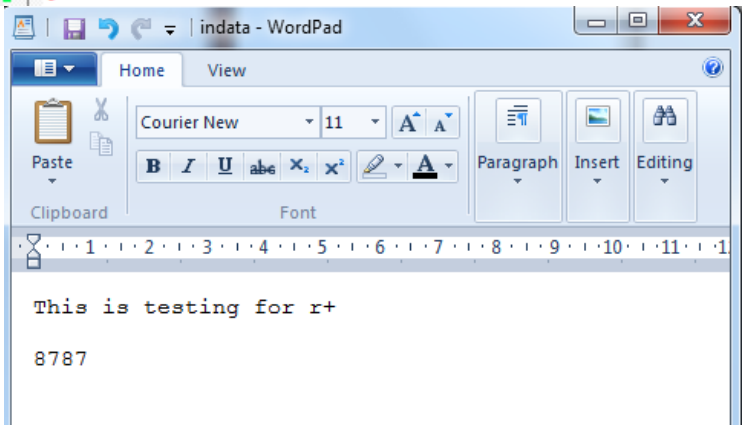
# r+

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




Now run the code again:

```
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  }
```

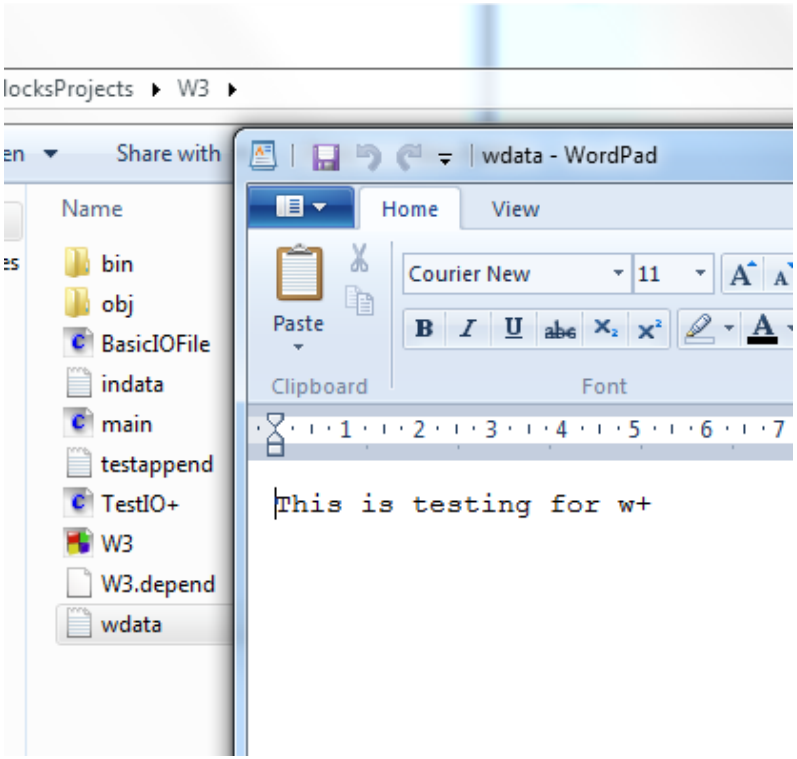


# w+

- Do not create any file
- Run the below code

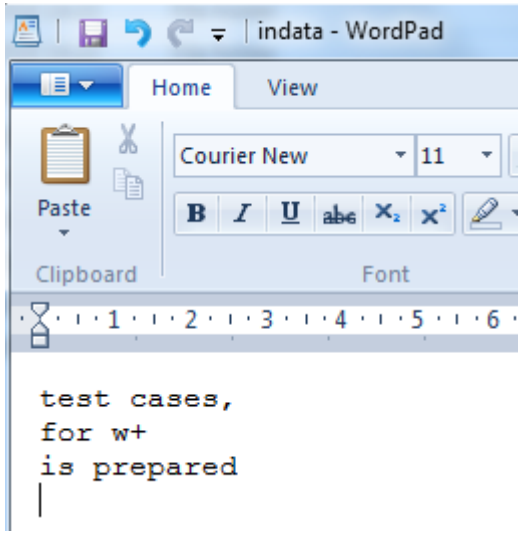
```c
#include <stdio.h>
int main()
{
    FILE *fp;

    fp = fopen("wdata.txt", "w+");
    fprintf(fp, "This is testing for w+\n");
    fclose(fp);
}
```

- File is created with given content

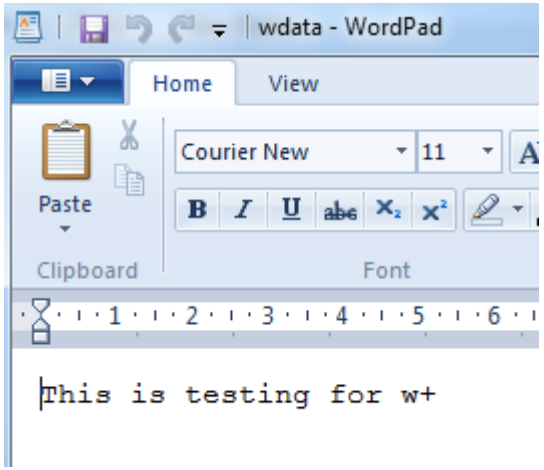


# w+

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




```c
#include <stdio.h>
int main()
{
    FILE *fp;

    fp = fopen("wdata.txt", "w+");
    fprintf(fp, "This is testing for w+\n");
    fclose(fp);
}
```




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.

```c
#define MAX 950
#define LIMIT 200

void one(int anarg, double second);
int fun_two(int one, char anarg);

int main(void)
{
    int localvar;
}

void one(int anarg, double second)
{
    int onelocal;
}

int fun_two(int one, char anarg)
{
    int localvar;

}
```

**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 |

```c
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 }
```

```c
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 }
23
```

others

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

| | Line | Message |
|---|---|---|
| | | === Build: Release in W3 (compiler: GNU GCC Compiler) === |
| Users\gizem... | | In function 'main': |
| Users\gizem... | 9 | warning: unused variable 'localvar' [-Wunused-variable] |
| Users\gizem... | | In function 'one': |
| Users\gizem... | 14 | warning: unused variable 'onelocal' [-Wunused-variable] |
| Users\gizem... | | In function 'fun_two': |
| Users\gizem... | 19 | error: called object 'one' is not a function or function pointer |
| Users\gizem... | 17 | note: declared here |
| Users\gizem... | 20 | warning: unused variable 'localvar' [-Wunused-variable] |
| Users\gizem... | 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