Introduction to Computers & Programming

Comp 1330/ First Semester 2024/2025

Instructor: Saif Harbia

Faculty of Engineering and Technology
Department of Computer Science

Chapter 06

Pointers and Modular Programming

Chapter Objectives:

1. Learn about **pointers** and **indirect addressing**.
2. Read from input files and write to output files using **file pointers**.
3. Understand the differences between **call-by-value** and **call-by-reference.'**
4. How to write functions that can **return multiple outputs.**
5. Modularize a program system
6. Document the flow of information using structure charts

And more….

6.1 POINTERS AND THE INDIRECTION OPERATOR

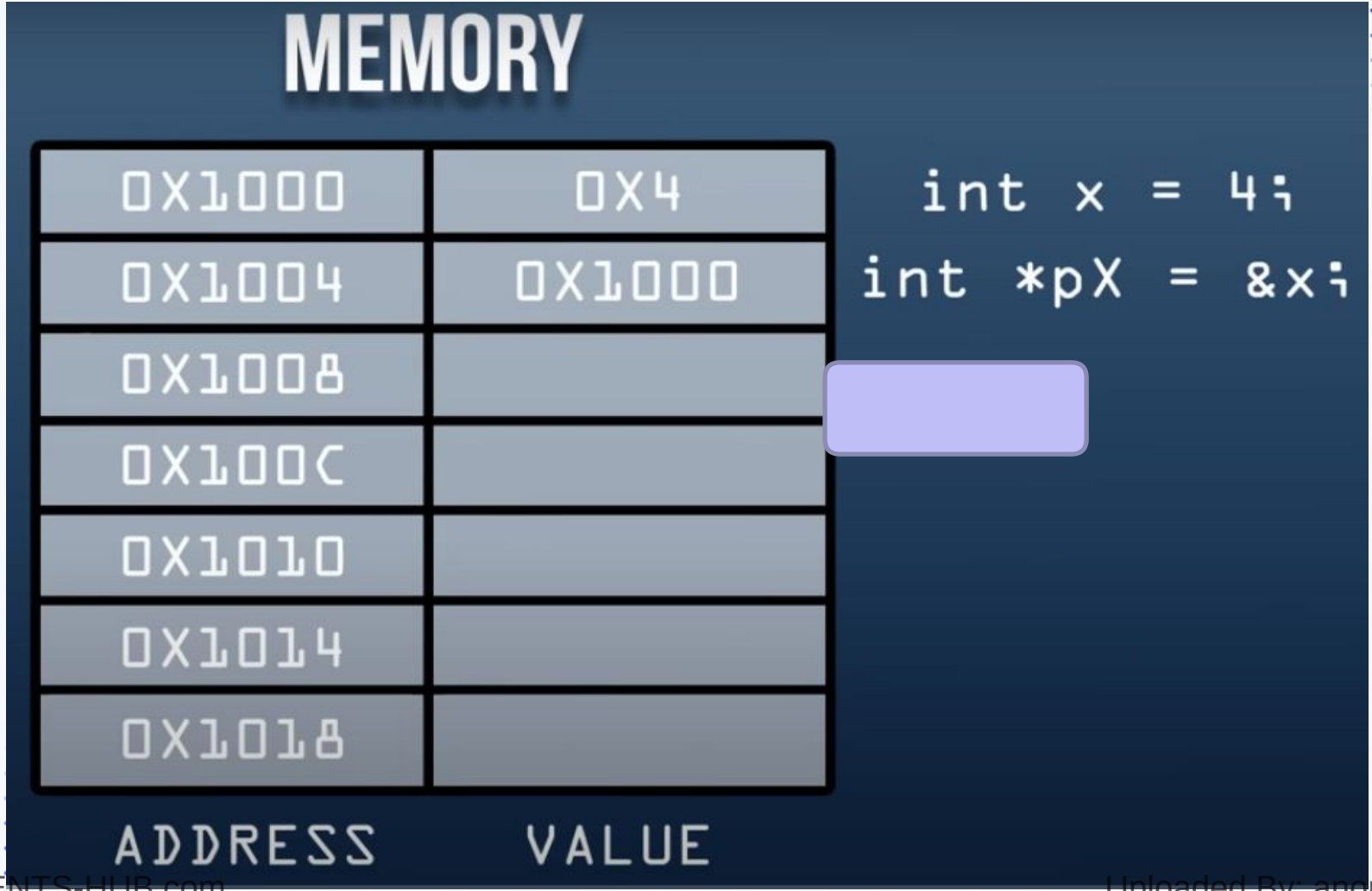
➢ The declaration: **float x;**

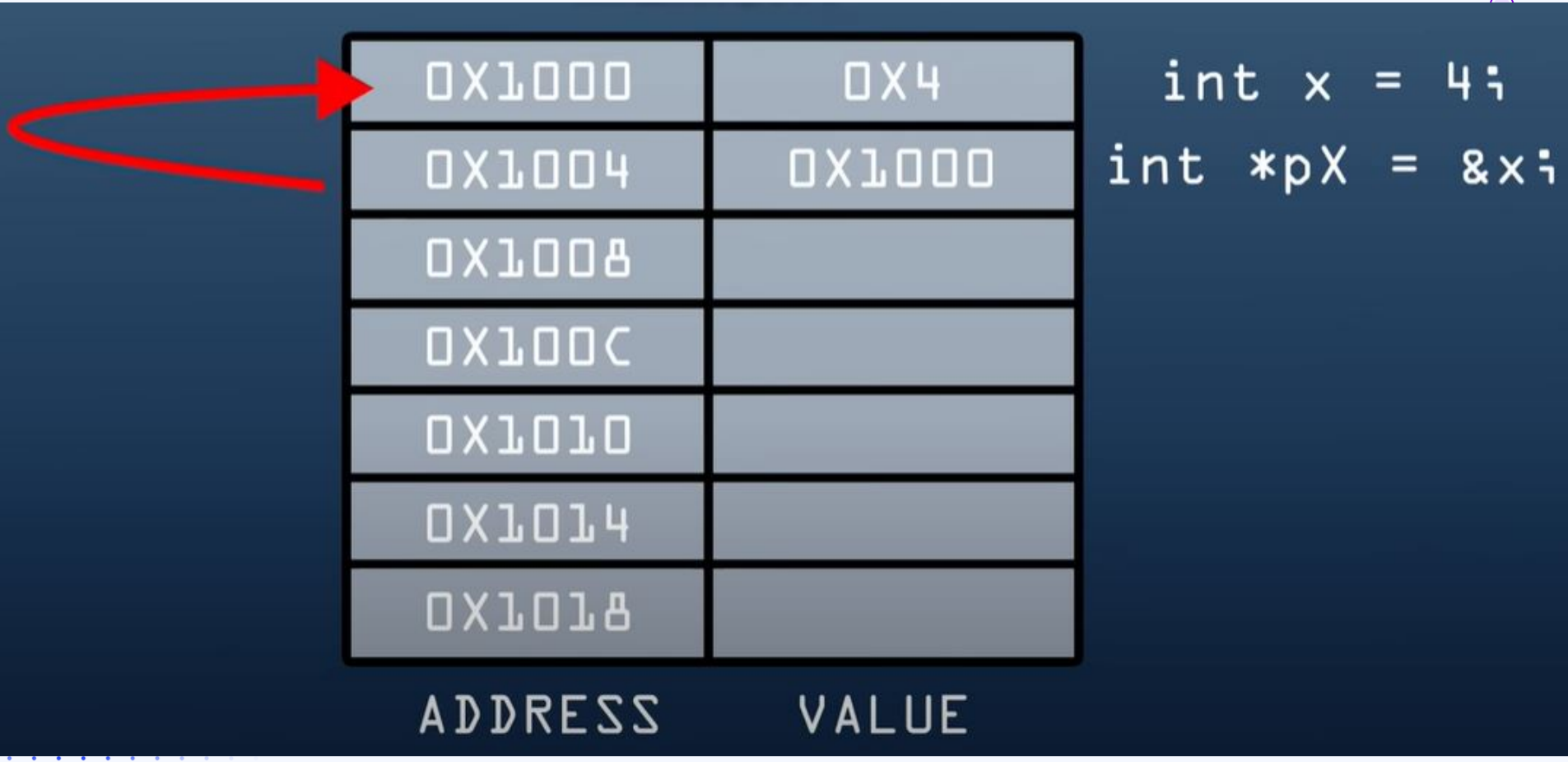
Identifies **x** as a **variable** of type **float**.

