CSE101-Lec#28-29

- Dynamic memory management

Outline

- Dynamic Memory management
  - malloc()
  - calloc()
  - realloc()
  - free()

Dynamic Memory Allocation

- The statement:

  ```
  int marks[100];
  ```

  allocates block of memory to 100 elements of type int and memory is also contiguous. If one `int` requires 4 bytes of memory , a total of 400 bytes are allocated.

Why this approach of declaring array is not useful?

- This may lead to wastage of memory if all allocated memory is not utilized.

- Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.
- There are 4 library functions under "**stdlib.h**" for dynamic memory allocation.

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer 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

malloc()

- The name malloc stands for "memory allocation".
- The `malloc()` function allocates a block of memory of specified size from the memory heap.
- Syntax:

```
void * malloc(size);
```

- Here size is the number of bytes of storage to be allocated.
- If memory is allocated successfully , it returns a **pointer to first location** of newly allocated block of memory.
- If memory is not allocated i.e. no enough space exists for new block or some other reason, returns **NULL**.

malloc()

- Return type of `malloc()` is void pointer , it has to be **cast** to the type of data being dealt with.
- memory allocated by `malloc()` by default contain the garbage values.
- Example:

```
int *p;
p=(int*)malloc(n*sizeof(int));
```

- In the above example, p is **pointer** of type integer
- `int*` tells to what type it will be pointing. `int` tells that the `malloc()` function is type casted to return the address of integer variable.
- `n` is the number of elements

Program to allocate memory to integers.

```
#include <stdio.h>
#include <stdlib.h> /*required for dynamic
   memory*/
int main()
{
 int number, *ptr, i;
 printf("How many ints would you like store?");
 scanf("%d", &number);
 ptr = (int *)malloc(number*sizeof(int));
   /*allocate memory*/
 for(i=0 ; i<number ; i++) {
  *(ptr+i) = i;
 }
 for(i=0 ; i<number ; i++){
   printf("%d\n", *(ptr + i));
 }
 return 0;
}
```

```
How many ints would you like store? 3
0
1
2
```

calloc()

- The name calloc stands for "contiguous allocation".
- It provides access to memory, which is available for dynamic allocation of variable-sized blocks of memory.
- Syntax:

```
void *calloc(size_t nitems, size_t size);
```

- calloc is similar to malloc, but the main **difference** is that the values stored in the allocated memory space is **zero** by default. With malloc, the allocated memory could have any garbage value.
- `calloc()` requires **two arguments**.

1. The **first** is the number of variables you'd like to allocate memory for.
2. The **second** is the size of each variable.

calloc()

- If memory is allocated successfully, function `calloc()` returns a pointer to the first location of newly allocated block of memory otherwise returns NULL
- Memory allocated by `calloc()` by default contains the zero values.
- E.g. If we want to allocate memory for storing n integer numbers in contiguous memory locations

```
int *p;
p=(int*)calloc(n, sizeof(int));
```

Program to show calloc() function

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
 float *x;
 int i,n;
 printf("how many elements do u want?");
 scanf("%d",&n);
 x=(float*)calloc(n,sizeof(float));
 if(x!=NULL)
 {
   printf("data is=\n");
   for(i=0;i<n;i++)
     printf("\n x[%d]=%d ",i,*(x+i));
 }
 else
   printf("calloc failed");
 getch();
}
```

```
how many elements do u want 3
0
0
0
```

Difference between malloc() and calloc()

	calloc()	malloc()
Function:	Allocates a region of memory large enough to hold "n elements" of "size" bytes each.	Allocates "size" bytes of memory.
Syntax:	`void *calloc (number_of_blocks, size_in_bytes);`	`void *malloc (size_in_bytes);`
No. of arguments:	2	1
Contents of allocated memory:	The allocated region is initialized to zero.	The contents of allocated memory are not changed. i.e., the memory contains garbage values.
Return value:	`void pointer (void *).` If the allocation succeeds, a pointer to the block of memory is returned.	`void pointer (void *).` If the allocation succeeds, a pointer to the block of memory is returned.

realloc()

- Now suppose you've allocated a certain number of bytes for an array but later find that you want to add values to it. You could copy everything into a larger array, which is inefficient, or you can allocate more bytes using `realloc()`, without losing your data.
- `realloc()` takes **two** arguments.
  1. The **first** is the pointer referencing the memory.
  2. The **second** is the total number of bytes you want to reallocate.
- Passing zero as the second argument is the equivalent of calling free.
- Syntax:

```
void *realloc(pointerToObject, newsize);
```

`realloc()`

- If memory is allocated successfully, function `realloc()` returns a pointer to the first location of newly allocated block of memory which may be at same site or at new site and copy the contents from previous location to a new location if required , otherwise returns NULL.

```
#include<stdio.h>
#include <stdlib.h>
int main()
{
 int *ptr, i;
 ptr = (int *)calloc(5, sizeof(int));
 *ptr = 1;
 *(ptr+1) = 2;
 ptr[2] = 4;
 ptr[3] = 8;
 ptr[4] = 16;
 ptr = (int *)realloc(ptr, 7*sizeof(int));
 printf("Now allocating more memory... \n");
 ptr[5] = 32; /* now it's legal! */
 ptr[6] = 64;
 for(i=0 ; i<7 ; i++){
    printf("ptr[%d] holds %d\n", i, ptr[i]);
 }
}
```

This example uses calloc to allocate memory then realloc

```
Now allocating more memory...
   ptr[0] holds 1
   ptr[1] holds 2
   ptr[2] holds 4
   ptr[3] holds 8
   ptr[4] holds 16
   ptr[5] holds 32
   ptr[6] holds 64
```

free()

- Deallocates a memory block allocated by previous call to `malloc(), calloc()` `or realloc()` and return it to memory to be used for other purposes.
- Syntax:

```
void *free(void *block);
```

- The argument of function `free()` is the pointer to block of memory which is to be freed.

free()

- The `realloc()` function can behave the same as `free()` function provided the second argument passed to `realloc()` is 0.

```
free(ptr);
```

which is equivalent to

```
realloc(ptr,0);
```

Memory Leak

- A condition caused by a program that does not free up the extra memory it allocates.
- It occurs when the dynamically allocated memory is no longer needed but it is not freed.
- If we continuously keep on allocating the memory without freeing it for reuse, the entire heap storage will be exhausted.
- In such circumstances, the memory allocation functions will start failing and program will start behaving unexpectedly

Next Class: Derived Types

cse101@lpu.co.in