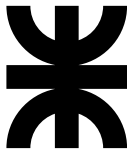


Informática II

Asignación dinámica de memoria

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2019 –

Asignación dinámica de memoria

- ▶ Asignación dinámica de memoria es la capacidad de obtener más memoria para datos en tiempo de ejecución de un programa

Asignación dinámica de memoria

- ▶ Asignación dinámica de memoria es la capacidad de obtener más memoria para datos en tiempo de ejecución de un programa
- ▶ Se puede solicitar y devolver memoria al sistema operativo

Asignación dinámica de memoria

- ▶ Asignación dinámica de memoria es la capacidad de obtener más memoria para datos en tiempo de ejecución de un programa
- ▶ Se puede solicitar y devolver memoria al sistema operativo
- ▶ La asignación dinámica de memoria, o memoria dinámica existe en un segmento llamado *heap*

Asignación dinámica de memoria

- ▶ Asignación dinámica de memoria es la capacidad de obtener más memoria para datos en tiempo de ejecución de un programa
- ▶ Se puede solicitar y devolver memoria al sistema operativo
- ▶ La asignación dinámica de memoria, o memoria dinámica existe en un segmento llamado *heap*
- ▶ Las funciones `malloc` y `free`, junto al operador `sizeof` se utilizan en la asignación dinámica de memoria (archivo de cabecera `stdlib.h`)

Disposición de la memoria en un programas C

- ▶ Cuando se carga un programa en memoria, esta queda organizada en bloques llamados *segmentos*.

Disposición de la memoria en un programas C

- ▶ Cuando se carga un programa en memoria, esta queda organizada en bloques llamados *segmentos*.
- ▶ Cada segmento tiene permisos de lectura, escritura y ejecución

Disposición de la memoria en un programas C

- ▶ Cuando se carga un programa en memoria, esta queda organizada en bloques llamados *segmentos*.
- ▶ Cada segmento tiene permisos de lectura, escritura y ejecución

Los segmentos son:

Disposición de la memoria en un programas C

- ▶ Cuando se carga un programa en memoria, esta queda organizada en bloques llamados *segmentos*.
- ▶ Cada segmento tiene permisos de lectura, escritura y ejecución

Los segmentos son:

text: contiene código binario del programa (solo lectura). Una única copia para varias instancias del programa.

Disposición de la memoria en un programas C

- ▶ Cuando se carga un programa en memoria, esta queda organizada en bloques llamados *segmentos*.
- ▶ Cada segmento tiene permisos de lectura, escritura y ejecución

Los segmentos son:

text: contiene código binario del programa (solo lectura). Una única copia para varias instancias del programa.

data: subdividido en dos

Disposición de la memoria en un programas C

- ▶ Cuando se carga un programa en memoria, esta queda organizada en bloques llamados *segmentos*.
- ▶ Cada segmento tiene permisos de lectura, escritura y ejecución

Los segmentos son:

text: contiene código binario del programa (solo lectura). Una única copia para varias instancias del programa.

data: subdividido en dos

- ▶ Datos no inicializados – variables globales y estáticas

Disposición de la memoria en un programas C

- ▶ Cuando se carga un programa en memoria, esta queda organizada en bloques llamados *segmentos*.
- ▶ Cada segmento tiene permisos de lectura, escritura y ejecución

Los segmentos son:

text: contiene código binario del programa (solo lectura). Una única copia para varias instancias del programa.

data: subdividido en dos

- ▶ Datos no inicializados – variables globales y estáticas
- ▶ Datos inicializados explícitamente – variables globales, estáticas, y datos constantes (solo lectura)

Disposición de la memoria en un programas C

- ▶ Cuando se carga un programa en memoria, esta queda organizada en bloques llamados *segmentos*.
- ▶ Cada segmento tiene permisos de lectura, escritura y ejecución

Los segmentos son:

text: contiene código binario del programa (solo lectura). Una única copia para varias instancias del programa.

data: subdividido en dos

- ▶ Datos no inicializados – variables globales y estáticas
- ▶ Datos inicializados explícitamente – variables globales, estáticas, y datos constantes (solo lectura)

stack: para almacenar variables locales, argumentos pasados a funciones, y dirección de retorno (crece hacia abajo) [stack frames]

Disposición de la memoria en un programas C

- ▶ Cuando se carga un programa en memoria, esta queda organizada en bloques llamados *segmentos*.
- ▶ Cada segmento tiene permisos de lectura, escritura y ejecución

Los segmentos son:

text: contiene código binario del programa (solo lectura). Una única copia para varias instancias del programa.

