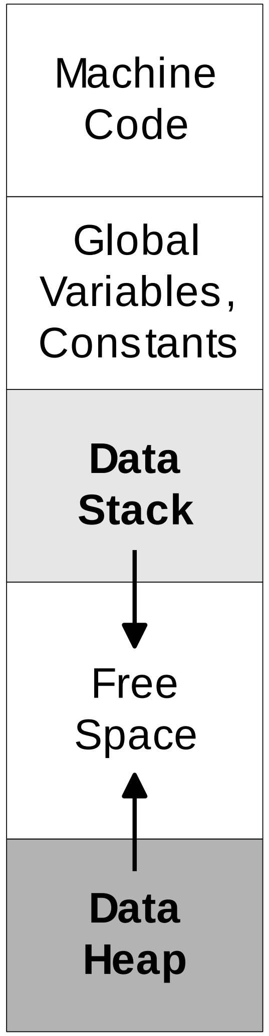
Dynamic Memory Allocation in C

By: Atiya Jokhio

Program Parts

- Space for program code includes space for machine language code and data
- Data broken into:
  - space for global variables and constants
  - data stack - expands/shrinks while program runs
  - data heap - expands/shrinks while program runs
- Local variables in functions allocated when function starts:
  - space put aside on the data stack
  - when function ends, space is freed up
  - must know size of data item (int, array, etc.) when allocated (*static allocation*)



Limits of Static Allocation

What if we don’t know how much space we will need ahead of time?

Example:

- ask user how many numbers to read in
- read set of numbers in to array (of appropriate size)
- calculate the average (look at all numbers)
- calculate the variance (based on the average)

Problem: how big do we make the array??

- using static allocation, have to make the array as big as the user might specify (might not be big enough)

Dynamic Memory Allocation

Allow the program to allocate some variables (notably arrays), during the program, based on variables in program (dynamically)

Previous example: ask the user how many numbers to read, then allocate array of appropriate size

Idea: user has routines to request some amount of memory, the user then uses this memory, and returns it when they are done

- memory allocated in the *Data Heap*

Dynamic Memory Allocation

- **Dynamic Memory Allocation** is allocation and freeing of memory according to your programming needs. Dynamic memory is managed and served with pointers that point to the newly allocated memory space in an area which we call the heap.
- Now you can create and destroy an array of elements dynamically at runtime without any problems. To sum up, the automatic memory management uses the stack, and the dynamic memory allocation uses the heap.

Memory Management Functions

- C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:
  - malloc()
  - calloc()
  - free()
  - realloc()

Memory Management Functions

calloc - routine used to allocate arrays of memory

malloc - routine used to allocate a single block of memory

realloc - routine used to extend the amount of space allocated previously

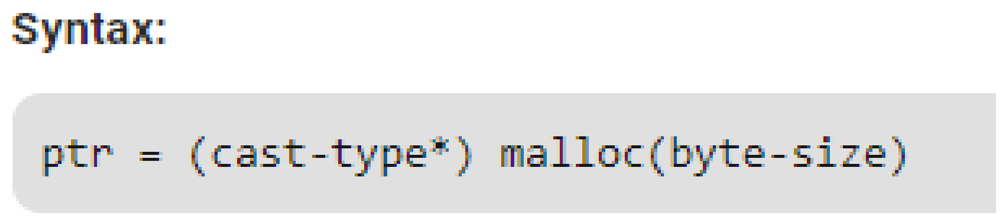
free - routine used to tell program a piece of memory no longer needed

- note: memory allocated dynamically does not go away at the end of functions, you MUST explicitly free it up

Memory Management Functions

- **malloc()**

  is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form.



Syntax:

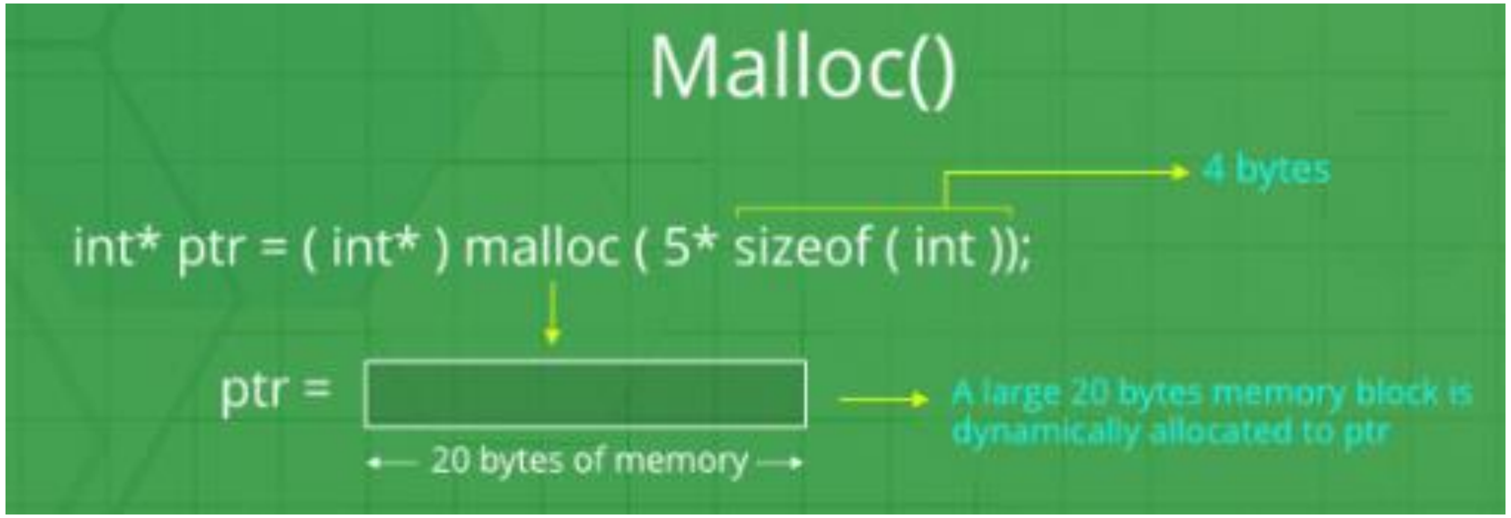
```
ptr = (cast-type*) malloc(byte-size)
```

