ADMINISTRACIÓN DE MEMORIA

La administración de memoria representa a todos los procedimientos y métodos encargados de lograr una máxima utilidad de la misma, es decir, se encarga de organizar los programas y procesos que se ejecutan, de modo que el espacio disponible se aproveche de la mejor manera posible.

1. **Definición y estructura de la memoria**

En general, una computadora tiene dos tipos de memoria.

- La memoria RAM (random-access memory) es usada para ejecutar distintas tareas.
- la memoria del disco es gestionada para almacenar archivos. La memoria misma se organiza en direcciones de memoria o también conocidas como direcciones, cada dirección ocupa un byte de almacenamiento y se establece de manera secuencial.

Generalmente se pueden distinguir tres tipos de memoria.

- En primer lugar, está la ocupada por el código del programa.
- la destinada a datos estáticos, que se gestiona en tiempo de compilación.
- la memoria que debe gestionarse en tiempo de ejecución.

Hay poco que decir sobre la memoria ocupada por el programa. Su tamaño es conocido en tiempo de compilación y, en la mayoría de los lenguajes modernos, se puede decir que es "invisible": el código no puede referirse así mismo ni modificarse.

La memoria destinada a datos estáticos se utiliza para las variables globales y algunas otras como las variables estáticas de las funciones.

La gestión de esta memoria es sencilla y se puede realizar en tiempo de compilación.

Al hablar de la memoria gestionada en tiempo de ejecución, podemos distinguir entre la memoria que se utiliza para albergar los objetos que se crean y destruyen con la ejecución de las funciones (parámetros, variables locales y algunos datos generados por el compilador) y la que se suele conocer como memoria dinámica, que se reserva explícitamente por el programador o que se necesita para almacenar objetos con tiempos de vida o tamaños desconocidos en tiempo de compilación. La memoria asociada a las funciones será gestionada mediante una pila. La memoria dinámica se localiza en el heap.

Disposición de la memoria en C/C++

Después de compilar un programa en C o en C++, se crea un archivo binario ejecutable (.exe) que al ejecutar se carga en la memoria RAM de manera organizada. Los computadores no acceden a las instrucciones del programa directamente del almacenamiento secundario debido a que el tiempo de acceso es mayor comparado al de la memoria RAM. La memoria RAM es más rápida que el almacenamiento secundario, sin embargo, tiene un límite en su capacidad de almacenamiento, por lo tanto para los programadores es necesario hacer uso de esta de la manera más eficiente.

La estructura de la memoria de un programa en C/C++ está constituida principalmente por los siguientes segmentos:

- Segmento de texto o segmento de código

  También conocido como la memoria ocupada por el programa, es una de las secciones de un programa en un archivo de objeto o en la memoria, que contiene instrucciones ejecutables. Como región de memoria, un segmento de texto se puede colocar debajo del montón o la pila para evitar que los montones y los desbordamientos de la pila lo sobrescriban. Por lo general, el segmento de texto se puede compartir, por lo que solo se necesita una copia en la memoria para los programas que se ejecutan con frecuencia, como los editores de texto, el compilador de C, los shells, etc. Además, el segmento de texto suele ser de solo lectura, para evitar que un programa modifique accidentalmente sus instrucciones.

- Segmento de datos inicializados

  Es parte del espacio de direcciones virtuales de un programa, que contiene las variables globales y variables estáticas que inicializa el programador. Debido a que los valores de las variables pueden cambiar durante la ejecución del programa, este segmento de memoria tiene permiso de lectura y escritura.

```c
#incLude <stdio.h>

 K variables globales almacenadas en la parte de
   lectura-escritura del segmento de datos inicializados
  »
int gLobaL_var = 50;
char* heLLo = "HeLLo WorLd";
```

```c
  K variables globales almacenadas en la parte de solo lectura del
   segmento de datos inicializados
  »
const int gLobaL_var2 = 30;

int main() {
    variable estática almacenada en el segmento de datos inicializados
   static int a = 10;
    …
   return 0;
}
```