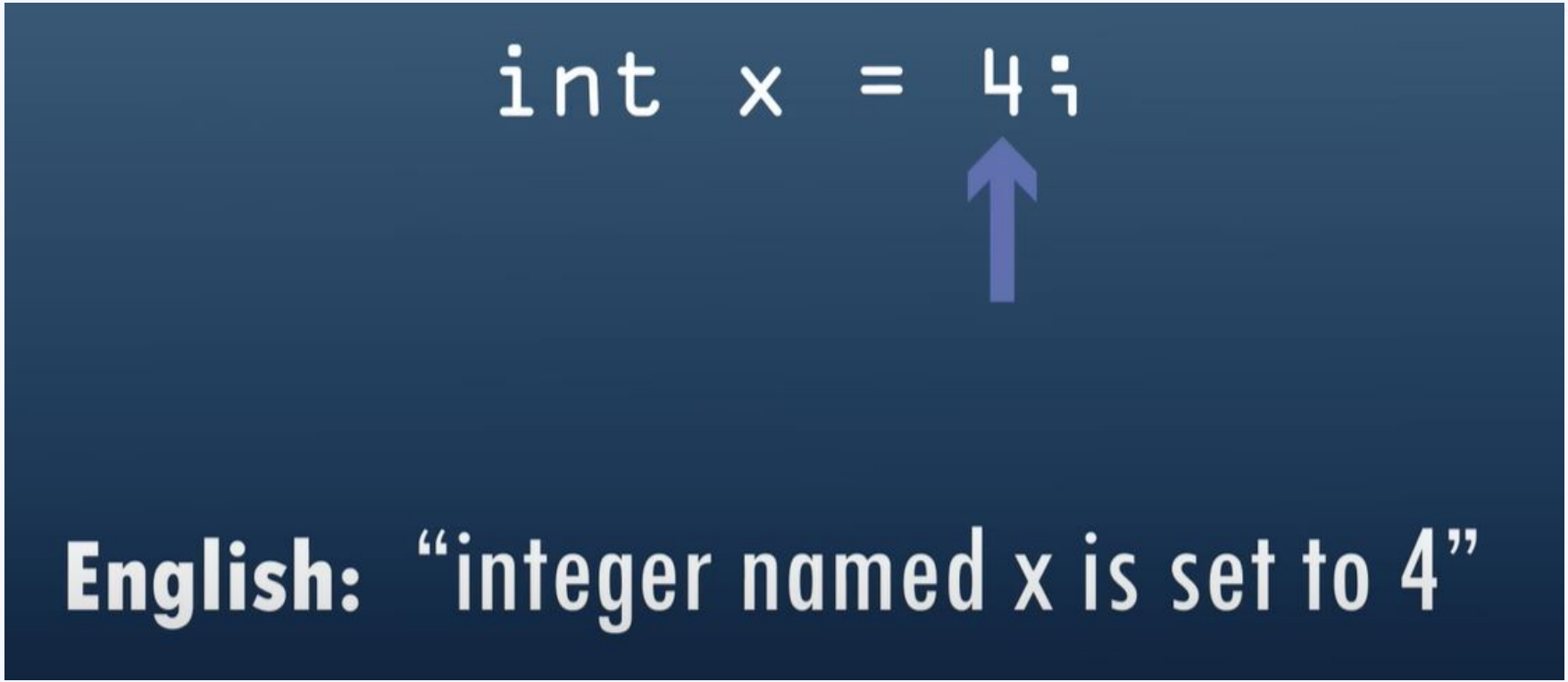
➢ The declaration: **float *p;** **OR** **float* p;**

Identifies **p** as a **pointer variable** of type "**pointer to float .**"

➢ This means that we can store the memory address of a type float variable in **p**. (1).







```
int * pX = &x;
```

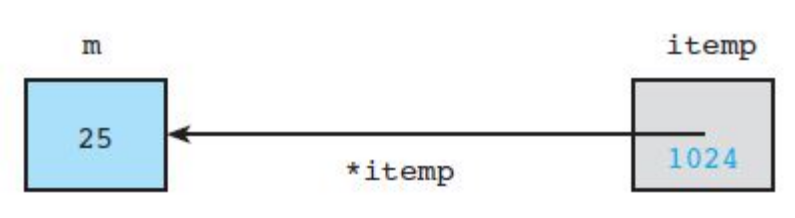
English: “integer pointer named pX is set to the address of x”

➢ **Example**:

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

- Allocate storage for an **int** variable **m** and a **pointer** variable **itemp**.
- Stores the memory address of **m** in pointer **itemp**, It applies the unary address of operator **&** to variable **m** to get its address which is then stored in **itemp** . (1)

Figure 6.1



INDIRECT REFERENCE

- When the unary indirection operator * is applied to a pointer variable, it has the effect of **following the pointer** referenced by its operand.

