Programación en C

Alocación de memoria

Alocación de memoria

Vimos que podemos usar pointers para manipular memoria de la manera que deseemos. También vimos cómo es que los distintos tipos de datos se guardan en memoria.

La pregunta es: ¿De donde sale esta memoria?

- ¿Cómo es que C nos da una porción de memoria para usar?
- ¿Cómo obtiene C esa memoria?
- ¿Dónde están en memoria el resto de los **procesos**?

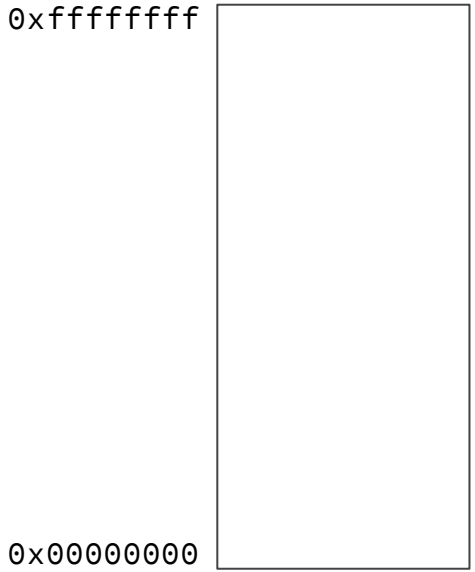
Proceso: instancia de un programa en ejecución

El espacio de direcciones

Cada proceso tiene acceso a un espacio de direcciones de memoria.

La cantidad máxima de memoria que podemos utilizar está limitada al tamaño de nuestro espacio de direcciones. Si no tenemos una dirección para usar, no tenemos donde guardar datos.

A la derecha tenemos un diagrama de un espacio de memoria que iremos completando con las distintas secciones que usa un proceso.



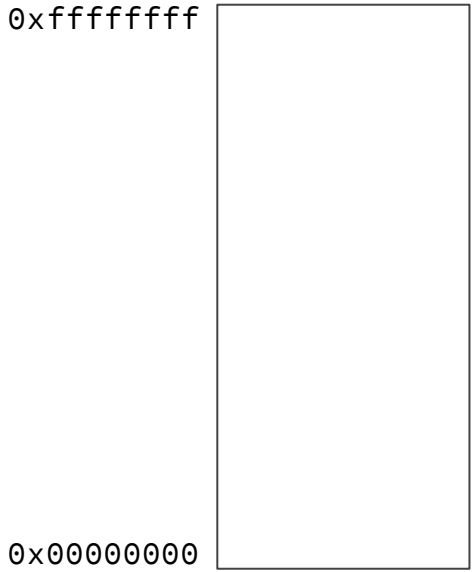
El espacio de direcciones

La arquitectura donde estamos trabajando determina el tamaño de nuestro espacio de direcciones.

Más específicamente, esto está dado por el tamaño de palabra del procesador. Un CPU solo puede referenciar memoria mediante una dirección que entre en una palabra.

Tamaño de palabra más común hoy en día: 64 bits.

El tamaño de los pointers está atado al tamaño de palabra.

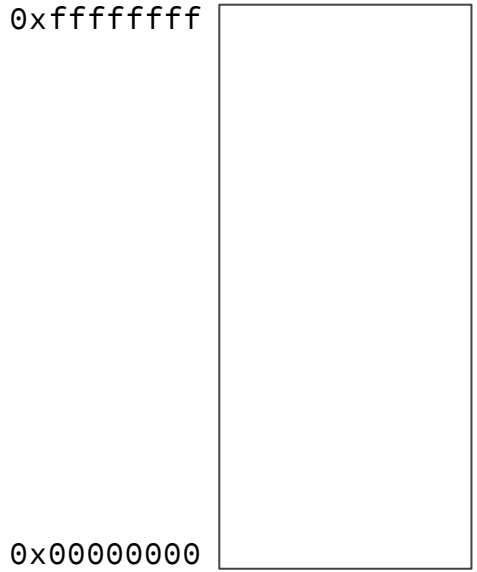


El espacio de direcciones

Un pointer de 32 bits puede tomar valores entre: `0x0` y `0xffffffff`. Esto le permite acceder a 4GB de memoria.

Un pointer de 64 bits entre: `0x0` y `0xffffffffffffffff`. Esto se traduce a 16 exabytes. 1 exabyte = 10^9 GB.

En el diagrama asumimos 32 bits para tener una notación más compacta.

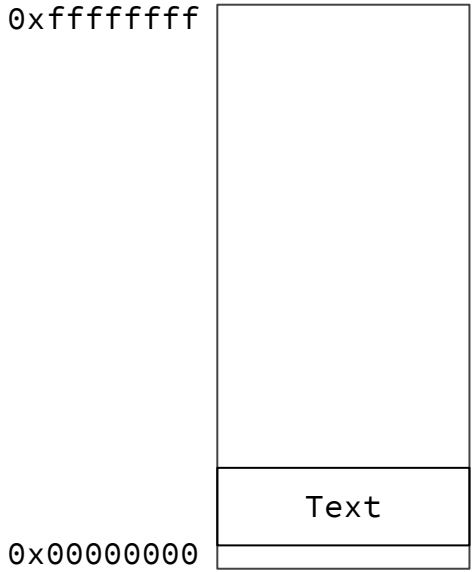


Código

Primero que nada, el proceso necesita tener acceso al programa que va a ejecutar.

El código del programa (en binario, ya compilado), junto con todas las librerías que use, se aloca en la sección llamada **Text**.

El resto de la memoria se usa para datos. Dependiendo cómo se aloquen en el programa estos datos, van a terminar en distintas secciones.



Tiempo de vida (lifetime)

C99: “El tiempo de vida de un objeto es el período de tiempo en la ejecución del programa durante el cual se garantiza que va a tener almacenamiento.

Un objeto existe, tiene una dirección constante y mantiene su último valor guardado durante su tiempo de vida.

Si se refiere a un objeto fuera de su tiempo de vida, el comportamiento es indefinido”.

El tiempo de vida de un objeto está determinado por la duración de su almacenamiento (storage duration).

C99 6.2.4.1/2

Duración de almacenamiento

El estándar define tres duraciones de almacenamiento:

- Estática
- Automática
- Alocada

Para evitar confusiones nos referiremos a la duración “alocada” como “dinámica”.

Este es el nombre que usa el estándar de C++ y es más claro. Al fin y al cabo, todo almacenamiento debe ser **alocado** en memoria.

De las 3 duraciones surgen 3 tipos de alocaciones.

Alocación estática

Alocación estática

C99: “El tiempo de vida del objeto es la ejecución entera del programa. Su valor es inicializado solo una vez, previo al inicio del programa”.

1. El objeto siempre existe
2. El objeto siempre va a ser inicializado.
3. Todo esto se garantiza antes de que corra la función **`main`**.

¿Qué objetos son alocados estáticamente?

- Variables declaradas fuera de una función.
- Variables dentro de una función declaradas como **`static`**.

C99 6.2.4.3

Alocación estática: inicialización

C99: “Un objeto con alocación estática sólo puede ser inicializado con una constante.

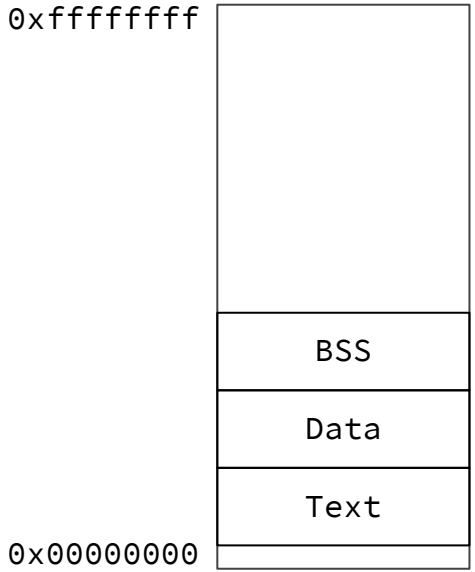
Si no es inicializado explícitamente:

- Si es un pointer, es inicializado a NULL.
- Si es un tipo aritmético, es inicializado a 0.
- Si es un array/struct, todos los miembros son inicializados siguiendo estas reglas.
- Si es una union, el primer miembro es inicializado siguiendo estas reglas.”

Alocación estática

Los objetos alocados estáticamente tienen sus propias secciones de memoria. Estas son de tamaño fijo ya que el espacio que ocupan es conocido en tiempo de compilación.

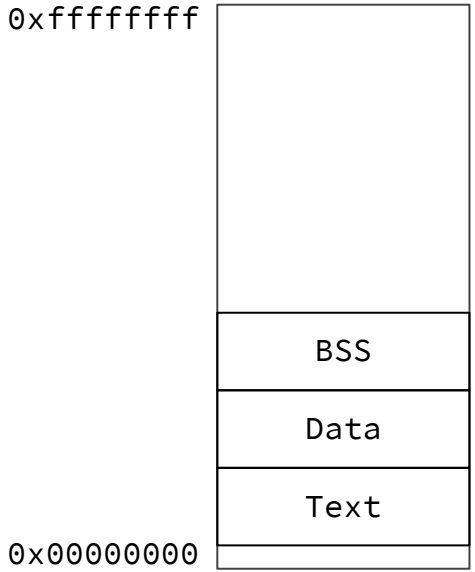
- BSS: para las variables no inicializadas.
- Data: para las variables inicializadas.



Alocación estática

Cuando el proceso arranca la sección Data es copiada tal cual se encuentra en el ejecutable directo a memoria.

Cómo el BSS por definición está compuesto puramente de ceros, es generado en el momento con el tamaño especificado en el ejecutable.



Ex 2.1, 2.2, 2.3, 2.4

Alocación estática

Cómo la estructura de las dos secciones de memoria están fijas en el ejecutable, la memoria en esas secciones va a ser idéntica para todos los procesos que corran el mismo programa.

Por lo tanto, todos los objetos que están alocados estáticamente **siempre** van a estar en las mismas posiciones **relativas** entre sí.

La posición **absoluta** depende de varios factores y puede cambiar entre diferentes ejecuciones de un mismo programa, pero es algo que no vamos a cubrir.

Temas relacionados: Position Independent Code, ASLR.

Alocación estática

Si yo declaro dos variables:

```
char a;

int b;
```

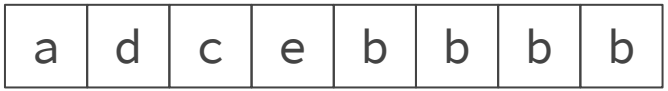
Y el compilador decide que a se aloca antes que b.

&a < &b en **todos** los procesos.

Alocación estática

A diferencia de un struct donde el orden de declaración de las variables importa, el compilador puede ordenar las variables para minimizar padding o hacer otras optimizaciones.

```
char a;
int b;
char c;
char d;
char e;
```



Ex 4.1

Problema: múltiples procesos

¿Qué pasa si corremos dos procesos a partir del mismo programa de manera simultánea?