- Segmento de datos no-inicializados

  También llamado el segmento “bss” (block started by symbol) llamado así por un antiguo operador de ensamblador. El kernel inicializa los datos en este segmento a cero (0) antes de que el programa comience a ejecutarse. Los datos no inicializados comienzan al final del segmento de datos y contienen todas las variables globales y variables estáticas que se inicializan en cero o que no tienen una inicialización explícita en el código fuente.

```
#incLude <stdio.h>

 varible global no inicializada almacenada en el segmento bss
int variabLe_gLobaL;

int main()
{
    varible estática no inicializada almacenada en el segmento bss
   static int variabLe_estatica;

    ¨
   printf("variabLe_gLobaL = %d\n", variabLe_gLobaL;); printf("variabLe estatica =
   %d\n", variabLe_estatica); return 0;
}
```

- Memoria de la pila (Stack)

El segmento de la pila sigue la estructura LIFO (Last In First Out) y crece hasta la dirección inferior, pero depende de la arquitectura de la computadora. La pila crece enla dirección opuesta al Heap. El segmento de pila almacena el valor de las variables locales y los valores de los parámetros pasados a una función junto con información adicional como la dirección de retorno de la instrucción, que se ejecutará después de una llamada a la función.

El registro de puntero de pila realiza un seguimiento de la parte superior de la pila y su cambio de valor cuando se realizan acciones de inserción/extracción en el segmento. Los valores se pasan a la pila cuando una función stack frame es llamada. El stack frame almacena el valor de las variables temporales de la función y algunas variables automáticas que almacenan información adicional como la dirección de retorno y los detalles del entorno de la persona que llama (registros de memoria). Cada vez que la función se llama a sí misma recursivamente, se crea un nuevo stack frame, lo que permite que un conjunto de variables de un stack frame no interfiera con otras variables de una instancia diferente de la función. Así es como funcionan las funciones recursivas.

```c
#incLude <stdio.h>

void func() {
     variables locales almacenadas en el stack
     when the function call is made
    int a, b;
}

int main() {
     variables locales almacenadas en el stack
    int LocaL = 5; char name[26]; func();
     ¨
    return 0;
}
```

Aquí, todas las variables se almacenan en la pila, ya que se declaran dentro del alcance de su función principal. Estas variables solo ocupan el espacio en la memoria hasta que se ejecuta su función. Por ejemplo, en el código anterior, el primer main() comienza su ejecución y se crea un stack frame para main() y se inserta en la pila del programa con datos de las variables local y nombre. Luego, en main, llamamos a func,

creándose otro stack frame y se apila por separado, conteniendo los datos de las variables a y b. Después de la ejecución de func, su stack frame se desapila y su variable no se asigna, cuando finaliza el programa, el stack frame principal también se habrá desapilado.

- Memoria con reserva dinámica (Heap)

El Heap (montón) se usa para la memoria que se asigna durante el tiempo de ejecución (memoria asignada dinámicamente). El Heap generalmente comienza al final del segmento bss y crece y se reduce en la dirección opuesta a la pila. Los comandos como malloc, calloc, free, realloc, etc. se usan para administrar las asignaciones en el segmento del montón, pero esto es un tema que se tratará más adelante

.

2. Diferencia entre dato y dirección

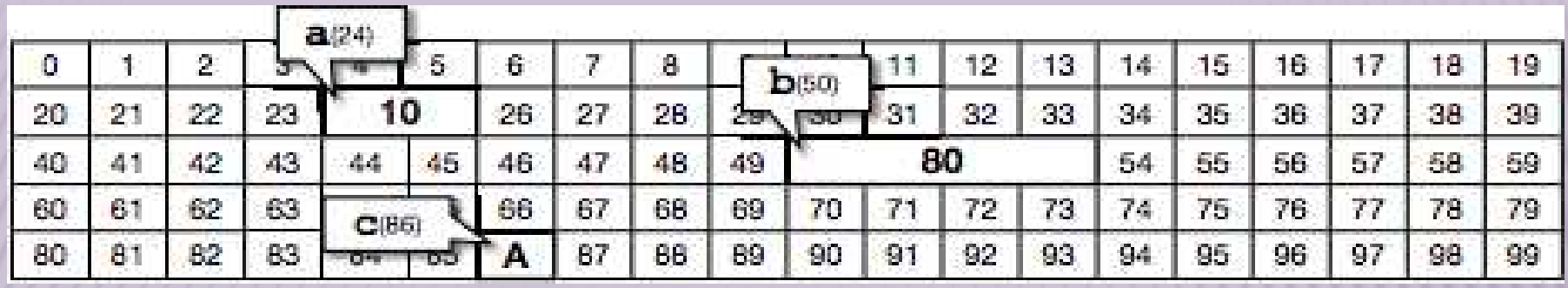
Anteriormente se planteó la estructura y disposición general de la memoria, ahora veamos cómo se distribuye la misma de manera interna para poder finalmente diferenciar entre un dato ocupado en memoria y su dirección.

