Dynamic memory Allocation

SDC OSW CSE 3541

Department of Computer Science & Engineering
ITER, Siksha 'O' Anusandhan Deemed To Be University
Jagamohan Nagar, Jagamara, Bhubaneswar, Odisha - 751030

Text Book(s)

Jeri R. Hanly & Elliot B. Koffman

Problem Solving and Program Design in C

Seventh Edition, Pearson Education

Robert Love

LINUX System Programming

Second Edition, SPD, O'REILLY

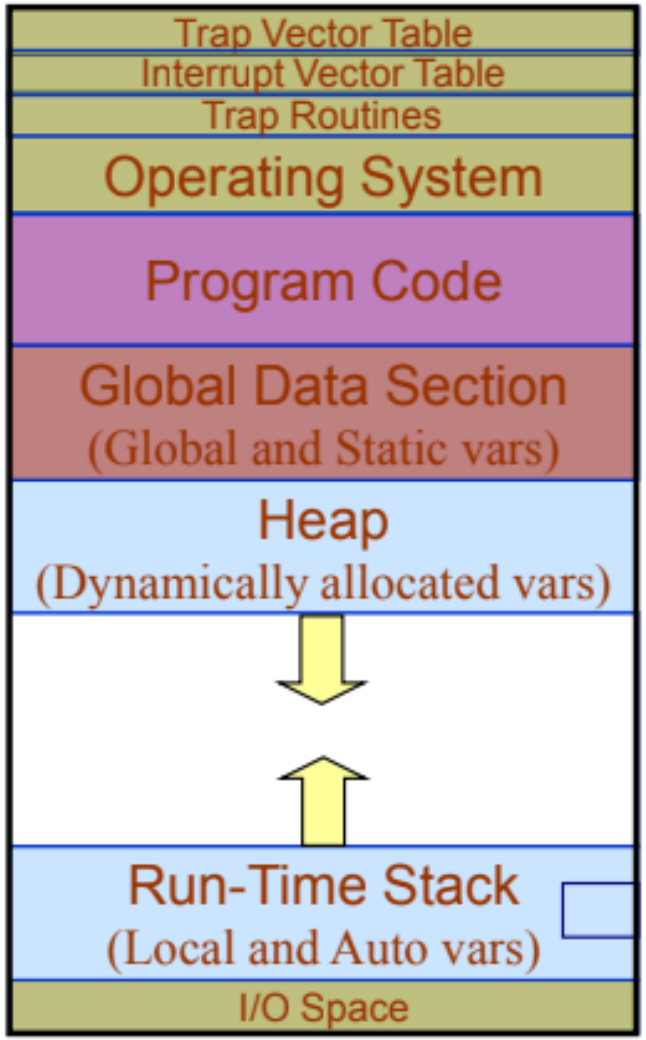
Talk Flow

Introduction

- In some situations, data is dynamic in nature.
  - Amount of data cannot be predicted beforehand.
  - Number of data item keeps changing during program execution.
- **Dynamic Memory Allocation :** Ability of a program to use more memory space at execution time
  - Memory space required can be specified at the time of execution.
  - C supports allocating memory dynamically using library routines.