Sabemos que las secciones donde se encuentran los objetos alocados estáticamente van a ser idénticas y posiblemente estén en las mismas direcciones.

¿Qué pasa cuando ambos procesos acceden a la misma variable y por lo tanto a la misma **dirección de memoria**?

Ex 5.1, 5.2

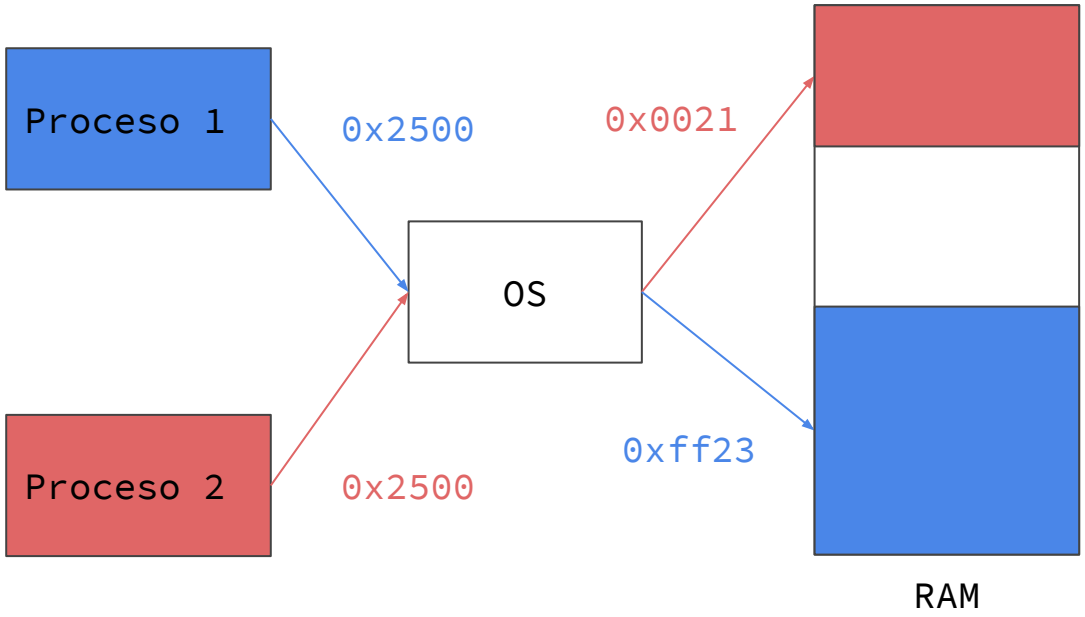
Memoria virtual

Una de las funcionalidades de un Sistema Operativo es la memoria virtual.

Cuando un proceso inicia, el SO le brinda un espacio de direcciones completamente privado, de manera que ningún proceso puede interactuar con el espacio de memoria de otro.

Cada proceso puede utilizar esta memoria sin preocuparse por el resto de las cosas corriendo en el sistema. El SO se encarga de traducir las direcciones virtuales de cada proceso a direcciones de memoria física.

Memoria virtual



Memoria virtual

Sin memoria virtual, trabajar en un lenguaje como C con acceso a pointers es propenso a errores y además muy inseguro.

Podríamos:

- Leer memoria de otro proceso. Ej: contraseñas guardadas en el navegador.
- Crashear otro proceso, incluido el sistema operativo entero.
- Sobrepasar cualquier tipo de “lock screen”.
- Y muchas cosas más...

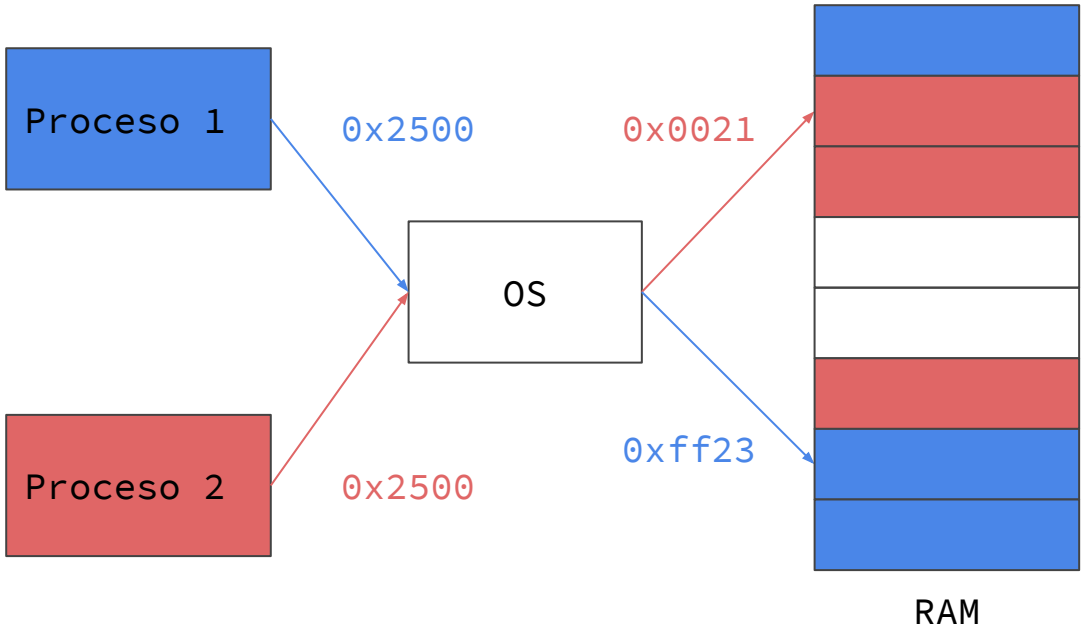
Alocación de memoria virtual

El SO no le da todo el espacio de memoria a todos los procesos. Cuando un proceso necesita memoria le debe pedir al SO.

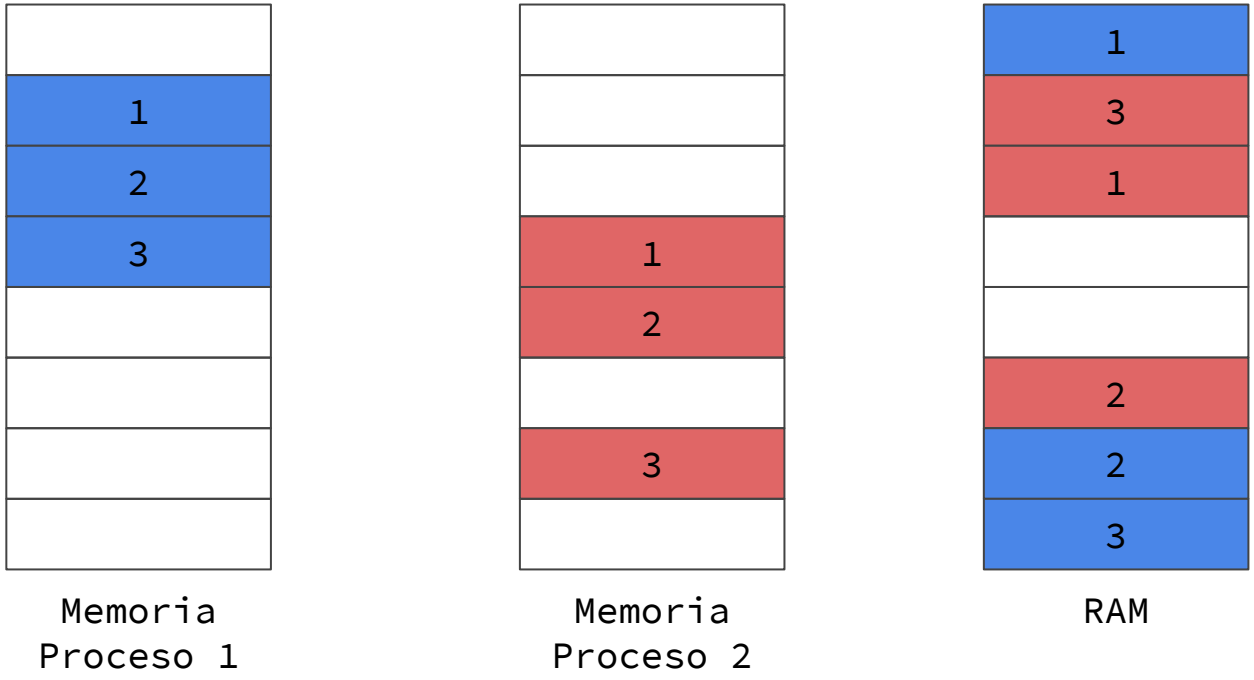
El SO asigna memoria en páginas. Cada página tiene un tamaño fijo, el más común es 4kb (Linux, Windows, MacOS). Del lado del proceso una página que le fue asignada por el SO se dice que está **mapeada**.

Cuando el proceso arranca, va a solicitar suficiente memoria como para alocar todas las variables estáticas más el código del programa.

Memoria virtual



Memoria virtual



Página en blanco == No mapeada

Permisos

Cada página puede tener distintos tipos de permisos de acceso, dependiendo lo que requiera el proceso.

Los permisos son: read, write y execute.

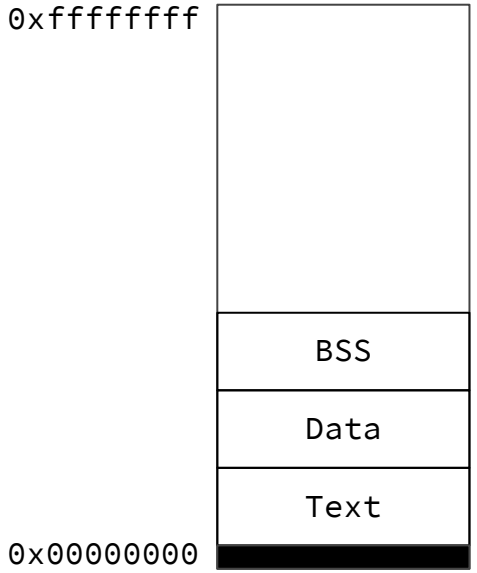
Segmentation Fault es una señal generada por el SO. Ocurre cuando el proceso intenta realizar una operación en una página que no tiene los suficientes permisos. Esto incluye cualquier tipo de operación realizada en una página no mapeada.

NULL

En la **mayoría de los casos**, el compilador va a elegir la dirección 0 como el null pointer.

Para facilitar la detección de la desreferenciación de un null pointer, la primera página de memoria nunca es mapeada por el proceso, garantizando un Segmentation Fault.

Esto sigue siendo **undefined behavior** y puede no ocurrir en ciertos casos, particularmente en sistemas embebidos, donde la memoria es escasa.