Como se mencionó al principio, la memoria de la computadora se compone de un conjunto de bytes numerados consecutivamente. Luego, podemos almacenar datos en celdas de memoria formadas por grupos de 1 o más bytes según sea la longitud del dato que vayamos a almacenar. Teniendo el siguiente programa:

```
#incLude <stdio.h>

int main() {
    int a = 10; 2 bytes long b = 80;                    
    4 bytes char c = 'A'            1 byte return 0;
}
```

El programa al comenzar a ejecutarse reservará una cierta cantidad de bytes de memoria que dependerá de la longitud de sus tipos de datos. De esta forma, la distribución de la memoria de la computadora podría llegar a ser la siguiente:



Donde cada celda representa 1 byte de memoria cuya dirección está dada por un número secuencial. Según el ejemplo, la dirección de la variable a es 24 y ocupa una celda compuesta por 2 bytes, la dirección de la variable b es 50 y utiliza 4 bytes. La variable c requiere un único byte y se ubica en la dirección de memoria número 86.

Al anteponer el operador & "ampersand" al identificador de una variable obtenemos su dirección de memoria. Por lo tanto el valor de la variable `a` es 10 y el valor de `&a` es su dirección de memoria que, en este caso, es 24.

Finalmente, un dato puede ser información almacenada en una variable, en el caso del ejemplo para las variables a, b y c, este puede ser un carácter o un número que se encuentra en la memoria y cada tipo ocupa un número de casillas distinto, en cambio la dirección permite acceder a la ubicación de cada celda y es un valor en sí mismo (expresado a menudo en código hexadecimal), por tanto, también puede almacenarse en otras celdas de la memoria. Teniendo esto en cuenta ambas se diferencian en que cada celda de memoria tiene una única dirección que indica su posición relativa en la memoria mientras que los datos son los que se almacenan en las celdas de memoria, es decir, constituyen el contenido de dicha celda. La dirección es única y permanente, el contenido, por otro lado, puede cambiar mientras se ejecuta un programa.

3. Diferencia entre declaración estática y declaración dinámica

La declaración o asignación de memoria es un proceso mediante el cual los programas y servicios informáticos se asignan con espacio de memoria física o virtual. La asignación de memoria se realiza antes o en el momento de la ejecución del programa. Existen dos tipos de memoria en nuestra máquina, una es la memoria estática y otra es la memoria dinámica; ambas memorias son gestionadas por nuestro Sistema Operativo.

Nuestro sistema operativo nos ayuda en la asignación y desasignación (allocation, deallocation) de bloques de memoria durante el tiempo de compilación o durante el tiempo de

ejecución de nuestro programa. Cuando la memoria se asigna durante el tiempo de compilación, se almacena en la memoria estática y se conoce como asignación de memoria estática, y cuando la memoria se asigna durante el tiempo de ejecución, se almacena en la memoria dinámica y se conoce como asignación de memoria dinámica.

- Declaración de memoria estática

En la asignación de memoria estática, la memoria de los datos se asigna cuando se inicia el programa. El tamaño se fija cuando se crea el programa. Se aplica a variables globales, variables de ámbito de archivo y variables calificadas con funciones internas estáticas definidas. Esta asignación de memoria es fija y no se puede cambiar, es decir, aumentar o disminuir después de la asignación. Por lo tanto, los requisitos de memoria exactos deben conocerse de antemano. Ejemplo:

```
#incLude <stdio.h> main (){
   int a[5] = {10,20,30,40,50};
   int i;
   printf ("Los eLementos deL array son:");
   for ( i=0; i<5; i +) printf ("%d",
      a[i]);
}
SaLida:
Los eLementos deL array son: 10, 20, 30, 40, 50
```

- Declaración de memoria dinámica

A veces, el espacio constante asignado en tiempo de compilación puede ser insuficiente y, para aumentar el espacio durante el tiempo de ejecución, llegamos al concepto de asignación de memoria dinámica. La asignación y desasignación de memoria en tiempo de ejecución en C se realizan usando memoria dinámica. Es un método en el que usamos diferentes funciones de biblioteca como malloc(), calloc(), realloc() y free() para asignar y desasignar un bloque de memoria durante el tiempo de ejecución.

