C Programming for Engineers

Arrays & Pointers

ICEN 360– Spring 2017

Prof. Dola Saha

Classroom Assignment

- Matrix Addition/Subtraction – two matrices should have same number of rows and columns.

$$\mathbf{A}+\mathbf{B}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \cdots & a_{1n}+b_{1n} \\ a_{21}+b_{21} & a_{22}+b_{22} & \cdots & a_{2n}+b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}+b_{m1} & a_{m2}+b_{m2} & \cdots & a_{mn}+b_{mn} \end{bmatrix}$$

Addition

$$\begin{bmatrix} 1 & 3 \\ 1 & 0 \\ 1 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 7 & 5 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1+0 & 3+0 \\ 1+7 & 0+5 \\ 1+2 & 2+1 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 8 & 5 \\ 3 & 3 \end{bmatrix}$$

Subtraction

$$\begin{bmatrix} 1 & 3 \\ 1 & 0 \\ 1 & 2 \end{bmatrix} - \begin{bmatrix} 0 & 0 \\ 7 & 5 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1-0 & 3-0 \\ 1-7 & 0-5 \\ 1-2 & 2-1 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ -6 & -5 \\ -1 & 1 \end{bmatrix}$$

https://en.wikipedia.org/wiki/Matrix_addition

Variable Length Array

- In early versions of C, all arrays had constant size.
- If size is unknown at compilation time
  - Use dynamic memory allocation with `malloc`

- The C standard allows a variable-length array
  - An array whose length, or size, is defined in terms of an expression evaluated at execution time.

Variable Length Array Code (1)

```
1  // Fig. 6.23: fig06_23.c
2  // Using variable-length arrays in C99
3  #include <stdio.h>
4
5  // function prototypes
6  void print1DArray(size_t size, int array[size]);
7  void print2DArray(int row, int col, int array[row][col]);
8
9  int main(void)
10 {
11    printf("%s", "Enter size of a one-dimensional array: ");
12    int arraySize; // size of 1-D array
13    scanf("%d", &arraySize);
14
15    int array[arraySize]; // declare 1-D variable-length array
16
17    printf("%s", "Enter number of rows and columns in a 2-D array: ");
18    int row1, col1; // number of rows and columns in a 2-D array
19    scanf("%d %d", &row1, &col1);
20
21    int array2D1[row1][col1]; // declare 2-D variable-length array
22
```

Variable Length Array Code (2)

```
23       printf("%s",
24          "Enter number of rows and columns in another 2-D array: ");
25       int row2, col2; // number of rows and columns in another 2-D array
26       scanf("%d %d", &row2, &col2);
27
28       int array2D2[row2][col2]; // declare 2-D variable-length array
29
30       // test sizeof operator on VLA
31       printf("\nsizeof(array) yields array size of %d bytes\n",
32          sizeof(array));
33
34       // assign elements of 1-D VLA
35       for (size_t i = 0; i < arraySize; ++i) {
36          array[i] = i * i;
37       }
38
39       // assign elements of first 2-D VLA
40       for (size_t i = 0; i < row1; ++i) {
41          for (size_t j = 0; j < col1; ++j) {
42             array2D1[i][j] = i + j;
43          }
44       }
45
```

Variable Length Array Code (3)

```
46       // assign elements of second 2-D VLA
47       for (size_t i = 0; i < row2; ++i) {
48          for (size_t j = 0; j < col2; ++j) {
49             array2D2[i][j] = i + j;
50          }
51       }
52
53       puts("\nOne-dimensional array:");
54       print1DArray(arraySize, array); // pass 1-D VLA to function
55
56       puts("\nFirst two-dimensional array:");
57       print2DArray(row1, col1, array2D1); // pass 2-D VLA to function
58
59       puts("\nSecond two-dimensional array:");
60       print2DArray(row2, col2, array2D2); // pass other 2-D VLA to function
61    }
62
63    void print1DArray(size_t size, int array[size])
64    {
65       // output contents of array
66       for (size_t i = 0; i < size; i++) {
67          printf("array[%d] = %d\n", i, array[i]);
68       }
69    }
```

