BIRZEIT UNIVERSITY

Pointers

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

By: Mamoun Nawahdah (PhD)
2013/2014

Chapter Objectives

- To learn about pointers and indirect addressing.
- To see how to access external data files in a program and to be able to read from input files 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*.

New Uses of & and *

- & when applied to a variable, yields its **address** (pointer to the variable).
- * when applied to an address (**pointer**), **fetches** the **value** stored at that address.

Pointers and the Indirection Operator

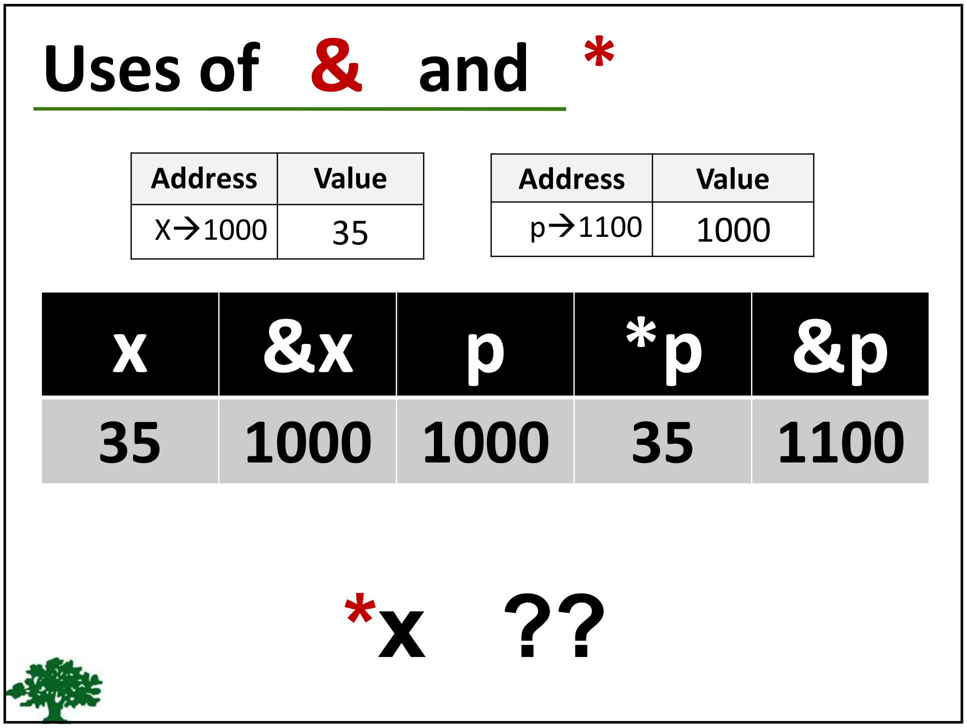
```
int x = 35;

int* p = &x;   // p points at x now.

int y = *p;
// y has the value pointed out by the pointer p.
*p= 13;
// 13 was inserted to the place pointed by p.
```

Address	Value
X→1000	35

Address	Value
p→1100	1000

Address	Value
y→1200	35

Address	Value
X→1000	13



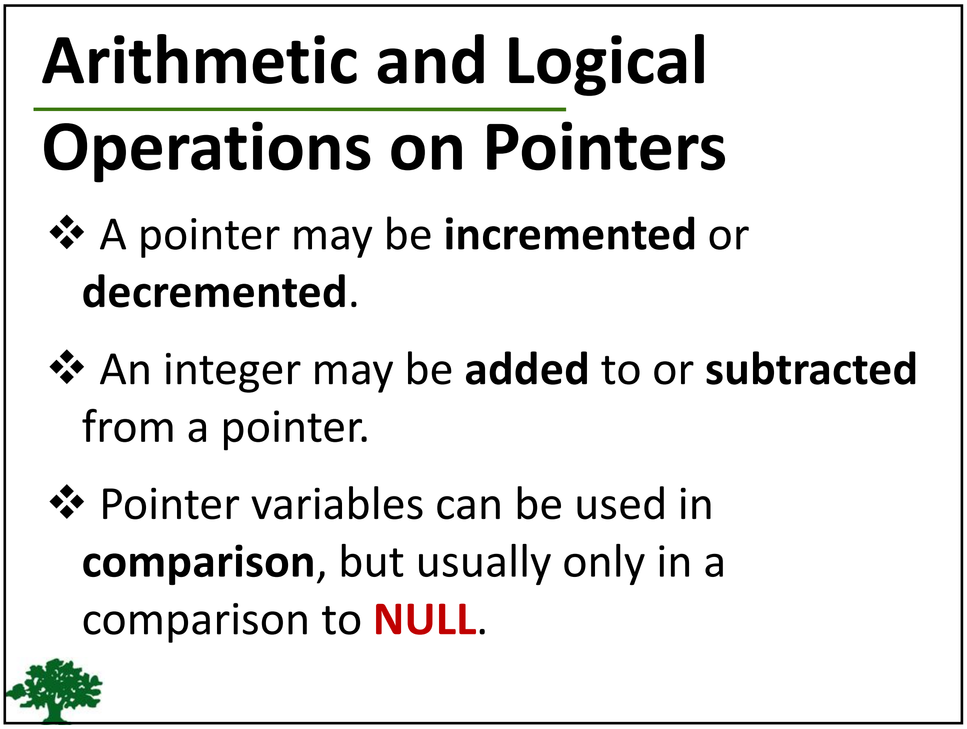
Uses of & and *

Address	Value
X→1000	35

Address	Value
p→1100	1000

x	&x	p	*p	&p
35	1000	1000	35	1100

*x ??



