Review of C/C++ for Numerical Methods (part 2)

mod .2021.03.22

Dynamic Allocation

Dynamic Allocation

Q. Does this syntax of declaring array give compile or run-time error? Explain why.

Case 1: array declaration using variable m,n

```
int m=3;
int n=3;
float array[m][n];
```

Case 2: array declaration using variable *m,n* inside a function

```
void func1(a,m,n)
{…
  float a[m][n];
  …}
```

Dynamic Allocation

- So far, we declared 1-D or 2-D arrays with fixed dimension in C/C++

```
e.g. double a[4] = { 2, 2, 3, 4 };
```

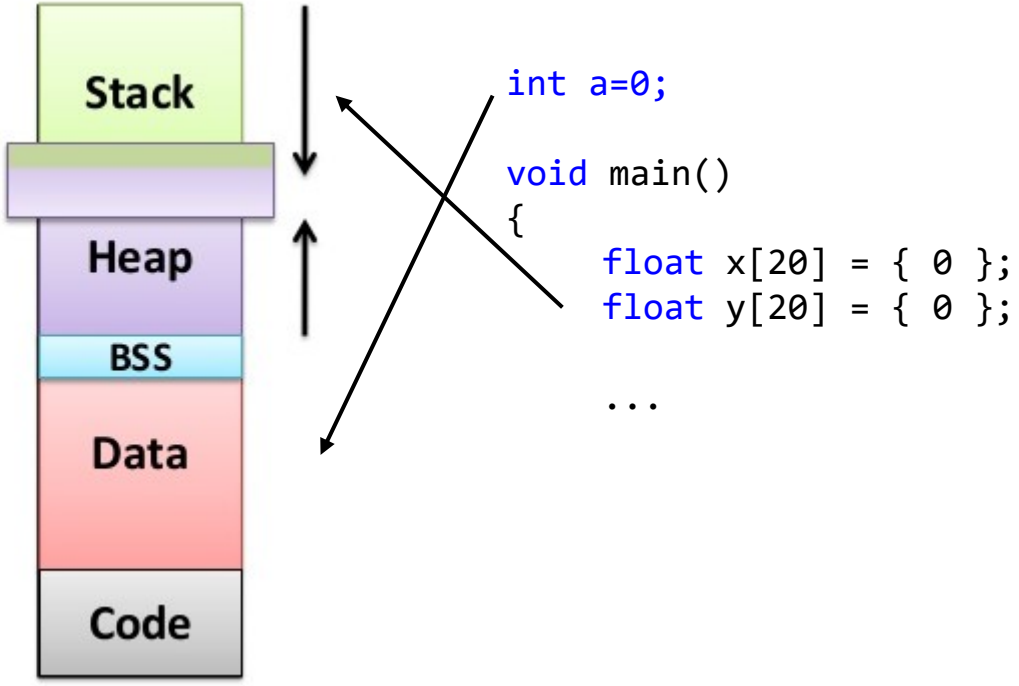
- How can the user set the size of the array (i.e., change the value of m by n) during the run-time?

- Is it possible to declare arrays without knowing its size?

Dynamic Allocation

- In the compile process, memories (address and size) for the **local** variables including the arrays are determined in the location called **'stack area'.**
- Once, the memory is set you cannot resize the memory for local variables.
- C program asks you to designate the size of memory of a vector or array in the compile process.
- So, if you declare a 1-D or 2-D array without constant dimensions, the compiler gives you an error message.

- Stack
  - automatic (default), local
  - Initialized/uninitialized
- Data
  - Global, static, extern
  - BSS: Block Started by Symbol
  - BBS: Uninitialized Data Seg.
- Code
  - program instructions
- Heap
  - malloc, calloc



Dynamic Allocation

Dynamic memory allocation.

- If we want to designate the dimension of a 1-D vector or 2-D matrix *while* the program is running, then we store the array in the memory location called the‘**heap**’.
- In this tutorial, you will practice how to construct and initialize a 1-D and 2-D array with variable dimensions using dynamic memory allocation.
- You will also learn how to make a function that creates/initializes 1-D/2-D arrays.

Cautions with Dynamic memory allocation.

