Memoria

Abril Diaz Miguez

Agosto 2023

Apunte de la materia Algoritmos y Programación II cátedra Mendez-Pandolfo. En éste, se presenta una introducción al tema memoria, almacenamiento de variables en un programa en ejecución, de suma importancia en la materia.

Contents

1 Introducción

Antes de estudiar temas más avanzados en informática, como punteros, memoria dinámica y estructura de datos, tres conceptos centrales de la materia, me parece una buena idea repasar desde el principio temas de memoria y almacenamiento de información en la computadora. Estos temas se verán en más profundidad a lo largo de la carrera, este apunte no pretende explicar todos los detalles de los mismos, sino presentar una base común necesaria y suficiente para que los alumnos que cursen Algoritmos y Programación II puedan comprender y adquirir los conceptos de la materia.

Cabe destacar que los conceptos de memoria se aplican a todos los lenguajes de programación, pero que debido a que el curso se dicta en el lenguaje C, los ejemplos y las explicaciones prácticas estarán en ese lenguaje.

2 Memoria 101: ¿Qué es "la memoria"?

Suponete que tenés que escribir un informe para un trabajo práctico. Como hoy en día nadie entregaría un informe escrito a mano, vas a tu computadora, abrís un editor de texto (como Word, aunque no es el único), escribís tu informe, guardando de vez en cuando para no perder el progreso. Esta serie de acciones son muy útiles para explicar los diferentes espacios que tiene la computadora para almacenar información.

Por un lado está lo que se conoce comúnmente como "disco", porque antes era... sí, un disco. Hoy en día la electrónica y los componentes de la computadora avanzaron y evolucionaron mucho, así que puede no ser exactamente un disco[1], pero el nombre se mantiene. En el disco, se guarda toda la información (archivos, programas, etc.), que no queremos que desaparezcan cuando apagás la computadora. Si reiniciás tu computadora y encontrás el informe que venías escribiendo en Word, es porque lo guardaste en disco. No importa cuántas veces reinicies, apagues, suspendas o cierres y abras nuevamente la sesión de tu computadora, todo lo que se guardó en disco puede ser accedido nuevamente. Los archivos no se pierden. A esta característica se la conoce como no volátil, y es la principal ventaja del disco. Bueno, ésa y su precio: un disco es mucho más barato que otros componentes de almacenamiento. La gran desventaja es del disco es lo lento que es en comparación a otros componentes. Lo que me lleva al siguiente componente.

¿Nunca te pasó que estabas escribiendo el informe del ejemplo y se le acabó la batería a tu computadora, causando que se apague? Y cuando la cargabas y reiniciabas, resultó que una buena parte de lo que habías escrito se había perdido, no se había guardado. Esto ocurre porque no existe solamente el disco a la hora de hablar de almacenamiento. También está el componente llamado Random Access Memory, RAM; "memoria" para los amigos. Lo que importa

[1] El llamado "disco duro" sí tiene forma de disco, incluso en la actualidad. Pero componentes como el disco de estado sólido, aunque se llaman disco, no tienen forma de disco... Detalles.

de esta nueva zona de almacenamiento es que es mucho más rápida que el disco, pero esto implica que también es más cara. Otra gran característica de la RAM, y la que mencionamos al principio del párrafo, es que es volátil: no da persistencia de datos, cuando se corta la corriente de electricidad, los datos, la información, se pierde.

Éstos son los dos principales componentes de almacenamiento en la computadora[2], y los que nos importan en este curso. Pero, ¿cuál es la relación entre RAM y disco? ¿Cuándo se usa cada una? ¿Para qué quiero las dos?

2.1 ¿RAM o Disco?

Es fácil confundir estos dos componentes. Después de todo, ambos sirven para guardar información, ¿no? Y aunque la respuesta es sí, en realidad sirven propósitos distintos, por lo que compararlos sólo sirve para entender aún más sus diferencias y sus roles particulares.

El disco es memoria no volátil, y te provee persistencia de datos. En castellano, el disco te permite guardar información incluso cuando la compu se apaga, o se queda sin batería, o se suspende su sesión actual. Permite que la información se almacene de una manera permanente[3]. Esta característica nos es súper útil para guardar información que queremos que sea, justamente, permanente, o al menos tan permanente como nosotros querramos. Y que sea un componente barato también es un gran beneficio.

Uno podría pensar que con este tipo de almacenamiento es suficiente. Se tiene persistencia, y encima barata. ¿Qué más podría querer uno? La respuesta a eso es sencilla, pero importante: velocidad. El disco es lento, sobre todo en comparación a otros componentes. Ir a buscar algo en disco es muy costoso para un programa, que busca ser lo más rápido posible para una mejor experiencia de usuario.