Arithmetic and Logical Operations on Pointers

- A pointer may be **incremented** or **decremented**.
- An integer may be **added** to or **subtracted** from a pointer.
- Pointer variables can be used in **comparison**, but usually only in a comparison to **NULL**.

Arithmetic Operations on Pointers

❖ When an integer is added to a pointer, the new pointer value is changed by the integer times the number of bytes in the data variable the pointer is pointing to.

❖ Example:

p = &x; // size of **int** is **4 bytes**

p = p + 2; // address is increased by 8 (2*4) bytes.

Address	**Value**
X→1000	35

Address	**Value**
p→1100	1000

Address	**Value**
1008	????

Address	**Value**
p→1100	1008

What is the use of Pointers?

❖ Pointers can be used to operate on **variable length arrays**.

❖ Pointers can be "**cheaper**" to pass around a program.

❖ You could program without using them, but you would be making life more easier by using them.

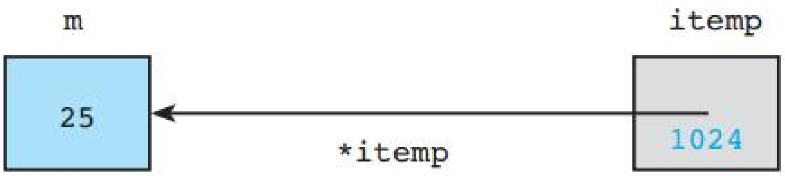
The True Horror of Pointer

- ❖ Each pointer **Always** points something.
- ❖ **No bounds checking**: pointers can point outside the program.
- ❖ **No type checking**: you can cast a pointer to anything.

You just have to be careful while using pointers

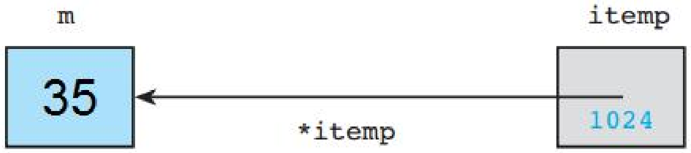
Example:

```
int m = 25;
int *itemp;    // a pointer to an integer
itemp = &m; // Store address of m in pointer itemp
```



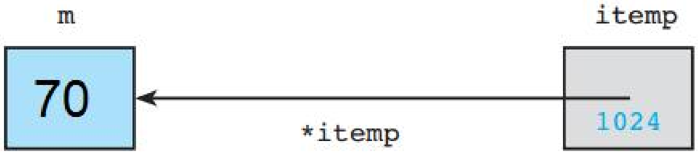
Example: Indirect Reference

*itemp = 35; // stores **35** in the variable **m** that is pointed to by **itemp**.