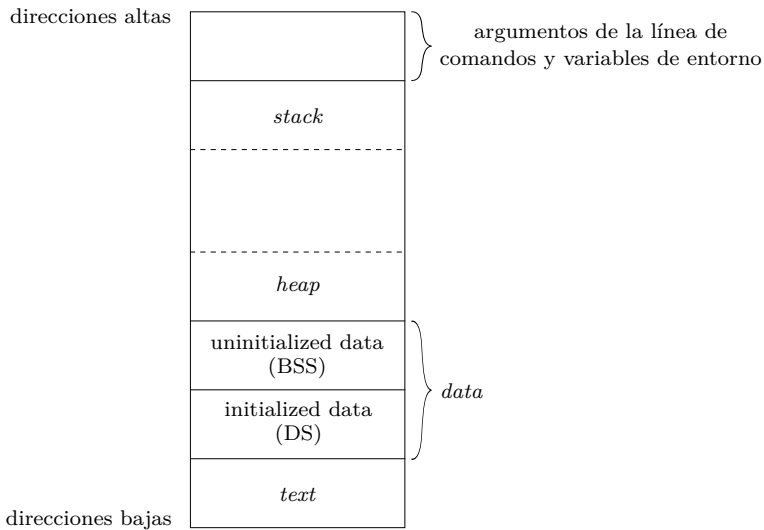
data: subdividido en dos

- ▶ Datos no inicializados – variables globales y estáticas
- ▶ Datos inicializados explícitamente – variables globales, estáticas, y datos constantes (solo lectura)

stack: para almacenar variables locales, argumentos pasados a funciones, y dirección de retorno (crece hacia abajo) [stack frames]

heap: cuando se asigna memoria (`malloc`) en tiempo de ejecución la memoria se obtiene del *heap* (crece hacia arriba)

Disposición de la memoria en un programas C



Segmento de datos – Ejemplos

Una cadena definida como

```
char s[] = "Hola mundo";
```

o un enunciado

```
int debug = 1;
```

fuera de `main` (global) almacena las variables en el área de datos inicializados como read-write.

Segmento de datos – Ejemplos

Una cadena definida como

```
char s[] = "Hola mundo";
```

o un enunciado

```
int debug = 1;
```

fuera de `main` (global) almacena las variables en el área de datos inicializados como read-write.

Un enunciado global como

```
const char *string = "Hola mundo";
```

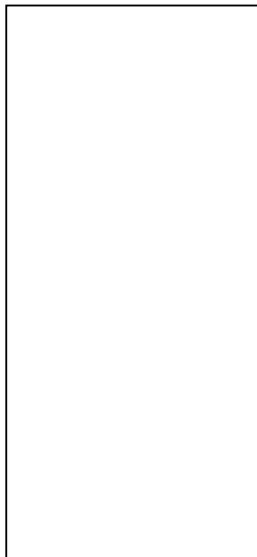
almacena la cadena literal en el área inicializada como read-only, y la variable puntero en el área inicializada como read-write.