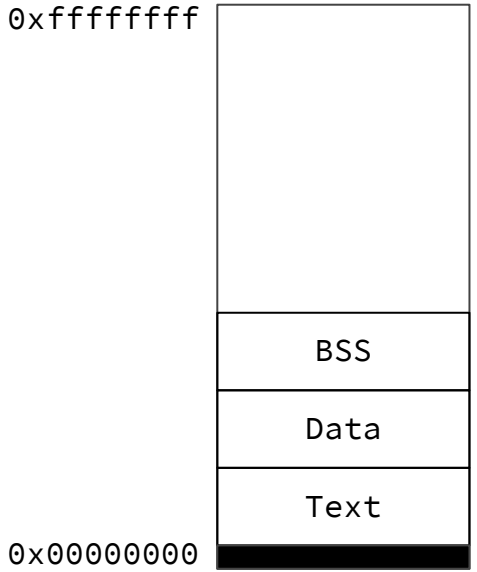


Secciones != páginas

Las distintas secciones de memoria que crea el proceso son puramente lógicas y pueden ocupar más de una página.

Las secciones BSS y Data también pueden encontrarse en la misma página ya que tienen los mismos permisos.

Sin embargo, Text siempre debe estar en páginas separadas de BSS y Data ya que requiere **execute**.

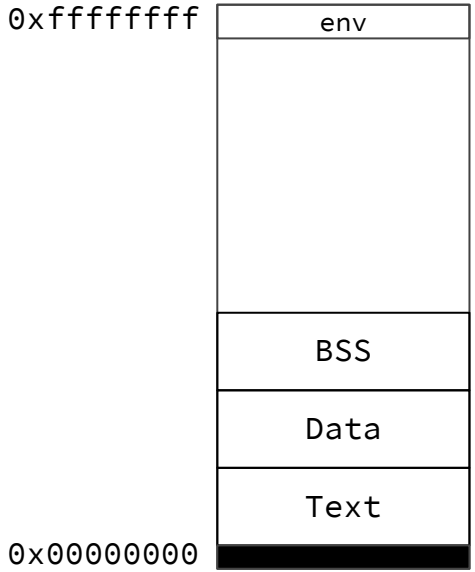


Entorno

En las direcciones de memoria más altas se aloca espacio para el entorno del programa.

Esto incluye:

- Variables de entorno
- Argumentos de la línea de comandos



Alocación automática

Alocación automática

C99: “El tiempo de vida del objeto va desde la entrada al bloque con el que está asociado hasta que la ejecución de ese bloque termine de cualquier manera (Ingresar en un sub-bloque o llamar una función suspende, pero no termina, la ejecución del bloque actual).

Si se ingresa al bloque recursivamente, una nueva instancia del objeto es creado cada vez.

El valor inicial del objeto es indefinido, a menos que sea inicializado.”

Bloque: todo el código que esté encerrado entre `{ }`.

C99 6.2.4.4/5/6

Alocación automática

¿Qué objetos son alocados estáticamente?

- Toda variable declarada dentro de un bloque (función, loop, etc).
- Siempre y cuando no sean declaradas **`static`** (o **`extern`**).

No son inicializados automáticamente

- Leer su valor antes que sea inicializado es undefined behavior.

Solo existen **dentro el bloque**

Ex 6.1, 6.2, 6.3, 6.4, 6.5

Alocación automática

En alocación estática, sabemos en tiempo de compilación exactamente cuántas variables vamos a tener junto con su tamaño. En cambio, en alocación automática vamos a necesitar más o menos memoria dependiendo que pase en runtime.

Cada vez que entramos a un bloque nuevo necesitamos alocar memoria para sus variables.

- Llamar a una función
- Entrar en un loop
- Entrar en un if, o en un else

Alocación automática

En vez de una sección de memoria fija, las variables automáticas deben vivir en una sección que sea también variable en runtime.

Esta sección se crea al arrancar el programa y crece según es necesario para acomodar nuevas alocaciones. Si la sección pasa a sobrepasar el tamaño de la página, otra página debe ser solicitada al sistema operativo para que pueda continuar creciendo.

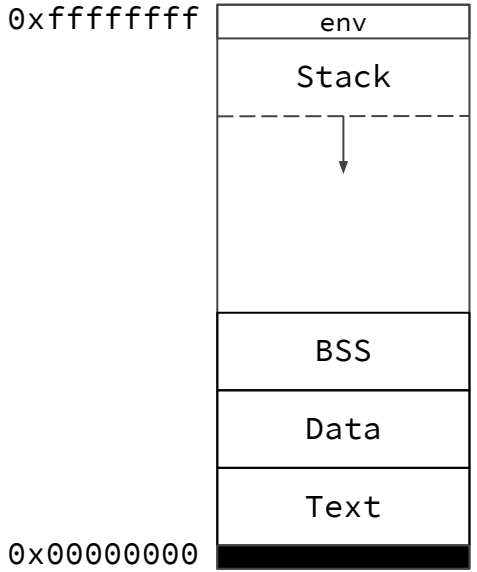
El Stack

Probablemente conocen StackOverflow, ya sea la página o la excepción de Java.

Estos hacen alusión a este stack, usado por la gran mayoría de los programas.

El propósito del stack es guardar valores temporales para facilitar la ejecución del programa:

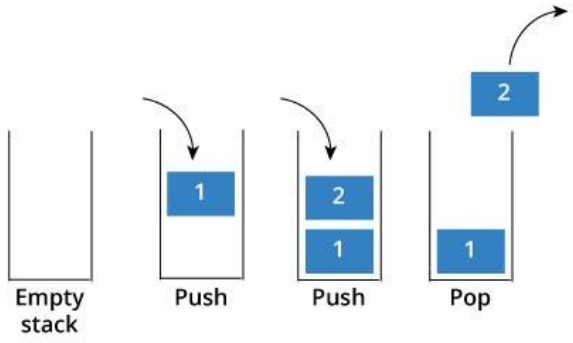
- Alocaciones automáticas
- Dónde volver cuando ejecutamos **`return`**



¿Qué es un stack?

Un stack es una estructura de datos que funciona como una colección de objetos, con dos operaciones:

- Push: agrega un objeto al stack
- Pop: remueve el último objeto agregado al stack

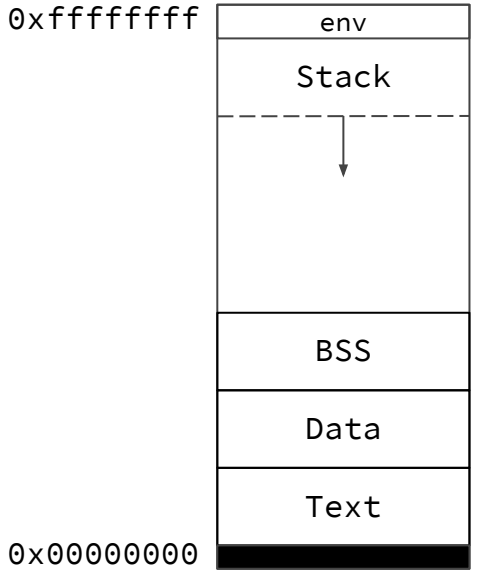


El Stack

“El Stack” hablando de la sección de memoria, implementa esta misma funcionalidad.

El inicio o “bottom” del stack se encuentra en las direcciones de memoria altas. Los elementos se agregan de manera que crece hacia las direcciones de memoria bajas.

Se podría decir que está “invertido”.

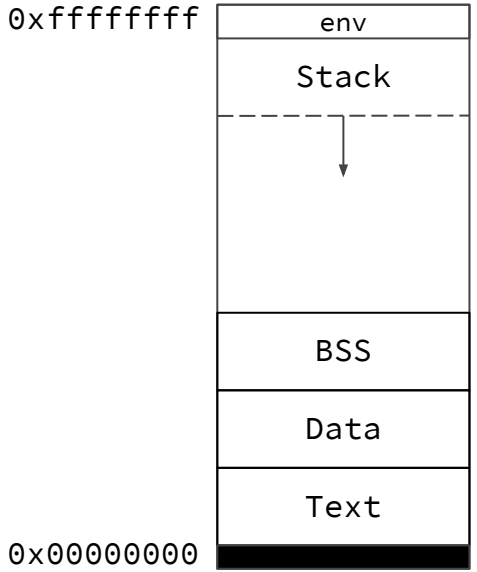


El Stack

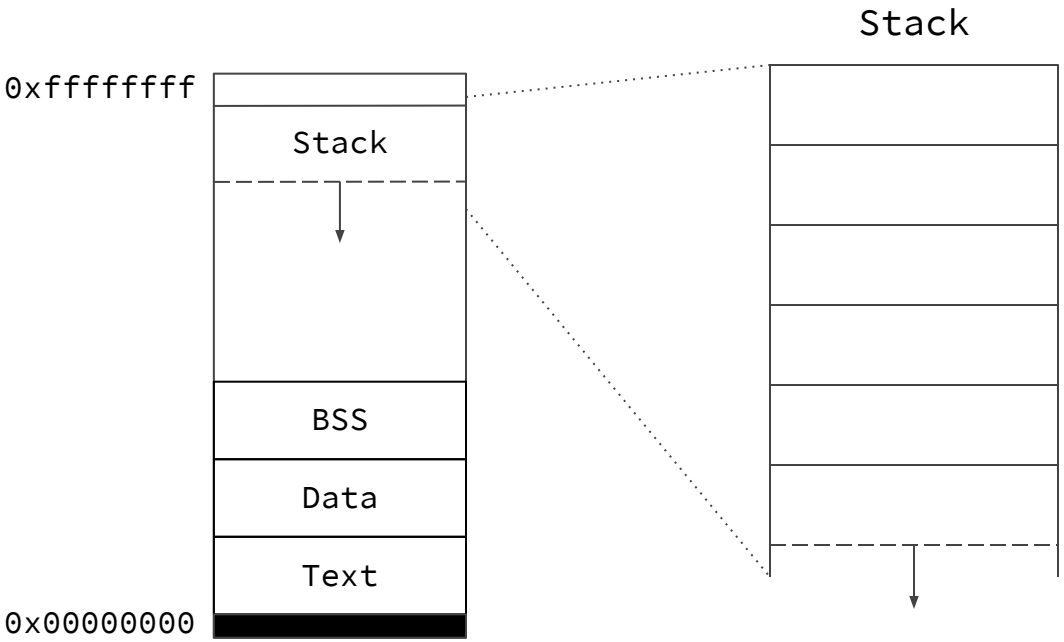
A diferencia de las otras secciones, el stack va a crecer durante la ejecución del programa.

Al iniciar el proceso se alocan una cierta cantidad de páginas. Conforme el stack crece el SO automáticamente le aloca nuevas páginas. Esto se repite hasta que se llega al tamaño máximo del stack.

Lo que en Java se notifica con un StackOverflow, en C probablemente se vea como un Segmentation Fault debido a utilizar una página no mapeada.