printf("%d", *itemp); // displays the new value of m ➔ 35

*itemp = 2 * (*itemp); // doubles the variable pointed to by itemp

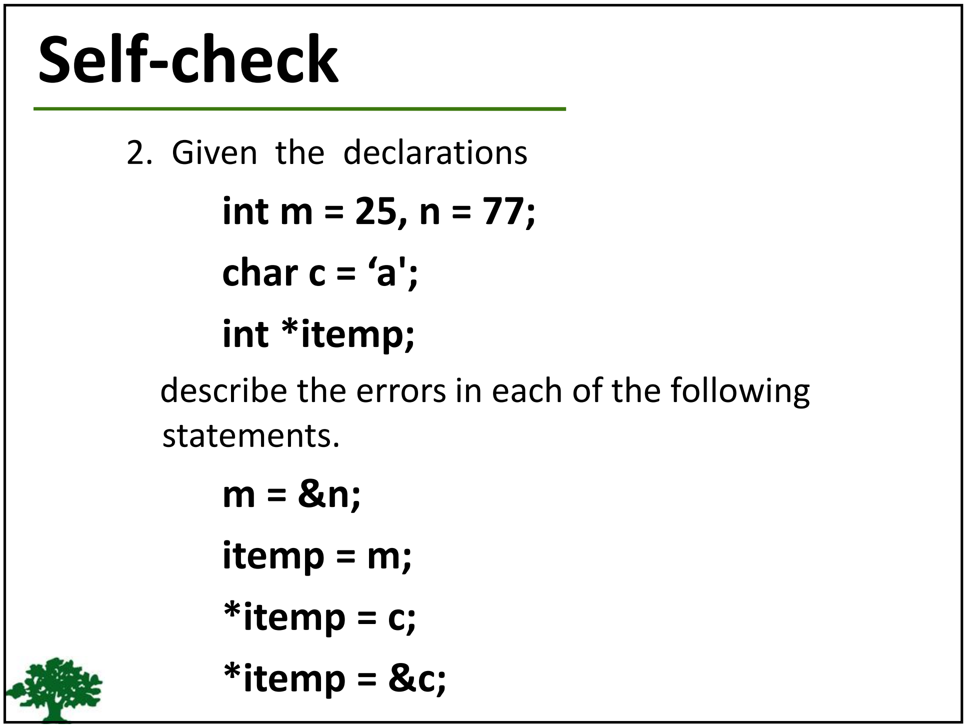


Self-check

1. Trace the execution of the following fragment.

```
int m = 10, n = 5;
int *mp, *np;
mp = &m;
np = &n;
*mp = *mp + *np;
*np = *mp – *np;
printf("%d %d\n%d %d\n", m, *mp, n, *np);
```

```
15   15
10   10
```