Memory Management Functions

- **For Example:**

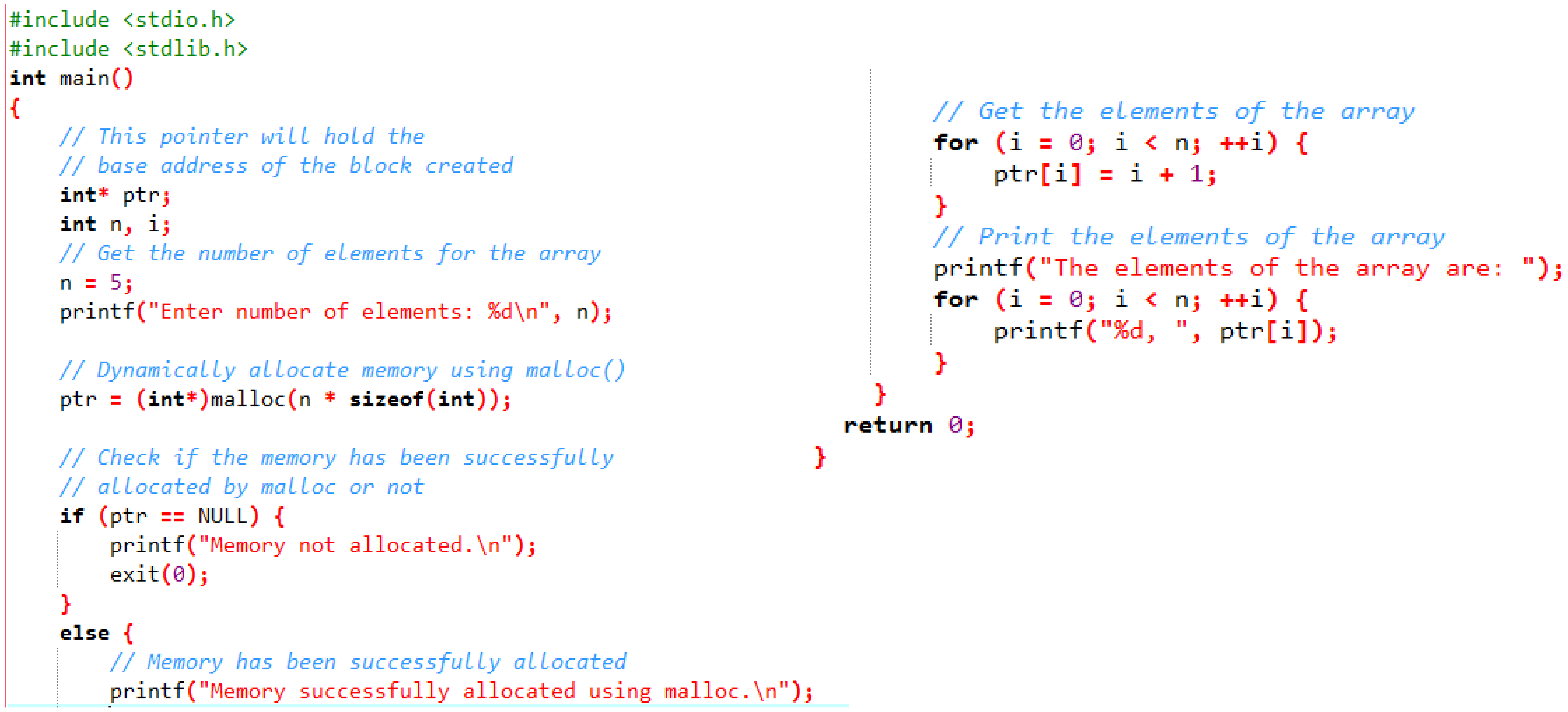
ptr = (int*) malloc(sizeof (int));

Since the size of int is 4 bytes, this statement will allocate 20 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.


Malloc()
4 bytes
int* ptr = (int*) malloc (5* sizeof (int));
ptr =
20 bytes of memory
A large 20 bytes memory block is dynamically allocated to ptr

Memory Management Functions

Example:

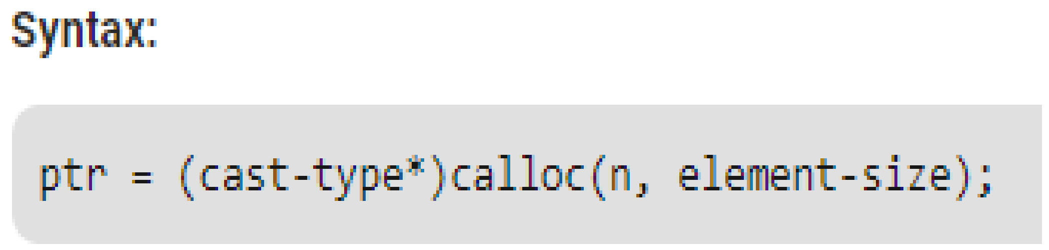


```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;
    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using malloc()
    ptr = (int*)malloc(n * sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }
        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }
  return 0;
}
```

Memory Management Functions

- **calloc()**

  “calloc” or **“contiguous allocation”** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value ‘0’.



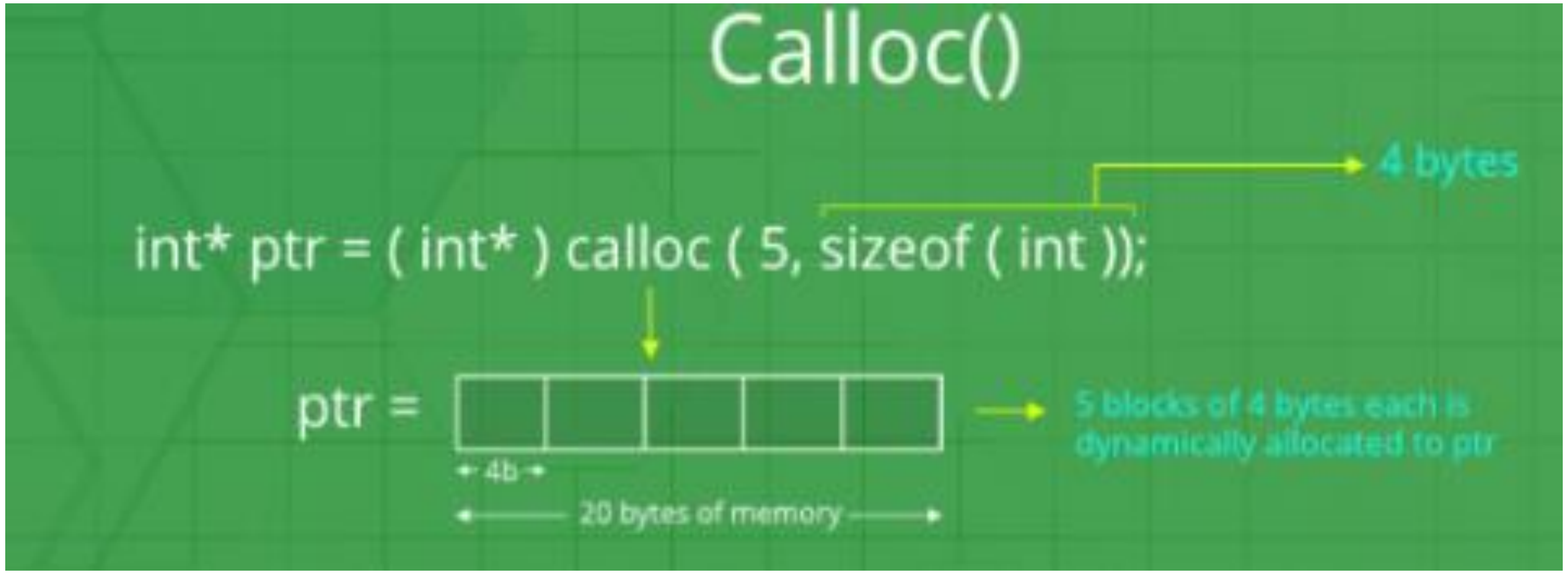
Syntax:

```
ptr = (cast-type*)calloc(n, element-size);
```

