Gestión dinámica de memoria

Sofía Beatriz Pérez
Daniel Agustín Rosso

sperez@iua.edu.ar
drosso@iua.edu.ar

Centro Regional Univesitario Córdoba
Instituto Univeristario Areonáutico

Informática II - Clase número 3 - Ciclo lectivo 2022

Agenda

Disclaimer

Los siguientes slides tienen el objetivo de dar soporte al dictado de la asignatura. De ninguna manera pueden sustituir los apuntes tomados en clases y/o la asistencia a las mismas.
Es importante mencionar que todos este material se encuentra en un proceso de mejora continua.
Si encuentra bugs, errores de ortografía o redacción, por favor repórtelo a sperez@iua.edu.ar y/o drosso@iua.edu.ar. También puede abrir issues en el repositorio de este link: ▸ infoI_IUA_GitLab

Introducción

La asignación dinámica de memoria significa que un programa puede solicitar o devolver memoria al sistema operativo en **tiempo de ejecución**. Es decir, mientras se está ejecutando.

Para realizar esto en C[1], se utilizan las funciones malloc y free [2], junto al operador sizeof.

[1] C++ usa otras alternativas mejoradas que se estudiarán luego.
[2] Muchos compiladores soportan otras, pero estas son las más importantes

Distribución de la memoria de un programa I

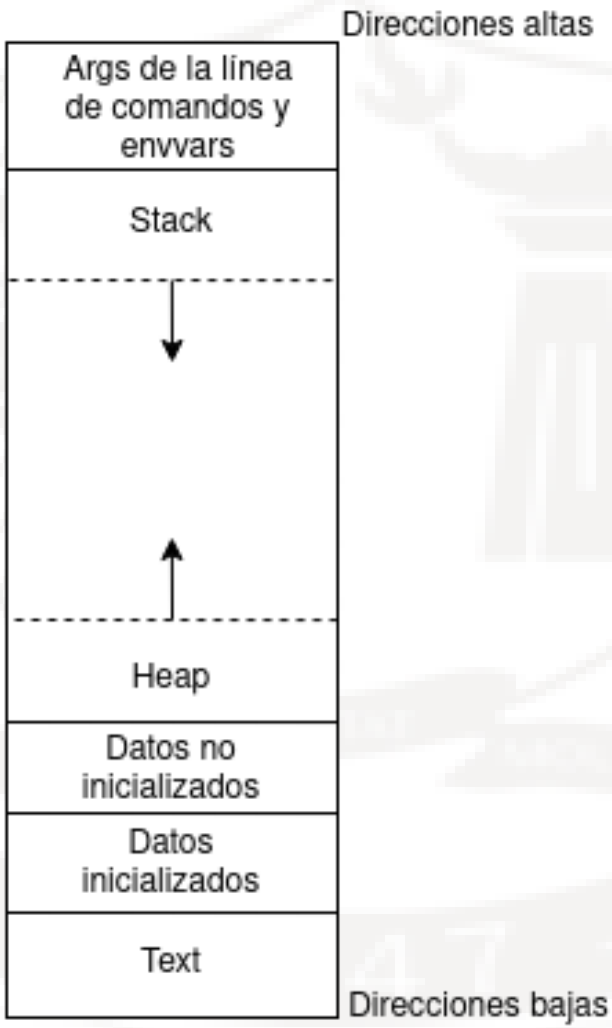
Cuando se carga un programa en memoria, este queda organizado en bloques llamados segmentos:

- text: es también conocido como segmento de código. Es un segmento de sólo lectura y contiene las instrucciones a ejecutar.
- data: se subdivide en dos partes
  - Datos inicializados: almacena las variables globales y estáticas inicializadas previamente por el desarrollador.
  - Datos no inicializados: almacena las variables globales y estáticas no inicializadas previamente.

Distribución de la memoria de un programa II

- stack: almacena variables locales, argumentos pasados a funciones, y las direcciones de memoria a la que el programa debe retornar luego de la ejecución de una función. (Crece hacia abajo)
- heap: es el segmento donde la asignación dinámica de memoria sucede. (Crece hacia arriba)

Distribución de la memoria de un programa III



Las funciones malloc & free I

Cada vez que la función malloc() es llamada, una porción de la memoria libre del sistema es asignada. Por el contrario, cada llamada a la función free() libera memoria para ser utilizada por otro programa o el sistema operativo.

El prototipo de la función malloc() es:

```
1  void * malloc(number_of_bytes);
```

Tras una llamada fructífera devuelve un puntero al primer byte de la región de memoria dispuesta en el montón.

```
1  char *p=NULL;
2  p = (char *) malloc(1000); /*obtener 1000 bytes*/
```

Las funciones malloc & free II

Como el segmento del montón no es infinito, luego de intentar asignar memoria se debe verificar que el puntero sea distinto de NULL. Eso significa que la asignación fue exitosa.

El prototipo de la función free() es:

```
1 void free(void *p);
```

En este ejemplo, se asume que el puntero *p fue asignado previamente por una llamada a malloc. Llamar a free con un argumento inválido, dañaría el sistema de asignación.

