COMP0204: Introduction to Programming for Robotics and AI

Lecture 7: Dynamic Memory Allocation in C

Course lead: Dr Sophia Bano

MEng Robotics and AI

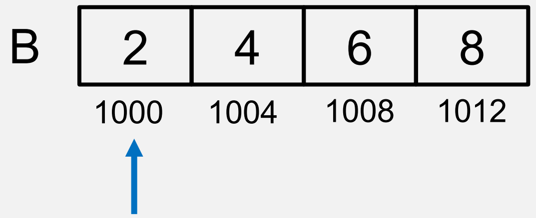
UCL Computer Science

Feedback

Recap (previous week)

- Pointers basics - declaration, referencing, dereferencing
- Pass by value vs pass by reference
- Arithmetic on Pointers
- Logical operators on Pointers
- Accessing arrays with Pointers
- 2D and multi-dimensional arrays with Pointers

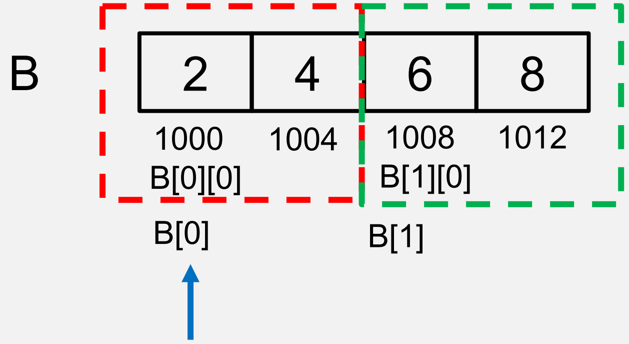
Array processing with pointers - 1D Array




```
int B[4] ={2, 4, 6, 8};
```

- B – pointer to first element of the array
- B is pointing to address 1000
- *B → *(B+0) → *(&B[0]) → B[0] = 2
- *(B+i) = B[i]

Array processing with pointers - 2D Array



int B[2][2] = {{2, 4}, {6, 8}};

Two 1D arrays

with 2 elements each

B – pointer to the first 1D array

*B → returns pointer to the first element B[0] = &B[0][0]

**B → *(*B) → *(*(B+0)) → *(&B[0][0]) → B[0][0] = 2

((B+i)+j) = B[i][j]

Array processing with pointers - 2D Array

Sum of all elements of a 2D array

```
int main(){
    int A[2][2] = {{2,4},{6,8}};
    int sum = 0;

    for (int i = 0; i < 2; i++){
        for (int j = 0; j < 2; j++)
            sum += *(*(A+i)+j);
    }
    printf("Sum = %d\n", sum);

    return 0;
}
```

int B[2][2] = {{2, 4}, {6, 8}};

Two 1D arrays

with 2 elements each

Array processing with pointers - 2D Array

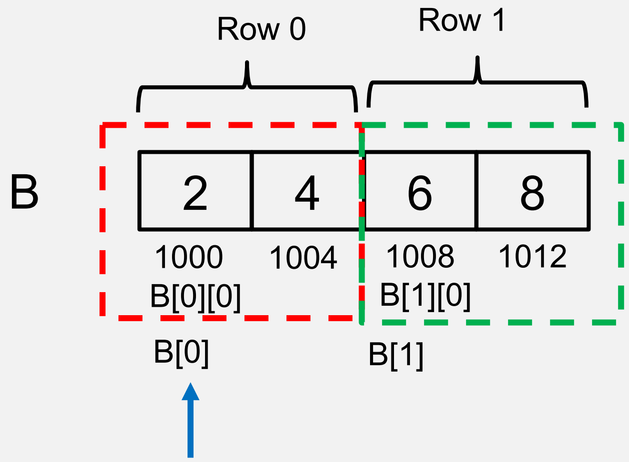
Sum of all elements of a 2D array

```
int main(){
    int A[2][2] = {{2,4},{6,8}};
    int sum = 0;

    for (int i = 0; i < 2; i++){
        for (int j = 0; j < 2; j++)
            sum += *(*(A+i)+j);
    }
    printf("Sum = %d\n", sum);

    return 0;
}
```

Pass by reference using pointers

```
void sum1(int (*A)[2], int m, int n){
    int i, j;
    int sum = 0;
    for (i = 0; i < m; i++){
        for (j = 0; j < n; j++){
            sum += *(*(A + i) + j);
        }
    }
    printf("Sum = %d\n", sum);
}

int main(){
    int A[2][2] = {{2,4},{6,8}};
    sum1(A, 2, 2);
    return 0;
}
```

Array processing with pointers - 3D Array



B – pointer to the first 1D array

int B[2][2] = {{2, 4}, {6, 8}};

Two 1D arrays

with 2 elements each

*B → returns pointer to the first element B[0] = &B[0][0]

**B → *(*B) → *(*(B+0)) → *(&B[0][0]) → B[0][0] = 2

((B+i)+j) = B[i][j]

Array processing with pointers - 3D Array

	1st 2D array				2nd 2D array			
B	2	4	6	8	10	12	14	18
	1000	1004	1008	1012	1016	1020	1024	1028

1st 2D array

2	4
6	8

2nd 2D array

10	12
14	18

```
int B[2][2][2] =      {{{2, 4}, {6, 8}}, {{10, 12}, {14, 18}}};
```

