C Programming - Deck 18

Dynamic Memory Allocation

Prof. Jyotiprakash Mishra
mail@jyotiprakash.org

Dynamic Memory Allocation

- Memory allocated at runtime, not compile time
- Allocated from heap memory
- Size can be determined during program execution
- Must be manually freed to avoid memory leaks
- Four key functions: malloc, calloc, realloc, free
- Header file: `<stdlib.h>`
- Returns NULL if allocation fails
- Always check for NULL before using

Memory Allocation Functions

- **malloc()**: Allocates uninitialized memory
- Syntax: `void* malloc(size_t size);`
- **calloc()**: Allocates zero-initialized memory
- Syntax: `void* calloc(size_t num, size_t size);`
- **realloc()**: Resizes previously allocated memory
- Syntax: `void* realloc(void* ptr, size_t size);`
- **free()**: Deallocates memory
- Syntax: `void free(void* ptr);`

Program 1: Basic malloc and free

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  int *ptr;
  ptr = (int*)malloc(sizeof(int));
  if (ptr == NULL) {
    printf("Memory allocation failed\n");
    return 1;
  }
  *ptr = 42;
  printf("Value: %d\n", *ptr);
  printf("Address: %p\n", (void*)ptr);
  free(ptr);
  printf("Memory freed\n");
  return 0;
}
```

Output:

```
Value: 42
Address: 0x7f9a8c405820
Memory freed
```

Basic allocation and deallocation

Program 2: malloc vs calloc

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  int *arr1 = (int*)malloc(5 * sizeof(int));
  int *arr2 = (int*)calloc(5, sizeof(int));
  int i;
  printf("malloc (uninitialized):\n");
  for (i = 0; i < 5; i++) {
    printf("%d ", arr1[i]);
  }
  printf("\ncalloc (zero-initialized):\n");
  for (i = 0; i < 5; i++) {
    printf("%d ", arr2[i]);
  }
  printf("\n");
  free(arr1);
  free(arr2);
  return 0;
}
```

Output:

```
malloc (uninitialized):
0 32767 0 0 1606416992
calloc (zero-initialized):
0 0 0 0 0
```

calloc initializes to zero, malloc doesn't

Program 3: Dynamic Array Allocation

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int n, i;
  int *arr;
  printf("Enter size: ");
  scanf("%d", &n);
  arr = (int*)malloc(n * sizeof(int));
  if (arr == NULL) {
    printf("Allocation failed\n");
    return 1;
  }
  for (i = 0; i < n; i++) {
    arr[i] = i * 10;
  }
  printf("Array: ");
  for (i = 0; i < n; i++) {
    printf("%d ", arr[i]);
  }
  printf("\n");
  free(arr);
  return 0;
}
```

Output:

```
Enter size: 5
Array: 0 10 20 30 40
```

Runtime size determination

Program 4: realloc - Resizing Array

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  int *arr = (int*)malloc(3 * sizeof(int));
  int i;
  arr[0] = 10; arr[1] = 20; arr[2] = 30;
  printf("Original: ");
  for (i = 0; i < 3; i++) printf("%d ", arr[i]);
  arr = (int*)realloc(arr, 5 * sizeof(int));
  if (arr == NULL) {
    printf("Reallocation failed\n");
    return 1;
  }
  arr[3] = 40; arr[4] = 50;
  printf("\nResized: ");
  for (i = 0; i < 5; i++) printf("%d ", arr[i]);
  printf("\n");
  free(arr);
  return 0;
}
```

Output:

```
Original: 10 20 30
Resized: 10 20 30 40 50
```

realloc preserves existing data

Program 5: Dynamic String Allocation

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main() {
  char *str;
  int len = 20;
  str = (char*)malloc(len * sizeof(char));
  if (str == NULL) {
    printf("Allocation failed\n");
    return 1;
  }
  strcpy(str, "Hello World");
  printf("String: %s\n", str);
  printf("Length: %lu\n", strlen(str));
  free(str);
  return 0;
}
```

Output:

