


# Advanced Procedural Programming




Lecture 3: Dynamic Memory Allocation

Gemma O'Callaghan

## Overview


- Types of Memory Management in C
- Why Dynamic Memory Management?
- Memory Management Functions:
  - Malloc
  - Calloc
  - Realloc
  - Free
- Common Errors



2

## Memory Management in C


- Memory Management in C
  - Static** - Static-duration variables are allocated in main memory, usually along with the executable code of the program, and persist for the lifetime of the program.
  - Automatic** - Automatic-duration variables are allocated on the stack and come and go as functions are called and return.
  - The size of the allocation must be compile-time constant.
  - The lifetime of allocated memory can also be a concern. Neither static- nor automatic-duration memory is adequate for all situations. Automatic-allocated data cannot persist across multiple function calls, while static data persists for the life of the program whether it is needed or not.



3

## Memory Management in C

- If the required size is not known until run-time (for example, if data of arbitrary size is being read from the user or from a disk file), then using fixed-size data objects is inadequate.
- C also does not have automatic garbage collection like Java does. Therefore a C programmer must manage all dynamic memory used during the program execution.
- These limitations are avoided by using dynamic memory allocation in which memory is more explicitly (but more flexibly) managed, typically, by allocating it from the *free store* (informally called the "heap"), an area of memory structured for this purpose.



4

## Why Dynamic Memory Allocation?

- When you don't know how many objects of some kind you need (e.g. linked list elements);
- When you need to have data structures of size known only at runtime (e.g. strings based on unknown input);
- When you know at compile time their size, but it's just too big for the stack. (> 1MB)
- When you need to have an object whose lifetime is different than what automatic variables, which are scope-bound (=>are destroyed when the execution exits from the scope in which they are declared), can have (e.g. data that must be shared between different objects with different lifetimes and deleted when no one uses it anymore).



5

## Memory Management Functions

- If you are unsure of the size in advance there are library functions available to dynamically manage memory.
- Must include `stdlib.h` at start of program to use.
  - `Malloc()`
  - `Calloc()`
  - `Realloc()`
  - `Free()`



6

## Malloc

- Memory Allocation
- This function allocates a contiguous block of memory and returns a void pointer to the start of the allocated block which can then be casted into a pointer of any type.
- Syntax:
  - `ptr = (cast-type*)malloc(byte-size)`
  - where `ptr` is a pointer of cast-type.
  - Returns a pointer to an area of memory with size equal to `byte-size`.
  - If the space is insufficient then the allocation fails and a NULL pointer is returned. **(NB: you must check for this in the code)**



7

## Malloc

- Example:
  - `ptr = (int*)malloc(100*sizeof(int));`
  - This statement will allocate either 200 or 400 according to size of `int` 2 or 4 bytes respectively and the pointer points to the address of the first byte of memory.
- No initialisation so ensure that you initialise with valid values before using.



8

## Malloc Program Example – Find Sum

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)malloc(n*sizeof(int));
    if(ptr==NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }
```



9

## Malloc Program Example – Find Sum

```
printf("Enter elements of array: ");
for(i=0;i<n;++i)
{
    scanf("%d",ptr+i);
    sum+=*(ptr+i);
}
printf("Sum=%d",sum);
free(ptr);
return 0;
}
```



10

## Calloc

- Contiguous Allocation
- This function allocates a contiguous block of memory and returns a pointer to the start of the allocated block and initialises the memory to zeros.
- Syntax:
  - `ptr=(cast-type*)calloc(n,element-size);`
  - This statement will allocate contiguous space in memory for an array of *n* elements.



11

## Calloc

- Example:
  - `ptr=(float*)calloc(25,sizeof(float));`
  - This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.



12

## Calloc Example Program – Find Sum

```
#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);
    }
}
```



13

## Calloc Example Program – Find Sum

```
printf("Enter elements of array: ");
for(i=0;i<n;++i)
{
    scanf("%d",ptr+i);
    sum+=*(ptr+i);
}
printf("Sum=%d",sum);
free(ptr);
return 0;
}
```



14

## Realloc

- If the previously allocated memory is insufficient or more than sufficient. Then, you can change memory size previously allocated using realloc().
- Syntax:
  - `ptr=(cast*)realloc(ptr,newsize);`
- Example:
  - `ptr = (int*)malloc(100*sizeof(int));`
  - `ptr=(int*)realloc(ptr, (200*sizeof(int));`



15

## Realloc Program Example

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *ptr,i,n1,n2;
    printf("Enter size of array: ");
    scanf("%d",&n1);
    ptr=(int*)malloc(n1*sizeof(int));
    printf("Address of previously allocated memory: ");
    for(i=0;i<n1;++i)
        printf("%p\t",ptr+i);
}
```



16

## Realloc Program Example

```
printf("\nEnter new size of array: ");
scanf("%d",&n2);
ptr=(int*)realloc(ptr,n2);
for(i=0;i<n2;++i)
    printf("%p\t",ptr+i);
return 0;
}
```



17

## Free

- Dynamically allocated memory with either `calloc()` or `malloc()` does not get returned on its own.
- The programmer must use `free()` explicitly to release space.
- The argument to `free()` is any address that was returned by a prior call to `malloc` or `calloc`.
- Syntax:
  - `free(ptr);`



18

## Common Errors

- **Not checking for allocation failures.** Memory allocation is not guaranteed to succeed, and may instead return a null pointer. If there's no check for successful allocation implemented, this usually leads to a crash of the program, due to the resulting segmentation fault on the null pointer dereference.
- **Memory leaks.** Failure to deallocate memory using `free` leads to buildup of non-reusable memory, which is no longer used by the program. This wastes memory resources and can lead to allocation failures when these resources are exhausted.



19

## Common Errors

- **Logical errors.** All allocations must follow the same pattern: allocation using `malloc`, usage to store data, deallocation using `free`.
- Failures to adhere to this pattern, such as memory usage after a call to `free` (dangling pointer) or before a call to `malloc` (wild pointer), calling `free` twice ("double free"), etc., usually causes a segmentation fault and results in a crash of the program.
- These errors can be transient and hard to debug – for example, freed memory is usually not immediately reclaimed by the OS, and thus dangling pointers may persist for a while and appear to work.



20