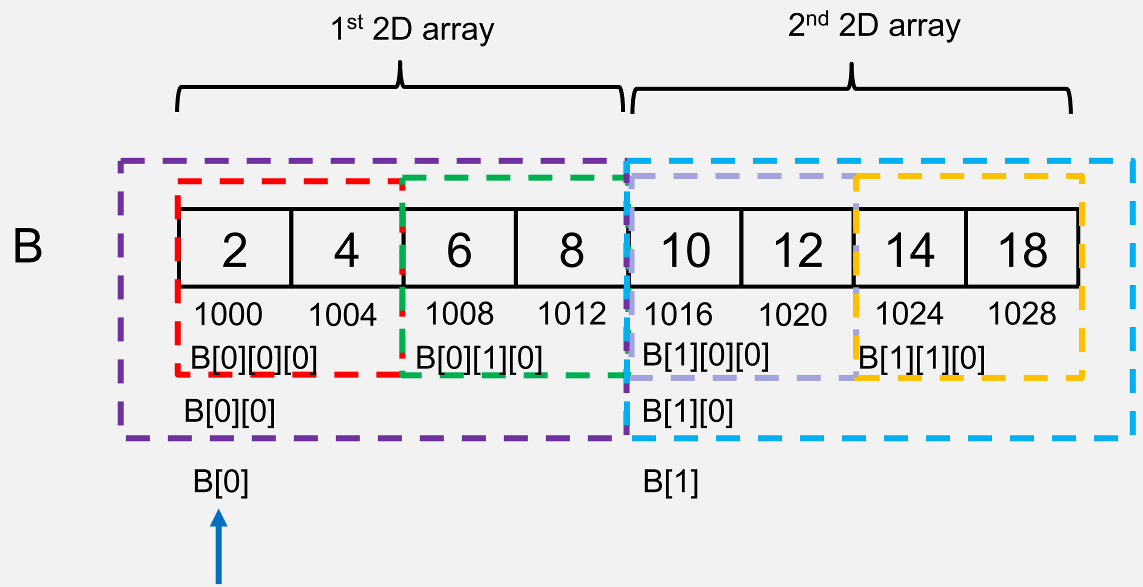
Two 2D arrays

with 2 1D arrays each

with 2 elements each

B is the pointer to the first 2D array

Array processing with pointers - 3D Array



((*(B+i)+j)+k) = B[i][j][k]

Accessing 3D array with pointers

Declare a 3D array C of size 2 x 2 x 2.

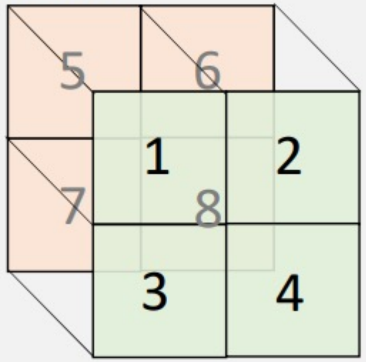
Make the memory map showing how this will appear in the memory.

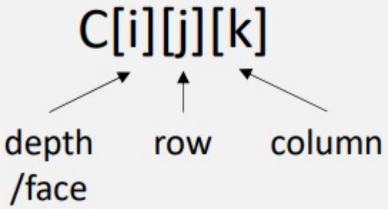
Print C, C[0], C[0][0]. What is the result? Discuss

Access C[0][0][1] and C[1][1][0] using pointers arithmetic

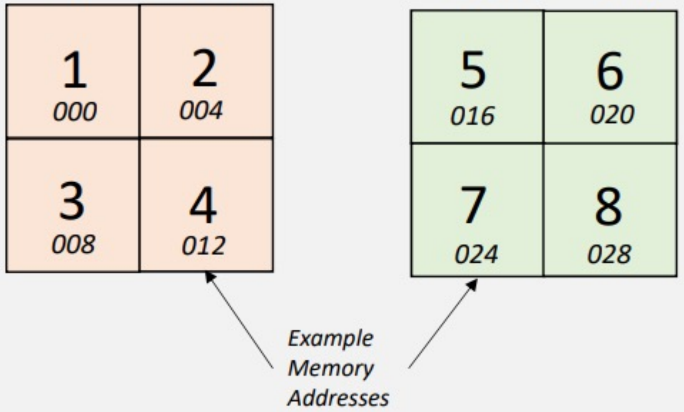
Memory Map:

3D Cube





Depth 1 – C[0] ***Depth 2 – C[1]***



Code:

```c
int main(){
    // Initialising 3D array
    int C[2][2][2] = {{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}};

    // All output the same value = the (starting) memory address...
    printf("C: %d\n", C); // ... of the whole array
    printf("C[0]: %d\n", C[0]); // ... of the first subarray
    printf("C[0][0]: %d\n", C[0][0]); // .. of the first sub-subarray

    printf("\n");

    // Access via pointer arithmetic
    printf("C[0][0][1] = %d\n", *(**C + 1)); // expect 2
    printf("C[1][1][0] = %d\n", **(*(C + 1) + 1)); // expect 7

    return 0;
}
```

Output:

```
C: 6422272
C[0]: 6422272
C[0][0]: 6422272

C[0][0][1] = 2
C[1][1][0] = 7
```

Devayan Patel

Today’s lecture

- Understanding the Memory
- Static vs Dynamic Memory
- Dynamic Memory Allocation (DMA)
- Importance of Pointers in DMA
- Diving into the built-in functions for DMA

- Brief introduction to sorting

Dynamic Memory Allocation

Static Memory Allocation

- **Memory allocated** during **compile time**
- This memory allocates is **fixed**
- This memory **cannot** be **increased** or **decreased** during run time.

```
int main(){
    int B[4] = {2,4,6,8};

    return 0;
}
```

Static Memory Allocation - Issues

- **Size is fixed at the time of declaration**
  - User cannot increase or decrease the size of the array at run time

- If the values stored in the array at run time is **less** than the size declared

  → Wastage of memory

- If the values stored in the array at run time is **more** than the size declared

  → Program may crash or misbehave

Dynamic Memory Allocation (DMA)

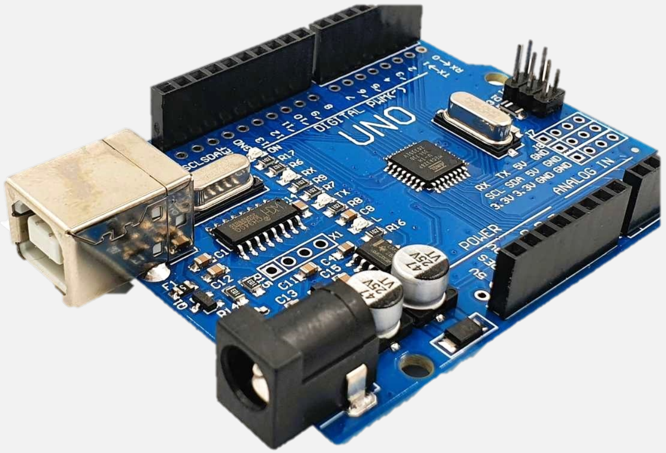
- The process of allocating memory at the time of execution (**run time**)