Variable Length Array Code (4)

```
70
71  void print2DArray(size_t row, size_t col, int array[row][col])
72  {
73     // output contents of array
74     for (size_t i = 0; i < row; ++i) {
75        for (size_t j = 0; j < col; ++j) {
76           printf("%5d", array[i][j]);
77        }
78
79        puts("");
80     }
81  }
```

Variable Length Array Code (5)

```
Enter size of a one-dimensional array: 6
Enter number of rows and columns in a 2-D array: 2 5
Enter number of rows and columns in another 2-D array: 4 3

sizeof(array) yields array size of 24 bytes

One-dimensional array:
array[0] = 0
array[1] = 1
array[2] = 4
array[3] = 9
array[4] = 16
array[5] = 25

First two-dimensional array:
    0    1    2    3    4
    1    2    3    4    5

Second two-dimensional array:
    0    1    2
    1    2    3
    2    3    4
    3    4    5
```

Matrix Multiplication

- If A is a $n \times m$ matrix and B is a $m \times p$ matrix, then Matrix Multiplication is given by following formula

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mp} \end{pmatrix}$$

$$\mathbf{AB} = \begin{pmatrix} (\mathbf{AB})_{11} & (\mathbf{AB})_{12} & \cdots & (\mathbf{AB})_{1p} \\ (\mathbf{AB})_{21} & (\mathbf{AB})_{22} & \cdots & (\mathbf{AB})_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ (\mathbf{AB})_{n1} & (\mathbf{AB})_{n2} & \cdots & (\mathbf{AB})_{np} \end{pmatrix}$$

$$(\mathbf{AB})_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}$$

https://en.wikipedia.org/wiki/Matrix_multiplication

Matrix Multiplication - Illustrated

$$\mathbf{AB} = \begin{pmatrix} a & b & c \\ x & y & z \end{pmatrix} \begin{pmatrix} \alpha & \rho \\ \beta & \sigma \\ \gamma & \tau \end{pmatrix} = \begin{pmatrix} a\alpha + b\beta + c\gamma & a\rho + b\sigma + c\tau \\ x\alpha + y\beta + z\gamma & x\rho + y\sigma + z\tau \end{pmatrix}$$

$$\mathbf{BA} = \begin{pmatrix} \alpha & \rho \\ \beta & \sigma \\ \gamma & \tau \end{pmatrix} \begin{pmatrix} a & b & c \\ x & y & z \end{pmatrix} = \begin{pmatrix} \alpha a + \rho x & \alpha b + \rho y & \alpha c + \rho z \\ \beta a + \sigma x & \beta b + \sigma y & \beta c + \sigma z \\ \gamma a + \tau x & \gamma b + \tau y & \gamma c + \tau z \end{pmatrix}$$

Random Number Generation

- The `rand` function generates an integer between `0` and `RAND_MAX` (a symbolic constant defined in the `<stdlib.h>` header).
  - `i = rand();`
- To get a range of values, use remainder operation.
  - `i = rand()%N; // random values in {0 to N-1}`

Random Number Generation Code

```
1  // Fig. 5.11: fig05_11.c
2  // Shifted, scaled random integers produced by 1 + rand() % 6.
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main(void)
7  {
8     // loop 20 times
9     for (unsigned int i = 1; i <= 20; ++i) {
10
11        // pick random number from 1 to 6 and output it
12        printf("%10d", 1 + (rand() % 6));
13
14        // if counter is divisible by 5, begin new line of output
15        if (i % 5 == 0) {
16           puts("");
17        }
18     }
19 }
```

```
         6         6         5         5         6
         5         1         1         5         3
         6         6         2         4         2
         6         2         3         4         1
```

Pseudorandom numbers

- Function `rand` generates pseudorandom numbers.
- Calling `rand` repeatedly produces a sequence of numbers that appears to be random.
- Randomizing
  - A program conditioned to produce a different sequence of random numbers for each execution
  - Accomplished with the standard library function `srand`.
- Function `srand()` takes an unsigned integer argument and seeds function `rand()` to produce a different sequence of random numbers for each execution of the program.

