File Credits: https://profile.iiita.ac.in/bibhas.ghoshal/ITP_2019/lecture_slides/Dynamic_Memory_Allocation.pdf

Dynamic Memory Allocation

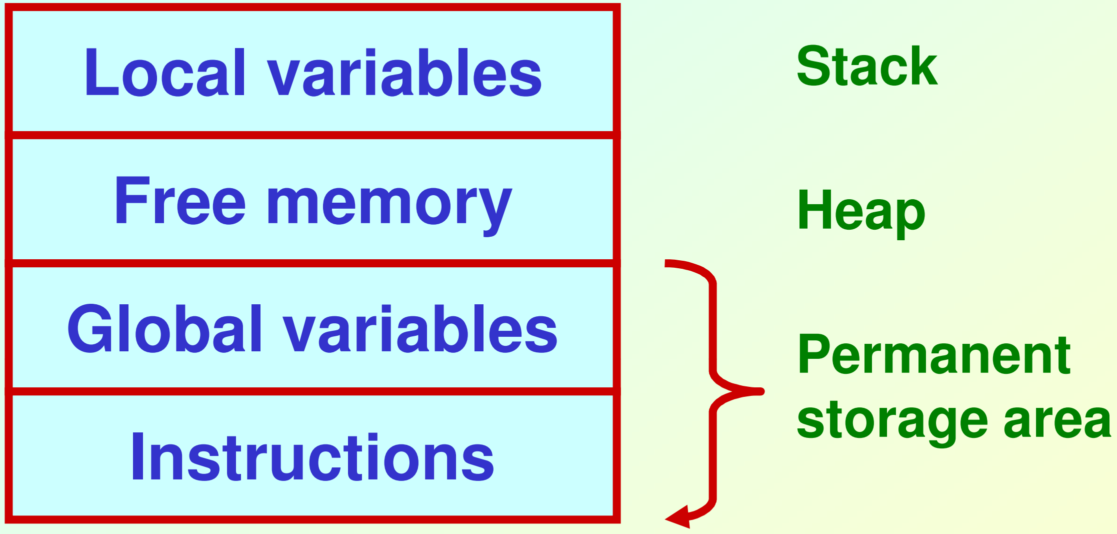
Basic Idea

- Many a time we face situations where data are dynamic in nature.
  - Amount of data cannot be predicted beforehand.
  - Number of data items keeps changing during program execution.
- Such situations can be handled more easily and effectively using dynamic memory management techniques.

Contd.

- C language requires the number of elements in an array to be specified at compile time.
  - Often leads to wastage or memory space or program failure.
  - **C-99 allows this, however**
- Dynamic Memory Allocation
  - Memory space required can be specified at the time of execution.
  - C supports allocating and freeing memory dynamically using library routines.

Memory Allocation Process in C



Contd.

- The program instructions and the global variables are stored in a region known as ***permanent storage area***.
- The local variables are stored in another area called ***stack***.
- The memory space between these two areas is available for dynamic allocation during execution of the program.
  - This free region is called the ***heap***.
  - The size of the heap keeps changing.

Memory Allocation Functions

- `malloc`
  - Allocates requested number 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.

Allocating a Block of Memory

- A block of memory can be allocated using the function `malloc`.
  - Reserves a block of memory of specified size and returns a pointer of type `void`.
  - The return pointer can be type-casted to any pointer type.
- General format:

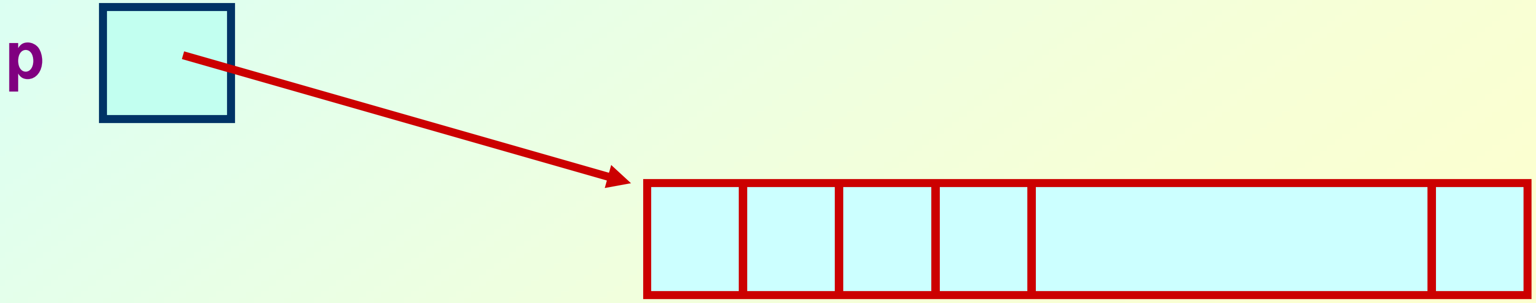
```
ptr =  (type *) malloc (byte_size);
```

