Lecture_2_Memory_Management

February 3, 2025

1 Introduction

Today, we'll dive into the core principles of memory in C:

1. The **stack** and **heap**: automatic (stack-managed) vs manual (heap-managed) memory, and when to use each.
2. **Dynamic memory allocation**: using `malloc`, `calloc` and `realloc` to allocate memory at runtime.
3. Properly freeing memory with `free` to avoid memory leaks.
4. Addressing problems like memory leaks, dangling pointers, and fragmentation.

Mastering these concepts is essential for writing efficient, reliable, and bug-free C programs.

---

2 Working with memory

Understanding memory is an important aspect of C programming. When you declare a variable using a basic data type, C automatically allocates space for the variable in an area of memory called the `stack`.

Figure 1: Demonstration of stack behavior. Nothing moves physically when variables are pushed onto or popped off of a stack in memory. Only the values stored in the memory managed by the stack are changed, as illustrated here. It's possible and common to intermix push and pop operations. Source: weber.edu

An `int` variable, for example, is typically allocated 4 bytes of memory when declared. We can confirm this by using the `sizeof` operator:

[1]:
```
%%writefile int-size.c

#include <stdio.h>

int main() {
  int x;
  printf("%ld", sizeof(x));
  return 0;
}
```

```
Writing int-size.c
```

[2]:
```
%%script bash
gcc int-size.c -o int-size
./int-size
```

```
4
```

As another example, an array with a specified size is allocated **contiguous blocks** of memory with each block the size for one element:

[3]:
```
%%writefile array-size.c

#include <stdio.h>

int main() {
  int arr[10];
  printf("%ld", sizeof(arr));
  return 0;
}
```

```
Writing array-size.c
```

[4]:
```
%%script bash
gcc array-size.c -o array-size
./array-size
```

```
40
```

So long as your program explicitly declares a basic data type or an array size, memory is automatically managed. However, you have probably already been wishing to implement a program where the array size is undecided until runtime.

Dynamic memory allocation is the process of allocating and freeing memory as needed. Now you can prompt at runtime for the number of array elements and then create an array with that many elements. Dynamic memory is managed with pointers that point to newly allocated blocks of memory in an area called the **heap**.

Figure 2: Demonstration of heap behavior. The heap allocates memory by finding and returning the first memory block large enough to satisfy the request. Memory is returned or freed in any convenient order. When the program deallocates or releases two adjacent memory blocks, the heap merges them to form a single block. Doing this allows the heap to meet future demands for large memory blocks. The cross-hatched block illustrates a request for a large block (twice the size of the colored blocks) of memory. Source: weber.edu

In addition to automatic memory management using the stack and dynamic memory allocation using the heap, there is **statically managed data** in **main memory** for variables that persist for the lifetime of the program.

Key differences between `stack` and `heap` allocations:

Parameter	STACK	HEAP
Basic	Memory is allocated in a contiguous block.	Memory is allocated in any random order.
Allocation and De-allocation	Automatic by compiler instructions.	Manual by the programmer.
Cost	Less	More
Implementation	Easy	Hard
Access time	Faster	Slower
Main Issue	Shortage of memory	Memory fragmentation
Locality of reference	Excellent	Adequate
Safety	Thread safe, data stored can only be accessed by the owner	Not Thread safe, data stored visible to all threads
Flexibility	Fixed-size	Resizing is possible
Data type structure	Linear	Hierarchical
Preferred	Static memory allocation is preferred in an array.	Heap memory allocation is preferred in the linked list.
Size	Smaller than heap memory.	Larger than stack memory.

The `stdlib.h` library includes memory management functions. The statement `#include <stdlib.h>` at the top of your program gives you access to the following:

- `malloc(size)` — dynamically allocates `size` bytes of memory in the **heap** and returns a pointer to the allocated memory.
- `calloc(num_items, item_size)` — dynamically allocates memory in the **heap** and returns a pointer to a contiguous block of memory that has `num_items` items, each of size `item_size` bytes. Typically used for arrays, structures, and other derived data types. The allocated memory is initialized to 0.
- `realloc(ptr, new_size)` — resizes the memory pointed to by `ptr` to `new_size` bytes. The newly allocated memory is not initialized.
- `free(ptr)` — releases the block of memory pointed to by `ptr`.

  When you no longer need a block of allocated memory, use the function `free()` to make the block available to be allocated again.