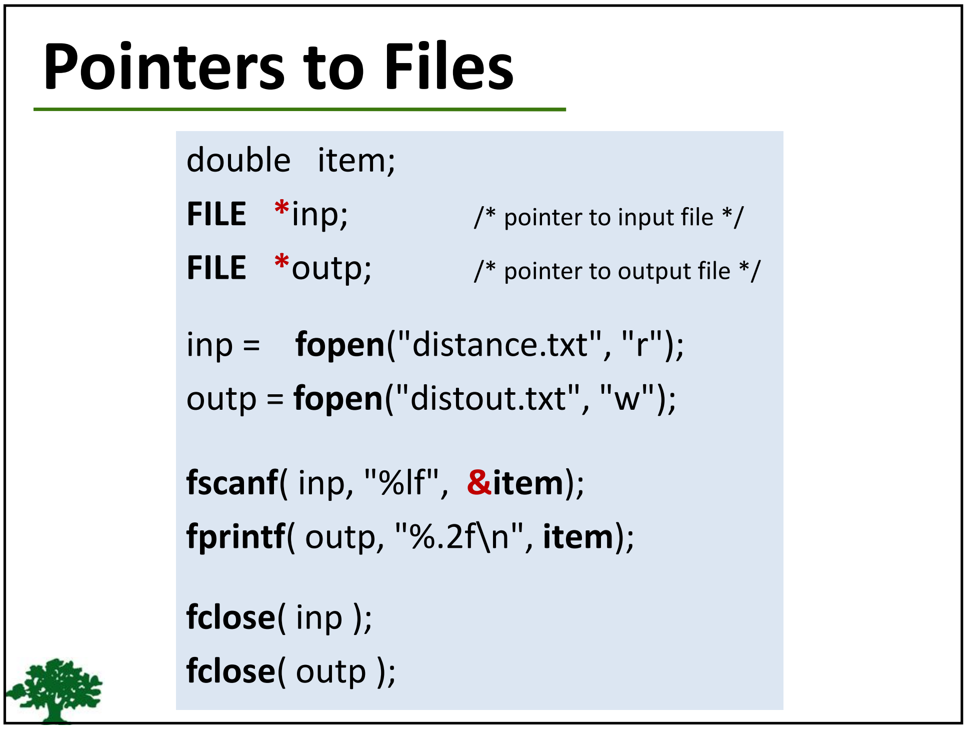
Self-check

2. Given the declarations

```
int m = 25, n = 77;
char c = ‘a';
int *itemp;
```

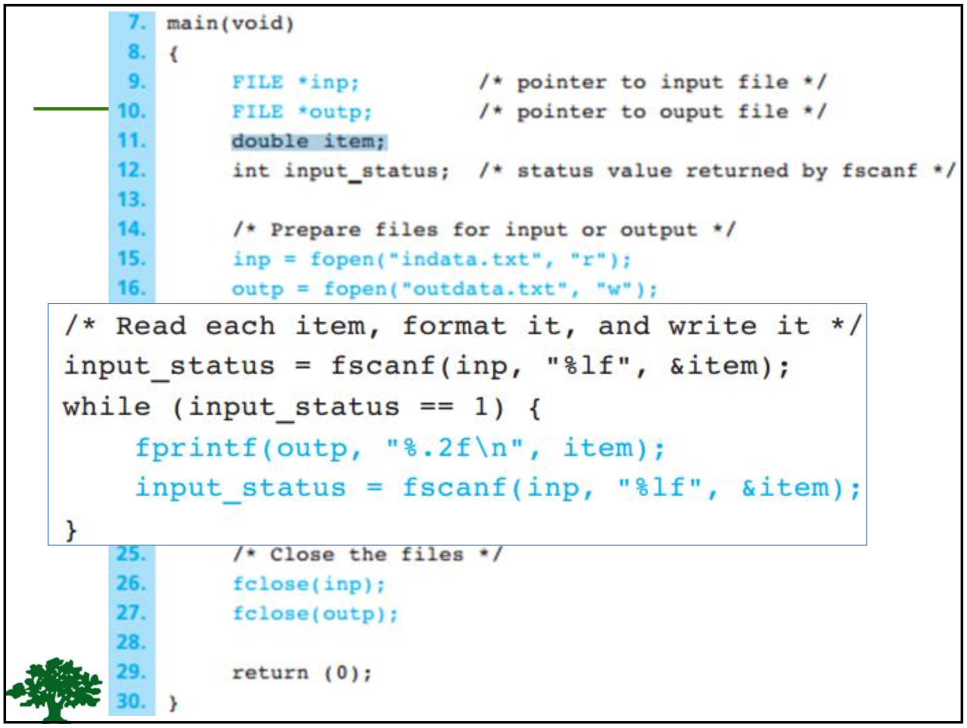
describe the errors in each of the following statements.

```
m = &n;
itemp = m;
*itemp = c;
*itemp = &c;
```



Pointers to Files

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

inp =    fopen("distance.txt", "r");
outp = fopen("distout.txt", "w");

fscanf( inp, "%lf",  &item);
fprintf( outp, "%.2f\n", item);

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



```
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");
/* Read 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);
}
```

Types of Functions

- **No input** arguments, **no** value **returned** – **void functions without arguments**.
- **Input** arguments, **no** value **returned** - **void functions with arguments**.
- **Single** value **returned**.
- **Multiple** value **returned**.

Arguments passed by values

- **Argument lists** are used to communicate information from the **main** function to its function subprograms.
  - Arguments make functions more versatile because they allow us to execute the same function with different sets of data.
- **Return values** are used to communicate information from the function subprogram back to the **main** program.

Arguments passed by values

- When a function is called, it is given **a copy of the values** that are passed in as arguments.
  - If you manipulate the value of an argument, it has no impact on its value in the **main** function.
  - Therefore, these are called **input parameters**, because they can only bring information into the function, and not back out.

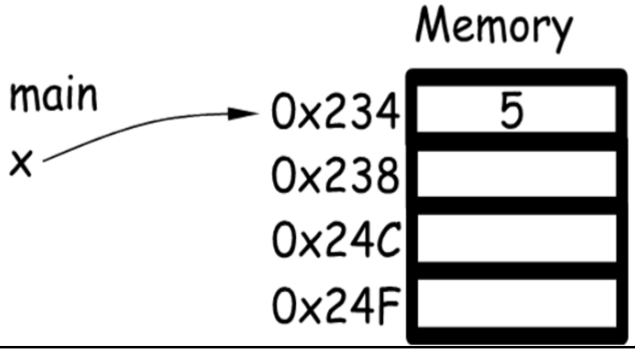
Example with pass by value

```
void myFunc(int);
int main(void){
    int x = 5;
    myFunc(x);
    printf("%d\n",  x);
}
void myFunc(int arg){
    arg  = 4;
}
```

/* Output */
5

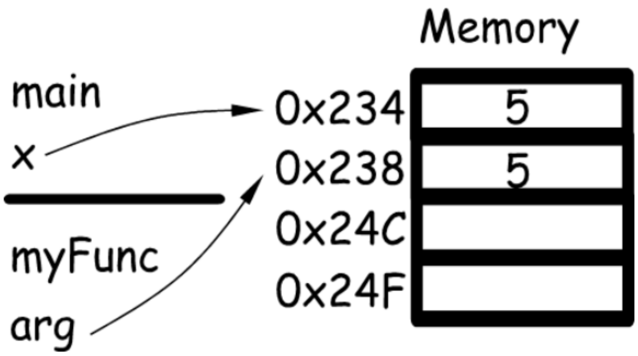
In main: int x = 5;

- In main, **x** is assigned the value 5.
- This places the value 5 in the memory cell reserved for **x**
- In this case, it is at address 0x234



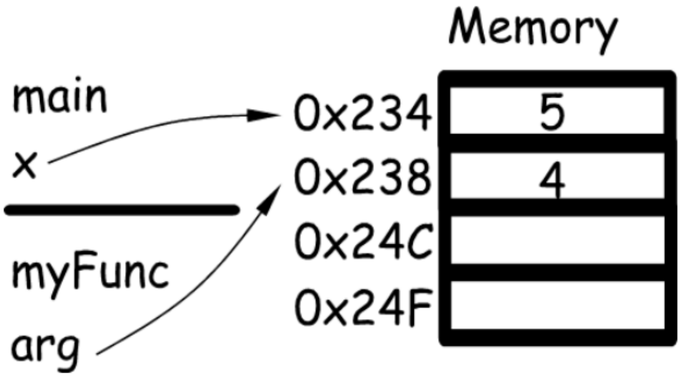
In main: myFunc(x);