El Stack



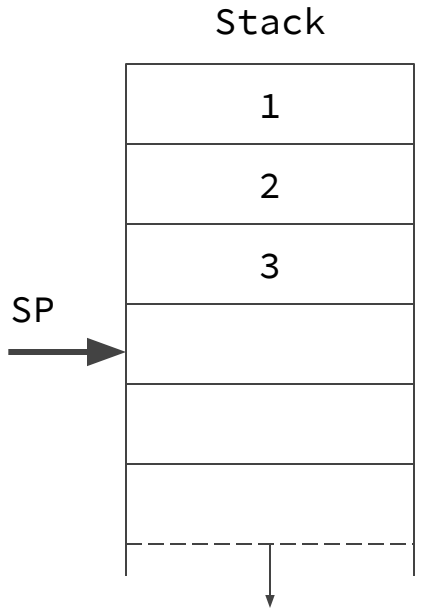
El Stack

Se representa como una sección de memoria contigua.

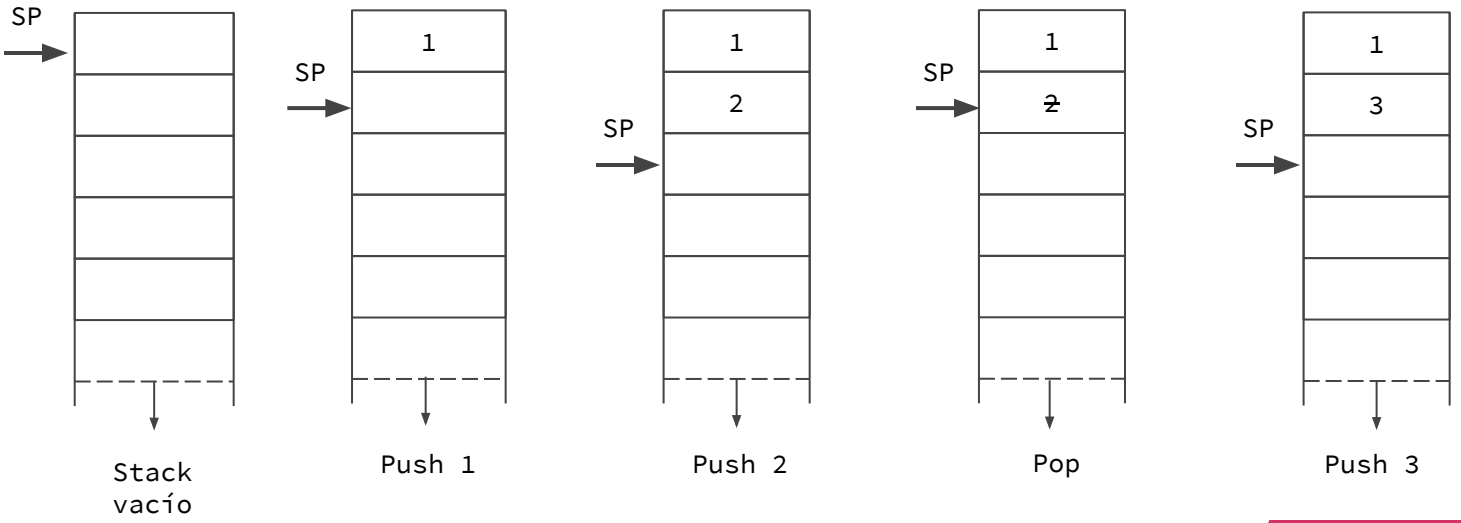
El bottom del stack es fijo, por lo que no necesitamos actualizar su posición. El top del stack, sin embargo, cambia dependiendo cuantos elementos tiene.

Stack Pointer (SP): pointer al top del stack

Conforme vamos agregando elementos, avanzamos el SP. Para remover un elemento, simplemente hacemos retroceder el SP.



El Stack



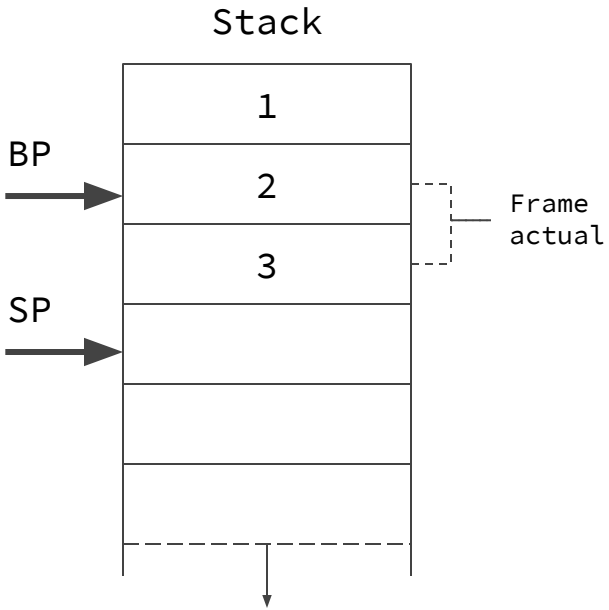
El SP apunta al primer lugar vacío.

Function frame

Un function frame es la parte del stack que pertenece a una función. Cada función que se llama crea un nuevo function frame.

Para marcar el function frame de la función que se está ejecutando en este momento, utilizamos otro pointer.

Base pointer (BP): marca el inicio del function frame actual.



Function frame

Las variables alocadas automáticamente son alocadas dentro del function frame de cada función. Cada vez que llamamos a una función se va a crear un nuevo frame con el espacio suficiente para guardar todas sus variables.

Esta alocación se hace simplemente avanzando el Stack Pointer la cantidad de bytes que sean necesarios. Esto se hace decrementando el valor del pointer. Si quiero alocar 8 bytes: SP = SP - 8 (pointer arithmetic como `char*`).

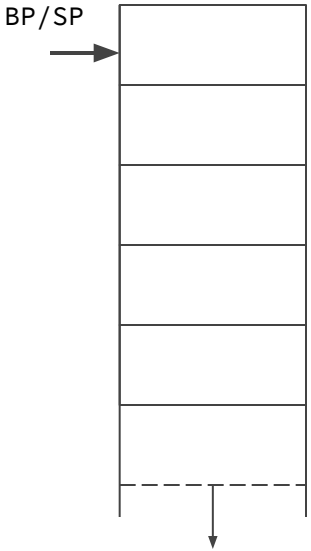
Para referenciar una variable, el compilador usa un offset desde el Base Pointer.

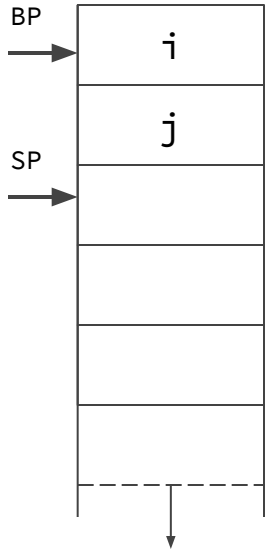
Function frame

```
void function() {

    int i;

    int j;

}
```

Para ser más acotados en la slide:

- Cada espacio en el stack son 4 bytes.
- Seguimos usando 32 bits. Los pointer ocupan 4 bytes.





El compilador asigna:

&i = BP

&j = BP - 4

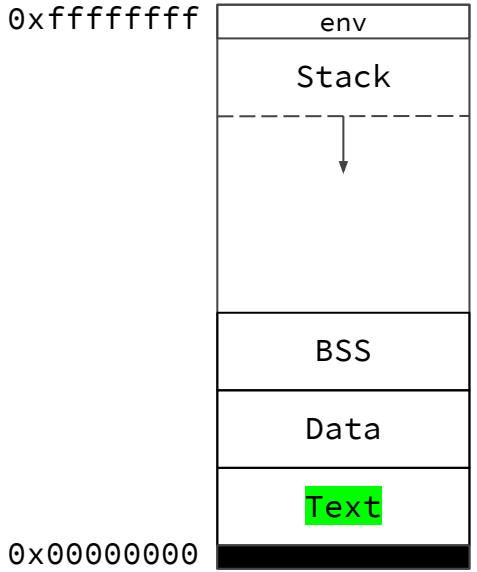
SP = SP - 8

Instruction pointer

Recordamos que la sección Text contiene el código que está ejecutando el programa.

Además del Stack Pointer y el Base Pointer para marcar el frame de función actual, también debemos mantener un pointer a la instrucción que estamos ejecutando. Por instrucción nos referimos a código binario, pero podemos verlo como la línea del source code ahora.

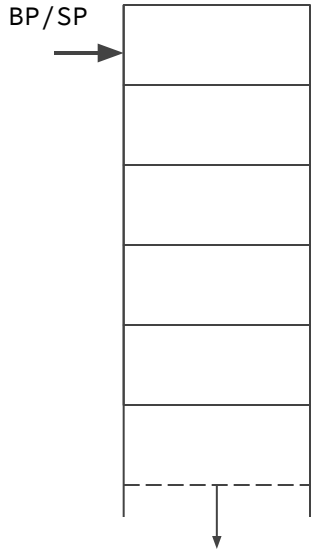
Instruction Pointer (IP): marca la instrucción que se está ejecutando.



Ejecución

IP

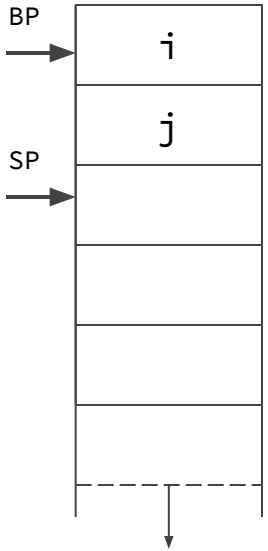
```
00  void function() {
01      int i = get_five();
02      int j = get_five();
03  }
04
05  int get_five() {
06      int five = 5;
07      return five;
08  }
```



Ejecución

```
00  void function() {
01      int i = get_five();
02      int j = get_five();
03  }
04
05  int get_five() {
06      int five = 5;
07      return five;
08  }
```

IP →



IP++

SP = SP - 8

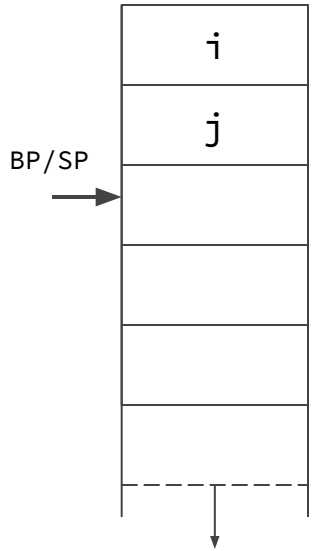
IP++: próxima instrucción.

El CPU lo ejecuta automáticamente.