Es por esto que surgen un montón de componentes para aliviar la carga, que se verán en más profundidad en futuras materias, y entre ellos está la RAM, o memoria. La memoria es no volátil (así que si se apaga la computadora, se pierde la información guardada en ella), y más cara, pero mucho (mucho) más rápida. Es natural entonces que se tenga mucho menos almacenamiento de este tipo, en comparación a disco (una máquina puede tener 1 Tera de disco y 32Gb de RAM, por ejemplo).

Pero las diferencias entre ambos componentes permiten que se utilicen con diferentes fines. Es el disco el que mantiene una copia de todos los archivos y programas en tu computadora, y es la RAM la zona que tiene una copia de

[2]Digo principales porque son los que más almacenamiento proporcionan, pero hay otras estructuras en la computadora que proporcionan almacenamiento y son de suma importancia, como los registros en la CPU y las cachés. Pero éstos escapan de los contenidos de este apunte y de la materia. Se verán más adelante en la carrera.

[3]Toma el "permanente" entre comillas, no vivimos en un mundo perfecto y se trata de un componente electrónico: puede haber fallos, y eventualmente los transistores dentro del disco dejan de almacenar la información correcta por la pérdida de energía. Pero para los fines prácticos del día y día que vos y yo vivimos, la información es permanente, existe hasta que nosotros decidamos borrarla

los archivos y programas que vos querés que sean rápidamente accedidos o ejecutados. ¿Y cuáles son esos archivos y programas? ¡Los que querés ejecutar ahora, por supuesto! En definitiva, en la RAM se encuentra una copia de todos aquellos programas que vos querés ejecutar actualmente en tu computadora, y todos los archivos que tenés abiertos actualmente. Esto es para que la ejecución de los programas y la interacción con los archivos sea rápida, proveyendo una buena experiencia de usuario. Aquellos archivos que no estás mirando, o aquellos programas que no estás ejecutando, solamente están en disco. ¿Por qué desperdiciarías preciado espacio de RAM en programas o archivos que el usuario no está usando?

Una pregunta que podés estar haciéndote es ¿y quién controla que algo pase de disco a RAM? ¿Quién administra el disco, la RAM, y esos componentes? La respuesta corta es el sistema operativo. La respuesta larga es que lo vas a ver en Sistemas Operativos, pero en pocas palabras, es un programa especial, que tiene todos los permisos posibles y que está encargado de controlar el acceso a memoria y disco, entre muchas otras cosas. Pero no te adelantes, eso lo verás en otras materias.

En definitiva, cuando estés trabajando en tu informe y le des a `Ctrl+S`, lo que estás haciendo es pedirle al sistema operativo que pase una copia de tu informe, que vive en la RAM, a la copia original del informe, que está desactualizado y vive en disco. Y si antes de apretar las teclas, se te acaba la batería de la compu, que sepas que lo que perdiste de modificaciones es porque estaban guardadas en la RAM, pero no en disco.

Con todo esto dicho, que no es poco, tal vez no queda del todo claro cómo se organiza en sí la memoria. ¿Qué hay en la memoria en sí? Ya dijimos que "información" o "datos", pero esos son conceptos más abstractos. Específicamente, ¿qué hay en memoria? ¿Y cómo se accede a esa información?

2.2 ¿Qué hay en memoria?

Ya sabemos que en memoria, al igual que en disco, se guarda información. Sólo que en memoria se guarda aquella información pertinente a archivos, programas y demás que se estén ejecutando actualmente en tu computadora. Esa app para escuchar música, ese editor de texto, el programa para ejecutar juegos, todos esos programas, y todos los archivos que lo componen, fueron copiados a RAM por el sistema operativo. Pero, ¿qué se copió? ¿Si pudiese ir a la RAM con una lupa muy (muy) grande, qué vería?

Por supuesto que en la RAM, al ser un componente electrónico, sólo hay corriente (o la ausencia de ella). Pero en informática no nos interesa ir a tan bajo nivel, llegar tan a los fierros. Lo más bajo nivel que vamos a estudiar nosotros en esta carrera es la representación de "hay electricidad" y "no hay electricidad" con un concepto llamado bit. Un bit es la unidad mínima de información, y vale 0 cuando "no hay electricidad" y 1 cuando "sí hay electricidad" en el componente. Un bit sólo tiene los dos estados que recién se mencionaron, así que por su cuenta no nos da mucha información. Pero si se empieza a juntar a muchos, muchos, muchos bits, llegamos a los programas que se ejecutan hoy en

día, con toda la complejidad que implican.