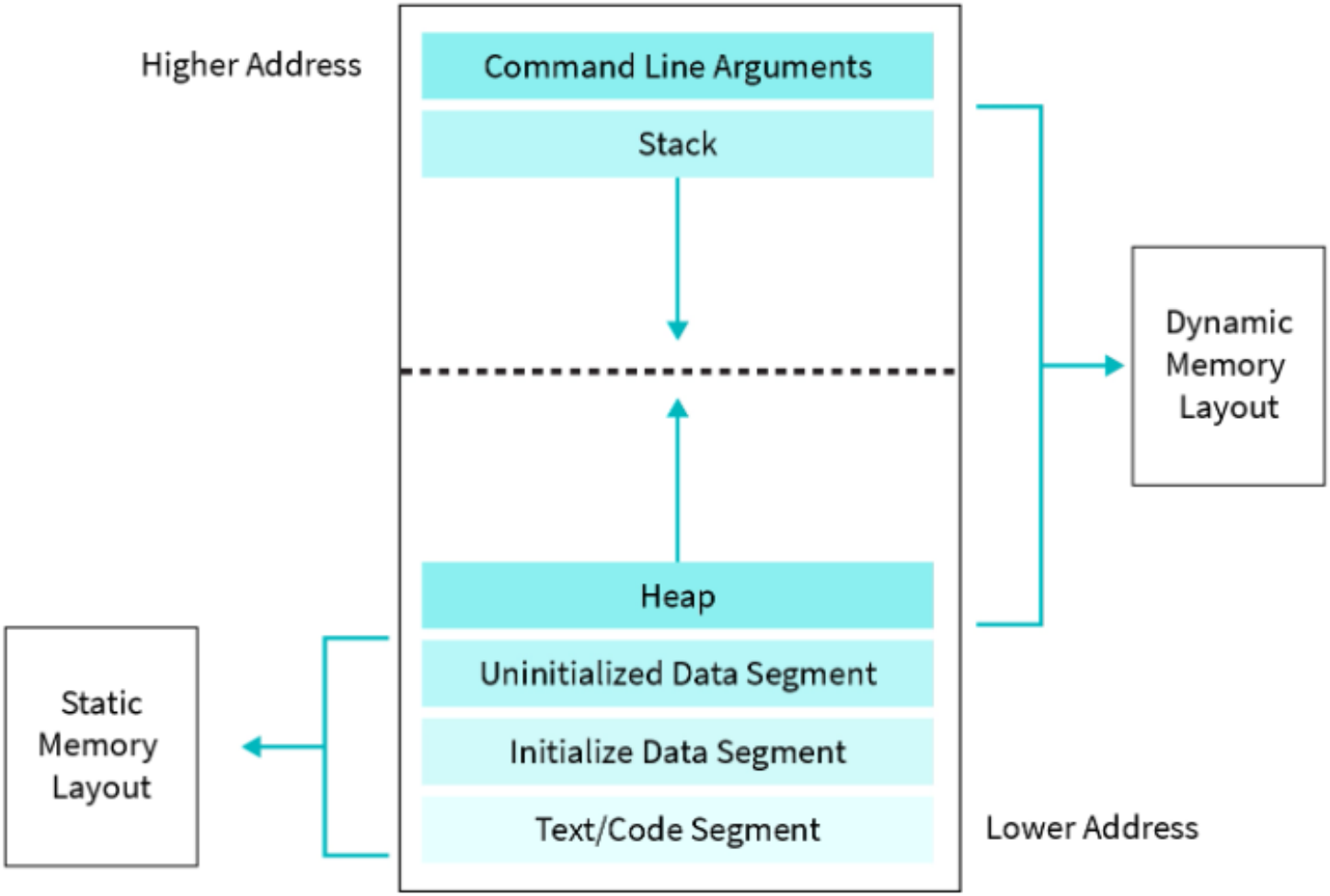
Memory in C

- Variable memory is allocated in three areas :
  - Global data section
  - Runtime stack
  - Dynamically allocated heap
- Global variables are allocated in the **global data section** and are accessible from all parts of the program.
- Local variables are allocated during execution on the run-time stack
- **Dynamically allocated variables** are item created during run-time and are allocated on the heap.



Layout of a Program Image

- The diagram mentioned below shows a visual representation of how RAM loads a program written in C into several segments.



Memory Layout

Description of Program Segments

- **Text segment :**
  - Compiled binary file contains instructions stored in the text segment of memory.
  - Text segment has read-only permission that prevents the program from accidental modifications.
- **Initialized data segment :** Initialized data segment holds initialized values for external, global, static, and constant variables in a C program.
- **Uninitialized data segment :** An uninitialized data segment is also known as bss (block started by symbol). The program loaded allocates memory for this segment when it loads. Every data in bss is initialized to arithmetic 0 and pointers to null pointer by the kernel before the C program executes.

Description of Program Segments Contd...

- **Stack :**
  - Stack segment stores local variables, function parameters, and essential information like the return address after a function call.
  - Stack pointer registers track the top of the stack, updating with push/pop actions.
- **Heap :** Heap is used for memory which is allocated during the run time (dynamically allocated memory). Heap generally begins at the end of bss segment and, they grow and shrink in the opposite direction of the Stack.
- **Command line arguments :** When a program executes with arguments passed from the console like argv and argc and other environment variables, the value of these variables gets stored in this memory layout in C.

Memory Allocation Functions

- **malloc**
  - Allocates requested number of bytes and returns a pointer to the first byte of the allocated space.
- **calloc**
  - Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
- **realloc**
  - Modifies the size of previously allocated space.
- **free**
  - Frees previously allocated space.