Ejecución

```
00  void function() {
01      int i = get_five();
02      int j = get_five();
03  }
04
05  int get_five() {
06      int five = 5;
07      return five;
08  }
```

IP →



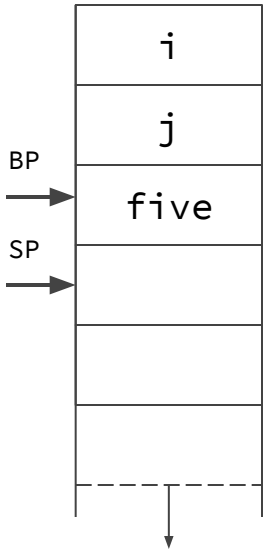
IP = &get_five

BP = SP

Ejecución

```
00  void function() {
01      int i = get_five();
02      int j = get_five();
03  }
04
05  int get_five() {
06      int five = 5;
07      return five;
08  }
```

IP → 06



IP++

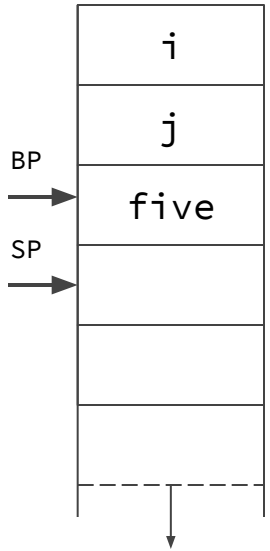
SP = SP - 4

Ejecución

```
00  void function() {
01      int i = get_five();
02      int j = get_five();
03  }
04
05  int get_five() {
06      int five = 5;
07      return five;
08  }
```

IP → 07

???



IP = ??

BP = ??

SP = ??

???

Ejecución

Después del **`return`** necesitamos una manera de poder restaurar el function frame anterior y de ejecutar la próxima instrucción.

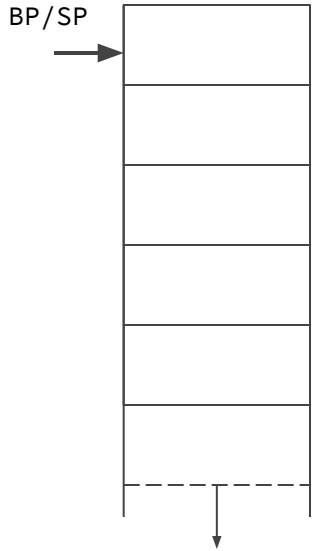
La única manera que podemos hacer esto es guardando los valores que necesitamos antes de sobreescribirlos.

¿Dónde los guardamos? ¡En el stack!

Ejecución

IP

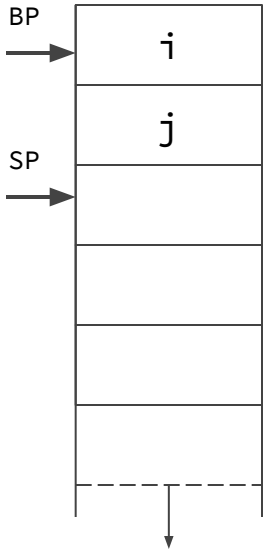
```
00  void function() {
01      int i = get_five();
02      int j = get_five();
03  }
04
05  int get_five() {
06      int five = 5;
07      return five;
08  }
```



Ejecución

```
00  void function() {
01      int i = get_five();
02      int j = get_five();
03  }
04
05  int get_five() {
06      int five = 5;
07      return five;
08  }
```

IP



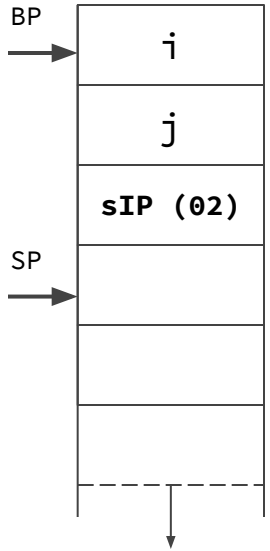
IP++

SP = SP - 8

Ejecución

```
00  void function() {
01      int i = get_five();
02      int j = get_five();
03  }
04
05  int get_five() {
06      int five = 5;
07      return five;
08  }
```

IP



Push IP + 1 (SP -= 4)

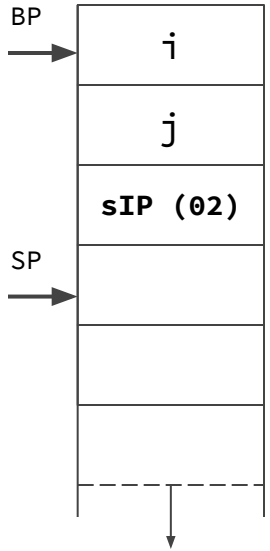
i, j son solo alocaciones.

Distinguimos **sIP** con negrita para marcar que es un valor que pusheamos.

Ejecución

```
00  void function() {
01      int i = get_five();
02      int j = get_five();
03  }
04
05  int get_five() {
06      int five = 5;
07      return five;
08  }
```

IP →

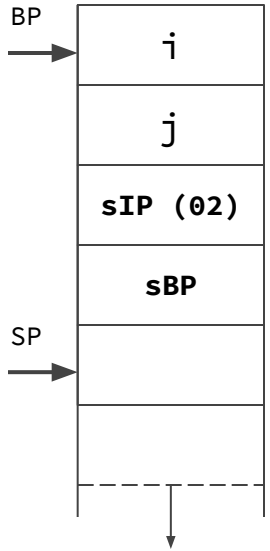


IP = &get_five

Ejecución

```
00  void function() {
01      int i = get_five();
02      int j = get_five();
03  }
04
05  int get_five() {
06      int five = 5;
07      return five;
08  }
```

IP →

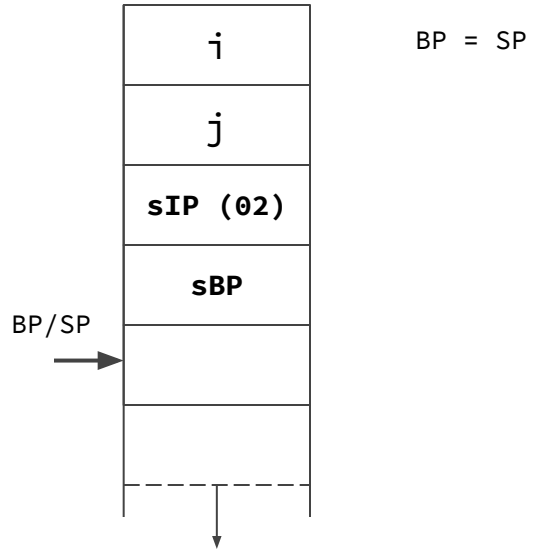


Push BP (SP -= 4)

Ejecución

```
00  void function() {
01      int i = get_five();
02      int j = get_five();
03  }
04
05  int get_five() {
06      int five = 5;
07      return five;
08  }
```

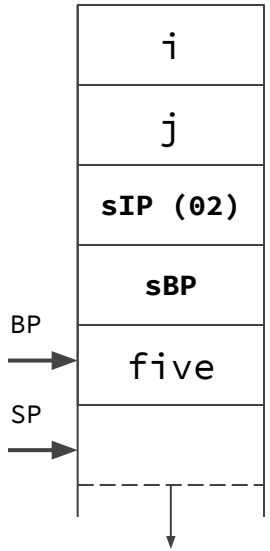
IP →



Ejecución

```
00  void function() {
01      int i = get_five();
02      int j = get_five();
03  }
04
05  int get_five() {
06      int five = 5;
07      return five;
08  }
```

IP →



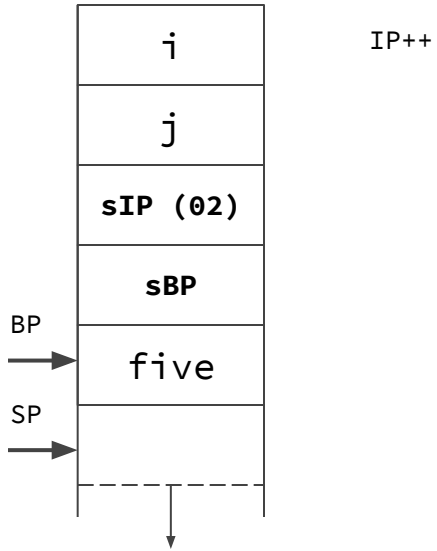
IP++

SP = SP - 4

Ejecución

```
00  void function() {
01      int i = get_five();
02      int j = get_five();
03  }
04
05  int get_five() {
06      int five = 5;
07      return five;
08  }
```

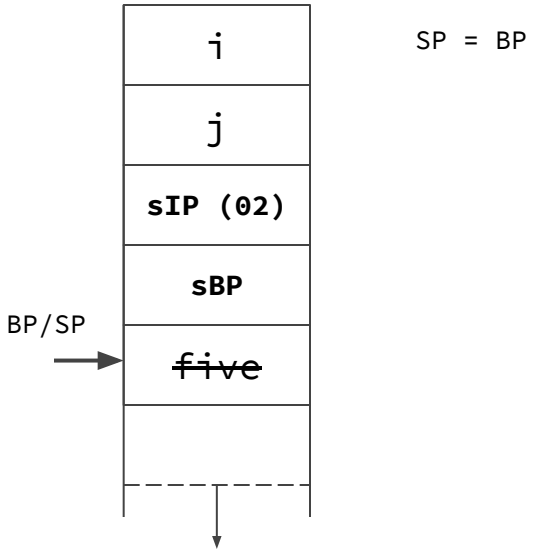
IP →



Ejecución

```
00  void function() {
01      int i = get_five();
02      int j = get_five();
03  }
04
05  int get_five() {
06      int five = 5;
07      return five;
08  }
```

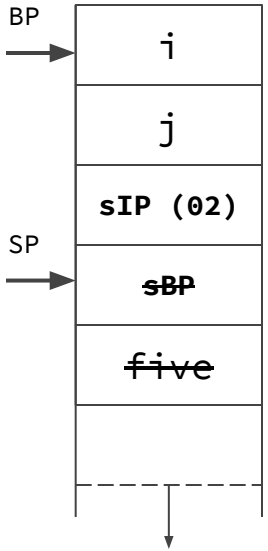
IP →



Ejecución

```
00  void function() {
01      int i = get_five();
02      int j = get_five();
03  }
04
05  int get_five() {
06      int five = 5;
07      return five;
08  }
```

IP →



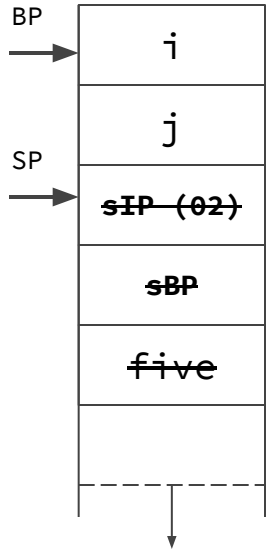
BP = Pop (SP += 4)

Pop retorna sBP

Ejecución

```
00  void function() {
01      int i = get_five();
02      int j = get_five();
03  }
04
05  int get_five() {
06      int five = 5;
07      return five;
08  }
```

IP → 02



IP = Pop (SP += 4)

Pop retorna sIP

Otros bloques

Los demás bloques que no son funciones no tienen sus propios frames.