Aunque es cierto que un bit es la unidad de información, en realidad nada hoy se almacena en un bit. Todo elemento informático se almacena en al menos un byte. Un byte son ocho bits agrupados. Como cada bit sólo puede valer 1 ó 0, los bytes son una combinación de 1s y 0s. ¿Esto quiere decir que solamente puedo escribir números con 1s y 0s, no puedo escribir el 4 u 8 o 1928534224234 en bytes? Por supuesto que se pueden escribir esos números en bytes, pero no con esos valores. Los bytes están en otro sistema numérico, llamado binario; el nombre implica que solamente se tiene dos valores distintos, el 1 y el 0.

Así, el número 0 se escribe como 0. El número 1 es 1. Pero el número 2 en decimal es 10 es binario. Y el tres en decimal es 11 en binario. Otro sistema numérico que tal vez escuchaste mencionar es el hexadecimal, que tiene 16 valores diferentes: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F (sí, las letritas también son "números", tal vez lo viste en algún mensaje de error). Más detalles de binario vs decimal vs hexa y otros sitemas numéricos, cómo hacer pasaje entre ellos y demás, se verá en otras materias.

En definitiva, ¿qué hay en memoria? Hay una tira de 1s y 0s, que si sabés dónde empiezan y dónde terminan, es posible descifrar la información allí guardada[4]. Todo archivo, todo programa, todo lo que esté almacenado en tu computadora, todo se traduce a una gran tira de 1s y 0s. Entonces, si estoy escribiendo mi informe para el trabajo práctico y escuchando música a la vez, hay al menos dos aplicaciones ejecutándose actualmente, por lo que hay dos programas que están siendo ejecutados[5], que decimos que "viven en memoria". Eso significa que hay dos grandes tiras de 1s y 0s en RAM, una para cada programa, que son la información pertinente a cada aplicación (incluyen el código en sí para que las aplicaciones se ejecuten, las pistas de sonido de la aplicación y la data del informe, etc).

Pero, ¿cómo sabemos dónde están estas tiras de 1s y 0s? La RAM no será tan grande como el disco, pero es grande. ¿Cómo sabe el sistema operativo, o quien sea, a dónde tiene que ir a buscar esta información de mi app en la RAM?

2.3 Direcciones de memoria

Pensá en una calle con casas.

Figure 1: Primer vistazo de memoria

[4]Este trabajo lo podrías hacer vos, pero en realidad nos interesa que lo pueda hacer la CPU, el componente de tu computadora encargado de ejecutar las instrucciones de todos los programas para que éstos funcionen y vos puedas interactuar con ellos

[5]Detalle: cuando un programa se ejecuta se lo conoce como proceso

Todas las casas son iguales. Cada casa es capaz de alojar a un inquilino, son monoambientes. Si un amigo tuyo se muda a una de estas casas, y vos querés ir a saludarlo, ¿cómo sabés a qué casa ir? Si son todas iguales... ¿cómo las distinguís? Esto mismo se preguntaron los de la constructora, así que le dieron a cada casa, que almacena a un inquilino independiente del otro, una dirección, un número, que puede ser alto o bajo, y que hace a la casa única: la diferencia de todas las otras casas.

Figure 2: Primer vistazo de memoria con direcciones

Si tu amigo vive en la casa 5, con que te diga que vayas a la casa 5 es suficiente, sabés a cuál de todas las casas ir. No hay ninguna otra casa con la dirección 5, está reservada para la casa de tu amigo.

En esta analogía, cada casa es un espacio de memoria. Es una porción de la memoria que se reserva, y solamente una cosa, un byte, puede estar en ella. Para poder encontrar ese byte (que es lo que nos interesa al fin y al cabo, queremos guardar información) cada porción de memoria tiene una dirección, única, que denominamos (de manera muy original) dirección de memoria. Cada inquilino representa al byte que vive en ese espacio de memoria.

Entonces, cuando se quiere buscar un byte en específico, conocer su dirección de memoria es condición necesaria y suficiente. Podemos ir a buscarlo en memoria y obtener el contenido. Pero que se entienda la diferencia: una cosa es la dirección de memoria y otra es el valor que en ese espacio de memoria se almacena. La dirección de la casa de tu amigo no es lo mismo que tu amigo. Uno se lo decís al tachero y con el otro ves el mundial.

Esto de ninguna manera implica que toda la información de un programa entra en un byte, ni de cerca. Cada aplicación ocupará un espacio, más o menos grande, de la RAM, reservando múltiples direcciones de memoria, para todo lo que necesita almacenar del programa.

Pero... ¿qué necesita almacenar un programa?

3 Las partes de un programa

Un programa en ejecución, llamado proceso, tiene cuatro partes principales: el code segment, el data segment, el stack y el heap. ¿Qué quiere decir esto? Todo el programa está en memoria, sí, pero eso no quiere decir que está todo

mezclado. La información necesaria y suficiente para que el proceso se ejecute correctamente se encuentra dividida en estos cuatro componentes. Repasemos cada una.

