"Advanced C Programming"

DMA

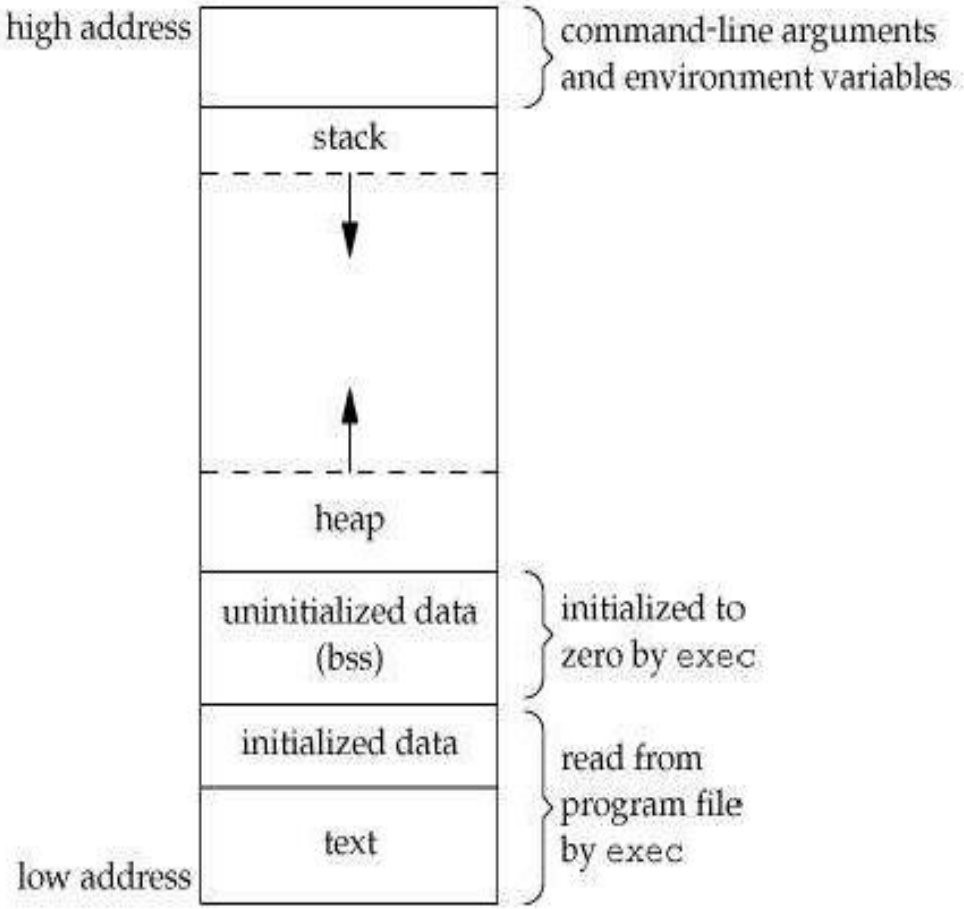
Need of Dynamic memory allocation

- The exact size of array is **unknown**
- The size of array you have declared initially can be sometimes **insufficient** and sometimes **more than required**
- Definition: Dynamic Memory allocation is a technique in which the memory will be allocated to the variables **during the execution of the program** by explicit request of the programmer.

DMA functions

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer to the first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	deallocate the previously allocated space
realloc()	Change the size of previously allocated space

Memory Management



- Code/Text Segment
- Data Segment
- Stack Segment
- Heap Segment

Memory Management…

Segment	Description
Code Segment	Contains the **executable code**. All the executable instructions, functions are kept here. This region is **read-only**.
Data Segment	a) Initialized Data Segment – Stores the variables which are initialized (Read-only → Constants, Read-Write→ Global and Static Variables). b) Block Started by Symbol (BSS) Segment – Global and Static variables which are uninitialized
Stack Segment	(a) Command Line arguments (b) Call Stack - Local variables of a function, return value & arguments to the function
Heap Segment	Any memory allocated at **run-time** (Using malloc, calloc, realloc etc,.)

DMA functions: malloc()

The function malloc() reserves a block of memory of specified size and **return a pointer of type void (void *)** which can be casted into pointer of any form.

Syntax:

#include <stdlib.h> → **Header File**

void *malloc(size_t size); → **Prototype**

ptr = (data-type*) malloc(size) → Function Call

- Here, *ptr* is pointer of "data-type".
- The malloc() function returns a pointer to an area of memory with size.
- If the space is insufficient, allocation fails and returns NULL pointer.

DMA functions: malloc()

Example:

```
ptr = (int*)malloc(100*sizeof(int));
```

This statement will allocate 400 bytes and the pointer points to the address of first byte of memory.

Note: sizeof(int) is 4bytes

DMA functions: calloc()

calloc is used to allocate “n contiguous blocks” of memory, the allocated region is **initialized to zeroes**.

