Dynamic Memory Allocation

Dynamic memory allocation is the allocation of memory at run time i.e., when program executes then required memory is asked from the user and memory is allocated.

In many applications it is not possible to predict how much memory would be needed by the program at run time. For example if we declare an array of integers-

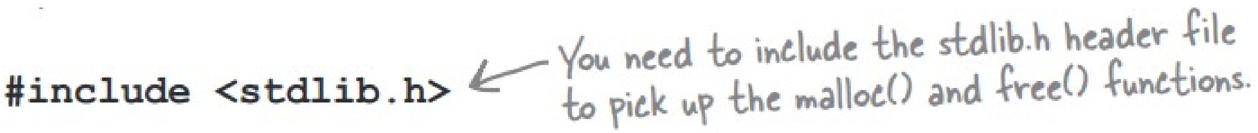
```
int emp_no[200]; 50
```

In an array, it is must to specify the size of array while declaring, so the size of this array will be fixed during runtime. Now two types of problems may occur. The first case is that the number of values to be stored is less than the size of array -and hence there is wastage of memory. For example if we have to store only 50 values in the above array, then space for 150 values(300 bytes) is wasted. In second case our program fails if we want to store more values than the size of array, for example If there is need to store 205 values in the above array.

To overcome these problems we should be able to allocate memory at run time. The process of allocating memory at the time of execution is called dynamic memory allocation.

The allocation and release of this memory space can be done with the help of some built-in-functions whose prototypes are found in alloc.h and stdlib.h header files. These functions take memory from a memory area called heap and release this memory whenever not required, so that it can be used again for some other purpose.

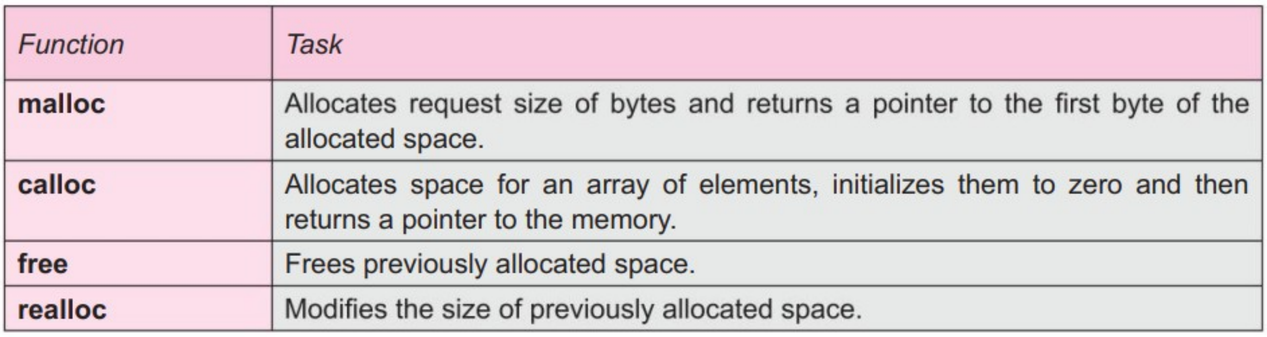
Pointers play an important role in dynamic memory allocation because we can access the dynamically allocated memory only through pointers.



```c
#include <stdlib.h>
```

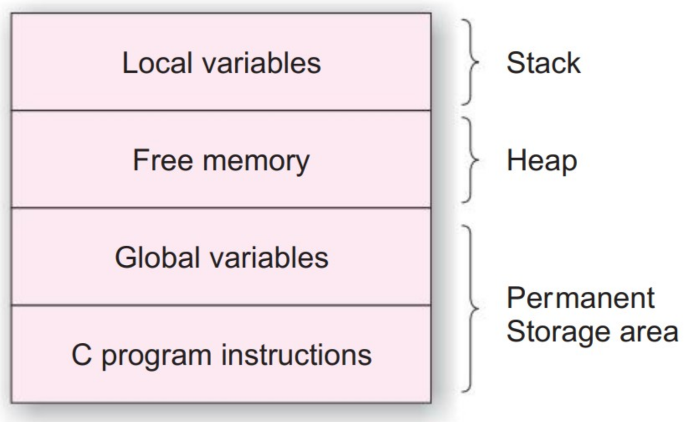
You need to include the stdlib.h header file to pick up the malloc() and free() functions.

The function that asks for memory is called malloc() for memory allocation. malloc() takes a single parameter: the number of bytes that you need. Most of the time, you probably don’t know exactly how much memory you need in bytes, so malloc() is almost always used with an operator called sizeof, like this:



Function	*Task*
malloc	Allocates request size of bytes and returns a pointer to the first byte of the allocated space.
calloc	Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
free	Frees previously allocated space.
realloc	Modifies the size of previously allocated space.