- We call the function **myFunc** and pass it the value of **x**
- **myFunc** allocates a **new** memory cell for its formal parameter **arg**
- The value 5 (a **copy** of the value in **x**) is placed in **arg**



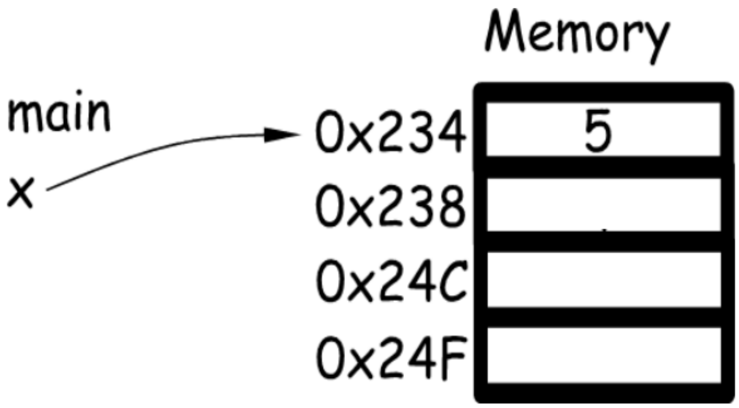
In myFunc: arg = 4;

- In **myFunc**, **arg** is assigned the value of 4
- This places the value 4 in the memory cell for **arg**
- This is not the same cell as **x**



In main: printf(“%d\n”, x);

- Back in **main**, when we print out **x**, the value it points to is still 5.



Arguments passed by Reference

- **What if we want our changes to the value in the function to affect the value in the main function?**
- We can accomplish that by ***passing the address of a variable*** as argument to a function and manipulate that variable inside the function.

Arguments passed by Reference

- In the **formal parameter** list, we put a * in front of the parameter name.
  - This defines a **pointer**, which means that we will be passing the **address** of the value, rather than the value itself.
- In the function call, we put an & in front of the argument name.
  - The & tells the compiler to pass the **address** of the variable, not its value.
- When we need to access the value of the argument in the function, we put a * in front of the variable name
  - This * tells the compiler to access the value pointed to by the address in the variable.

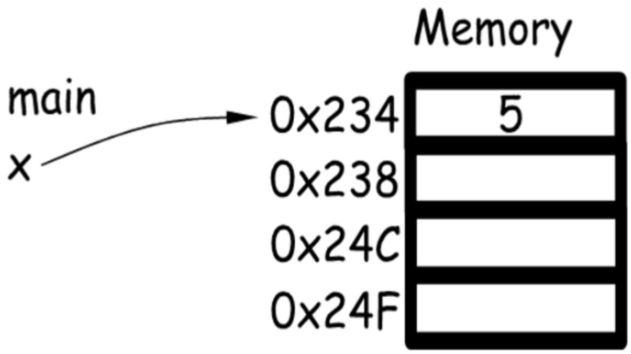
Example with pass by reference

```
void myFunc(int* arg);
int main(void){
   int x = 5;
   myFunc(&x);
   printf(“%d\n”, x);
}
void myFunc(int* arg){
   *arg = 4;
}
```

```
/* Output */
4
```

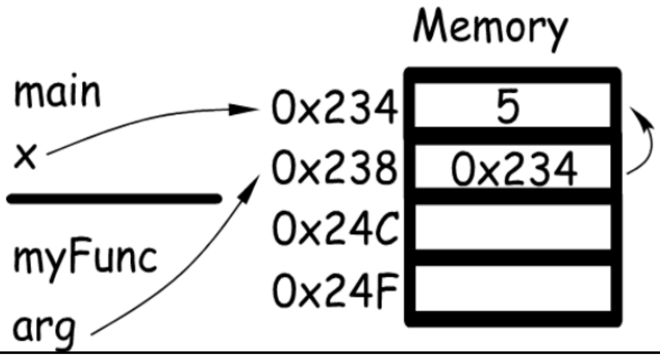
In main: int x = 5;

- ❖ In main, **x** is assigned the value 5.
- ❖ The address of the memory cell for **x** is 0x234 The value of **&x** is 0x234



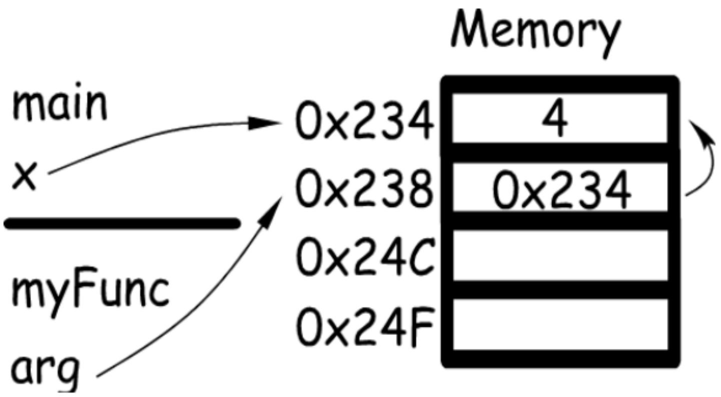
In main: myFunc(&x);