El code y data segment se verán en más detalle en futuras materias, pero a modo de introducción, en el code segment es donde se almacenan todas las líneas de código del programa. Todo el código que luego la CPU deberá ejecutar para que el programa se ejecute, se almacena en el code segment. Por otro lado, en el data segment se almacenan valores que son compartidos por todo el programa, que son globales. No te preocupes si no se termina de entender este concepto, lo volveremos a ver cuando veamos variables.

El stack es el segmento con el que probablemente más experiencia tengas. Seguramente ya lo escuchaste mencionar: "donde viven las variables". Para ser más específico, esta parte del proceso no es fija, va aumentando y disminuyendo a lo largo que el programa se ejecuta, y sirve para almacenar valores necesarios para el programa. Ya lo veremos en más detalle cuando repasemos variables. El heap es el cuarto y último componente de un proceso y, al igual que el stack, se modifica a lo largo de la ejecución del programa, y sirve para almacenar valores necesarios para el proceso. Pero la gran diferencia con el stack es cómo se amplía y disminuye a lo largo de la ejecución: a la hora de programar el programa ya se sabe cómo se va a modificar el stack; cómo se agranda o disminuye el heap depende de la ejecución del programa. Ya lo veremos en más detalle a lo largo de la materia.

Para terminar de entender para qué se usa cada uno de estos componentes y cómo se relacionan, pasemos a entender uno de los grandes conceptos de la programación: la variable.

4 ¿Qué son las variables?

Ya hablamos de memoria y disco, de cómo acceder a memoria y de las partes de un programa. Pero todavía no te dije qué es esta "información" en sí que quiero guardar de un programa. Sabemos que son 1s y 0s, y que están en una tira en disco, y que cuando quiero ejecutar el programa, necesito copiar esa tira de 1s y 0s a RAM. Pero ¿qué es esa tira? ¿Qué se almacena allí?

Si estás cursando esta materia, se presupone que ya programaste previamente en la carrera, así que el concepto de variable no debería ser uno muy ajeno. Pero no viene mal este repaso, que también puede ayudar a aquellos que no trabajaron antes con C.

En la gran mayoría de los lenguajes de programación existen las llamadas *variables*, que uno puede pensar como cajitas que almacenan información. Esta información puede ser una letra, un número entero, un número con coma, entre muchos otros. En algunos lenguajes, como Python o JavaScript, las variables pueden almacenar cualquier cosa en cualquier momento. Pueden primero almacenar una letra, luego almacenar un número, una referencia a un archivo y finalmente un vector que representa una entrada en una base de datos, siempre con el mismo nombre y sin tener que declararla ("crearla") nuevamente.

Código de este estilo es perfectamente posible en Python (aunque dudoso y no recomendado):

```
variable_versatil = 'A'
print("Mi variable almacena la letra:", variable_versatil)

variable_versatil = 23
print("Mi variable ahora almacena el número:",
        variable_versatil)


archivo = open('example.txt', 'r')
variable_versatil = archivo.read()
print("Mi variable ahora tiene el contenido de un archivo:",
        variable_versatil)
archivo.close()


variable_versatil = {
    'id': 1,
    'name': 'Juan Diaz',
    'age': 27,
    'location': 'Buenos Aires'
}
print("Variable ahora es un diccionario con información de un cliente",
    variable_versatil)
```

Este código en Python muestra la variable `variable_versatil`, que puede tener diferentes valores a lo largo del mismo programa. Dependiendo del valor que se almacena en la variable, diferentes operaciones se le pueden aplicar a la variable en sí, como suma, búsqueda de un valor en la estructura, concatenación de caracteres, etc.

¿Y a esta información que está en la variable, cómo se accede? ¿Cómo partimos de un nombre como `variable_versatil` y llegamos a la letra `'A'` o al número `23`, o al archivo `example.txt` o a esa entrada en el diccionario? Acá es donde entra en juego todo lo que charlamos hasta ahora. Hay un programa, que en Python se llama intérprete, que relaciona el nombre `variable_versatil` con la dirección de memoria donde esa información está guardada. Entonces cada vez que en tu programa escribas la variable `variable_versatil`, ya sea para imprimirla, o para modificarla, o la operación que vos quieras, el intérprete va a buscar a memoria esa dirección de memoria y obtiene el valor que allí se encuentra. Pensá en el nombre de la variable como un apodo más lindo que decir la dirección de memoria en sí; en vez de pedirle a la computadora que "vaya a lo que hay en la dirección XYZ y pise lo que hay ahí con tal otro valor", que las variables tengan nombre te permite usar esa palabra que vos elegís para no tener que saberte la dirección de memoria; la computadora se encarga de tener una tabla actualizada con las equivalencias entre los nombres de las variables y

las direcciones de memoria.