Contd.

- **Examples**

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

  - A memory space equivalent to ***100 times the size of an int*** bytes is reserved.
  - The address of the first byte of the allocated memory is assigned to the pointer `p` of type `int`.



400 bytes of space

Contd.

```
cptr = (char *) malloc (20);
```

- Allocates 20 bytes of space for the pointer `cptr` of type `char`.

```
sptr = (struct stud *) malloc
        (10 * sizeof (struct stud));
```

- Allocates space for a structure array of 10 elements. `sptr` points to a structure element of type “`struct stud`”.

Points to Note

- `malloc` always allocates a block of contiguous bytes.
  - The allocation can fail if sufficient contiguous memory space is not available.
  - If it fails, `malloc` returns `NULL`.

```
if  ((p = (int *) malloc(100 * sizeof(int))) == NULL)
  {
      printf (”\n Memory cannot be allocated”);
      exit();
  }
```

Releasing the Used Space

- When we no longer need the data stored in a block of memory, we may release the block for future use.
- How?
  - By using the `free` function.
- General syntax:

```
        free (ptr);
```

where `ptr` is a pointer to a memory block which has been previously created using `malloc`.

Example 1: using 1-D array

```
#include <stdio.h>

main()
{
  int i,N;
  float *height;
  float sum=0,avg;

  printf("Input no. of students\n");
  scanf("%d", &N);

  height = (float *)
       malloc(N * sizeof(float));
```

```
printf("Input heights for %d
students \n",N);
  for (i=0; i<N; i++)
   scanf ("%f", &height[i]);

  for(i=0;i<N;i++)
    sum += height[i];

  avg = sum / (float) N;

  printf("Average height = %f \n",
               avg);
  free (height);
}
```

Example 2: Bubble sort on array of structures

```
#include <stdio.h>
typedef struct
{
    int  roll;
    char  dept_code[25];
    float  cgpa;
} stud;

main()
{
  stud  *class, t;
  int  j, k, n;

  scanf  ("%d", &n);
        /* no. of students */
height = (stud *)
    malloc(n * sizeof(stud));
```

```
for (k=0; k<n; k++)
  scanf ("%d %s %f", &class[k].roll,
               class[k].dept_code,
               &class[k].cgpa);
for (j=0; j<n-1; j++)
  for (k=1; k<n-j; k++)
  {
    if (class[k-1].roll >
                   class[k].roll)
    {
       t = class[k-1];
       class[k-1] = class[k];
       class[k] = t;
     }
   }
    <<<< PRINT THE RECORDS >>>>
}
```

Some Points

- `class` is a pointer to the starting address of the allocated memory block.
- We can therefore access the individual elements of the structure array using array notation:
  - Example: `class[k]`, `class[k].roll`, etc.
- As an alternative, we can also access using *pointers* and the *arrow notation*:
  - Example: `(class+k)`, `(class+k)->roll`, etc.

Altering the Size of a Block

- Sometimes we need to alter the size of some previously allocated memory block.
  - More memory needed.
  - Memory allocated is larger than necessary.
- How?
  - By using the `realloc` function.
- If the original allocation is done as:

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

then reallocation of space may be done as:

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

Contd.

- The new memory block may or may not begin at the same place as the old one.
    - If it does not find space, it will create it in an entirely different region and move the contents of the old block into the new block.
- The function guarantees that the old data remains intact.
- If it is unable to allocate, it returns `NULL` and frees the original block.