Segmento *stack* – Ejemplo

```
1 #include <stdio.h>
2 int res; /* En BSS */
3
4 int cuadrado(int x)
5 {
6     return x*x;
7 }
8
9 int cuadradoDeSuma(int x, int y)
10 {
11     int z = cuadrado(x+y);
12     return z;
13 }
14
15 int main(void)
16 {
17     int a = 2, b = 3;
18     res = cuadradoDeSuma(a, b);
19     printf("Resultado: %d\n", res);
20     return 0;
21 }
```

Segmento *stack* – Ejemplo

```
1 #include <stdio.h>
2 int res; /* En BSS */
3
4 int cuadrado(int x)
5 {
6     return x*x;
7 }
8
9 int cuadradoDeSuma(int x, int y)
10 {
11     int z = cuadrado(x+y);
12     return z;
13 }
14
15 int main(void)
16 {
17     int a = 2, b = 3;
18     res = cuadradoDeSuma(a, b);
19     printf("Resultado: %d\n", res);
20     return 0;
21 }
```



Segmento *stack* – Ejemplo

```
1 #include <stdio.h>
2 int res; /* En BSS */
3
4 int cuadrado(int x)
5 {
6     return x*x;
7 }
8
9 int cuadradoDeSuma(int x, int y)
10 {
11     int z = cuadrado(x+y);
12     return z;
13 }
14
15 int main(void)
16 {
17     int a = 2, b = 3;
18     res = cuadradoDeSuma(a, b);
19     printf("Resultado: %d\n", res);
20     return 0;
21 }
```



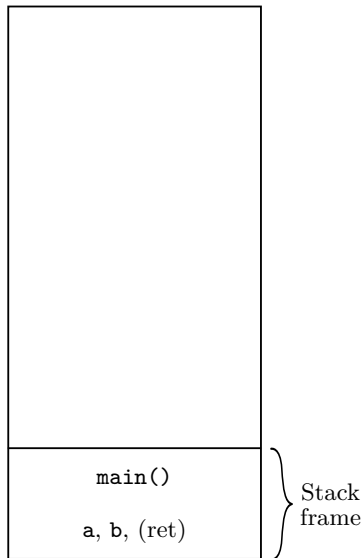
Segmento *stack* – Ejemplo

```
1 #include <stdio.h>
2 int res; /* En BSS */
3
4 int cuadrado(int x)
5 {
6     return x*x;
7 }
8
9 int cuadradoDeSuma(int x, int y)
10 {
11     int z = cuadrado(x+y);
12     return z;
13 }
14
15 int main(void)
16 {
17     int a = 2, b = 3;
18     res = cuadradoDeSuma(a, b);
19     printf("Resultado: %d\n", res);
20     return 0;
21 }
```



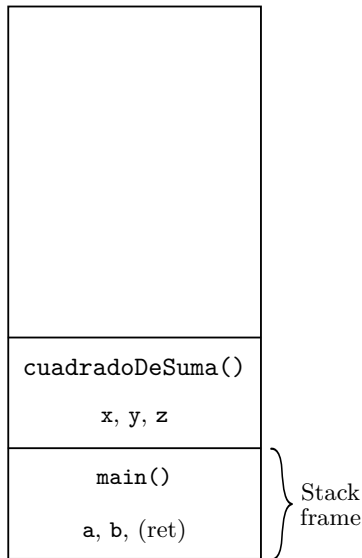
Segmento *stack* – Ejemplo

```
1 #include <stdio.h>
2 int res; /* En BSS */
3
4 int cuadrado(int x)
5 {
6     return x*x;
7 }
8
9 int cuadradoDeSuma(int x, int y)
10 {
11     int z = cuadrado(x+y);
12     return z;
13 }
14
15 int main(void)
16 {
17     int a = 2, b = 3;
18     res = cuadradoDeSuma(a, b);
19     printf("Resultado: %d\n", res);
20     return 0;
21 }
```



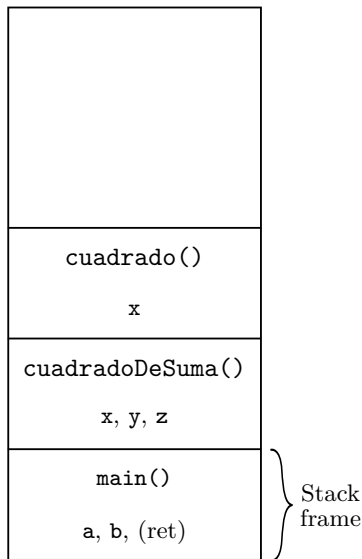
Segmento *stack* – Ejemplo

```
1 #include <stdio.h>
2 int res; /* En BSS */
3
4 int cuadrado(int x)
5 {
6     return x*x;
7 }
8
9 int cuadradoDeSuma(int x, int y)
10 {
11     int z = cuadrado(x+y);
12     return z;
13 }
14
15 int main(void)
16 {
17     int a = 2, b = 3;
18     res = cuadradoDeSuma(a, b);
19     printf("Resultado: %d\n", res);
20     return 0;
21 }
```