Es posible que el compilador agrande el function frame actual al entrar a un bloque y cuando el programa sale del bloque achique el frame para desocupar espacio en el stack.

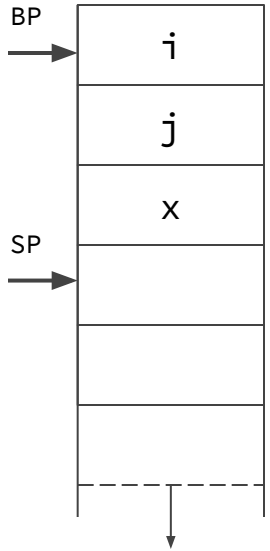
También puede ser que se aloquen todas las variables al inicio de la función, sin importar a qué bloque pertenecen.

El compilador puede tomar la decisión que quiera, ya que referenciar un objeto fuera de su lifetime es **undefined behavior**.

Otros bloques (Opción 1)

```
00  void function() {
01      int i;
02      int j;
03      {
04          int x;
05      }
06  }
```

IP



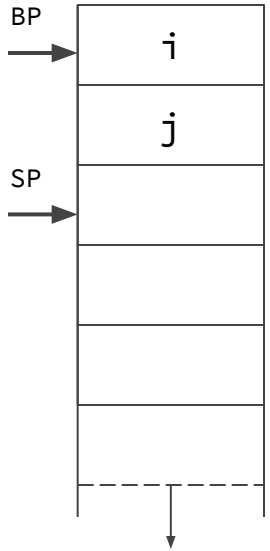
x ya está alocada fuera de su bloque

SP -= 12

Otros bloques (Opción 2)

```
00  void function() {
01      int i;
02      int j;
03      {
04          int x;
05      }
06  }
```

IP



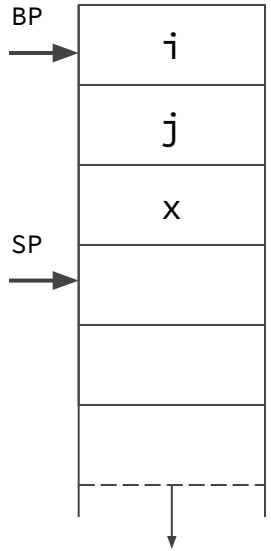
Sólo i, j están alocadas.

SP -= 8

Otros bloques (Opción 2)

```
00  void function() {
01      int i;
02      int j;
03      {
04          int x;
05      }
06  }
```

IP → 04



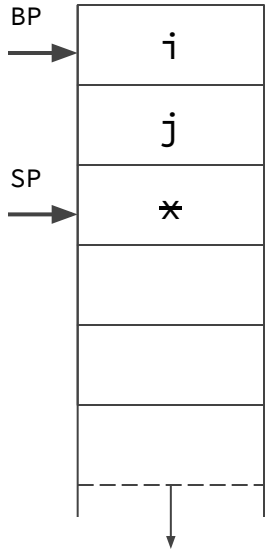
x se aloca al entrar a su bloque

SP -= 4

Otros bloques (Opción 2)

```
00   void function() {
01        int i;
02        int j;
03        {
04             int x;
05        }
06   }
```

IP



Al salir del bloque, x se desaloca.

Puedo usar ese espacio para otro bloque.

SP += 4

Arrays en el stack

Los miembros de un array se alocan contiguamente en memoria. Conforme avanzamos en el array la dirección donde está alocado el miembro debe **subir**.

La alocación del stack va desde direcciones altas a direcciones bajas. Sin embargo, la lectura/escritura de los datos va desde direcciones bajas a altas.

El primer elemento del array va a estar cerca del top del stack.

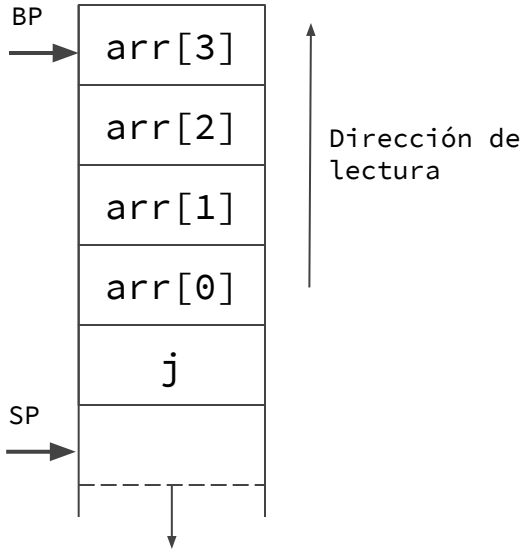
Si leemos/escribimos más allá de los límites del array (overflow), vamos a sobreescribir function frames anteriores.

Lo mismo se aplica a structs y unions.

Arrays en el stack

```
00   void function() {
01        int arr[4];
02        int j;
03   }
```

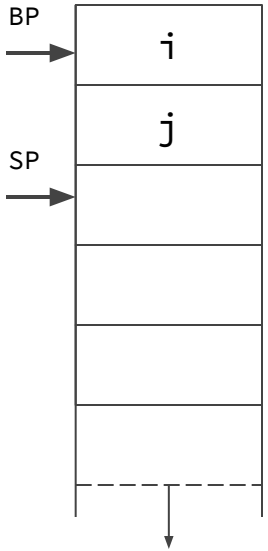
IP



Arrays en el stack

```
00  void function() {
01      int i, j;
02      other_function();
03  }
04
05  void other_function() {
06      int arr[2];
07  }
08
```

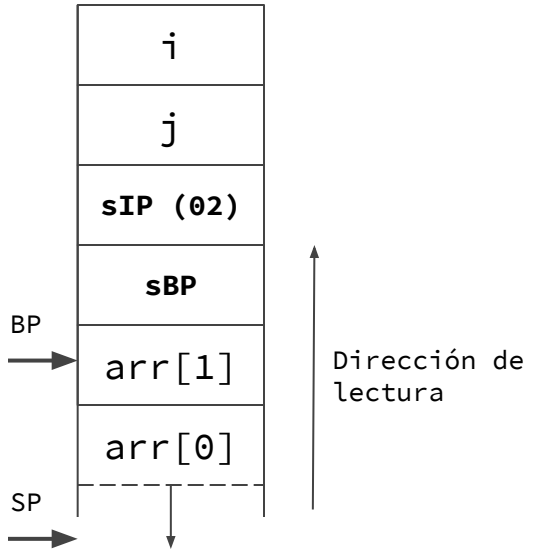
IP



Arrays en el stack

```
00   void function() {
01        int i, j;
02        other_function();
03   }
04
05   void other_function() {
06        int arr[2];
07   }
08
```

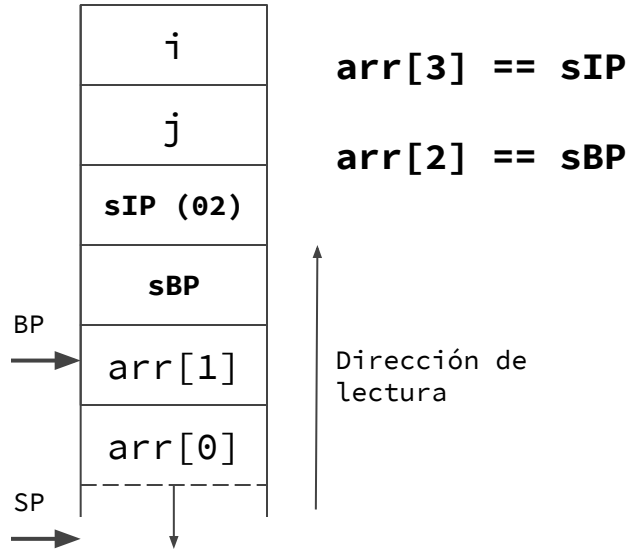
IP →



Arrays en el stack

```
00   void function() {
01        int i, j;
02        other_function();
03   }
04
05   void other_function() {
06        int arr[2];
07   }
08
```

IP →



Ex 8.1

Arrays en el stack

Escribir fuera de los límites de un array en el stack en el **mejor** de los casos causa Segmentation Fault.

En casos peores, corrupción de memoria.

En el peor de los casos, le da al usuario control total de la computadora.

Existen **mitigaciones** contra esto, como el canary value. Si el “usuario” ya tomó control del IP, PIE y ASLR dificultan las cosas.

String literals

Cuando declaramos un string **dentro de una función** de la siguiente manera:

```
char *str = “hello”;
```

Lo que se guarda en el stack es **solo** el pointer al inicio del string. El string en sí se va a alocar estáticamente. Modificar este string es undefined behavior.

Si en cambio hacemos:

```
char str[] = “hello”;
```

El array que contiene el string se va a alocar en el stack y puede ser modificado.

Ex 10.1, 10.2 | C99 6.4.5.6

String literals

Si el string está alocado estáticamente, podríamos modificar su valor y la próxima vez que usemos la variable

```
char *str = “hello”;
```

El contenido va a ser otro.

Para simplificar las cosas, es común que los compiladores guarden los string literals declarados de esta manera en una sección **readonly** de la memoria. Esto garantiza Segmentation Fault al querer modificar el contenido.

```
const char *str = “hello”;
```

Alocación dinámica

Hasta ahora

Alocaciones estáticas:

- Fijadas en tiempo de compilación
- Existentes durante todo el programa
  - No tenemos que preocuparnos por su lifetime
  - Siempre ocupan espacio en memoria

Alocaciones automáticas:

- Varían en runtime, dependiendo a qué bloques entramos
- Existen sólo durante su bloque
  - Solo podemos usarlas dentro de este bloque
  - Liberan memoria automáticamente

Problemas

No tenemos forma de:

- Pasar alocaciones de memoria a distintos lugares del programa.
  - Si salimos del bloque de una alocación automática, el objeto deja de existir.
  - No podemos retornar una alocación de memoria en una función.
- Aumentar o disminuir la memoria que consumimos.
  - Las alocaciones estáticas tienen un tamaño fijo.
  - Lograr esto con alocaciones automáticas sería muy complejo.
- Remover la responsabilidad de alocación del usuario de nuestro código.
  - Las alocaciones sólo pueden ser “top-down” desde main a las funciones que llama, de estas funciones a las funciones que ellas llaman, etc.