Memory Management Functions

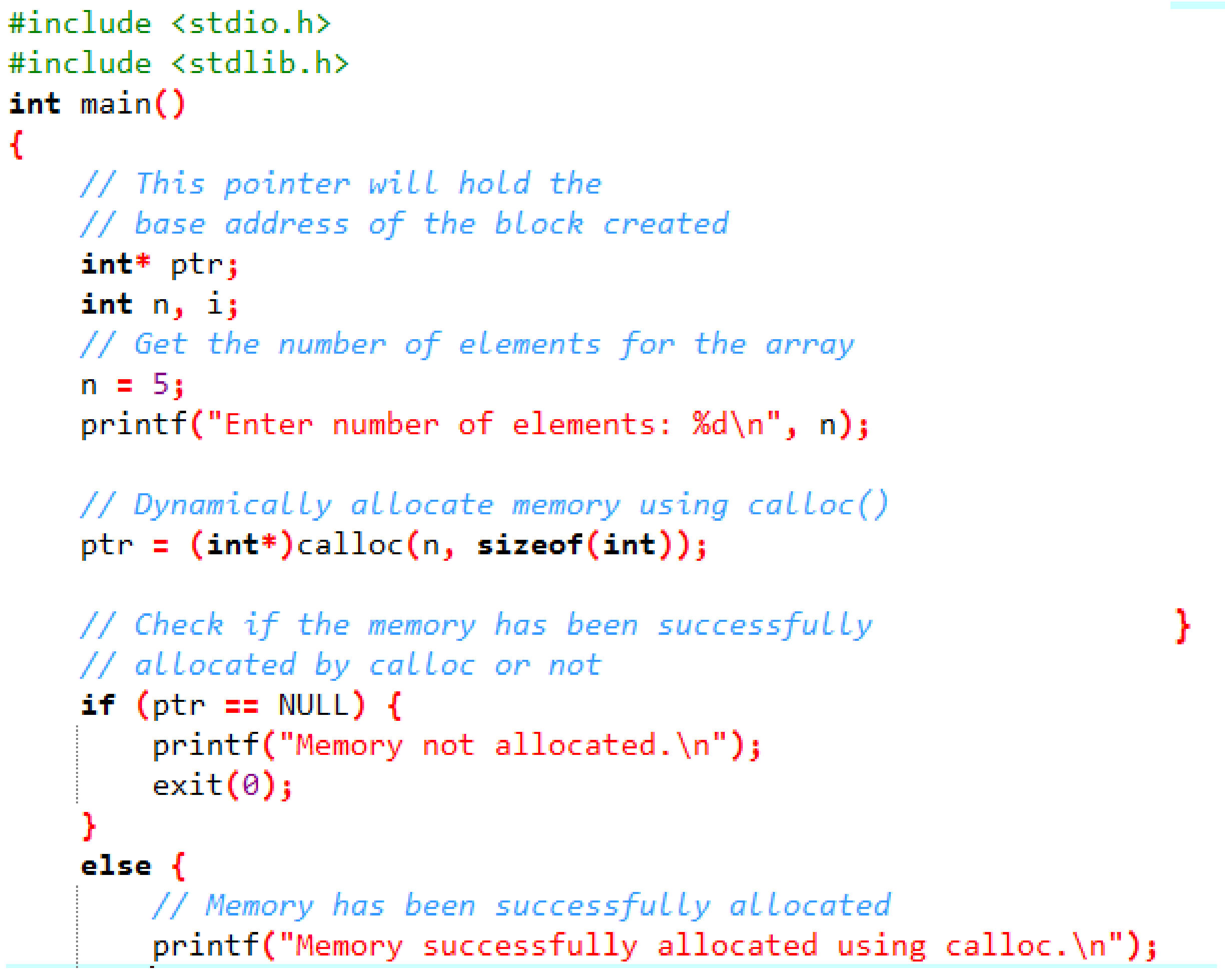
ptr = (float*) calloc (25, sizeof(float));

This statement allocates contiguous space in memory for 25 elements each with the size of the float.

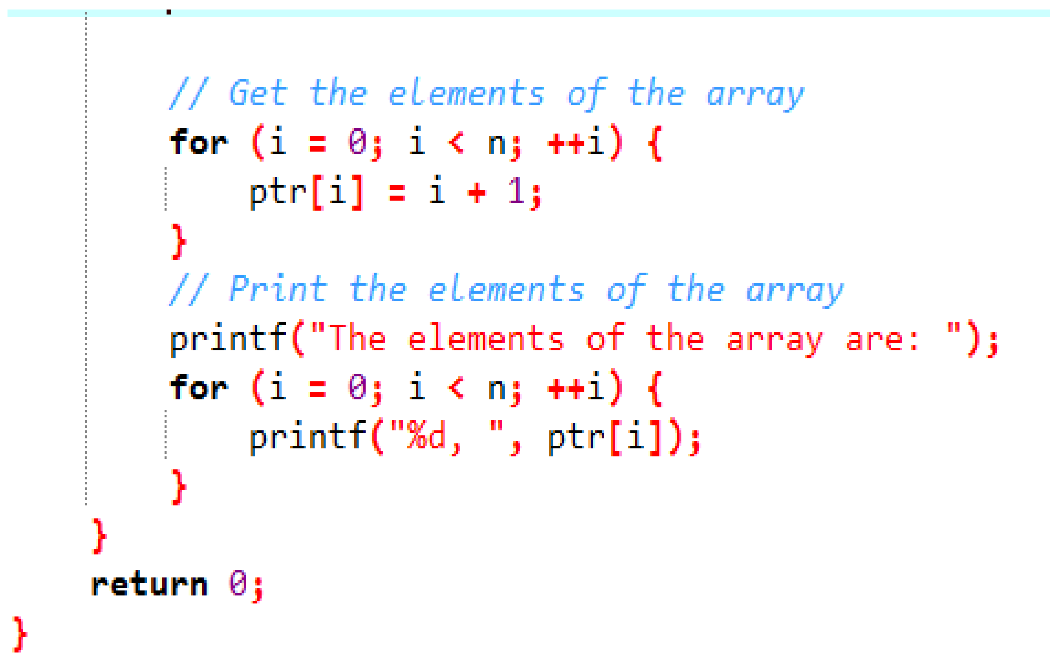


```
int* ptr = ( int* ) calloc ( 5, sizeof ( int ));
```

Memory Management Functions



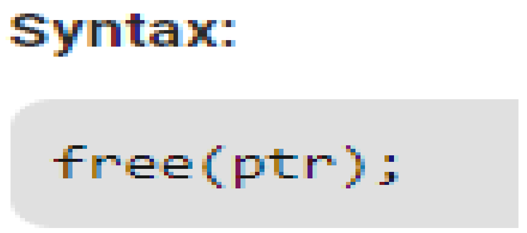
```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;
    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by calloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
        // Memory has been successfully allocated
        printf("Memory successfully allocated using calloc.\n");
```



```c
        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }
        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }
    return 0;
}
```