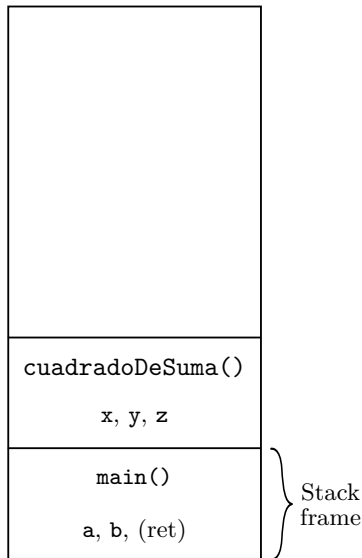
Segmento *stack* – Ejemplo

```
1 #include <stdio.h>
2 int res; /* En BSS */
3
4 int cuadrado(int x)
5 {
6     return x*x;
7 }
8
9 int cuadradoDeSuma(int x, int y)
10 {
11     int z = cuadrado(x+y);
12     return z;
13 }
14
15 int main(void)
16 {
17     int a = 2, b = 3;
18     res = cuadradoDeSuma(a, b);
19     printf("Resultado: %d\n", res);
20     return 0;
21 }
```



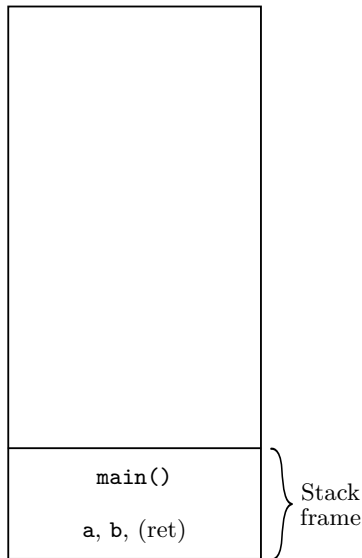
Segmento *stack* – Ejemplo

```
1 #include <stdio.h>
2 int res; /* En BSS */
3
4 int cuadrado(int x)
5 {
6     return x*x;
7 }
8
9 int cuadradoDeSuma(int x, int y)
10 {
11     int z = cuadrado(x+y);
12     return z;
13 }
14
15 int main(void)
16 {
17     int a = 2, b = 3;
18     res = cuadradoDeSuma(a, b);
19     printf("Resultado: %d\n", res);
20     return 0;
21 }
```



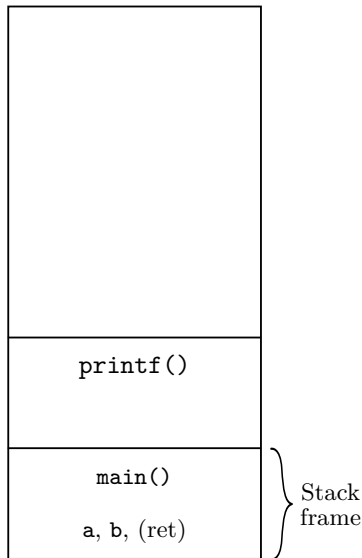
Segmento *stack* – Ejemplo

```
1 #include <stdio.h>
2 int res; /* En BSS */
3
4 int cuadrado(int x)
5 {
6     return x*x;
7 }
8
9 int cuadradoDeSuma(int x, int y)
10 {
11     int z = cuadrado(x+y);
12     return z;
13 }
14
15 int main(void)
16 {
17     int a = 2, b = 3;
18     res = cuadradoDeSuma(a, b);
19     printf("Resultado: %d\n", res);
20     return 0;
21 }
```



