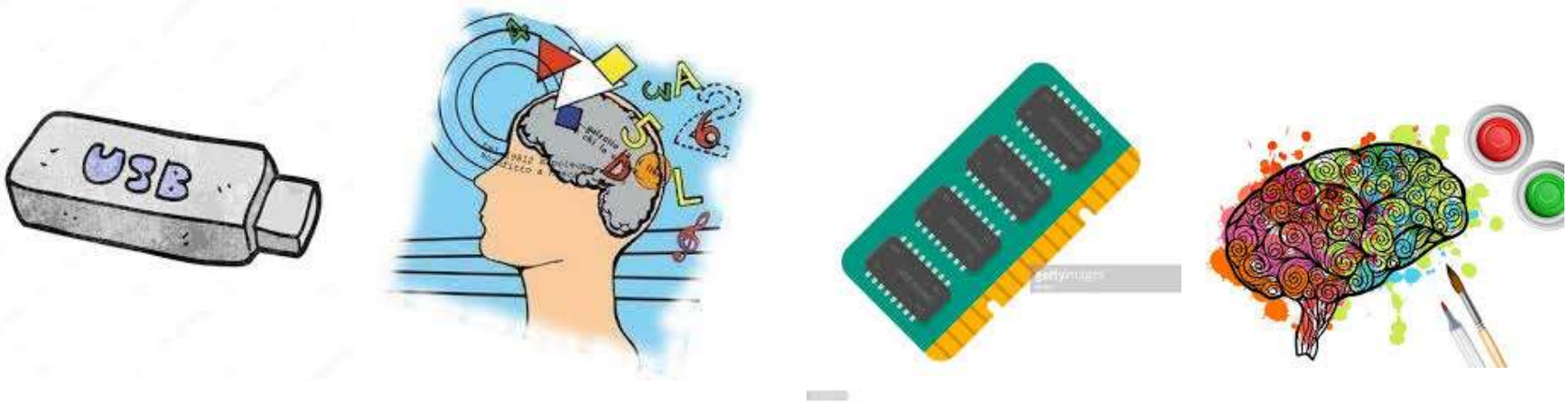


1.6 Almacenamiento en tiempo de ejecución

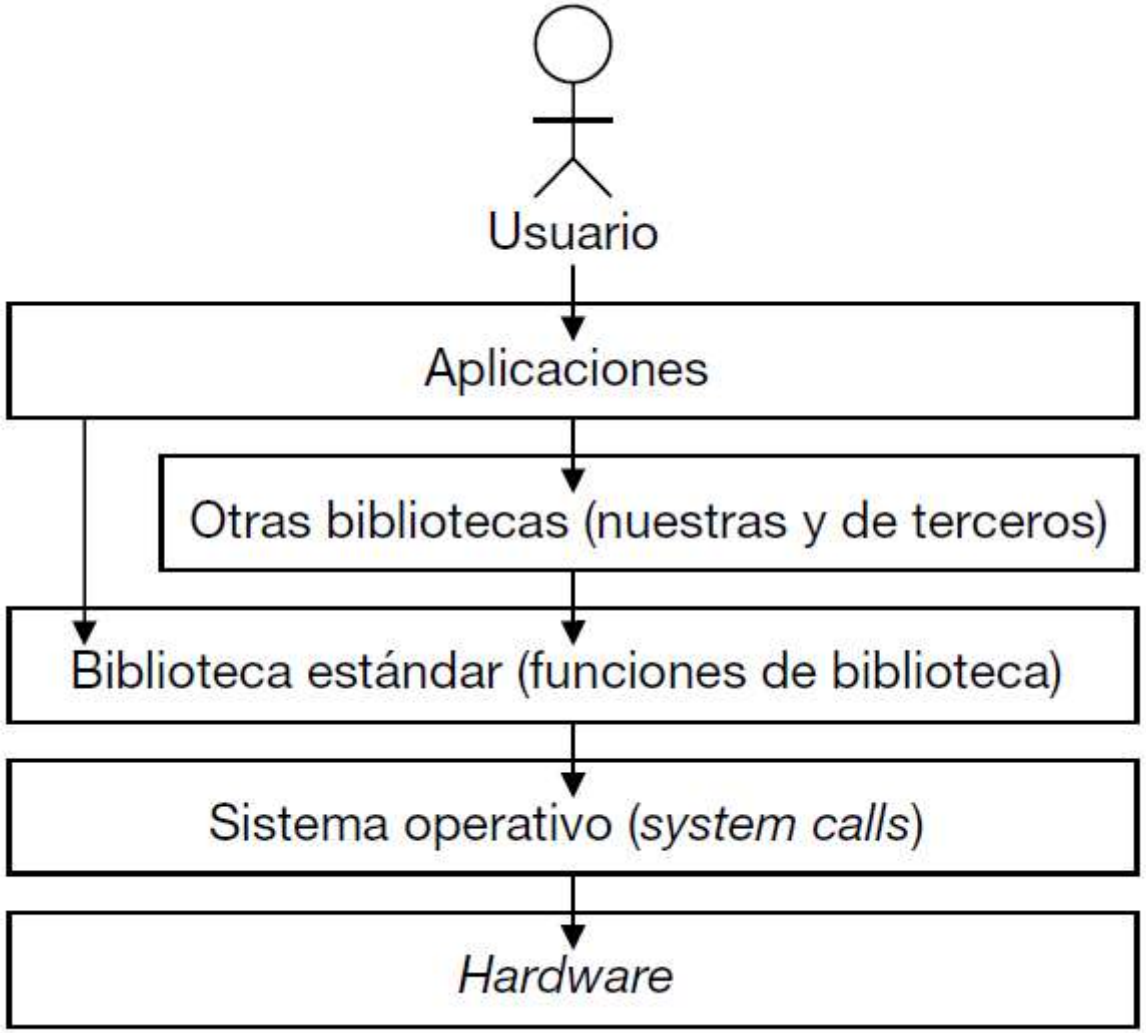
1.6.1 Memoria dinámica

- Se trata del almacenamiento que se solicita durante la ejecución de un programa.
- A medida que un programa en ejecución realiza operaciones y genera nuevos datos requiere más espacio, el cual es administrado por el sistema operativo para almacenar estos datos.