Se considera un tema muy importante porque casi todas las estructuras de datos (como listas enlazadas, pilas, colas, árboles, etc.) están asociadas con el concepto de asignación de memoria dinámica.

```
 ﷺPrograma para calcular la suma de n números ingresados por el usuario
#incLude <stdio.h> #incLude <stdLib.h> int
main() {

  int n, i, *ptr, sum = 0;
  printf("Ingresa eL numero de eLementos: "); scanf("%d", &n);
  ptr = (int*) maLLoc(n * sizeof(int));

   ﷺSi la memoria no puede ser asignada
  if(ptr  ﷺNULL) { printf("Error."); exit(0);
  }

  printf("Ingresa Los eLementos: ");
  for(i = 0; i < n;  +i) {
    scanf("%d", ptr + i); sum += *(ptr + i);
  }

  printf("Suma = %d", sum);

   ﷺdesasignar la memoria
  free(ptr);
  return 0;
}

SaLida:
Ingresa eL numero de eLementos: 3 Ingresa Los eLementos:
100
20
36
Suma = 156
```

Ya que conocemos las definiciones de cada una, podemos diferenciarlas.

Declaración de memoria estática	Declaración de memoria dinámica
La memoria constante (invariable) se reserva en tiempo de compilación de nuestro programa y no se puede modificar.	La memoria dinámica (variable) se reserva en tiempo de ejecución de nuestro programa y se puede modificar.
Se utiliza en el momento de la compilación de nuestro programa y también se conoce como asignación de memoria en tiempo de compilación.	Se utiliza en el tiempo de ejecución de nuestro programa y también se conoce como asignación de memoria en tiempo de ejecución.
No podemos asignar o desasignar un bloque de memoria durante el tiempo de ejecución.	Podemos asignar y desasignar un bloque de memoria durante el tiempo de ejecución.
El espacio de pila se utiliza en la asignación de memoria estática.	El espacio de almacenamiento dinámico se utiliza en la asignación de memoria dinámica.
No proporciona reutilización de la memoria mientras se ejecuta el programa. Por lo tanto, es menos eficiente.	Proporciona reutilización de la memoria mientras se ejecuta el programa. Por lo tanto, es más eficiente.

4. Diferencia entre declaración continua (o lineal) y no continua (o no lineal)

- La asignación de memoria contigua

La asignación de memoria contigua es un tipo de técnica de asignación de memoria en la que a los procesos se les asigna un bloque continuo de espacio en la memoria. Este bloque puede ser de tamaño fijo para todos los procesos en un esquema de partición de tamaño fijo o puede ser de tamaño variable dependiendo de los requerimientos del proceso en un esquema de partición de tamaño variable. Como su nombre lo indica, asignamos bloques contiguos de memoria a cada proceso usando esta técnica. Entonces, cada vez que un proceso quiere ingresar a la memoria principal, asignamos un segmento continuo desde el espacio totalmente vacío al proceso en función de su tamaño.

- Asignación no contigua

En la asignación no contigua, el sistema operativo necesita mantener la tabla que se llama Tabla de página para cada proceso que contiene la dirección base de cada bloque que adquiere el proceso en el espacio de memoria. En la asignación de memoria no contigua, diferentes partes de un proceso se asignan a diferentes lugares en la memoria principal. Se permite la expansión, lo que no es posible en otras técnicas, como la asignación de memoria dinámica o estática contigua. Es por eso que se necesita la paginación para garantizar una asignación de memoria efectiva. La paginación se realiza para eliminar la fragmentación externa.

En la asignación de memoria no contigua, el espacio de memoria libre disponible está disperso y todo el espacio de memoria libre no está en un solo lugar. Así que esto lleva mucho tiempo. En la asignación de memoria no contigua, un proceso adquirirá el espacio de memoria, pero no en un lugar, sino en diferentes ubicaciones según los requisitos del proceso. Esta técnica de asignación de memoria no contigua reduce el desperdicio de memoria que conduce a la fragmentación interna y externa. Esto utiliza todo el espacio de memoria libre que se crea mediante un proceso diferente.