Esta característica de "versatilidad de la variable" no se aplica a las variables en C. Las variables en C son tipadas, tienen un tipo, que determina qué se puede almacenar en ella. Fuerzan a la variable a sólo contener valores que pertenecen a la misma categoría. Algunas de las categorías más conocidas son `char` para las letras, también llamadas caracteres, `int` para los enteros y `float` y `double` para los números con coma. Esta característica le impone al programador a ser más cuidadoso a la hora de declarar y utilizar variables, pero le permite al programa ser más estructurado y, por lo tanto, más rápidamente compilado. La compilación es el proceso donde se traduce el código que nosotros implementamos en C a assembly, un lenguaje intermedio entre C y binario. Un programa en C, a diferencia de un programa en Python, primero debe ser compilado, para luego ser ejecutado la cantidad de veces que se quiera. Esta rigidez a la hora de declarar y utilizar variables le resuelve varios problemas al compilador, haciendo que su trabajo sea más sencillo y, por lo tanto, permitiendo que lo complete en menos tiempo que si las variables pudiesen ser de cualquier tipo en cualquier momento.

Así, cuando se declara una variable ("se la crea") se le agrega a la izquierda del nombre una palabra especial, reservada, que le indica al compilador qué tipo de información se almacena en esa variable. Por ejemplo, una letra, un número y un archivo se declararían de esta manera en C:

```
#include <stdio.h>

int main() {
    char letra = 'A'; //declaro una variable que almacena la letra 'A'
    int numero = 23; //declaro una variable que almacena el número 23
    FILE *archivo; //se verá en el apunte de punteros

    archivo = fopen("example.txt", "r"); //abro el archivo

    if (archivo == NULL) {
        printf("No se pudo abrir el archivo.\n");
        return -1;
    }

    fclose(archivo); //cierro el archivo

    return 0;
}
```

Por otro lado, la implementación y el uso de diccionarios, o hashes, es tema de la materia, por lo que no mostraré el ejemplo en código.

Pero, ¿por qué quiere saber el compilador el tipo de variable? ¿Para qué le sirve saber si mi variable almacena una letra, un número, o cualquier otra cosa? Al compilador le importa esta información porque el tipo de variable, además de otras cosas, determina el tamaño que se debe reservar en memoria para ese elemento.

Un caracter[6], `char`, requiere un byte de tamaño. Un `int`, tipo de variable que almacena un número entero, requiere de cuatro u ocho bytes, dependiendo la arquitectura de la computadora[7].

¿Te acordás que habíamos mencionado en una sección anterior que podíamos pensar a la memoria como una calle con casas, cada una con una dirección de memoria única, y a los bytes que se almacenan en esas casas como a los inquilinos? Para almacenar un `char` necesitaríamos una casa, mientras que para almacenar un `int` necesitaríamos cuatro u ocho, dependiendo la arquitectura. Este tamaño predeterminado le permite al compilador saber la cantidad de bytes, de memoria, que se debe reservar para cada variable. Y no sólo eso, le permite a la computadora saber cuánta memoria ir a buscar cuando se quiera levantar un dato. Si se quiere leer el `char letra`, con ir a la dirección de memoria especificada y levantar un byte es suficiente. Si se quiere leer el `int numero`, se debe ir a la dirección de memoria del primer byte, que el compilador la conoce porque la relaciona al nombre de la variable, y leer los cuatro (u ocho) bytes siguientes, ya que entre todos ellos componen el número.

Por ejemplo, si pudiésemos ir con una lupa a la RAM a buscar mis variables `letra` y `numero` veríamos algo así

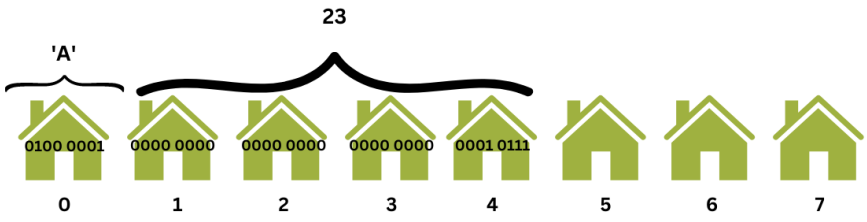

Figure 3: Memoria con variables en binario

Donde en la dirección 0 se guardó el valor de `letra` y en las direcciones 1, 2, 3 y 4 se guardó el valor de `numero`. ¿Por qué la letra `'A'` no aparece como 'A' y el número `23` no aparece como 23? Acordate lo que dijimos al principio: en memoria, hay una tira de 0s y 1s, que representan si hay o no corriente, nada más. No hay letras, ni números en decimal, ni puntos para representar la coma de los números con decimales, ni nada por el estilo. Todo dato que nosotros entendemos (números, textos, letras, símbolos), deben ser traducidos a 0s y 1s. Los números se traducen con equivalencias entre el sistema decimal y