- This provides an **indirect reference (**also called **dereferencing)** to the cell that is selected by the **pointer** variable.

TABLE 6.1 References with Pointers

Reference	Cell referenced	Cell Type (Value)
itemp	gray shaded cell	pointer (1024)
*itemp	cell in color	int (25)

***itemp is equal to m**

```
int * pX = &x;
int y = *pX;
```

English: “integer named y is set to the thing pointed to by pX”

➢ EXAMPLE 6.2:

int m = 20; => stores 20 in the variable **m.**

***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 value currently stored in **m** (1)

POINTERS TO FILES

- To use files in this way, we must declare pointer variables of type **FILE ***.

  - **FILE *inp;** /* pointer to input file */
  - **FILE *outp;** /* pointer to output file */ (1)

- The operating system must prepare a file for input or output before permitting access. (2)

  - **inp = fopen("distance.txt", "r");** (3)
  - **outp = fopen("distout.txt", "w");**

- **"r" => read (scan) data from the file opened.**
- **"w" => write to distout.txt.**
- **outp** is initialized as an output file pointer(4)

POINTERS TO FILES

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

➢ Function **fscanf** must first be given an input file pointer like **inp** (1)

➢ **fprintf** differs from function **printf** only in its requirement of an output file pointer like **outp** as its first argument.

➢ Like **scanf** , function **fscanf** returns either the number of items read or a negative value (**EOF**) if the end of file character is detected.

- **fclose(inp);** => closes input file (2)
- **fclose(outp);** => closes output file

- **Figure 6.2**

➢ Types of functions with parameters

Functions without parameters and return values

```
void welcomeMsg(){
   printf("Welcome");
}
```

Functions with parameters and no return values

```
void sum(int x, int y){
   int sum = x + y;
   printf("%d", sum);
}
```

```
int sum(int x, int y){
   int sum = x + y;
   return sum;
}
```

Functions with parameters and return values

6.2 FUNCTIONS WITH OUTPUT PARAMETERS

- When a function call executes, the computer allocates memory space in the function data area for each formal parameter.

- The value of each actual parameter is stored in the memory cell allocated to its corresponding formal parameter

```
int result = Sum(x,y);        //function call (actual parameters)
int  Sum(int num1, int num2);    //function definition (formal parameters)
```

- **Or**, we can use the **address of operator (&)** to store the actual parameter's address instead of its value.

- Next, we discuss how a function uses **pointers** and the **indirection operator (*)** to return results to the function that calls it.

➢ **Figure 6.5**

- In our previous examples, all the formal parameters of a function represent inputs to the function from the calling function.

- In function separate, only the first formal parameter, **num** , is an **input.**

- the other three formal parameters— **signp , wholep** , and **fracp** — are **output parameters** (1)

- **Output parameters** are declared as **pointers**.