Segmento *stack* – Ejemplo

```
1 #include <stdio.h>
2 int res; /* En BSS */
3
4 int cuadrado(int x)
5 {
6     return x*x;
7 }
8
9 int cuadradoDeSuma(int x, int y)
10 {
11     int z = cuadrado(x+y);
12     return z;
13 }
14
15 int main(void)
16 {
17     int a = 2, b = 3;
18     res = cuadradoDeSuma(a, b);
19     printf("Resultado: %d\n", res);
20     return 0;
21 }
```



Asignación dinámica de memoria

Ciclo de vida de las variables en el *heap* son:

1. asignar espacio a la variable usando `malloc()` o `calloc()`

Asignación dinámica de memoria

Ciclo de vida de las variables en el *heap* son:

1. asignar espacio a la variable usando `malloc()` o `calloc()`
2. redimensionar la variable utilizando `realloc()` [Opcional]

Asignación dinámica de memoria

Ciclo de vida de las variables en el *heap* son:

1. asignar espacio a la variable usando `malloc()` o `calloc()`
2. redimensionar la variable utilizando `realloc()` [Opcional]
3. liberar la memoria con `free()`

Asignación dinámica de memoria

Ciclo de vida de las variables en el *heap* son:

1. asignar espacio a la variable usando `malloc()` o `calloc()`
2. redimensionar la variable utilizando `realloc()` [Opcional]
3. liberar la memoria con `free()`

Prototipos de funciones – archivo de cabecera `stdlib.h`

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


Asignación dinámica de memoria

Ciclo de vida de las variables en el *heap* son:

1. asignar espacio a la variable usando `malloc()` o `calloc()`
2. redimensionar la variable utilizando `realloc()` [Opcional]
3. liberar la memoria con `free()`

Prototipos de funciones – archivo de cabecera `stdlib.h`

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

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

Asignación dinámica de memoria

Ciclo de vida de las variables en el *heap* son:

1. asignar espacio a la variable usando `malloc()` o `calloc()`
2. redimensionar la variable utilizando `realloc()` [Opcional]
3. liberar la memoria con `free()`

Prototipos de funciones – archivo de cabecera `stdlib.h`

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

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

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

Asignación dinámica de memoria

Ciclo de vida de las variables en el *heap* son:

1. asignar espacio a la variable usando `malloc()` o `calloc()`
2. redimensionar la variable utilizando `realloc()` [Opcional]
3. liberar la memoria con `free()`

Prototipos de funciones – archivo de cabecera `stdlib.h`

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

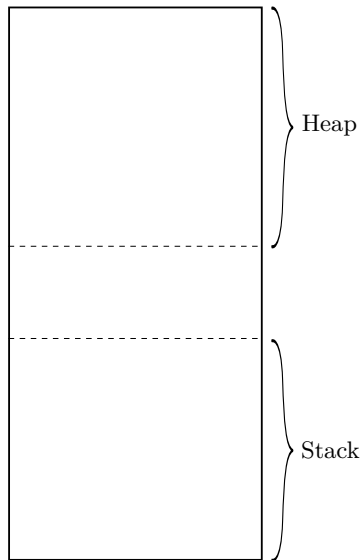
```
void *calloc(size_t nmemb, size_t size);
```