[6]Siendo estrictos, un caracter es una categoría de tipo de dato que compone a las letras, tanto mayúscula como minúscula, pero que también incluye símbolos como los de exclamación y pregunta, puntos y demás. Todos los símbolos se pueden encontrar en la tabla ASCII.

[7]Que dependan de la arquitectura no quiere decir que a veces hacen falta cuatro y a veces hacen falta ocho bytes. Quiere decir que algunas computadoras tienen una arquitectura de 32bits, por lo que, entre otras cosas, todos los ints que se interpreten en esa computadora se asumirá que tienen un tamaño de cuatro bytes. Y si la arquitectura es de 64bits, se asumirá que los ints se almacenan en ocho bytes.

el sistema binario, es matemática, vas a ver cómo hacerlo en futuras materias. Los símbolos, como las letras, se traducen con la tabla ASCII.

Hasta acá, vimos cómo declarar variables en C y cómo hace la computadora detrás de cámara para acceder a la dirección de memoria correcta para ir a buscar el valor que la variable en sí almacena. Pero, ¿dónde viven estas variables? Sabemos que es en memoria, porque ahí es donde el programa vive cuando se está ejecutando, pero ¿en qué parte del programa? ¿Code segment? ¿Data segment?

4.1 ¿Dónde viven las variables de mi programa?

Esta pregunta puede prestar a confusión, así que vamos de a poco. Una variable declarada en un programa puede estar en dos de los cuatro segmentos: en el data segment o en el stack. La línea de código que declara una variable y/o le asigna un valor sólo puede vivir en un lugar: en el code segment. El code segment es donde se almacenan todas las instrucciones, todas las líneas de un programa (lo que vos escribís cuando programás). En el heap no viven variables de por sí, como las conocemos, sino que variables del stack pueden tener direcciones de memoria a espacios de memoria en el heap. Pero de esto hablaremos más en el apunte de punteros. Las variables que vos declarás cuando programás pueden vivir en cualquiera de los otros dos segmentos: data segment y stack.

Y, ¿en qué se diferencia cada segmento? ¿Qué diferencia hace si vive en uno o en otro? Acá es donde nos metemos a hablar del scope, el alcance, de un programa y una función.

En principio, sabemos intuitivamente, luego de usar por años la computadora y el celular, que la ejecución de un programa no influye en la ejecución de otro programa, más allá de alguna posible disminución en la velocidad de ejecución. Si ahora abrís la app de Spotify en tu computadora, eso no va a impactar de ninguna manera en la ejecución del programa que estés utilizando para leer este apunte. Esto nos indica que hay ciertos límites entre un programa y otro, que aunque la RAM es compartida por todos los programas a un nivel físico, eso no quiere decir que cualquier programa puede acceder a cualquier parte de la RAM en cualquier momento.[8].

Podemos establecer como un primer nivel de alcance, entonces, que lo que se declara en un programa no puede ser accedido desde otro programa[9]. Esto implica que si yo declaro una variable en un programa, ningún otro podrá acceder a ella.

Pero éste no es el único nivel de scope que manejamos en los programas. Si lo fuese, cualquier variable, declarada en cualquier lugar, sería accesible desde cualquier lugar. Y no es el caso. Estas variables, que pueden ser accedidas desde cualquier función del programa, existen y se conocen como *variables globales*.

[8] De hecho, es uno de los tantos trabajos del sistema operativo el asegurar que los procesos no accedan a memoria que no es suya, protegiendo así los programas en ejecución y sus datos sensibles.

[9] Más adelante en la carrera y en tus estudios verás que es posible, y muy útil, comunicar dos o múltiples programas distintos, y estudiarás diferentes formas de hacerlo.

Para declararlas, basta con escribir su nombre (y en lenguajes tipados, su tipo) por fuera de cualquier función. Sin embargo, el uso de variables globales es altamente desalentado en general, y prohibido en Algoritmos y Programación II. El porqué se verá en profundidad en próximas materias, pero está relacionado al hecho de que es una variable que puede ser modificada desde cualquier lado, y sin ningún tipo de control; esto implica un alto nivel de descontrol a la hora de consultar el valor de la variable y de depender de ella para calcular o tomar decisiones. ¿Qué pasaría si se tiene una variable global que determina el cambio del dólar y que se utiliza en tu programa de cálculo financiero, pero que sin aviso, otra función lo modifica? Alteraría todos los resultados sin explicación, y no sería un error fácilmente encontrado.