El operador size_of I

Es un operador unario que retorna la longitud en **bytes** de la variables precedida por el. A los fines prácticos, se puede tomar el valor retornado por este operador como equivalente a un entero sin signo).

El operador size_of II

```
1  #include <stdio.h>
2  int main()
3  {
4      int     a=10;
5      float   b=10.30;
6      char    c='a';
7      double  d=10.189;
8
9      printf("Espacio en bytes de a: %ld\n",sizeof(a));
10     printf("Espacio en bytes de b: %ld\n",sizeof(b));
11     printf("Espacio en bytes de c: %ld\n",sizeof(c));
12     printf("Espacio en bytes de d: %ld\n",sizeof(d));
13     return(0);
14 }
```

Gestionando memoria dinámica con malloc() I

```
1  #include <stdio.h>
2  #include <stdlib.h>
3   int main()
4   {
5       int      *p=NULL;
6       int cantidad=0,ii=0,valor=0;
7       printf("Ingrese la cantidad de elementos\n");
8       scanf("%d",&cantidad);
9      p=(int *) malloc(cantidad*sizeof(int));
10      if(p==NULL)
11      {
12          printf("No hay memoria disponible\n");
13          exit(1);
14      }
15      else
```

Gestionando memoria dinámica con malloc() II

```
16      {
17          for(ii=0;ii<cantidad;ii++)
18          {
19              printf("Ingrese el elemento %d\n",ii);
20              scanf("%d",&valor);
21              *(p+ii)=valor;
22          }
23
24          for(ii=0;ii<cantidad;ii++)
25          {
26              printf("%d\n",*(p+ii));
27          }
28      }
29      free(p);
30      return(0);}
```

La función realloc() I

La función realloc() cambia el tamaño de la memoria asignado previamente por una llamada a malloc().
El nuevo tamaño reservado de memoria puede ser mayor o menor al asignado previamente.

Protitotipo de realloc();

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

La función realloc() II

```
1  #include <stdio.h>
2
3  #include <stdlib.h>
4
5  int main() {
6     int * p = NULL;
7     int ii=0;
8     /*Solicitando espacio para 15 valores enteros*/
9     p = (int *) malloc(15 * sizeof(int));
10    /*Carga de datos*/
11    for (ii = 0; ii < 15; ii++)
12    {
13       *(p + ii) = ii;
14    }
15
```

La función realloc() III

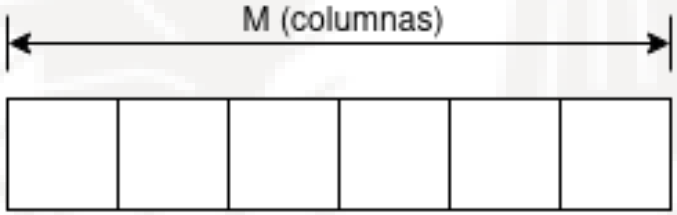
```
16      printf("Start address %p\n", p);
17      /*Impresion de datos*/
18      for (ii = 0; ii < 15; ii++)
19      {
20          printf("%d\n", *(p + ii));
21      }
22      /*Redimension de la memoria*/
23      p = (int *) realloc(p, 25 * sizeof(int));
24      /*Carga de nuevos datos*/
25      for (ii = 0; ii < 10; ii++)
26      {
27          *(p + ii + 14) = ii;
28      }
29
30
```

La función realloc() IV

```
31      printf("Start address %p\n", p);
32    /*Impresion total*/
33      for (ii = 0; ii < 25; ii++)
34      {
35        printf("%d\n", *(p + ii));
36      }
37    /*Impresion de posiciones de memoria */
38      for (ii = 0; ii < 25; ii++)
39      {
40        printf("%p\n", (p + ii));
41      }
42      free(p);
43      return (0);
44  }
```