1.6.1 Memoria dinámica

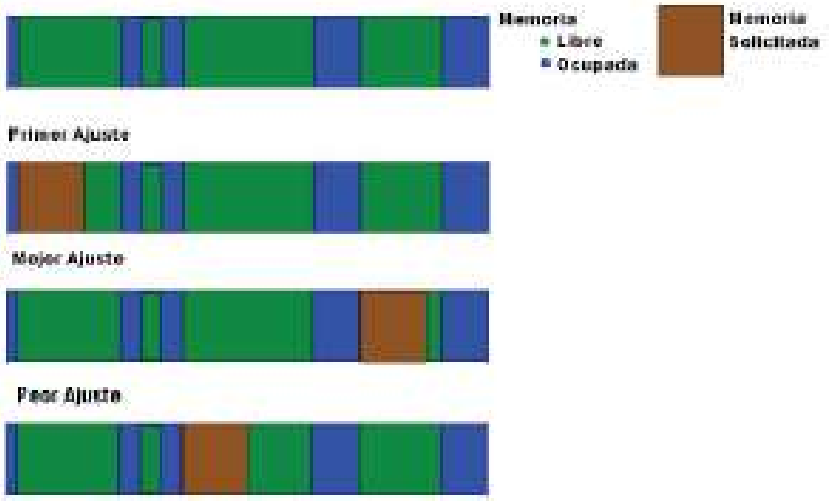
- Para los programadores (capa de abstracción de desarrollo) existen diferentes mecanismos para el manejo de memoria la mayoría trabajan de manera automática y algunos otros se consideran “manuales”.
- En el lenguaje C el mecanismo para el manejo de memoria de forma directa es el apuntador.