- Check the size of the heap of the RAM.
- De-allocate(‘free’) memory when it is not needed.
- Do not de-allocate which is not initialized or allocated.
- Do not use the array which was de-allocated

Dynamic Allocation- Malloc(memory allocation) Function

- **Prototype**

```
void * malloc(size_t size);
```

size of memory you want to allocate

almost same as unsigned integer type

Characteristics of "void * " type (void pointer type)

- void pointer type can be assigned arbitrary types
- void pointer type can perform pointer operations
- in order to use the void pointer type properly, casting is necessary

returns a pointer to the allocated memory, or NULL if the request fails.

- **Example usage**

```
(int *)malloc(sizeof(int) * (_row));
                              // 1-D array dynamic allocation
```

: The function can not determine which data type to use, so you must set the data type

Dynamic Allocation- 1D-array Example

Dynamic Allocation- 1D

```
val = 1;
int *vecC;
// Memory allocation
vecC = (int*)malloc(sizeof(int) * (_row));


// Initialization with a value
for (i = 0; i < _row; i++)
    (vecC)[i] = val;
// Print vector
printVec(vecC, _row);
// Free allocated memory when program ends
free(vecC);

.......
```

Dynamic Allocation- 1D-array Example 2

Dynamic Allocation- 1D using functions

```
int *vecD;
// Memory allocation
createVec(&vecD, _row);
// Initilization with values
initVec(vecD, _row,1);
// Print vector
printVec(vecD, _row);
// Free allocated memory
free(vecD);
```

```
void createVec(int** _vec, int _row){
     *_vec = (int*)malloc(sizeof(int) * (_row));
}

void initVec(int* _vec, int _row, int _val)
{
     int i;
     for(i = 0; i < _row; i++)
               (_vec)[i] = _val;
}
```

*You are passing '**int *vec**' to the function '**createVec( )**' without allocating size and memory of the 1-D array. Thus, you need to pass the address of '**int *vec**' as '**&vec**'

-> the function receives it as '**int** _vec**' (2 pointer notation)

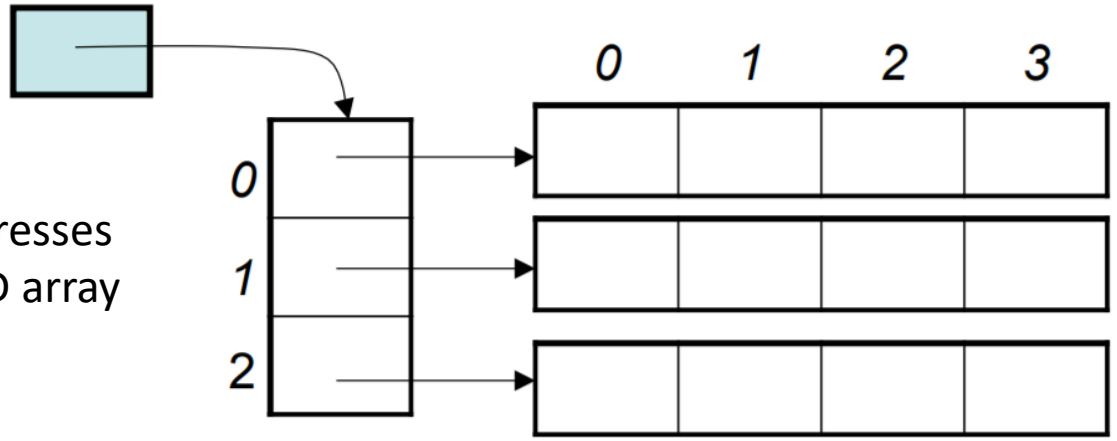
Once, the memory of the 1-D array is allocated then you can pass the name of the array to a function

Dynamic Allocation- 2D-array Example

Dynamic Allocation- 2D

```
// 1. allocate row array first
matA = (int**)malloc(sizeof(int*) * (_row));
// 2. Then, allocate column
for(int i = 0; i < _row; i++)
    (matA)[i] = (int*)malloc(sizeof(int) * (_col));
```

A 2D array is a 1D array of (references to) 1D arrays.