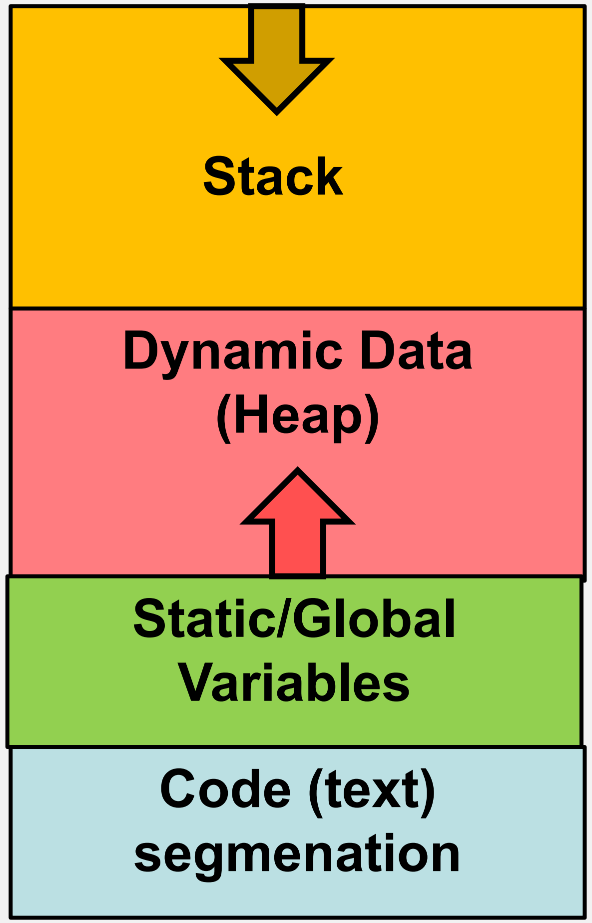
But why is DMA important and needed?

DMA Importance in Embedded Devices for Robotics

Arduino boards typically have limited static memory (RAM)

- DMA enables the use of dynamic data structures.
- DMA allows to allocate memory based on the actual data size at runtime.
- DMA can help in avoiding memory fragmentation issues.
- DMA allows for more efficient use of memory resources.





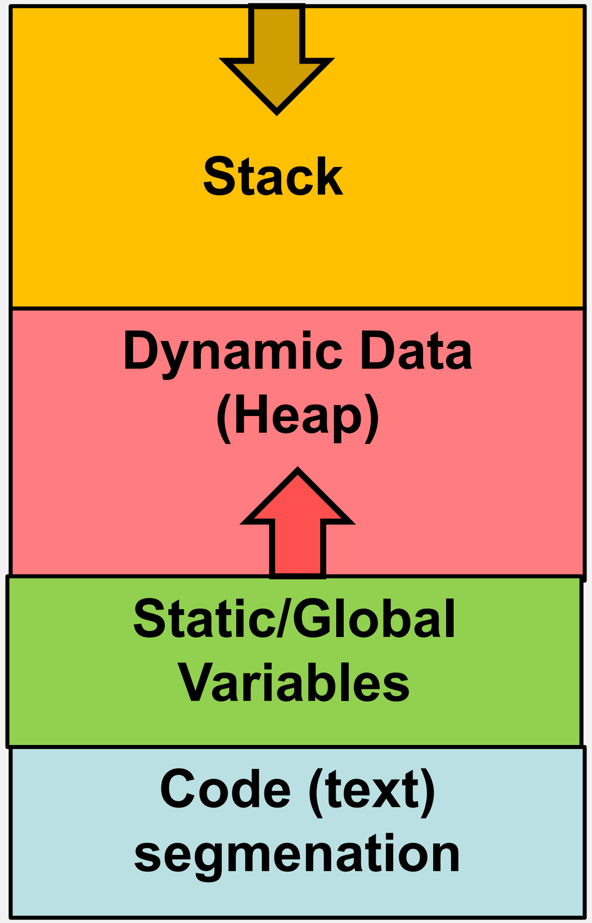
RAM Memory

- When a program creates variables, they are stored in memory at a **memory address**.

- Two main ways in which memory is allocated and organized with C:

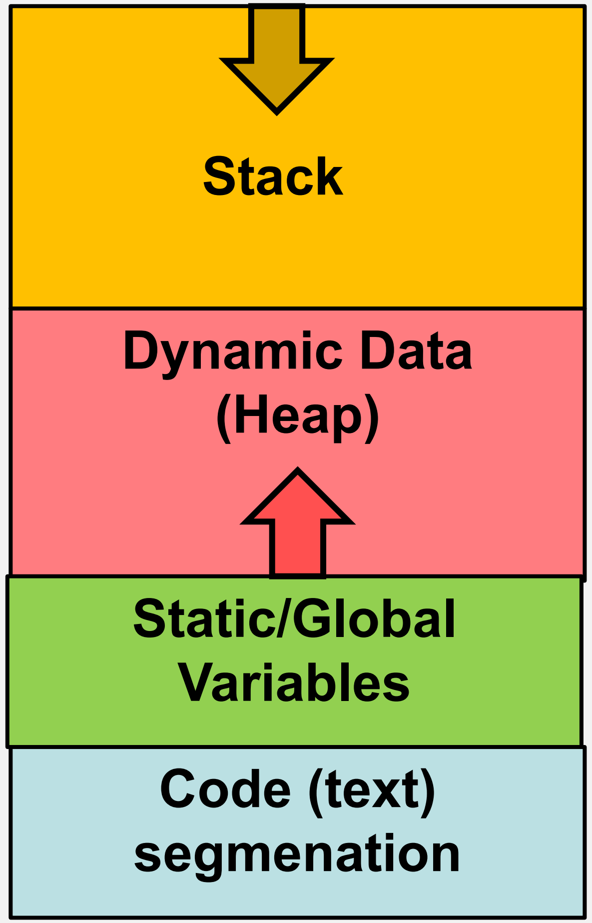
Stack vs Heap

Stack vs Heap



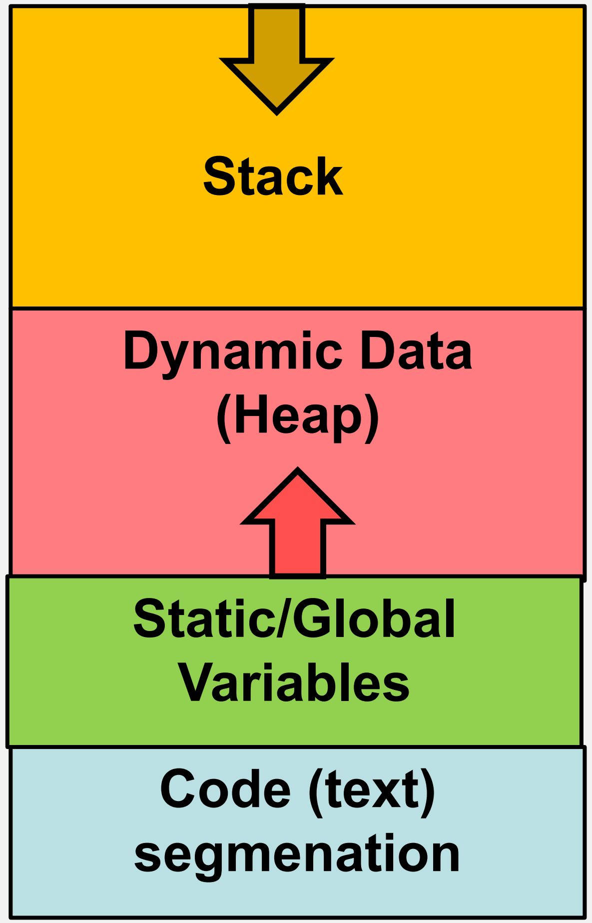
- Memory is an ordered list of locations with unique address to store data.
- Regular variables are stored on the **stack**
  - As the program needs more memory for variables, the data is stacked **in order** right next to the existing memory allocated for already existing variables.

Stack



- Region of memory that follows a **Last In, First Out (LIFO)** structure.
- Memory allocation and deallocation in the stack are **automatic and fast**.
- Used for the storage of local variables, function call information, and control flow data.
- Size of the stack is limited, and it is determined at **compile-time**.
- Memory is reclaimed automatically when a function exits.

Heap



- Region of memory used for **dynamic memory allocation**.
- Managed manually, and the **programmer is responsible** for allocating and deallocating memory.
- Used for storing **data structures whose size is not known** at compile-time, like arrays and structures.
- Memory allocation in the heap is **more flexible** than the stack, and the size can be adjusted during runtime.
- Memory in the heap must be explicitly deallocated to avoid **memory leaks**.

What do we need to access heap?

What do we need to access heap?

- **Pointers** play an important role in dynamic memory allocation.
- Allocated memory can only be accessed through **pointers**.
- There are certain built-in functions that can help in allocating or deallocating some memory space at run time.

Dynamic Memory Allocation (DMA)

- The process of allocating memory at the time of execution (run time):

Built in Functions: declared in the header file <stdlib.h>

1. malloc()
2. calloc()
3. realloc()
4. free()

DMA using malloc()

- malloc() – short for **Memory Allocation**

- Dynamically allocates a single large block of contiguous memory based on the specified size in the heap

Syntax : (void*) malloc(size_t size)   // size_t - unsigned int

- It returns a pointer to the first byte of the allocated memory on success or NULL if the allocation fails.

DMA using malloc()

- commonly used when required memory size is not known until runtime.

Why void Pointer?

- malloc doesn't know what it is pointing to.
- It simply allocates memory requested by the user without knowing the type of data to be stored inside the memory
- The void pointer is then typecasted to an appreciate type

```
int *arr;
arr = (int *)malloc(5 * sizeof(int));     // Allocate memory for an array of 5 integers
```

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

int main(){
    int i, n;
    int *A = NULL;

    printf("Enter the number of integers: ");
    scanf("%d", &n);
    A = (int *)malloc(n * sizeof(int));
    if (A == NULL){
        printf("Memory allocation failed\n");
        return 1;
    }
    for (i = 0; i < n; i++){
        printf("Enter an integer: ");
        scanf("%d", A+i);
    }
    for (i = 0; i < n; i++)
        printf("%d ", *(A+i));
    return 0;
}
```

Initializing NULL pointer

Memory allocation at run time

DMA using malloc()

Example 1: Allocating float type memory

```
ptr = (float *)malloc(40);   // allocates 10 float type slots.
```

Example 2: Allocating double type memory

```
ptr = (double *)malloc(80); // allocates 10 double type slots.
```

Example 3: Allocating long type memory

```
ptr = (long *)malloc(40);    // allocates 5 long type slots.
```

Example 4: Allocating char type memory

```
ptr = (char *)malloc(10);    // allocates 10 char type slots.
```

Assigning/Accessing data in DMA – malloc()

```c
int main (){
    int *DMA;
    DMA = (int*) malloc (40); //10 slots are allocated
    int *pointer = DMA; // assign first element address
    *pointer = 29;
     pointer += 6;
    *pointer = 67;
    for (int i = 0; i < 10; i++){
        printf("DMA [%d] = %d\n", i, *DMA);
        DMA++;
    }
    return 0
}
```

```
DMA [0] = 29
DMA [1] = 5505109
DMA [2] = 5374021
DMA [3] = 4259918
DMA [4] = 4522061
DMA [5] = 4456509
DMA [6] = 67
DMA [7] = 5505099
DMA [8] = 5242959
DMA [9] = 4325421
```

0	1	2	3	4	5	6	7	8	9

DMA using calloc()

- calloc() – short for **Clear Allocation**

- Dynamically allocates a **specified number of blocks of memory**, each of a specified size.

Syntax : (void*) calloc(size_t num_elements, size_t element_size)

- It returns a pointer to the first byte of the allocated memory on success or NULL if the allocation fails.

calloc vs malloc

calloc needs two arguments | malloc needs one argument

Allocation with calloc:

```
A = (int *)calloc(n, sizeof(int));
```

Same allocation with malloc:

```
A = (int *)malloc(10 * sizeof(int));
```

calloc vs malloc

calloc	malloc
calloc needs two arguments	Malloc needs one argument
Allocated memory is initialized to zero	Allocated memory is initialized with garbage

Both functions return NULL when insufficient memory in the heap.

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

int main(){
    int i, n;
    int *A = NULL;

    printf("Enter the number of integers: ");
    scanf("%d", &n);
    A = (int *)calloc(n,  sizeof(int));
    if (A == NULL){
        printf("Memory allocation failed\n");
        return 1;
    }
    for (i = 0; i < n; i++){
        printf("Enter an integer: ");
        scanf("%d", A+i);
    }
    for (i = 0; i < n; i++)
        printf("%d ", *(A+i));
    return 0;
}
```

Initializing NULL pointer

Clear memory allocation at run time

Assigning/Accessing data in DMA – calloc()

```c
int main (){
    int *DMA;
    DMA = (int *)calloc(10, sizeof(int));//10 slots are allocated
    int *pointer = DMA; // assign first element address
    *pointer = 29;
     pointer += 6;
    *pointer = 67;
    for (int i = 0; i < 10; i++){
        printf("DMA [%d] = %d\n", i, *DMA);
        DMA++;
    }
    return 0
}
```

```
DMA [0] = 29
DMA [1] = 0
DMA [2] = 0
DMA [3] = 0
DMA [4] = 0
DMA [5] = 0
DMA [6] = 67
DMA [7] = 0
DMA [8] = 0
DMA [9] = 0
```

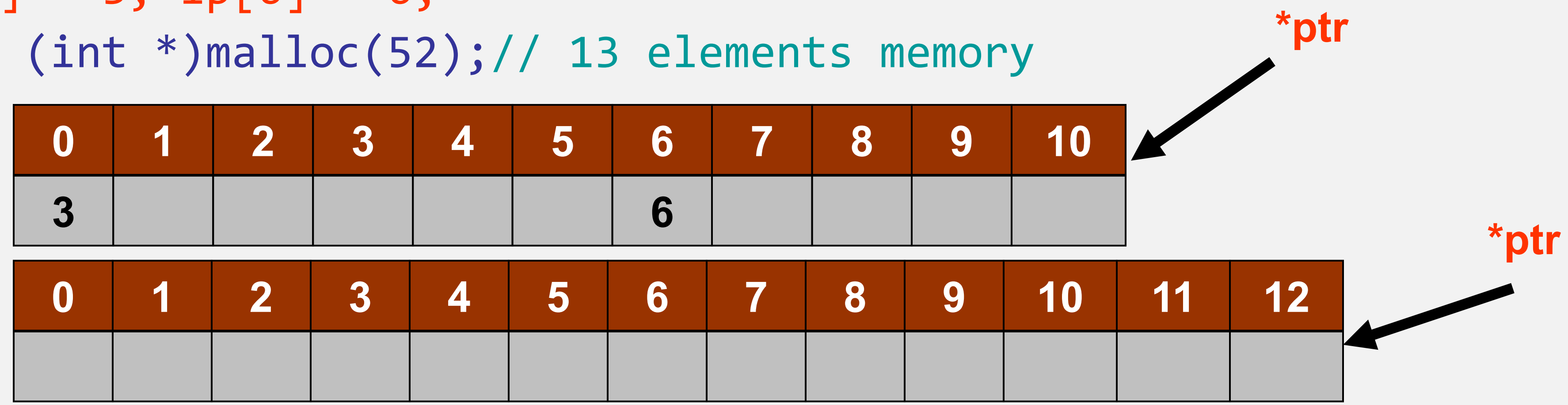
0	1	2	3	4	5	6	7	8	9
29	0	0	0	0	0	67	0	0	0

Problem with DMA using malloc and calloc

- The major problem with DMA using malloc/calloc is that, when it is recursively called upon same pointer type variable, it removes the old information.

For Example:-

```
int *ptr;
ptr = (int *)malloc(44); // 11 elements memory
ptr[0] = 3; ip[6] = 6;
ptr = (int *)malloc(52);// 13 elements memory
```



DMA using realloc()

- realloc() – short for **Reallocation**

- Change the size of the memory block without losing the old data.

  Syntax : (void*) realloc(void *ptr , size_t new_size)

- It returns a pointer to resized memory block If reallocation fails, it returns NULL, and the original block is unchanged.

DMA using realloc (Example)

```
int *ptr;
iptr = (int *)malloc(44);
ptr[0] = 3;
ptr[6] = 6;
ptr = (int *)realloc(ptr, 52);
```

*iptr

0	1	2	3	4	5	6	7	8	9	10
3						6				

*iptr

0	1	2	3	4	5	6	7	8	9	10	11	12
3						6						

DMA using realloc()

```
int *A = (int *)malloc(10 * sizeof(int));
A = (int *)realloc(A, 20 * sizeof(int));
```

- Allocate memory of size 20 * sizeof(int)
- Moves the contents of the old block to a new block
  - Increase or decrease the size of the block.
  - If the new size is larger, the content of the old block is preserved up to the original size.
  - If the new size is smaller, the excess data is truncated.
- Newly allocated bytes are uninitialized and contains garbage values.

```c
int main(){
    int i;
    int *A = (int *)malloc(2 * sizeof(int));

    if (A == NULL){
        printf("Memory allocation failed\n");
        return 1;
    }
    printf("Enter two integers: ");
    for (i = 0; i < 2; i++){
        scanf("%d", A+i);
    }
    // Allocate memory for 3 more integers
    A = (int *)realloc(A, 5 * sizeof(int));
    if (A == NULL){
        printf("Memory allocation failed\n");
        return 1;
    }
    printf("Enter three more integers: ");
    for (i = 2; i < 5; i++){
        scanf("%d", A+i);
    }
    for (i = 0; i < 5; i++)
        printf("%d ", *(A+i));
    return 0;
}
```

Allocate memory for 2 integers at run time

Reallocate memory for 3 more integers

Release Dynamically Allocated Memory

- free() - release the dynamically allocated memory in heap.