Este tipo de variables, las variables globales, viven en el data segment, al igual que las constantes globales, que son las primas de las variables globales, pero sin el problema que éstas presentan, ya que no pueden ser modificadas, al ser declaradas como constantes. Este tipo de variables sí están permitidos por la cátedra, y pueden ser utilizadas en los trabajos.

Con eso dicho, si no podemos usar variable globales en el trabajo, ¿cómo almacenamos información a lo largo del programa? ¿Hay otro tipo de variables que sí podamos usar? La respuesta es un rotundo sí, y se trata del tipo de variables que venís usando desde que empezaste a programar: las *variables locales*.

Las variables locales son aquellas que se declaran dentro de una función. A nivel práctico, en C significa que las declarás entre las llaves – de la implementación de una función; las variables `letra` y `numero`, aquí declaradas

```
#include <stdio.h>

int main() {
    char letra = 'A'; //declaro una variable que almacena la letra 'A'

    int numero = 23; //declaro una variable que almacena el número 23

    return 0;
}
```

son variables locales. A nivel conceptual, sin embargo, que sean variables locales significa que son variables que solamente existen dentro de esa función. Que no pueden ser accedidas desde otras funciones[10] y que solamente pueden ser manipuladas mientras esta función se esté ejecutando. ¿Por qué? La respuesta corta es porque viven en el stackframe de la función. La respuesta larga está en la siguiente sección.

[10] Por ahora... ver apunte de punteros.

4.2 Stack vs stackframe

Para hablar del stackframe, primero tenemos que hablar del stack. El stack es uno de los cuatro componentes básicos de un programa, y almacena, entre otras cosas, las variables locales de cada función. Pensá en el stack como una pila de platos: sólo podés agregar y quitar elementos desde el tope.

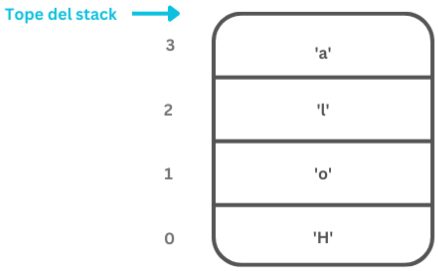

Figure 4: Gráfico ilustrativo del stack. Sólo se puede interactuar con el elemento en el tope del stack.

Este espacio de memoria es controlado por el sistema operativo; él se encarga de liberar la memoria reservada en este espacio, por lo que es muy conveniente para un programador. No tenemos que preocuparnos de liberar esa memoria que reservamos al declarar una variable, el SO lo hace por nosotros.

Pero que el stack se comporte como una pila implica que sólo podemos interactuar con el tope del stack. No es posible manipular elementos en el medio del stack. Si querés leer la letra 'o' del stack del ejemplo en la figura 4, primero deberíamos quitar la letra 'a' y la letra 'l', hasta que el tope del stack apunte a la letra 'o'. Fijate que entonces debería descartar esas dos letras para llegar a la deseada.

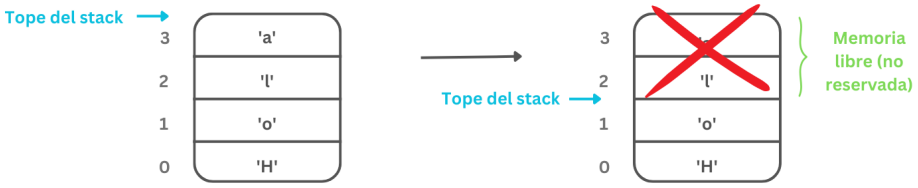

Figure 5: Para llegar a la letra 'o', se deben liberar los espacios de memoria de la 'a' y la 'l'.

Al liberar esos espacios de memoria, éstos ya no le pertenecen al stack, son libres. El heap[11] o cualquier otro programa puede reservarlo y pisarlo con su

[11] Menciono el heap y no los otros dos segmentos porque es el único, además del stack, que

propia información. Al liberar esos espacios de memoria, efectivamente perdimos la información que allí había guardada.

Recapitulando, el stack es uno de los cuatro componentes de un programa, y es uno de los dos, además del heap, que varía su tamaño a lo largo de la ejecución de un programa. Es muy útil para el programador ya que provee almacenamiento que el sistema operativo se encarga de liberar (a diferencia del heap). También se trata de una estructura relativamente rápida de acceso, en comparación con el heap. Pero al ser una pila, implica que sólo podemos interactuar con ella por el tope. Si queremos agregar elementos, se agregan encima del tope; si queremos visitar un elemento intermedio, debemos quitar todos los elementos entre el tope y el elemento deseado, perdiéndolos en el proceso.