Randomizing with a seed

```
1   // Fig. 5.13: fig05_13.c
2   // Randomizing the die-rolling program.
3   #include <stdlib.h>
4   #include <stdio.h>
5
6   int main(void)
7   {
8      unsigned int seed; // number used to seed the random number generator
9
10     printf("%s", "Enter seed: ");
11     scanf("%u", &seed); // note %u for unsigned int
12
13     srand(seed); // seed the random number generator
14
15     // loop 10 times
16     for (unsigned int i = 1; i <= 10; ++i) {
17
18        // pick a random number from 1 to 6 and output it
19        printf("%10d", 1 + (rand() % 6));
20
21        // if counter is divisible by 5, begin a new line of output
22        if (i % 5 == 0) {
23           puts("");
24        }
25     }
26  }
```

Randomize without providing a seed

- To randomize without entering a seed each time, use a statement like
  `srand(time(NULL));`
- The function prototype for time is in `<time.h>`.

```
// Fig. 5.14: fig05_14.c
// Simulating the game of craps.
#include <stdio.h>
#include <stdlib.h>
#include <time.h> // contains prototype for function time

// enumeration constants represent game status
enum Status { CONTINUE, WON, LOST };

int rollDice(void); // function prototype

int main(void)
{
   // randomize random number generator using current time
   srand(time(NULL));

   int myPoint; // player must make this point to win
   enum Status gameStatus; // can contain CONTINUE, WON, or LOST
   int die1 = 1 + (rand() % 6); // pick random die1 value
```

Classroom Assignment

- Use Random Number generation to assign random values to two nxm matrices (A and B), then add / subtract the matrices and print the result matrix.

What does the code do?

```
1  // ex06_18.c
2  // What does this program do?
3  #include <stdio.h>
4  #define SIZE 10
5
6  // function prototype
7  void someFunction(const int b[], size_t startIndex, size_t size);
8
9  // function main begins program execution
10 int main(void)
11 {
12    int a[SIZE] = { 8, 3, 1, 2, 6, 0, 9, 7, 4, 5 }; // initialize a
13
14    puts("Answer is:");
15    someFunction(a, 0, SIZE);
16    puts("");
17 }
18
19 // What does this function do?
20 void someFunction(const int b[], size_t startIndex, size_t size)
21 {
22    if (startIndex < size) {
23       someFunction(b, startIndex + 1, size);
24       printf("%d  ", b[startIndex]);
25    }
26 }
```

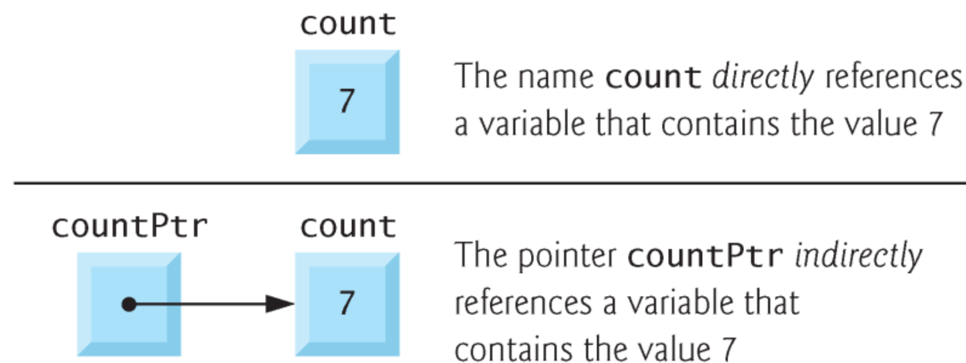
Pointers

- Pointers are variables whose values are *memory addresses*.
- A variable name *directly* references a value, and a pointer *indirectly* references a value.
- Referencing a value through a pointer is called indirection.