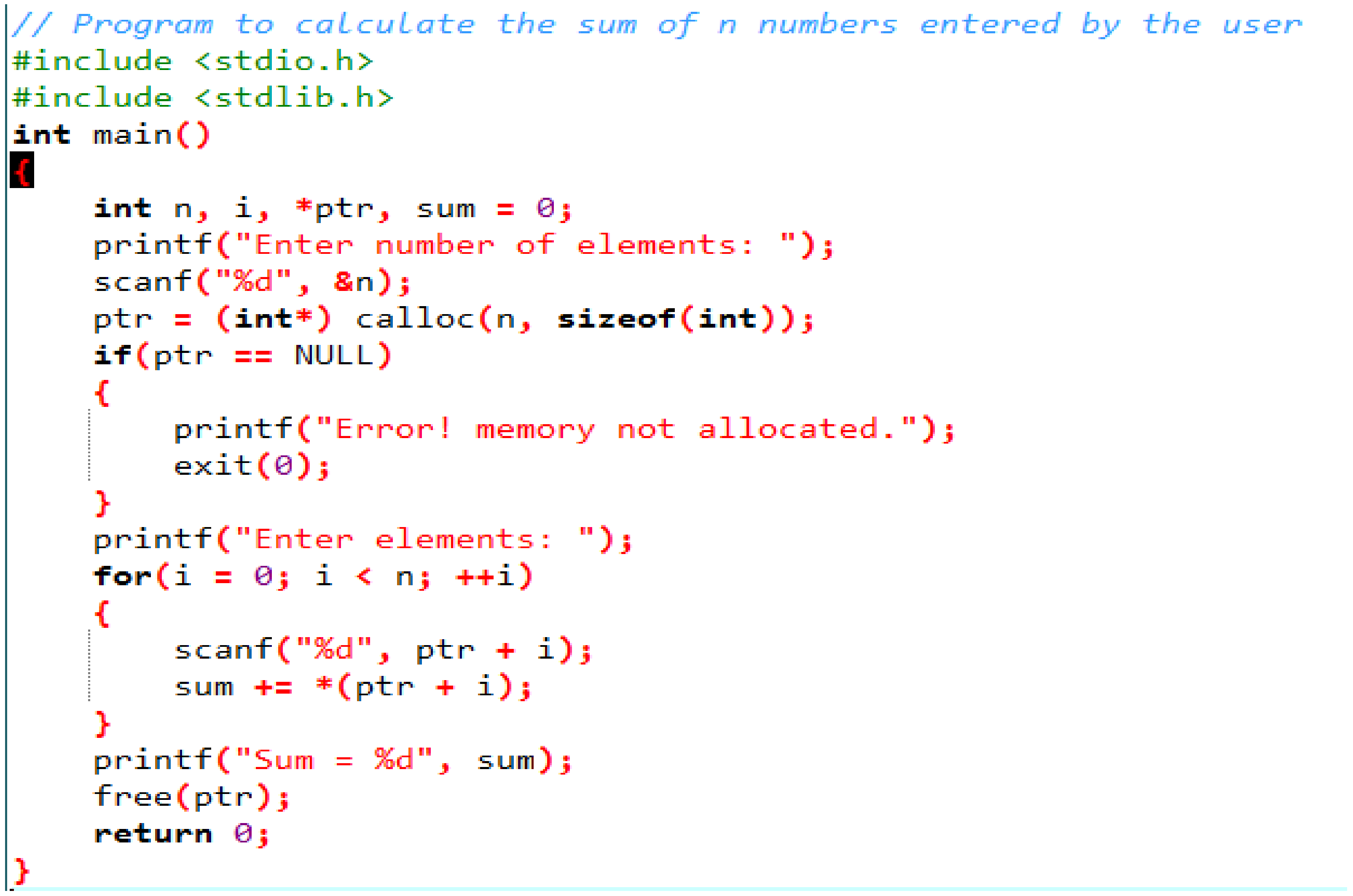
Memory Management Functions

- **Free()**
- is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.


Syntax:

```
free(ptr);
```

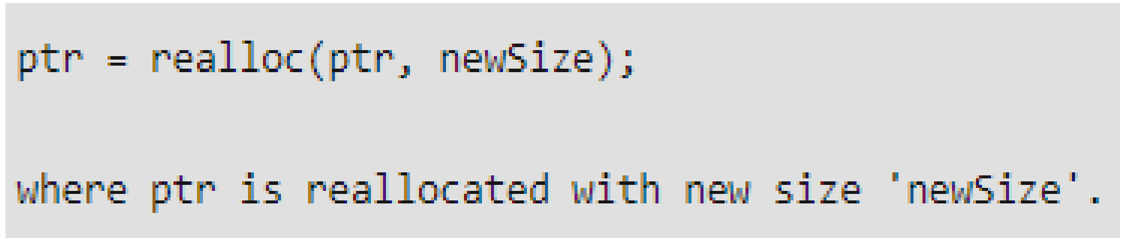
Memory Management Functions



```
// Program to calculate the sum of n numbers entered by the user
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n, i, *ptr, sum = 0;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    ptr = (int*) calloc(n, sizeof(int));
    if(ptr == NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }
    printf("Enter elements: ");
    for(i = 0; i < n; ++i)
    {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }
    printf("Sum = %d", sum);
    free(ptr);
    return 0;
}
```