Creando arrays dinámicos de más de una dimensión I

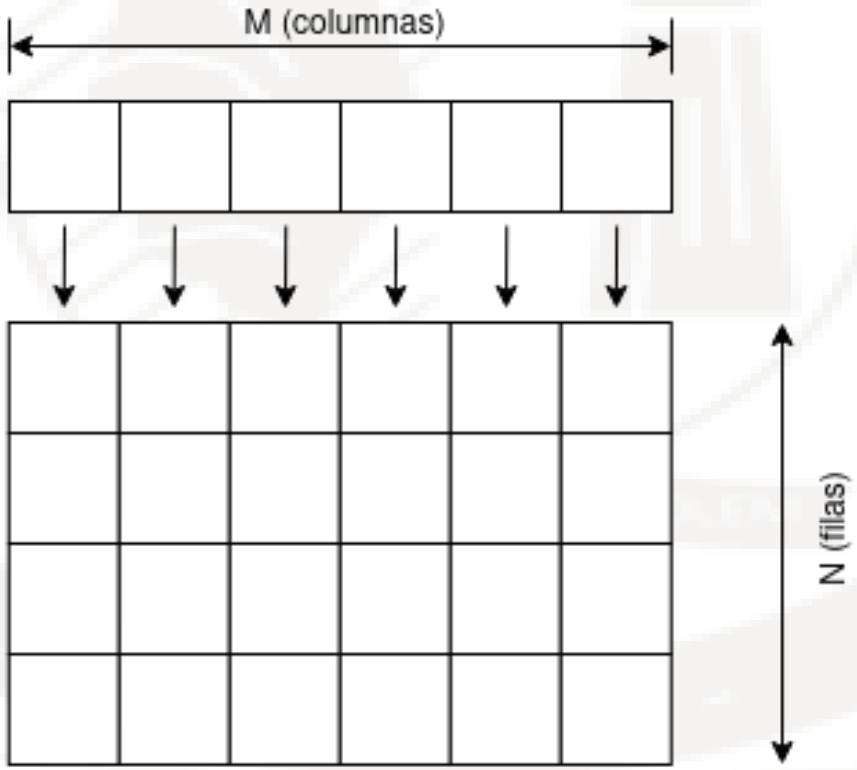
Supongamos que se necesita crear un arreglo de dos dimensiones (M columnas por N filas) de números enteros.
En primer instancia, se creará un arreglo de punteros de tipo int, capaz de almacenar M referencias a columnas:




```
1  int **A = (int **)malloc(M * sizeof(int *));
2  if (A == NULL)
3  { printf("No hay suficiente memoria");
4    exit(0);
5  }
```

Creando arrays dinámicos de más de una dimensión II

Ahora, haremos que cada puntero de M apunte a un espacio de memoria de N elementos:



Creando arrays dinámicos de más de una dimensión III

```
1  for (ii = 0; ii < M; ii++)
2  {
3      A[ii] = (int *)malloc(N * sizeof(int));
4      if (A[ii] == NULL)
5      {
6          printf("No hay suficiente memoria");
7          exit(0);
8      }
9  }
```

Creando arrays dinámicos de más de una dimensión IV

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #define M 4
4  #define N 5
5  int main()
6  {
7      int ii=0,jj=0;
8      /*Columnas*/
9      int **A = (int **)malloc(M * sizeof(int *));
10     if (A == NULL)
11     {
12         printf("No hay memoria suficiente");
13         exit(0);
14     }
15
```

Creando arrays dinámicos de más de una dimensión V

```
16      /*Filas*/
17      for (ii = 0; ii < M; ii++)
18      {
19          *(A+ii) = (int *)malloc(N * sizeof(int));
20          if (*(A+ii) == NULL)
21          {
22              printf("No hay memoria suficiente");
23              exit(0);
24          }
25      }
26
27
28
29
30
```

Creando arrays dinámicos de más de una dimensión VI

```
31          /*Carga de datos*/
32          for (ii = 0; ii < M; ii++)
33          {
34              for (jj = 0; jj < N; jj++) {
35                  *(*(A + ii) + jj) = rand() % 100;
36              }
37          }
38          /*Impresion de  datos*/
39          for (ii = 0; ii < M; ii++)
40          {
41              for (jj = 0; jj < N; jj++) {
42                  printf("%d ", *(*(A + ii) + jj));
43              }
44              printf("\n");
45          }
```

Creando arrays dinámicos de más de una dimensión VII

```
46
47          /*Liberacion de filas*/
48          for (ii = 0; ii < M; ii++) {
49              free(*(A + ii));
50          }
51
52          free(A);
53
54          return(0);
55          }
```

Manejo de memoria dinámica en C++ I

C++ provee dos operadores para el manejo de memoria dinámica **new** y **delete**. Cabe destacar que C++ también soporta malloc() and free(), pero new y delete tiene algunas ventajas como por ejemplo el manejo de excepciones.

```
1  pointer = new type;
2  delete pointer;
```

Si al momento de ejecutar el operador new no hay memoria disponible, el operador fallará arrojado una excepción que debe ser capturada por nuestro código:

Manejo de memoria dinámica en C++ II

```
1  #include <iostream>
2  #include <new>
3  using namespace std;
4  int main()
5  {
6      int *p;
7      try
8      {
9          p = new int;
10     }
11     catch (...)
12     {
13         cout << "No hay suficiente memoria";
14         return(1);
15     }
```

Manejo de memoria dinámica en C++ III

```
16
17        *p=100;
18        cout << "el valor de p es "<<*p;
19
20        }
```

Reportes de valgrind I

```
1  #include<stdlib.h>
2  #include<stdio.h>
3
4
5  #include <stdio.h>
6  #include <stdlib.h>
7   int main()
8   {
9       int      *p=NULL;
10      int ii=0;
11      p=(int *) malloc(10*sizeof(int));
12      if(p==NULL)
13      {
14          printf("No hay memoria disponible\n");
15          exit(1);
```

Reportes de valgrind II

```
16      }
17      else
18      {
19          for(ii=0;ii<15;ii++)
20          {
21              *(p+ii)=ii;
22          }
23
24          for(ii=0;ii<10;ii++)
25          {
26              printf("%d\n",*(p+ii));
27          }
28      }
29      //free(p);
30      return(0);}
```

¡Muchas gracias!

Consultas:

sperez@iua.edu.ar
drosso@iua.edu.ar