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

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

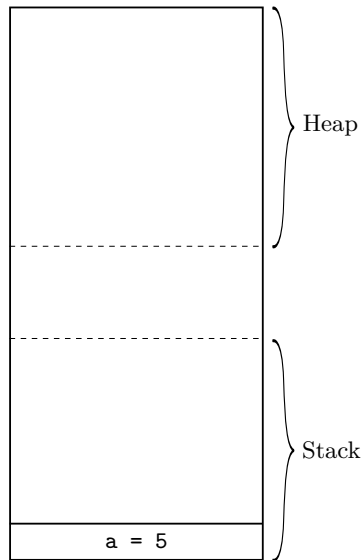
Segmento *heap* – Ejemplo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int a = 5; /* ¿Dónde está? */
7
8
9
10
11
12
13
14
15
16
17     return 0;
18 }
```



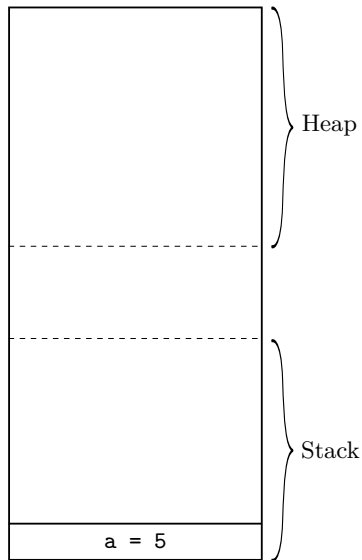
Segmento *heap* – Ejemplo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int a = 5; /* ¿Dónde está? */
7
8
9
10
11
12
13
14
15
16
17     return 0;
18 }
```



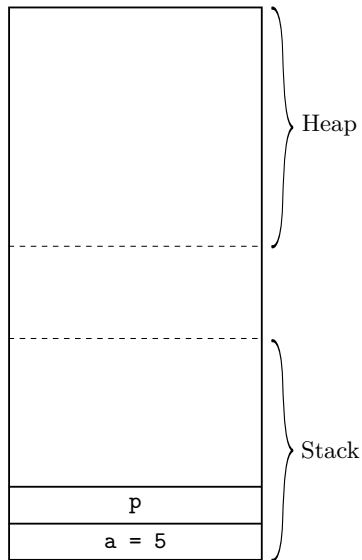
Segmento *heap* – Ejemplo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int a = 5; /* En stack */
7
8     int *p;
9
10
11
12
13
14
15
16
17     return 0;
18 }
```



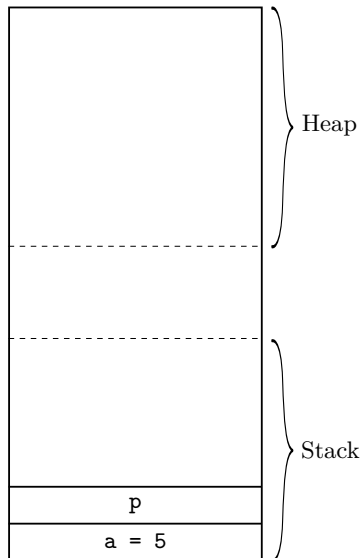
Segmento *heap* – Ejemplo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int a = 5; /* En stack */
7
8     int *p;
9
10
11
12
13
14
15
16
17     return 0;
18 }
```



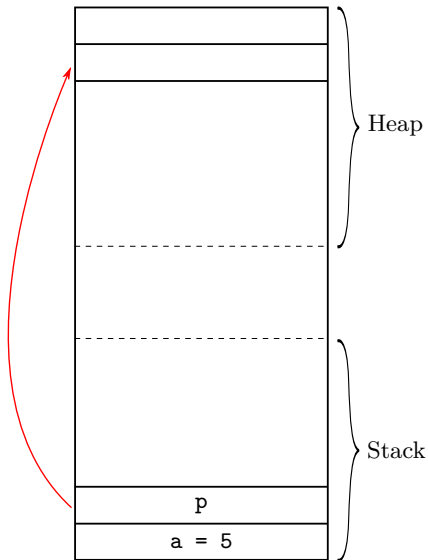
Segmento *heap* – Ejemplo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int a = 5; /* En stack */
7
8     int *p;
9     p = (int *)malloc(sizeof(int));
10
11
12
13
14
15
16
17     return 0;
18 }
```



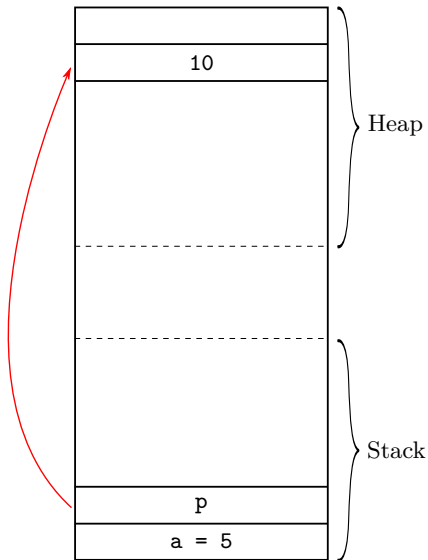
Segmento *heap* – Ejemplo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int a = 5; /* En stack */
7
8     int *p;
9     p = (int *)malloc(sizeof(int));
10
11
12
13
14
15
16
17     return 0;
18 }
```