3 The `malloc` function

The `malloc()` function allocates a specified number of contiguous bytes in memory. For example:

[5]:
```
%%writefile malloc.c

#include <stdio.h>
#include <stdlib.h>
int main() {
  int *ptr;
  ptr = malloc(10*sizeof(*ptr));  /* a block of 10 ints */
```

```
  if (ptr != NULL) {
    *(ptr+2) = 20;  /* assign 20 to third int */
  }
  printf("3rd element equals to %d", *(ptr + 2));
  return 0;
}
```

```
Overwriting malloc.c
```

[6]:
```
%%script bash
gcc malloc.c -o malloc
./malloc
```

```
3rd element equals to 20
```

The allocated memory is **contiguous** and can be treated as an **array**. Instead of using brackets `[ ]` to refer to elements, pointer arithmetic is used to traverse the array. You are advised to use `+` to refer to array elements. Using `++` or `+=` changes the address stored by the pointer.

If the allocation is unsuccessful, `NULL` is returned. Because of this, you should include code to check for a `NULL` pointer.

> A simple two-dimensional array requires `(rows*columns)*sizeof(datatype)` bytes of memory.

4 The `free` function

The `free()` function is a memory management function that is called to **release memory**. By freeing memory, you make more available for use later in your program. For example:

[7]:
```
%%writefile free.c

#include <stdio.h>
#include <stdlib.h>
int main() {
  int *ptr;
  ptr = malloc(10*sizeof(*ptr));  /* a block of 10 ints */
  if (ptr != NULL) {
    *(ptr+2) = 20;  /* assign 20 to third int */
  }
  printf("%d\n", *(ptr + 2));

  free(ptr);

  return 0;
}
```

```
Overwriting free.c
```

[8]:
```
%%script bash
gcc free.c -o free
./free
```

```
20
```

5 The calloc function

`calloc()` stands for contiguous allocation and is used for dynamically allocating memory in the heap. It allocates memory based on the size of a specific item, such as a structure.

Key differences between `malloc` and `calloc`

Feature	`malloc`	`calloc`
Initialization	Does not initialize memory (may contain garbage values).	Initializes memory to zero.
Number of Arguments	Takes 1 argument: the total size in bytes.	Takes 2 arguments: number of items and the size of each item.
Performance	Slightly faster (no initialization).	Slightly slower due to zeroing the memory.

The program below uses `calloc` to allocate memory for a structure and `malloc` to allocate memory for the string within the structure:

[9]:
```
%%writefile calloc.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
  int num;
  char *info;
} record;

int main() {
  record *recs;
  int num_recs = 3;
  int k;
  char str[ ] = "This is information";

  recs = calloc(num_recs, sizeof(record));
  if (recs != NULL) {
    for (k = 0; k < num_recs; k++) {
      (recs+k)->num = k;
      (recs+k)->info = malloc(sizeof(str));
      strcpy((recs+k)->info, str);
```

```
    }
  }

  for (k = 0; k < num_recs; k++) {
    printf("%d\t%s\n", (recs+k)->num, (recs+k)->info);
  }

  return 0;
}
```

```
Overwriting calloc.c
```

[10]:
```
%%script bash
gcc calloc.c -o calloc
./calloc
```

```
0       This is information
1       This is information
2       This is information
```

`calloc` allocates blocks of memory within a **contiguous block** of memory for an array of structure elements. You can navigate from one structure to the next with pointer arithmetic.

After allocating room for a structure, memory must be allocated for the string within the structure. Using a pointer for the `info` member allows any length string to be stored.

Dynamically allocated structures are the basis of **linked lists** and **binary trees** as well as other data structures.

6 The `realloc` function

The `realloc()` function expands a current block to include additional memory. For example:

[11]:
```
%%writefile realloc.c

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

int main() {
  int *ptr;
  ptr = malloc(10*sizeof(*ptr));  /* a block of 10 ints */
  if (ptr != NULL) {
    *(ptr+2) = 20;  /* assign 20 to third int */
  }
  //ptr = realloc(ptr, 100*sizeof(*ptr)); /* 100 ints */
  *(ptr+30) = 66;
  printf("%d %d", *(ptr+2), *(ptr+30));

  return 0;
```

```
}
```

```
Overwriting realloc.c
```

[12]:
```
%%script bash
gcc realloc.c -o realloc
./realloc
```

```
20 66
```

`realloc` leaves the original content in memory and expands the block to allow for more storage.

7 Allocating memory for strings

When allocating memory for a string pointer, you may want to use string length rather than the sizeof operator for calculating bytes. Consider the following program:

[13]:
```
%%writefile strings.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
  char str20[20];
  char *str = NULL;

  strcpy(str20, "12345");
  printf("str20 size: %ld\n", sizeof(str20));
  printf("str20 length: %ld\n", strlen(str20));
  str = malloc(strlen(str20)+1); /* make room for \0 */
  strcpy(str, str20);
  printf("%s", str);

  return 0;
}
```

```
Overwriting strings.c
```

[14]:
```
%%script bash
gcc strings.c -o strings
./strings
```

```
str20 size: 20
str20 length: 5
12345
```

The `strcpy` function in C is used to copy a string from one character array (source) to another (destination):

`strcpy(destination, source)`

It is part of the `<string.h>` library.