Declaración de memoria contigua	Declaración de memoria no contigua
La asignación de memoria contigua asigna bloques consecutivos de memoria a un archivo/proceso.	La asignación de memoria no contigua asigna bloques separados de memoria a un archivo/proceso.
Más rápido en ejecución.	Más lento en ejecución.
Tanto la fragmentación interna como la fragmentación externa se producen.	Solo se produce fragmentación externa.
Incluye la asignación de una sola partición y la asignación de varias particiones.	Incluye paginación y segmentación.
Existen dos tipos: 1. Partición fija (o estática) 2. Partición dinámica	Es de cinco tipos: 1. Paginación 2. Paginación multinivel 3. Paginación invertida 4. Segmentación 5. Paginación segmentada
Podría ser visualizado e implementado usando Arrays.	Podría implementarse mediante listas enlazadas.

5. Apuntadores o punteros

Los apuntadores son una característica fundamental de la programación en C y otros lenguajes de programación. Son variables que almacenan direcciones de memoria de otras variables.

- Definición: Un apuntador es una variable que almacena una dirección de memoria. La dirección puede ser de cualquier tipo de datos, como enteros, flotantes, caracteres, etc.
- Declaración: Los apuntadores se declaran con el operador de dirección de memoria "&" y el operador de apuntador "*". Por ejemplo, si quieres declarar un apuntador a una variable entera, se haría de la siguiente manera:

```
int *ptr;
```

- Uso: Para usar un apuntador, primero debes asignarle una dirección de memoria de una variable existente, utilizando el operador de dirección de memoria "&". Por ejemplo:

```
int x = 10;
int *ptr;
```

```
ptr = &x;
```

- Manejo: Una vez que tienes un apuntador, puedes usarlo para acceder y manipular el valor de la variable a la que apunta. Para acceder al valor, debes de referenciar el apuntador con el operador de apuntador "*". Por ejemplo:

```
#incLude <stdio.h>

int main() {
   int x = 10; int *ptr; ptr = &x;
   *ptr = 20;
   printf("EL vaLor de x es: %d", x);
   return 0;
}
```

Hay algunos otros conceptos importantes adicionales que debes conocer sobre los apuntadores:

- Tamaño de un apuntador: El tamaño de un apuntador depende de la arquitectura de la computadora en la que se está ejecutando el programa. Por ejemplo, en una computadora de 32 bits, el tamaño de un apuntador a cualquier tipo de datos es de 4 bytes, mientras que, en una de 64 bits, el tamaño es de 8 bytes.

- Null pointer: Un apuntador "null" es un apuntador que no apunta a ninguna dirección de memoria válida. Se puede inicializar un apuntador a null con la constante "NULL". Es importante tener en cuenta que un apuntador null no se puede usar para acceder a una variable, ya que eso resultaría en una violación de segmento.
- Apuntadores a apuntadores: Es posible tener apuntadores que apunten a otros apuntadores. Estos se conocen como "apuntadores a apuntadores". Por ejemplo, si quieres declarar un apuntador a un apuntador a una variable entera, se haría de la siguiente manera:

```
int **ptr;
```

- Arrays y apuntadores: En C, los arrays y los apuntadores están estrechamente relacionados. De hecho, cuando se pasa un array a una función, se pasa realmente un apuntador al primer elemento del array. Además, el nombre de un array es en realidad un apuntador a su primer elemento.
- Funciones y apuntadores: Es posible usar apuntadores como argumentos de funciones y también se pueden devolver apuntadores desde funciones. Esto es útil cuando se quiere modificar variables en una función o cuando se quiere devolver un puntero a un bloque de memoria dinámica.
- Alias de apuntadores: Es posible que varios apuntadores apunten a la misma dirección de memoria. Estos se conocen como "alias de apuntadores". Es importante tener en cuenta que esto puede causar efectos secundarios no deseados si se manipulan los valores a través de diferentes apuntadores.

En resumen, los apuntadores son una herramienta versátil y poderosa en la programación en C, pero también son potencialmente peligrosos si no se comprenden y se usan adecuadamente. Por lo tanto, es importante estudiar y practicar su uso antes de incorporarlos en tus programas.

Otros puntos a tomar en cuenta son:

6. Operaciones de asignación y liberación dinámica de la memoria

Las operaciones de asignación y liberación dinámica de la memoria son un aspecto importante de la programación en C y C++. La asignación dinámica de memoria se refiere a la asignación de memoria en tiempo de ejecución durante la ejecución del programa. En este caso, el programador es responsable de liberar la memoria una vez que ya no se necesita.

Las operaciones de asignación dinámica de memoria se realizan mediante la función malloc() en C y new en C++. La función malloc() se utiliza para asignar un bloque de memoria de un tamaño específico en bytes, y devuelve un puntero a la primera dirección de memoria asignada. Por otro lado, el operador new en C++ se utiliza para asignar memoria para un objeto de un tipo determinado.