Declaring Pointers

- Pointers must be defined before they can be used.
- The definition
  - `int *countPtr, count;`

  specifies that variable `countPtr` is of type `int *` (i.e., a pointer to an integer).
- The variable `count` is defined to be an `int`, *not* a pointer to an `int`.



Initializing Pointers

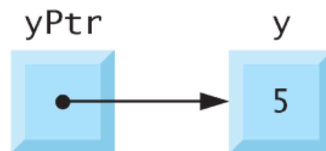
- Pointers should be initialized when they're defined or they can be assigned a value.
- A pointer may be initialized to `NULL`, `0` or an address.
- A pointer with the value `NULL` points to *nothing*.
- `NULL` is a *symbolic constant* defined in the <stddef.h> header (and several other headers, such as <stdio.h>).
- 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.

Pointer Operator

- The &, or address operator, is a unary operator that returns the address of its operand.
- Example definition
  - ```
    int y = 5;
    int *yPtr;
    ```

  the statement
  - ```
    yPtr = &y;
    ```

  assigns the *address* of the variable y to pointer variable `yPtr`.
- Variable `yPtr` is then said to "point to" y.



Graphical Representation

Memory Representation

Indirection (*) Operator

- The unary * operator, commonly referred to as the indirection operator or dereferencing operator, returns the *value* of the object to which its operand (i.e., a pointer) points.
- Example:
  - `printf("%d", *yPtr);`

  prints the value of variable that `yPtr` is pointing to

  In this case it is `y`, whose value is 5.
- Using * in this manner is called dereferencing a pointer.

Using & and *

```
3   #include <stdio.h>
4
5   int main(void)
6   {
7      int a = 7;
8      int *aPtr = &a; // set aPtr to the address of a
9
10     printf("The address of a is %p"
11            "\nThe value of aPtr is %p", &a, aPtr);
12
13     printf("\n\nThe value of a is %d"
14            "\nThe value of *aPtr is %d", a, *aPtr);
15
16     printf("\n\nShowing that * and & are complements of "
17            "each other\n&*aPtr = %p"
18            "\n*&aPtr = %p\n", &*aPtr, *&aPtr);
19  }
```

```
The address of a is 0028FEC0
The value of aPtr is 0028FEC0

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

Showing that * and & are complements of each other
&*aPtr = 0028FEC0
*&aPtr = 0028FEC0
```

Pass by value

```
1   // Fig. 7.6: fig07_06.c
2   // Cube a variable using pass-by-value.
3   #include <stdio.h>
4
5   int cubeByValue(int n); // prototype
6
7   int main(void)
8   {
9      int number = 5; // initialize number
10
11     printf("The original value of number is %d", number);
12
13     // pass number by value to cubeByValue
14     number = cubeByValue(number);
15
16     printf("\nThe new value of number is %d\n", number);
17  }
18
19  // calculate and return cube of integer argument
20  int cubeByValue(int n)
21  {
22     return n * n * n; // cube local variable n and return result
23  }
```

Pass by reference – simulating with Pointer

```
1   // Fig. 7.7: fig07_07.c
2   // Cube a variable using pass-by-reference with a pointer argument.
3
4   #include <stdio.h>
5
6   void cubeByReference(int *nPtr); // function prototype
7
8   int main(void)
9   {
10     int number = 5; // initialize number
11
12     printf("The original value of number is %d", number);
13
14     // pass address of number to cubeByReference
15     cubeByReference(&number);
16
17     printf("\nThe new value of number is %d\n", number);
18  }
19
20  // calculate cube of *nPtr; actually modifies number in main
21  void cubeByReference(int *nPtr)
22  {
23     *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
24  }
```