Alocación dinámica

C soluciona estos problemas brindándonos funciones para alocar y desalocar memoria como deseemos. Este conjunto de funciones son:

- **`malloc`**: aloca un espacio en memoria con el tamaño deseado.
- **`calloc`**: aloca un espacio en memoria para un array de objetos, inicializado con ceros.
- **`realloc`**: desaloca un espacio de memoria y realoca su contenido en otro espacio con el tamaño deseado.
- **`free`**: desaloca un espacio de memoria.

Características

C99: “El pointer retornado si la alocación es exitosa es alineado tal que puede ser asignado a un pointer de cualquier tipo de objeto y luego utilizado para acceder a tal objeto o a un array de tal objeto en el espacio alocado...”.

Al no saber qué va a ser almacenado en el espacio de memoria que alocamos, el lenguaje debe garantizar que cualquier elemento va a estar propiamente alineado.

“...(hasta que ese espacio sea explícitamente desalocado). El tiempo de vida de un objeto alocado extiende desde su alocación hasta su desalocación”.

Tenemos control total sobre el lifetime de los objetos.

C99 7.20.3.1

Características

C99: “Cada alocación rendirá un pointer a un objeto disjunto de cualquier otro objeto. El pointer devuelto apunta al principio del espacio alocado.”

Garantías estándar.

“Si el espacio no puede ser alocado, se retornará un NULL pointer.”

Al ser complementamente dependiente de las condiciones en runtime, es posible que el programa solicite más memoria de la disponible en el sistema.

C99 7.20.3

malloc

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

`malloc` retorna un pointer al inicio de un espacio contiguo alocado en memoria con el tamaño indicado por `size`.

El contenido de la memoria es indefinido.

Tenemos un problema, que tipo de pointer debería retornar `malloc`? La librería estándar no tiene manera de saber para qué vamos a utilizar la memoria alocada.

No tenemos generics para especificar el tipo que necesitamos.

C99 7.20.3.3

Antes de Generics

Cuando no había Generics en Java (previo a Java 1.5), las colecciones simplemente guardaban Objects.

En vez de `List<Integer>` o `List<Point>` solo teníamos acceso a `List`.

Sin embargo, si sabes los elementos que guarda la lista, aunque el compiler solo sepa que son Object, podes castearlos devuelta a su verdadero tipo.

El equivalente (a grandes rasgos) de Object en C es el **`void*`**.

`void*` significa que tenemos un pointer a *algo*. No sabemos qué.

Ex 11.1, 11.2, 11.3

void*

Como los **void*** no tienen información sobre su tipo, no podemos usar pointer arithmetic con ellos. Después de todo, pointer arithmetic se basa en el **sizeof** de los objetos referenciados. **void** en sí mismo no es un tipo de objeto válido, es un *tipo incompleto*.

Sin embargo con **void*** seguimos teniendo la dirección de memoria disponible para utilizar. Simplemente tenemos que castear el pointer al tipo que deseemos y podemos utilizar el espacio de memoria como si fuera un pointer normal del tipo que casteamos.

```
char *c_ptr = (char *) malloc(sizeof(char));
```

Ex 12.1, 12.2

void*

Al igual que con Object, podemos asignar cualquier pointer a una variable declarada como **`void*`** sin necesidad de castear. Además C permite asignar **`void*`** a cualquier variable que sea un pointer.

También, al igual que en Java antes de tener Generics, los **`void*`** se utilizan para poder tener colecciones de objetos reutilizables. Lo mismo para todo tipo de funciones donde no nos interesa el tipo del elemento.

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

La librería estándar no tiene forma de saber el tipo del objeto que vamos a guardar, *ni le interesa*.

Ex 13.1

calloc

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

`calloc` aloca un espacio contiguo en memoria para un número `nmemb` de objetos de tamaño `size`. Además garantiza que el **contenido** de esta memoria va a estar **lleno de ceros**.

`calloc` tiene una signature más amigable para alocar arrays, pero lo mismo puede ser logrado utilizando `malloc`.

```
char *str = calloc(5, sizeof(char));
```

```
char *str = malloc(5 * sizeof(char));
```

Ex 14.1, 14.2 | C99 7.20.3.1

free

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

`free` es la contracara de `malloc` y `calloc`. Recibe un pointer alocado por una de estas dos funciones y libera la memoria asociada a él.

`free` no necesita saber el tamaño del espacio alocado ya que la librería mantiene un registro del tamaño de cada espacio ya alocado.

Esta es una de las razones por las que llamar `free` sobre un pointer que no fue alocado mediante las funciones mencionadas (o `realloc`) es **undefined behavior**.

La excepción es NULL, que es ignorado.

Ex 15.1, 15.2 | C99 7.20.3.2

realloc

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

`realloc` combina la funcionalidad de `free` y `malloc` para permitir modificar el tamaño de los espacios de memoria asignados. El procedimiento es:

1. Alocar un nuevo espacio de memoria con el tamaño `size`
2. Copiar el contenido del espacio de memoria apuntado por `ptr` al nuevo espacio de memoria.
3. Desalocar el espacio de memoria apuntado por `ptr`.
4. Retornar un pointer al nuevo espacio alocado.

C99 7.20.3.4

`realloc`

Esta misma funcionalidad podría ser implementada combinando `free` y `malloc`. La librería estándar ofrece `realloc` en parte porque es un patrón muy común pero más importante porque **puede reutilizar la memoria ya alocada**.

En vez de alocar un espacio nuevo, `realloc` puede achicar o agrandar el espacio ya alocado si existe la posibilidad. Esto ahorra la copia de todo el contenido en memoria y el overhead de alocar un espacio y desalocar el otro.

Por esto es posible que `realloc` retorne el mismo pointer que pasamos como parámetro.

Alocación dinámica

La alocación dinámica es la última pieza que necesitamos para escribir programas en C ya que nos permite alocar memoria con el tamaño que necesitemos y durante el tiempo que necesitemos.

Sin embargo, siempre tenemos que tener claro cuando dejamos de usar una alocación de memoria para poder liberarla.

Ex 17.1, 17.2

Memory leaks

Cuando dejamos de usar una alocación de memoria tenemos que liberarla con `free`. Sin esto, la memoria nunca se libera y causamos un “memory leak”.

Si perdemos control de un pointer a un espacio de memoria alocado dinámicamente **no tenemos manera de recuperarlo**. Ese espacio va a permanecer alocado hasta que el programa termine.

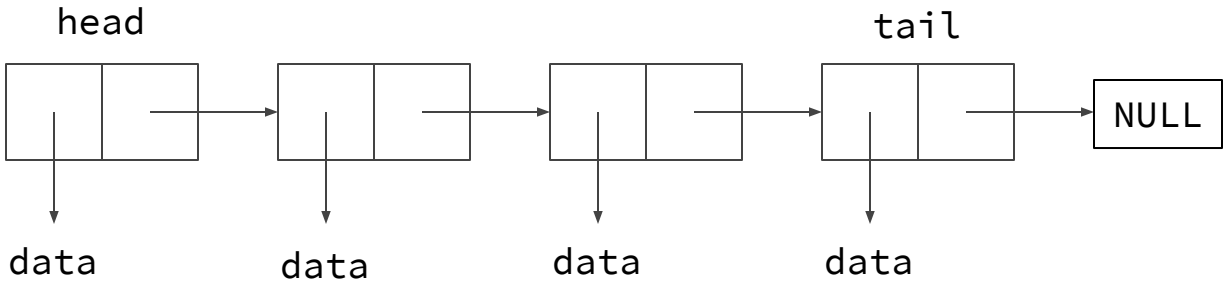
Que tan perceptible es un leak depende de la frecuencia con la que ocurre y el tiempo está corriendo el programa. Un servidor leakeando memoria puede durar horas o hasta días. Poco a poco el proceso consume más memoria hasta que crashea.

Ex 18.1

Linked Lists

Aprovechando la alocación dinámica podemos implementar una linked list o lista enlazada.

Para hacerla reutilizable, vamos a guardar **`void *`**, de manera que podemos tener una lista de cualquier tipo de objeto. Técnicamente, cualquier tipo de pointer sería válido, pero usar **`void *`** hace nuestra intención más clara.



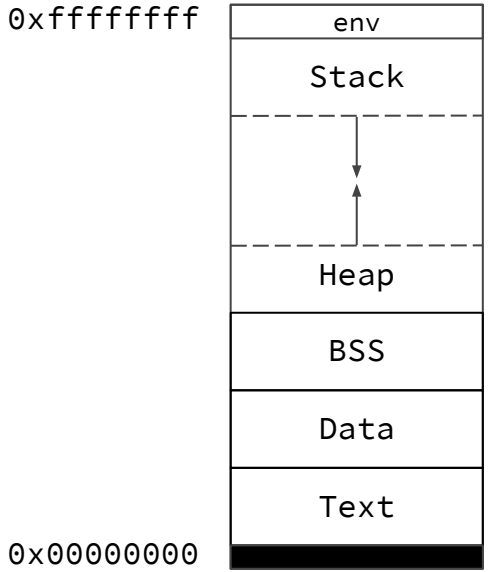
Ex 19.1 | https://visualgo.net/en/list

¿Dónde se encuentran las alocaciones dinámicas?

Las alocaciones dinámicas se encuentran en una sección de memoria comúnmente llamada “heap”.

No tiene relación alguna hoy en día con la estructura de datos del mismo nombre.

Esta sección de memoria es manejada por la librería estándar que implementa las funciones de alocación.

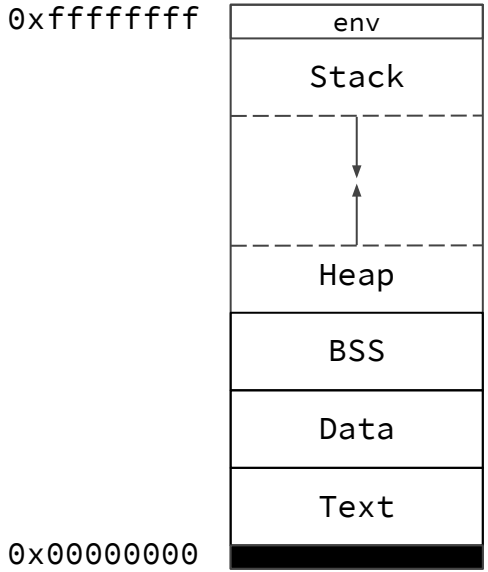


El Heap

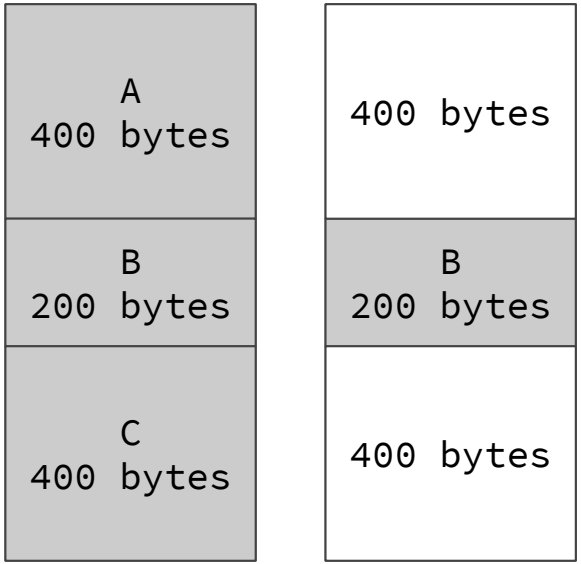
A diferencia del stack, que siempre va a estar ordenado según las funciones que el programa va ejecutando, el heap es totalmente desordenado.