Memory Management Functions

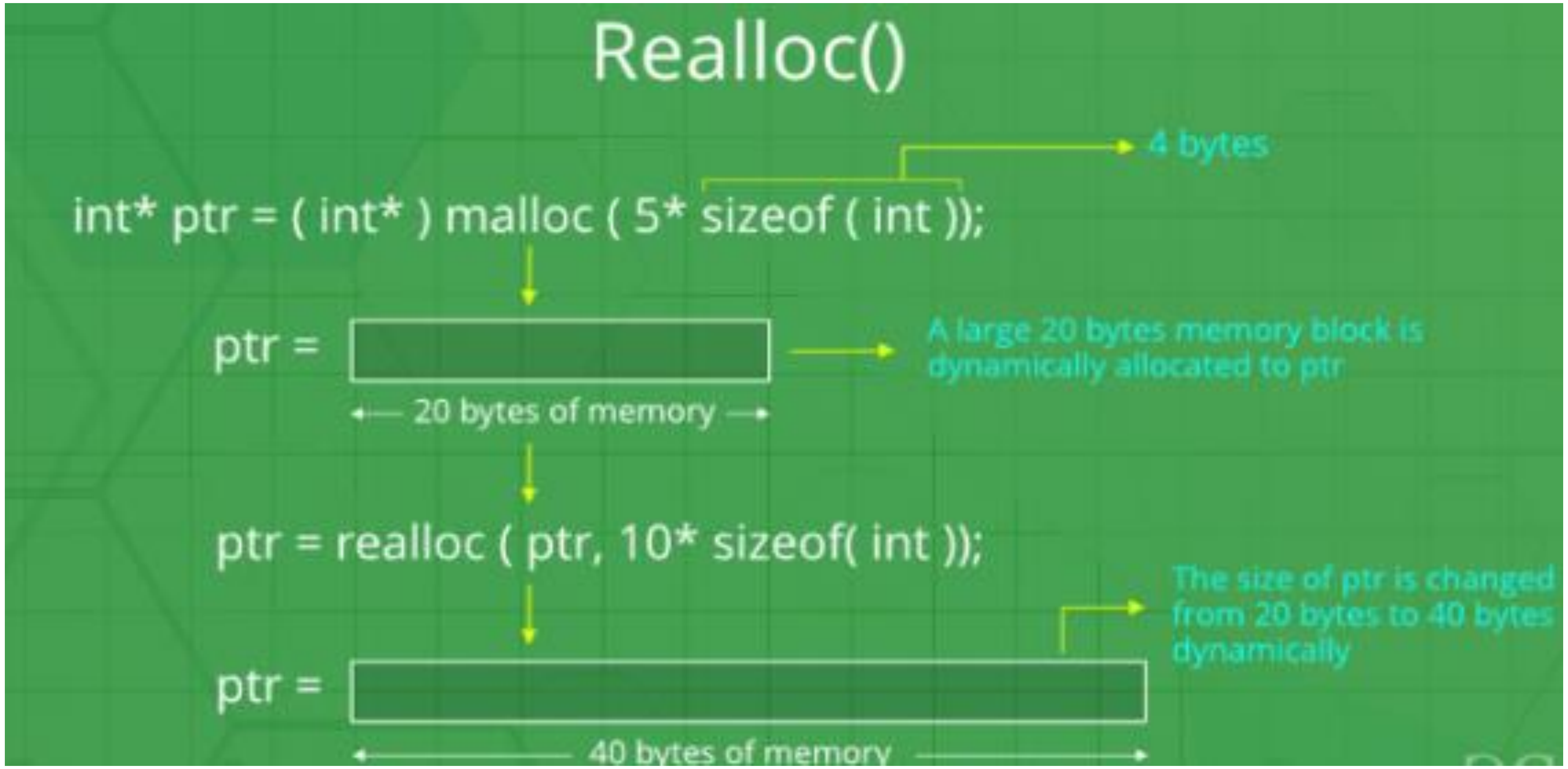
- **realloc()**
- **“realloc”** or **“re-allocation”** method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**.
- Syntax



```
ptr = realloc(ptr, newSize);

where ptr is reallocated with new size 'newSize'.
```

Memory Management Functions

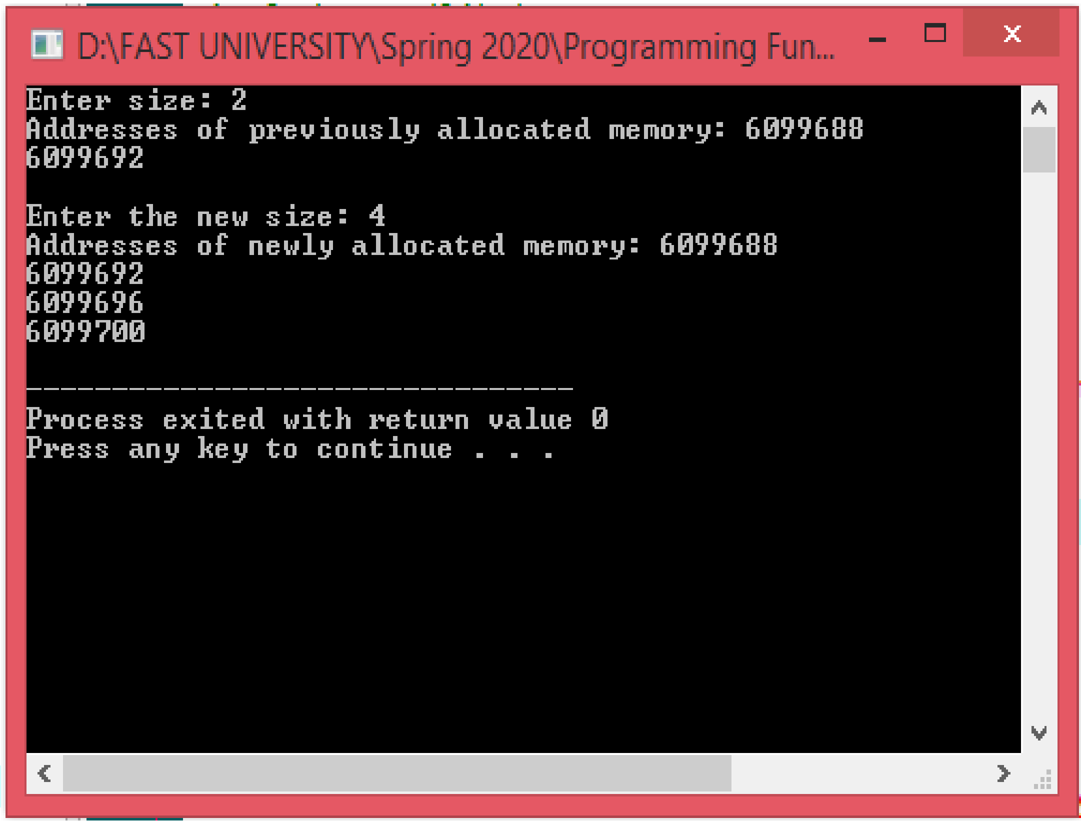
- **realloc()**


Realloc()
4 bytes
int* ptr = (int*) malloc (5* sizeof (int));
ptr =
A large 20 bytes memory block is dynamically allocated to ptr
20 bytes of memory
ptr = realloc (ptr, 10* sizeof(int));
The size of ptr is changed from 20 bytes to 40 bytes dynamically
ptr =
40 bytes of memory