La liberación dinámica de memoria se realiza mediante la función free() en C y el operador delete en C++. La función free() se utiliza para liberar la memoria previamente asignada por malloc(), mientras que el operador delete se utiliza para liberar la memoria asignada por el operador new.

- Asignación dinámica de memoria: La asignación dinámica de memoria permite a los programadores asignar memoria en tiempo de ejecución para un tamaño específico en bytes. Esto es útil cuando se necesita memoria adicional para almacenar datos mientras se ejecuta el programa, y el tamaño exacto de los datos no se conoce en tiempo de compilación.

En C, la función malloc() se utiliza para asignar memoria dinámicamente. Por ejemplo, el siguiente código asigna memoria para almacenar un entero:

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

En C++, el operador new se utiliza para asignar memoria dinámicamente. Por ejemplo, el siguiente código asigna memoria para un objeto de tipo int:

```
int *p = new int;
```

- Liberación dinámica de memoria: Una vez que se ha asignado memoria dinámicamente, es importante liberarla una vez que ya no se necesita. Si no se libera la memoria, puede ocurrir una fuga de memoria, lo que puede resultar en una pérdida de recursos y errores en el programa.

En C, la función free() se utiliza para liberar la memoria dinámicamente asignada. Por ejemplo, el siguiente código libera la memoria asignada para un entero:

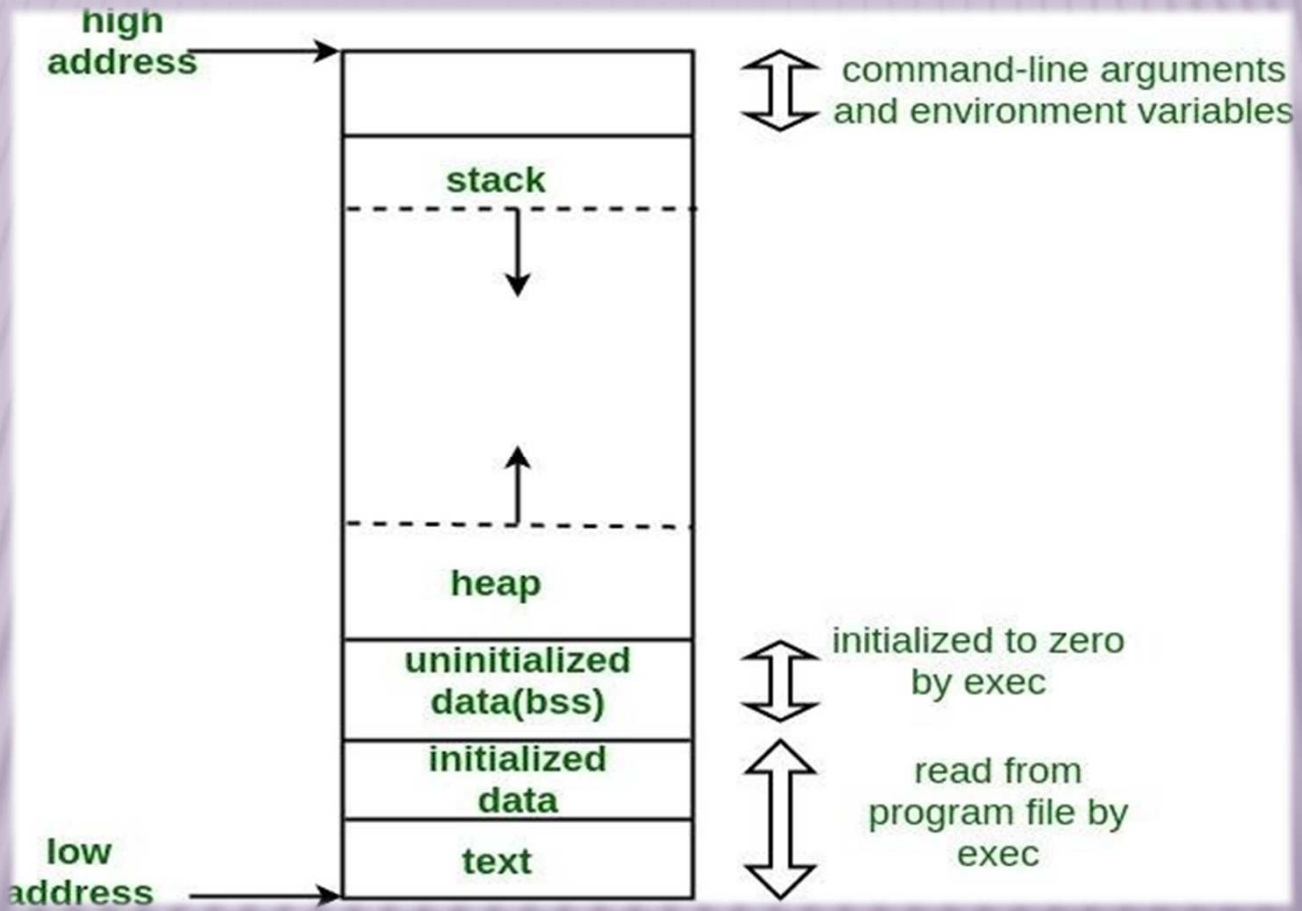
```
int p;
p = (int *) malloc(sizeof(int));
. . .
free(p);
```