Esto se debe a que el usuario tiene completo control sobre el lifetime de las alocaciones.

No hay manera de poder reordenar las alocaciones, ya que necesitamos mantener fijas sus direcciones de memoria. Esto introduce **fragmentación externa**.



Fragmentación externa



free(A)
free(C)

Puedo hacer malloc(500)?

No, a pesar que tengo 800 bytes libres.

Fragmentación interna

También existe **fragmentación interna** debido a los datos que la librería tiene que mantener cada vez que alocamos un espacio de memoria.

Así es como `free` no requiere que le indiquemos el tamaño de la memoria a liberar. Cada llamada a `malloc` debe no sólo alocar el espacio requerido por el usuario, sino también un extra para incluir toda la metadata necesaria.

Además, `malloc` va a estar obligado a alinear cada espacio de memoria de manera que se pueda guardar cualquier tipo de dato posible.

Fragmentación interna

Si vemos el sourcecode de malloc vamos a encontrar que cada alocación (chunk) de memoria ocupa 2 **`size_t`** extra.

Finalmente, tenemos que tomar en cuenta las optimizaciones para disminuir la **fragmentación externa**. Para disminuir la cantidad de espacios de memoria inútiles que pueden quedar atrás si hacemos muchas alocaciones chicas, `malloc` establece un mínimo de espacio que va a alocar.

Alocar espacio para guardar 1 **`char`**, que ocupa 1 byte, puede costar **32 bytes**.

https://sourceware.org/glibc/wiki/MallocInternals
https://github.com/lattera/glibc/blob/master/malloc/malloc.c

El algoritmo detrás de malloc

La alocación dinámica debe ser **extremadamente eficiente**. Es utilizada por todos los programas y además difícil de implementar.

Como alocador de propósito general, `malloc` tiene que ser:

- Rápido
- Conservador de espacio
- Portable
- Configurable

Debido a esto, su algoritmo es también extremadamente complejo.

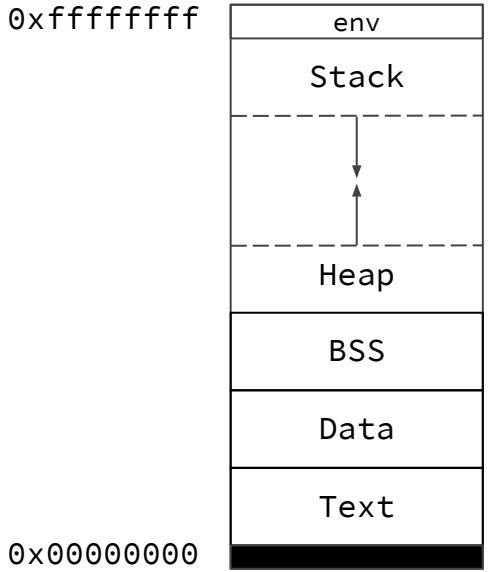
https://raw.githubusercontent.com/cloudburst/libheap/master/heap.png

¿Cómo obtiene memoria `malloc`?

Dejando de lado la mayor parte del algoritmo. ¿Cómo es que `malloc` obtiene más memoria para asignarle al programa?

Al igual que el stack, el heap puede crecer el runtime. A diferencia del stack, cuyo tamaño es modificado automáticamente por el SO, el heap debe solicitar nuevas páginas manualmente.

Para esto debe hablar con el SO mediante **system calls**.



System calls

Otra funcionalidad del sistema operativo son las **system calls**.

Cada proceso corre sobre el sistema operativo cómo si estuviera aislado completamente. Cuando necesita realizar una acción que afecte el exterior, como por ejemplo alocarse más memoria, necesita hablar con el sistema operativo.

Las system calls son el conjunto de “funciones” que el sistema operativo expone a cada proceso para poder realizar estas acciones.

C provee un set de funciones que son wrappers a system calls.

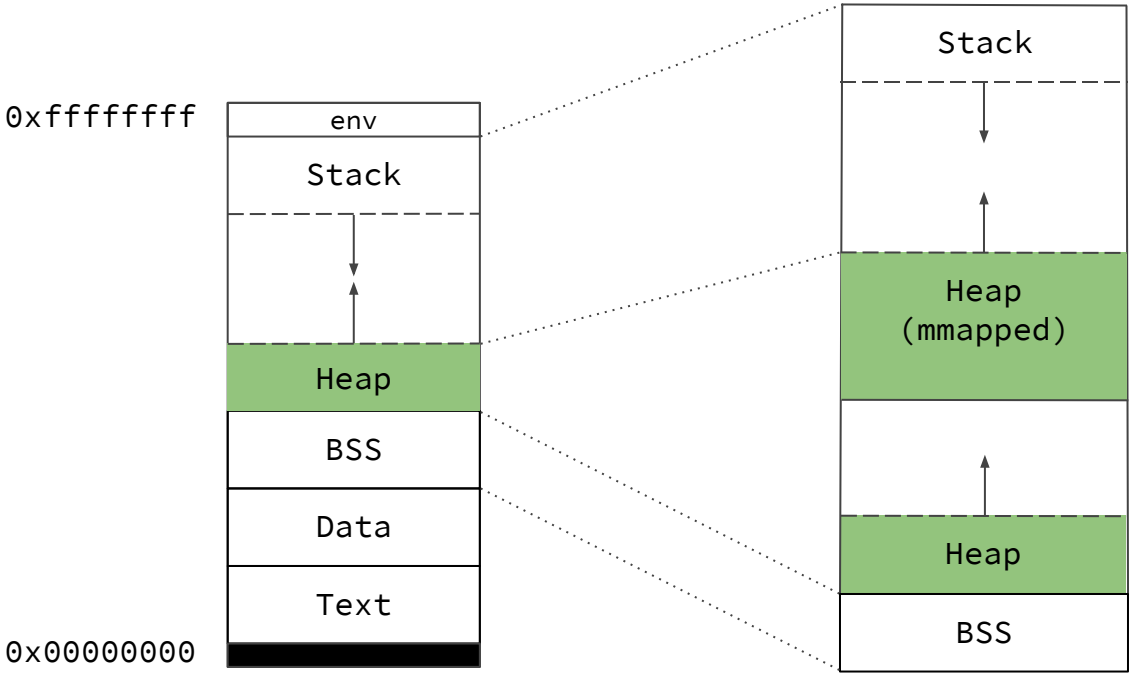
https://man7.org/linux/man-pages/man2/syscalls.2.html

System calls

En Linux, hay dos system calls diferentes que `malloc` puede usar.

- `brk/sbrk`: modifican el “break” del programa. El break es el final de la sección de datos, donde se encuentra BSS.
  - Para alocar más memoria, movemos el break hacia direcciones más altas, para desalocar movemos el break a direcciones más bajas.
- `mmap`: aloca nuevas páginas al proceso. Las nuevas páginas van a estar entre el break y el stack.
  - Para desalocar las páginas llamamos otra system call, `munmap`.

mmap vs brk



Páginas mapeadas por **mmap**

Break, modificado por **brk**

System calls

Qué system call (`brk`/`mmap`) va a usar `malloc` depende de varios factores.

Lo importante es que toda system call tiene un overhead **significante**, no es simplemente llamar a una función del programa.

`malloc` sólo va a pedir más memoria al SO si es que no tiene ninguna manera de reutilizar el espacio ya disponible. Además, cuando solicite memoria lo va a hacer en páginas. El SO sólo puede alocar en páginas y todo el espacio extra puede ser utilizado para nuevas alocaciones, evitando más system calls.

strace -p PID (15.2)

Stack vs Heap

Si comparamos el trabajo necesario para alocar una variable en el Heap con alocar la misma variable en el Stack, la diferencia es considerable.

Para alocar en el stack, simplemente necesitamos disminuir el valor del stack pointer.

Para alocar en el heap, tenemos que llamar a `malloc`, un algoritmo complejo que además posiblemente involucre system calls.

En lo posible, si no necesitamos la funcionalidad de la alocación dinámica, es mucho más eficiente y seguro (eliminamos leaks) usar alocación automática.

Overflow en el Heap

Al escribir fuera de los límites del array en el stack podemos sobreescribir datos del function frame. Esto va a traer problemas cuando retornemos de la función y es dentro de todo fácil de detectar.

En cambio en el heap vamos a sobrescribir contenido de otras alocaciones de memoria. Este tipo de problemas sólo se van a detectar cuando vayamos a usar una de las alocaciones corruptas o decidamos liberarlas con `free`.

Además, no tenemos garantías de que alocaciones se encuentran secuencialmente en memoria. Esto depende de su tamaño, que alocaciones se liberaron antes, etc.

¿Qué pasa en Java?

Como vimos, en Java nunca trabajamos con el verdadero objeto, sólo con las referencias.

En Java el stack se va a usar sólo para variables primitivas y referencias a objetos (el equivalente a pointers en C).

Todo lo que se cree con `new` va a ser alocado en el heap.

La principal diferencia con las alocaciones de C es que en Java no tenemos que manejar manualmente las alocaciones dinámicas.

Esto se maneja con un **garbage collector**.

Garbage collection

Muchos lenguajes de alto nivel optan por utilizar garbage collection en vez de dejar la alocación de memoria en manos del programador. El garbage collector va a liberar periódicamente las alocaciones de memoria que ya no se **referencian**.

Esto “elimina” los memory leaks a cambio de menor performance, ya que se necesita mantener registro de qué objeto es usado dónde. Además, cada vez que el garbage collector se ejecuta, la ejecución completa del programa debe ser detenida. Esto introduce impredecibilidad en el programa.

La versión Java de Minecraft ha tenido muchos problemas de performance debido al garbage collector.

https://gist.github.com/radfast/6f966fc0577b3e1a980d

Leaks en garbage collection

Garbage collection elimina memory leaks sólo en el caso en que dejamos de referenciar un objeto. El equivalente sería alocar un espacio de memoria en C y retornar de la función sin mantener una referencia al pointer, ni llamar a `free`.

Sin embargo, podemos seguir leakeando memoria si simplemente no eliminamos referencias a los objetos. Solo basta con que haya una sola referencia a un objeto presente para que el garbage collector no lo pueda liberar.

Por ejemplo: no eliminar objetos cuando salen de la pantalla en un juego.

https://github.com/ldurniat/Asteroids