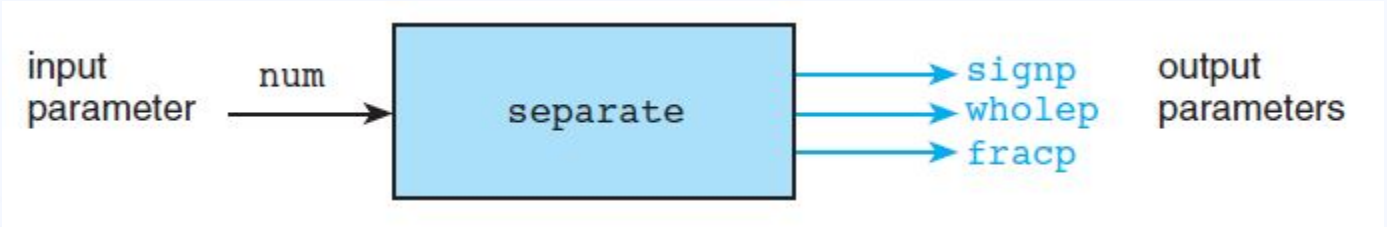


Figure 6.4

Call-by-value vs Call-by reference

Call-by-value

Call-by reference

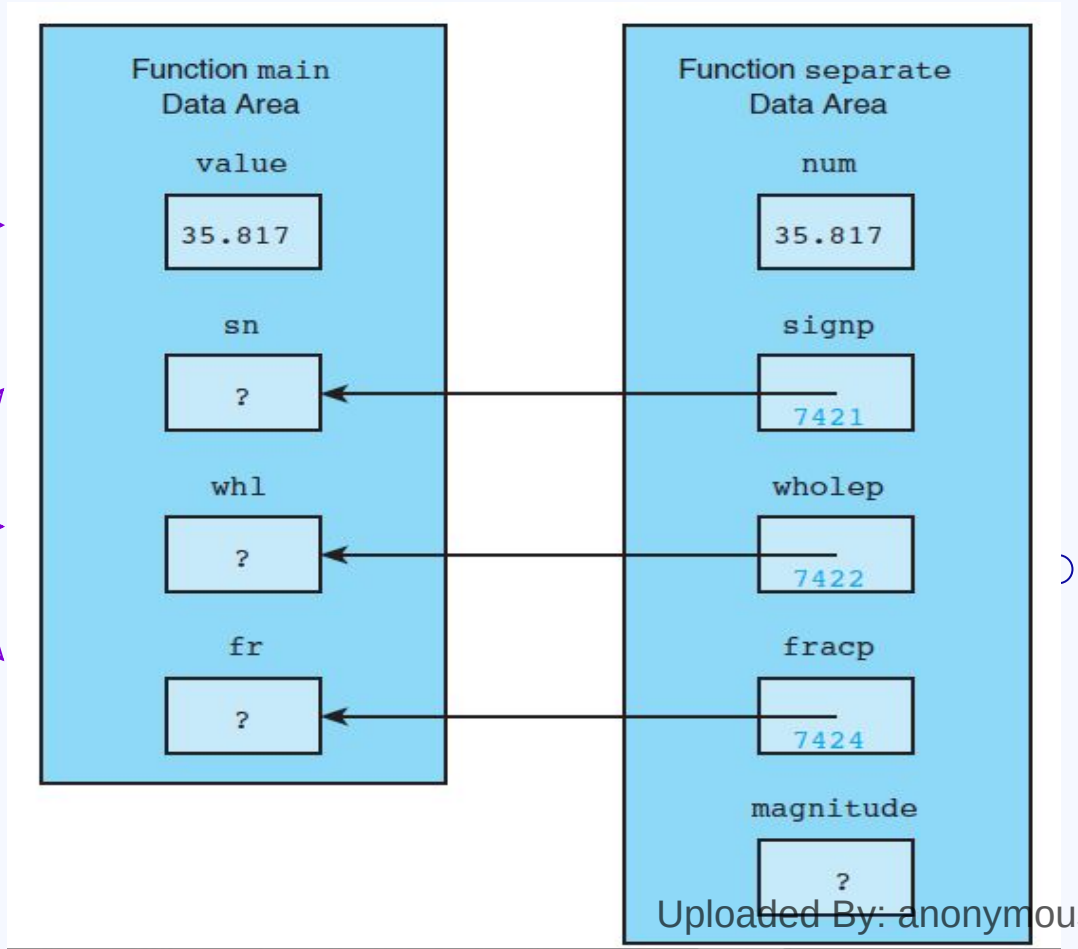


Figure 6.6

- The values of the actual output arguments in the call to separate are useless.
- These values are also of data types that do not match the types of the corresponding formal parameters
- In general if a reference **x** is of type " **any-type** ," the reference **&x** is of type "**pointer to any-type** ," that is, " **any-type *** ."

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)`

- ***signp = '+';** => follows the pointer in **signp** to the cell that function **main** calls **sn** and stores in it the character '+'.

- ***wholep = floor(magnitude);** => follows the pointer in **wholep** to the cell called **whl** by **main** and stores the integer 35 there.