Allocating a Block of Memory

- **A block of memory can be allocated using the function malloc.**
  - Reserves a block of memory of specified size and returns a pointer of type void.
  - The return pointer can be type -casted to any pointer type.
- **General Format**

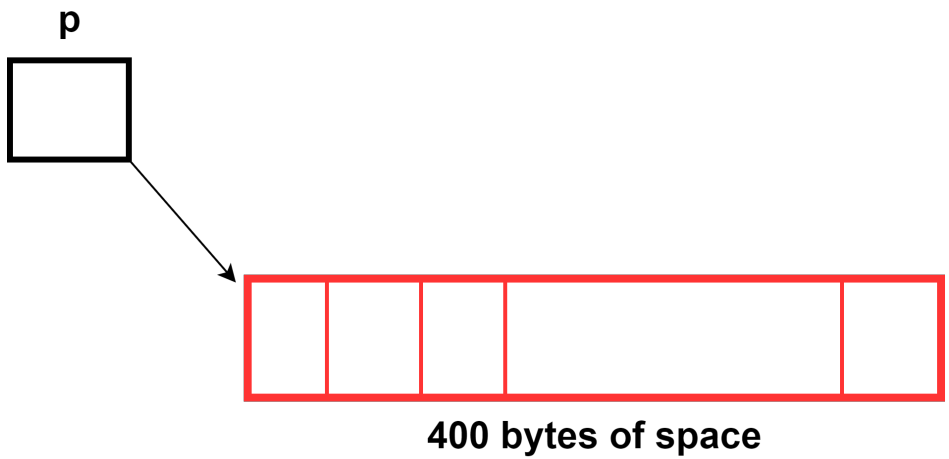
  ptr = (type *) malloc (byte size);

Function malloc

Examples

p = (int *) malloc(100 * sizeof(int));

- A memory space equivalent to 100 times the size of an int bytes is reserved.
- The address of the first byte of the allocated memory is assigned to the pointer p of type int.



Contd.

- **Example**

  cptr = (char *) malloc (20);

  - Allocates 20 bytes of space for the pointer cptr of type char.

- **Example**

  sptr = (struct stud *) malloc (10 * sizeof (struct stud));

  - Allocates space for a structure array of 10 elements. sptr points to a structure element of type “struct stud”.

Points to remember

- **malloc always allocates a block of contiguous bytes.**
  - The allocation can fail if sufficient contiguous memory space is not available.
  - If it fails, malloc returns NULL.

```
if  ((p = (int *) malloc(100 * sizeof(int))) == NULL)
  {
     printf ("\n Memory cannot be allocated");
     exit();
  }
```

Example of malloc function

- Create a program to allocate a memory of 100 characters using malloc(). Ask user to enter your name. Read the response with scanf() function and assign the user name to the allocated memory. Finally display your name to the screen.

Example of malloc function

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
      int i, N = 100;
      char *Name;
      Name = (char*)malloc(N * sizeof(char));
      printf("Enter your name:");
      scanf("%s",Name);
      printf("Your Name is = %s \n", Name);
      free(Name);
      return 0;
}
```

Function calloc

- calloc() is a variant of malloc().
- calloc() takes two arguments: the number of "things" to be allocated and the size of each "thing" (in bytes).
- calloc() returns the address of the chunk of memory that was allocated.
- calloc() also sets all the values in the allocated memory to zeros (malloc() doesn't).
- calloc() is also used to dynamically allocate arrays.

General Format

ptr = (type *) calloc (number, size);

Contd.

- **Example**

  p = (int *) calloc (100 , sizeof(int));

  - This line of code allocates memory for an array of 100 integers and assigns the address of that memory to the pointer variable 'p'.

Example of calloc function

- Program to check dynamic memory is allocated using calloc() function

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *ptr;
    ptr = calloc (5, sizeof(int));
    if (ptr != NULL)
    {
        printf (" Memory is created successfully \n");
    }
    else
        printf (" Memory is not created ");
    return 0;
}
```

Altering the Size of a Block - realloc function

- **Sometimes we need to alter the size of some previously allocated memory block.**
  - More memory needed.
  - Memory allocated is larger than necessary.
- **How ?**
  - By using the **realloc** function.
- **If the original allocation is done as:**

  ptr = malloc (size);

  - reallocation of space may be done as:
    ptr = realloc (ptr, newsize);

Contd.

- **The new memory block may or may not begin at the same place as the old one.**
  - If it does not find space, it will create it in an entirely different region and move the contents of the old block into the new block.
- **The function guarantees that the old data remains intact.**
- **If it is unable to allocate, it returns NULL and frees the original block.**