Recordando… Capas de abstracción



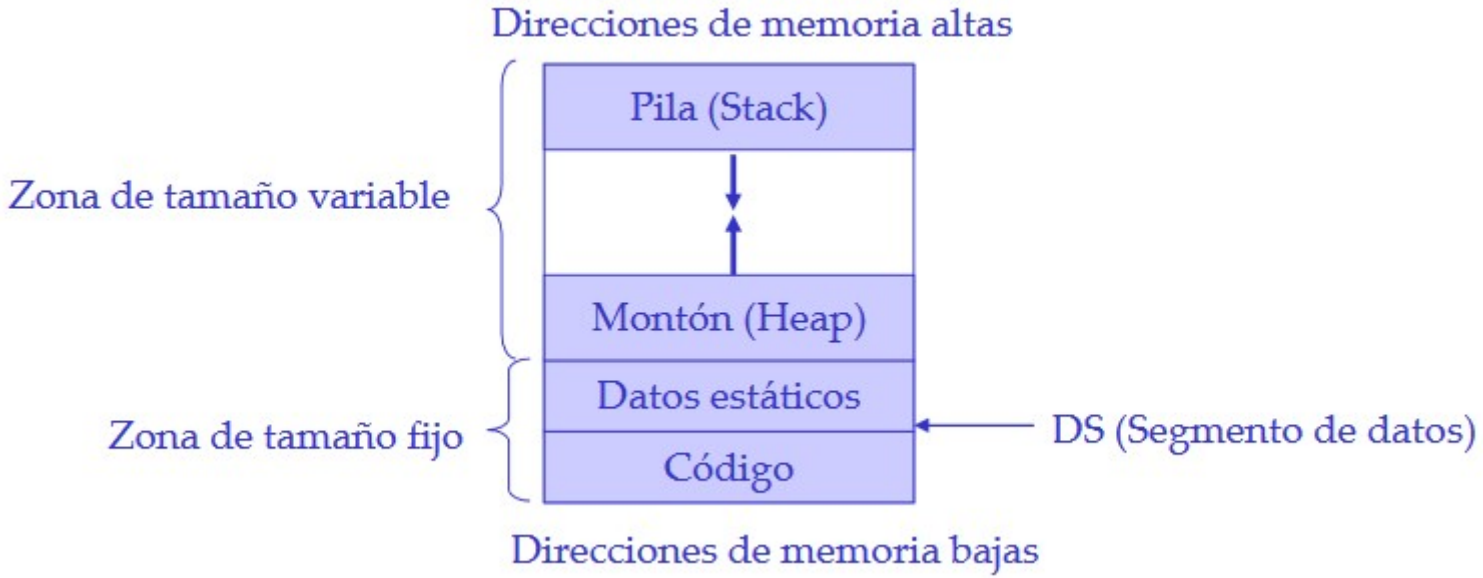
1.6.2 Adm de memoria dinámica (HW – S.O.)

- Un problema común de asignación de memoria dinámica, es cómo satisfacer una necesidad de tamaño n con una lista de huecos libres. Las estrategias más comunes para asignar algún hueco son:
  - Primer ajuste
  - Mejor ajuste
  - Peor ajuste



1.6.3 Adm. de Memoria en el lenguaje

- La organización de memoria en tiempo de ejecución en el siguiente nivel de abstracción (programación de alto nivel) depende del tipo de lenguaje.
- En general en los lenguajes estructurados se establece de la siguiente manera:



1.6.3 Adm. de Memoria en el lenguaje

- Al ejecutar un programa, el primer responsable de la memoria es el sistema operativo.
- En el proceso de compilación un elemento llamado “cargador” asigna la cantidad y el lugar de memoria para ubicar el programa en la zona correspondiente.
- El sistema operativo detecta la posible colisión entre el “heap” y la pila. En esos casos puede cerrar el programa o aumentar la cantidad de memoria asignada

1.6.3.1 Stack & Heap

- La pila es una región especial de memoria que almacena de manera temporal las variables creadas por cada función.
- Cada vez que se declara una variable, se mete a la pila, y cuando se finaliza la ejecución de la función respectiva, se libera de manera automática ese espacio de memoria.
- La ventaja del uso de la pila para almacenar variables es que la administración de memoria se realiza automáticamente.
- El tamaño de las variables que se pueden almacenar es limitado.