- ❖ When we call **myFunc**, we pass it **&x**, the address of **x**.
- ❖ This value is stored in the memory cell for arg. **arg == 0x234, *arg == 5**



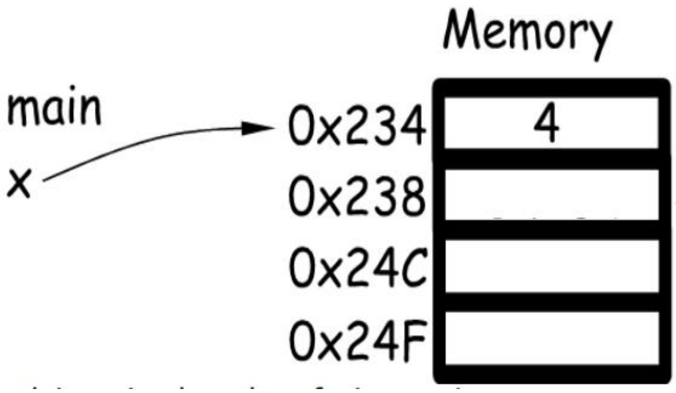
In myFunc: *arg = 4;

- When we set the value of ***arg**, we are setting the value pointed to by **arg** → the value at 0x234

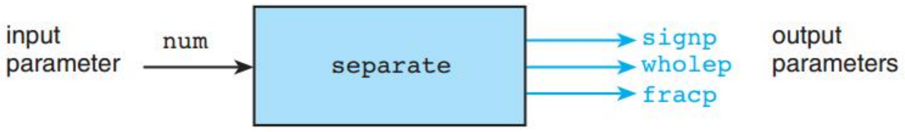


In main: printf("%d\n", x);

- Back in **main**, the value of **x** is now 4



Example: separate function



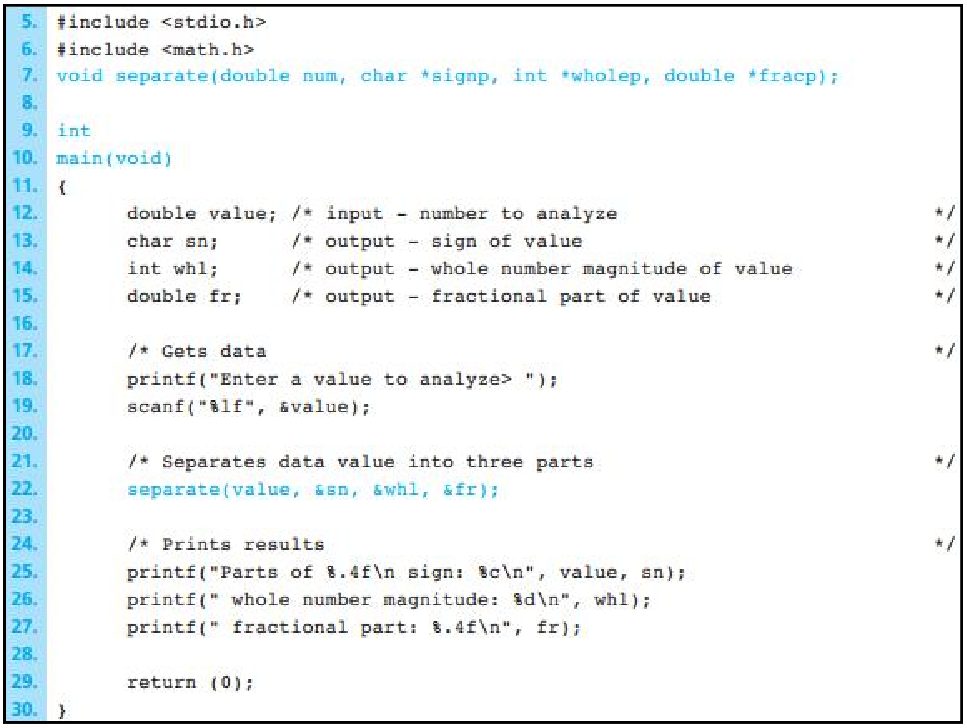
Enter a value to analyze> **35.817**

Parts of 35.8170

sign: **+**

whole number magnitude: **35**

fractional part: **0.8170**



```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);
}
```