Example of realloc function

- **Case 1 :** If the given size is bigger than the actual, it will check whether it can expand the already available memory. If it is possible it will simply resize the memory.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int *ptr,i;
    ptr = malloc(sizeof(int)); //allocating memory for only
         1 integer
    ptr[0] = 1;
    ptr = realloc(ptr, 3 * sizeof(int));
    ptr[1] = 2;
    ptr[2] = 3;
    for(i = 0; i < 3; i++)
          printf("%d\n",ptr[i]);
    return 0;
}
```

Example of realloc function

- **Case 2 :** An example to demonstrate **the use of realloc when the pointer is initially NULL and size is provided.** If the pointer passed to realloc is NULL, it behaves like malloc and allocates a new memory block of the specified size. If the pointer is not NULL, it resizes the existing memory block to the given size.

Example of realloc function

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *ptr = NULL;
    int newSize = 5;
    ptr = (int*)realloc(ptr, newSize * sizeof(int));
    if (ptr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }
    for (int i = 0; i < newSize; i++) {
        ptr[i] = i * 2;  // Access and modify the allocated
            memory
    }
    for (int i = 0; i < newSize; i++) {
        printf("%d ", ptr[i]);  // Display the values in
            the allocated memory
    }
    free(ptr);
}
```

Example of realloc function

- **Case 3 : In realloc if ptr is predefined and size will be zero.** In this example, the memory is initially allocated for an array of integers, and then the realloc function is used to resize the memory block to zero bytes. This is equivalent to freeing the memory using the free function. Therefore after resizing to zero bytes, the pointer is still valid but should not be dereferenced. Accessing or modifying the memory could lead to undefined behavior.

Example of realloc function

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int* ptr = (int*)malloc(5 * sizeof(int));
    if (ptr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }
    // Print the values before reallocating
    for (int i = 0; i < 5; ++i) {
        printf("%d ", ptr[i]);
    }
    // Resize the memory block to zero bytes (similar to
        free)
    ptr = (int*)realloc(ptr, 0);
    if (ptr == NULL) {
        printf("Memory reallocation failed\n");
        return 1; // Exit with an error code
    }
```

Example of realloc function

```
    // After reallocating to zero bytes, the pointer is
        still valid but should not be dereferenced
    // Print the values after reallocating
    for (int i = 0; i < 5; ++i) {
        printf("%d ", ptr[i]);
    }
    // Free the memory (it's equivalent to realloc(ptr, 0))
    free(ptr);
    return 0;
}
```

Releasing the Used Space - free function

- **When we no longer need the data stored in a block of memory, we may release the block for future use.**
- **How ?**
  - By using the **free** function.
- **General Syntax :**

  free (ptr);

  where ptr is a pointer to a memory block which has been previously created using malloc.
- **When we free() some memory, the memory is not erased or destroyed.**
- **Instead, the operating system is informed that we don't need the memory any more, so it may use it for e.g. another program.**
- **Trying to use memory after freeing it can cause a segmentation violation (program crash).**

Releasing the Used Space - free function

- **Case 1 : A case if free() is used more than once.** Using free() more than once on the same memory address can lead to undefined behavior in C. **Once memory has been deallocated with free(), the pointer becomes invalid, and any further attempt to free the same memory can result in unpredictable consequences.** Here's an example that illustrates the issue:

Example

- However, there is a mistake in the code where free(ptr) is called again, even though the memory has already been freed. This leads to undefined behavior, and the program might crash or produce unexpected results.