In contrast, **malloc** does not touch the contents of the allocated block of memory, which means it contains **garbage values**.

Syntax:

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

Here, *ptr* is pointer of cast-type. The calloc() function returns a pointer to an area of memory with n elements-size.

DMA functions: calloc()

Example:

ptr=(int*)calloc(100, sizeof(float));

This statement will allocate 100 * 4 = 400 bytes and the pointer points to the address of first byte of memory.

Note: sizeof(float) is 4bytes

DMA functions: calloc() and malloc()

calloc()	malloc()
Initialize with zeros by default.	Uninitialized variables will have garbage values
2 Arguments	1 Argument
Allocates memory as a single contiguous block	Allocates memory as a single contiguous block
Need to free the memory explicitly	need to free the memory
If there is no enough space then returns NULL	If there is no enough space then returns NULL

DMA functions: free()

The **free**() function frees the memory space pointed to by *ptr*, which must have been returned by a previous call to **malloc**(), **calloc**() or **realloc**()

Syntax:

void free(void *ptr);

The **free**() function **returns no value.**

Example

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc()/calloc() function.

```
#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)); //memory allocated using
malloc  if(ptr==NULL) {
        printf("Error! memory not allocated."); exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i) {
        scanf("%d",ptr+i);
    }
    for(i=0;i<n;++i) {
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr); return 0; }
```

DMA functions: realloc()

- The **realloc**() function changes the size of the memory block pointed to by *ptr* to *size* bytes.
- **The contents will be unchanged** in the range from the start of the region up to the minimum of the old and new sizes.
- If the new size is larger than the old size, the added memory will *not* be initialized.

Syntax:

ptr = (data-type *) realloc(ptr, new-size);

Example program on realloc()

```
#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: \n");
    for(i=0;i<n1;++i)
        printf("%u\t",ptr+i);
    printf("\nEnter elements of array: \n");
    for(i=0;i<n1;++i)
    {
       scanf("%d",ptr+i);
    }
```

```
for(i=0;i<n1;++i)
    {
       printf("%d \t",*(ptr+i));
    }
printf("\nEnter new size of array: \n");
    scanf("%d",&n2);
    ptr=(int*)realloc(ptr, n2);
    for(i=0;i<n2;++i)
         printf("%u\t",ptr+i);
    for(i=0;i<n2;++i)
    {
       printf("%d \t",*(ptr+i));
    }
      free(ptr);
    return 0;

}
```

DMA for structures

```
#include <stdio.h>
#include<stdlib.h>
struct name
{
  int id;
  char name[30];
};
int main ()
{
  struct name *ptr;
  int i, n;
  ptr = (struct name *) malloc (sizeof (struct name));
  printf ("Enter name and id respectively:\n");
  scanf ("%s %d", ptr->name, &ptr->id);

        printf ("Displaying Infromation:\n");
      printf ("%s\t%d\t\n", ptr->name, ptr->id);
}
```

DMA: Example with structures

```
#include <stdio.h>
#include<stdlib.h>
struct name {
   int a;
   float b;
   char c[30];
};
int main(){
   struct name *ptr;
   int i,n;
   printf("Enter n: ");
   scanf("%d",&n);
   ptr=(struct name*)malloc(n*sizeof(struct
name));
/* Above statement allocates the memory
for n structures with pointer ptr pointing to
base address */
```

```
for(i=0;i<n;++i){
      printf("Enter string, integer
a n d   f l o a t i n g   n u m b e r
respectively:\n");
   scanf("%s %d %f",&(ptr+i)->c,
&(ptr+i)->a,&(ptr+i)->b);

}
 printf("Displaying Info:\n");
   for(i=0;i<n;++i)

printf("%s\t%d\t%.2f\n",(ptr+i)->c
,(ptr+i)->a,(ptr+i)->b);
   return 0;
}
```

Command line arguments

Command line arguments

int main(int argc, char *argv[])

argc : argument count
argv: argument values in string form

Needful conversion operations has to be performed before using doing operations.

Ex: atoi – string int

Command line arguments – Printing

```
#include <stdio.h>

int main( int argc, char *argv[] )  {
int i;
   printf("Program name s %s\n", argv[0]);
   printf("argument count is %d\n", argc);

     for(i=1;i<argc;i++)
        printf("argument %d  supplied is %s\n",i, argv[i]);
}
```

Command line arguments – sum operation

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[])
{
  int i, sum;
  printf ("Program name s %s\n", argv[0]);
  printf ("argument count is %d\n", argc);

  for (i = 1; i < argc; i++)
    printf ("argument %d  supplied is %s\n", i, argv[i]);

  for (i = 1; i < argc; i++)
    sum += atoi (argv[i]);

  printf ("sum of given arguments is %d\n", sum);
}
```