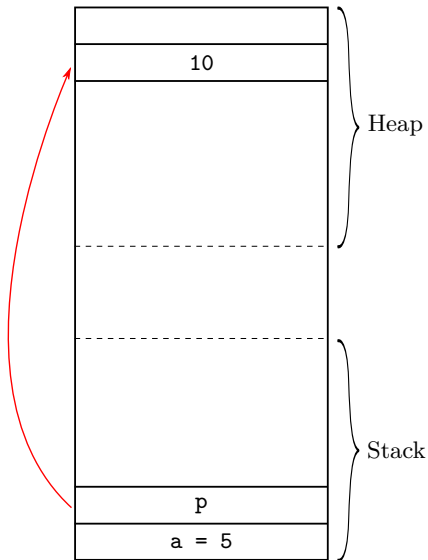
Segmento *heap* – Ejemplo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int a = 5; /* En stack */
7
8     int *p;
9     p = (int *)malloc(sizeof(int));
10    *p = 10;
11
12
13
14
15
16
17    return 0;
18 }
```



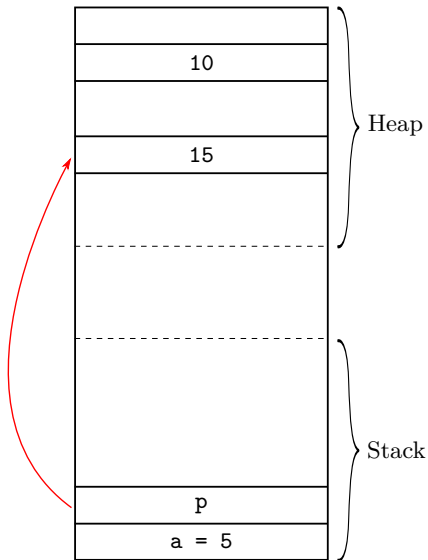
Segmento *heap* – Ejemplo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int a = 5; /* En stack */
7
8     int *p;
9     p = (int *)malloc(sizeof(int));
10    *p = 10;
11
12
13    p = (int *)malloc(sizeof(int));
14    *p = 15;
15
16
17    return 0;
18 }
```



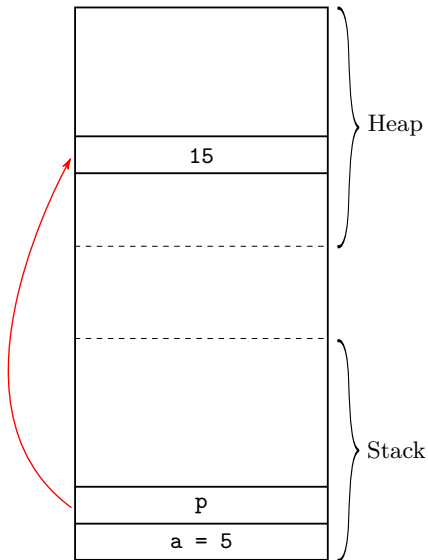
Segmento *heap* – Ejemplo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int a = 5; /* En stack */
7
8     int *p;
9     p = (int *)malloc(sizeof(int));
10    *p = 10;
11
12
13    p = (int *)malloc(sizeof(int));
14    *p = 15;
15
16
17    return 0;
18 }
```



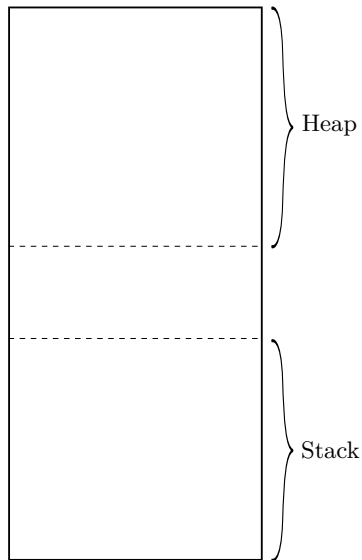
Segmento *heap* – Ejemplo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int a = 5; /* En stack */
7
8     int *p;
9     p = (int *)malloc(sizeof(int));
10    *p = 10;
11    free(p);
12
13    p = (int *)malloc(sizeof(int));
14    *p = 15;
15
16
17    return 0;
18 }
```