```
String: Hello World
Length: 11
```

Dynamic string allocation

Program 6: Dynamic 2D Array (Method 1)

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  int **arr;
  int rows = 3, cols = 4;
  int i, j;
  arr = (int**)malloc(rows * sizeof(int*));
  for (i = 0; i < rows; i++) {
    arr[i] = (int*)malloc(cols * sizeof(int));
  }
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      arr[i][j] = i * cols + j;
    }
  }
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      printf("%2d ", arr[i][j]);
    }
    printf("\n");
  }
  for (i = 0; i < rows; i++) free(arr[i]);
  free(arr);
  return 0;
}
```

Output:

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

Array of pointers approach

Program 7: Dynamic 2D Array (Contiguous Memory)

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  int *arr;
  int rows = 3, cols = 4;
  int i, j;
  arr = (int*)malloc(rows*cols*sizeof(int)); Single contiguous block
  if (arr == NULL) {
    printf("Allocation failed\n");
    return 1;
  }
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      arr[i * cols + j] = i * cols + j;
    }
  }
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      printf("%2d ", arr[i * cols + j]);
    }
    printf("\n");
  }
  free(arr);
  return 0;
}
```

Output:

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

Program 8: Jagged Array (Variable Row Lengths)

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  int **arr;
  int rows = 3;
  int cols[] = {2, 4, 3};
  int i, j;
  arr = (int**)malloc(rows * sizeof(int*));
  for (i = 0; i < rows; i++) {
    arr[i] = (int*)malloc(cols[i]*sizeof(int));
  }
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols[i]; j++) {
      arr[i][j] = i * 10 + j;
    }
  }
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols[i]; j++) {
      printf("%d ", arr[i][j]);
    }
    printf("\n");
  }
  for (i = 0; i < rows; i++) free(arr[i]);
  free(arr);
  return 0;
}
```

Output:

```
0 1
10 11 12 13
20 21 22
```

Each row has different length

Program 9: Dynamic Structure Allocation

```c
#include <stdio.h>
#include <stdlib.h>
struct Point {
  int x;
  int y;
};
int main() {
  struct Point *p;
  p = (struct Point*)malloc(sizeof(struct Point));
  if (p == NULL) {
    printf("Allocation failed\n");
    return 1;
  }
  p->x = 10;
  p->y = 20;
  printf("Point: (%d, %d)\n", p->x, p->y);
  free(p);
  return 0;
}
```

Output:

```
Point: (10, 20)
```

Allocating single structure

Program 10: Dynamic Array of Structures

```c
#include <stdio.h>
#include <stdlib.h>
struct Student {
  int roll;
  int marks;
};
int main() {
  struct Student *arr;
  int n = 3, i;
  arr = (struct Student*)malloc(
    n * sizeof(struct Student));
  arr[0].roll = 1; arr[0].marks = 85;
  arr[1].roll = 2; arr[1].marks = 90;
  arr[2].roll = 3; arr[2].marks = 78;
  for (i = 0; i < n; i++) {
    printf("Roll: %d, Marks: %d\n",
      arr[i].roll, arr[i].marks);
  }
  free(arr);
  return 0;
}
```

Output:

```
Roll: 1, Marks: 85
Roll: 2, Marks: 90
Roll: 3, Marks: 78
```

Array of structures on heap

Program 11: Growing Array with realloc

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  int *arr = NULL;
  int size = 0, capacity = 0;
  int i, val;
  int inputs[] = {10, 20, 30, 40, 50};
  for (i = 0; i < 5; i++) {
    if (size == capacity) {
      capacity = (capacity == 0) ? 1 : capacity * 2;
      arr = (int*)realloc(arr,
        capacity * sizeof(int));
    }
    arr[size++] = inputs[i];
  }
  printf("Array (%d elements): ", size);
  for (i = 0; i < size; i++) {
    printf("%d ", arr[i]);
  }
  printf("\n");
  free(arr);
  return 0;
}
```

Output:

```
Array (5 elements): 10 20 30 40 50
```

Dynamic array that grows automatically

Memory Leaks

- Memory allocated but never freed
- Program keeps consuming memory
- Eventually causes performance issues or crashes
- Common causes:
  - Forgetting to call free()
  - Losing pointer to allocated memory
  - Early return without freeing
  - Exception/error without cleanup
- Prevention: Always pair malloc/calloc with free
- Set pointer to NULL after freeing
- Use valgrind or similar tools to detect leaks

Program 12: Memory Leak Example (WRONG)