This approach is better memory management because you aren't allocating more space than is needed to a pointer. When using **`strlen`** to determine the number of bytes needed for a string, be sure to include one extra byte for the `NULL` character `'\0'`.

A `char` is always one byte, so there is no need to multiply the memory requirements by `sizeof(char)`.

8 Dynamic Arrays

Many algorithms implement a **dynamic array** because this allows the number of elements to grow as needed.

Because elements are not allocated all at once, dynamic arrays typically use a structure to keep track of current array size, current capacity, and a pointer to the elements, as in the following program:

[15]:
```
%%writefile dynamic-arrays.c

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

typedef struct {
  int *elements;
  int size;
  int cap;
} dyn_array;

int main() {
  dyn_array arr;
  int i;

  /* initialize array */
  arr.size = 0;
  arr.elements = calloc(1, sizeof(*arr.elements));
  arr.cap = 1;  /* room for 1 element */

  /* expand by 5 more elements */
  arr.elements = realloc(arr.elements, (5 + arr.cap)*sizeof(*arr.elements));
  if (arr.elements != NULL)
    arr.cap += 5; /* increase capacity */

  /* add an element and increase size */
  if (arr.size < arr.cap) {
    arr.elements[arr.size] = 20; /* add element to array */
    arr.size++;
```

```
  }
  else
    printf("Need to expand array.");

  /* display array elements */
  for (i = 0; i < arr.cap; i++)
    printf("Element %d: %d\n", i, arr.elements[ i ]);

  return 0;
}
```

```
Overwriting dynamic-arrays.c
```

[16]:
```
%%script bash
gcc dynamic-arrays.c -o dynamic-arrays
./dynamic-arrays
```

```
Element 0: 20
Element 1: 0
Element 2: 0
Element 3: 0
Element 4: 0
Element 5: 0
```

The entire program is written in main() for demonstration purposes. To properly implement a dynamic array, sub-tasks should be broken down into functions such as `init_array()`, `increase_array()`, `add_element()`, and `display_array()`. The error checking was also skipped to keep the demo short.

9 Addressing Memory Management Issues in C

9.1 Memory Leaks

A memory leak occurs when dynamically allocated memory is not explicitly deallocated, it remains occupied, leading to memory leaks or if all references to allocated memory are lost before calling `free()`, the memory remains allocated but inaccessible.

[17]:
```
%%writefile leak.c

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

void leak() {
  int *p = malloc(sizeof(int)); // Allocating memory
  if (p == NULL) {
    printf("Memory allocation failed!\n");
    return;
  }
```

```
  p = NULL;  // Lost reference to allocated memory (leak)
}

int main() {
    leak();
    printf("Memory leak occurred!\n");
    return 0;
}
```

```
Overwriting leak.c
```

[18]:
```
%%script bash
gcc leak.c -o leak
./leak
```

```
Memory leak occurred!
```

9.2 Dangling Pointers

Accessing memory after it has been **freed** leads to **undefined behavior**, as the memory may be reused or overwritten by other parts of the program. This is known as using a **dangling pointer**, which can cause crashes or unpredictable results.

```
int *p = malloc(sizeof(int));
free(p);
*p = 10; // Undefined behavior!
```

9.3 Double Free

Double free leads to undefined behavior. It can cause a crash, memory corruption, or other unpredictable results.

```
free(p);
free(p); // Crash or corruption
```

9.4 Fragmentation

Frequent allocation and deallocation of memory can create small, fragmented memory blocks that are too tiny for future allocations, leading to inefficient memory usage and potential performance issues. This phenomenon is known as memory fragmentation.

[19]:
```
%%writefile fragmentation.c

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

int main() {
    // Allocate three separate memory blocks
    int *a = malloc(100 * sizeof(int));
    int *b = malloc(50 * sizeof(int));
```

```
    int *c = malloc(200 * sizeof(int));

    // Free the first and third blocks, leaving fragmented memory
    free(a);
    free(c);

    // Now, try to allocate a large block
    int *d = malloc(250 * sizeof(int));

    if (d == NULL) {
        printf("Memory allocation failed due to fragmentation!\n");
    } else {
        printf("Memory allocation successful!\n");
        free(d); // Free allocated memory
    }

    free(b); // Free remaining memory
    return 0;
}
```

```
Overwriting fragmentation.c
```

[20]:
```
%%script bash
gcc fragmentation.c -o fragmentation
./fragmentation
```

```
Memory allocation successful!
```

[20]:
```
```