```c
#include <stdlib.h>
#include <stdio.h>
int main() {
    // Allocate memory for an integer
    int *ptr = (int *)malloc(sizeof(int));
    if (ptr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }
    // Assign a value to the allocated memory
    *ptr = 42;
    // Free the allocated memory
    free(ptr);
    // Attempt to free the memory again(undefined behavior)
    free(ptr);
    return 0;
}
```

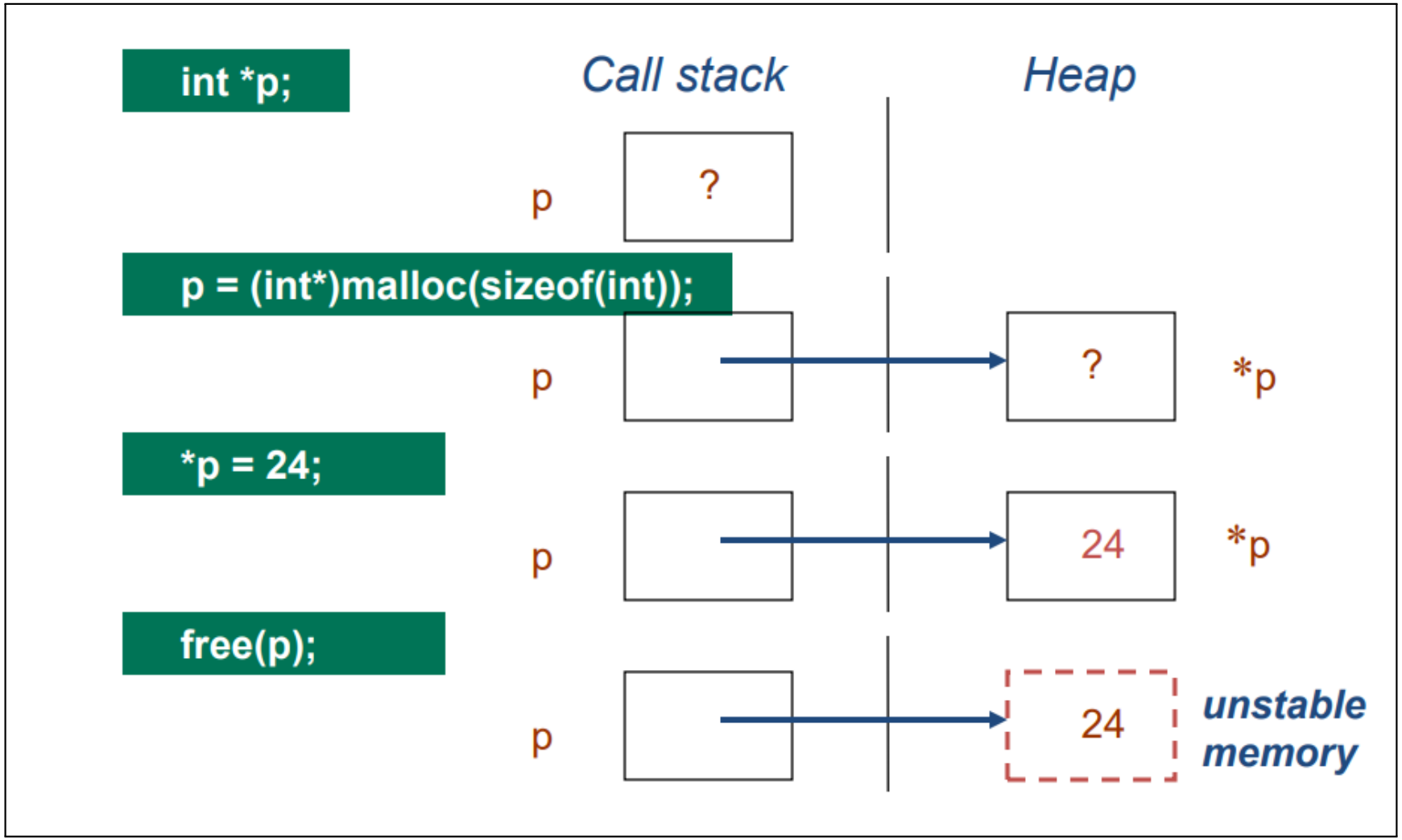
Example

- To avoid such issues, it's a good practice to set the pointer to NULL after freeing the memory. This helps in preventing accidental reuse of the pointer. This ensures that the pointer is not used after the memory has been freed, reducing the chances of undefined behavior.

```
// Free the allocated memory
free(ptr);
// Set the pointer to NULL
ptr = NULL;
// Now, it is safe to check and free the memory again if
    needed
if (ptr != NULL) {
    free(ptr);
   }
```

Pictorial representation of Dynamic Allocation

- **Example 1:**



Dynamic Memory Allocation in 1-D array

- Once the memory is allocated, it can be used with pointers, or with array notation.
- **Example :**