En C++, el operador delete se utiliza para liberar la memoria dinámicamente asignada. Por ejemplo, el siguiente código libera la memoria asignada para un objeto de tipo int:

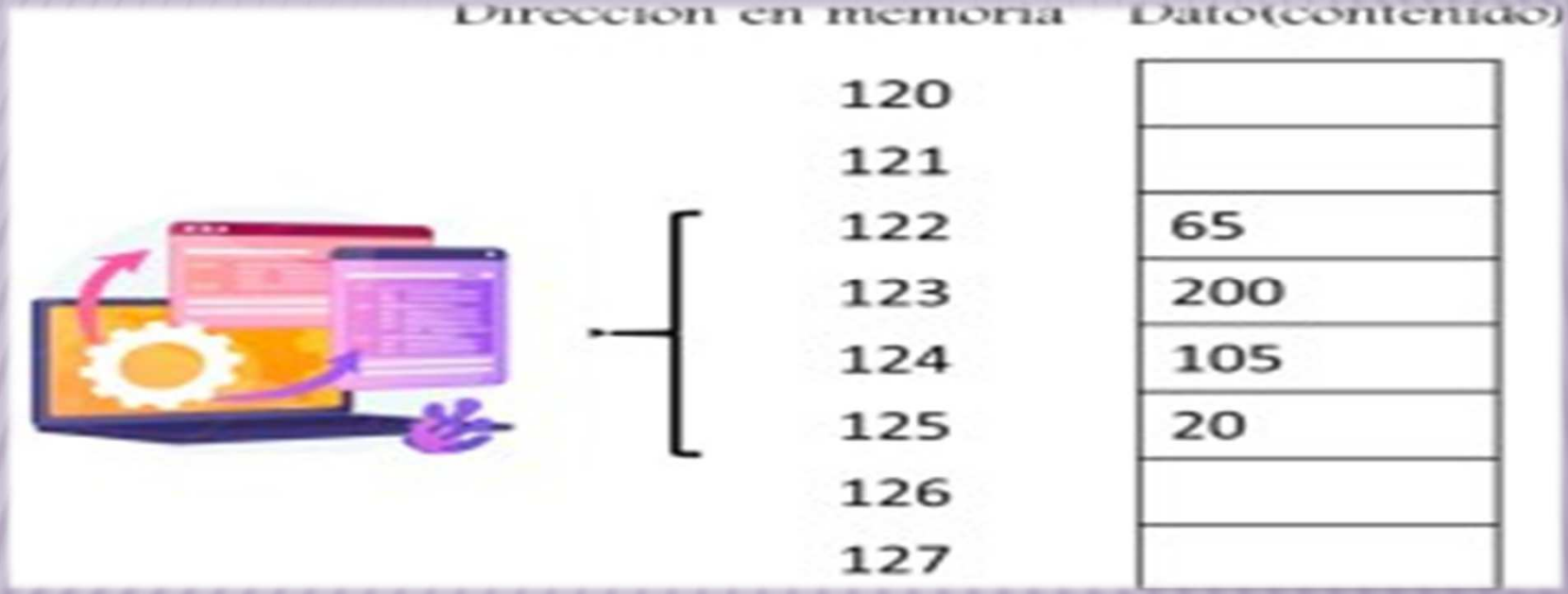
```
int *p = new int;
. . .
delete p;
```

Es importante tener en cuenta que la liberación dinámica de memoria debe realizarse con cuidado y siguiendo buenas prácticas de programación. Por ejemplo, es importante liberar solo la memoria que se ha asignado previamente y no liberar la misma memoria más de una vez. También es importante asegurarse de que los punteros que apuntan a la memoria liberada ya no se utilizan antes de liberar la memoria.