Memory Management Functions

- **realloc(): Example**

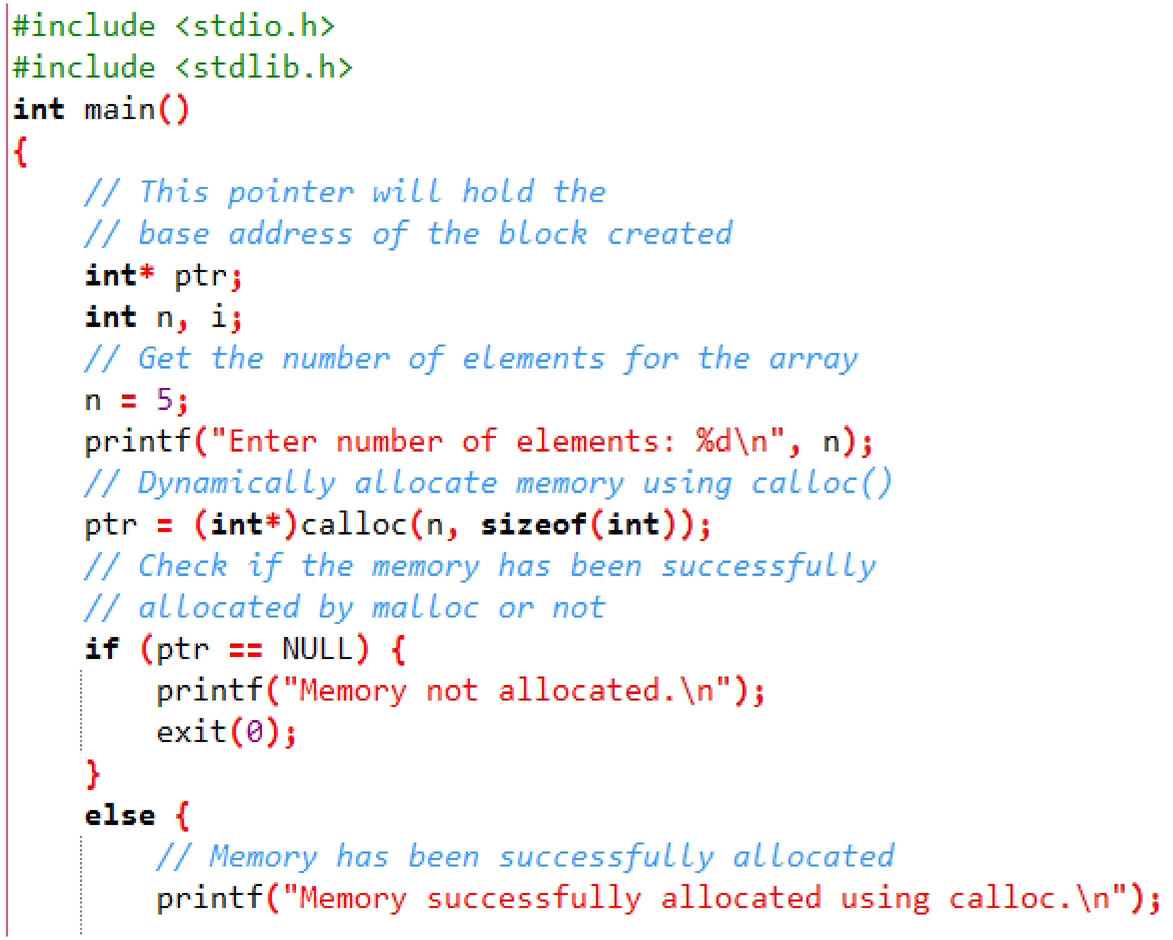
```c
#include <stdlib.h>
int main()
{
    int *ptr, i , n1, n2;
    printf("Enter size: ");
    scanf("%d", &n1);

    ptr = (int*) malloc(n1 * sizeof(int));
    printf("Addresses of previously allocated memory: ");
    for(i = 0; i < n1; ++i)
         printf("%u\n",ptr + i);

    printf("\nEnter the new size: ");
    scanf("%d", &n2);
    // rellocating the memory
    ptr = realloc(ptr, n2 * sizeof(int));

    printf("Addresses of newly allocated memory: ");
    for(i = 0; i < n2; ++i)
         printf("%u\n", ptr + i);

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


D:\FAST UNIVERSITY\Spring 2020\Programming Fun...

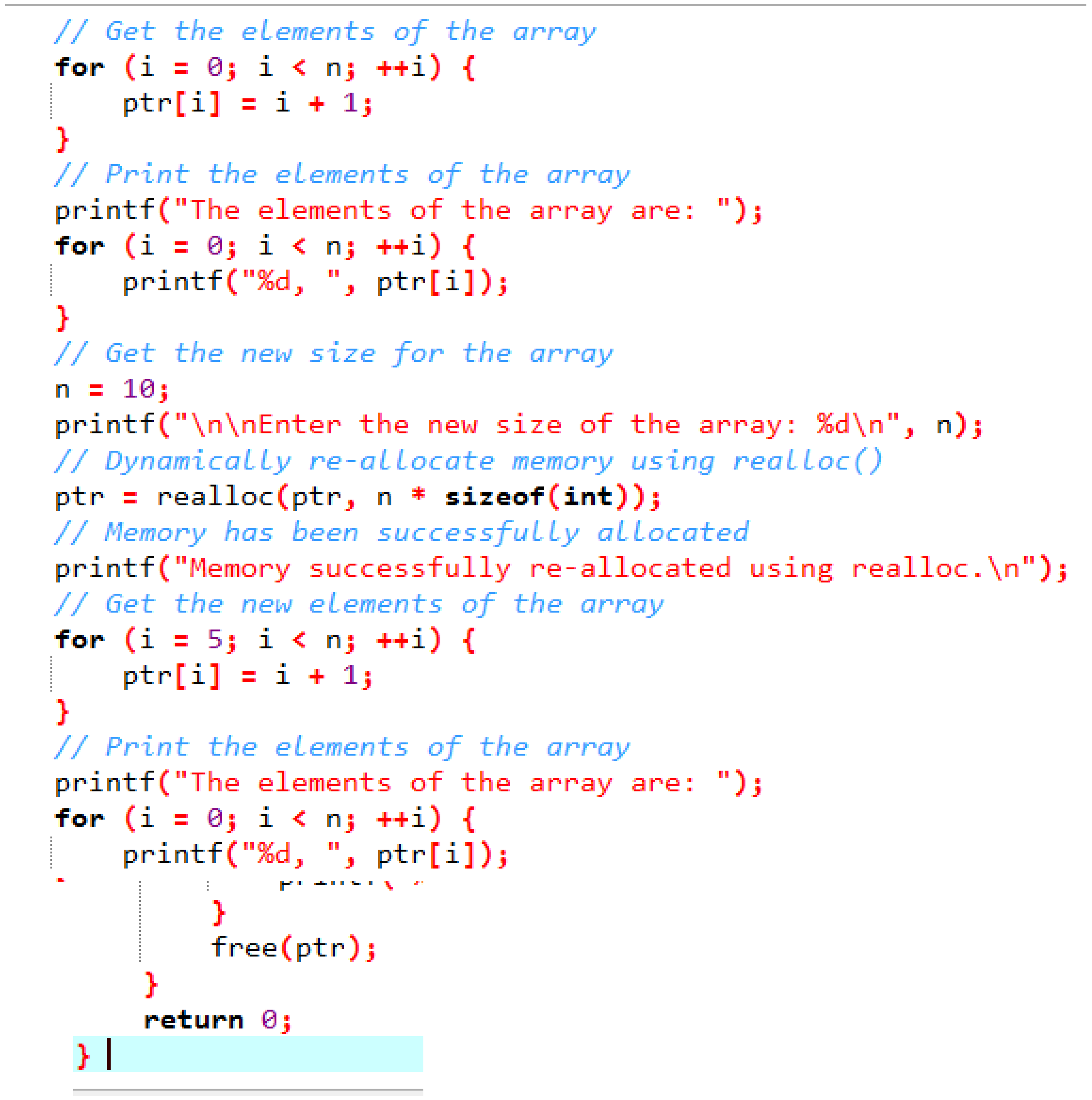
```
Enter size: 2
Addresses of previously allocated memory: 6099688
6099692

Enter the new size: 4
Addresses of newly allocated memory: 6099688
6099692
6099696
6099700

--------------------------------

Process exited with return value 0
Press any key to continue . . .
```

Memory Management Functions

- **realloc(): Example**



```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;
    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);
    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));
    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
        // Memory has been successfully allocated
        printf("Memory successfully allocated using calloc.\n");
```