```
#include<stdio.h>
int main()
{
 int *p,n,i;
 scanf("%d".&n);
 p = (int*)malloc(n * sizeof(int));
 for(i=0;i<n;i++)
    printf("%d",&p[i]);
 return 0;
}
```

- The n integers allocated can be accessed as *p, *(p+1), *(p+2)...., *(p+n-1) or just as p[0], p[1], p[2],....., p[n-1]

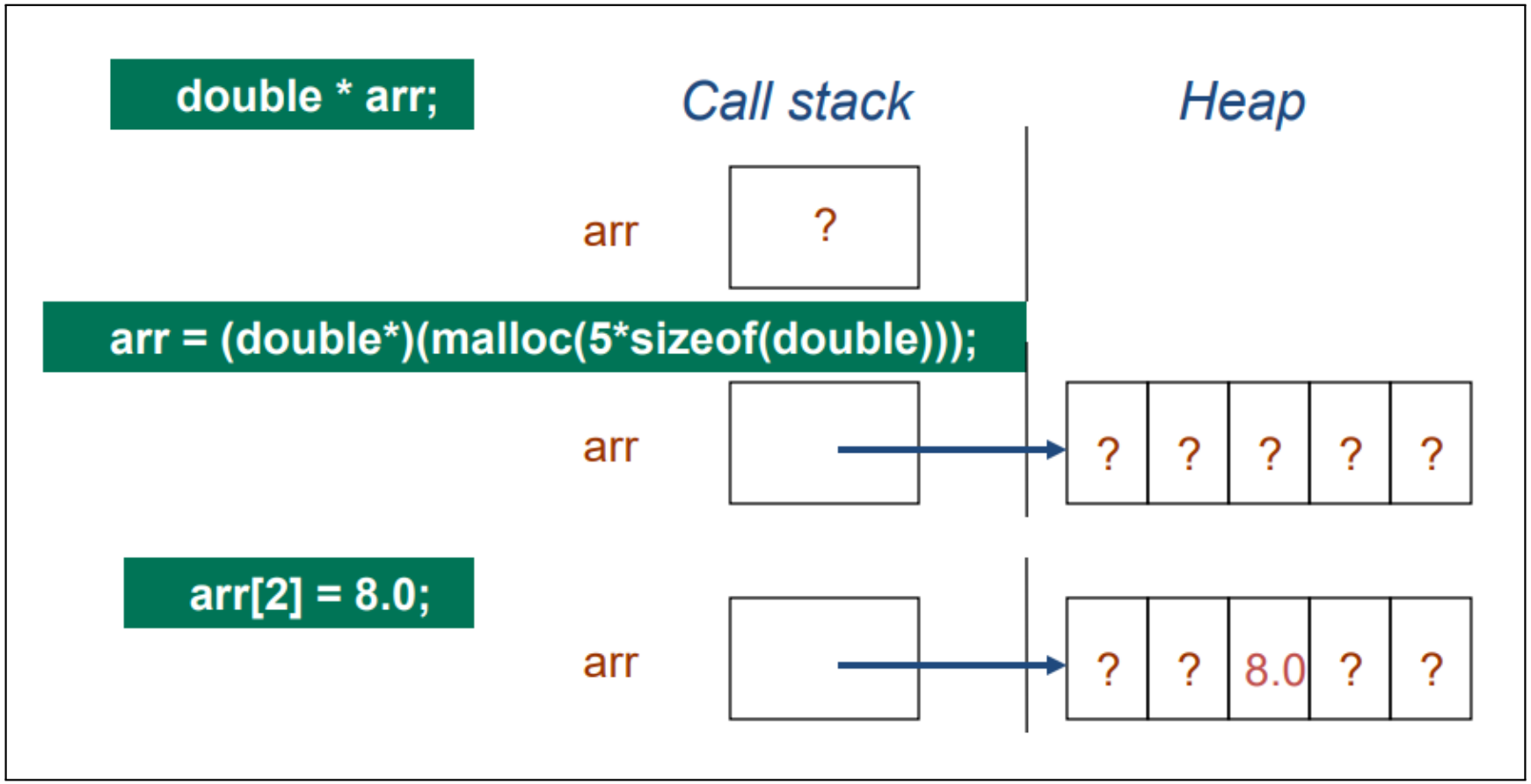
Extracting a portion of an array by using pointer

- Using a pointer can mean dynamic memory allocation instead of a variable length array. If, just for sake of using a pointer for the writing process, too, the program can be adapted as follows:
- **Example :**

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
 int *p,n,i, sum = 0;
 scanf("%d",&n);
 p = (int*)malloc(n * sizeof(int));
 for(i=0;i<n;i++)
    scanf("%d",(p+i));
 sum += *(p+i);
 printf("%d",sum);
 return 0;
}
```

Pictorial representation of Dynamic Allocation

- **Example 2: Pointer to an array**



Example of 1-D Array using malloc

- Calculating the average height of n students within a class

```
#include<stdio.h>
int main()
{
      int i,N; float *height; float sum=0, avg;
      printf("Enter the no. of students:");
      scanf("%d".&N);
      height = (float*)malloc(N * sizeof(float));
      printf("Enter the height of each student:");
      for(i=0;i<N;i++){
            scanf("%f",&height[i]);
         sum += height[i];
      }
      avg = sum / (float) N;
      printf("Average height = %f \n", avg);
      free(height);
      return 0;
}
```

Example of 1-D Array using calloc

- Calculating the average height of n students within a class

```
#include<stdio.h>
int main()
{
      int i,N; float *height; float sum=0, avg;
      printf("Enter the no. of students:");
      scanf("%d".&N);
      height = (float*)calloc(N , sizeof(float));
      printf("Enter the height of each student:");
      for(i=0;i<N;i++){
            scanf("%f",&height[i]);
         sum += height[i];
      }
      avg = sum / (float) N;
      printf("Average height = %f \n", avg);
      free(height);
      return 0;
}
```