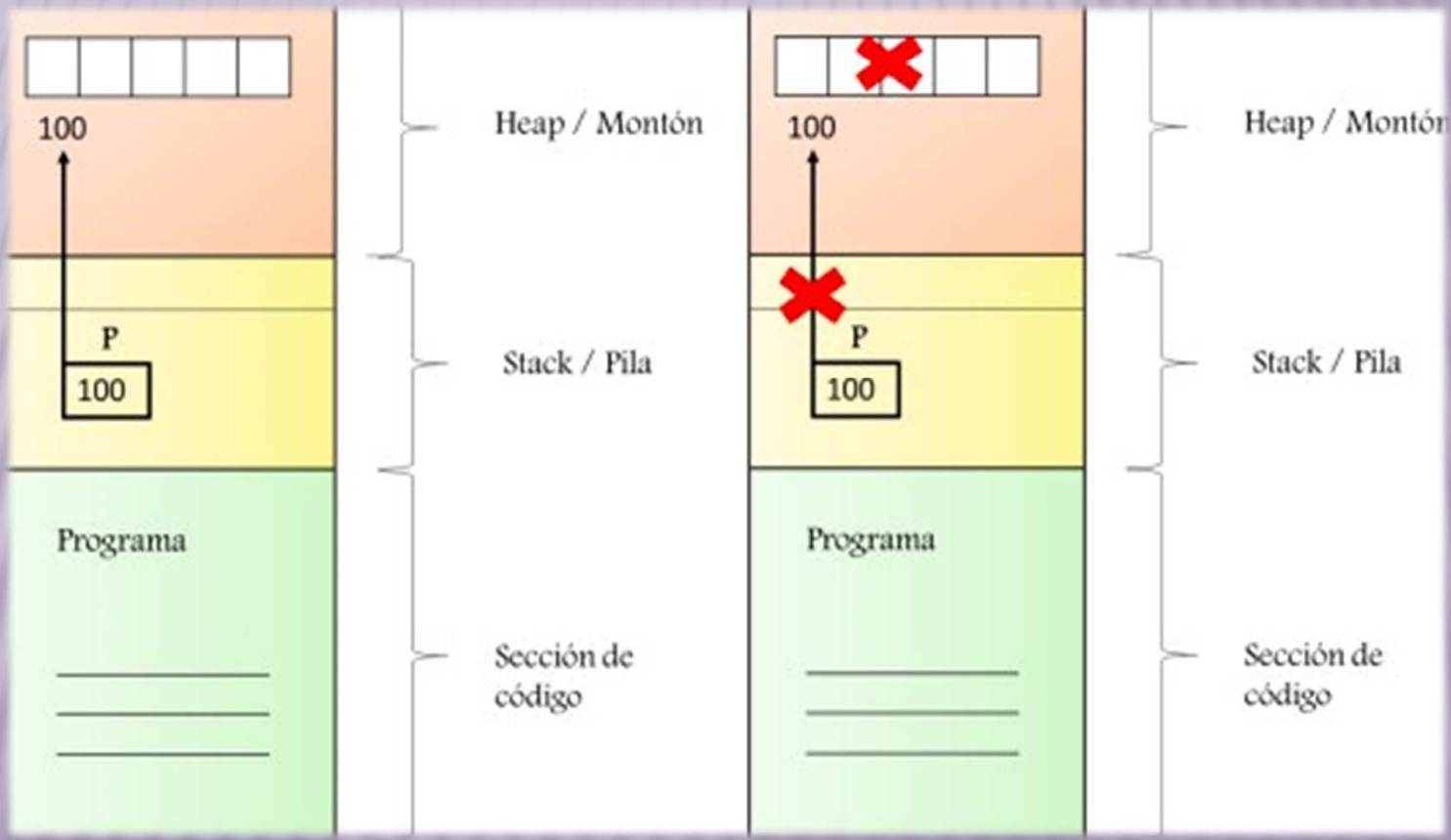
Estructura de la *memoria* en C



Dato vs Dirección



Asignación y liberación dinámica de la memoria



Representación gráfica de los apuntadores