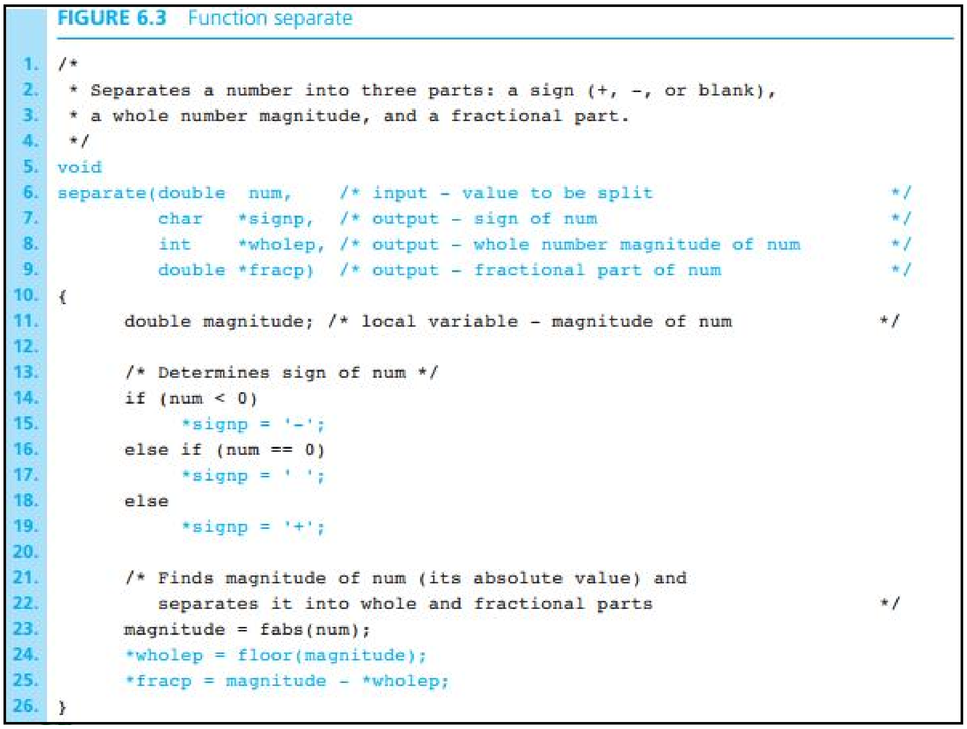
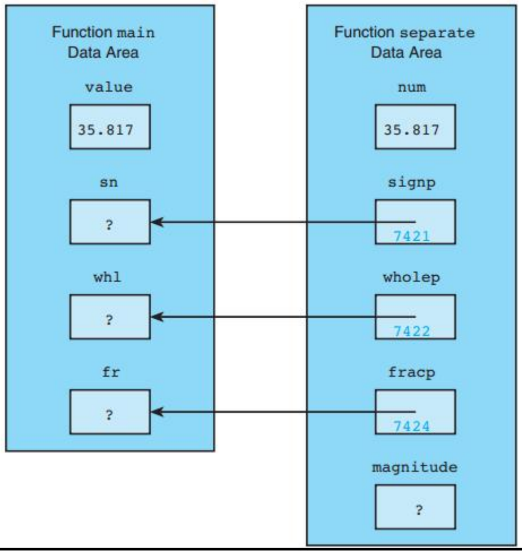


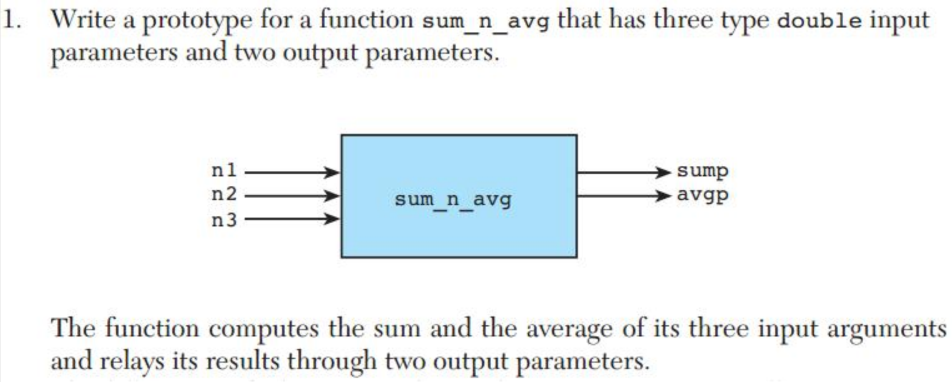
FIGURE 6.3 Function separate

```
/*
 * Separates a number into three parts: a sign (+, -, or blank),
 * a whole number magnitude, and a fractional part.
 */
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;
}
```

Parameter Correspondence for separate(value, &sn, &whl, &fr);



Self-Check



1. Write a prototype for a function `sum_n_avg` that has three type `double` input parameters and two output parameters.

n1
n2
n3
sum_n_avg
sump
avgp

The function computes the sum and the average of its three input arguments and relays its results through two output parameters.

Scope of Names

- ❖ The scope of a name refers to the region of a program where a particular meaning of a name is visible or can be referenced.
- ❖ **#define** variables scope begins at their definition and ends at the end of the source file. All functions can "see" these variables.