- ***fracp = magnitude - *wholep;** => uses two indirect references:

1. One accesses the value in main 's local variable **whl** through the pointer **wholep.**

2. The other accesses **fr** of main through the pointer **fracp** to give the final output argument the value 0.817 .

MEANINGS OF * SYMBOL

1. We studied its use as the binary operator meaning **multiplication**.

2. **The * ’s in the declarations of the function’s formal parameters are part of the names of the parameters’ data types. These * ’s should be read as “pointer to.”** **char *signp;** (1)

3. **The * has a completely different meaning when it is used as the unary indirection operator in the function body. Here it means “follow the pointer.”**

Thus, when used in a reference,

***signp means follow the pointer in signp**

6.3 MULTIPLE CALLS TO A FUNCTION WITH INPUT/OUTPUT PARAMETERS

- **Example 6.4, p.328**
- **Figure 6.7** (1)
- **TABLE 6.3 p.330** Trace of Program to Sort Three Numbers
- **Data Area after execution of:**

temp = *smp

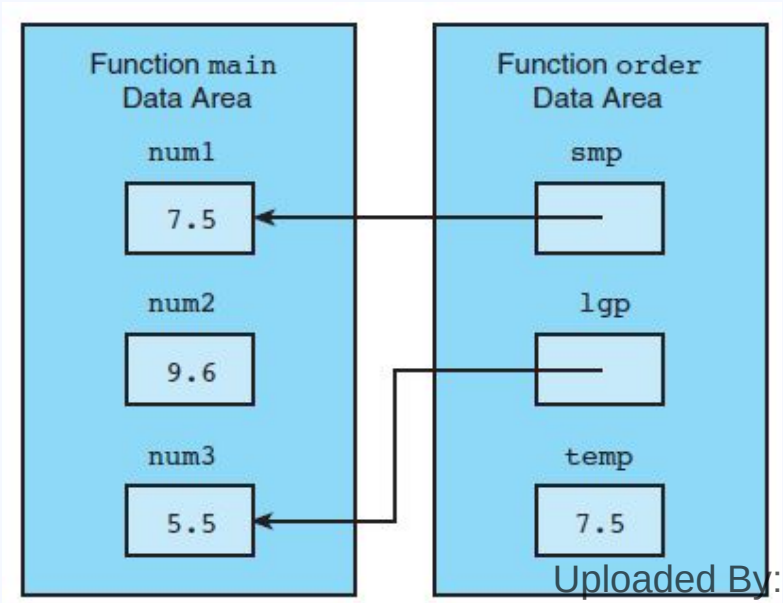


Figure 6.8

6.3 MULTIPLE CALLS TO A FUNCTION WITH INPUT/OUTPUT PARAMETERS

- **TABLE 6.4 p.332** Different Kinds of Function Subprograms (1).

- Although all the kinds of functions in Table 6.4 are useful in developing program systems, we recommend that you use the first kind whenever it is possible to do so. (2).

6.4 SCOPE OF NAMES

- **The scope of a name**: the region of a program where a particular meaning of a name is visible or can be referenced.

- **Constant macros**: their scope begins at their definition and continues to the end of the source file. This means that all functions can access them.

- **Function subprogram:** its scope begins with its prototype and continues to the end of the source file.

- **Formal parameters and local variables** are visible only from their declaration to the closing brace of the function in which they are declared

- **Figure 6.9 => (See Table 6.5 p336)**

6.5 FORMAL OUTPUT PARAMETERS AS ACTUAL ARGUMENTS

- Sometimes a function needs to pass its own output parameter as an argument when it calls another function.

- **Figure 6.10** (1)

- In all other calls to **scanf** , we applied the **address-of operator &** to each variable to be filled. However, because **nump** and **denomp** store addresses, we can use them directly in the call to **scanf** :

```
status = scanf("%d %c%d", nump, &slash, denomp);
```

6.5 FORMAL OUTPUT PARAMETERS AS ACTUAL ARGUMENTS

Figure 6.11: data areas for **scan_fraction** and the function calling it.