Stores addresses for each 1D array

Dynamic Allocation- 2D-array Example

Dynamic Allocation- 2D

```
int **matA;
int _row, _col;
...
// 1. allocate row array
matA = (int**)malloc(sizeof(int*) * (_row));
// 2. Then, allocate column
for(int i = 0; i < _row; i++)
     (matA)[i] = (int*)malloc(sizeof(int) * (_col));

// Initialize matrix
for(int i = 0; i < _row; i++){
     for(int j = 0; j < _col;j++)
          (matA)[i][j] = 1;
}
printMat(matA, _row, _col);
// Free allocated memory when program ends
free(matA);
```

Dynamic Allocation- 2D-array Example 2

Dynamic Allocation- 2D using functions

```
int **matB;
createMat(&matB,_row, _col);
initMat(matB,_row, _col,3);
printMat(matB, _row, _col);
system("pause" );
free(matA);
free(matB);
```

```
void createMat(int*** _mat, int _row, int _col)
{
     int i;
     *_mat = (int**)malloc(sizeof(int) * (_row));
     for(i = 0; i < _row; i++)
          (*_mat)[i] = (int*)malloc(sizeof(int) * (_col));
}

void initMat(int** _mat, int _row, int _col, int _val)
{
     int i, j;
     for(i = 0; i < _row; i++)
     {
          for(j = 0; j < _col;j++)
               (_mat)[i][j] = _val;
     }
}
```

Dynamic Allocation- 2D-array Example 2

Dynamic Allocation- 2D using functions

- You are passing '**int **mat**' to the function '**create_mat( )**' without allocating size and memory of the 2-D array. Thus, you need to pass the address of '**int **mat**' as '**&mat**' and the function receives it as **'int*** _mat'** (3 pointer notation)

```
createMat(&matB,_row, _col);
void createMat(int*** _mat, int _row, int _col)
```

- Notice how different '**malloc**' syntax is used in 'Main()' function and in 'create_mat()' function.
- Once, the memory of 2-D is allocated then you can pass the array to a function as

```
initMat(matB,_row, _col,3);
void initMat(int** _mat, int _row, int _col, int _val)
```

Dynamic Allocation- in C++

Dynamic Allocation- C++

```
// C++ syntax- 1D dynamic

// Dynamic Allocation
int* matC = new int[_row];

// Initialize
for (i = 0; i < _row; i++)
  matC[i] =0;

// Free allocated memory
delete[] matC;
```

```
// C++ syntax- 2D dynamic

// Dynamic Allocation
int** matC = new int*[_row];
for (i = 0; i < _row; i++)
  matC[i] = new int[_col];

// Initialize
. . .

// Free allocated memory
for (i = 0; i < _row; i++)
  delete[] matC[i];
delete[] matC;
```

Exercise

Add functions to your “NM.h, NM.cpp”

- Dynamically allocate
  - you need to check if the array is already initialized or allocated
- Initialize
  - you need to check if the array is allocated
- Print

for a 2D array, when the requested size of array is *m* by *n*

Create a short program that let the user input the size of matrix **A** and the element values. Print the matrix after the user input.

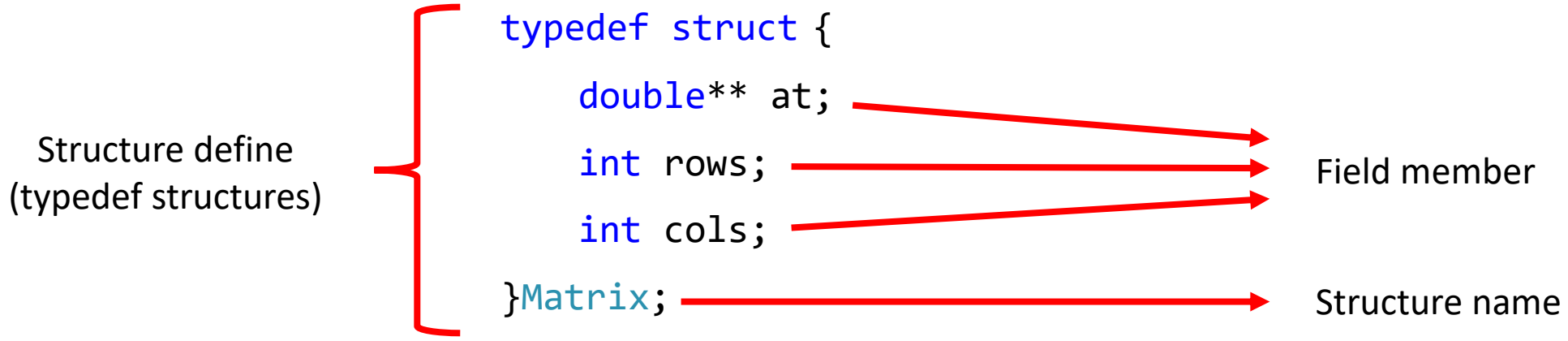
Structure

Structure

The Concept of the Structure.