```c
// Get the elements of the array
for (i = 0; i < n; ++i) {
    ptr[i] = i + 1;
}
// Print the elements of the array
printf("The elements of the array are: ");
for (i = 0; i < n; ++i) {
    printf("%d, ", ptr[i]);
}
// Get the new size for the array
n = 10;
printf("\n\nEnter the new size of the array: %d\n", n);
// Dynamically re-allocate memory using realloc()
ptr = realloc(ptr, n * sizeof(int));
// Memory has been successfully allocated
printf("Memory successfully re-allocated using realloc.\n");
// Get the new elements of the array
for (i = 5; i < n; ++i) {
    ptr[i] = i + 1;
}
// Print the elements of the array
printf("The elements of the array are: ");
for (i = 0; i < n; ++i) {
    printf("%d, ", ptr[i]);
```

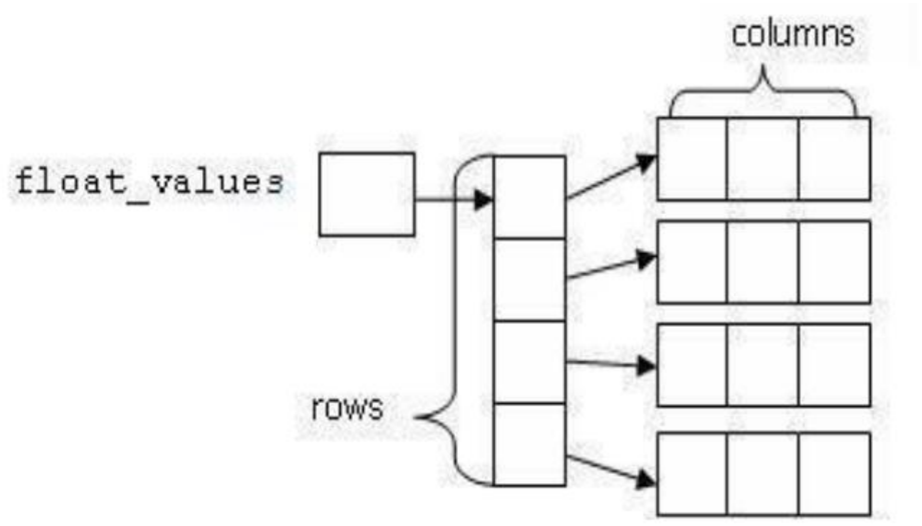
```c
        }
        free(ptr);
    }
    return 0;
}
```

DYNAMIC 2D-ARRAY ALLOCATION

Think of the memory to be allocated to a pointer to pointer variable as two dimensional. It has 'rows' and 'columns'; i.e., if the size is m x n, there will be 'm' rows, and for each row, there will be 'n' columns.

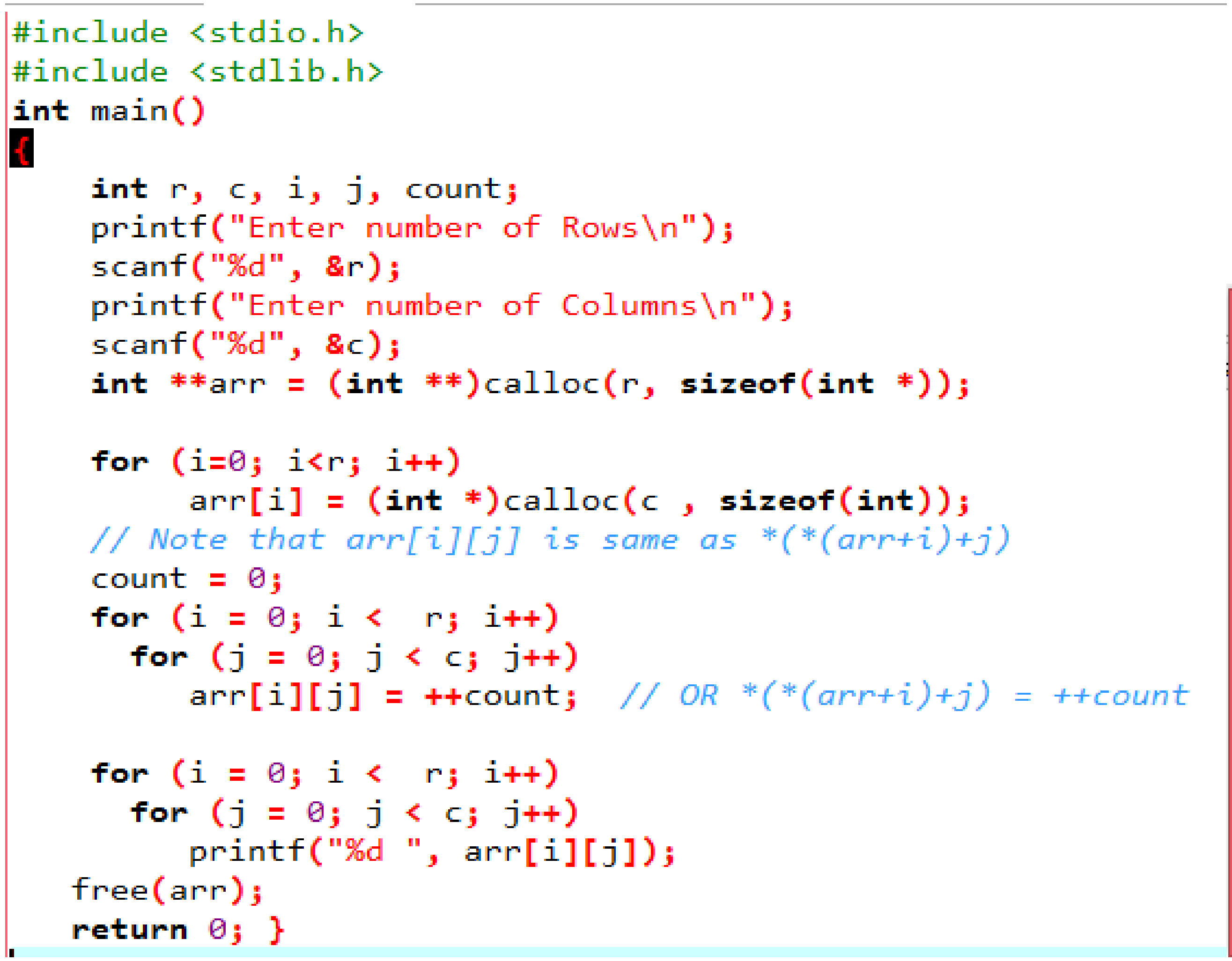
- First, allocate memory for 'm' rows.
- Secondly, allocate 'n' columns for each of the 'm' rows.

Allocating memory to a pointer to pointer to float values. Let the number of rows be '4' and the number of columns '3'.

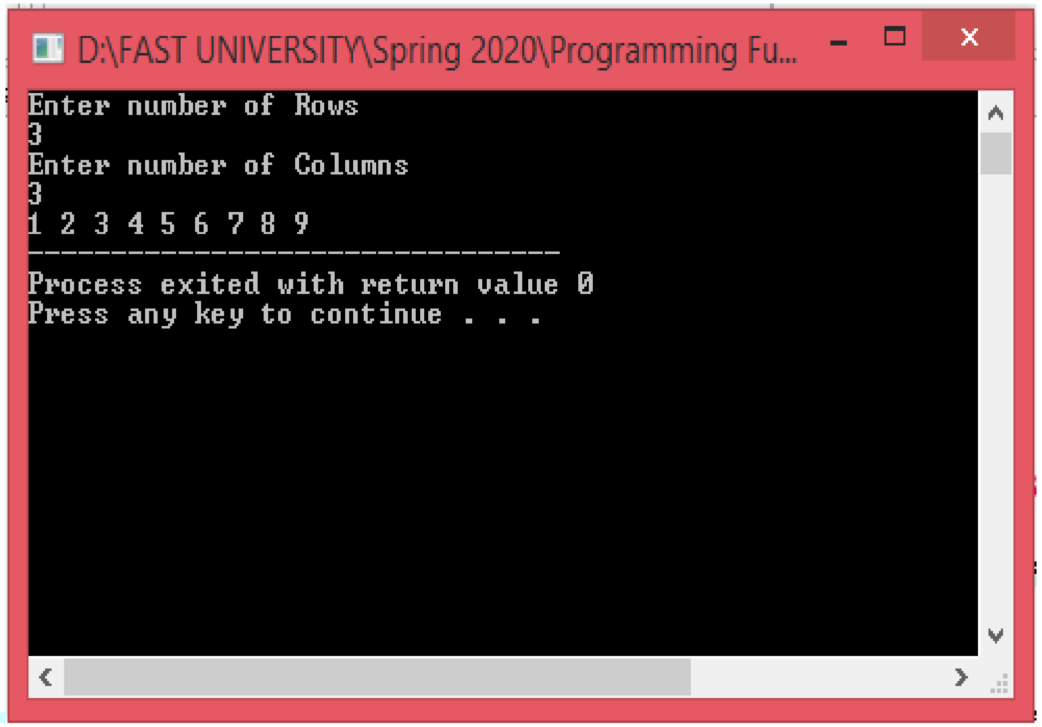


2D array using DMA

- **Using pointer to a pointer**