Scope of Names

- ❖ The scope of the name of a function begins with the function prototype and ends with the end of the source file.
- ❖ All formal parameter names and local variables are visible only from their declarations to the closing brace of the function in which they are declared.

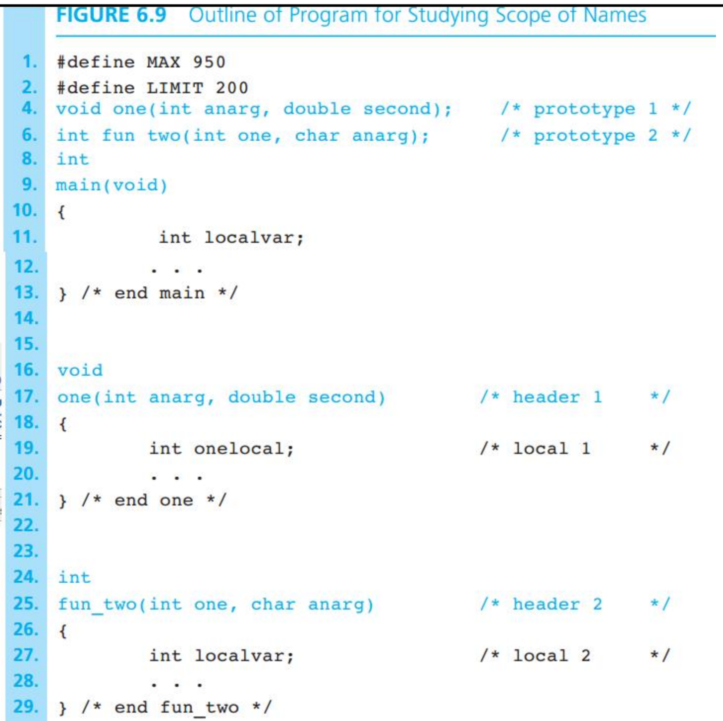


FIGURE 6.9 Outline of Program for Studying Scope of Names

```
#define MAX 950
#define LIMIT 200
void one(int anarg, double second);      /* prototype 1 */
int fun two(int one, char anarg);        /* prototype 2 */
int
main(void)
{
          int localvar;
         . . .
} /* end main */


void
one(int anarg, double second)         /* header 1     */
{
         int onelocal;                /* local 1      */
         . . .
} /* end one */


int
fun_two(int one, char anarg)          /* header 2     */
{
         int localvar;                /* local 2      */
         . . .
} /* end fun_two */
```

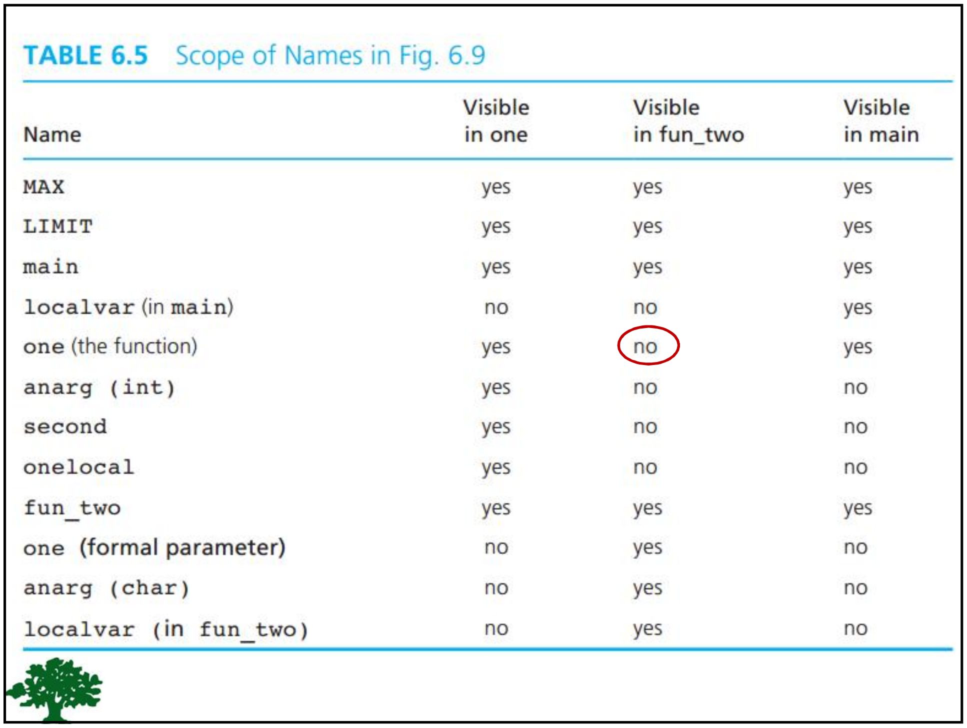


TABLE 6.5 Scope of Names in Fig. 6.9

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