Importance:

- Memory allocated dynamically using malloc, calloc, or realloc will not be released automatically after use. The space remains and cannot be used.
- This memory must be deallocated to prevent **memory leaks**.

Syntax : (void*) free(void *ptr)

takes a pointer to the memory block to be deallocated

Memory leaks

- A **memory leak** occurs when the program allocates memory dynamically (using functions like malloc, calloc, or realloc) but **fails to release that memory** before the program terminates.

- **Memory leaks** can lead to inefficient memory usage, reduced performance, and, in extreme cases, program crashes due to running out of available memory.

```c
#include <stdio.h>
#include <stdlib.h>
int main(){
    int i, n;
    int *A = NULL;

    printf("Enter the number of integers: ");
    scanf("%d", &n);
    A = (int *)malloc(n * sizeof(int));
    if (A == NULL){
        printf("Memory allocation failed\n");
        return 1;
    }
    for (i = 0; i < n; i++){
        printf("Enter an integer: ");
        scanf("%d", A+i);
    }
    for (i = 0; i < n; i++)
        printf("%d ", *(A+i));
    free(A);
    A = NULL;
    return 0;
}
```

Initializing NULL pointer

Memory allocation at run time

Deallocate memory after use.
Assign NULL to A pointer

Memory Leaks - Example

Forgot to free the space and no longer has a pointer to it anymore. It cannot be allocated anymore and cannot used anymore.

```
void process_data(void){
        int *pointer;
        pointer = (int *)realloc (pointer,200000);
    }

int main (){
    process_data();
    return 0;
}
```

Forgot to free the memory after use here

What will happen if process_data() is called multiple times?

Exercise – DMA

Write a C program to dynamically allocate an integer, a character and a string and assign a value to them.

Exercise – DMA


```
int main(){
    int *A = NULL;
    char *CH = NULL;
    char *str = NULL;

    A = (int *)malloc(sizeof(int));
    CH = (char *)malloc(sizeof(char));
    str = (char *)malloc(10 * sizeof(char));

    if (A == NULL || CH == NULL || str == NULL){
        printf("Memory allocation failed\n");
        return 1;
    }
    *A = 29;
    *CH = 'A';
    str = "Hello";

    printf("Integer = %d\n", *A);
    printf("Char = %c\n", *CH);
    printf("String = %s\n", str);
    return 0;
}
```

There are two issues in this code. Can you spot them?

Use strcpy(str, "Hello"); to avoid memory leak

Free memory allocations for A, CH and str

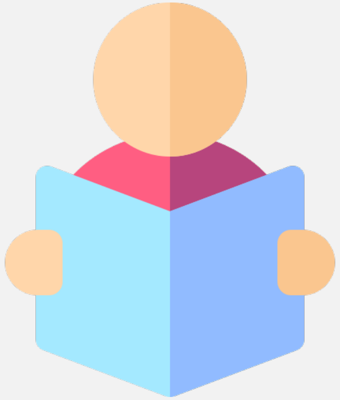
Exercise – String copy using pointers

Given a string “Hello World!“, copy it to another string using pointers without using strcpy function.

Exercise – String copy using pointers


```c
#include <stdio.h>

int main()
{
    char str1[] = "Hello, World!";
    char str2[20];

    char *src = str1;
    char *dest = str2;
    while (*src != '\0')
    {
        *dest = *src;
        src++;
        dest++;
    }
    *dest = '\0'; // Ensure null-termination
    printf("str1: %s\n", str1);
    printf("str2: %s\n", str2);

    free(str2);
    return 0;
}
```

Additional reading and coding



Book:

C Programming for Absolute Beginner's Guide by Greg Perry and Dean Miller

(E-book available in UCL Library)

Chapter 26: Maximizing your computer's memory

A Quick Introduction to Sorting

Sorting – a quick introduction

- **Sorting** and **Searching** are among the most common parts of any programming systems.
  - Very useful in Database systems.

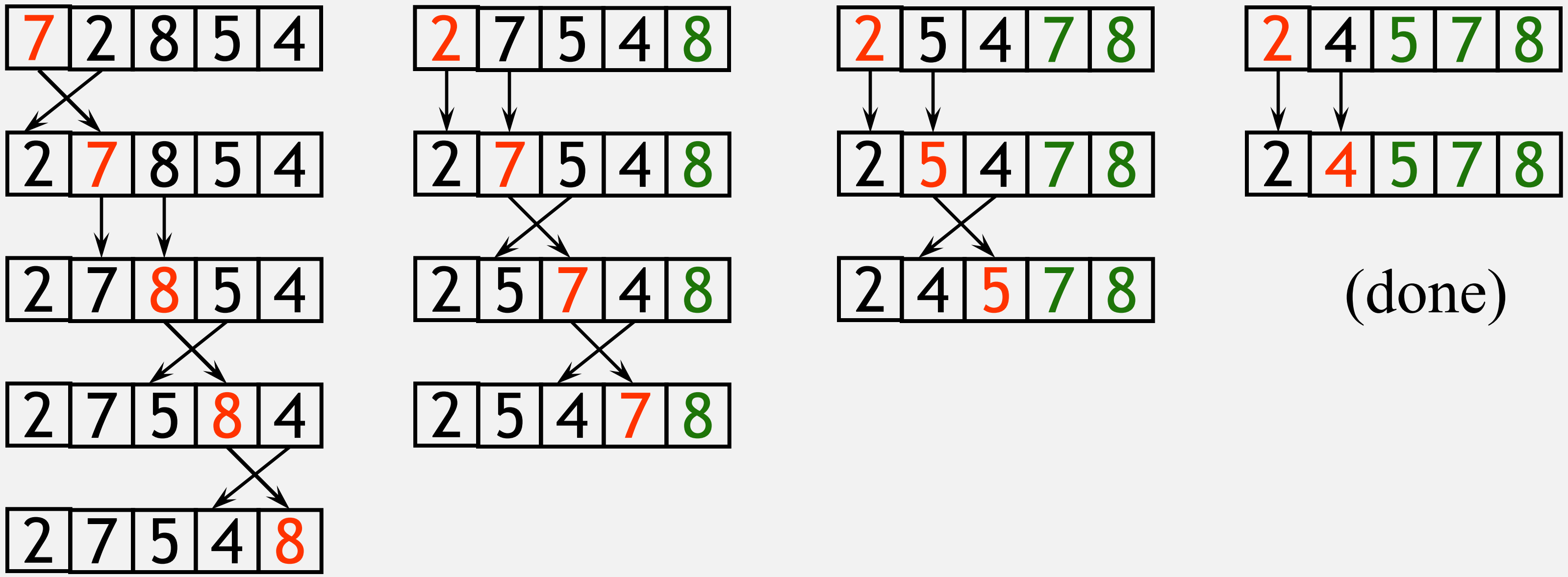
- Almost all small and large programs used Sorting and Searching Algorithms.