Changing 1D Array Size With realloc

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int n=5,i; int *ptr = (int*)calloc(n, sizeof(int));
    printf("Enter the array elements:");
    for (i = 0; i < n; ++i) {
       scanf("%d",&ptr[i]);
    }
    int m = 10; // New Size of the array
    ptr = (int*)realloc(ptr, m * sizeof(int));
    printf("Enter the  rest elements of the array:");
    for (i = n; i < m; ++i) {
       scanf("%d",&ptr[i]);
    }
    for (i = 0; i < m; ++i) {
        printf("%d, ", ptr[i]); //Printing the elements
    }
    free(ptr);
}
```

Dynamic Memory Allocation of 2-D Arrays

- Now think of a 2-d array of dimension [M][N] as M 1-d arrays, each with N elements, such that the starting address of the M arrays are contiguous (so the starting address of k-th row can be found by adding 1 to the starting address of (k-1)-th row)
- Now, allocate the M arrays, each of N elements, with p[k] holding the pointer for the k-th row array
- Now p can be subscripted and used as a 2-d array
- Address of p[i][j] = *(p+i) + j (note that *(p+i) is a pointer itself, and p is a pointer to a pointer)

Using a single pointer

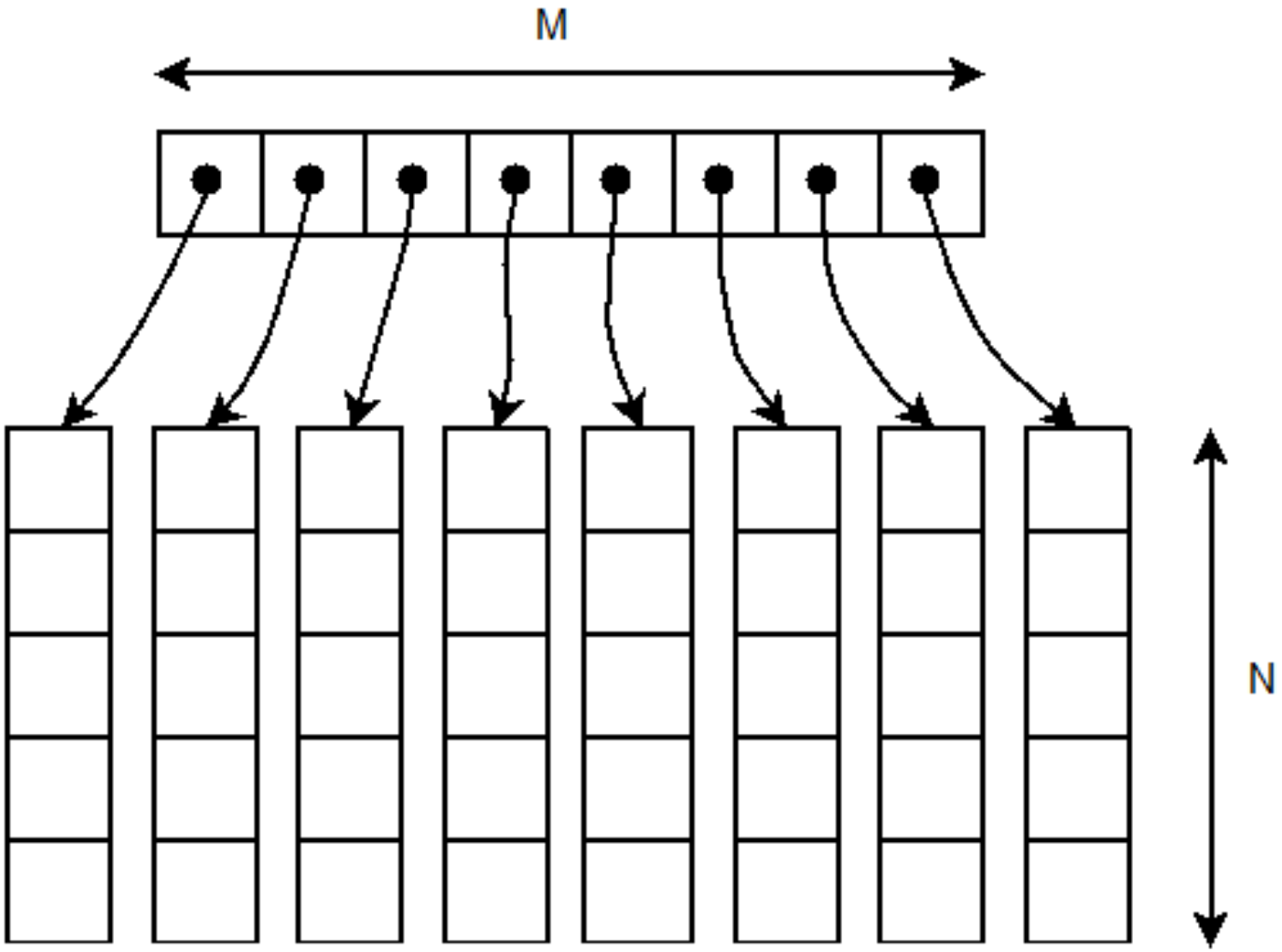
- Consider a 2D array of 3 rows and 4 columns in each row

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int row = 3, column = 4;
    int* ptr = malloc((row * column) * sizeof(int));
    for (int i = 0; i < row * column; i++)
        ptr[i] = i + 1;

    /* Accessing the array values */
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < column; j++)
            printf("%d ", ptr[i * column + j]);
        printf("\n");
    }
    free(ptr);
    return 0;
}
```