```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int r, c, i, j, count;
    printf("Enter number of Rows\n");
    scanf("%d", &r);
    printf("Enter number of Columns\n");
    scanf("%d", &c);
    int **arr = (int **)calloc(r, sizeof(int *));

    for (i=0; i<r; i++)
         arr[i] = (int *)calloc(c , sizeof(int));
    // Note that arr[i][j] is same as *(*(arr+i)+j)
    count = 0;
    for (i = 0; i <  r; i++)
      for (j = 0; j < c; j++)
         arr[i][j] = ++count;  // OR *(*(arr+i)+j) = ++count

    for (i = 0; i <  r; i++)
      for (j = 0; j < c; j++)
         printf("%d ", arr[i][j]);
   free(arr);
   return 0; }
```



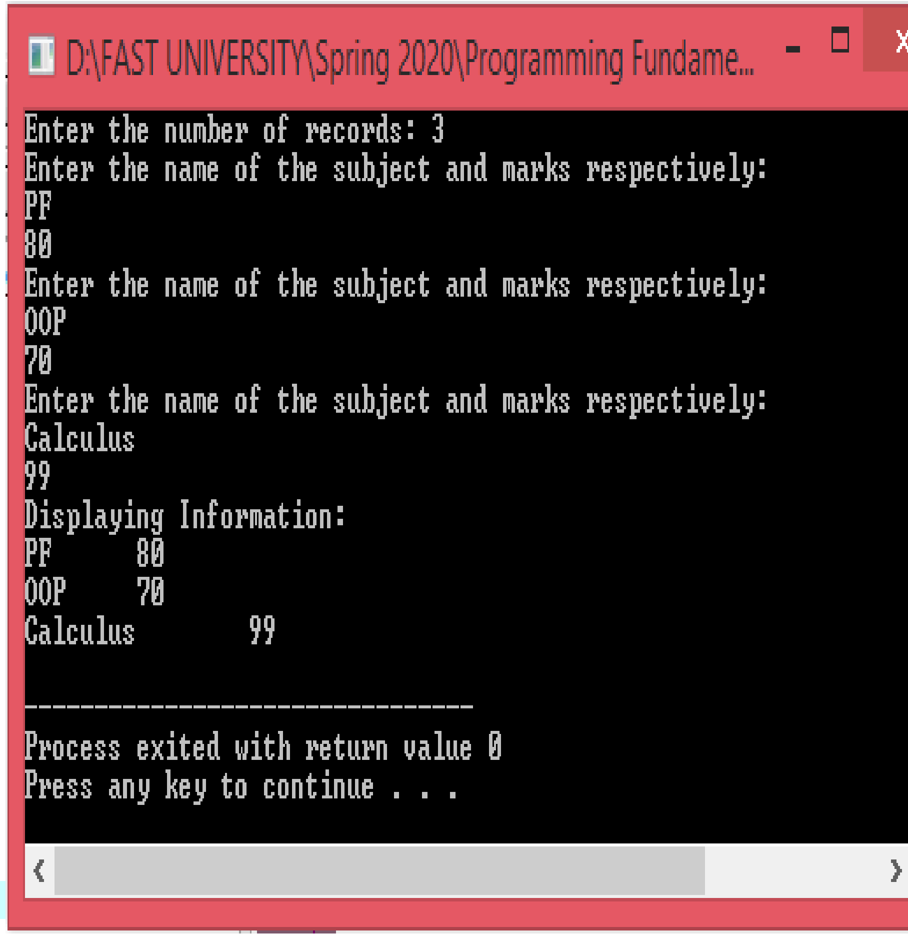
D:\FAST UNIVERSITY\Spring 2020\Programming Fu...

```
Enter number of Rows
3
Enter number of Columns
3
1 2 3 4 5 6 7 8 9
--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

Dynamic Memory Allocation for Structure

- This program asks the user to store the value of noOfRecords and allocates the memory for the noOfRecords structure variables dynamically using the malloc() function.

```c
#include <stdio.h>
#include <stdlib.h>
struct course {
    int marks;
    char subject[30];
};
int main() {
    struct course *ptr;
    int i, noOfRecords;
    printf("Enter the number of records: ");
    scanf("%d", &noOfRecords);

    // Memory allocation for noOfRecords structures
    ptr = (struct course *) malloc(noOfRecords * sizeof(struct course));
    for (i = 0; i < noOfRecords; ++i) {
        printf("Enter the name of the subject and marks respectively:\n");
        scanf("%s %d", &(ptr + i)->subject, &(ptr + i)->marks);
    }

    printf("Displaying Information:\n");
    for (i = 0; i < noOfRecords; ++i)
        printf("%s\t%d\n", (ptr + i)->subject, (ptr + i)->marks);
    return 0;
}
```



```
Enter the number of records: 3
Enter the name of the subject and marks respectively:
PF
80
Enter the name of the subject and marks respectively:
OOP
70
Enter the name of the subject and marks respectively:
Calculus
99
Displaying Information:
PF      80
OOP     70
Calculus        99

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```