- There are almost dozens of efficient sorting algorithms exist, but most commonly known or used are bubble sort, quick sort, insertion sort, selection sort, merge sort.
- We will only cover bubble sort (to give a flavor of sorting) – MORE TO COME IN NEXT TERM

Bubble Sort

- In first iteration, compare each element (except the last one) with its neighbor to the left
  - If they are out of order, **swap them**
  - This puts the largest element at the very end
  - The last element is now in the correct and final place
- In second iteration, Compare each element (except the last *two*) with its neighbor to the left
  - If they are out of order, swap them
  - This puts the second largest element next to last
  - The last two elements are now in their correct and final places
- In third iteration, Compare each element (except the last *three*) with its neighbor to the left
  - Continue as above until you have no unsorted elements on the left

Example of Bubble Sort



String Manipulation – Compare and Copy

- Comparing strings
  - `int strcmp(char *s1, char *s2 )`
    - Compares character by character
    - Returns
      - Zero if strings are equal
      - Negative value if first string is less than second string
      - Positive value if first string is greater than second string
  - `char *strcpy( char *s1, char *s2 )`
    - Copies second argument into first argument
      - First argument must be large enough to store string and terminating null character

Text Array - Bubblesort (Example)

Korea	England	USA	Brazil	Austria

```
void bubbleSort(char a[ ][ ], int size)
{
  int outer, inner; char temp[90];
  for (outer = 0; outer < size-1; outer++)
  {
        for (inner = 0; inner < size-outer-1; inner++)
        {
                if ( strcmp (a[inner], a[inner + 1]) > 0 )
                {
                        strcpy (temp, a[inner] );
                        strcpy (a[inner], a[inner+1] );
                        strcpy (a[inner+1], temp);
                }
        } // end of inner loop
  } // end of outer loop
} // end of function
```

Text Array (Example)

Austria	Brazil	~~England~~	~~Korea~~	~~USA~~

Size	Outer	Inner
5	3	0

```
void bubbleSort(char a [ ][ ], int size)
{
   int outer, inner; char temp[90];
   for (outer = 0; outer < size-1; outer++)
   {
            for (inner = 0; inner < size-outer-1; inner++)
            {
                     if ( strcmp(a[inner], a[inner + 1]) > 0 )
                     {
                              strcpy (temp, a[inner] );
                              strcpy (a[inner], a[inner+1] );
                              strcpy (a[inner+1], temp);
                     }
            }
   }
}
```

Sorting Floating Array (Code)

```
void bubbleSort(float a [ ], int size)
{
   int outer, inner;
   for (outer = 0; outer < size-1; outer++)
   {
            for (inner = 0; inner < size-outer-1; inner++)
            {
                     if ( a[inner] > a[inner+1] )
                     {
                              float temp = a[inner];
                              a[inner] = a[inner+1];
                              a[inner+1] = temp;
                     }
            } // end of inner loop
   } // end of outer loop
} // end of function
```

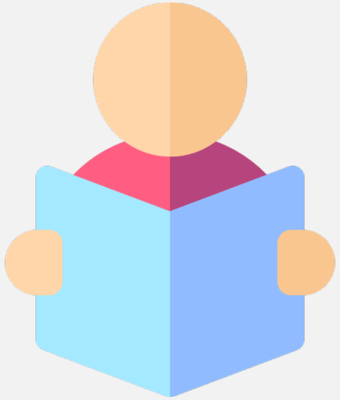
Descending Order Sorting Array (Code)

```
void bubbleSort (int a [ ], int size)
{
  int outer, inner;
  for (outer = 0; outer < size-1; outer++)
  {
        for (inner = 0; inner < size-outer-1; inner++)
        {
              if ( a[inner] < a[inner+1] )
              {
                    int temp = a[inner];
                    a[inner] = a[inner+1];
                    a[inner+1] = temp;
              }
        } // end of inner loop
  } // end of outer loop
} // end of function
```

Calling Bubble Sort Function in Main Code

```
#include <stdio.h>
void bubbleSort ( int array [ ], int size );
void main (void)
{
        int array[100];
        for ( int i < 0; i < 100; i++ )
        {
                array[i] = 100-i;
        }

        bubbleSort (array, 100);

} // end of main code
```

Additional reading and coding



Book:

C Programming for Absolute Beginner's Guide by Greg Perry and Dean Miller

(E-book available in UCL Library)

Chapter 23: Alphabetizing and Arranging Your Data

Summary

- Understanding the Memory
- Static vs Dynamic Memory
- Dynamic Memory Allocation (DMA)
- Importance of Pointers in DMA
- Diving into the built-in functions for DMA

- Brief introduction to sorting
  - Bubble sort – with integer and text examples

Assessment 5: Inperson lab assessment (12%)

- 27^{th} Nov 2023
- Timed 45 minutes exercises

This will be a closed notes assessment. No lecture slides permitted.