Pass by value (1)

Step 1: Before `main` calls `cubeByValue`:

```
int main(void)
{
    int number = 5;

    number = cubeByValue(number);
}
```

number: 5

```
int cubeByValue(int n)
{
    return n * n * n;
}
```

n: undefined

Step 2: After `cubeByValue` receives the call:

```
int main(void)
{
    int number = 5;

    number = cubeByValue(number);
}
```

number: 5

```
int cubeByValue( int n )
{
    return n * n * n;
}
```

n: 5

Pass by value (2)

Step 3: After **cubeByValue** cubes parameter **n** and before **cubeByValue** returns to **main**:

```
int main(void)
{
    int number = 5;

    number = cubeByValue(number);
}
```



```
int cubeByValue(int n)
{
    return n * n * n;
}
```


Step 4: After **cubeByValue** returns to **main** and before assigning the result to **number**:

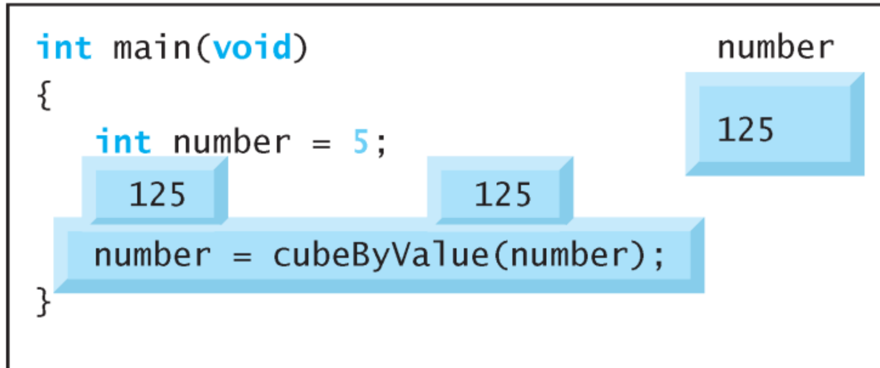
```
int main(void)
{
    int number = 5;

    number = cubeByValue(number);
}
```

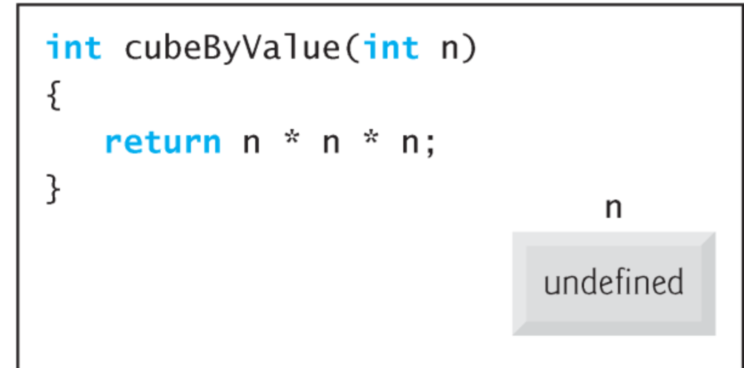


```
int cubeByValue(int n)
{
    return n * n * n;
}
```

Pass by value (3)

Step 5: After `main` completes the assignment to `number`:





Pass by reference (1)

Step 1: Before `main` calls `cubeByReference`:

```
int main(void)
{
    int number = 5;

    cubeByReference(&number);
}
```

number: 5

```
void cubeByReference(int *nPtr)
{
    *nPtr = *nPtr * *nPtr * *nPtr;
}
```

nPtr: undefined

Step 2: After `cubeByReference` receives the call and before `*nPtr` is cubed:

```
int main(void)
{
    int number = 5;

    cubeByReference(&number);
}
```

number: 5

```
void cubeByReference( int *nPtr )
{
    *nPtr = *nPtr * *nPtr * *nPtr;
}
```

nPtr

call establishes this pointer

Pass by reference (2)