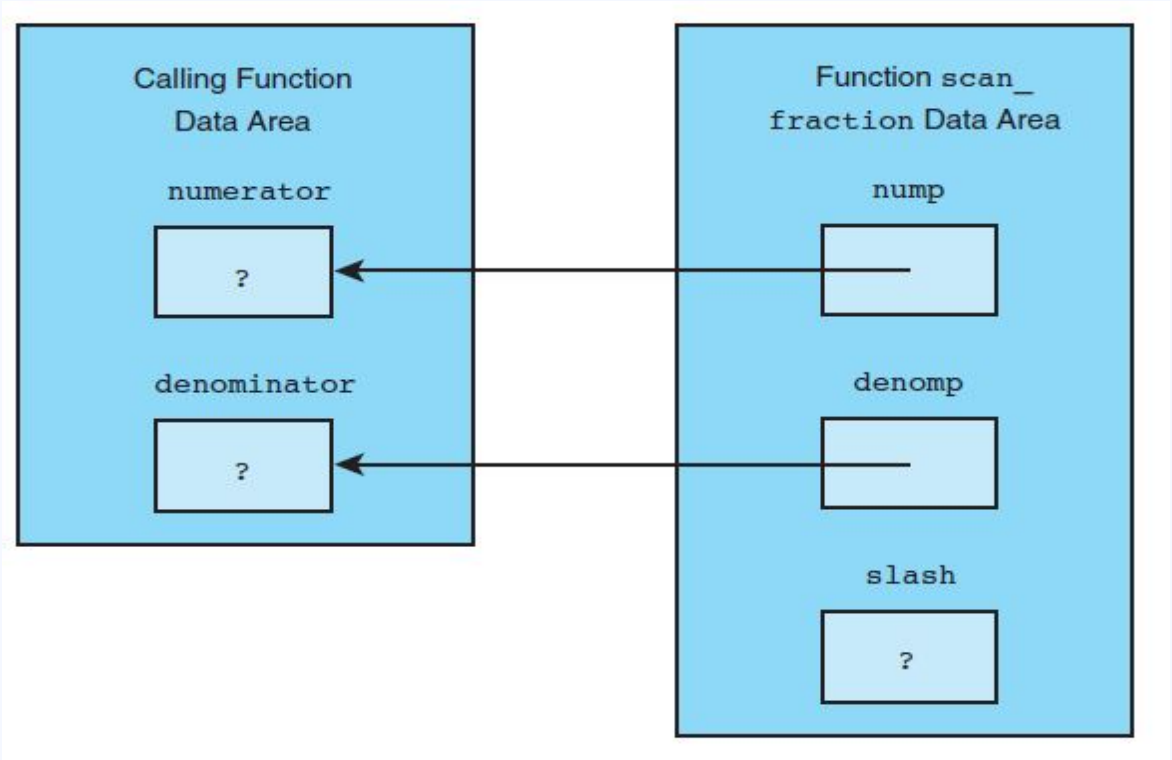


Table 6.6 p.339 gives you guidelines for function arguments of type int , double , and char

CASE STUDY (Homework)
P.347 - 355

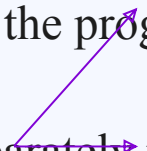
Arithmetic with Common Fractions

6.7 DEBUGGING AND TESTING A PROGRAM SYSTEM

- If we keep each function to a manageable size, the likelihood of error increases much more slowly. It is also easier to read and test each function.
- **A unit test** is a preliminary test of a single function, performed independently of the complete program system, to locate and correct errors more easily. (1)

Testing Types:

- **Top-down testing**: testing a program by using **stubs** to trace the call sequence and verify the correctness of the program's control flow.



- **Bottom-up Testing**: separately testing individual functions before inserting them in a program system.

- **System integration tests**: Tests of the entire system

DEBUGGING TIPS FOR PROGRAM SYSTEMS

1. Carefully **document** each function parameter and local variable using comments as you write the code. Also, describe the function's purpose using comments.
2. Create a **trace of execution** by displaying the function name as you enter it.
3. **Trace or display the values of all input and input/output** parameters upon entry to a function.
4. **Trace or display the values of all function outputs** after returning from a function. Verify that these values are correct by hand computation.
5. Make sure you **declare all input/output and output parameters as pointer types.**
6. Make sure that a function **stub assigns a value** to the variable pointed to by each output parameter.

Refernces

Problem Solving and Program Design in C, 7th Ed., by Jeri R. Hanly and Elliot B. Koffman

Pointers Explained:
https://www.youtube.com/watch?v=2ybLD6_2gKM&list=WL&index=35&t=25s