Everything covered so far is included in this assessment with a **focus on lecture 6 and lecture 7**.

Reminder:

If you miss the inperson assessment, apply for EC.

No grade will be given to remotely submitted assessment.

Assessment 6 – (20%)

- To be announced during Lecture 8 - **28th Nov 2023**

- **Deadline: 6th Dec 2023** (EOD)

Important:

- ncurses library is installed and works on your computer (macOS, Windows, etc).
- Run the started ncurses code in the next few slides to ensure it works on your laptop.
- Reachout to us during the lab session 8, if any issues with ncurses installation.

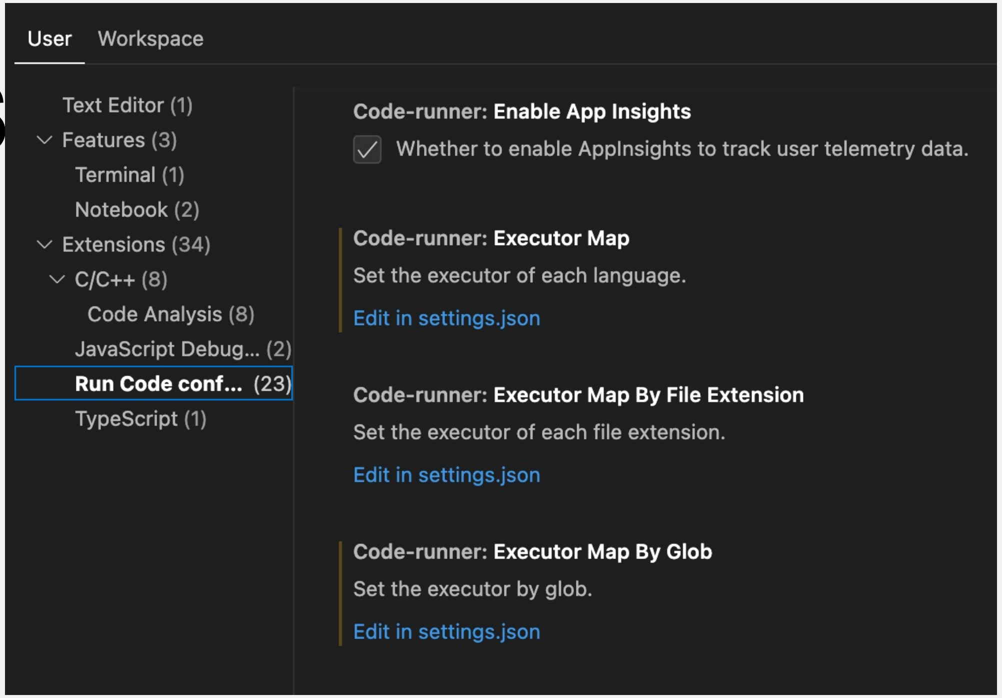
Ncurses installation on macOS

- Install ncurses

```
brew install ncurses
```

Go in VS code settings → Under user → C/C++ → run code conf
- Look for Code-runner: Executor map

-Click on ‘edit settings.json’



Add –lncurses as highlighted

```
{
    "workbench.colorTheme": "Default Dark Modern",
    "code-runner.runInTerminal": true,
    "debug.onTaskErrors": "debugAnyway",
    "code-runner.executorMap": {

        "javascript": "node",
        "java": "cd $dir && javac $fileName && java $fileNameWithoutExt",
        "c": "cd $dir && gcc $fileName -o $fileNameWithoutExt -lncurses && $dir$fileNameWithoutExt",
        "zig": "zig run",
        "cpp": "cd $dir && g++ $fileName -o $fileNameWithoutExt && $dir$fileNameWithoutExt",
        "objective-c": "cd $dir && gcc -framework Cocoa $fileName -o $fileNameWithoutExt && $dir$fileNameWithoutExt",
```

Ncurses on Windows

- MinGW by default has it installed.

Follow the same settings in VSCode as mentioned in the previous slide.

Add -lncurses -DNCURSES_STATIC as the flag

```
{
    "workbench.colorTheme": "Default Dark Modern",
    "code-runner.runInTerminal": true,
    "debug.onTaskErrors": "debugAnyway",
    "code-runner.executorMap": {

        "javascript": "node",
        "java": "cd $dir && javac $fileName && java $fileNameWithoutExt",
        "c": "cd $dir && gcc $fileName -o $fileNameWithoutExt -lncurses && $dir$fileNameWithoutExt",
        "zig": "zig run",
        "cpp": "cd $dir && g++ $fileName -o $fileNameWithoutExt && $dir$fileNameWithoutExt",
        "objective-c": "cd $dir && gcc -framework Cocoa $fileName -o $fileNameWithoutExt && $dir$fileNameWithoutExt",
```

To include in your code,

```
#include <ncurses/ncurses.h>
```

Ncurses test codes

- Taking user input

```c
// Write a code that takes your grade as input and displays your grade.

#include <stdio.h>
#include <ncurses.h>

int main()
{
    int grade;

    /* Curses Initialisations */
    initscr();
    raw();
    keypad(stdscr, TRUE);
    noecho();

    printw("Enter your grade:\n");
    grade = getch();

    printw("\nYour grade is %c", grade);
    getch();

    return 0;
}
```

Ncurses test codes

```
#include <ncurses/ncurses.h>
int main()
{
    int ch;
    initscr();
    raw();
    keypad(stdscr, TRUE);
    noecho();

    printw("Type any character to see it in bold\n");
    while (1){
        ch = getch();

        if (ch == KEY_LEFT)
            printw("Left arrow is pressed\n");
        else if (ch == KEY_RIGHT)
            printw("Right arrow is pressed\n");
        else if (ch == KEY_UP)
            printw("Up arrow is pressed\n");
        else if (ch == KEY_DOWN)
            printw("Down arrow is pressed\n");
        // if ESC is pressed, end the program
        else if (ch == 27)
            break;
    }
    //getch();
    endwin();
    return 0;
}
```

Taking arrow keys as input