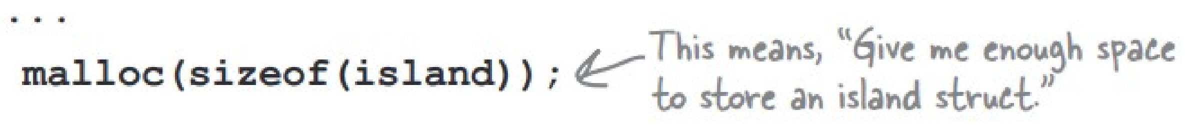
Memory Allocation Process



The program instructions and global and static variables are stored in a region known as permanent storage area and the local variables are stored in another area called stack. The memory space that is located between these two regions is available for dynamic allocation during execution of the program. This free memory region is called the heap. The size of the heap keeps changing when program is executed due to creation and death of variables that are local to functions and blocks.

malloc()

Declaration: void *ma11oc(size_t size);



```
...
malloc(sizeof(island));
```

This means, “Give me enough space to store an island struct.”

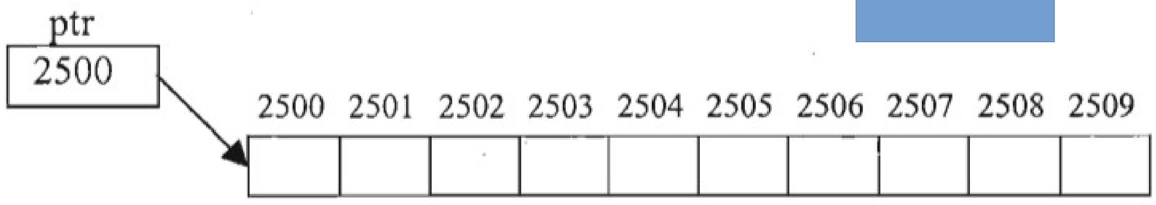
This function is used to allocate memory dynamically. The argument size specifies the number of bytes to be allocated. The type size _t is defined in stdlib.h as unsigned int. On success: malloc( ) returns a pointer to the first byte of allocated memory. The returned pointer is of type void, which can be typecast to appropriate type of pointer. It is generally used as-

```
ptr = (datatype *) malloc( specified size)
```

Here ptr is a pointer of type datatype and specified size is the size in bytes required to be reserved in memory. The expression (datatype *) is used to typecast the pointer returned by malloc( ).

For example-

```
int *ptr;
ptr=(int*) malloc(10);
```



This allocates 10 contiguous bytes of memory space and the address of first byte is stored in the pointer variable ptr. This space can hold 5 integers. The allocated memory contains garbage value. We can use sizeof operator to make the program portable and more readable

```
ptr = ( int * ) malloc ( 5 * sizeof ( int ) );
```

This allocates the memory space to hold five integer values.

If there is not sufficient memory available in heap then malloc( ) returns NULL. So we should always check the value returned by malloc( ).

```
                ptr = (float *) malloc(10*sizeof(float) );
        if ( ptr = = NULL )
                printf("Sufficient memory not available");
```

Unlike memory allocated for variables and arrays, dynamically allocated memory has no name associated with it. So it can be accessed only through pointers. We have a pointer which points to the first byte of the allocated memory and we can access the subsequent bytes using pointer arithmetic.

```
#include<stdio.h>
#include<alloc.h>
int main()
{
        int *p, n, i;
printf("Enter the number of integers");
scanf("%d", &n);
p=(int )malloc(n*sizeof(int));
if(p==NULL)
{
        printf("Memory not available ");
        exit(1);
}
for(i=0;i<n;i++)
{
        printf("Enter an integer: ");
        scanf("%d", p+i);
}
for(i=0;i<n;i++)
        printf("%d\t",*(p+i));
}
return 0;
```

```
#include <stdio.h>
 #include <stdlib.h>
 #define NULL 0
 main()
 {
 int *p, *table;
 int size;
 printf(“\nWhat is the size of table?”);
scanf(“%d”,size);
 printf(“\n”)

 /*------------Memory allocation --------------*/
 if((table = (int*)malloc(size *sizeof(int))) == NULL)
 {
 printf(“No space available \n”);
 exit(1);
 }
 printf(“\n Address of the first byte is %u\n”, table);

 /* Reading table values*/
 printf(“\nInput table values\n”);
 for (p=table; p<table + size; p++)
 scanf(“%d”,p);

 /* Printing table values in reverse order*/
 for (p = table + size –1; p >= table; p – –)
 printf(“%d is stored at address %u \n”,*p,p);
}
```

Output

```
 What is the size of the table? 5
 Address of the first byte is 2262
 Input table values
 11 12 13 14 15
 15 is stored at address 2270
 14 is stored at address 2268
 13 is stored at address 2266
 12 is stored at address 2264
 11 is stored at address 2262
```

calloc()

Declaration: void *calloc(size_t n, size_t size);

The calloc( ) function is used to allocate multiple blocks of memory. It is somewhat similar to malloc( ) function except for two differences.

