Chapter 11

Dynamic Memory Management

Dr. Niroj Kumar Pani

nirojpani@gmail.com

Department of Computer Science Engineering & Applications

Indira Gandhi Institute of Technology

Sarang, Odisha

Chapter Outline…

- Dynamic Memory Management: Why & What?
- Dynamic Memory Management: The Process
- Functions in C for Dynamic Memory Management
- Programming Examples
- Assignments

Dynamic Memory Management: Why & What?

- **Why Dynamic Memory Management?:**
  - **So far, in all our programs, the variables are allocated with memory space at compilation time.**
  - **However, in some programming situations** (particularly when dynamic storage is required with structures like arrays, linked-lists, stacks, queues etc.) **this way of memory allocation may not be efficient as it may cause wastage of memory space or failure of the program (due to memory overflow).**

    For example, let us consider a string "name" **that intends to store the name of a person in a program. When we declare the string as:**

    ```
    char name[10];
    ```

    it is given 11 bytes **of space at compilation time** (and this is fixed throughout the program). **In such case, if the name entered is less than** 10 characters **then there will be a wastage of memory space and if the name entered is more than** 10 characters **then there will be a runtime error.**

- C language has a technique called "dynamic memory management" to deal with such situation.
- By using it memory space could be allocated to variables at runtime and hence they will be allocated the exact amount of memory space they need. Therefore, there shall be neither wastage of memory space nor any chance of runtime error due to memory overflow.

- **What is Dynamic Memory Management? [Definition]:**

  It refers to the technique that allows allocation of memory space to variables at runtime, and release of the memory space at runtime, thus optimizing the use of storage space.

Dynamic Memory Management: The Process

- As discussed in "Chapter 9 - Storage Classes of Variables", the "heap segment" of the computer's memory is used for dynamic memory allocation.
- During program execution (i.e., at runtime), when we need some memory space (to be allocated to some variable), a program called the "memory manager" is invoked (through some pre-defined functions) to allocate memory. The memory manager searches the heap segment for the requested amount of space. There are three methods of searching:
  - First-fit: Allocate the first space that is big enough.
  - Best-fit: Allocate the smallest space that is big enough.
  - Worst-fit: Allocate the largest space that is big enough.

  If the required space is available, it is given, otherwise the memory manager indicates "memory overflow" by returning a NULL.

- **Similarly, when we want to release unwanted memory during program execution** (i.e., at runtime)**, another program called the "garbage collector" is invoked** (through some pre-defined functions)**. It returns the unused memory space to he heap segment.**

Functions in C for Dynamic Memory Management

- C language provides four pre-defined functions for dynamic memory management that can be used to allocate, release and reallocate memory during program execution:
    1. malloc()
    2. calloc()
    3. realloc()
    4. free()

  [NOTE]: To use these functions we must include <stdlib.h> in our program.

- **malloc(): It allocates the requested size of bytes and returns a pointer of type** void **to the first byte of the allocated space. If it fails to allocate, it returns** NULL.

Syntax:

```
ptr = (cast_type*)malloc(byte_size);
/* "ptr" is a pointer of type "cast_type". */
```

e.g.,

```
int *y;
y = (int*)malloc(50*sizeof(int));
```

- **calloc(): It allocates the requested size of bytes, initializes them to zero, and then returns a pointer of type** void **to the first byte of the allocated space. If it fails to allocate, it returns** NULL.

Syntax:

```
ptr = (cast_type*)calloc(n, element_size);
/* "ptr" is a pointer of type "cast_type". */
```

e.g.,

```
int *y;
y = (int*)calloc(50, sizeof(int));
```

- **realloc(): It modifies the size of previously allocated memory space.**

Syntax:

```
ptr = (cast_type*)realloc(ptr, new_size);
/* "ptr" is a pointers of type "cast_type". */
/* "ptr" points to a previously allocated memory space. */
```

e.g.,

```
int *y;
y = (int*)malloc(100*sizeof(int));
y = (int*)realloc(y, 150*sizeof(int));
```

- **free(): It frees (releases) previously allocated memory space.**

Syntax:

```
free(ptr);
/* "ptr" points to a previously allocated memory space. */
```

e.g.,

```
int *y;
y = (int*)calloc(50, sizeof(int));
free(y);
```

Programming Examples

- **Programming Example 1:**

```
/* PR11_1.c: Program that stores 'n' integers (the value of 'n' is specified at runtime) in an array and
displays them in reverse order . */

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

void main()
{
    int size, *numArray, *i;
    printf("\nHow many integers do you want to enter?: ");
    scanf("%d", &size);

    /* Allocating Memory */
    numArray = (int *)malloc(size * sizeof(int));
    if(numArray == NULL)
    {
        printf("No space available.");
        exit(1);
    }
    printf("Space allocated for %d integers.", size);
```

[Cont.]

```
    /* Reading the integers */
    printf("\n\nEnter %d integers: ",size);
    for(i=numArray; i<numArray+size; i++)  /* An array name corresponds to
                                              its base address. */
    {
        scanf("%d", i);
    }

    /* Printing the integers in reverse order */
    printf("The numbers in the reverse order are: ");
    for(i=numArray+size-1; i>=numArray; i--)
    {
        printf("%d ", *i);
    }

    getch();
}
```

Output

```
How many integers do you want to enter?: 5
Space allocated for %d integers.

Enter 5 integers: 1 3 5 7 9
The numbers in the reverse order are: 9 7 5 3 1
```

Programming Example 2:

```
/* PR11_2.c: Write a program to store a word in a string whose size is determined by the user at
runtime and then modify the same to store a larger word. */

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

void main()
{
   int size;
    char *word;
    printf("\nEnter the size of the word: ");
    scanf("%d", &size);

    /* Allocating Memory */
    word = (char *)calloc(size, sizeof(char));
    if(word == NULL)
    {
        printf("No space available.");
        exit(1);
    }
    printf("Space allocated for a %d lettered word.", size);
```

[Cont.]

```
/* Reading the word */
printf("\n\nEnter the %d lettered word: ", size);
scanf("%s", word);

/* Printing the word */
printf("You have just entered: %s\n", word);

/* Reallocating memory for another word*/
printf("\nReallocating memory...\n");
printf("\nEnter the size of another word: ");
scanf("%d", &size);
word = (char *)realloc(word, size*sizeof(char));
if(word == NULL)
{
    printf("No space available.");
    exit(1);
}
printf("Space allocated for a %d lettered word.", size);

/* Reading the word */
printf("\n\nEnter the %d lettered word: ", size);
scanf("%s", word);
```

[Cont.]

```
    /* Printing the word */
     printf("You have just entered: %s\n", word);

     /* Freeing memory */
     free(word);

     getch();
}
```

Output

```
Enter the size of the word: 5
Space allocated for a 5 lettered word.

Enter the 5 lettered word: India
You have just entered: India

Reallocating memory...

Enter the size of another word: 7
Space allocated for a 7 lettered word.

Enter the 7 lettered word: America
You have just entered: America
```

Assignments

Complete the experiments given in "Lab Manual - Section 13"**.**

End of Chapter 11