1.6.3.1 Stack & Heap

- El montículo es una región de la memoria que no es administrada de manera automática.
- Se puede ver como una sección “libre” es de mayor tamaño que la pila
- A diferencia de la pila, el heap no tiene restricciones en el tamaño de las variables (mas allá de la memoria disponible)

1.6.3.1 Stack & Heap

Ventajas y Desventajas

Stack

- Acceso más rápido
- No se tiene que liberar la memoria de manera explicita.
- El espacio se maneja de manera eficiente
- Las variables no se pueden cambiar de tamaño
- Las variables generalmente mantienen un ámbito local

1.6.3.1 Stack & Heap

Ventajas y Desventajas

Heap

- No hay límite en el tamaño de la memoria (hablando en términos de variables)
- Acceso más lento que en la pila
- El programador debe administrar la memoria

1.6.3.2 Funciones del lenguaje

- El lenguaje C cuenta con funciones definidas dentro de su biblioteca estándar para que el programador pueda manejar y administrar la memoria de los programas.
- Estas funciones tienen diferentes características de acuerdo a su uso
- Las funciones principales son
    - **malloc()**
    - **calloc()**
    - **realloc()**
    - **free()**

malloc

- La función malloc es la forma de implementar el uso de memoria dinámica. Está definida en la biblioteca stdlib.h.

```
void* malloc (size_t tamaño);
```

La función malloc toma como argumento el número de bytes a asignarse y devuelve un apuntador (***tipo void***) que corresponde al espacio de memoria asignada.

Un apuntador void puede asignarse a una variable de cualquier tipo.

malloc

- Cuando se utiliza la función malloc, devuelve un apuntador a una localidad de memoria específica a partir de la cual se reserva la cantidad de espacio solicitada.
- El manejo de ese apuntador se realiza de la misma manera en un arreglo.
- El apuntador devuelto por malloc es un apuntador especial por lo cual no es necesario asociarlo con alguna variable.

malloc (ejemplo)

```
main(){
     int *apuntador;
     apuntador = (int *)malloc(4);
     *apuntador = 10;
     *(apuntador+1) = 100;
     *(apuntador+2) = 200;
     *(apuntador+3) = 300;

     for (int i=0;i<4;i++){
          printf("%d \n",*(apuntador+i));
     }
}
```

calloc

- La función calloc, recibe como argumento dos parámetros, el primero es el numero de localidades a reservar y el segundo es el tamaño de cada una de esas localidades

```
void* calloc (size_t num_loc, size_t tamaño);
```

- Devuelve un apuntador “void” y establece ceros como valores iniciales en las localidades de memoria que reserva.

realloc

- Esta función redistribuye un área de memoria la cual fue previamente asignada por *malloc* o *calloc.*

```
void* realloc(void* ptr, int size);
```

- Si no hay suficiente memoria para la redistribución se devuelve un apuntador nulo y el espacio de memoria previo no se libera.

Ejemplo (realloc)

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

int main() {
    int *ptr = (int *)malloc(sizeof(int)*3);
    int i;
    int *ptr_new;
    * ptr = 10;
    *(ptr + 1) = 20;
    *(ptr + 2) = 30;

    ptr_new = (int *)realloc(ptr, sizeof(int)*10);

    for(i = 0; i < 10; i++){
        *(ptr_new+i)=i*10;
        printf("%d ", *(ptr_new + i));
    }
}
```

free

- Cuando una zona de memoria reservada con malloc ya no se necesita, debe ser liberada mediante la función free

```
void free (void* ptr);
```

- ptr es un apuntador de cualquier tipo que apunta a un área reservada previamente por malloc

sizeof

- Devuelve el tamaño de una variable (en bytes)

sizeof(var)

El tamaño se puede ver como la cantidad de localidades de memoria que requiere la variable.

Ejemplo (sizeof)

```
main(){
     int a = 10;
     float f = 100;
     int b = sizeof(a);
     printf("%d \n",b);
}

float b = sizeof(a);
long b = sizeof(a);
```