The first one is that it takes two arguments. The first argument specifies the number of blocks and the second one specifies the size of each block. For example,

```
ptr = ( int * ) calloc ( 5 , sizeof(int) );
```

This allocates 5 blocks of memory, each block contains 2 bytes and the starting address is stored in the pointer variable ptr, which is of type int. An equivalent malloc( ) call would be -

```
ptr = ( int * ) malloc ( 5 * sizeof(int) );
```

Here we have to do the calculation ourselves by multiplying, but calloc( ) function does the calculation for us.

The other difference between calloc( ) and malloc( ) is that the memory allocated by malloc() contains garbage value while the memory allocated by calloc( ) is initialized to zero. But this initialization by calloc( ) is not very reliable, so it is better to explicitly initialize the elements whenever there is need to do so.
Like malloc( ), calloc() also returns NULL if there is not sufficient memory available in the heap.

realloc()

Declaration: void *realloc(void *ptr, size_t newsize);

We may want to increase or decrease the memory .allocated by malloc( ) or calloc( ). The function realloc( ) is used to change the size of the memory block. It alters the size of the memory block without losing the old data. This is known as reallocation of memory.

This function takes two arguments, first is a pointer to the block of memory that was previously allocated by malloc( ) or calloc( ) and second one is the new size for that block. For example -

```
ptr = (int *) malloc ( size );
```

This statement allocates the memory of the specified size and then starting address of this memory block is stored in the pointer variable ptr. If we :want to change the size of this memory block, then we can use realloc( ) as -

```
ptr = (int *) realloc ( ptr , newsize );
```

This statement allocates the memory space of newsize bytes, and the starting address of this memory block is stored in the pointer variable ptr. The newsize may be smaller or larger than the old size.
If the newsize is larger, then the old data is not lost and the newly allocated bytes are uninitialized. The starting address contained in ptr may change if there is not sufficient memory at the old address to store all the bytes consecutively. This function moves the contents of old block into the new block and the data of the old block is not lost. On failure, realloc( ) returns NULL.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
char *input;
int len;
input = (char *)malloc(sizeof(char)*1024);
```

```c
if(input==NULL)
{
puts("Unable to allocate buffer! Oh no!");
exit(1);
}
puts("Type something long and boring:");
fgets(input,1023,stdin);

len = strlen(input);
if(realloc(input,sizeof(char)*(len+1))==NULL)
{
puts("Unable to reallocate buffer!");
exit(1);
}
puts("Memory reallocated.");
puts("You wrote:");
printf("\"%s\"\n",input);
return(0);
}
```

free()

Declaration: void free(void *p)

The dynamically allocated memory is not automatically released; it will exist till the end of program. If we have finished working with the memory allocated dynamically, it is our responsibility to release that memory so that it can be reused. The function free() is used to release the memory space allocated dynamically. The memory released by free() is made available to the heap again and can be used for some other purpose. For example -

```
free(ptr);
```

Here ptr is a pointer variable that contains the base address of a memory block created by malloc( ) or calloc( ). Once a memory location is freed it should not be used. We should -not try to free any memory location that· was not allocated by malloc( ), calloc( ) or realloc( ).

When the program tenninates all the memory is released automatically by the operating system but it is a good practice to free whatever has been allocated dynamically. We won't get any errors if we don't free the dynamically allocated memory, but this would lead to memory leak i.e. memory is slowly leaking away and can be reused only after the termination of program.

For example consider this function-

```
func ( )
{
        int *ptr;
        ptr=(int*) malloc(10*sizeof(int)); 20 bytes
        .............
}
```

Here we have allocated memory for 10 integers through malloc(), so each time this function is called, space for 10 integers would be reserved. We know that the local variables vanish when the function terminates, and since ptr is a local pointer variable so it will be deallocated automatically at the termination of function. But the space allocated dynamically is not deallocated automatically, so that space remains there and can't be used, leading to memory leaks. We should free the memory space by putting a call to free( ) at the end of the function.

Since the memory space allocated dynamically is not released after the ternination of function, so it is valid to return a pointer to dynamically allocated memory. For example -

```
int *func()
{
        int *ptr;
        ptr=(int*) malloc(10*sizeof(int));
        ............
        return ptr;
}
```

Here we have allocated memory through malloc() in func(), and returned a pointer to this memory. Now the calling function receives the starting address of this memory, so it can use this memory. Note that now the call to function free() should be placed in the calling function when it has finished working with this memory. Here func() is declared as a function returning pointer. Recall that it is not valid to return address of a local variable since it vanishes after the termination of function.

```
#include <stdlib.h>
int main()
{
int *age;
age = (int )malloc(sizeof(int)1);
if(age==NULL)
{
puts("Out of Memory or something!");
exit(1);
}
printf("How old are you in years? ");
scanf("%d",age);
age = 365;
printf("You're over %d days old!\n",*age);
free(age);
return(0);
}
```