Using an array pointers

- As shown below, we can dynamically create an array of size M pointers and then dynamically allocate memory of size N for each row:



Example

- Consider a 2D array of 3 rows and 4 columns in each row

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
    int row = 3, column = 4, i, j;
    int* arr[row];
    for (i = 0; i < row; i++)
        arr[i] = (int*)malloc(column * sizeof(int));
    printf("Enter the array elements :");
    for (i = 0; i < row; i++)
        for (j = 0; j < column; j++)
            scanf("%d",&arr[i][j]);
    for (i = 0; i < row; i++)
        for (j = 0; j < column; j++)
            printf("%d ", arr[i][j]);
    for (int i = 0; i < row; i++)
        free(arr[i]);
}
```

Dangling Pointer Problem

- A dangling pointer in C is a pointer that points to a memory location that has been deallocated or is no longer valid. Example :

```
int *ptr = malloc(sizeof(int));
*ptr = 10; free(ptr);
*ptr = 20; //ptr is now a dangling pointer
```

- Another common cause of dangling pointers is using pointers to memory that is automatically deallocated when a function returns or an object goes out of scope. Example :

```
void fun(){
    int *ptr = malloc(sizeof(int));
    *ptr = 5;
    return;  // ptr is now a dangling pointer
}
void main() {
    fun();
    *ptr = 10;
}
```

Introduction to VALGRIND Tool

- **valgrind - a suite of tools for debugging and profiling programs.**
  - Valgrind is an instrumentation framework for building dynamic analysis tools.
  - It is particularly useful for detecting memory leaks, memory corruption, and undefined memory usage in programs.
  - One of the most commonly used tools is **Memcheck**, which can detect memory-related errors.

Overview of VALGRIND Tool

- **Dynamic Analysis:**
  - Valgrind performs dynamic binary instrumentation, meaning it runs your program in a virtual environment and observes its behavior at runtime.

- **Instrumentation Framework:**
  - It provides an infrastructure for building tools that analyze memory usage, thread synchronization, and other dynamic aspects of program behavior.

- **Platform Compatibility:**
  - Valgrind works on Linux and can analyze programs written in various languages, including C and C++.

Using VALGRIND Tool

- To use Valgrind, typically run our program through the Valgrind tool. The basic command is as follows:

  valgrind [valgrind-options] your-program [program-options]

- For example, to check a program named **"my-program"** , we would run:

  valgrind ./my-program

Using VALGRIND Tool

- We can customize the behavior of Valgrind by specifying various options.
  - **-v or - -verbose:** Increase verbosity.
  - **- -leak-check:** Enable or disable leak checking.
  - **- -track-origins:** Provide information about the origins of uninitialized values.
  - **- -tool = toolname:** Choose a specific Valgrind tool (memcheck).

Using Memcheck Tool:

Memory Error Detection:

- Memcheck is the most widely used tool provided by Valgrind. It can detect various memory-related errors, including memory leaks, use of undefined values, and memory corruption.

Instrumentation:

- Memcheck uses a form of dynamic binary instrumentation to track memory allocations, deallocations, and accesses at runtime.

Example: Running Valgrind with Memcheck

valgrind - -leak - check = full ./my-program

Sample Questions

- Create a program that dynamically allocates an array and allows the user to resize it by doubling its size whenever it becomes full.
- Define a function that dynamically allocates memory for an array, initializes the array, and returns a pointer to the array.
- Write a C program that dynamically allocates memory for two arrays, copies the contents of one array to another, and then prints both arrays.
- Implement a program that dynamically allocates memory for two strings, concatenates them, and then prints the result.
- Write a program that dynamically allocates memory for two matrices, performs matrix addition, and prints the result.

THANK YOU