- A set of related field members
- Each field member can be defined with a different data type
- Structure declaration and definition

  : Structure variables, Tagged structures, Type-defined structures



Structure

- Structure example with memory allocation

```c
typedef struct {
    double** at;
    int rows;
    int cols;
}Matrix;

int main() {
    Matrix A, B, C;

    A = createMat(3, 5);

    …
}
```

```c
Matrix createMat(int _rows, int _cols) {

    Matrix Out;

    // 1. allocate row first
    Out.at = (double**)malloc(sizeof(double*) * _rows);

    // 2. allocate column
    for (int i = 0; i < _rows; i++)
        Out.at[i] = (double*)malloc(sizeof(double) * _cols);

    // 3. Initialize matrix with values
    Out.rows = _rows;
    Out.cols = _cols;

    for (int i = 0; i < _rows; i++)
        for (int j = 0; j < _cols; j++)
            Out.at[i][j] = 0;

    return Out;
}
```

You can make a matrix of the size you want

result :

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Structure

- Main function

```c
#include "NM_tutorial2.h"

int main()
{
	Matrix A, B, C;

	A = createMat(5, 3);
	initMat(A, 10);

	B = createMat(5, 3);
	initMat(B, 5);

	C = addMat(A, B);
	printMat(C);

	free(A);
	free(B);
	free(C);

	system("PAUSE");
	return 0;
}
```

- Header file

```c
#ifndef _NM_tutorial2_H
#define _NM_tutorial2_H

#include <stdio.h>

typedef struct {
	double** at;
	int rows, cols;
}Matrix;

Matrix createMat(int _rows, int _cols);
void initMat(Matrix _mat, double _val);
void printMat(Matrix _mat);
Matrix addMat(Matrix _matA, Matrix _matB);

#endif
```

You do not need to enter the size of the matrix
because you can know the size of the matrix through the structure
(A.rows, A.cols)

Result

```
15.000000	15.000000	15.000000
15.000000	15.000000	15.000000
15.000000	15.000000	15.000000
15.000000	15.000000	15.000000
15.000000	15.000000	15.000000
```

Structure

- Header C file

```
#include "NM_tutorial2.h"

Matrix createMat(int _rows, int _cols) {
	Matrix Out;
	Out.at = (double**)malloc(sizeof(double*) * _rows);
	for (int i = 0; i < _rows; i++)
		Out.at[i] = (double*)malloc(sizeof(double) * _cols);
	Out.rows = _rows;
	Out.cols = _cols;

	for (int i = 0; i < _rows; i++)
		for (int j = 0; j < _cols; j++)
			Out.at[i][j] = 0;
	return Out;
}
void initMat(Matrix _mat, double _val) {
	for (int i = 0; i < _mat.rows; i++)
		for (int j = 0; j < _mat.cols; j++)
			_mat.at[i][j] = _val;
}
void printMat(Matrix _mat) {
	for (int i = 0; i < _mat.rows; i++) {
		for (int j = 0; j < _mat.cols; j++)
			printf("%f\t", _mat.at[i][j]);
		printf("\n");
	}
}
Matrix addMat(Matrix _matA, Matrix _matB) {
	if (_matA.rows != _matB.rows || _matA.cols != _matB.cols) {
		printf("Error : check matrix size\n");
		return createMat(0, 0);
	}

	else {
		Matrix Out = createMat(_matA.rows, _matB.cols);
		initMat(Out, 0);

		for (int i = 0; i < _matA.rows; i++)
			for (int j = 0; j < _matB.cols; j++)
				Out.at[i][j] += _matA.at[i][j] + _matB.at[i][j];
		return Out;
	}
}
```

Stop execution
if matrix size does not match