Segmento *heap* – Ejemplo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int a = 5; /* En stack */
7
8     int *p;
9     p = (int *)malloc(sizeof(int));
10    *p = 10;
11    free(p);
12
13    p = (int *)malloc(sizeof(int));
14    *p = 15;
15    free(p);
16
17    return 0;
18 }
```



Asignación dinámica de memoria – Ejemplos

Tipos básicos y arreglos

```
int *edad = malloc(sizeof(int));
```

Asignación dinámica de memoria – Ejemplos

Tipos básicos y arreglos

```
int *edad = malloc(sizeof(int));
```

```
double *promedio = malloc(sizeof(double));
```


Asignación dinámica de memoria – Ejemplos

Tipos básicos y arreglos

```
int *edad = malloc(sizeof(int));
```

```
double *promedio = malloc(sizeof(double));
```

```
char *cadena = malloc(20 * sizeof(char));
```

Asignación dinámica de memoria – Ejemplos

Tipos básicos y arreglos

```
int *edad = malloc(sizeof(int));
```

```
double *promedio = malloc(sizeof(double));
```

```
char *cadena = malloc(20 * sizeof(char));
```

```
float *lista = malloc(3 * sizeof(float));
```

Asignación dinámica de memoria – Ejemplos

Tipos básicos y arreglos

```
int *edad = malloc(sizeof(int));

double *promedio = malloc(sizeof(double));

char *cadena = malloc(20 * sizeof(char));

float *lista = malloc(3 * sizeof(float));
```

Estructuras

```
struct paciente {
    char apellido[20];
    char nombre[20];
    int edad;
    float peso;
    float altura;
};

struct paciente *jperez = malloc(sizeof(struct paciente));
```

Asignación dinámica de memoria – Ejemplos

Tipos básicos y arreglos

```
int *edad = malloc(sizeof(int));

double *promedio = malloc(sizeof(double));

char *cadena = malloc(20 * sizeof(char));

float *lista = malloc(3 * sizeof(float));
```

Estructuras

```
struct paciente {
    char apellido[20];
    char nombre[20];
    int edad;
    float peso;
    float altura;
};

struct paciente *jperez = malloc(sizeof(struct paciente));
```

No olvidar liberar la memoria con `free()` [Memory leak]

Actividad práctica

1. Escribir un programa que reserve espacio en memoria para las variables de tipos básicos del ejemplo, le asigne valores, imprima los valores, y libere la memoria.
2. Escribir un programa que calcule y muestre en pantalla el promedio de una serie de valores enteros. La interacción con el usuario es la siguiente:
 - ▶ Le solicita la cantidad de valores a promediar
 - ▶ Le solicita los datos
 - ▶ Imprime el promedio calculado

Calcular el promedio con una función que tenga el siguiente prototipo

```
float promedio(int * , int );
```

El programa debe reservar memoria para almacenar los datos previo a llamar a la función. Interacción con el usuario (entrada/salida):

```
Ingrese la cantidad de datos a promediar: 4
Ingrese el dato #1: 1
Ingrese el dato #2: 3
Ingrese el dato #3: 7
Ingrese el dato #4: 11
El promedio es: 5.500000
```