```c
#include <stdio.h>
#include <stdlib.h>
void leak() {
  int *ptr = (int*)malloc(100 * sizeof(int));
  printf("Allocated memory\n");
}
int main() {
  int i;
  for (i = 0; i < 5; i++) {
    leak();
  }
  printf("Memory leaked 5 times!\n");
  return 0;
}
```

Output:

```
Allocated memory
Allocated memory
Allocated memory
Allocated memory
Allocated memory
Memory leaked 5 times!
```

Memory not freed in leak() function

Correct version:

```c
void noLeak() {
  int *ptr = (int*)malloc(100 * sizeof(int));
  printf("Allocated memory\n");
  free(ptr);
}
```

Program 13: Lost Pointer (Memory Leak)

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  int *ptr1 = (int*)malloc(10 * sizeof(int));
  printf("Allocated at: %p\n", (void*)ptr1);
  ptr1 = (int*)malloc(20 * sizeof(int));
  printf("New allocation: %p\n", (void*)ptr1);
  printf("First allocation leaked!\n");
  free(ptr1);
  return 0;
}
```

Output:

```
Allocated at: 0x7f9a8c405820
New allocation: 0x7f9a8c405850
First allocation leaked!
```

Lost reference to first allocation

Correct:

```c
int *ptr1 = (int*)malloc(10 * sizeof(int));
free(ptr1);
ptr1 = (int*)malloc(20 * sizeof(int));
free(ptr1);
```

Program 14: Double Free (WRONG)

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  int *ptr = (int*)malloc(sizeof(int));
  *ptr = 42;
  printf("Value: %d\n", *ptr);
  free(ptr);
  printf("Freed once\n");
  free(ptr);
  printf("Freed twice - CRASH!\n");
  return 0;
}
```

Correct:

```c
free(ptr);
ptr = NULL;
if (ptr != NULL) {
  free(ptr);
}
```

Output:

```
Value: 42
Freed once
Segmentation fault (core dumped)
```

Never free same pointer twice!

Program 15: Using Freed Memory (Dangling Pointer)

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  int *ptr = (int*)malloc(sizeof(int));
  *ptr = 42;
  printf("Before free: %d\n", *ptr);
  free(ptr);
  printf("After free: %d (undefined!)\n", *ptr);
  *ptr = 100;
  printf("Modified: %d (dangerous!)\n", *ptr);
  return 0;
}
```

Output:

```
Before free: 42
After free: 42 (undefined!)
Modified: 100 (dangerous!)
```

Undefined behavior - may crash or corrupt

Correct:

```c
free(ptr);
ptr = NULL;
```

Program 16: Dynamic String Array

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main() {
  char **arr;
  int n = 3, i;
  arr = (char**)malloc(n * sizeof(char*));
  arr[0] = (char*)malloc(10 * sizeof(char));
  arr[1] = (char*)malloc(10 * sizeof(char));
  arr[2] = (char*)malloc(10 * sizeof(char));
  strcpy(arr[0], "Apple");
  strcpy(arr[1], "Banana");
  strcpy(arr[2], "Cherry");
  for (i = 0; i < n; i++) {
    printf("%s\n", arr[i]);
  }
  for (i = 0; i < n; i++) free(arr[i]);
  free(arr);
  return 0;
}
```

Output:

```
Apple
Banana
Cherry
```

Array of dynamic strings

Program 17: Flexible Array Member in Structure

```c
#include <stdio.h>
#include <stdlib.h>
struct Array {
  int size;
  int data[];
};
int main() {
  int n = 5, i;
  struct Array *arr;
  arr = (struct Array*)malloc(
    sizeof(struct Array) + n * sizeof(int));
  arr->size = n;
  for (i = 0; i < n; i++) {
    arr->data[i] = i * 10;
  }
  printf("Size: %d\n", arr->size);
  printf("Data: ");
  for (i = 0; i < arr->size; i++) {
    printf("%d ", arr->data[i]);
  }
  printf("\n");
  free(arr);
  return 0;
}
```

Output:

```
Size: 5
Data: 0 10 20 30 40
```

Flexible array member (C99 feature)

Program 18: Matrix Multiplication with Dynamic Memory

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  int **a, **b, **c;
  int r = 2, c1 = 2, c2 = 2;
  int i, j, k;
  a = (int**)malloc(r * sizeof(int*));
  b = (int**)malloc(c1 * sizeof(int*));
  c = (int**)malloc(r * sizeof(int*));
  for (i = 0; i < r; i++) a[i] = (int*)malloc(c1*sizeof(int));
  for (i = 0; i < c1; i++) b[i] = (int*)malloc(c2*sizeof(int));
  for (i = 0; i < r; i++) c[i] = (int*)malloc(c2*sizeof(int));
  a[0][0]=1; a[0][1]=2; a[1][0]=3; a[1][1]=4;
  b[0][0]=5; b[0][1]=6; b[1][0]=7; b[1][1]=8;
  for(i=0;i<r;i++)
    for(j=0;j<c2;j++) {
      c[i][j]=0;
      for(k=0;k<c1;k++) c[i][j]+=a[i][k]*b[k][j];
    }
  printf("Result:\n");
  for(i=0;i<r;i++){for(j=0;j<c2;j++)printf("%d ",c[i][j]);printf("\n");}
  for(i=0;i<r;i++)free(a[i]);free(a);
  for(i=0;i<c1;i++)free(b[i]);free(b);
  for(i=0;i<r;i++)free(c[i]);free(c);
  return 0;
}
```

Output:

```
Result:
19 22
43 50
```

Dynamic 2D arrays for computation

Program 19: Shrinking Array with realloc

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  int *arr;
  int i;
  arr = (int*)malloc(10 * sizeof(int));
  for (i = 0; i < 10; i++) {
    arr[i] = i;
  }
  printf("Original 10 elements: ");
  for (i = 0; i < 10; i++) {
    printf("%d ", arr[i]);
  }
  arr = (int*)realloc(arr, 5 * sizeof(int));
  printf("\nAfter shrinking to 5: ");
  for (i = 0; i < 5; i++) {
    printf("%d ", arr[i]);
  }
  printf("\n");
  free(arr);
  return 0;
}
```

Output:

```
Original 10 elements: 0 1 2 3 4 5 6 7 8 9
After shrinking to 5: 0 1 2 3 4
```

realloc can reduce size too

Program 20: Linked List with Dynamic Allocation

```c
#include <stdio.h>
#include <stdlib.h>
struct Node {
  int data;
  struct Node *next;
};
int main() {
  struct Node *head, *temp;
  int i;
  head = NULL;
  for (i = 1; i <= 5; i++) {
    temp = (struct Node*)malloc(sizeof(struct Node));
    temp->data = i * 10;
    temp->next = head;
    head = temp;
  }
  temp = head;
  printf("List: ");
  while (temp != NULL) {
    printf("%d -> ", temp->data);
    temp = temp->next;
  }
  printf("NULL\n");
  while (head != NULL) {
    temp = head;
    head = head->next;
    free(temp);
  }
  return 0;
}
```

Output:

```
List: 50 -> 40 -> 30 -> 20 -> 10 -> NULL
```

Dynamic nodes, proper cleanup

Best Practices

- Always check if malloc/calloc/realloc returns NULL
- Always free allocated memory when done
- Set pointer to NULL after freeing
- Don't access memory after freeing (dangling pointer)
- Don't free the same pointer twice
- Match every malloc/calloc with exactly one free
- For 2D arrays, free in reverse order of allocation
- Use valgrind or similar tools to detect memory issues
- Prefer calloc when you need zero-initialized memory

Key Takeaways

- Dynamic memory allocated from heap at runtime
- malloc: uninitialized, calloc: zero-initialized
- realloc: resize existing allocation
- free: deallocate memory
- Memory leaks occur when memory not freed
- Dangling pointers refer to freed memory
- Double free causes undefined behavior
- Essential for flexible data structures
- Enables runtime size determination
- Requires careful management to avoid bugs