Con esto en mente, vayamos ahora sí al concepto de stackframe. El stackframe de una función es el espacio en el stack reservado específicamente para esa función. Todas las variables que se declaren en esa función, así como otros datos importantes para ella, se almacenan en su stackframe. Cuidado, el stackframe no es otro stack, es simplemente una parte del stack que se reservó para esta función en particular.

Si ahora tenemos en cuenta el hecho de que el stack sólo permite aumentarse para arriba, empezamos a entender que los stackframes están uno encima del otro, y que no es posible pasar de uno a otro a diestra y siniestra. Si declaro una variable en `main` y luego en otra función auxiliar, que es llamada desde `main`, para acceder a la variable de `main`, debo eliminar del stack la variable de la función auxiliar, perdiéndola en el proceso.

Visto en un ejemplo, si yo tengo este código

```
#include <stdio.h>

void aux() {
    char letra2 = 'Z';
}

int main() {
    char letra = 'A';
    aux();

    return 0;
}
```

el stack evolucionará de la siguiente manera. Primero se genera el stackframe de la función `main`, y luego se reserva espacio para la variable `letra`. Más tarde se llama a la función `aux`, por lo que se genera un nuevo stackframe, encima del de la función `main`, con su data específica. Ahora se reserva espacio para la variable `letra2`. En este punto, no es posible acceder a la variable `letra` sin

varía su tamaño a lo largo de la ejecución. Tanto el code como el data segment tienen tamaño fijo una vez compilado el programa.

perder la información almacenada en la variable `letra2` y toda la data especial de la función `aux`. Cuando se termina de ejecutar la función `aux`, se libera la memoria reservada para sus variables locales y para la data específica de ella. Y finalmente, se termina de ejecutar la función `main` y se libera el stackframe reservado para esta función.

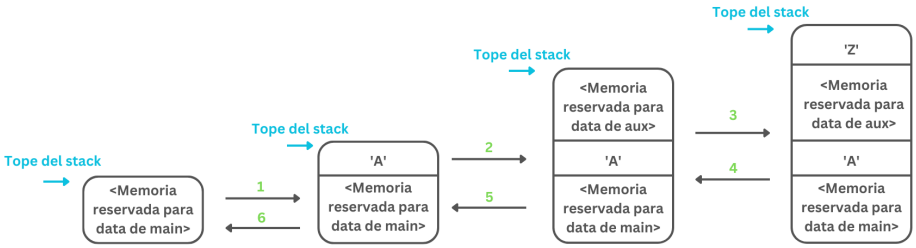

Figure 6: Movimientos del stack

Es cierto que ambas variables, `letra` y `letra2`, viven en el stack. Ambas son variables locales. Pero son variables locales de diferentes funciones, por lo que viven en diferentes stackframes. No es posible acceder a `letra` desde la función `aux`, porque debería liberar toda la memoria reservada para `aux` para llegar a esa variable. Que la variable sea local quiere decir que es específica de ese stackframe, que vive en él y sólo existe mientras la función se esté ejecutando, porque cuando la función deja de ejecutarse, el espacio del stack reservado para esa función, su stackframe, es liberado. Y no es posible recuperarlo.

Esto parece ser muy restrictivo a primera vista, y la realidad es que lo es. Pero esta organización del stack no es azarosa: fue diseñada de esta manera debido a la naturaleza de las llamadas en un programa. Una función A que llama a B, que a su vez llama a C, implica que ahora estamos parados en C, pero que cuando esta función termina, queremos volver a B, no a A. Pensá en una muñeca *mamushka*, o en una función recursiva. Recién volveremos a A cuando B termine. Esta estructura implica una organización de pila, que es justamente lo que nos provee el stack. Teniendo en cuenta esta cualidad para el diseño, obtenemos mayor velocidad de ejecución y aprovechamos lo más posible el espacio de RAM. También es cierto que una pila es una de las estructuras de datos más sencillas de implementar, como ya lo verás en esta materia.

5 Conclusiones

Redondeando los conceptos, es de suma importancia entender cómo se organiza la memoria y dónde se guarda cada parte de un programa. Entender conceptualmente qué es el stack, cómo se declaran las variables, dónde viven las variables, y por cuánto tiempo, te ayudarán a programar código eficaz, y con un poco de práctica a aprovechar la distribución de memoria para que tus programas sean más performantes.

Este apunte es importante por sí sólo, y los conceptos en él descriptos serán revisados en futuras materias a lo largo de tu carrera. Pero también son la base para entender otro de los grandes temas de la programación, y en particular de esta materia: punteros.